Etude et évaluation de réseaux ATM pour l'interconnexion dans des systèmes multiprocesseurs

Olivier Jean-Pie Ondoa

▶ **To cite this version:**

Olivier Jean-Pie Ondoa. Etude et évaluation de réseaux ATM pour l'interconnexion dans des systèmes multiprocesseurs. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1997. Français. NNT : . tel-00004957

HAL Id: tel-00004957

https://theses.hal.science/tel-00004957

Submitted on 20 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Olivier Jean-Pie ONDOA

pour obtenir le grade de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 mars 1992)

(Spécialité: **Microélectonique**)

====================

Etude et évaluation de réseaux ATM pour l'interconnexion dans des systèmes multiprocesseurs

====================

Date de soutenance: September 1997

Composition du jury:

Paul JACQUET :	Président
Christian FRABOUL :	Rapporteur
Vincent OLIVE :	Rapporteur
Chantal ROBACH :	Examinateur
Guy MAZARE :	Examinateur

Thèse préparée au sein du Laboratoire:

Logiciels Systèmes Réseaux (LSR)

Ce document est organisé en deux parties:

- La première partie, en français, donne un aperçu général des travaux accomplis et résultats obtenus dans le cadre de ma thèse.
- La deuxième partie, en anglais, donne une description détaillée de ces travaux et résultats.

The present document is organized in two majeur parts:

- The first part, in french, gives a general overview of the accomplished works and results obtained during my Ph.D.

- The second part, in english, gives detailed description of these works and results.

Résumé

L'ATM (Asynchronous Transfer Mode) est un nouveau protocole de communication actuellement proposé comme standard pour les Réseaux Numériques à Intégration de Services (RNIS) large bande.
Il est basé sur la commutation et le relais de petits paquets de taille fixe, 53 octets, appelés *cellules*. Une cellule ATM est constituée d'un entête de 5 octets contenant les informations qui permettent d'identifier le chemin de la cellule à travers le réseau, et une charge utile de 48 octets qui contient les données des utilisateurs.
L'ATM a été à l'origine développé comme standard international pour les réseaux métropolitains (Wide Area Network: WAN) basés sur la fibre optique (Synchronous Optical NETwork: SONET).
Il s'est ensuite révelé interessant dans les réseaux locaux (Local Area Network: LAN).
Au sein de notre laboratoire, le Groupe de Recherche sur les Architectures Matérielles (GRAM) étudie les réseaux d'interconnexion basés sur l'ATM, car il pense que de tels réseaux peuvent satisfaire les besoins en communication sans cesse croissants dans les machines parallèles.
Dans notre approche, nous voulons rapidement tester et comparer différentes architectures de réseau sans le délai et coût de la réalisation d'un prototype materiel.
Les outils traditionnels de conception de circuits VLSI (Very Large Scale Integrated), tels que les outils VHDL (Very high speed integrated circuit Hardware Design Language), Verilog, Compass, Cadence, etc..., permettent de réaliser cet objectif.
Cependant, ces outils n'autorisent l'utilisation que de charges qui sont définies de manière statique avant le début de la simulation, à partir de modèles analytiques ou de traces d'exécution d'une application réelle sur une machine parallèle.
Or, de telles charges ne sont pas appropriées pour des tests et évaluations réalistes de nos architectures.
En effet, il y a une étroite corrélation entre les performances et la charge d'un réseau d'interconnexion. Cette corrélation est dynamique et très difficile à modèliser.

Sur ce, il a fallu concevoir un nouveau type de plateforme de simulation qui non seulement ne nécessite pas la réalisation de prototypes matériels, mais aussi permette de simuler nos implémentations de réseaux d'interconnexion directement sous des charges générées au cours de l'exécution d'applications réelles.
Le GRAM s'intéresse à deux nouveaux interfaces de programmation d'applications (Application Programming Interface: API): PVM (Parallel Virtual Machine) développé au laboratoire national d'Oak Ridge (Oak Ridge National Laboratory: ORNL), et MPI (Message Passing Interface) développé à l'université du Tennessee Knoxville.
Ils fournissent tous les deux des librairies C et Fortran permettant d'écrire des programmes parallèles de type plus général MIMD (Multiple Instruction Multiple Data) ou restreint SPMD (Single Program Multiple Data), et d'utiliser une collection d'ordinateurs scalaires, vectoriels, multiprocesseurs, etc..., interconnectés par un ou plusieurs réseaux comme une seule et unique ressource de traitement.
Bien que MPI offre un jeu de primitives de communication plus varié, et qu'il possède la notion de type de donnée dérivé, que n'a pas PVM et qui permet de transférer directement des données non contigües en mémoire, par exemple les éléments d'une colonne d'une matrice, réduisant ainsi le nombre de recopies mémoire à mémoire au cours des opération de communications, j'ai implémenté une plateforme PVM, tout simplement parce que MPI n'était pas officiellement sortie lorsque j'ai commencé ces travaux fin 1993.
Sur cette plateforme, les processus des applications se connectent à un processus qui émule le réseau d'interconnexion sous test, et communiquent à travers lui, générant ainsi directement sa charge.
La communication entre les processus des applications et le processus réseau se fait aux moyens de segments de mémoire partagés des IPCs (Inter-Process Communication) d'Unix. Notons que l'utilisation de ces IPCs implique que les applications et les implementations réseaux doivent être décrites en langage de programmation C.
Actuellement, la platforme PVM est opérationnelle. J'ai implémenté au niveau du transfert des registres (Register Transfer Level: RTL), un premier réseau d'interconnexion basé sur l'ATM, donnant ainsi naissance au prototype réseau qui est présenté dans ce mémoire. C'est un réseau d'interconnexion 3-étage constitué de commutateurs BMX (Bus Matrix Switch) sur sa couche ATM, et d'une couche d'adaptation à l'ATM (ATM Adaptation Layer:

AAL) qui est version allégée de l'AAL type 3/4 définie par le CCITT.
Le fonctionnement de ce prototype réseau a été testé, sur la plateforme, au cours de l'exécution de la résolution parallèle de systèmes d'équations linéaires, et de la muliplication parallèle de matrices.
Les performances de ce prototype réseau ont été évaluées, sur la plateforme, au cours de l'exécution d'une application où un processus source envoie des messages de tailles différentes à un processus de destination à travers le prototype réseau, les deux processus mesurant à chaque fois le délai du transfert.
Il apparaît que les mesures expérimentales consolident les mesures théoriques obtenues suite à une analyse détaillée de l'architecture et du fonctionnement de prototype réseau, et ceux effectuées sur la plateforme materielle PVM/-ATM(AAL3/4) de l'université du Minnesota Minneapolis.
En bref, le transfert d'un message de 4 octets dure en théorie 193 cycles d'horloge, et le délai de transfert croît de manière linéaire en fonction de la taille des messages. De plus, la bande passante du prototype réseau utilisée au cours du transfert des messages croît sensiblement pour de petites tailles, puis sature pour des tailles plus grandes à un maximum de 0.06 octets/sec, soit 8.9 Mbits/sec pour des liens de transmission à 155.52 Mbits/sec, ou 35.5 Mbits/sec pour des liens de transmission à 622.08 Mbits/sec.
La plateforme PVM a enfin permis de mesurer l'impact des conflits au sein du prototype réseau sur la résolution parallèle d'un système d'équations linéaires programmé pour l'occasion. Il s'est avéré que ces conflits ont rallongé cette résolution d'un délai supplémentaire de 36% du temps de résolution total.
Je n'ai pas encore conduit une véritable étude comparative sur la plateforme PVM, car je n'ai réalisé qu'un seul prototype réseau jusqu'à présent.
Néanmoins, les résultats des tests de fonctionnement, de l'évaluation des performances, et de la mesure de l'impact des conflits au sein du prototype réseau indiquent que la plateforme PVM est outil très prometteur pour notre étude des réseaux d'interconnexion basés sur l'ATM.

Abstract

The *Asynchronous Transfer Mode (ATM)* is an emerging communication protocol which is actually proposed as standard for *Broadband-Integrated Services Digital Network (B-ISDN)*.
It is based on switching and/or relaying fixed-length packets called *cells*. An ATM cell consists of a 5-byte header which identifies the cell's route through the network, and a 48-byte payload which contains the message data.
ATM was originally developed as an international standard for *Synchronous Optical Network (SONET)*-based *Wide Area Networks (WANs)*.
Its attractiveness for *Local Area Networks (LANs)* has resulted in ATM LANs.
In our research team, we investigate the interconnection networks based on ATM while studying *Parallel and Distributed Machines*, because we think they can satisfy the unceasingly growth of the communication needs within these machines.
In our approach, we want to quickly test for correctness and compare different architecture alternatives without the delay and expense of an hardware prototype.
Conventional Very Large Scale Integrated (VLSI) circuits design tools, such as Very high speed integrated circuit Hardware Design Language (VHDL) tools, VERILOG, COMPASS, CADENCE, etc., enable to meet this objective.
However, these tools allow the use of only workloads that are statically defined, that is at the beginning of the simulation, from either analytical models or communication traces from real application executions over parallel machines.
Now, such workloads are not appropriate for the realistic tests and evaluations of our different interconnection network implementations.
Indeed, there is a close correlation between an interconnection network and its workload. The behavior of applications has an impact over the performances of interconnection networks. And conversely, these performances have an impact over the developement of the communications of applications. This correlation is dynamic, hence extremely difficult to characterize in an analytical model of worload generation. In addition, it turns the use of

the communication traces from the execution of an application over a parallel machine, with a given interconnection network, to synthesize the workload of a different interconnection network ineffective.
It then became of a prime necessity to design a platform, which not only obviates the need to achieve hardware prototypes, but also to simulates our interconnection network implementations directly under the workloads which are dynamically generated while running real applications on it.
In our research group, we are interested in two emerging Application Programming Interfaces (API): the Parallel Virtual Machine (PVM) developed at the Oak Ridge National Laboratory (ORNL) under the auspices of the Faculty Research Program of Oak Ridge Associated Univerties, and the Message Passing Interface (MPI) developed at the University of Tennessee Knoxville. Both, they provide C and FORTRAN libraries allowing to write fully general Multiple Instruction Multiple Data (MIMD) and more restricted Single Program Multiple Data (SPMD) apllications, and allow the use of a collection of scalar, vectorial, parallel (also multiprocessor), or even special-purpose computers interconnected by one or more networks as a single computational resource.
Although MPI is right now viewed as being more mainstream than PVM, I implemented a PVM platform, simply because MPI was not released when I started my Ph.D. investigations, that is, late 1993.
Over this platform, the processes from PVM applications plug themselves in a process, which emulates the interconnection network implementation under test, and communicate each with another via it, hence generating directly its workload.
Since all processes run on the same machine, the communication between the network and application processes happen via the Unix shared memory Inter-Process Communication (IPC) facilities. This implies, that the PVM applications and interconnection network implementations must be written in the C programming language.
Actually, the PVM platform is operational. I implemented at the Register Transfer Level (RTL) a first interconnection network based on ATM, hence yielding the network prototype whose architecture is presented in this dissertation. It is a 3-stage network, with *Bus Matrix Swithes (BMX)* at the ATM layer and an *ATM Adaptation Layer (AAL)* which is a lightened version of the type 3/4.
The operation of this prototype was tested while running linear equation sys-

tem parallel resolutions and matrix parallel multiplications on the platform. Its performances were evaluated on the platform, while running an application in which a source process sends messages of variable sizes to a destination process, with both processes measuring the transfer delay for each message. It shown up that the experimental measurement results are consistent with the treoretical measurement results, which were directly derived from the architecture and operation analysis of the network prototype, and the results of the measurements performed over a PVM/ATM(AAL3/4) hardware platform at the University of Minnesota Minneapolis.
In summary, it theoretically takes 193 clock cycles to transfer a 4-byte message via the network C prototype, and the transfer delay of this network linearly increases as the message size. In addition, the throughput achieved by this network sharply increases for small message sizes, whereas for higher message sizes it quickly saturates to achieve an approximative maximum value of 0.06 bytes/cycle, that is 8.9 Mbits/sec for a network speed of 155.52 Mbits/sec or 35.5 Mbits/sec for a network speed of 622.08 Mbits/sec.
The platform enabled to measure the impact of the contentions in the bosom of the network prototype over the parallel resolution of a system of 4 linear equations each with 4 variables. It shown up, that the contentions lengthened the resolution of an extra delay, which is nearly 36% of the overall resolution time.
I did not yet lead, over the platform, comparative studies since I have implemented only one interconnection network so far.
Nevertheless, the results of the network prorotype operation tests, performances evaluation, and impact measurement indicate that the PVM platform is a promising tool for our investigations in ATM-based interconnection networks.

Contents

I Présentation Générale des Travaux **1**

1 Introduction **3**

2 Présentation Générale de l'ATM **5**

2.1 Structure des Réseaux ATM 6
2.2 Le Système de Transmission 6
2.3 La Couche Physique 7
2.4 La Couche ATM 8
2.4.1 Les Commutateurs 8
2.4.2 La Topologie d'Interconnexion 9
2.5 La Couche AAL 10

3 Le Prototype Réseau **13**

3.1 La Couche Physique 14
3.2 La Couche ATM 14
3.2.1 Les Commutateurs 14
3.2.2 La Topologie d'Interconnexion 18
3.3 La Couche AAL 20
3.3.1 Présentation Générale 20
3.3.2 Les Cartes AAL 22
3.4 Vision Générale du Prototype 25
3.5 Le Service de Données en mode Non Connecté au dessus du Prototype 26

4 La Plateforme PVM **27**

5 Tests et Expériences **31**
5.1 Le Test Structurel du Prototype de Réseau 31
5.2 Le Test de la Plateforme PVM 32
5.2.1 La Résolution d'Equations Linéaires 32
5.2.2 La Multiplication de Matrices 34
5.3 L'Evaluation des Performances du Prototype Réseau 35
5.4 L'Impact des Conflits dans le Prototype Réseau 37

6 Conclusion **41**

II The Detailled Dissertation **43**

7 Introduction **45**

8 ATM Overview **51**
8.1 ATM Network Architectures 52
8.2 The Underlying Transmission System 53
8.2.1 Framed/SDH Transmission System 54
8.2.2 Framed/PDH Transmission System 55
8.2.3 Cell-based Transmission System 56
8.2.4 Asynchronous Transmission System 56
8.3 The Physical Layer (PL) 57
8.3.1 The Physical Media (PM) Sublayer 58
8.3.2 The Transmission Convergence (TC) Sublayer 58
8.4 The ATM layer (ATM) 59
8.4.1 The Header Translation 62
8.4.2 The Routing 62
8.4.3 The Queueing 62
8.4.4 The Interconnection Topology 65
8.5 The ATM Adaptation Layer (AAL) 68
8.5.1 The AAL Services 68
8.5.2 The AAL Data Units 70
8.5.3 The AAL Functions 72
8.5.4 Connectionless Data Services 74
8.6 Traffic Control 76

8.6.1 Source Policing . . . 76
8.6.2 CBR Connections . . . 78
8.6.3 VBR Connections . . . 79
8.6.4 UBR Connections . . . 81
8.6.5 ABR Connections . . . 82
8.6.6 Source Shaping . . . 84
8.6.7 Delay and Loss Priorities Management . . . 84
8.7 InterNetworking . . . 85
8.8 ATM Overview Conclusion . . . 87

9 The Network Prototype 89
9.1 The PL Layer . . . 90
9.2 The ATM Layer . . . 90
9.2.1 ATM Switches . . . 90
9.2.2 The Interconnection Topology . . . 99
9.3 The AAL Layer . . . 102
9.3.1 General Description . . . 102
9.3.2 The AAL Units . . . 108
9.4 The Network Prototype . . . 125
9.5 The Connectionless Data Service . . . 126

10 The PVM Platform 127
10.1 PVM . . . 128
10.1.1 The User Library . . . 128
10.1.2 The Daemon Side . . . 130
10.2 MPI . . . 130
10.2.1 The MPI Send Communication Primitives . . . 131
10.2.2 The MPI Receive Communication Primitives . . . 134
10.2.3 The MPI Broadcast Communication Primitive . . . 135
10.3 The Platform Itself . . . 136

11 Tests and Experiments 141
11.1 Network Prototype Operation Tests . . . 141
11.1.1 PPD Unit Simulation Tests . . . 146
11.1.2 SPD Unit Simulation Tests . . . 149
11.1.3 SPD-PPD Link Simulation Test . . . 151
11.1.4 BMX Switch Simulation Tests . . . 153

11.1.5 AAL Unit Simulation Tests 155
11.2 PVM Platform Operation Tests 159
11.2.1 Linear Equation System Distributed Resolution 160
11.2.2 Matrix Distributed Multiplication 164
11.3 Prototype Performance Measurements 166
11.3.1 Theoretical Performances of the Platform 168
11.3.2 Experimental Performances of the Prototype 170
11.4 Contentions Impact over The execution of an Application . . . 176
11.5 Tests and Measurements Conclusion 179

12 Conclusion and Perspectives 181

III Appendixes 185

A PPD Units' C Code and Data Structures 187
A.1 PPD Units' C Data Structures 187
A.2 PPD Units' C Code 187

B SPD Units' C Code and Data Structures 191
B.1 SPD Units' C Data Structures 191
B.2 SPD Units' C Code 191

C Network Prototype's Interconnection Topology, and Predefined Virtual Connections 193
C.1 PPD and SPD units Interconnection Algorithm 193
C.2 Establishment Algorithm of Predefined Virtual Connections . 194

D CRC Code Generation and Verification 197
D.1 CRC Code Generation Algorithm 197
D.2 CRC Code Verification Algorithm 198

E AAL Units' C Data Structures, and Finite State Machine Graphs 199
E.1 AAL Units' C Data Structures 199
E.2 AAL Units' Finite State Machine 0 201
E.3 AAL Units' Finite State Machine 1 202
E.4 AAL Units' Finite State Machine 2 203

E.5 AAL Units' Finite State Machine 3 204
E.6 AAL Units' Finite State Machine 4 205
E.7 AAL Units' Finite State Machine 5 206
E.8 AAL Units' Finite State Machine 6 207
E.9 AAL Units' Finite State Machine 7 208
E.10 AAL Units' Finite State Machine 8 209
E.11 AAL Units' Finite State Machine 9 210
E.12 AAL Units' Finite State Machine 10 211
E.13 AAL Units' C Code Skeleton 212

F PPD Unit Simulation Tests **213**
F.1 Header Translation and Successful
Routing of ATM Cells:
Stimuli File 213
F.2 Header Translation and Successful
Routing of ATM Cells:
Traces File 214
F.3 Header Translation and Routing
Suspension of ATM Cells:
Stimuli File 219
F.4 Header Translation and Routing
Suspension of ATM Cells:
Traces File 220
F.5 Chain of the Header Translation and
Routing Suspension, Header Translation and
Successful Routing of ATM Cells:
Stimuli File 223
F.6 Chain of the Header Translation and
Routing Suspension, Header Translation and
Successful Routing of ATM Cells:
Traces File 223

G SPD Unit Simulation Tests **229**
G.1 Successful Transmission of ATM Cells:
Stimuli File 229
G.2 Successful Transmission of ATM Cells:
Traces File 229

G.3 ATM Cell Transmissions Suspension: Stimuli File . . . 234
G.4 ATM Cell Transmissions Suspension: Traces File . . . 235
G.5 Transmission Suspension and Successful Transmission Chain: Stimuli File . . . 239
G.6 Transmission Suspension and Successful Transmission Chain: Traces File . . . 239

H SPD PPD Unit Link Simulation Tests 245
H.1 SPD PPD Unit Link Simulation Test: Stimuli File . . . 245
H.2 SPD PPD Unit Link Simulation Test: Traces File . . . 246

I BMX Switch Simulation Test 255
I.1 BMX Switch Simulation Test: Stimuli File . . . 255
I.2 BMX Switch Simulation Test: Traces File . . . 256

J AAL Units Simulation Test: Traces File 259

K PVM Platform Operation Tests 273
K.1 Linear Equation System Distributed Resolution: Startup Program C Code . . . 273
K.2 Linear Equation System Distributed Resolution: Resolution Program C Code . . . 275
K.3 Linear Equation System Distributed Resolution: I/O File Examples . . . 280
K.3.1 Traces File from the Experimental Resolution of the Example 4 . . . 281
K.3.2 Traces File from the Theoretical Resolution of the Example 4 . . . 283

K.4 Matrix Distributed Multiplication:
Startup Program C Code 285
K.5 Matrix Distributed Multiplication:
Multiplication Program C Code 287
K.6 Matrix Distributed Multiplication
I/O File Examples 289

IV Bibliography 291

List of Figures

2.1 Structure d'une Cellule ATM 5
2.2 Modele en Couche des Réseaux ATM 7
2.3 Quelques Topologies de Réseau 9

3.1 Les Commutateurs BMX du Prototype Réseau 15
3.2 Architecture 8-bit des modules PPD 15
3.3 Structure des Entrées des Tables de Routage 16
3.4 Architecture 8-bit des modules SPD 17
3.5 Architecture du Processeur interne des modules SPD 17
3.6 Topologie d'Interconnexion du Prototype Réseau 19
3.7 Les 4 Chemins possibles pour Une Communication Point-à-Point 19
3.8 Les 4 Chemins possibles pour Une Diffusion 20
3.9 Format des Entêtes PVM 21
3.10 Format des Entêtes de la sous couche SAR 21
3.11 Plan de Masse des Cartes du Prototype 23
3.12 Vue Générale Prototype Réseau 25

5.1 Algorithme de Gauss Jordan 33
5.2 Algorithme Séquentiel de Multiplication de Matrices 34
5.3 délai de transfert VS taille des messages 36
5.4 bande passante utilisée VS taille des messages 38

8.1 ATM cell Structure 51
8.2 Protocol Reference Model 53
8.3 Layers Model of ATM Networks 54
8.4 SDH STM-1 Frame 54
8.5 Virtual Container of a Degree 4 (VC-4) 55
8.6 PLCP mapping into PDH Frames 56

8.7 Asynchronous Transmission Link 57
8.8 Flow of information between TC and PM 58
8.9 Few Digital Transmission systems used by the PL 58
8.10 ATM Cell Header Formats 60
8.11 ATM Switch basic Operation 61
8.12 Switch with Output Queues 63
8.13 Switch with Input Queues 63
8.14 Switch with a Central Queue 64
8.15 Switch with Multiple Queues 65
8.16 Network Topologies 66
8.17 3-Stage Networks 67
8.18 Spatial Topologies 68
8.19 AAL Service Classes 69
8.20 Data Units: from Applications to ATM Cells via the AAL . . 70
8.21 AAL-1 SAR-PDU Format 71
8.22 AAL-2 SAR-PDU Format 71
8.23 AAL-3/4 SAR-PDU Format 71
8.24 AAL-3/4 CPCS-PDU Format 72
8.25 AAL-5 CS-PDU and SAR-PDU Format 72
8.26 CDV and CDVT 76
8.27 Equivalent Forms of GCRA 77
8.28 GCRA Tests for T=10, τ=2 79
8.29 Spacing within Bursts 80
8.30 Maximum Size of Bursts with only Spacing 81
8.31 Maximum Size of Bursts with Spacing and Interburst 82
8.32 Minimum, Allowed, and Peak Cell Rate 82
8.33 Resource Management Cell Format 83
8.34 Source Shaping and Policing 84
8.35 Delay and Loss Priorities VS Buffers 85
8.36 ATM as LAN Interconnection 85
8.37 ATM to Network End Stations and Servers 86
8.38 IP over ATM 86
8.39 LAN Emulation 86

9.1 BMX Switches of the Prototype 91
9.2 PPD Units Architecture 92
9.3 Translation Table Entries Structure 93

9.4 SPD Units Behavior 95
9.5 SPD Unit Architecture 96
9.6 SPD Processing Unit Architecture 97
9.7 Prototype Interconnection Network Topology 99
9.8 Point-To-Point Communication Paths from Src 5 to Dst 14 . . 100
9.9 Broadcast Communication Paths from Src 5 101
9.10 PVM or Prototype AAL CS Headers 103
9.11 Prototype AAL SAR Headers 104
9.12 Calculation of a CRC Code 105
9.13 CRC Codes Calculation Circuits 106
9.14 Prototype AAL Units Architecture Floor Plan 109
9.15 Base+Offset Fill up Mechanism 115
9.16 Message Descriptor 122
9.17 The Network Prototype Overview 125

10.1 PVM Advise and Normal Communication Modes 129

11.1 Stimuli and Traces Files Formats 142
11.2 Input and Output Stimuli Chained Lists Structures 143
11.3 PPD Unit Test Platform 146
11.4 SPD Unit Test Platform 149
11.5 SPD-PPD Link Test Platform 152
11.6 BMX Switch Test Platform 154
11.7 AAL Units Test Platform 155
11.8 An AAL Unit Simulation Test's Screen Display History 157
11.9 Linear Equation System Resolution Gauss Jordan Sequential Pseudocode 160
11.10 Linear Equation System Distributed Resolution Input/Output File Formats 162
11.11 Linear Equation System Distributed Resolution Communication Schemes Example 163
11.12 Matrix Multiplication Sequential Pseudocode 164
11.13 Matrix Distributed Multiplication Input/Output File Formats 165
11.14 Matrix Distributed Multiplication Steps and Communication Schemes 167
11.15 Echo Program Overview 172
11.16 Direct mode: One-way Delay versus message size 173

11.17 Direct mode: Achievable Throughput versus message size . . . 175
11.18 Network Prototype Contentions Impact over an Application Execution . . . 177

List of Tables

2.1 AAL Functions 11

4.1 Les Primitives de Communication C Ré-écrites 29

8.1 Payload Type Field Encoding 60
8.2 AAL Data Unit Header and Trailer Sizes 71
8.3 AAL Functions 73
8.4 ATM Connection Types 78

9.1 8-bit Bus Bandwidth vs Technology 90
9.2 Names and Identifier Numbers of Commands to AAL Units . . 111

10.1 Basic MPI Datatypes 133
10.2 C Communication Primitives Synopsis 137

11.1 One-Way Delay 173
11.2 Direct Route 174
11.3 r_{max} values in Mbits/sec 176

Part I

Présentation Générale des Travaux

Chapter 1

Introduction

Au sein du Groupe de Recherche sur les Architectures Matérielles (GRAM) du laboratoire Logiciels Systèmes Réseaux (LSR), nous étudions les réseaux d'interconnexion basés sur l'ATM (Asynchronous Transfer Mode). Nous pensons que de tels réseaux sont capables de satisfaire aux besoins en communication sans cesse croissants dans les machines parallèles.
Bien que les outils traditionnels d'aide à la conception des circuits VLSI (Very Large Scale Integrated) tels que: les outils VHDL (Very high speed integrated circuit Hardware Design Language), Verilog, Compass, Cadence, etc..., permettent de tester et comparer rapidement différentx choix d'architectures sans le délai et coût de la réalisation d'un prototype matériel, ils n'autorisent cependant que l'utilisation de charges statiquement définies, et donc inappropriées pour des tests et évaluations réalistes.
Pour remédier à ce problème, j'ai développé une plateforme PVM qui non seulement ne nécessite pas la réalisation de prototypes matériels, mais permet surtout la simulation des réseaux sous tests directement sous des charges générées lors d'exécutions d'applications PVM réelles, écrites dans un style plus général MIMD (Multiple Insrtuction Multiple Data) ou restreint SPMD (Single Program Multiple Data).
Un premier réseau d'interconnexion ATM a été entièrement conçu et décrit au niveau du transfert des registres (Register Transfer Level: RTL) en langage de programmation C, donnant ainsi naissance au prototype réseau C qui est présenté dans ce mémoire.
Son fonctionnement a été testé sur la plateforme PVM au cours de l'exécution de la résolution de systèmes d'équations linéaires et multiplication de matri-

ces.
Ses performances ont été évaluées sur la plateforme PVM lors de l'exécution d'une apllication echo entre un processus source et destination. Les résultats obtenus sont ceux attendus. En effet, son délai de transfert croît de façon linéaire avec la taille des messages, et sa bande passante utile croît rapidement pour des petites tailles de messages et sature pour des tailles plus grandes.
La plateforme PVM a en outre permis de mesurer l'impact des conflits au sein du prototype réseau sur l'exécution d'une application.
La suite du résumé en français de ce mémoire est organisée comme suite. Le chapitre 2 résume l'aperçu général de l'ATM qui est donnée au chapitre 8. Le chapitre 3 résume la description de l'architecture du prototype réseau qui est faite au chapitre 9. Le chapitre 4 résume la présentation de la plateforme PVM qui est donnée au chapitre 10. Le chapitre 5 résume la description des tests de fonctionnement et évaluations des performances, ainsi que de la mesure de l'impact des conflits au sein, du prototype réseau qui est faite au chapitre 11. Le chapitre 6 conclu le résumé en français de ce mémoire.

Chapter 2

Présentation Générale de l'ATM

L'ATM (Asynchronous Transfer Mode) est un nouveau protocole de communication basé sur la commutation de circuits virtuels sur lesquels, les informations sont transportées dans de petits paquets de taille fixe, 53 octets, appelés cellules.
Comme l'indique la Figure 2.1, chaque cellule ATM est constituée d'un entête de 5 octets identifiant un chemin à travers le réseau, et une charge utile de 48 octets contenant les informations utilisatrices.

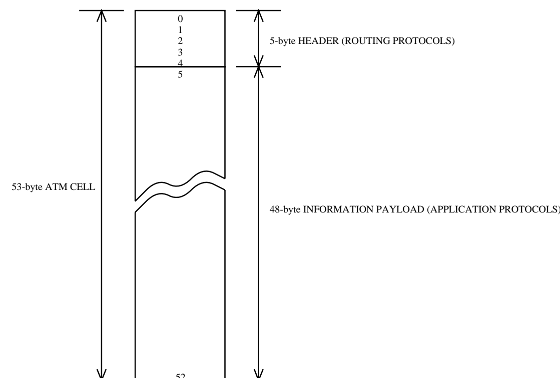


Figure 2.1: Structure d'une Cellule ATM

Ce qui rend l'ATM intéressant, c'est son aptitude non seulement à trans-

porter de façon intégrée sur le même support de transmission des flux aussi divers et contrastés que l'audio, la vidéo, et les données, mais aussi à offrir des débits de transmission variant de quelques Mbits/sec à quelques Gbits/sec.

2.1 Structure des Réseaux ATM

Les réseaux ATM sont structurés en couches et en plans. Le modèle de réference du protocole ATM identifie trois plans:

- un Plan de Gestion pour la gestion du réseau,
- un Plan de Contrôle pour l'ouverture et la fermeture des connexions,
- un Plan Usager pour le transfert des données,

et trois couches:

- une Couche Physique pour le transport des cellules ATM entre deux commutateurs de la couche ATM,
- une Couche ATM pour la commutation, le multiplexage, et le démultiplexage des cellules ATM à travers le réseau,
- une Couche d'Adaptation à l'ATM pour l'adaptation des flux générés par les applications au format ATM.

La Figure 2.2 montre la pile des protocoles et couches du modèle de communication basé sur l'ATM.
La façon dont sont implémentées ces couches différencie un réseau ATM donné d'un autre.

2.2 Le Système de Transmission

Les liens de transmission sont structurés en trames récurrentes se répétant avec une période T dont la valeur est de 125μs.
Selon la taille et la structure des trames, on distingue les différents systèmes de transmission suivants:

- tramé/SDH

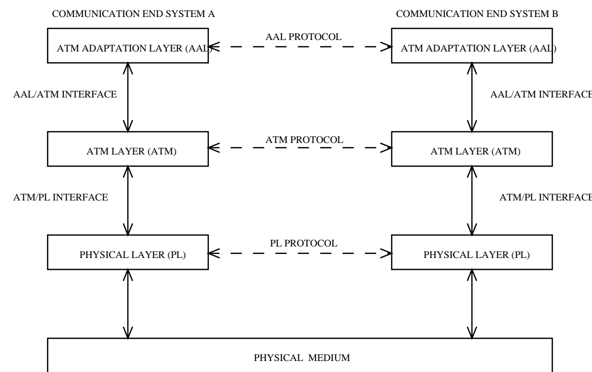


Figure 2.2: Modele en Couche des Réseaux ATM

- tramé/PDH
- basé sur les cellules
- asynchrone

2.3 La Couche Physique

La couche physique est en quelque sorte une couche d'adaptation du flux de cellules généré par la couche ATM au système de transmission de support. Afin de fournir à la couche ATM un service unique, non dépendant du système de transmission sous-jacent, la couche physique a été scindée en:

- une sous-couche Média Physique (Physical Media: PM) dépendante du système de transmission de support, et effectuant essentièlement la transmission à proprement parlé des cellules,
- une sous-couche Convergence de Transmission (Transmission Convergence: TC) effectuant essentièlement le contrôle d'erreurs sur les entêtes des cellules, la délimitation des cellules, et la projection des cellules dans la charge utile des trames des liens de transmission.

2.4 La Couche ATM

Le rôle principal de la couche ATM est d'aiguiller les cellules à travers le réseau.

2.4.1 Les Commutateurs

Les commutateurs sont les opérateurs fondementaux de la couche ATM. Ils effectuent essentiellement:

- la translation des entêtes des cellules
- le routage des cellules
- le stockage des cellules dans les files d'attente
- le contrôle de saturation des files d'attente

La localisation des files d'attente a une incidence sur la vitesse de fonctionnement d'un commutateur. En effet, lorsqu'elles sont en entrée, le commutateur peut fonctionner à la même vitesse que les liens de transmission. Par contre, lorsqu'elles sont en sortie, le commutateur doit fonctionner au moins n fois plus vite que les liens de transmission, n étant le nombre de ses entrées.
Les files d'attente en sortie saturent lorsque le taux d'utilisation des liens d'entrée approche 1. Tandis que, celles en entrée saturent alors que ce taux est inférieur à 0.586. Cependant, les files d'attente en sortie nécessitent de plus grandes tailles.
Une façon de réduire la capacité de stockage totale requise dans un commutateur avec des files d'attente en sortie tout en conservant un niveau de performance équivalent, est de prévoir une file d'attente centrale. Seulement, la gestion de cette dernière est plutôt complexe.
L'approche files d'attente multiple en sortie conduit aux mêmes performances que l'approche files d'attente en sortie classique, mais avec un commutateur pouvant fonctionner à la même vitesse que les liens de transmission. Les commutateurs de type BMX (Bus Matrix Switch) utilisent cette approche.

2.4.2 La Topologie d'Interconnexion

La façon dont sont interconnectés les commutateurs définit la topologie du réseau. La Figure 2.3 présente quelques unes des topologies souvent citées dans la littérature.

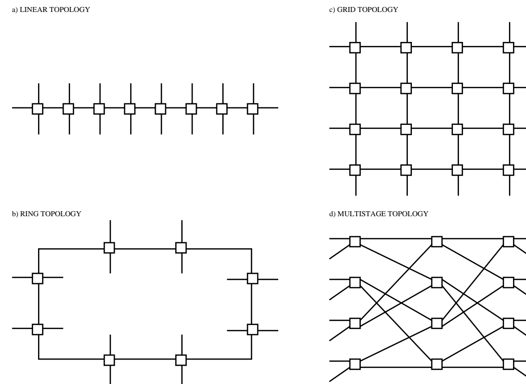


Figure 2.3: Quelques Topologies de Réseau

Les commutateurs peuvent être interconnectés en ligne droite ou en anneau. Le routage dans ces deux cas est très simple, et le nombre de liens requis croit de manière linéaire avec celui des noeuds de traitement et de modules mémoire à interconnecter. Cependant, ces types de topologies sont bloquantes.
Les commutateurs peuvent aussi être interconnectés suivant une grille pour par exemple constituer des crossbars qui sont non seulement non bloquants, mais leur nombre de liens croit de façon linéaire avec celui des noeuds de traitement et de modules mémoire à interconnecter. Par contre, le nombre de commutateurs dans les crossbars croit de façon quadratique avec celui des noeuds de traitement et de modules mémoire à interconnecter.
Les topologies multi-étages ont été étudiées pour réduire le nombre de commutateurs requis dans les réseaux crossbars, tout essayant de conserver des performances équivalentes.
Les réseaux de Clos ou Clos(e,m,r) sont des réseaux 3-étages ayant la structure [S_{em},S_{rr},S_{me}]. Cela signifie qu'ils sont constitués de r commutateurs S_{em} sur l'étage d'entrée, m commutateurs S_{rr} sur l'étage intermédiaire, et r commutateurs S_{me} sur l'étage de sortie, sachant qu'un commutateur S_{ij} a i entrées et j sorties.

Si $m \geq 2e$-1 alors le réseau de Clos(e,m,r) est non bloquant, c'est-à-dire qu'un chemin pourra toujours être établi entre une entrée et une sortie libre du réseau.
Sinon, si $m \geq e$ alors le réseau de Clos(e,m,r) est réarrangeable, c'est-à-dire qu'un chemin pourra toujours être établi entre une entrée et une sortie libre du réseau quitte à detourner quelques chemins déjà existants. Les réseaux de Bénès sont des exemples de réseaux réarrangeables. Ce sont des réseaux de Clos(2,2,2^k) se construisant recursivement à partir de deux réseaux de Clos(2,2,2^{k-1}), avec $k \geq 2$.
Sinon, le réseau de Clos(e,m,r) est bloquant, c'est-à-dire qu'il y aura des cas où il ne sera pas possible d'établir un chemin entre une entrée et une sortie libre du réseau.
Les commutateurs peuvent aussi être interconnectés suivant une topologie spatiale. Les réseaux qui en résultent, offrent un plus grand nombre de chemins entre une entrée et une sortie libre. Cependant, le routage y est plus complexe.

2.5 La Couche AAL

Le rôle de la couche AAL est non seulement de projeter le flux d'information généré par les applications dans la charge utile des cellules, mais aussi de fournir au dessus de la couche ATM, un service adapté aux contraintes des applications.
Cependant, autant il est difficile d'intégrer les contraintes des applications dans une seule et même couche AAL, essentiellement à cause de la difficulté à prédire l'évolution des applications et à la non compatibilité de certaines contraintes, autant il n'est pas réaliste de prévoir une AAL différente pour chaque type d'applications.
Le compromis qui a été trouvé consiste à regrouper les différents services à fournir aux applications en fonction des paramètres de communication suivants:

- débit de transmission (constant ou non)
- mode de connexion (connecté ou non connecté)
- synchronisation entre la source et la destination (avec ou sans)

C'est ainsi que le CCITT a pu définir les cinq types de couche AAL suivants:

- AAL type 1: pour les services à débit constant, tels que la transmission de la voix classique et l'émulation de circuit
- AAL type 2: pour les services à débit variable avec une synchronisation entre la source et la destination, tel que le transport video et d'un signal audio
- AAL type 3: pour les services de données en mode connecté, et la signalisation
- AAL type 4: pour les services de données en mode non connecté (actuellement combiné avec l'AAL type 3 pour constituer l'AAL type 3/4)
- AAL type 5: pour des services de données simples et efficaces en mode non connecté

Le Tableau 2.1 donne un bref aperçu des fonctions executées dans chaque type de couche AAL.

Type d'AAL	1	2	3/4	5
Fonctions de la sous couche SAR				
Segmentation et Réassemblage	x	x	x	x
Transfert de Blocks de Données	x			
Traitement des Erreurs dans les Cellules	x	x	x	
Traitement des Pertes et Insertions des Cellules	x	x	x	
Traitement des Cellules Partiellement Remplies		x	x	
Fonctions de la sous couche CS				
Récuprération d'Horloge	x	x		
Compensation de la Variation des Delais	x			
Contrôle de Flux			x	
Indication d'Espace Mémoire			x	
Bourrage des Messages			x	x
Traitement du Recouvrement des Messages			x	
Délimitation des Messages		x	x	
Multiplexage and Démultiplexage des Messages			x	
Detection des Erreurs dans les Messages				x

Table 2.1: AAL Functions

Chapter 3

Le Prototype Réseau

Au sein du Groupe de Recherche sur les Architectures Materielles (GRAM) du laboratoire Logiciels Systèmes Réseaux (LSR), nous étudions les réseaux d'interconnexion basés sur l'ATM car, nous pensons qu'ils sont capables de satisfaire aux besoins en communication sans cesse croissants dans les machines parallèles.
Cette étude passe par la réalisation de prototypes. Ces prototypes sont d'abord testés pour vérifier si leur fonctionnement est conforme aux spécifications. Ensuite, leurs performances sont évaluées.
Les tests de fonctionnement et évaluations des performances permettent non seulement d'ajuster certains paramètres de réseau, mais aussi de comparer différentes alternatives.
Dans notre démarche, nous avons écarté l'utilisation de prototypes matériels, car ils sont plus coûteux et leur délai de réalisation est plus élevé. Nous avons aussi écarté l'utilisation de prototypes issus d'outils traditionnels d'aide à la conception de circuits intégrés tels que, les outils VHDL (Very high speed integrated circuit Hardware Design Language), Verilog, Compass, Cadence, etc. Les charges de tels prototypes ne peuvent être décrites que de manière statique avant le début de la simulation, et sont donc de ce fait inappropriées pour des tests et évaluations réalistes.
Nous avons opté pour l'utilisation de prototypes définit en langage de programmation C. Ces derniers peuvent être testés et évalués directement sous des charges générées lors d'exécutions de rélles applications distribuées, écrites en C dans un style plus général MIMD (Multiple Instrruction Multiple Data) ou restreint SPMD (Single Program Multiple Data).

La suite de ce chapitre donne une description sommaire d'une première architecture de réseau ATM que j'ai conçu dans le cadre notre étude, et que j'ai décrit au niveau du transfert des registres (Register Transfer Level: RTL) en C, donnant ainsi naissance à l'unique prototype réseau C que nous avons réalisé jusqu'à présent.

3.1 La Couche Physique

Le système de transmission que j'ai retenu pour le prototype réseau est de type asynchrone. C'est celui qui est le plus adapté à la transmission octet par octet des cellules. Ses liens de transmission sont organisés en octets récurrents de période T. La couche physique au dessus n'effectue que le transport octet par octet des cellules entre les commutateurs. Elle n'effectue donc ni l'adaptation du rythme des cellules générées par la couche ATM à celui du système de transmission, ni le contrôle d'erreurs sur les entêtes des cellules, et ni la délimitation des cellules.
C'est dans ce contexte qu'un bus 8-bit fonctionnant à la fréquence $\frac{1}{T}$ implémente à la fois le système de transmission et la couche physique sur le prototype.

3.2 La Couche ATM

3.2.1 Les Commutateurs

Les commutateurs de la couche ATM du prototype réseau sont du type BMX. Ils utilisent l'approche files d'attente multiple en sortie qui leur permettent d'avoir les mêmes performances que les commutateurs avec files d'attente en sortie classiques, mais en pouvant fonctionner à la même vitesse que les liens de transmission.
Comme l'indique la Figure 3.1, chaque commutateur du prototype réseau a 4 entrées et 4 sorties.
A chaque entrée se trouve un module PPD (Primary Packet Distributor). Et, à chaque sortie se trouve un module SPD (Secondary Packet Distributor).
La Figure 3.2 montre l'architecture interne 8-bit des modules PPD que j'ai définie.
Les cellules sont transmises aux modules PPD octet par octet. Au fur et

Figure 3.1: Les Commutateurs BMX du Prototype Réseau

Figure 3.2: Architecture 8-bit des modules PPD

à mesure qu'ils reçoivent ces octets, ils traduisent les champs VPI (Virtual Path Indentifier) des entêtes et stockent les cellules dans les files d'attente conformememt aux informations contenues dans leur table de routage, à la manière d'un opérateur pipeline.
Cependant, vu l'architecture très simple du système de transmission et de la couche physique du prototype réseau, les modules PPD ont besoin d'un signal 1-bit supplémentaire leur indiquant les octets sur les liens de transmission qui transportent des fragments de cellules. Ils utilisent aussi un compteur modulo 53 pour délimiter les cellules.
Les tables de routages dans les modules PPD sont des mémoires à accès direct de 256 entrées de taille 52 bits, avec un port de lecture et un port d'ecriture. La Figure 3.3 décrit la structure des entrées des tables de routage.

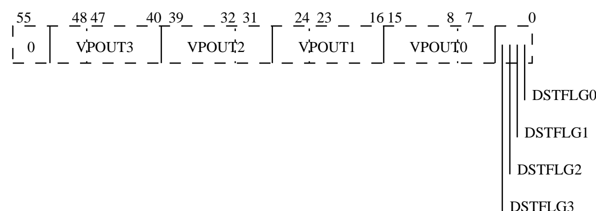


Figure 3.3: Structure des Entrées des Tables de Routage

Ainsi, lorsqu'un module PPD reçoit une cellule, il consulte l'entrée de sa table de routage qui est indiquée par le champ VPI de cette cellule. Si le bit $DSTFLG_i$ de cette entrée est positionné à 1, alors une copie de la cellule entrante, comportant la sous chaîne de bits $VPOUT_i$ de cette entrée dans le champ VPI de son entête, sera stockée dans la file d'attente numéro i, pour i variant de 0 à 3. Au cours de cette opération, si l'une des files d'attente dont le bit DSTFLG correspondant est positionné à 1 est pleine à l'issue de la traduction de l'entête de la cellule entrante, alors aucune copie de cette cellule n'est stockée dans aucune file d'attente, et le module PPD concerné envoie un signal de suspension de transmission en amont.
La Figure 3.4 montre l'architecture interne 8-bit des modules SPD que j'ai définie.
Le composant essentiel dans l'architecture des modules SPD est le processeur interne. La Figure 3.5 montre l'architecture interne ce processeur.
Sa partie opérative a une achitecture à base de multiplexeurs, et le fonction-

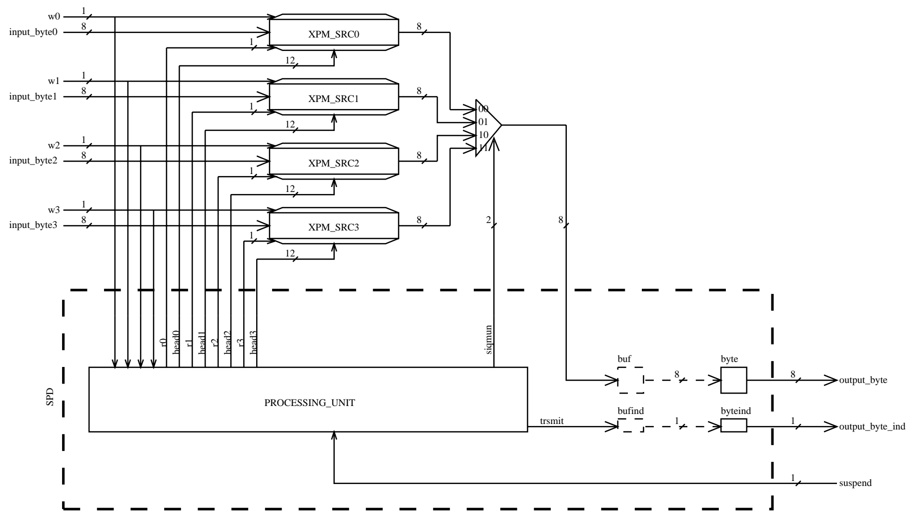


Figure 3.4: Architecture 8-bit des modules SPD

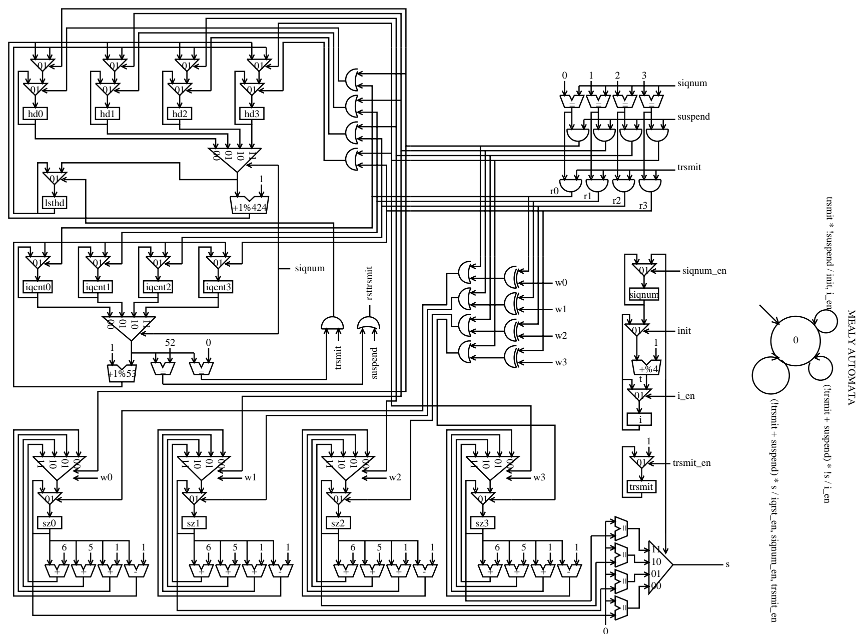


Figure 3.5: Architecture du Processeur interne des modules SPD

nement de sa partie contrôle est décrite par un automate de MEALY.
Essentiellement, les modules SPD extraient les cellules de leurs files d'attente et les transmettent sur leur lien de sortie octet par octet, en accompagnant chaque transmission d'octet d'un envoi d'un signal d'un bit pour indiquer son appartenance à une cellule ATM.
Les modules SPD utilisent une stratégie basée sur *le tourniquet* pour selectionner les files d'attente qui doivent être servies. Et, ils utilisent une stratégie First In First Out (FIFO) pour selectionner, à l'intérieur des files d'attente, les cellules qui doivent être transmises.
Lorsqu'un module SPD reçoit un signal de suspension, il interrompt sa transmission courrante, et fait un tour de selection de ses files d'attente.

3.2.2 La Topologie d'Interconnexion

Les topologies d'interconnexion peuvent classées en trois groupes:

- non bloquante
- réarrangeable
- bloquante

Les topologies réarrangeables réalisent un excellent compromis entre les topologies non bloquantes qui sont plus coûteuses et les topologies bloquantes qui offrent des performances médiocres.
C'est pour cette raison que j'ai choisi pour topologie d'interconnexion du prototype réseau, celle décrite dans la Figure 3.6.
C'est un réseau de Clos(4,4,4) réarrangeable. Il fournit 4 chemins possibles aux communications point-à-point entre chaque entrée et chaque sortie, et 4 chemins possibles aux diffusions depuis chaque entrée du prototype réseau.
La Figure 3.7 nous montre les 4 chemins qui existent pour les communications point-à-point entre l'entrée numéro 5 et la sortie numéro 14 du prototype, tandis que la Figure 3.8 nous montre les 4 chemins qui existent pour les diffusions depuis l'entrée numéro 5 du prototype réseau.

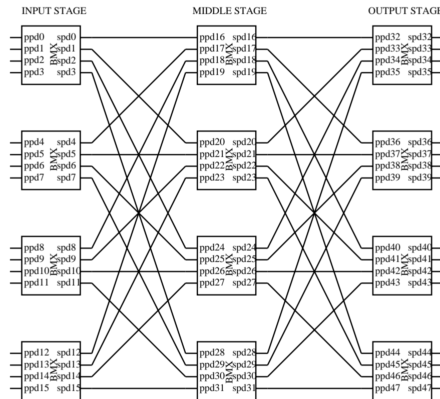


Figure 3.6: Topologie d'Interconnexion du Prototype Réseau

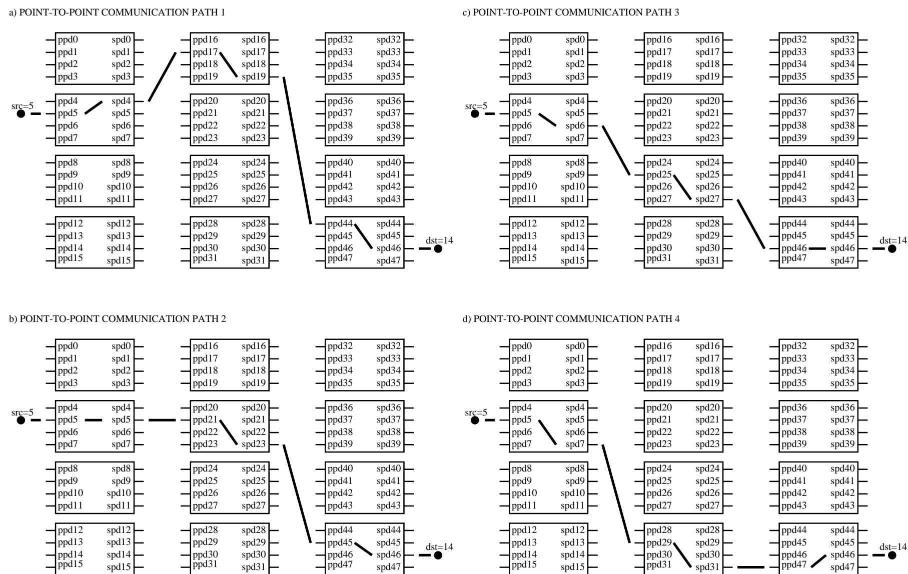


Figure 3.7: Les 4 Chemins possibles pour Une Communication Point-à-Point

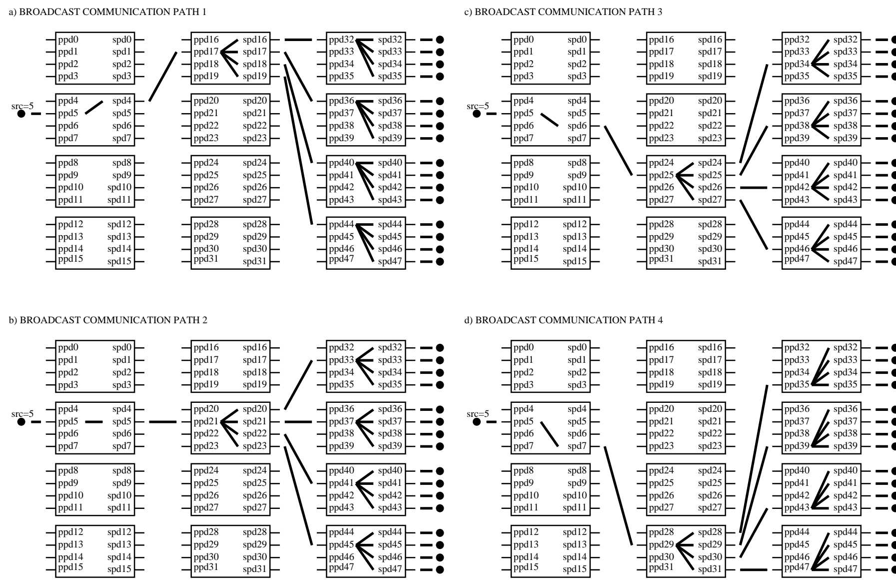
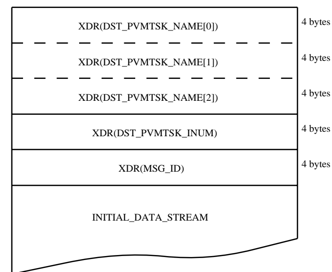


Figure 3.8: Les 4 Chemins possibles pour Une Diffusion

3.3 La Couche AAL

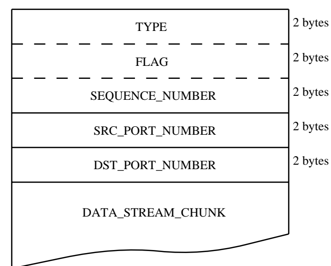
3.3.1 Présentation Générale

La couche AAL du prototype réseau est une version allégée de la couche AAL type 3/4 définie par le CCITT.
Conformement au Tableau 2.1, sa sous-couche SAR (Segmentation And Reassembly) n'effectue que la segmentation et le réassemblage des messages, et le traitement des erreurs, pertes, et insertions des cellules, tandis que sa sous-couche CS (Convergence Sub-layer) n'effectue que la délimitation, le multiplexage, et le démultiplexage des messages.
De plus, le format des unités de donnée de protocole de ces deux sous couches a été légèrement modifié. La Figure 3.9 montre le format des entêtes PVM qui sont utilisés comme entêtes de la sous-couche CS du prototype réseau.
La Figure 3.10 montre le format des entêtes de la sous-couche SAR du prootype réseau.
Elle a été implémentée de cette manière afin de permettre une mise en œuvre naturelle et optimale de PVM au dessus d'elle.



XDR : EXTERNAL (or MACHINE INDEPENDENT)
DATA REPRESENTATION
INUM : INSTANCE NUMBER
MSG_ID : MESSAGE IDENTIFIER
DST : DESTINATION
PVMTSK : PVM TASK

Figure 3.9: Format des Entêtes PVM



TYPE ::= ACK or MESSAGE
FLAG ::= BEGINNING and/or MIDDLE and/or END

SRC : SOURCE
DST : DESTINATION

Figure 3.10: Format des Entêtes de la sous couche SAR

En effet, la sous couche SAR peut ne pas s'occuper du problème des cellules partiellement remplies, car PVM le fait déjà d'une façon implicite.
En outre, la sous couche CS peut être exemptée d'effectuer un contrôle de flux de bout-en-bout, car un contrôle de flux point-à-point est implementé dans le couche ATM.
La sous couche CS peut aussi être dispensée d'indiquer à la couche PVM les tailles des espaces mémoire où vont être stockés les messages reçus, car PVM peut résoudre cette question en mettant, si besoin est, les tailles des messages au début de ces derniers.
Par ailleurs, il n'y a pas en PVM une motivation irrésistible à voir la sous couche CS générer un flot continu de cellules, et donc d'effectuer le bourrage des messages.
Enfin, la sous couche SAR utilise une plage de numérotation des séquences de transmission des cellules suffisamment large de manière à éviter à la sous-couche CS d'avoir à contrôler le recouvrement des messages, tellement il est peu probable qu'il se produise.

3.3.2 Les Cartes AAL

La Figure 3.11 présente le plan de masse des cartes AAL du prototype réseau que j'ai implementé.
Ces cartes AAL communiquent avec les machines hôtes par un bus de données de 32 bits et un ensemble de signaux de contrôle.
Elles transmettent (respectivement reçoivent) les cellules aux (respectivement des) commutateurs en aval (respectivement amont) par des liens de transmission accompagnés de signaux 1-bit indiquant la présence ou l'absence de données valides sur ces liens, et reçoivent d'eux (respectivement leur transmettent) des signaux de suspension 1-bit en cas de saturation de leurs files d'attente (respectivement leur buffer de reception).
En interne, ces cartes AAL sont chacunes constituées de: onze controleurs, un buffer d'émission, un buffer de reception, un compteur modulo 53 pour la délimitation des cellules reçues des commutateurs, et d'un ensemble de registres utiliser pour la gestion des cellules et des messages dans les buffers d'émission et de reception.
Les buffers d'émission et de reception sont structurés en blocs de 53 octets, et sont gérés chacun par une liste chainée de blocs libres, et une liste de blocs alloués. Cela permet une gestion simple et efficace de ces buffers. Par

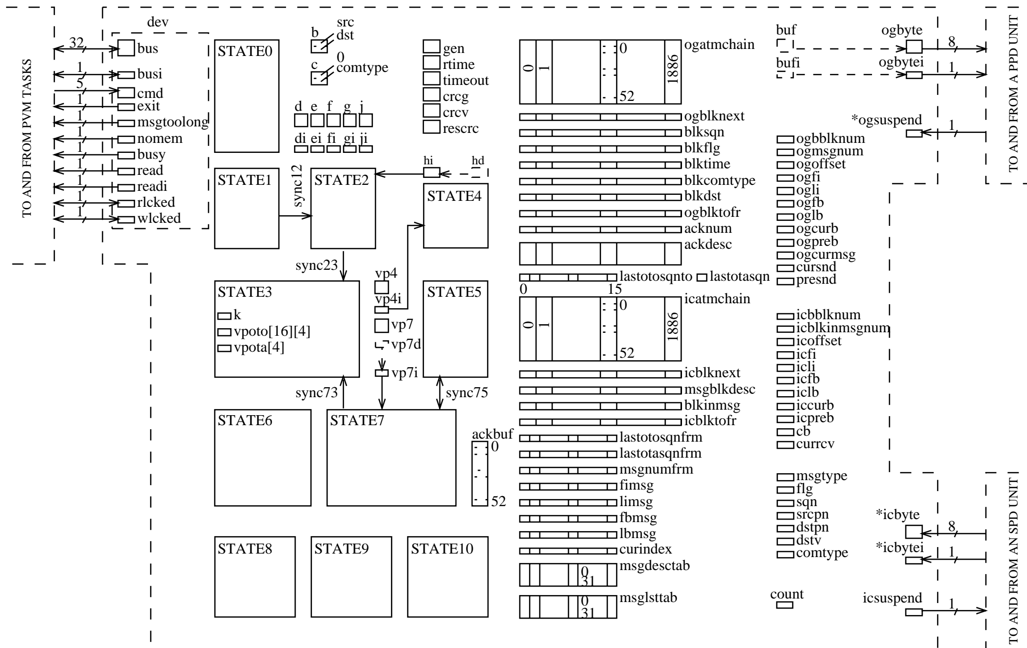


Figure 3.11: Plan de Masse des Cartes du Prototype

exemple, le mécanisme d'insertion d'une cellule dans un buffer est constitué de l'allocation du bloc de tête de la liste des blocs libres de ce buffer, et de son insertion en queue de la liste des bocks alloués de ce même buffer. Et, le mécanisme de retrait d'une cellule d'un buffer est constitué de l'extraction du bloc alloué contenant cette cellule de la liste des blocs alloués de ce buffer, et de son insertion en queue de la liste des blocs libres de ce même buffer. Ces mécanimes sont effectués en un coût constant.
Le controleur STATE0 gère l'interface de communication avec les machines hôtes, et effectue la segmentation en ligne des messages, directement dans le buffer d'émission, au cours de leur reception.
Le controleur STATE1 constitue les entêtes PVM et celles de la sous couche SAR et les insère aux endroits appropriés dans les blocks alloués du buffer d'émission.
Le controleur STATE2 constitue les entêtes ATM, assisté des controleurs STATE3 et STATE4, et les insère aux endroits appropriés dans les blocks alloués du buffer d'émission.
Le controleur STATE5 a la charge de la transmission et retransmission après timeout des cellules du buffer d'émission, du calcul en ligne pendant ces transmissions et retransmissions des CRCs correspondant, et du retrait des cellules du buffer d'émission sur reception des acquittements correspondants ou après un nombre maximum de retransmissions. Il a aussi en charge la transmission et le calcul en ligne durant la transmission des CRCs des accusés de reception générés par le controleur STATE7.
Le controleur STATE6 a la charge de la reception des cellules en provenance des commutateurs en amont, et de leur rejet ou insertion dans le buffer de reception après vérification de leur CRC et de la destination.
Le controleur STATE7 parcourt les cellules du buffer de reception. Si la cellule courante est un accusé de reception, alors il réduit le nombre d'acquittements attendu de la cellule du buffer d'émission correspondante, et la retire du buffer de reception. Sinon, il génère, assisté du controleur STATE3, un accusé de reception pour cette cellule si cela n'a pas encore été fait, et laisse au controleur STATE5 le soin de le transmettre. Ensuite, il l'utilise si possible dans le réassemblage d'un message. Il a aussi en charge de retirer du buffer de reception les cellules appartenant à des messages arrivés au terme de leur durée de vie.
Le controleur STATE8 sert les messages aux processus des applications.
Le controleur STATE9 réduit tous les timers positifs de la carte d'une unité

à chaque cycle d'horloge.
Le controleur STATE10 marque les messages qui sont arrivés au terme de leur durée de vie.

3.4 Vision Générale du Prototype

Comme le montre la vue générale du prototype réseau donnée dans la Figure 3.12, ce dernier a 16 ports d'entrée/sortie.

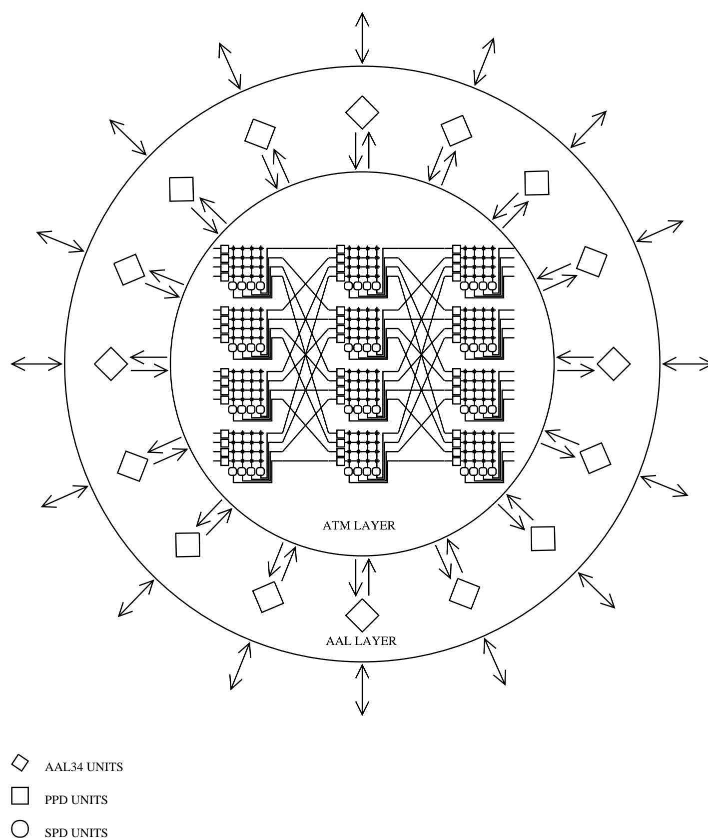


Figure 3.12: Vue Générale Prototype Réseau

En gros, il est constitué de 16 cartes AAL, 48 modules PPD, et 48 modules SPD.

Le noyau du programme qui simule le fonctionnement du prototype réseau est defini par l'extrait de code C suivant:

```
...
struct ppd ppdlst[48];
struct spd spdlst[48];
struct aal3_4 aal34lst[16];
...
for( ; ; )
{
  for (i=0; i<48; i++) spd_scheduler(&spdlst[i]);
  for (i=0; i<16; i++) aal34_scheduler(&aal34lst[i]);
  for (i=0; i<48; i++) ppd_scheduler(&ppdlst[i]);
}
...
```

Vu que les procédures spd_scheduler(), aal34_scheduler(), et ppd_scheduler() simulent le fonctionnement des modules SPD et PPD, et des cartes AAL qui leur sont passés en paramètres lors de leurs appels pendant un cycle d'horloge, alors nous pouvons dire qu'une itération de la boucle "for" infinie correspond à un cycle d'horloge.

3.5 Le Service de Données en mode Non Connecté au dessus du Prototype

La couche AAL du prototype réseau implémente un service de données en mode non connecté. Cela veut tout simplement dire que les communications à travers le prototype réseau se déroulent sur des connexions sans contrat de débit et établies une fois pour toute à la configuration du prototype.
Chaque AAL source dispose de 4 chemins différents vers une AAL de destination donnée pour ses communications point-à-point, et de 4 différents chemins pour ses diffusions.
De plus, les cellules d'un même message ne sont plus tenues de suivre le même itinéraire.
Les cartes AAL du prototype réseau utilisent une stratégie basée sur le tourniquet pour sélectionner les itinéraires à suivre par les cellules d'une communication quelconque.

Chapter 4

La Plateforme PVM

Au sein de notre laboratoire, le Groupe de Recherche sur les Architectures Materielles (GRAM) s'interesse à deux interfaces de programmation d'applications (Application Programming Interface: API) parallèles:

- PVM (Parallel Virtual Machine) developpé au laboratoire national d'Oak Ridge (Oak Ridge National Laboratory: ORNL).
- MPI (Message Passing Interface) developpé à l'université du Tennessee Knoxville.

Ces deux interfaces permettent toutes les deux d'écrire des programmes parallèles de type plus général MIMD (Multiple Instruction Multiple Data) ou restreint SPMD (Single Program Multiple Data), et d'utiliser une collection d'ordinateurs scalaires, vectoriels, multiprocesseurs, etc..., interconnectés par un ou plusieurs réseaux comme une seule et unique ressource de traitement: elles permettent une certaine "portabilité" des programmes parallèles.
Bien que MPI offre un jeu de primitives de communication plus varié, et qu'il possède la notion de type de donnée dérivé, que n'a pas PVM et qui permet de transférer directement des données non contigües en mémoire (par exemple les éléments d'une colonne d'une matrice), réduisant ainsi le nombre de recopies mémoire à mémoire au cours des opération de communications, j'ai implémenté une plateforme PVM, tout simplement parce que MPI n'était pas officiellement sorti lorsque j'ai commencé ces travaux fin 1993.
Sur cette plateforme, les processus des applications se connectent à un processus qui simule le réseau d'interconnexion sous test, et communiquent à

travers lui, générant ainsi directement sa charge.
La communication entre les processus des applications et le processus réseau se fait aux moyens de segments de mémoire partagés des IPCs (Inter-Process Communication) d'Unix. Notons que l'utilisation de ces IPCs implique que les applications et les implementations réseaux doivent être décrites en langage de programmation C.
En effet, le processus réseau crée un segment de mémoire partagé pour chacun de ses ports d'entrée/sortie au cours de sa phase de configuration.
Ensuite, les processus des applications se connectent sur ces ports d'entrée/-sortie en récupérant les segments de mémoire partagés correspondants.
Cependant, les primitives de communication standard de la librairie PVM copient les données des mémoires locales des processus appelants dans des *sockets* Unix UDP ou TCP, causant ainsi le déroulement des communications à travers Unix ou Ethernet.
Sur ce, j'ai reécrit ces primitives de façon à ce qu'elles copient les données des mémoires locales des processus appelants dans les segments de mémoire partagés correspondant aux ports d'entrée/sortie du prototype réseau auxquels ces processus sont connectés.
Le Tableau 4.1 fournit la liste des primitives que j'ai reécrit.
La syntaxe et la sémantique de ces primitives sont consistantes avec celles des primitives standard de la librairie PVM.
Cela permet de n'effectuer que de mineurs changements, de nature syntaxique essentiellement, dans les applications PVM existantes afin de les exécuter sur ma plateforme PVM.
Une présentation plus détaillée des interfaces PVM et MPI, ainsi que de la plateforme de simulation réalisée, figure au chapitre 10.

routine synopsis
atminitsend(dev) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ - initializes the send buffer of the AAL unit indicated by *dev*
atmputn*type***(dev, np, cnt)** **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ *type* ***np;** **int cnt;** - packs *cnt* successive elements, of the type indicated by *type*, from the calling process' - memory starting at the address *np* into the send buffer of the AAL unit indicated by *dev* - *type* must be int, or short, or long, or float, or dfloat, or cplx, or dcplx
atmputbytes(dev, np, cnt) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** **int cnt;** - packs *cnt* successive characters from the calling process' memory starting at - the address *np* into the send buffer of the AAL unit indicated by *dev*
atmputstring(dev, np) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** - packs the character string from the calling process' memory starting at the address *np* - into the send buffer of the AAL unit indicated by *dev*
atmgetn*type***(dev, np, cnt)** **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ *type* ***np;** **int cnt;** - unpacks *cnt* successive elements of the type indicated by *type* - from the receive buffer of the AAL unit indicated by *dev*, - and stores them to the calling process' memory starting at the address *np* - *type* must be int, or short, or long, or float, or dfloat, or cplx, or dcplx
atmgetbytes(dev, np, cnt) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** **int cnt;** - unpacks *cnt* successive characters from the receive buffer of the AAL unit indicated - by *dev*, and stores them to the calling process' memory starting at the address *np*
atmgetstring(dev, np) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** - unpacks a character string from the receive buffer of the AAL unit indicated by *dev*, - and stores it into the calling process' memory starting at the address *np*
atmsnd(dev, prcstab, proc, inum, type) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **struct prcsdescstr *prcstab;** /* reference to the process description table */ **char proc[3];** /* destination proces' 3-length name. */ **int inum;** /* destination process' instance number */ **int type;** /* message type */ - grants to the AAL unit indicated by *dev*, the permission to sends the last message packed into - its send buffer to any process whose name, instance number, and expected message type match - the last three parameters of the call. Note that, $type \geq 0$
atmrcv(dev, type) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **int type;** /* message type */ - selects a message on the receive buffer of the AAL unit indicated by *dev*, whose PVM header - matches the name, instance number, and expected message type specified by the calling process - This routine is blocking (i.e. it does not return until it selects a matching message)

Table 4.1: Les Primitives de Communication C Ré-écrites

Chapter 5

Tests et Expériences

5.1 Le Test Structurel du Prototype de Réseau

Le test de fonctionnement du prototype réseau a été effectué de manière structurelle aux differentes étapes de son developpement. C'est ainsi qu'ont été testés dans l'ordre, les modules PPD, les modules SPD, le fonctionnement en tandem entre un module SPD en amont et un module PPD en aval, les commutateurs, et les cartes AAL.
Au cours de tous ces tests, j'ai utilisé l'approche de test par simulation classique. Pour chacun des cas ci-dessus mentionné j'ai écrit un programme de simulation. Comme c'est le cas avec les outils classiques de simulation de circuits VLSI, chacun de ces programmes prend en entrée une description du ou des circuits à tester et un fichier contenant une description des stimulis d'entrée et une liste des stimulis de sortie, et retourne un fichier contenant les traces de la simulation. Notons cependant que le fichier de stimuli du programme de simulation des cartes AAL ne comporte pas de description des stimulis d'entrée, tout simplement parce que ce programme utilise le transfert d'un message entre deux processus d'une application réelle via le prototype réseau, pour ne s'intéresser qu'au fonctionnement des cartes AAL source et destination.

5.2 Le Test de la Plateforme PVM

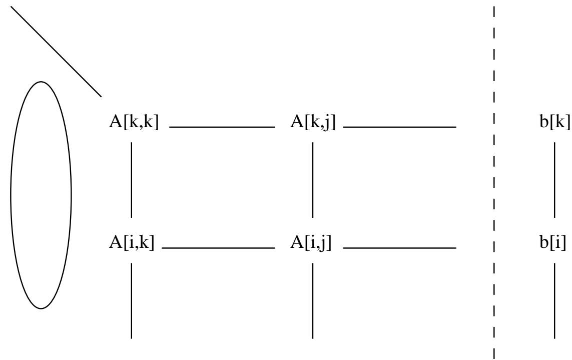
Le test de la plateforme PVM permet non seulement de vérifier que l'interface entre PVM et le prototype réseau fonctionne coome il le faut, mais aussi d'observer le comportement du prototype réseau directement sous une charge générée lors de l'exécution d'une application distribuée réelle.
Deux applications bien connues ont été utilisées: la résolution de systèmes d'équations linéaires et la multiplication de matrices.

5.2.1 La Résolution d'Equations Linéaires

Résoudre un système d'équations linéaires revient à trouver si possible un vecteur x de dimension p vérifiant l'équation Ax=b, où A est une matrice de dimension mxp et b un vecteur de dimension m, résultant du système à résoudre.
La Figure 5.1 présente le pseudocode décrivant l'algorithme de Gauss Jordan, très souvent utilisé pour la résolution de tels systèmes.
On y remarque une phase de triangulation et une autre de calcul du résultat. Lors de la phase de triangulation, les composantes A_{ik}, A_{kj} (respectivement b_k), et A_{kk} sont requises pour la transformation de la composante A_{ij}, avec 0≤k≤m-2, k+1≤i≤m-1, et k≤j≤m-1. La composante A_{kk} est appelée pivot. Au cours de la phase de calcul du résultat, les composantes du vecteur résultat x sont calculées une par une de x_{m-1} à x_0. Les composantes b_k, A_{kk}, ainsi que toutes les composantes x_j et A_{kj} tel que k+1≤j≤m-1, sont requises pour le calcul de la composante x_k, pour k variant de m-1 à 0.
Pour les tests de la plateforme PVM, je me suis limité à la résolution des systèmes carrés, c'est-à-dire avec m=p=N, dont les systèmes Ab résultants sont des matrices de dimension Nx(N+1).
L'application de résolution distribuée d'un système d'équations linéaires est constituée:

- d'un processus d'initialisation, dont le code C est montré en Appendice K.1
- d'un ensemble de processus de résolution, dont le code C est montré en Appendice K.2

Le processus d'initialisation lit une première fois le système Ab initial dans le fichier qui lui passé en paramètre sur sa ligne de commande, et détermine



Let A be an NxN Matrix, and x and b be two N-size vectors.

```
for (k=0; k<N-2, k++)
    for (i=k+1; i<=N-1; i++)
        for (j=k; j<=N-1; j++)
            A[i,j] = A[i,j] - (A[i,k]*A[k,j])/A[k,k];
        endfor
        b[i] = b[i] - (A[i,k]*b[k])/A[k,k];
    endfor
endfor

for (k=N-1; k>=0; k--)
    x[k] = (b[k] - A[k,k+1]*x[k+1] - ... - A[k,N-1]*x[N-1])/A[k,k];
endfor
```

Figure 5.1: Algorithme de Gauss Jordan

la valeur de N.
Il lit ensuite une seconde fois le système Ab initial. Cette fois, pour chaque composante lue, il initie un nouveau processus de résolution et lui passe la composante lue et la valeur de N via le prototype réseau. Il va ainsi en initier N*(N+1).
Le processus d'initialisation va recevoir les composantes du système Ab triangulé lors de la phase de triangulation et les composantes du vecteur x lors de la phase de calcul du résultat, et les écrire à la fin du fichier initial.

5.2.2 La Multiplication de Matrices

L'objectif ici est de calculer la matrice C de dimension MxN, résultat de la multiplication de la matrice A de dimension MxP par la matrice B de dimension PxN.
La Figure 5.2 présente l'algorithme séquentiel standard de multiplication de deux matrices.

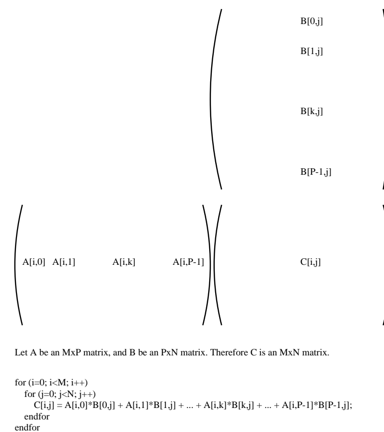


Figure 5.2: Algorithme Séquentiel de Multiplication de Matrices

Il en résulte que la composante C_{ij} de la matrice résultat C, est le produit vectoriel de la ième ligne de la matrice A par la jème colonne de la matrice B, d'où la formule:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} * B_{kj}$$

L'application de multiplication de deux matrices est constituée:

- d'un processus d'initialisation, dont le code C est montré en Appendice K.4
- d'un ensemble de processus de multiplication, dont le code C est montré en Appendice K.5

Le processus d'initialisation lit une première fois les matrices A et B à multiplier dans le fichier qui lui est passé en paramètre sur sa ligne de commande afin de déterminer les valeurs des paramètres M, P, et N.
Ensuite, il lit une seconde fois les deux matrices A et B. Cette fois, pour chaque composante lue, il initie un nouveau processus de multiplication et lui passe la composante lue ainsi que les valeurs des paramètres M, P, et N via le prototype réseau. Il va ainsi initier M*P+P*N processus de multiplication.
Les processus de multiplication vont par la suite coopérer pour calculer la matrice résultat dont les composantes sont passées au processus d'initialisation au fur et à mesure qu'elles sont calculées. Ce dernier va les écrire à la fin du fichier initial.

5.3 L'Evaluation des Performances du Prototype Réseau

Une analyse détaillée de l'architecture et du fonctionnement du prototype réseau a permis d'évaluer le delai théorique du transfert des messages de taille N, en nombre de cycles d'horloge.
J'ai obtenu l'égalité suivante (cf. Chapitre 11):
délai de transfert théorique =
$145+\lceil\frac{N+20}{36}\rceil*10+(\mathrm{Ndiv4})*38+\lceil\frac{Nmod4}{N}\rceil*(6+8*(\mathrm{Nmod4}))+$
$(\lceil\frac{N+20}{36}\rceil>1?58:0)+(\lceil\frac{N+20}{36}\rceil>2?(\lceil\frac{N+20}{36}\rceil-2)*61:0)$ cycles d'horloge.
Le délai de transfert a aussi été mesuré de façon expérimentale lors de l'exécution de l'application echo où un processus envoie un message de taille N à un autre via le prototype réseau, les deux processus collaborant pour mesurer la durée de la transmission.
La Figure 5.3 montre les courbes du delai de transfert théorique et expérimental en fonction de la taille des messages.

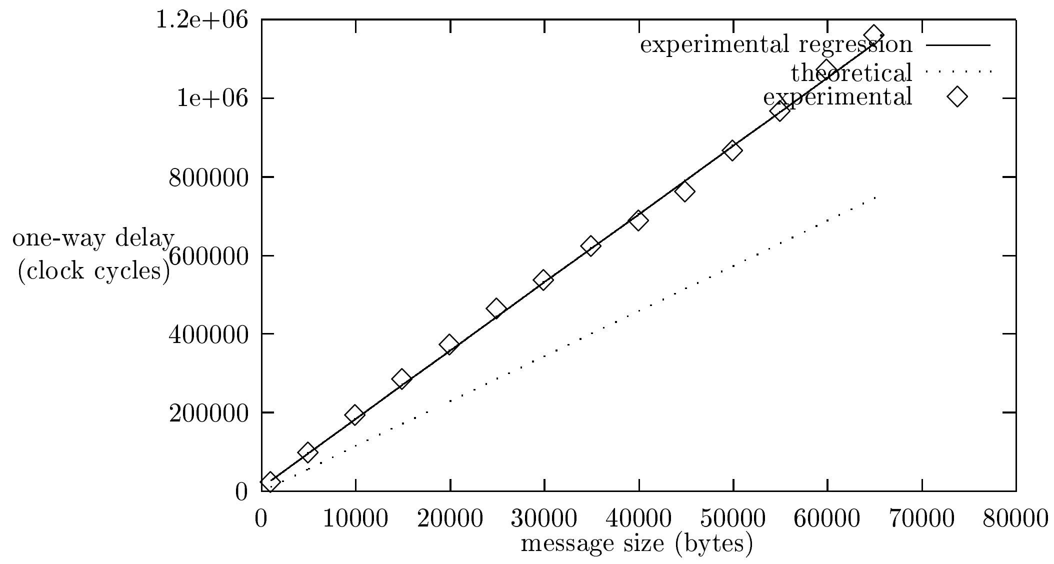
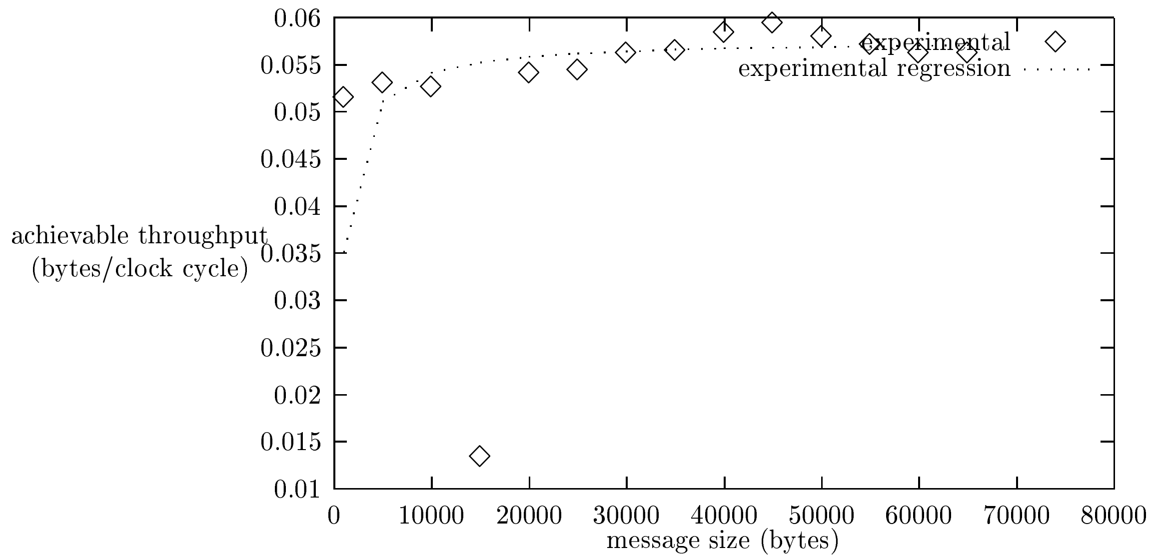


Figure 5.3: délai de transfert VS taille des messages

Au départ le serveur de notre laboratoire était une machine monoprocesseur gérée par le système d'exploitation Unix.
La synchronisation entre les processus réseau et applications a donc été conçue sous l'hypothèse selon laquelle, tous les processus sont exécutés sur un même processeur et leur ordonnancement est basé sur une stratégie appelée: *tourniquet* (round-robin).
Au moment de l'expérimentation, un second processeur lui avait été rajouté et Solaris était devenu son nouveau système d'exploitation. Or Solaris ne peut assurer un ordonnancement des processus basé sur le *tourniquet* que sur chacun des processeurs, et pas sur les deux à la fois.
Ainsi, lorsqu'un processus d'une application copie un message dans le buffer d'émission ou extrait un message du buffer de reception en plusieurs passes (ce qui est le cas lorsque le message est plus grand que quatre octets), il est possible que le processus réseau itère sa boucle plus d'une fois entre deux passes consecutives, si les deux processus ne s'exécutent pas sur le même processeur. D'où l'apparition du décalage observé entre les droites expérimentale et théorique.
La bande passante du prototype réseau utilisée pour le transfer d'un message est obtenue en divisant la taille de ce message par le delai mesuré à l'issue de ce transfert. La Figure 5.4 nous montre les courbes de la bande passante utilisée en fonction de la taille des messages.
Les résultats obtenus montrent d'une part que les mesures théoriques et expérimentales se consolident, et d'autre part qu'ils sont ceux attendus.
En effet, le délai de transfert croît de façon linéaire avec la taille des messages, et que la bande passante utilisée lors des transferts croît dans un premier temps, puis sature par la suite.

5.4 L'Impact des Conflits dans le Prototype Réseau

La plateforme PVM a été utilisée pour évaluer l'impact des conflits dans le prototype réseau sur l'exécution d'une application.
L'approche utilisée consiste à comparer le fichier contenant les traces des communications durant l'exécution d'une application sur la plateforme, et celui contenant les traces des communications durant une exécution théorique de



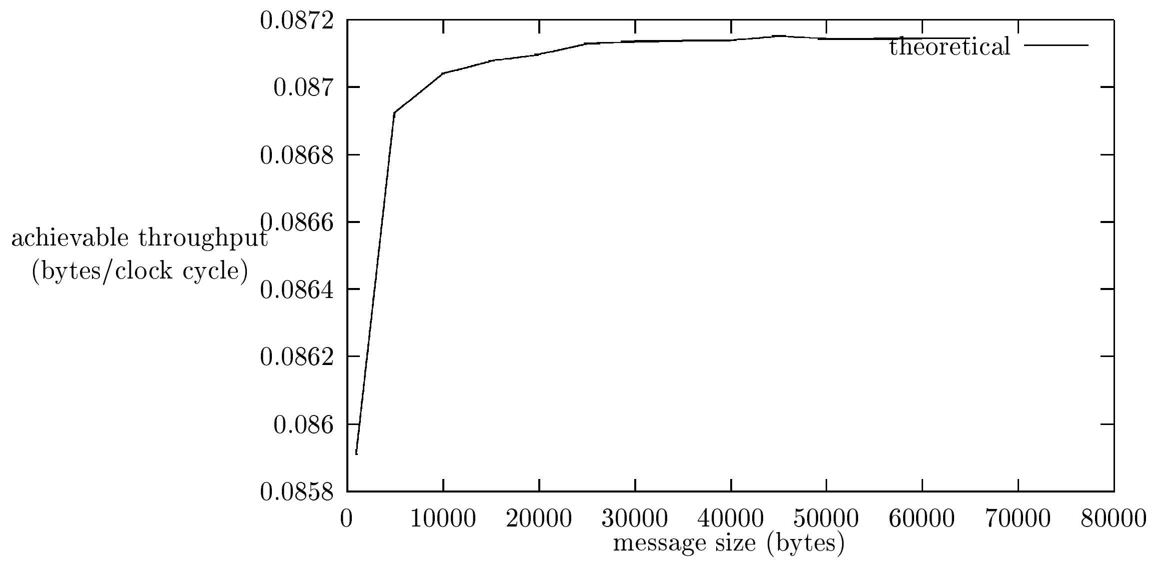


Figure 5.4: bande passante utilisée VS taille des messages

la même application qui suppose que le prototype réseau est sans conflits.
C'est ainsi que pour l'exécution de la résolution parallèle d'un système d'équations linéaires, mettant en œuvre 20 processus de calcul et leurs communications, j'ai pu observer que les conflits dans le prototype réseau ont rallongé l'exécution d'un delai supplémentaire de l'ordre de 36% du temps d'exécution total.
Il est intéressant d'étudier l'impact de la topologie et de la taille des files d'attente dans les commutateurs sur le ralentissement causé par la présence des conflits au sein du réseau d'interconnexion.
Malheureusement, nous n'avons pas eu le temps de procéder à ces études. Cependant, l'outil d'évaluation est en place, et est opérationnel.

Chapter 6

Conclusion

Le contexte des travaux présentés dans ce mémoire est l'étude des réseaux d'interconnexion basés sur l'ATM, car nous pensons qu'ils sont capables de satisfaire les besoins en communication sans cesse croissants dans les machines parallèles actuelles et futures.
Cette étude passe par la réalisation de prototypes réseau en langage de programmation C et d'une plateforme de simulation PVM permettant de tester et comparer rapidement différentes architectures de réseau directement sous des charges générées lors de l'exécution d'apllications distribuées réelles.
La plateforme PVM est actuellement opérationnelle. Un premier prototype réseau C a été réalisé et testé puis évalué sur la plateforme PVM. Les résultats de cette évaluation sont ceux attendus. En effet, le délai de transfert du prototype réseau croît de manière linéaire avec la taille des messages, et sa bande passante utile croît fortement pour de petites tailles de messages puis sature pour de plus grandes tailles. Cette plateforme a aussi permi de mesurer l'impact des conflits au sein du prototype réseau sur l'exécution d'une application.
Les résultats obtenus jusqu'à présent, indiquent que l'utilisation de la plateforme de simulation PVM et la rélisation de prototypes en langage de programmation C sont de bon outils pour notre étude sur les réseaux d'interconnexion basés sur l'ATM. Cependant, nous sommes loin d'avoir exploité les possibilités qu'elles offrent.
Jusqu'à présent, seules deux applications ont été exécutées sur la plateforme PVM. Il serait intéressant d'écrire de nouvelles, ou d'utiliser d'autres déjà existantes.

Il serait aussi intéressant pour un prototype réseau donné, de conserver son architecture et de ne faire varier que la taille des files d'attente, afin de mesurer l'impact sur l'occurrence des conflits par exemple.
Il serait aussi intéressant d'implémenter d'autres architectures, c'est-à-dire d'autres systèmes de transmission de support, couche physiques, commutateurs ATM, topologies d'interconnexion, cartes d'adaptation à l'ATM, contrôles de congestion, contrôles de flux, contrôles de traffic, un service de donnée en mode connecté, etc.
Il serait enfin intéressant de réaliser un prototype matériel correspondant à un prototype C et comparer leurs performances. Les performances du prototype réseau présenté dans ce mémoire devraient différer très peu de celles d'un eventuel prototype matériel correspondant.

Part II

The Detailled Dissertation

Chapter 7

Introduction

Interconnection networks have long been the bottleneck in the bosom of parallel computers.
With the passing of the years, microprocessors have became more and more powerful.
In the meantime, the interconnection networks have undergone an evolution that does not provide an reasonnable and satisfactory solution to fulfill the unceasingly growth of the communication needs within parallel computers due to the unceasingly increase of both microprocessors computing power, and memories speed and storage capacity, in terms of high transmission rates and low tranfer delay.
Among the various interconnection network architectures that have been designed, there is one based on point-to-point links. In this approach, processing nodes are connected either together in the case of distributed memory parallel computers, or to memory banks in the case of shared memory parallel computers, using point-to-point links. This first architecture type achieves interconnection networks without contention. However, the number of interconnection links required to achieve such interconnection networks quadratically increases with the number of processing nodes to interconnect.
There is another architecture type based on buses. In this second approach, all the processing nodes, and memory banks in the case of shared memory parallel computers, are connected to a central bus. This second architecture type achives interconnection networks, whose intrconnection links number linearly increases as the number of processing nodes to interconnect. However, in such an interconnection network, the bus is a unique resource shared

by all the processing nodes. Hence, it introduces a quite high number of contentions. In addition, the bandwidth of the bus is constant. Therefore, the part of this bandwidth which is dedicated to the communications of each processing node is in inverse ration to the number of processing nodes to interconnect.

When the number of processing nodes exceeds few tens, one rather uses intermediate architectures between these two extremes, with less (respectively more) contention, number of interconnection links, and bandwidth per processing node than that of the point-to-point links-based (respectively bus-based) architecture. For example, crossbar-based architecture requires $2N$ interconnection links, whereas multistage-based architecture requires $NlogN$ interconnection links.

In the intermediate architectures, a connection has to be established between either two processing nodes in the case of distributed memory parallel computers, or a processing node and a memory bank in the case of shared memory parallel computers before any corresponding communication begins.

If the information transfer mode over the connections is baseband (respectively synchronous), then the entire (respectively a part of the) bandwidth of each links crossed by a given connection, is dedicated to that connection for its entire life duration.

In both transfer modes, there is a considerable wastage of the interconnection network bandwidth when the communications are bursty.

The Asynchronous Transfer Mode (ATM) is an emerging network protocol. It is actually proposed as standard for Broadband Integrated Services Digital Networks (B-ISDN).

The bandwidth range, ranking from few Mbits/sec (Mega-bits per seconde) up to hundreds of Mbits/sec or even few Gbits/sec (Giga-bits per seconde), that provides ATM allows to build broadband transmission systems.

The virtual connection concept and packet routing mode used by ATM allow several connections to share a common interconnection link, in the time and space.

The asynchronous transfer mode current in ATM allows to partly solve the issue of bursty connections wasting their dedicated network bandwidth.

The use of queues and appropriate queue selection disciplines allow ATM to provide an elegant solution to the issue of contentions in the networks.

These are some of the reasons that motivated our research group to pay a particular attention in the design of the aforementioned intermediate in-

terconnection network architectures based on ATM when studying parallel computers.

In our approach, we want to quickly test for correctness and compare different architecture alternatives without the delay and expense of an hardware prototype.

Conventional Very Large Scale Integrated (VLSI) circuits design tools, such as Very high speed integrated circuit Hardware Design Language (VHDL) tools, VERILOG, COMPASS, CADENCE, etc., enable to meet this objective.

However, these tools allow the use of only workloads that are statically defined, that is at the beginning of the simulation, from either analytical models or communication traces from real application executions over parallel machines.

Now, such workloads are not appropriate for the realistic tests and evaluations of our different interconnection network implementations.

Indeed, there is a close correlation between an interconnection network and its workload. The behavior of applications has an impact over the performances of interconnection networks. And conversely, these performances have an impact over the developement of the communications of applications. This correlation is dynamic, hence extremely difficult to characterize in an analytical model of worload generation. In addition, it turns the use of the communication traces from the execution of an application over a parallel machine with a given interconnection network to synthesize the workload of a different interconnection network ineffective.

It then became of a prime necessity to design a platform, which not only obviates the need to achieve hardware prototypes, but also to simulates our interconnection network implementations directly under the workloads which are dynamically generated while running real applications on it.

In our research group, we are interested in two emerging Application Programming Interfaces (API): the Parallel Virtual Machine (PVM) developed at the Oak Ridge National Laboratory (ORNL) under the auspices of the Faculty Research Program of Oak Ridge Associated Univerties, and the Message Passing Interface (MPI) developed at the University of Tennessee Knoxville. Both, they provide C and FORTRAN libraries allowing to write fully general Multiple Instruction Multiple Data (MIMD) and more restricted Single Program Multiple Data (SPMD) apllications, and allow the use of a collection of scalar, vectorial, parallel (also multiprocessor), or even special-purpose com-

puters interconnected by one or more networks as a single computational resource.
Although MPI is right now viewed as being more mainstream than PVM, I implemented a PVM platform, simply because MPI was not released when I started my Ph.D. investigations, that is, late 1993.
Over this platform, the processes from PVM applications plug themselves in a process, which emulates the interconnection network implementation under test, and communicate each with another via it hence generating directly its workload.
Since all processes run on the same machine, the communication between the network and application processes happen via the Unix shared memory Inter-Process Communication (IPC) facilities. This implies, that the PVM applications and interconnection network implementations must be written in the C programming language.
Actually, the PVM platform is operational. I implemented at the Register Transfer Level (RTL) a first interconnection network based on ATM, hence yielding the network prototype whose architecture is presented in this dissertation.
The operation of this prototype was tested while running linear equation system parallel resolutions and matrix parallel multiplications on the platform. Its performances were evaluated on the platform while running an application in which a source process sends messages of variable sizes to a destination process, with both processes measuring the transfer delay for each message.
The platform enabled to measure the impact of the contentions in the bosom of the network prototype over the parallel resolution of a system of 4 linear equations each with 4 variables.
I did not yet lead, over the platform, comparative studies since I have implemented only one interconnection network so far.
Nevertheless, the results of the network prorotype operation tests, performances evaluation, and impact measurement indicate that the PVM platform is a promising tool for our investigations in ATM-based interconnection networks.
The remainder of this dissertation is organized as follows. Chapter 8 gives an overview of the ATM protocol. Chapter 9 describes the architecture of the network prototype. Chapter 10 first presents PVM and MPI. Then, it presents the PVM platform. Chapter 11 first presents the operation tests of the network prototype and PVM platform. Next, it discusses the per-

formances evaluation of the network prototype. Finally, it describes the measurement of the impact of the contentions within the network prototype. Chapter 12 summarizes this dissertation, and gives some perspectives.

Chapter 8

ATM Overview

ATM stands for Asynchronous Transfer Mode. It is a communication protocol, that has primarily been driven by telecommunication companies. Actually, it is proposed as the protocol standard for B-ISDN. ATM is by definition a connection-oriented network protocol. That is, a connection has to be established between two end points before any data transfer begins, and released when both end points are done.
ATM connections carry data into 53-byte packets called *cells*. As depicted in Figure 8.1, an ATM cell consists of a 5-byte header and a 48-byte payload.

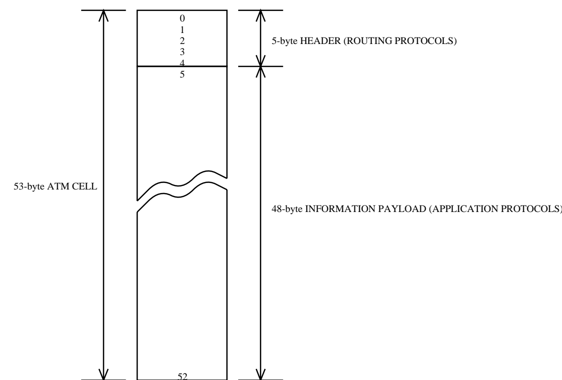


Figure 8.1: ATM cell Structure

The header of an ATM cell contains information used to route and manage that cell through the network all along its way to the destination, whereas

its payload contains the data of a message.
ATM connections are virtual. Indeed, there is no network physical resources exclusive dedication to ATM connections. Cells are mapped into the payload of the transmission link frames without any absolute position reference. Hence, over a given transmission link, all the cells belonging to a same connection carry in their headers a same logical channel number. This logical channel number, also called virtual connection identifier, uniquely and locally identifies that connection over that link.
What makes ATM extremely attractive, is its ability to shunt cells at very top speeds, and to carry in an integrated way, both real time traffic, such as voice and high resolution video, which can tolerate cell losses but not any delay, and non real time traffic, such as computer data and file transfer, which can tolerate a delay but not any cell loss, at different rates ranking from few Mega-bits up to hundred of Mega-bits or evan few Giga-bits per second (e.g. OC-3=155.52 Mbits/sec , OC-12=622.08 Mbits/sec), and under different loads.
The problem when carrying these different traffics on the same medium in an integrated way, is that the peak bandwidth requirement of these traffic sources may be quite high (as in high resolution full motion video), but the duration of the effective data transmission may be quite small. This results into a considerable wastage of the network bandwidth.
ATM attempts to solve this problem by statistically multiplexing several connections over the same link. That is, if a large number of connections are very bursty (i.e. $\frac{peak\ rate}{average\ rate} \geq 10$) [2], and even if their aggregate bandwidth requirement exceeds the capacity of a given link, they may all be assigned that link hoping statistically that, they will not all burst at the same time.

8.1 ATM Network Architectures

ATM networks are structured into layers and planes. The protocol reference model which defines this structure, is presented in Figure 8.2.
This model identifies 3 planes:

- the Management Plane used for the management of the network,
- the Control Plane used for the establishment and release of connections,

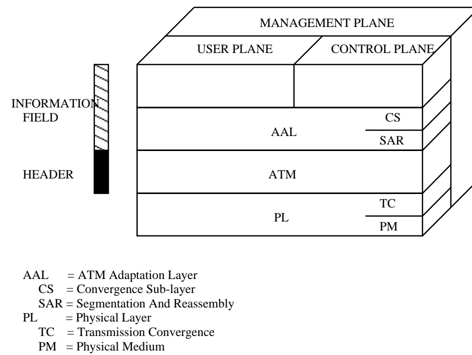


Figure 8.2: Protocol Reference Model

- the User Plane used for the data transfer,

and 3 layers:

- the Physical Layer (PL) layered directly above the physical medium,
- the ATM layer (ATM) layered directly above the physical layer,
- the ATM Adaptation Layer (AAL) layered directly above the ATM layer.

Figure 8.3 describes the protocols stack within an ATM-based communication model. The way the PL, ATM, and AAL layers are implemented, may result into the definition of a number of ATM network architectures.

8.2 The Underlying Transmission System

The transmission links used in ATM networks are organized into recurrent frames repeating with period T, whose value 125μs is derived from sampling the traditional 4KHz analog voice signal over phone lines at twice its frequency, that is 8KHz.
According to the size and structure of the frames over the transmission links, one may identify the following different types of transmission system:

- framed/SDH (Synchronous Digital Hierarchy),

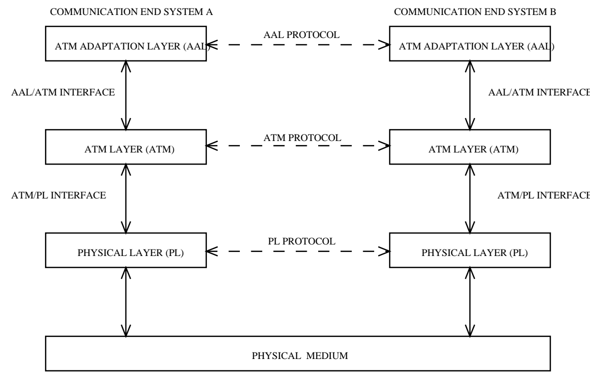


Figure 8.3: Layers Model of ATM Networks

- framed/PDH (Plesiochronous Digital Hierarchy),
- cell-based,
- asynchronous,

8.2.1 Framed/SDH Transmission System

Figure 8.4 described the structure of the STM-1 frames used over the links of an SDH transmission system.

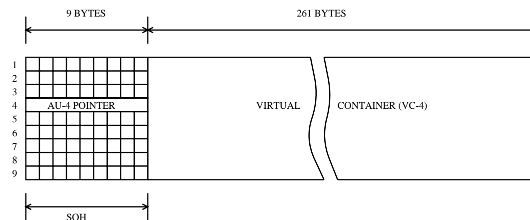


Figure 8.4: SDH STM-1 Frame

It appears that an STM-1 frame provides a capacity of 2430 bytes per 125μs (i.e. 155.52 Mbits/sec). These 2430 bytes are organized into 9 rows and 270 columns. The bytes are transmitted row by row.
The 9 first columns (i.e. 81 bytes) do not transport cells, but rather include

information used to delineate and handle the transmission frame.
The remaining 2349 bytes make up the Virtual Container of a degree 4 (VC-4). As shown in Figure 8.5, the VC-4 consists of one column of bytes (i.e. 9 bytes) including the POH (Path OverHead), and the container itself which provides a transmission capacity of 2340 bytes each 125μs (149.76 Mbits/sec). The POH consists of path management information. For example, the C2

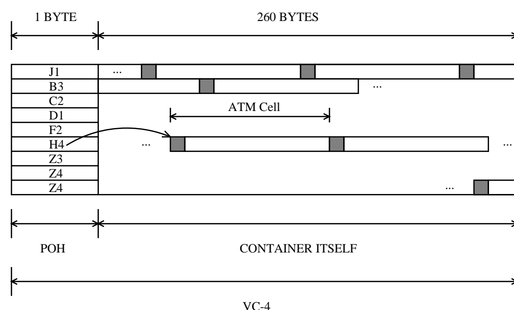


Figure 8.5: Virtual Container of a Degree 4 (VC-4)

byte specifies the type of the transmission frame payload (e.g. ATM), and the H4 byte is used to determine the offset between itself and the next cell boundary. H4 is used to verify the synchronization of cells into the frames, and not to establish it. The size of the container itself (2340 bytes), is not an integer multiple of an ATM cell size (i.e. 53 bytes). In addition to that, the digital flow generated by the PL layer is mapped continuously into the VC-4. Therefore, cell boundaries do not have fixed positions into the transmission frames. Hence, few cells may extend to the next frame.

8.2.2 Framed/PDH Transmission System

PDH frames do not have any particular structure. Two protocols may be used to map ATM cells into PDH frames.
The first one is the PLCP (Physical Layer Convergence Protocol). It adds a PLCP overhead of 4 bytes size to each ATM cell mapped into PDH frames (See Figure 8.6). From left to right in the PLCP overhead, the first two bytes contain a framing pattern, the third byte indicates the position of the current cell into the frame, and the last byte is used for management functions similar to the POH overhead in an SDH frame. With the PLCP mapping, up to 12

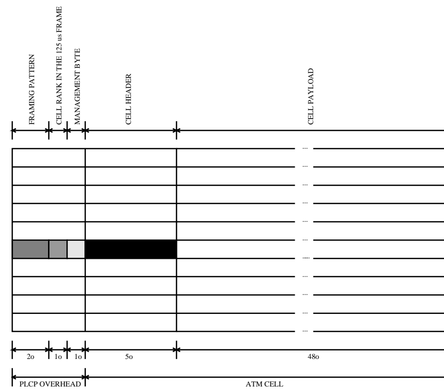


Figure 8.6: PLCP mapping into PDH Frames

ATM cells can be mapped at the same time into a frame of a 44736 Kbits/sec PDH link.
The second mapping protocol is the direct cell mapping. It simply maps ATM cells into PDH frames. Using this second mapping protocol, up to 13 ATM cells can be mapped simultaneously into a frame of a 44736 Kbits/sec.

8.2.3 Cell-based Transmission System

The frames used in this transmission system type match exactly ATM cells.

8.2.4 Asynchronous Transmission System

Asynchronous transmission systems are really unframed. Two techniques describe how ATM cells are transported in the payload of such transmission systems.
The first technique provides ATM transmission at 100 Mbits/sec, using a 4B/5B encoding and the physical supports developed for FDDI networks. In this first technique, a "TT" pair of symbols announces the beginning of an ATM cell, and the space between two consecutive cells is padded (justification function) with a variable number of "JK" pair of symbols (See Figure 8.7).
A Physical Layer Interface based on this first technique is often called: TAXI (Transparent Asynchronous Xmitter-receiver Interface). TAXI is the name given to Integrated Circuits developed for the implementation of FDDI phys-

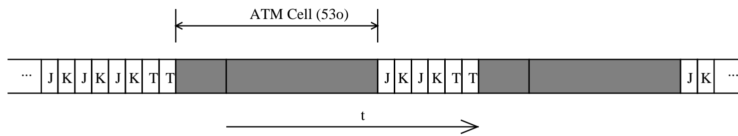


Figure 8.7: Asynchronous Transmission Link

ical layer functions.
The second technique deals with ATM transmission at 155.52 Mbits/sec. It uses an 8B/10B encoding and the physical supports developed using the FCS (Fiber Channel Standard). This last technique uses blocks of 27 cells size. The first cell of each block is used as a block separator. It is identified by means of its particular content.

8.3 The Physical Layer (PL)

After having chose an underlying transmission system, one may now implement an appropriate Physical layer. This layer provides a cells transport service between two entities of the ATM layer. Basically, the flow of cells generated by the ATM layer, can be transported in the payload of different digital transmission systems. However, in the ITU recommendations, the PL must provide the same type of service, including:

- the cells transmission,
- the delivery of each busy cell with a valid header,
- the delivery of timing information, for higher layer services such as Circuit Emulation,

to the ATM layer regardless of the underlying transmission system. This is achieved by further dividing the PL (See Figure 8.8), into two distinct sublayers:

- the Physical Media (PM) sublayer, layered directly above the physical medium,
- and the Transmission Convergence (TC) sublayer, layered directly above the TC sublayer.

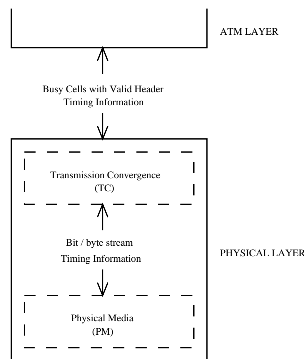


Figure 8.8: Flow of information between TC and PM

8.3.1 The Physical Media (PM) Sublayer

The PM sublayer consists of concrete transmission functions. It provides bit transmission and physical access to the media. Figure 8.9 lists few physical access rates. The STM-1 equivalent to the SONET OC-3 (155.52 Mbits/sec),

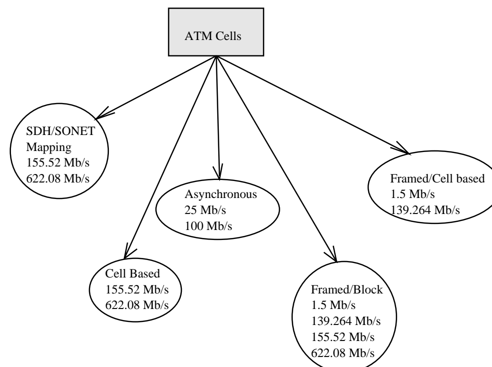


Figure 8.9: Few Digital Transmission systems used by the PL

and the STM-3 equivalent to the SONET OC-12 (622.08 Mbits/sec) have been designated as the customer access rates in B-ISDN.

8.3.2 The Transmission Convergence (TC) Sublayer

The TC sublayer is made up of the following major functions:

- the rate adaptation, which is in charge to match the rate of the flow of cells from the ATM layer, by padding it if necessary, and the rate of the underlying transmission system. Note that, padding information are not passed to the ATM layer at the receiving end.

- the HEC (Header Error Control), which is an error corrector code using the fifth byte of ATM cells, to correct all single and only detect all double errors in their headers.

- the cells delineation, which is used to determine cells' boundaries within the bit or byte stream from the PM sublayer at the receiving end.

- the mapping of the cell flow generated by the TC sublayer into the transmission system capacity.

Except for the HEC, the implementation of all these functions depends on the type of the underlying transmission system. Indeed, the way to pad, delineate, and map cells will be different according to whether the transmission links are SDH-based, PDH-based, cell-based, or asynchronous. For example, idle cells are used when relying on cell-based links, whereas idle symbols are used when rather relying on asynchronous links, for padding purpose.

8.4 The ATM layer (ATM)

The major role of the ATM layer is to shunt cells between entities of the AAL layer.
Switches are the basic operators of this layer. They only perform the header of cells, since the payload of cells is transparent to the ATM layer.
Figure 8.10 shows the formats of ATM cells' header at the User-Network Interface (UNI) and Network-Node Interface (NNI).
The way ATM switches use the Generic Flow Control (GFC) field is not fully specified. However, GFCs are meant to be used at the UNI and not at the NNI.
ATM switches use the Virtual Path Identifier (VPI) and Virtual Channel Identifier (VCI) fields for the cells routing through the network. The concatenation of these two fields allows the identification of the Virtual Channel Connection (VCC) to which a cell belongs to, whereas the VPI field alone

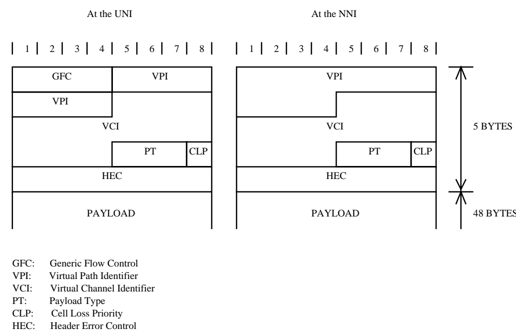


Figure 8.10: ATM Cell Header Formats

allows the identification of the Virtual Path Connection (VPC) to which a cell belongs to.
Note that, a VPC is used in order to switch a bundle of VCCs together as a singlt unit.
ATM switches use the Payload Type (PT) field to discriminate the cells carrying user data and those used for the maintenance or resource management within the network, in accordance with Table 8.1.

Payload Type	flow type	congestion	data unit type
000	**0** User Data	**0** not encoutered	**0**
001	**0** User Data	**0** not encoutered	**1**
010	**0** User Data	**1** encoutered	**0**
010	**0** User Data	**1** encoutered	**1**
		usage type	
100	**1** Network	**00** hop-by-hop maintenance	
101	**1** Network	**01** end-to-end maintenance	
110	**1** Network	**10** resource management	
111	**1** Network	**11** reserved	

Table 8.1: Payload Type Field Encoding

ATM switches use the Cell Loss Priority (CLP) field to determine the cells to be discarded in priority when congestions occur in their queues.
The Header Error Control (HEC) field is used by the Physical layer to detect and/or recover errors over ATM cell headers (See Section 8.3.2).
Figure 8.11 illustrates the basic operation of ATM switches.
In this figure, all cells with an header y on the inlet I_n are switched to the outlet O_2 with an header j, for example.
In addition, it is possible that two cells on different inlets (e.g. I_1 and I_n),

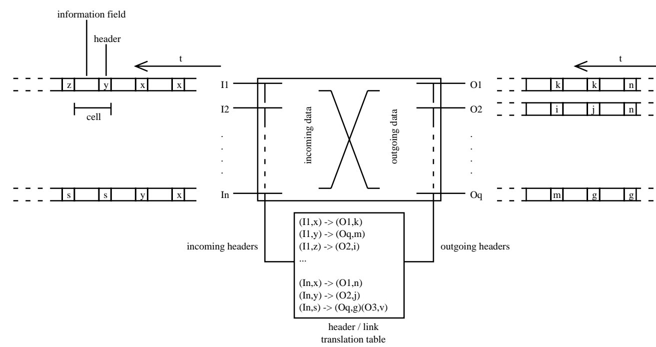


Figure 8.11: ATM Switch basic Operation

arrive simultaneously at the switch in destination to the same outlet (O_1). Only one of these cells will be put on that outlet. Therefore, somewhere in the switch, a queue has to be provided in order to buffer the other.
Upon this illustration, it appears that ATM switches perform the following three basic functions:

- header translation (also label switching)
- routing (also space switching)
- queueing

Since the queues provided in ATM switches are not of an infinite size, it may happen that the number of cells to be buffered into a particular queue exceeds the maximum of that this queue can sustain at a time.
It is the network congestion control which is in charge to deal with such a situation. The following strategies define the way a congestion control could be performed within a network [27]:

- Preallocation of resources to avoid congestion
- Allowing ATM switches to discard cells at will
- Using flow control
- Choking off input when congestion occurs

Hence, it appears that ATM switches may be involved into the enforcement of a congestion congestion within a network.

8.4.1 The Header Translation

The header translation function is generally implemented in tables, very often referred to as routing tables in the literature. It is application relating pairs of (incoming VCI/VPI, inlet), with pairs of (outgoing VCI/VPI, outlet). If the header translation function implemented allows one incoming pair to be related to more than one outgoing pair. Thus, the resulting switches may allow point-to-point, multicast, and broadcast communications. Otherwise, they will only allow point-to-point communications. Switches translate either VCIs only, or the entire VCI/VPIs. The former are VC switches, whereas the latter are VP switches.

8.4.2 The Routing

The routing function uses the result from the header translation function to route cells towards their destination outlets, through the internal interconnects of switches. The internal interconnects used into ATM switches might be either blocking or not. A blocking interconnect, such as a bus, does not have enough resources to provide simultaneous and independent paths between any arbitrary (idle inlet, idle outlet) pair, whereas a nonblocking switch, such as a crossbar, does. The Banyan switch [9] - [18], is a blocking switch example. Examples of nonblocking switches are given in [9]. Blocking interconnects tend to cause a quicker saturation of upstream queues than the nonblocking ones.

8.4.3 The Queueing

Even when the internal interconnect used into a switch is nonblocking, queueing may be unavoidable, as in the aforementioned case. To maintain transmission sequences all along connections, the enforced selection policy in queues, is the simple FIFO (First-In-First-Out) discipline. The location where the queueing function is performed in the switch, discriminates the internal architecture of one switch from another. According to whether the switch internally operates at the same speed as its input and output links, or not, queuing may be performed at the input, output, or even center of switches.

In a switch running N (number of input and output links) times as fast as

its input and output links, all cells that arrive during a particular time slot, can traverse the switch before the next time slot. But queueing will still be required at the switch output (See Figure 8.12), due to the simultaneous arrival of more than one cell intended to the same output link. In a switch

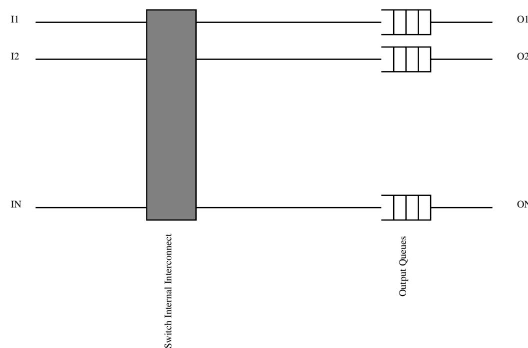


Figure 8.12: Switch with Output Queues

that runs at the same speed as its input and output links, only one cell can cross the switch over within each time slot. Hence, when several cells arrive simultaneously at the switch input, only one will be served, and the remaining ones, even if few of them are intended to different output links, must be queued at the switch input (See Figure 8.13) for subsequent admission. The arbitration logic, which decides which inlet has to be served, can range

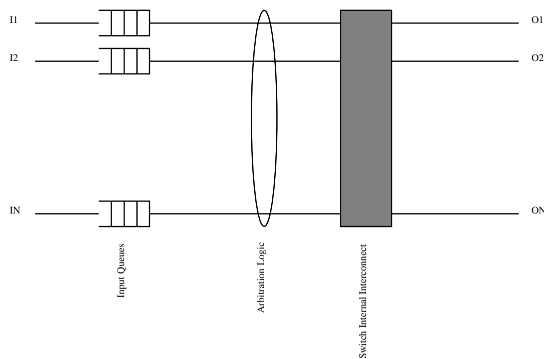


Figure 8.13: Switch with Input Queues

from a very simple round-robin up to a quite complex mechanism taking into

account the input queues' filling levels. As shown in [19], the mean queue length, and hence the mean waiting time, will be greater for input queueing than for output queueing. The input queueing suffers from the so-called HOL (Head Of the Line) blocking. Indeed, a cell that could traverse the switch to an idle output link during the current time slot, may have to wait in queue behind a cell whose destination output link is busy in the meantime. It is also shown that output queues saturate as the input links utilization approaches 1. Input queues on the other hand, rather saturate at an input links utilization, depending on N (number of input and output links), but approximatively equal to $(2 - \sqrt{2}) = 0.586$ when N is large. Hence, it is clear that a better performance is achieved when performing queueing at the output of switches. However, if we consider the number of places to be provided into queues as an issue, their total number is greater into the output queueing solution than into the input queueing solution, providing that queues have to be dimensioned for the worst case [1]. This total number can be reduced by providing a single central queue, which is shared between all inlets and outlets (See Figure 8.14). In this case, each incoming cell will directly be

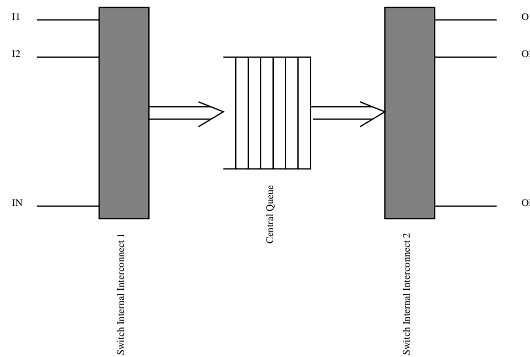


Figure 8.14: Switch with a Central Queue

stored into the central queue, and every outlet will select cells destined to it into the central queue, using a FIFO discipline. Since the central queue may be addressed in a random way, due to the merging of cells into it, a rather complex memory management system has to be provided. Nevertheless, this approach achieves the same performance as the classical output queueing approach. In the multiple queueing approach, each input queue is split into N

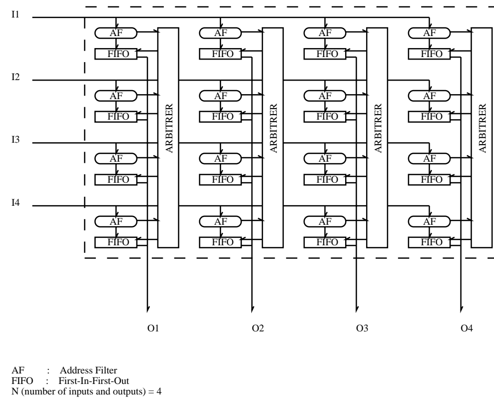


Figure 8.15: Switch with Multiple Queues

separate queues, one for each possible output link (See Figure 8.15). When a cell arrives on an input link, it is routed to a queue associated to that cell destination output link. Note that, cells buffering occurs after the routing operation. Thus, this approach is basically an output queueing one. It is implemented in the BMX (also BMS: Bus Matrix Switch) architecture presented in [21]. This solution achieves the same performance as the classical output queueing approach without resorting to a faster switch.

8.4.4 The Interconnection Topology

Once the internal architecture of switches are already defined, the way to interconnect these switches sets up the topology of the network. Several network topologies are possible. Figure 8.16 shows some of them, that are often referenced in the literature. Topologies a), in which switches are wired into a straight line, and b), in which switches are wired into a ring, are of one dimension. In such topologies, the routing is extremely simple, and the required number of switches linearly increases as the number of end-systems to interconnect. The drawback of these type of topologies is that, they are blocking. The topology c), in which switches are wired into a grid, allows to build a crossbar. It is a two dimensional topology, which is nonblocking. However, its required number of switches quadratically increases as the number of end-systems to interconnect. The multistage topology illustrated in d), where switches are organized into stages, is an approach intended to re-

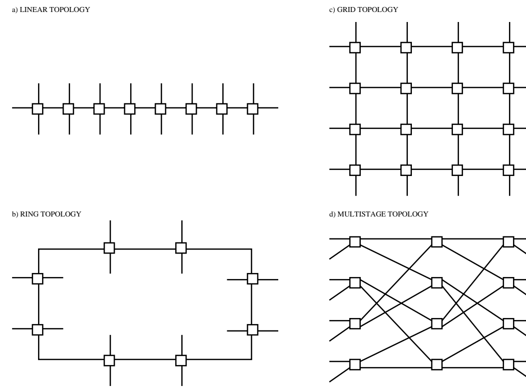


Figure 8.16: Network Topologies

duce the number of switches required into crossbars. But, the way stages are structured, may result into a multistage network having one of the following characteristics:

- nonblocking. That is, there is always a path between an idle input port and an idle output port.
- rearrangeable. That is, there is always a path between an idle input port and an idle output port, even if it entails to divert few already existing paths.
- blocking. That is, there might be some cases where, it will absolutely not be possible to find a path between an idle input port and an idle output port.

The possible different structures of particular three stages networks, named Clos networks, and often referenced as Clos(e,m,r), quite illustrate the aforementioned multistage networks characteristics (See Figure 8.17). Clos(e,m,r) networks have the $[S_{em}, S_{rr}, S_{me}]$ structure. That is, a Clos(e,m,r) network have r S_{em} switches on its input stage, m S_{rr} switches on its middle stage, and r S_{me} switches on its output stage, providing that an S_{ij} switch denotes an internally nonblocking switch, with i input and j output ports.
In a Clos(e,m,r) network, the output stage is the symmetrical of the input stage.

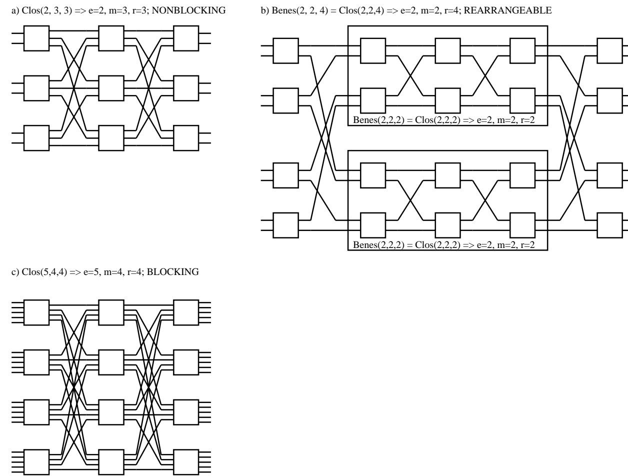


Figure 8.17: 3-Stage Networks

Moreover, the output port number i of the input stage switch number k is connected to the input port number k of the middle stage switch number i. And symmetrically, the output port number k of the middle stage switch number i is connected to the input port number i of the output stage switch number k.
Further more, Clos(e,m,r) networks comply with the following statements:

- If m≥2e-1, then resulting Clos(e,m,r) networks are nonblocking.
- Else, if m≥e, then resulting Clos(e,m,r) networks are rearrangeable. Benes networks are examples of rearrangeable networks. They are Clos(2,2,2^k) networks, recursively built each from two Clos(2,2,2^{k-1}) networks, where $k \geq 2$.
- Otherwise, resulting Clos(e,m,r) networks are blocking.

Switches can also be wired into topologies of dimension greater than or equal to three. Figure 8.18 presents one cube in a) as an example of topology of dimension three, and one hypercube (here a 4-cube) as an example of topology of dimension four. These spatial topologies provide a greater bandwidth. However, the routing in those, is further complex.

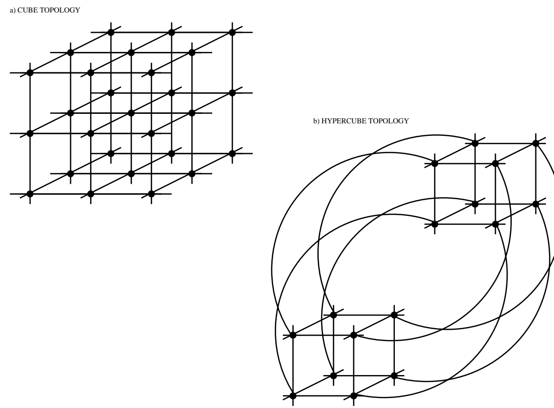


Figure 8.18: Spatial Topologies

8.5 The ATM Adaptation Layer (AAL)

8.5.1 The AAL Services

The objective of the AAL layer is not only to map flow of information generated by applications into the payload of cells, but also to provide, above the ATM layer, a service suitable to the constraints of applications.
Here, the issue on one hand, is the unification of all possible services into a single AAL.
Indeed, it is not only difficult to predict the evolution of applications, but also their communication requirements are extremely various and sometimes not compatible.
On the other hand, the dedication of a different service to each type of application, is not reasonable.
Hence, the tradeoff solution results in grouping the services required by applications, into classes according to the combination of the three following metrics, that may characterize any communication flow generated by applications:

- bit rate (constant or variable)
- connection mode (connection-oriented or connectionless)
- source and destination synchronization (with or without)

Figure 8.19 shows the four classes of services that have been thus defined so far. To provide these services, the CCITT has specify five types of AAL [24]:

- AAL Type 1: provides constant bit rate services, such as traditional voice transmission and circuit emulation
- AAL Type 2: transports variable bit rate video and audio information, and keeps a timing synchronization between source and destination
- AAL Type 3: supports connection-oriented data services and signaling
- AAL Type 4: supports connectionless data services (now combined with AAL Type 3 to make up the AAL Type 3/4)
- AAL Type 5: provides a simple and efficient AAL that can be used for bridged and routed Protocol Data Units (PDU)

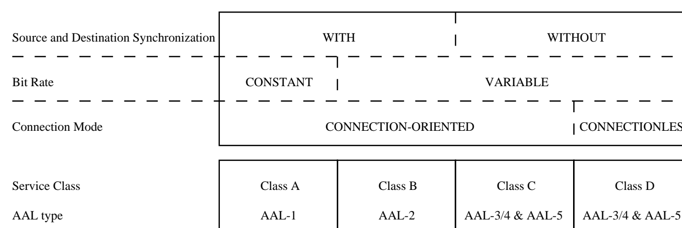


Figure 8.19: AAL Service Classes

All types of AAL are further divided into two sublayers:

- the Segmentation And Reassembly (SAR) sublayer, layered directly above the ATM layer
- the Convergence Sublayer (CS), layered directly above the SAR sublayer. For the AAL 3/4, this sublayer is divided into two sublayers:
  1. the Common Part Convergence Sublayer (CPCS), layered directly above the SAR sublayer
  2. the Service Specific Convergence Sublayer (SSCS), layered directly above the CPCS sublayer

8.5.2 The AAL Data Units

Figure 8.20 shows the transformations undergone by data units from applications to ATM cells through the ATM adaptation layer. At the transmitting

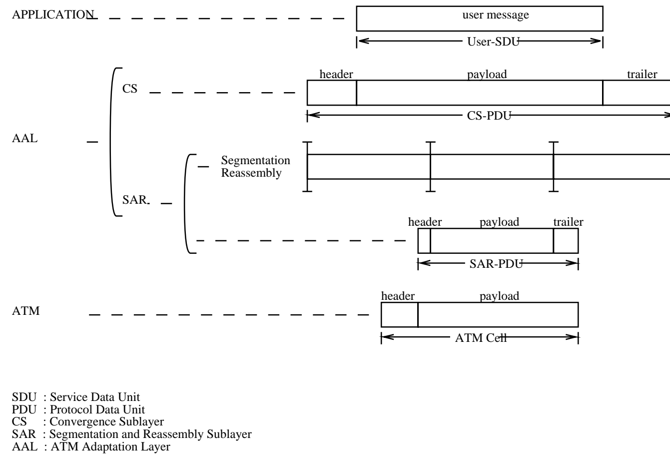
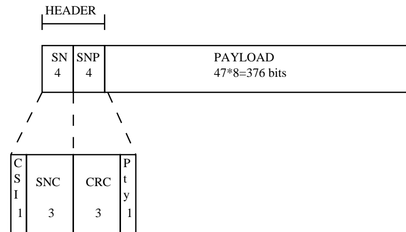


Figure 8.20: Data Units: from Applications to ATM Cells via the AAL

end, the CS sublayer receives a User-SDU, containing a user message. Then, it constructs a CS-PDU by adding an header and a trailer to this received User-SDU, which hence makes up the CS-PDU payload. The resulting CS-PDU is passed to the SAR sublayer, which in turn splits it, if necessary, into several SAR-PDU payloads, and then completes to construct the related SAR-PDUs that fit into ATM cell payloads, by adjunction of headers and trailers. Resulting SAR-PDUs are thereafter passed to the ATM layer, which finally generates the corresponding cells by adding an header to each received SAR-PDU (which hence constitutes a cell payload). Reverse operations are performed at the receiving end. The CS-PDU (respectively SAR-PDU) headers and trailers contain information that characterize the CS (respectively SAR) sublayers' functions. Their size (See Table 8.2) and format, differ from one type of AAL to another. Figure 8.21 describes the SAR-PDUs' header format of an AAL of type 1. Figure 8.22 describes the SAR-PDUs' header and trailer format of an AAL of type 2. Figure 8.23 describes the SAR-PDUs' header and trailer format of an AAL of type 3/4. Figure 8.24 describes the CPCS-PDUs' header and trailer format of an AAL

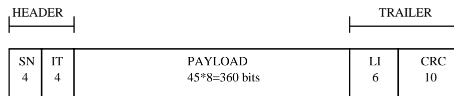
AAL Type	CS-PDU Header (size in bytes)	CS-PDU Trailer (size in bytes)	SAR-PDU Header (size in bytes)	SAR-PDU Trailer (size in bytes)
AAL-1	0	0	1	0
AAL-2	0	0	1	2
AAL-3/4	2	2	2	2
AAL-5	0	8 - 47	0	0

Table 8.2: AAL Data Unit Header and Trailer Sizes



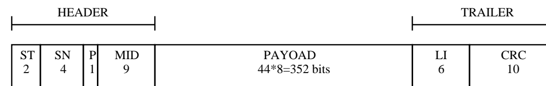
SN : Sequence Number (used to detect missing and inserted cells)
SNP : Sequence Number Protection (used to detect errors in the SN portion of the cell's payload)
CSI : Convergence Sublayer Information ; May carry a Residual Time Stamp (RTS), or a Block Pointer Indicator
SNC : Sequence Number Code (the sequence number itself)
CRC : Cyclic Redundancy Code
Pty : Parity bit

Figure 8.21: AAL-1 SAR-PDU Format



SN : Sequence Nmber (used to detect missing and inserted cells)
IT : Information Type (Beginning, or Middle or; End Cell)
LI : Length Indicator (Indicates the amount of bytes used in the cell's payload)
CRC : Cyclic Redundancy Code (used to detect errors in the cell's payload)

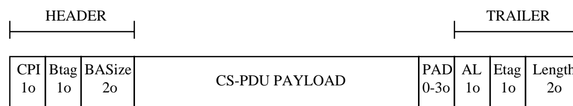
Figure 8.22: AAL-2 SAR-PDU Format



ST : Segment Type (Beginning, or Middle, or End Cell)
SN : Sequence Number (used to detect missing and inserted cells)
P : Priority (indicates the either loss or service priority of the cell)
MID : Multiplexing IDentification (used to discriminate cells from different flows on the same virtual connection)
LI : Length Indicator (indicates the amount of bytes used in the cell's payload)
CRC : Cyclic Redundancy Code (used to detect errors in the cell's payload)

Figure 8.23: AAL-3/4 SAR-PDU Format

of type 3/4. Figure 8.25 describes the format of CS-PDUs and SAR-PDUs



CPI : Common Part Indicator (identifies a specific CS protocol, if any)
Btag : Begin Tag (used to detect the spanning of several CS-PDUs, due to losses or insertions)
BASize : Buffer Allocation Size (assists the receiving AAL in memory allocations)
PAD : Padding (added to let the size of the CS-PDU payload be a multiple integer of 4 bytes)
AL : Alignment (a dummy byte to let the size of the CS-PDU trailer be equal to 4 bytes)
Etag : End Tag (used to detect the spanning of several CS-PDUs, de to losses or insertions)
Length : Length of the CS-PDU (indicates the size of the meaningful part of the CS-PDU payload)

Figure 8.24: AAL-3/4 CPCS-PDU Format

of an AAL of type 5.

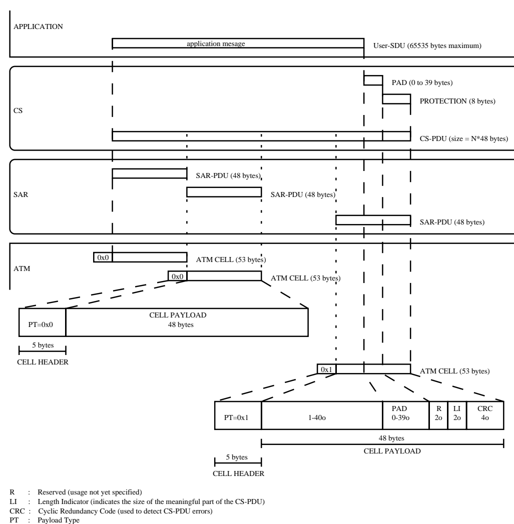


Figure 8.25: AAL-5 CS-PDU and SAR-PDU Format

8.5.3 The AAL Functions

Table 8.3 summarizes the functions performed by each type of AAL.

Type of AAL	1	2	3/4	5
SAR Functions				
Segmentation and Reassembly	x	x	x	x
Block Data Transfer	x			
Cell Errors Processing	x	x	x	
Cells Loss and Insertion Processing	x	x	x	
Partly Filled Cells Processing		x	x	
CS' Functions				
Time/Clock Recovery	x	x		
Delay Equalization (Jitter Compensation)	x			
Flow Control			x	
Memory Space Indication			x	
Message Padding			x	x
Message Spanning Processing			x	
Message Framing		x	x	
Message Multiplexing and Demultiplexing			x	
Message Errors Detection				x

Table 8.3: AAL Functions

8.5.3.1 Cells Insertion, Loss, and Errors Processing

Basically, inserted cells are simply discarded. Losses and errors recovery often deals with acknowledge receipts, missing and corrupted cells retransmission. However, this approach is not suitable anymore when providing real-time services, since it does not guarantee any bounded transfer delay. In this case, the AAL of destination may mask losses and errors by interpolating the received information. The interpolation of information yields a better recovery when combined with a message interleaving at the source. Nevertheless, the masking method does not fully satisfy applications such as video or high quality audio, for which it is extremely important to recover 100% of the original information. Hence, the Forward Error Correction (FEC) is performed, eventually combined with a source interleaving mechanism.

8.5.3.2 Delay Equalization

The delay equalization, is of crutial importance for continuous data flow applications (e.g. high quality audio, circuit emulation), which are extremely sensitive to the delay variation (also delay jitter). Receiving CS sublayers basically perform it, by storing received cells, at least momentarily for an amount of time equal to the propagation delay added to the maximum delay jitter, into a playout buffer.

8.5.3.3 Time/Clock Recovery

With an asynchronous type of transfer (it is the case with ATM), it is not possible to synchronize both source and destination clocks. Therefore, the CS sublayer at the receiving end, has to recover the source clock from the flow of incoming cells which carry a Residual Time Stamp (RTS), to provide a time/clock recovery service to applications. But for applications such as voice and video, the use of this RTS (also SRTS: Synchronous RTS) method is in general not necessary. The receiving CS sublayer may retrieve a rather rough source clock by means of the filling level of its playout buffer. If that filling level increases or decreases, then the cells delivery will speed up or slow down respectively.

8.5.3.4 Flow Control

As explained in [27], the ATM adaptation layer or even higher layers must provide a solution to deal with the situation where a sender systematically wants to transmit cells faster than the receiver accept them.
This situation can easily happen when the sender is running on a fast or lightly loaded machine, whereas the receiver is running on a slow or heavily loaded machine. Therefore, if the sender keeps pumping cells out at a high rate until the receiver is totally swamped, at a certain point the latter will simply not be able to handle cells as they arrive, and start to lose some, even if the transmission is error-free.
The usual strategy used to prevent such a situation is to introduce a flow control to throttle the sender into sending no faster than the receiver can handle the traffic.
Basically, the different flow control mechanisms either have the destination UNI sends a feedback message to the source UNI, in order for the former to let the latter knows whether or not it is able to keep up, or permit the destination UNI to simply discard violating cells at will with impunity.

8.5.4 Connectionless Data Services

ATM is by definition connection-oriented. That is, communication over ATM proceeds through three well defined phases:

- connection establishment,

- data transfer,
- connection release.

The connection establishment and release phases are performed by the signaling protocol in the control plane, while the data transfer phase is managed by protocols in the user plane (See Figure 8.2). Within the connection establishment phase, the user sends a connection request along with a set of traffic parameters, such as:

- Peak Cell Rate (PCR)
- Sustainable Cell Rate (SCR)/Allowed Cell Rate (ACR), often referred to as mean cell rate
- Minimum Cell Rate (MCR)
- Burst Tolerance (BT)

and Quality of Service (QoS) requirements of:

- Cell Loss Rate (CLR)
- Cell Transfer Delay (CTD)
- Cell Delay Variation (CDV), often referenced to as Two Point CDV. It defines the allowed maximum variation of the CTD incurred by cells when crossing the network (See Figure 8.26)
- Cell Delay Variation Tolerance (CDVT), often referred to as One Point CDV. It indicates the cells inter-arrival time maximum variation, sustained at the network access (See Figure 8.26)

to the network. The network uses the Connection Admission Control (CAC) function to decides whether a connection is accepted or not, at a given node. If a connection is accepted, then it computes a route that can best sustain the requested QoS requirements and bandwidh allocation. If such a route is found out, then the network sends a connection setup request to all the nodes on the computed route, and acknowledges the originating user, which may then proceeds with the transfer of data. The connection establishment phase is extremely time expensive, and the monitoring of leased lines is quite

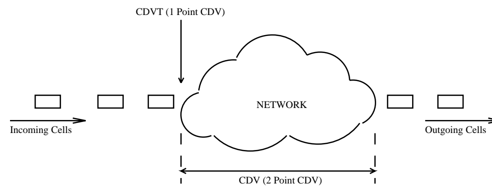


Figure 8.26: CDV and CDVT

complex.
For these reasons, the provision of a connectionless service, where communications take place without any previous connection setup, over ATM is more economic.
However, the use of *datagrams* is not possible in ATM. This is mainly because the small size of cells prevents to put the full destination address in every cell of a message [3, 4].
In addition, ATM is by definition connection-oriented. Therefore, the only way to avoid to incur overheads due to systematic estalishment and release of connections and/or the management of leased connections, is to let communications proceed through pre-established unleased connections.
The drawback when providing connectionless services is that, ATM cannot any longer guaranty any limited transfer delay, and hence Quality of Service (QoS) to the communications that use these services.

8.6 Traffic Control

The traffic control is a key issue for all networks, such as ATM networks, that intend to guarantee negotiated QoS to leased connections. Such networks use a rate based flow control to monitor leased connections at the network accesses, for possible traffic contract violation.

8.6.1 Source Policing

The aforementioned networks police misbehaving connections, by dropping or marking with low priority for loss, violating cells. Several network access policing (also UPC: User Policing Control at UNI, and NPC: Network Policing Control at NNI) mechanisms are under studies [25, 26]. The GCRA is

actually used as the standard algorithm for policing. It is defined as follows [7]:

GCRA(T, τ)
input t : **arrival time**;
output result: **binary decision**;
internal tat : **theoritical arrival time**;

if (t<tat-τ)
    result = NON-CONFORMING;
else
{   tat = max(t, tat) + T;
    result = CONFORMING;
}

Figure 8.27 shows two equivalent forms of the GCRA algorithm. The values

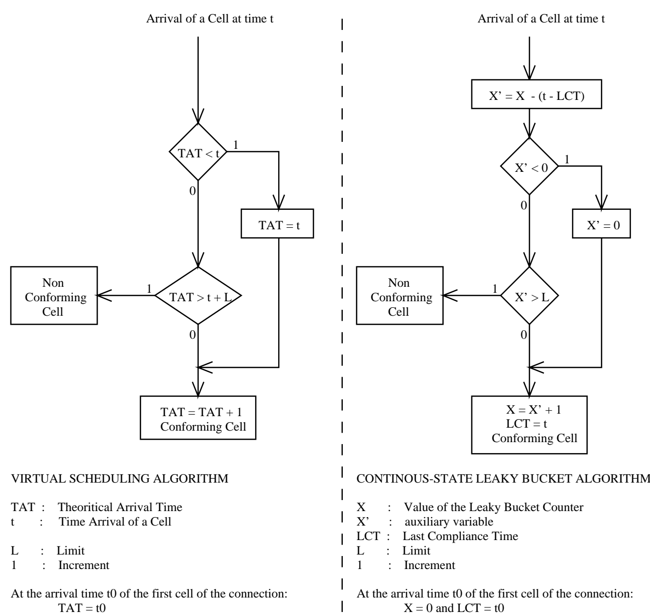


Figure 8.27: Equivalent Forms of GCRA

of T and τ are service-dependent, and hence connection type dependent.

Table 8.4 shows that the connection types can be rated into the two following distinct classes:

- Reserved Class (e.g. flexible circuit, Frame Relay virtual circuit). Sufficient network resources (e.g. buffers and bandwidth) have to be allocated for any connection belonging to this class. Thus, the network can guarantee a satisfactory delivery service

- Best Effort Class (e.g. traditional data). In contrast to the Reserved class, the network cannot guarantee satisfactory transmissions of information for connections belonging to this class due to the lack of network resources dedication to such connections. Nevertheless, the network will do the best effort to deliver the information at their destinations

Reserved Connection	Best Effort Connection
DBR:	
CBR (Constant Bit Rate)	
SBR:	Class X:
VBR rt (Variable Bit Rate real time)	UBR (Unspecified Bit Rate)
VBR nrt (Variable Bit Rate non real time)	Class Y:
	ABR (Available Bit Rate)
Other:	ABT/IT
ABT/DT (ATM Block Transfer/Delayed Transmission)	ABRT

Table 8.4: ATM Connection Types

8.6.2 CBR Connections

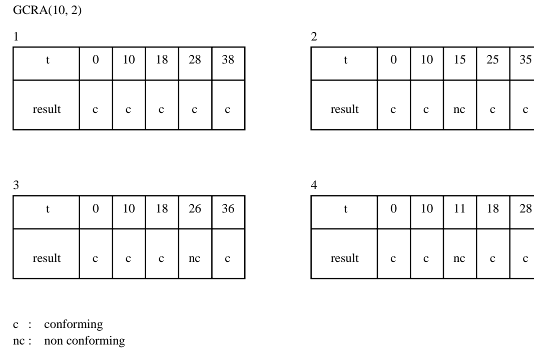
For CBR connections, which are characterized by the following parameters,

- CLR (e.g. 10^{-10})
- CTD (e.g. 100 ms)
- CDV (e.g. 10 ms)
- CDVT (e.g. 2 time slots)
- PCR (e.g. 170.2 cells/s for a 64 kb/s emulated circuit)

the expressions of T and τ are given by the following equalities:

- T=$\frac{LinkRate}{PCR}$. It can be interpreted as the minimum cell inter-arrival time
- τ=CDVT, where the CDVT is fixed by the network for every PCR

Figure 8.28 presents four GCRA(10, 2) (i.e. T=10, τ=2) execution examples. The GCRA algorithm allows the clumping of cells. That is, cells may arrive



1

t	0	10	18	28	38
result	c	c	c	c	c

2

t	0	10	15	25	35
result	c	c	nc	c	c

3

t	0	10	18	26	36
result	c	c	c	nc	c

4

t	0	10	11	18	28
result	c	c	nc	c	c

Figure 8.28: GCRA Tests for T=10, τ=2

back to back at link speed. The maximum size of allowed clumps, depends on T and τ as given by the following expression [7]:

$$\text{Clump Size} = \lfloor \frac{\tau}{T-1} + 1 \rfloor$$

8.6.3 VBR Connections

8.6.3.1 VBR rt Connections

VBR rt (real time) connections are negotiated with all the traffic parameters as the CBR connections. But the allocation of peak cell rates, as it is the case for CBR connections, results into a wastage of network resources when the source is sporadic (e.g. data sources, VBR codecs). For this reason, the following two additional traffic parameters are specified to define VBR rt connections:

- Sustainable Cell Rate (SCR) (e.g. mean cell rate = $\frac{1}{2}$ peak cell rate)

- Burst Tolerance (BT) (e.g. 500 time slots)

At any network access, cells belonging to a VBR rt connection must be consistent with both

- GCRA($\frac{linkrate}{PCR}$, CDVT)
- GCRA($\frac{linkrate}{SCR}$, BT+CDVT)

VBR rt connections are by definition bursty. In a burst, cells arrive with spacing δ at link speed (See Figure 8.29). Note that, when δ=1, one talks about cells clumping (See Section 8.6.2). The maximum size of bursts with

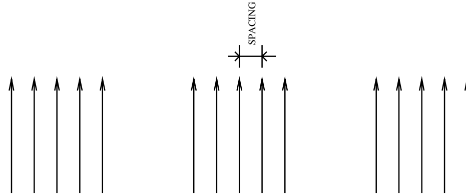


Figure 8.29: Spacing within Bursts

spacing δ is given by the expression [7]:

$$MBS_\delta = \lfloor \frac{\tau}{T-\delta} + 1 \rfloor$$

Figure 8.30 illustrates how the MBS_δ complies with the GCRA. There, a GCRA(100, 500), that is T=100 and τ=500, is considered. In the results of table 1, one can observe a burst of 9 cells with spacing δ=10 starting at time t=100. However, the maximum size of bursts allowed by the network in this case is MBS_{10}=6. Therefore, cells arrived at times t=160, t=170, and t=180, are declared being inconsistent. On the other hand, in the results of table 2, one can rather notice the occurrence of a burst of 8 cells with spacing δ=30. In this context, the maximum size of bursts allowed by the network is MBS_{30}=8. Therefore all cells in table 2 are consistent with the connection contract. Nevertheless, bursts of maximum size need to be spaced to allow a better traffic regulation through the network. Thus, assuming that T_i is the inter-arrival time of bursts of maximum size requested by the source, the expression of the maximum size of bursts allowed by the network is [7]:

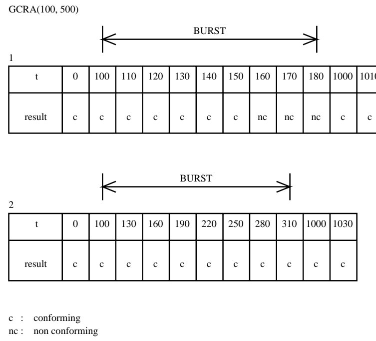


Figure 8.30: Maximum Size of Bursts with only Spacing

$$MBS_{\delta,T_i} = 1 + \lfloor \frac{min(T_i - T, \tau)}{T - \delta} \rfloor$$

Figure 8.31 is an example of how the MBS_{δ,T_i} matches with the GRCA. There, a GCRA(100, 500) is considered too. That is, T=100 and τ=500. One can also notice that the value of the spacing within any burst is δ=10. The results table suggests as value of the bursts of maximum size inter-arrival time T_i=300 (rf. first part of the results table) and T_i=200 (rf. second part of the results table). In this context, the maximum size of bursts sustained by the network is either $MBS_{10,300}$=3 or $MBS_{10,200}$=2. On the other hand, one can observe only bursts of 3 cells with spacing 10. As long as these bursts remain distant of at least 300 time slots, all cells will be consistent. But if the inter-arrival time of burst decrease down to 200 time slots, few cells might be inconsistent (e.g. the cell arrived at t=1420).

8.6.3.2 VBR nrt Connections

VBR nrt (non real time) connections are the same as VBR rt connections without the CDV specification.

8.6.4 UBR Connections

UBR connections are truly best effort connections. There is no traffic parameters specification, and hence no transmission contract between the source and the network when opening such connections. Thus, the network cannot guarantee neither any limited cell transfer delay, nor any cell loss rate due

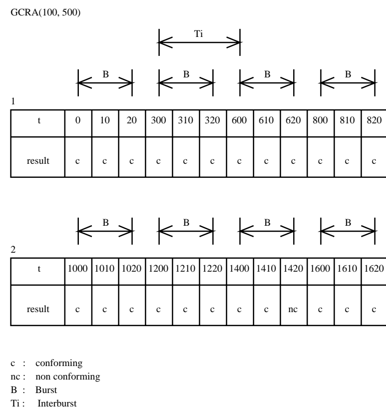


Figure 8.31: Maximum Size of Bursts with Spacing and Interburst

to network congestion for example. In short, the network cannot guarantee any throughput for these connection type.

8.6.5 ABR Connections

ABR connections are best effort connections too. But, in contrast to UBR connections, the network can guarantee a minimum throughput and a low cell loss rate for ABR connections. Nevertheless, their cell transfer delay still remains variable. The rate of any ABR connection, called ACR (Allowed Cell Rate), dynamically varies between an MCR (Minimum Cell Rate) and a PCR (Peak Cell Rate) (See Figure 8.32). The network or the destination

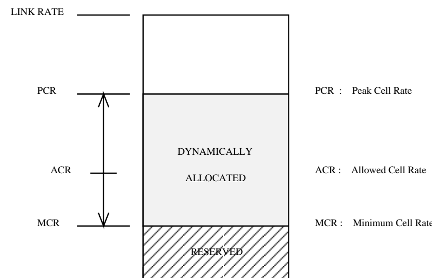


Figure 8.32: Minimum, Allowed, and Peak Cell Rate

use RM (Resource Management) cells (See Figure 8.33) or the header's PT field of ATM cells, to send flow control information, such as

- EFCI marking. The CI and NI bits are marked in forward and/or backward RM cells
- CI marking. In addition to the CI and NI bits, explicit rates (e.g. ER=75 Mbits/s) are marked in forward and/or backward RM cells; PT=$(110)_2$ in forward ATM cells
- Explicit rate marking. Only explicit rates are marked in forward and/or backward RM cells

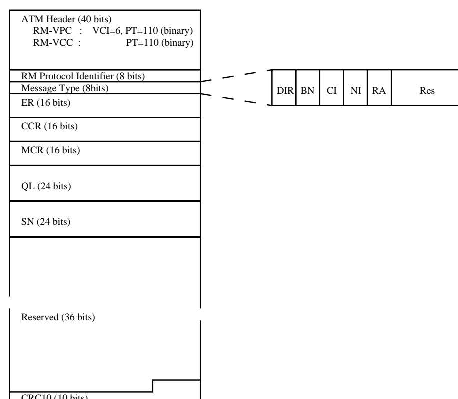


Figure 8.33: Resource Management Cell Format

to the appropriate network access, which may then either modulate the transfer rate, or notify the originating user for a drastic source quenching, depending on the severity level of the congestion condition. The transfer rate modulation may consist of exponentially reducing the current transmission rate when a congestion occurs, and linearly increasing the current transmission rate when the congestion disappears. However, the source network access must never exceed the explicit rate indicated by the network (i.e. ACR = min(ACR, ER)).

8.6.6 Source Shaping

In any case, it is noteworthy to perform the shaping of misbehaving connections prior to access the network, to prevent the deterioration of the contract booked at the connection establishment, due to the policing (also UPC or NPC) at the network access. Figure 8.34 presents a way to perform both source shaping and network access policing within an hierarchy of private and public ATM networks.

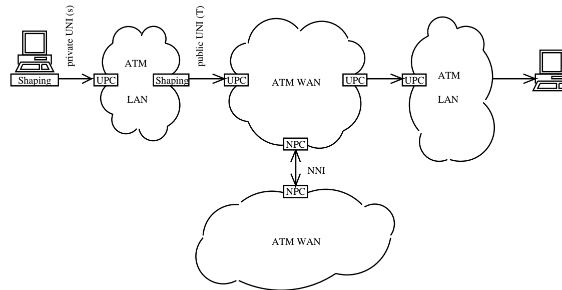


Figure 8.34: Source Shaping and Policing

8.6.7 Delay and Loss Priorities Management

The last, but not the least important function of the traffic control is the management of delay and loss priorities. Real time applications are extremely delay sensitive. Hence, their connections must undergo a preferential treatment at the network multiplexing points, to avoid higher delay priority cells to be stuck in the traffic behind lower delay priority cells. The delay priority function is based on the VPI/VCI. It hence affects the entire connection. It is implemented by using as many separate queues as the number of different delay priority levels, and first serving higher delay priority queues without starving lower delay priority queues. The loss priority function is rather CLP-based, and hence cell-based. It is intended to limit the buffer overflow effects to low loss priority cells (i.e. CLP set to 1). It is implemented as follows. Low loss priority cells are stored in a queue, only if that queue size has not yet reached a predefined threshold. The Figure 8.35 shows an example implementation of these two functions.

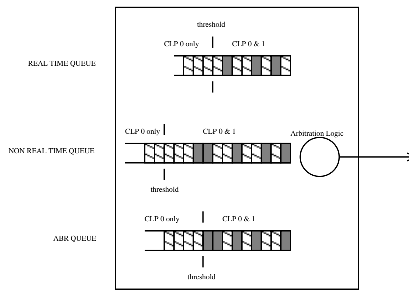


Figure 8.35: Delay and Loss Priorities VS Buffers

8.7 InterNetworking

This section introduces some of the protocols implemented above the AAL layer, to let ATM networks to communicate with other existing networks. The world's largest computer network, the Internet, with more than a million computers, uses the connectionless Internet Protocol (IP). For the huge existing investment in IP networks to remain useful, one must devise mechanisms to carry IP traffic over ATM networks.
Classical IP requires intermediate roters and bridges. Hence, ATM can be used between these routers and bridges to interconnect LANs (See Figure 8.36). Although such a combination may only require ATM interfaces

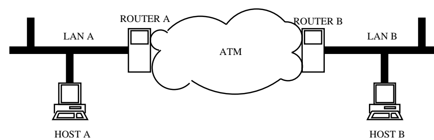


Figure 8.36: ATM as LAN Interconnection

in routers and bridges, it does not provide the ATM benefit to network end users. A way to avoid this problem, is to bring the ATM technology up to network end stations and servers (See Figure 8.37). This second ATM usage requires to rewrite upper layer protocols such as IP, Appletalk, IPX, NetBIOS, etc., directly over ATM. In addition, the NHRP novel IP model (See Figure 8.38), although not yet finalized, will obviate the use of intermediate routers for virtual connections that have paths in several networks.

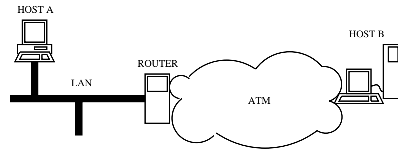


Figure 8.37: ATM to Network End Stations and Servers

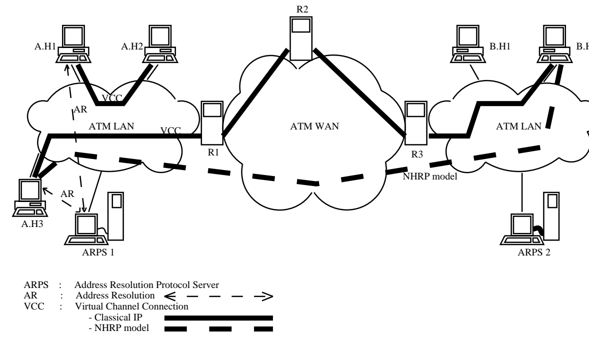


Figure 8.38: IP over ATM

To prevent the rewriting of upper layer protocols over ATM, one may implement the LAN Emulation (LANE), which emulates a LAN service, between these protocols and the AAL layer (See Figure 8.39). The LAN Emulation

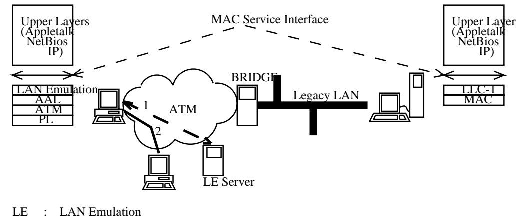


Figure 8.39: LAN Emulation

(LANE) feature will further contribute to the seamless integration of the variety of ATM communication services over new as well as legacy parts of future large computer networks.

8.8 ATM Overview Conclusion

In summary, to design an ATM network architecture, one must:

- choose the underlying transmission system, and implement the appropriate physical layer
- specify the internal architecture of ATM switches, and choose the interconnection topology, to define the ATM layer
- implement a congestion control, to deal with the problem of more cells arriving at an ATM switch than there are spaces in queues to store them all
- choose the service type to be provided at a given network access, and implement the appropriate AAL layer
- implement a flow control, to throttle the senders into sending no faster than the receivers can handle the traffic
- implement a traffic control, to let the network guarantee the QoS negotiated at connections establishment
- eventually implement few protocols above the AAL layer for InterNetworking purposes

Chapter 9

The Network Prototype

This chapter presents the architecture of the first ATM-based interconnection network that I implemented. I described this architecture at the Register Transfer Level (RTL) in the C programming language. The use of the C were motivated by the need to simulate the network prototypes directly under the workloads which are dynamically generated while running real parallel applications.
Conventional *VLSI* design tools, such as *VHDL, VERILOG, COMPASS, CADENCE*, etc., do not allow to meet this objective. Indeed when using these tools, the workload of the circuit under test has to be statically described at the beginning of the simulation.
Only few hardware design tools, such as the visual performance *Q+* of the *AT&T* Bell Laboratories, allow to take into account the external input events that may occur during the simulation. Even for these tools, the designer has to first stop the simulation, then alter some of the inputs, and finally let the simulation proceed.
Moreover, the C programming language usage in VLSI design, is not something that is totally new. Some VHDL analyzers translate the user submitted VHDL codes into intermediate C codes, which are thereafter compiled and linked in order to build the executable binary codes used by VHDL simulators.
The remainder of this chapter will present detailed implementations of the aforementioned prototype, layer by layer.

9.1 The PL Layer

A very simple PL layer has been implemented for the prototype. I chose an asynchronous underlying transmission system, simply because it is the choice that best match with our need to transmit cells byte per byte, and hence with the 8-bit architecture of the prototype switches. Indeed, transmission links are structured into a frame of size 8 bits, repeating with period T.
The PL layer itself, only performs the real transmission function. It receives a byte at the source and delivers it at the destination, within the aforementioned period T. Hence, it does not perform any of the rate adaptation, header error control, and cells delineation functions (See Section 8.3).
Consequently, the physical medium and PL layer stack, can be implemented by means of 8-bit buses, running at a frequency $\frac{1}{T}$. As shown in Table 9.1, using advanced CMOS technology, a 50 ns cycle bus clock can be realized, and the bandwidth of an 8-bit bus will hence be 160 Mbits/sec. However using ECL technology, the cycle of the bus clock can be decreased down to 10 ns, and the bandwidth of an 8-bit bus will therefore be 800 Mbits/sec [21].

Technology	Time Period T (ns)	8-bit Bus Bandwidth (Mbits/sec)
Advanced CMOS	50	160
ECL	10	800

Table 9.1: 8-bit Bus Bandwidth vs Technology

9.2 The ATM Layer

9.2.1 ATM Switches

I built the ATM layer of the prototype with 4x4 BMX switches because of two main reasons:

- The size 4 yet allows to set up an interconnection network with a small number of switches, but yet providing an interesting number of inputs and outputs.

- BMX switches achieve the same performance as the switches based on the classical output queueing approach without resorting to run faster than the transmission link (See Section 8.4.3).

As illustrated in Figure 9.1, a 4x4 BMX switch consists of four Primary Packet Distributor (PPD) units, four Secondary Packet Distributor (SPD) units, and sixteen Cross Point Memories (XPMs). PPD (respectively SPD)

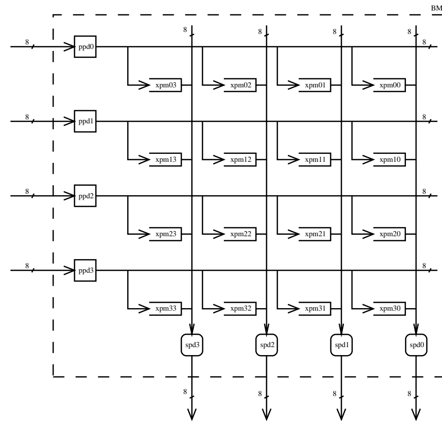


Figure 9.1: BMX Switches of the Prototype

units interface with the switch input (respectively output) ports. Each PPD and SPD unit has its own bytewise transmission interconnect. These interconnects are arranged in a 4x4 matrix. At each PPD and SPD interconnect cross point, there is an XPM, which is a circular FIFO of a maximum size, arbitrarily set to 8*53 bytes, and with one input port and one output port in order to enable concurrent reads and writes. PPD units store cells into XPMs, whereas SPD units remove them.
The congestion control strategy implemented uses a flow control mechanism. Indeed, when an upstream SPD or AAL unit begins to transmit the bytes of a cell toward a downstream PPD or AAL unit, the latter sends a suspension signal back to the former if the entire cell can not be stored in the destination queues without overflow. When the former receives a suspension signal, it first stops the current transmission and next tries either an another or the same transmission.
My PPD and SPD units' internal architecture proposals, are rather my personal implementation of what is described in summary in the FUJITSU's

paper [21]. In my implementation, PPD and SPD units are pipelined byte-wise operators. That is, they gradually process ATM cells in a byte by byte manner.
Basically, a PPD unit receives ATM cells from an upstream SPD or AAL unit, translates its header, determines the destination XPMs, then sends it to them. The SPD scans the XPMs at its interconnect, removes ATM cells from the XPMs, and sends them to a downstream PPD or AAL unit. Inside a BMX switch, dual PPD and SPD units operate independently and asynchronously.

9.2.1.1 PPD Units

Figure 9.2 roughly shows the internal architecture of a PPD unit. There,

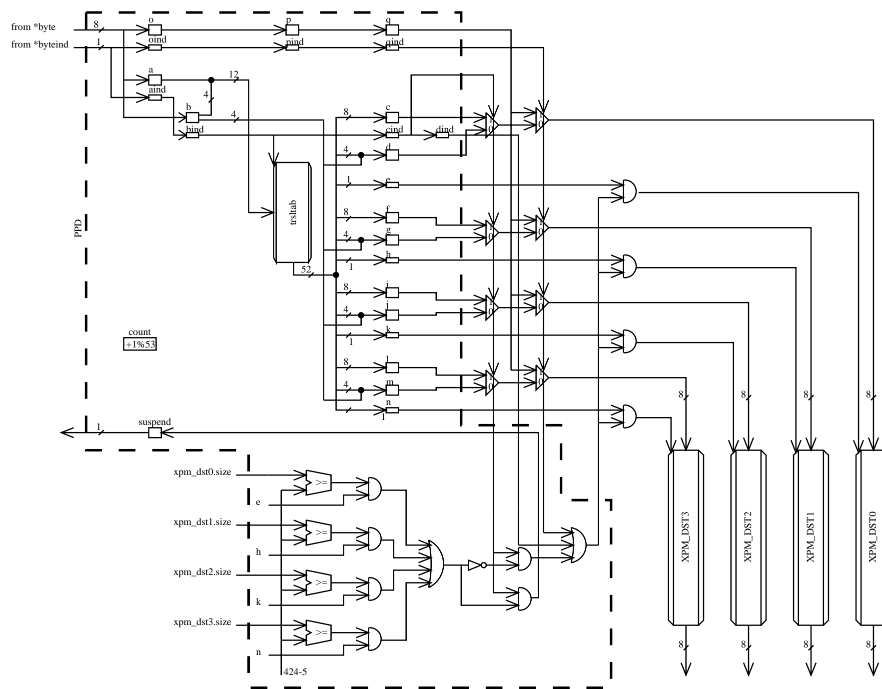


Figure 9.2: PPD Units Architecture

the "*byte" 8-bit input signal implements an asynchronous transmission link (See Section 9.1), whereas when set to 1, the "*byteind" 1-bit input signal

indicates that the byte currently on "*byte" is valid.
The switches implemented, are VP switches. That is, they only translate the headers' VPI field of the cells. In this case, the translation tables of my switches put in relation each incoming VPI, with up to four outgoing VPIs. An easy way to achieve that, is to store four outgoing VPIs and four additional bits, that selects the switch destination output ports, at the translation table entry indicated by the value of an incoming VPI. Figure 9.3 describes the format, that I defined for the translation tables' entries. According to

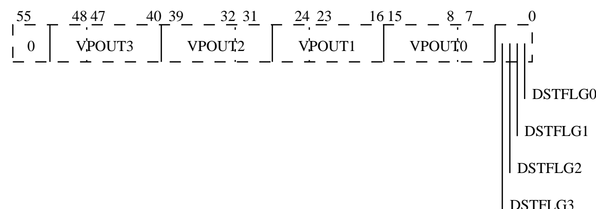


Figure 9.3: Translation Table Entries Structure

this format, if "DSTFLGi", with $0 \leq i \leq 3$, is set to '1' at any given translation table entry, then incoming cells with a VPI pointing to that entry, will be forwarded to the switch output port number i, with a new VPI equal to "VPOUTi".
The number of entries of the translation tables, indicates the the maximum value of the VPIs, and hence the maximum number of open connections that can simultaneously be sustained by every switch. I set this number to 256, to avoid having to declare in my C implementation of the PPD unit architecture, translation tables of size 2^{12} (that is 4096) in case the entire VPI field is used, and since restricting the aforementioned maximum number to 256 does not have any impact over the design of the PPD units, and hence the BMX switches, and hence the network prototype.
In this context, my switch translation tables are 256x52 memories, with each, one input port and one output port to allow concurrent reads and writes. In my implementation, I assume that translation tables reads and writes last each, only one switch clock cycle.
In addition to the header translation table (also routing table), the other important elements in the internal architecture of a PPD unit are the registers. A counter modulo 53, named "count", is used for cells delineation. It is

incremented by one whenever the PPD unit receives a byte. The beginning (respectively end) of a cell is identified by the arrival of a byte, and the value of "count" set to 0 (respectively 52).
The first byte of an incoming cell is stored in the "a" register, whereas its second byte is stored in the "b" register. The storage of the second byte, always triggers the translation process of the incoming VPI.
While the "o" register receives the third byte of the incoming cell, the "c" and "d" (respectively "f" and "g", "i" and "j", "l" and "m") registers receive "VPOUT0" (respectively "VPOUT1", "VPOUT2", "VPOUT3"), whereas the "e" (respectively "h", "k", "n") register receives "DSTFLG0" (respectively "DSTFLG1", "DSTFLG2", "DSTFLG3"), from the translation table.
As soon as the incoming VPI translation result has been stored, if the "e" (respectively "h", "k", "n") register is set to '1' and the destination XPM, that is "xpm_dst[0]" (respectively "xpm_dst[1]", "xpm_dst[2]", "xpm_dst[3]"), is full (i.e. xpm_dst[].size≥424-5), then the current cell transmission is suspended, due to a congestion.
In this case, the PPD unit cancels the forwarding process of the current cell transmission, and sends a suspend signal back to the upstream SPD or AAL unit, by setting its "suspend" register to '1'. Note that, by the time the suspend signal from the PPD unit reaches the upstream SPD or AAL unit, the latter would have already sent the fifth byte. When this fifth byte arrives at the PPD unit, it is simply ignored.
On the other hand, if all of the selected destination XPMs are not full, the PPD unit proceeds with the writing of the incoming cell's bytes in the selected destination XPMs, as follows:

1. while the "c", or "f", or "i", or "l" register contents move forward into "xpm_dst[0]", or "xpm_dst[1]", or "xpm_dst[2]", or "xpm_dst[3]" respectively, the third and fourth bytes of the incoming cell are stored in the "p" and "o" registers respectively

2. while the "d", or "g", or "j", or "m" register content moves forward into "xpm_dst[0]", or "xpm_dst[1]", or "xpm_dst[2]", or "xpm_dst[3]" respectively, the third, fourth, and fifth bytes of the incoming cell are stored in the "q", "p", and "o" registers respectively

3. the PPD unit enters in a loop, where it behaves as a three staged pipeline, until the current cell transmission is completed. Indeed, when

in this loop, the "q" register content moves forward into the selected destination XPMs, while the "p" register content moves forward into the "q" register, and the "o" register content moves forward into the "p" register, and the incoming byte moves forward into the "o" register.

The C data structures that define the PPD units' resources are shown in Appendix A.1. Whereas, the C code that simulates their behavior is presented in Appendix A.2.
A *ppd_scheduler(curppd)* call simulates the behavior of the PPD unit indicated by *curppd* during one network clock cycle.

9.2.1.2 SPD Units

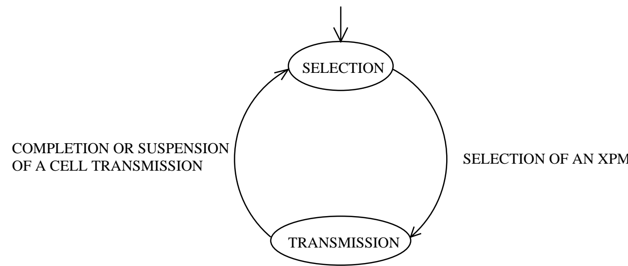


Figure 9.4: SPD Units Behavior

The two states machine illustrated in Figure 9.4, characterizes the behavior of SPD units. SPD units can run in either *selection* or *transmission* mode.
The *selection* mode is the initial one. When an SPD unit runs in this mode, it scans the XPMs on its interconnect in a *round-robin* manner, until it encounters a non-empty XPM, which is then selected. Then, the SPD unit switches in the *transmission* mode.
When an SPD unit runs in the *transmission* mode, it transmits bytes from the cell at the head of the selected XPM, until the entire cell is transmitted, or a *suspension* signal is received from the downstream PPD or AAL unit. Then, it switches back in the *selection* mode.
Figure 9.5 shows the internal architecture of an SPD unit, its interconnect (which is merely an 8-bit 4x1 multiplexor), as well as the way it interacts with its outer world in general, and XPMs in particular. The "buf" and "bufind"

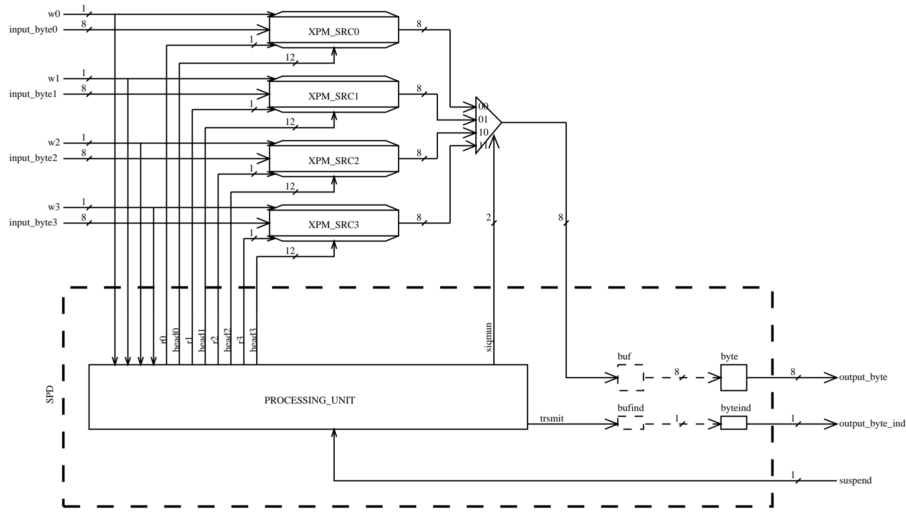


Figure 9.5: SPD Unit Architecture

registers are not part of the physical architecture of SPD units. They only seve to implement a network clock cycle delay of the signal propagation towards downstream PPD or AAL units.
The main component of the proposed architecture of SPD units, is the processing unit, whose internal architecture is shown in Figure 9.6. It consists of two circuits:

- the *selection* circuit. It is a 4-bit processor, whose datapath architecture is multiplexor-based, and consists of:

  1. the ”iqrst”, ”i”, ”siqnum”, and ”trsmit” registers,
  2. an adder modulo four,
  3. and five multiplexors.

  Its controller implements the MEALY One State Machine illustrated in Figure 9.6.
  Once an XPM is selected, the ”siqnum” register acquires its rank and the ”trsmit” register is set to ’1’.
  Note that the selection loop always starts from the XPM succeeding the last selected XPM. This allows each SPD unit to fairly select all its

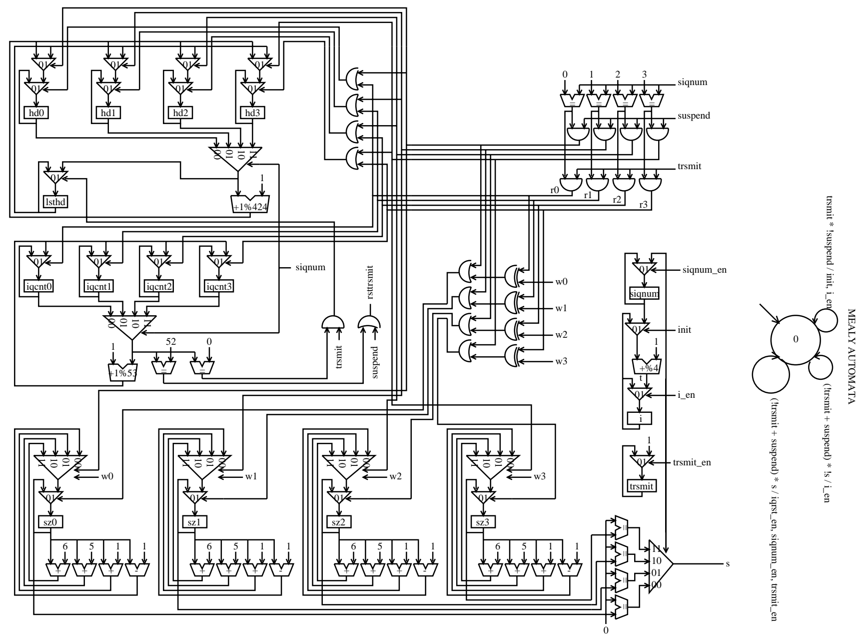


Figure 9.6: SPD Processing Unit Architecture

four XPMs.
In addition, the selection circuit runs four time faster than the transmission links. This enables each SPD unit to scan, in the worst case, all its four XPMs within the same network clock cycle. This prevents the introduction of unnecessary idle bytes between two successive cell transmissions on a given transmission link.

- the *multiplexing* circuit. It runs at the network (also transmission links) speed and consists of a set of registers, multiplexors, adders, and, or, and xor gates, used to manage the transmission of cells from XPMs.
  The "hd_i" register points at the head byte of XPM_i and the "sz_i" register contains the size of XPM_i, where 0≤i≤3.
  The "$iqcnt_i$" register is used for a cell delineation purpose while transmitting cells from XPM_i, with 0≤i≤3.
  The "lsthd" register receives the value of the "hd" register corresponding to the selected XPM, whenever a cell transmission begins.
  The "hd" (respectively "iqcnt") register corresponding to the selected XPM, is either incremented by one modulo 424 (respectively 53) whenever a byte is removed from the selected XPM, or updated with the value of the "lsthd" register (respectively set to '0') whenever a suspension signal is received.
  Within a network clock cycle, three different actions may affect the XPMs, and hence the "sz" registers. They are:

  1. a cell byte transmission,
  2. a cell transmission suspension,
  3. and a cell byte queueing.

  The two formers can not simultaneously affect the same "sz" register. They only affect the "sz" register corresponding to the selected XPM. The latter and any of the two formers may concurrently affect an identical "sz" register. Therefore, the "sz" register corresponding to the selected XPM may be either:

  1. incremented by either 5 or 6,
  2. or decremented by 1,
  3. or hold unchanged.

The "sz" registers not corresponding to the selected XPM may only be either

1. incremented by 1,
2. or hold unchanged.

The C data structures that define the SPD units' resources are shown in Appendixes A.1 and B.1. Whereas, the C code that simulates their behavior is presented in Appendix B.2.
An *spd_scheduler(curspd)* call simulates the behavior of the SPD unit indicated by *curspd* during one network clock cycle.

9.2.2 The Interconnection Topology

Twelve 4x4 BMX switches (See Section 9.2.1) has been interconnected as depicted in Figure 9.7, to makeup the 16x16 three stages interconnection network of the prototype. Since each of the switches used consists of one

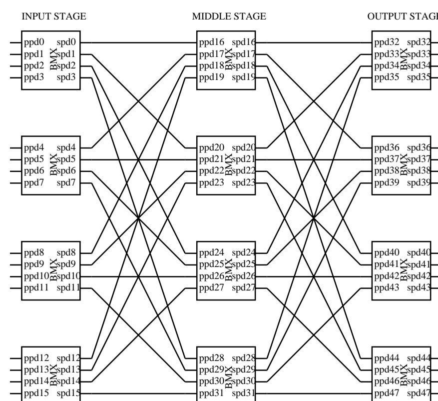


Figure 9.7: Prototype Interconnection Network Topology

PPD unit at each input port, and one SPD unit at each output port, the interconnection network of the prototype is hence made up of 48 PPD units, and 48 SPD units. Appendix C.1 shows the interconnection algorithm of PPD and SPD units. This algorithm assigns the first 16 PPD units to the input stage switches input ports, the next 16 PPD units to the middle stage switches input ports, and the last 16 PPD units to the output stage switches

input ports. Whereas, it assigns the first 16 SPD units to the input stage switches output ports, the next 16 SPD units to the middle stage switches output ports, and the last 16 SPD units to the output stage switches output ports. In the meantime, it connects the SPD units of the input (respectively middle) stage, to the PPD units of the middle (respectively output) stage as illustrated in Figure 9.7.

There are four different paths existing between any (input port, output port) pair in such an interconnection network. Appendix C.2 presents the algorithm, that computes the four different routes, both between each network pair of (input port, output port) with regard to point-to-point communications, and from each input port to all the output ports of the network with regard to broadcast communications. Figure 9.8 shows the four computed point-to-point paths between the network input port number 5 and output port number 14. Whereas, Figure 9.9 illustrates the four computed broadcast paths from the network input port number 5 to all the output ports. This

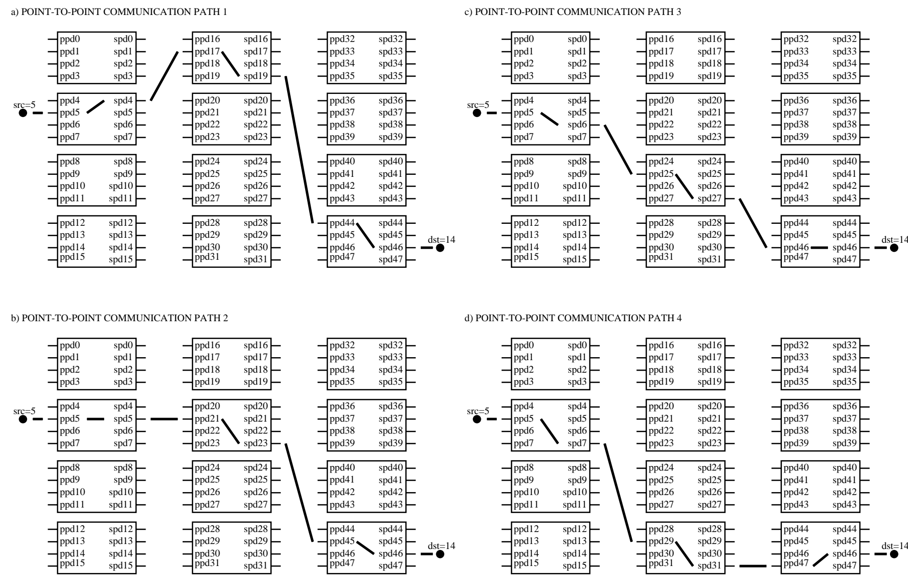


Figure 9.8: Point-To-Point Communication Paths from Src 5 to Dst 14

path computation algorithm also reserves entries in the routing (also header translation) tables of PPD units through the network, for all the computed

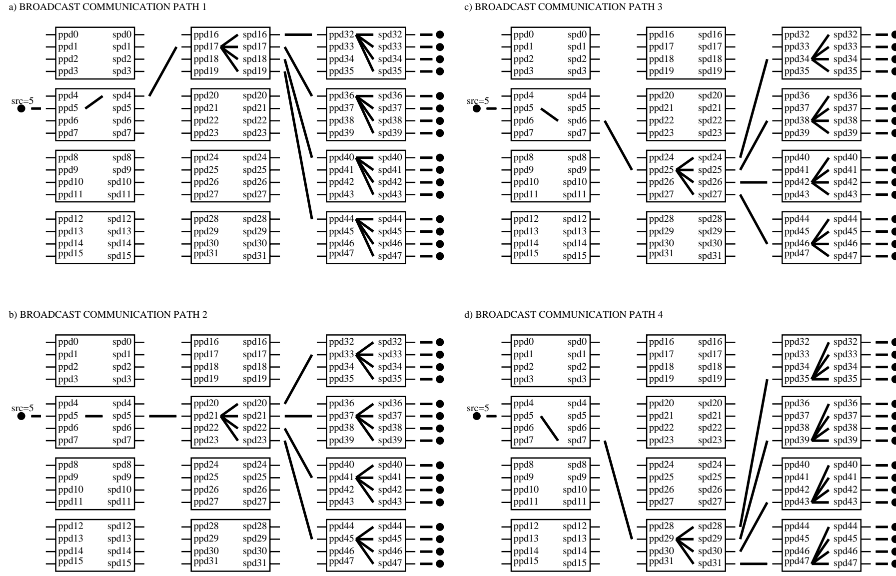


Figure 9.9: Broadcast Communication Paths from Src 5

routes.

Regarding a computed point-to-point path, if a is its reserved *vpi (virtual path identifier)* between the middle and output stages, and b is its reserved *vpi* between the input and middle stages, and c is its input stage access reserved *vpi*, then a is stored at the entry number b of the routing table of its middle stage input PPD unit, whereas b is stored at the entry number c of the routing table of its input stage input PPD unit.

And, regarding a computed broadcast path, if a_0, a_1, a_2, and a_3 are its reserved *vpi*s between the middle and output stages, and b is its reserved *vpi* between the input and middle stages, and c is its input stage access reserved *vpi*, then a_0, a_1, a_2, and a_3 are stored at the entry number b of the routing table of its middle stage input PPD unit, whereas b is stored at the entry number c of the routing table of its input stage input PPD unit.

According to the definitions given in Section 8.4.4, the interconnection network of the prototype is therefore a rearrangeable Clos(4,4,4) network, since e=4 and m=4 thus m$\geq$e.

9.3 The AAL Layer

9.3.1 General Description

The ATM adaptation layer of the network prototype, is a lightened version of that of the type 3/4 presented in Section 8.5.
In accordance with Table 8.3, its SAR sublayer performs only the message segmentation and reassembly, and cell error, loss, and insertion processing, whereas its CS sublayer performs only the message framing, multiplexing, and demultiplexing.
In addition, the CS and SAR sublayers' PDUs of the network prototype are slightly different from those described in Section 8.5.
I designed the ATM adaptation layer of the network prototype this way in order to seamless implement the *Parallel Virtual Machine (PVM)* message passing library [42, 43] over it.
Indeed:

- It is not necessary to have the SAR sublayer perform the partly filled cell processing, since PVM implicitely handles that issue.
- It is not necessary too to have the CS sublayer perform neither an end-to-end flow control, since an hop-by-hop flow control is implemented all along the network prototype, nor the memory space indication, since messages may carry their sizes at their fronts, for example.
- There is no compelling need in having the AAL layer generate a continuous flow of cells, and hence perform the message padding.
- The SAR sublayer uses a wide enough cell sequence numbering such that, the CS sublayer may not perform the message spanning processing, since it is unlikely to happen.

To understand how the prototype's AAL works, here is an overview of the transformations undergone by a message that passes through it.
We start with a single data stream, say a file that a given PVM task hands to the prototype's AAL at the source, in order to be sent to an another PVM task:

....

First, the source Convergence Sublayer puts a 20-byte PVM header at its front.
As shown in Figure 9.10, this PVM header contains the external (also machine independent) data representations of the "3-length name" and "instance number" of the destination PVM task, as well as the "message identifier". These information help the prototype's AAL at the destination, to

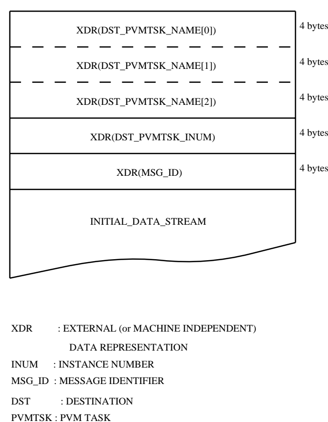


Figure 9.10: PVM or Prototype AAL CS Headers

deliver the message to the right PVM task.
If we abbreviate the PVM header as "P", the file to be sent now looks like this:

P....................................

Next, the source SAR sublayer breaks it up into manageable chunks,

P.... ....

and puts a SAR header at the front of each.
As illustrated in Figure 9.11, this SAR header contains the cell "type", "flag", and "sequence number", as well as the source and destination "port number". The "type" field indicates whether it is either a message or an acknowledgement chunk, when it is set to 0 or 1 respectively.
The "flag" field rather indicates whether it is a message beginning and/or middle and/or end chunk. This is used to delineate the message at the destination.

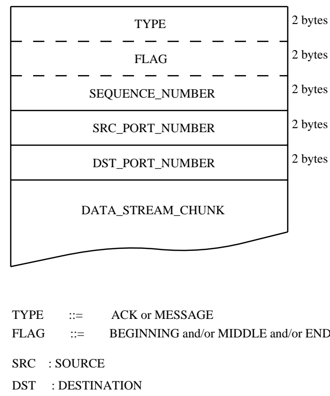


Figure 9.11: Prototype AAL SAR Headers

Each chunk has a sequence number. This is used to let the destination make sure that it gets the message chunks in the right order, and that it has not lost any.
The port numbers are used to keep track of different transmissions. The source and destination port numbers are reversed when the destination sends an acknowledge receipt back to the source.
If we represent the SAR header with ”S”, the file to be sent now looks like this:

SP.... S..... S..... S..... S..... S..... S..... S...

Next, the source SAR sublayer adds an ATM header (See Figure 8.10) at the front of each actual chunk. Since the ATM adaptation layer of the prototype provides connectionless data services, chunks belonging to the same message can take different routes to the destination. Different ATM headers may therefore be used for them.
If we represent ATM headers with ”A”, the file to be sent now looks like this:

ASP.... AS..... AS..... AS..... AS..... AS..... AS..... AS...

Finally, the source SAR sublayer inserts the chunks of the message to be sent into the ”send” buffer of the source. It will flying compute a Cyclic Redundancy Code (or CRC code) [27] over the SAR payload for each chunk during its transmission.

To achieve that, the source SAR sublayer divides the bit string resulting from the concatenation of the SAR payload part of the chunk being transmitted and a sixteen-zero-bit string, by the 10001000000100001 17-bit string (which corresponds to the CRC-CCITT generator polynomial $x^{16} + x^{12} + x^5 + 1$), using modulo 2 division and subtraction.
Recall that, modulo 2 addition and subtraction are identical to exclusive or.
The remainder of this division is the CRC code of the chunk being transmitted. It is put at the end of the chunk, hence making up the corresponding SAR trailer.
Figure 9.12 illustrates the CRC code calculation for a message 1101011011 and $x^4 + x + 1$ as the generator polynomial.

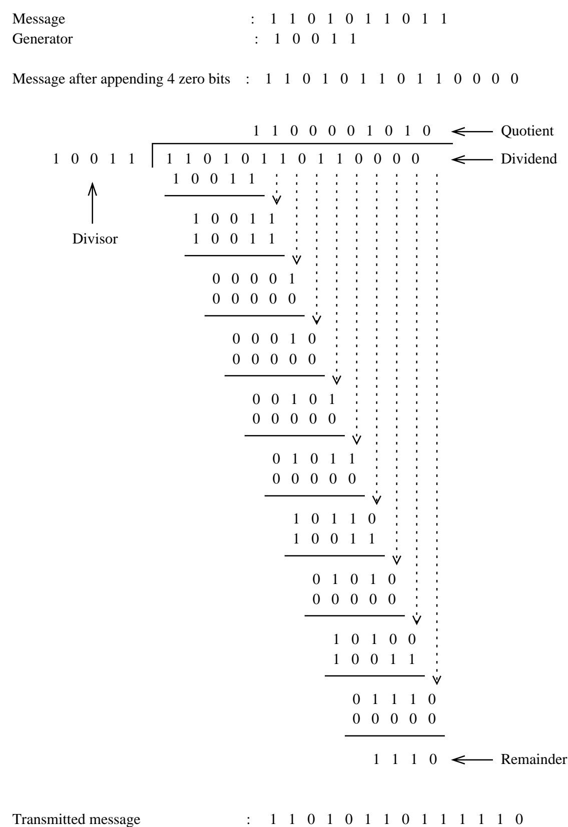


Figure 9.12: Calculation of a CRC Code

Figure 9.13 shows two circuits that compute CRC codes.

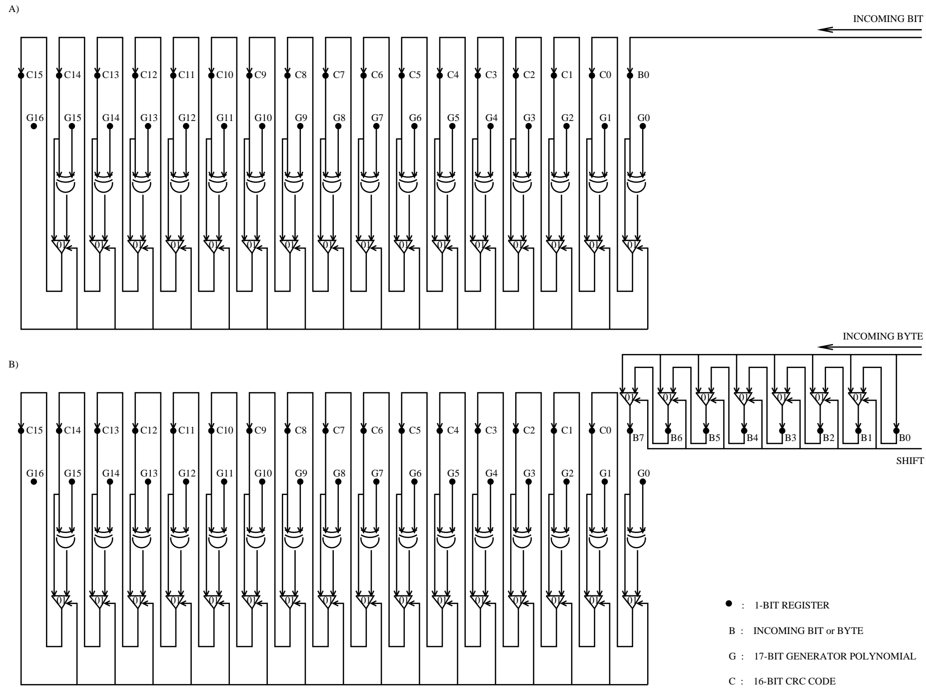


Figure 9.13: CRC Codes Calculation Circuits

The circuit A) runs at the same speed as the transmission links. It is designed to process one bit within one clock cycle. Then, it is used for bitwise transmission of cells over the transmission links. However, it is at the origin of the circuit B), which runs at least eight times faster than the transmission links. The latter is designed to process eight bits within one transmission links clock cycle. Hence, it is used for bytewise transmission of cells over the transmission links.
Both circuits implement the body of the CRC code calculation loop. The "c" register is updated by either:

- its previous value shifted by one bit to the left, if the "c" register's leftmost bit is set to 0,

or

- the result of the bitwise exclusive or between

    1. the "b" register's leftmost bit appended to the "c" register, and
    2. the "g" register which contains the coefficient of the generator polynomial,

  shifted by one bit to the left, if the "c" register's leftmost bit is set to 1.

During the aforementioned shift operations, the "b" register is shifted too. And, the "c" register's rightmost bit takes the value of the "b" register's leftmost bit.
The "c" register is initialized to 0, whereas the "b" register is assigned the incoming bit or byte, whenever a CRC code calculation loop begins.
The circuit B) is the one that is simulated on the prototype. Appendix D.1 shows the C code that generates the CRC code corresponding to a cell being transmitted. Whereas, Appendix D.2 rather shows the C code that recomputes the CRC code corresponding to a cell being received.
In both C codes, the bit string from the bit 23rd through the bit 8th of the "crc" variable, contains the computed CRC code when the calculation loop ends, that is, when the 48 bytes of the payload of the cell being transmitted or received are all processed.
If we abbreviate the computed CRC codes as "C", the file to be sent finally looks like this:

ASP....C AS.....C AS.....C AS.....C AS.....C AS.....C AS.....C AS... C

At this point, the chunks of the file to be sent make up an ATM cell each. The source SAR sublayer removes a given cell from its "send" buffer, when it receives the corresponding acknowledgement. Otherwise, it sends it again after a timeout.
When the transmitted cells arrive at the other end, the destination SAR sublayer first recomputes their CRC codes, throws away any cell whose result disagrees with the original or that is not at the right destination, and inserts the remaining ones into its "receive" buffer and sends the corresponding acknowledgements back to the source SAR sublayer.
Next, the destination SAR sublayer looks at the SAR headers of the inserted cells to gradually combine them to the original file. It will serve this file to the destination PVM task, only when it is completely recombined.
Finally, the destination SAR sublayer discards the recombined file after a timeout. Then, it removes the corresponding cells from its "receive" buffer.

9.3.2 The AAL Units

In this section, I describes the architecture of the 16 AAL units that make up the ATM adaptation layer of the prototype. Figure 9.14 shows the floor plan of the internal highly parallel architecture of these units.

9.3.2.1 The AAL units' Interfaces

Each AAL unit of the prototype uses three sets of signals to communicate with its outside world:

- The first set of signals is used to communicate with a downstream PPD unit. It is made up of:

  1. An 8-bit output signal called "ogbyte". This signal implements an asynchronous transmission link as described in Section 9.1.
  2. A 1-bit output signal called "ogbytei". When set to 1, this signal indicates that the byte currently on "ogbyte" is valid.
  3. A 1-bit input signal called "*ogsuspend". When set to 1, this signal tells the AAL unit to stop the current cell transmission, due

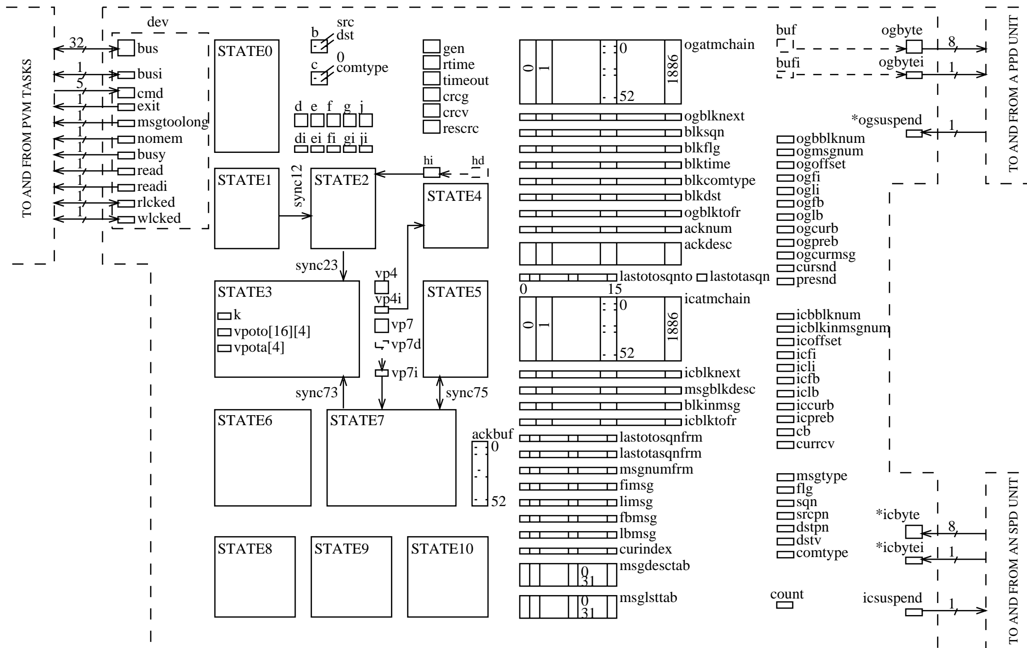


Figure 9.14: Prototype AAL Units Architecture Floor Plan

to a lack of space into at least one of the requested destination queues on the downstream PPD unit.

- The second set of signals is used to communicate with an upstream SPD unit. It consists of:
  1. An 8-bit input signal called "*icbyte". This signal implements an asynchronous transmission link as described in Section 9.1.
  2. A 1-bit input signal called "*icbytei". When set to 1, this signal indicates that the byte currently on "*icbyte" is valid.
  3. A 1-bit output signal called "icsuspend". When set to 1, the AAL unit requires the upstream SPD unit to stop the current cell transmission, due to a lack of space in its "receive" buffer.
- The third set of signals is used to communicate with concurrent PVM tasks running on an host machine. This third set includes:
  1. An 32-bit input/output signal called "bus". It implements an internal 32-bit bus of an host machine.
  2. A 1-bit input/output signal called "busi". When set to 1, this signal indicates that the data currently on "bus" are valid.
  3. A 5-bit input signal called "cmd". It is a command that the host machine sends to the AAL unit, telling it how to handle the valid data currently on "bus". Table 9.2 gives the names of all the commands that I have implemented so far, as well as their corresponding identifier numbers.
  4. A 1-bit output signal called "exit". The AAL unit sets this signal to 1, when the valid data currently on "bus" could not all fit into its "send" buffer, when at that time, the latter only contains cells from the message being packed into it.
  5. A 1-bit output signal called "msgtoolong". The AAL unit sets this signal to 1, when the valid data currently on "bus" could all exactly fit into the space left in its "send" buffer, when at that time, the latter only contains cells from the message being packed into it.

Command Name	Command Identifier Number
atmininsend	0
atmputnint	1
atmputnshort	2
atmputnlong	3
atmputnfloat	4
atmputndfloat	5
atmputncplx	6
atmputndcplx	7
atmputbytes	8
atmputstring	9
atmgetnint	10
atmgetnshort	11
atmgetnlong	12
atmgetnfloat	13
atmgetndfloat	14
atmgetncplx	15
atmgetndcplx	16
atmgetbytes	17
atmgetstring	18
atmsnd	19
atmrcv	20
atmsetdstandcomtype	21

Table 9.2: Names and Identifier Numbers of Commands to AAL Units

6. A 1-bit output signal called "nomem". The AAL unit sets this signal to 1, when there is not anymore space left into its "send" buffer.
7. A 1-bit output signal called "busy". The AAL unit sets it to 1 (respectively 0), when it starts (respectively ends) to process the valid data currently on "bus". And, when this signal is set to 1, the AAL unit cannot accept any additional command and data from PVM tasks.
8. A 1-bit output signal "read". When valid and set to 1, this signal indicates that the AAL unit grants the permission to read the "receive" buffer, to a requesting PVM task.
9. A 1-bit output signal called "readi". When set to 1, it indicates that the "read" signal is valid.
10. A 1-bit input/output signal called "rlcked". PVM tasks set this signal to 1, when they send a read requests (that is, a receive command called "atmrcv") to the AAL unit. They set it to 0 either right after the AAL unit ends to process their read requests if they have been issued a read permission denied, or after have read the "receive" buffer otherwise.
    Once a given PVM task has set "rlcked" to 1, the other PVM tasks cannot access the AAL unit to either read the "receive" buffer, or write into the "send" buffer.
11. A 1-bit input/output signal called "wlcked". PVM tasks set this signal to 1, when they send a write request (that is, an init_send command called "atminitsend") to the AAL unit. The latter sets this signal to 0, either right after it ends to process the PVM tasks' write requests if they have been issued a write permission denied, or after it has packed their message into the "send" buffer otherwise.
    Once a given PVM task has set "wlcked" to 1, the other PVM tasks cannot access the AAL unit to write into the "send" buffer. However, they can access the AAL unit to read the "receive" buffer, only if the former PVM task has stop its writing and is blocked waiting for available space to be freed in the "send" buffer.

At the UNIX operating system and C programming language levels, the first and second sets of signals describe communication facilities inside a single UNIX process called "network", between the AAL and PPD units on the one hand, the AAL and SPD units on the other hand. Therefore, they can be implemented by means of local variables and pointers.
Things are different regarding the third set of signals. Indeed, it describes the communication facilities between PVM tasks and the "network" process. In this case, they cannot be implemented by means of local variables and pointers, since the scope of these latters does not exceed the context of a single UNIX process. Fortunately, distinct UNIX processes can communicate each other using:

- signals [30]
- inter-process channels called "pipe" [32]
- the following Inter-Process Communication (IPC) facilities:
  1. message queues [34]
  2. shared memory [36]
  3. semaphores [38]

I chose the shared memory IPC facility, since it is an easy way to implement the underlying communication system between the PVM tasks on an host machine and an AAL unit. Hence, the signals of the third set are gathered in shared data structure called "dev".

9.3.2.2 The AAL Units' Components

Inside the prototype's AAL units, fundamental components are the "send" and "receive" buffers called "ogatmchain" and "icatmchain" respectively. They are structured into 1887 adjacent blocks of 53 bytes size each. And, they are each managed by means of one chained list of free blocks and one chained list of busy blocks. The sum of the sizes of these two chained lists, always equals 1887. This is because both chained lists are duals.
Indeed, whenever a cell is written into the "send" or "receive" buffer, the head block is extracted from the corresponding free blocks chained list and inserted at the tail into the corresponding busy blocks chained list.

Reciprocally, whenever a cell is removed from the "send" or "receive" buffer, the block containing the concerned cell is extracted from the corresponding busy blocks chained list and inserted at the tail into the corresponding free blocks chained list.
The "ogbblknext" (respectively "icbblknext") table contains the free and busy blocks chained lists of "ogatmchain" (respectively "icatmchain").
In "ogbblknext", the "ogfi" (respectively "ogfb") register indexes the position of the block at the head of the free (respectively busy) blocks chained list. Whereas, the "ogli" (respectively "oglb") register indexes the position of the block at the tail of the free (respectively busy) blocks chained list.
A similar reasoning can be made with regard to the "icfi", "icfb", "icli", and "iclb" registers and the "icbblknext" table.
This way of handling the "send" and "receive" buffers, allows constant cost cells insertion (respectively removal) into (respectively from) them.
In addition to the "send" and "receive" buffers, the other important components of the prototype's AAL units, are their eleven controllers. They are MOORE Finite State Machines, and they are named "STATEi", with $0 \leq i \leq 10$.
Controller "STATE0" is the AAL unit's component to which PVM tasks speak to. It receives and processes their commands and corresponding data. When it ends to process a received command, it sends the process reports back to the sending PVM task, and sets "busy" to 0.
Controller "STATE0" can accept, and hence process, at most one command at a time. A PVM task is allowed to send a command to it, when "busy" is set to 0. In this case, it puts a command identifier number into "cmd" and the related data if any into "bus", sets "busi" to 1, and blocks until "busi" and "busy" are both set to 0.
When "busi" is set to 1, controller "STATE0" sets "busy" to 1 and "busi" to 0. Then, it starts to process the received command and the related data if any.
If the received command is "atminitsend", then controller "STATE0" knows, that it is the beginning of the segmentation of a new message. It allocates the head free block of the "send" buffer, which then becomes its new tail busy block. Then, it positions the "ogcurmsg" register on this new tail busy block, hence marking in the busy blocks chained list, the head block of the message being packed into the "send" buffer. Finally, it marks the new tail busy block as not being ready to be transmitted by setting "blktime[ogcurmsg]" to -1.

Else, if the received command is of an "atmput" type (See Table 9.2 in Section 9.3.2.1), then controller "STATE0" fills up, in a byte by byte manner, the tail busy block of the "send" buffer with the significant part of "bus".
For example, if the received command is "atmputnshort" and data are not transmitted in their eXternal Data Representations (XDR), then controller "STATE0" will pack the first, then the second byte from the left of "bus", into the tail busy block of the "send" buffer.
The fill up process is based upon a well known mechanism, that uses a base and an offset. That is, incoming bytes are always stored at the location indicated by the base address increased by the offset (See Figure 9.15). The

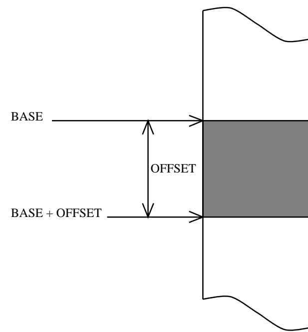


Figure 9.15: Base+Offset Fill up Mechanism

offset is incremented by one after each byte storage.
For controller "STATE0", the base is always the 16th byte of the tail busy block, and the offset current value is contained in the "ogoffset" register, where 0≤"ogoffset"≤36. Prior to store a byte from "bus" into the "send" buffer, controller "STATE0" checks if the value of the "ogoffset" register equals 36. If it is not the case, it proceeds with the writing. Else, it checks if "ogbblknum"<1887, that is if there are free blocks in the "send" buffer. If it is the case, it allocates the head free block which thus becomes the new tail busy block, then marks it as not being ready to be transmitted by setting the corresponding "blktime" table entry to -1, and finally proceeds with the writing. Else, it checks if "ogmsgnum">0, that is if there are more than one message in the "send" buffer. If it is the case, it sets "msgtoolong" to 1. If not, it sets "exit" to 1.
Else, if the received command is of an "atmget" type (See Table 9.2 in Sec-

tion 9.3.2.1), then controller "STATE0" reads the corresponding amount of bytes from the "receive" buffer's busy block indicated by the "currcv" register, and stores them to "bus", from left to right.
The read process is too based upon a mechanism using a base and an offset. That is, controller "STATE0" always reads bytes from the location indicated by the base address increased by the offset.
For that purpose, the 16th byte of the block indicated by "currcv" is used as base, whereas the offset is contained in the "icoffset" register, providing that $0 \leq$"icoffset"≤ 36. The "icoffset" register is incremented by one after each read.
Prior to read a byte from the "receive" buffer, controller "STATE0" checks if "icoffset" equals 36. If it is not the case, it merely proceeds with the reading. Else, it first moves "currcv" ahead to the next block in the blocks chained list of the message being read and then proceeds with the reading.
Else, if the received command is "atmsnd" or "atmrcv", then controller "STATE0" fills up, in a pipeline manner, the "d", "e", "f", "g", and "j" 32-bit registers with the data contained in "bus". That is, whenever each of these commands is received, "j" is assigned "g", "g" is assigned "f", "f" is assigned "e", "e" is assigned "d", and "d" is assigned "bus". After controller "STATE0" received five consecutive "atmsnd" (respectively "atmrcv") commands, it releases controller "STATE1" (respectively "STATE8") from its idle state by setting its internal state to 1, and the aforementioned registers contain in the following order, the eXternal Data Representations (XDR) of the:

1. message identifier
2. destination PVM task's instance number
3. destination PVM task's 3-length name

which make up the 20-byte PVM header of the message being packed into the "send" buffer. Else, if the received command is "atmsetdstandcomtype", then controller "STATE0" stores the most (respectively less) significant word (also 16 bits) of "bus", which contains the destination AAL unit number (respectively the communication type: point-to-point or broadcast), into the less significant word of the "b" (respectively "c") register.
Appendix E.2 shows a graph giving a more accurate description of controller

"STATE0".
When released from its idle state, Controller "STATE1" packs the 20-byte PVM header provided by the "j", "g", "f", "e", and "d" registers set, into the first block of the message being split into cells, that is, the "send" buffer's busy blocks chained list's block indicated by "ogcurmsg". Then, it goes from "ogcurmsg" to "oglb" through the aforementioned chained list.
At each step, it:

- sets up the current block's SAR header (See Figure 9.11), by:
  - setting the corresponding "flag" field to an appropriate value, hence indicating whether it is the beginning and/or middle and/or end cell,
  - assigning the "lastotasqn" register (respectively "lastotosqnto" table's entry corresponding to the destination AAL unit) to the corresponding "sequence number" field, since the current block is involved in a broadcast (respectively point-to-point) communication.
    The "lastotasqn" register and "lastotosqnto" table's entries are incremented by one modulo 2^{32} after having provided a sequence number.
  - writing the "b" register's less significant word, that is the number of the destination AAL unit, into the corresponding "destination port number" field
- assigns 1 to the "acknum" table's entry corresponding to the current block, and to the "ackdesc" table's entry corresponding to the current block and destination AAL unit number, since the current block is involved in a point-to-point communication.
  On the other hand, it assigns 16 to the "acknum" table's entry corresponding to the current block, and 1 to all the "ackdesc" table's 16 entries corresponding to the current block, since the current block is involved in a broadcast communication.

Before it returns to its idle state, controller "STATE1" sends the "sync12" signal to controller "STATE2".
Appendix E.3 shows a graph giving a more accurate description of controller "STATE1".

When controller "STATE2" receives the "sync12" signal sent by controller "STATE1", it goes from "ogcurmsg" to "oglb" through the "send" buffer's busy blocks chained list.
At each step, it:

1. sends a VPI request, by means of the "sync23" signal, to controller "STATE3",
2. blocks until it gets an ATM header from controller "STATE4", that is until it receives an "hi" signal from it,
3. writes the received ATM header into the current block, and marks it as being ready for transmission by assigning 0 to the corresponding "blktime" table's entry.

Thereafter, it increments by one the number of messages currently in the "send" buffer, and releases the access to the AAL unit by assigning 0 to the "wlcked" signal.
Appendix E.4 shows a graph giving a more accurate description of controller "STATE2".
Controller "STATE3" disposes of two routing tables called: "vpoto" and "vpota".
"vpoto" provides a network access VPI for each of the possible four point-to-point paths, to each of the possible sixteen destination AAL units.
Instead, "vpota" provides a network access VPI for each of the possible four broadcast paths to all the possible sixteen destination AAL units.
When controller "STATE3" receives the "sync73" signal sent by controller "STATE7", that is the latter is requesting a network access VPI to send an acknowledge receipt back to the AAL unit indicated by the "srcpn" register, it gets one from "vpoto", using a round-robin mechanism. Next, it stores the obtained VPI into the "vp7" register. Finally, it assigns 1 to the "vp7d" signal. The use of "vp7d", causes the "vp7i" signal to be sent to controller "STATE7" during the next network clock cycle, when in turn, it is assigned 1. This is required, since controller "STATE3" is simulated before controller "STATE7".
Otherwise, when controller "STATE3" receives the "sync23" signal sent by controller "STATE2", it gets a network access VPI to the destination AAL unit indicated by the "dstpn" register from the appropriate routing table,

using a round-robin mechanism. Next, it stores the obtained VPI into the "vp4" register and sends the "vp4i" signal to controller "STATE4".
Appendix E.5 shows a graph giving a more accurate description of controller "STATE3".
When controller "STATE4" receives the "vp4i" signal sent by controller "STATE3", it synthesizes a 5-byte ATM header upon the "vp4" register's content. Next, it stores the resulting header in the "h" buffer. Finally, it sends the "hi" signal to controller "STATE2".
Appendix E.6 shows a graph giving a more accurate description of controller "STATE4".
When controller "STATE5" receives the "sync75" signal sent by controller "STATE7", then it transmits the acknowledge receipt currently contained into the 53-byte "ackbuf" block.
Otherwise, it goes through the "send" buffer's busy blocks chained list, and for each encountered block, it either frees it if the corresponding "ogblktofr" table's entry is 1, or transmits it if the corresponding "blktime" table's entry is 0.
Controller "STATE5" uses a bytewise transmission. To transmit a byte, it assigns it "buf" and 1 to "bufi". Hence, the transmitted byte reaches the downstream PPD unit one network clock cycle after, when "buf" and 1 are assigned to the "ogbyte" and "ogbytei" registers, respectively.
While transmitting bytes from a particular block, controller "STATE5" may receive the "*ogsuspend" signal from the downstream PPD unit, due to a lack of space into at least one of the requested destination queues. In this case, it stops the transmission. Next, cycle after cycle, it keeps on trying to retransmit the considered block, until it completes its transmission without having to stop again.
In accordance with what is stated in Section 9.3.1 regarding of the CRC code calculation circuits, controller "STATE5" uses the circuit B) of the Figure 9.13, but it runs 10 times faster than the transmission links speed. The first 8 cycles are used to flying compute the CRC code corresponding to a particular block while transmitting its bytes. Whereas, the last 2 cycles are used to pack the final result contained in the "crcg" register, at the end of the transmitted block.
Appendix E.7 shows a graph giving a more accurate description of controller "STATE5".
Controller "STATE6" receives bytes from the upstream SPD unit. It stores

them to the tail of the "receive" buffer's busy blocks chained list, and flying computes their corresponding CRC codes.
It uses the "count" register to delineate cells. That is, whenever it receives a byte, it increments "count" by one modulo 53. This way, "count" equals 0 (respectively 52) means controller "state6" is waiting for the head (respectively tail) byte of an ATM cell.
After having filled up the tail of the "receive" buffer's busy blocks chained list, controller "STATE6" checks if the recomputed CRC code agrees with the original, and if the destination is right.
If this is the case, it marks the "receive" buffer's tail busy block as not yet belonging to a message by assigning 0 to the corresponding "blkinmsg" table's entry, and allocates the "receive" buffer's head free block, which hence becomes the new "receive" buffer's tail busy block.
Otherwise, it will discards the corrupted or misdelivered cell when it overwrites the tail of the "receive" buffer's busy blocks chained list with the next cell.
it runs "STATE6" uses too the circuit B) of the Figure 9.13, but running 10 times faster than the transmission links speed. The first 8 cycles are used to flying compute the CRC code corresponding to a particular block while receiving its bytes. Whereas, the last 2 cycles are used to eventually allocate the "receive" buffer's head free block, upon the CRC codes agreement and destination rightness.
Appendix E.8 shows a graph giving a more accurate description of controller "STATE6".
Controller "STATE7" goes through the "receive" buffer's busy blocks chained list. At each step, it checks whether or not the current block has been already used to reassemble a message.
If that is the case, it frees it after a reasonable timeout, providing it has been already served to the destination PVM tasks by then.
On the other hand, if the current block has not been already used to reassemble a message, then controller "STATE7" checks whether or not it is an acknowledge receipt.
If that is the case, controller "STATE7" goes through the "send" buffer's busy blocks chained list, looking for the corresponding block.
When it finds it, if the "ackdesc" table's entry corresponding to the found block and acknowledge receipt source AAL unit equals 1, then it assigns 0 to the aforementioned "ackdesc" table's entry, and reduces by one the number

of acknowledge receipts expected for the found block.
But, if the current block is not an acknowledge receipt, then controller "STATE7" starts to set up the acknowledge receipt to be sent back to the source AAL unit.
Then, it sends the "sync73" signal to controller "STATE3", requesting a network access VPI for a point-to-point transmission towards the source AAL unit, and blocks until it receives the "vp7i" signal.
Thereafter, it ends setting up the started acknowledge receipt.
Next, it sends the "sync75" signal to controller "STATE5", requesting it to transmit the resulting acknowledge receipt.
Finally, it uses the current block to reassemble the message it belongs to, if its sequence number is the expected one from this source AAL unit and for the communication type it is involved in.
Controller "STATE7" disposes of the "msgblkdesc", "blkinmsg", "msglsttab", "msgdesctab", "lbmsg", "fbmsg", "limsg", "fimsg", "msgnumfrm", "lastotasqnfrm", and "lastotosqnfrm" tables, to reassemble messages.

- The "msgblkdesc" table is of 1887 size. It contains the blocks chained list of each message present in the "receive" buffer. In fact, construct such a list means reassemble a message.

- The "blkinmsg" table is of 1887 size. Each entry of this table indicates whether the corresponding "receive" buffer's block has been already used to reassemble a message when set to 1, or not when set to 0.

- The "msglsttab" table is of size 16, since they are 16 possible source AAL units. Each entry of this table consists of 32 message descriptors. Therefore, each destination AAL unit may sustain at most 32 messages from each source AAL unit at a time.
  Figure 9.16 illustrates the format of a message descriptor.

  - The first three fields are assigned the PVM header contained in the message head block.
  - When set to 1, the "complete" field indicates that the message can be served to any PVM task requesting it. It is set to:
    * 1, when controller "STATE7" processes the tail block of the message,

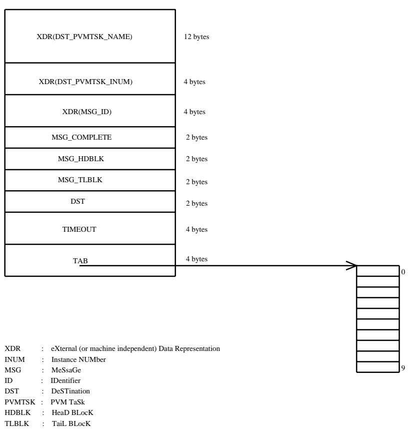


Figure 9.16: Message Descriptor

    * 0, when the message becomes obsolete after a timeout.

- The "hdblk" (respectively "tlblk") field indicates the location of the message head (respectively tail) block on the "msgblkdesc" table.
  The "hdblk" field is assigned the message head block. Whereas, the "tlblk" field is gradually assigned the remaining blocks of the message.
- The "dst" field indicates whether the message is involved in a point-to-point or broadcast communication. It is set up using the SAR header's destination port number field of the message head block.
- When set to 0, the "timeout" field indicates that the message is obsolete. It is initially assigned a positive value, arbitrarily set to 40000 network clock cycles, when controller "STATE7" processes the message tail block. It is thereafter decremented by one at every cycle by controller "STATE9", until it equals 0.
- The "tab" field is a table of size 10, containing the instance numbers list of all the destination PVM tasks to which the message has been already served. It is used to prevent serving a message twice to the same destination PVM task.

- The "msgdesctab" table is of size 16. Each entry contains one busy message descriptors chained list, and one free message descriptors chained list. These lists are used for the corresponding "msglsttab" table's entry management.
- The "fimsg", "limsg", "fbmsg", and "lbmsg" tables are of size 16. Given a particular "msgdesctab" table's entry, their corresponding entries respectively point at the first free, last free, first busy, and last busy message descriptors of the corresponding "msglsttab" table's entry.
- The "msgnumfrm" table is of size 16. Each entry indicates the number of complete and unobsolete messages that have been received from the corresponding source AAL unit.
- The "lastotasqnfrm" table is of size 16. Each entry indicates the expected block sequence number for broadcast transmissions from the corresponding source AAL unit.
- The "lastotasqnfrm" table is of size 16. Each entry indicates the expected block sequence number for point-to-point transmissions from the corresponding source AAL unit.

In this context, if the current block is a message head block, then controller "STATE7" allocates the head free message descriptor of the "msglsttab" table's entry corresponding to the source AAL unit, which then becomes the corresponding new tail busy message descriptor. Then, it sets it up in accordance with what is stated in the message descriptor's fields.
Otherwise, controller "STATE7" does not allocate any message descriptor. It merely goes through the busy message descriptors chained list of the "msglsttab" table's entry corresponding to the source AAL unit, looking for the first incomplete message involved in the type of communication similar to that of the current block.
When it finds such an incomplete message, it inserts the current block at the tail of the found message blocks chained list, and sets up the corresponding message descriptor in accordance with what is stated in the message descriptor's fields.
In both cases, controller "STATE7" assigns 1 to the "blkinmsg" table's entry corresponding to the current block, and increments the corresponding "lastotosqnfrm" (respectively "lastotasqnfrm") table's entry by one modulo 2^{32},

since it deals with a poin-to-point (respectively broadcast) communication.
Appendix E.9 shows a graph giving a more accurate description of controller "STATE7".
Controller "STATE8" is a kind of messages server. When released from its idle state, it goes through the complete and unobsolete messages list from every source AAL unit, looking for one message whose PVM header is compatible with the one that has been provided by a "receive" buffer read requesting PVM task, and stored into the "j", "g", "f", "e", and "d" registers, and that has not been already served to that task.
If controller "STATE8" finds such a message, then it marks it as having been served to the requesting PVM task.
Then, it sets "currcv" to the 16th byte of the head block of the found message, and "icoffset" to 20, hence initializing the base and offset used by the "receive" buffer read mechanism.
Finally, it sends the "read" and "readi" signals towards the host machine.
On the other hand, if it did not find such a message, it only sends the "readi" signal towards the host machine.
Appendix E.10 shows a graph giving a more accurate description of controller "STATE8".
Controller "STATE9" decrements by one all the AAL unit's timers at every network clock cycle.
Appendix E.11 shows a graph giving a more accurate description of controller "STATE9".
Controller "STATE10" discards the obsolete messages from the "receive" buffer.
Appendix E.12 shows a graph giving a more accurate description of controller "STATE10".

9.3.2.3 The AAL Units' C Data Structures and Implementation

The C data structures that define the AAL units' resources are shown in Appendix E.1. Whereas, the C code that simulates their behavior is presented in Appendix E.13.
An *aal34_scheduler(curaal34)* call simulates the behavior of the AAL unit indicated by *curaal34* during one network clock cycle. In fact, it simulates the behavior of the "STATE5", "STATE4", "STATE3", "STATE2", "STATE1", "STATE10", "STATE9", "STATE8", "STATE7", "STATE6",

and "STATE0" controllers, in that order. Moreover, it cannot change the internal states of these controllers more than once, since they run at the same speed as the transmission links, and hence the network.

9.4 The Network Prototype

As illustrated in Figure 9.17, the ATM network prototype described in the previous sections of this chapter, can interconnect up to 16 hosts machines. It consists of 16 AAL34 units at the AAL layer, 48 PPD and 48 SPD units

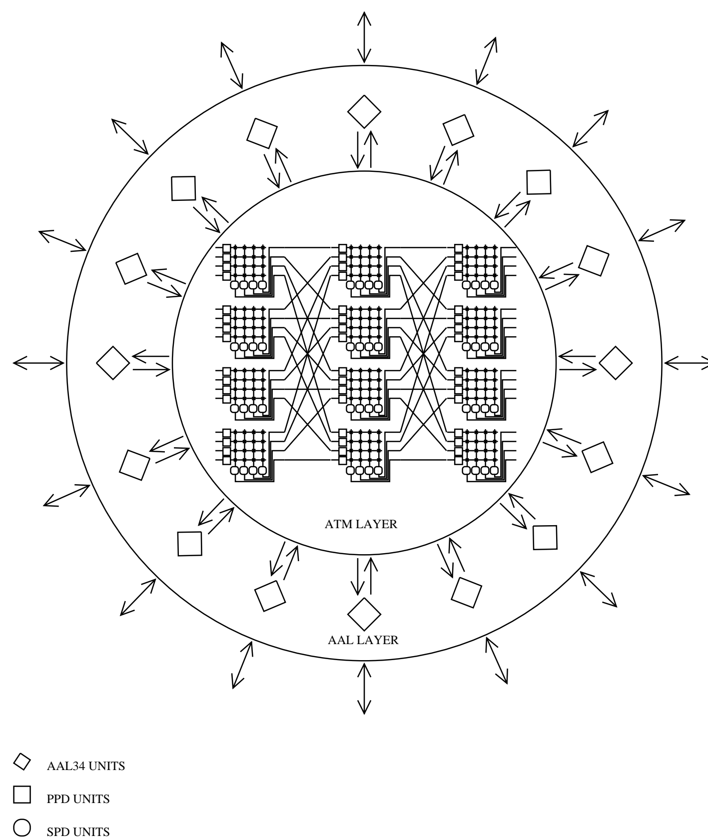


Figure 9.17: The Network Prototype Overview

at the ATM layer.

The kernel of program that simulates the behavior of this network prototype is defined by the following C code:

```
...
struct ppd ppdlst[48];
struct spd spdlst[48];
struct aal3_4 aal34lst[16];
...
for( ; ; )
{
  for (i=0; i<48; i++) spd_scheduler(&spdlst[i]);
  for (i=0; i<16; i++) aal34_scheduler(&aal34lst[i]);
  for (i=0; i<48; i++) ppd_scheduler(&ppdlst[i]);
}
...
```

It is an infinite loop whose body consists of three finite loops. The first (respectively second and third) finite loop consists of 48 (respectively 16 and 48) iterations. Each iteration simulates the behavior of a different SPD (respectively AAL34 and PPD) unit during one network clock cycle.
In this context, one iteration of the infinite loop can be considered as one network clock cycle.

9.5 The Connectionless Data Service

The ATM adaptation layer of the prototype provides a connectionless data service in accordance with what is stated in Section 8.5.4. That is, each source AAL unit uses a set of predefined paths to send its messages to their destination AAL units.
Moreover, cells which belong to the same message may take different routes to reach the destination AAL unit. To take advantage of that feature, a source AAL unit transmits the "send" buffer's busy blocks which belong to the same message over the existing four different routes to the destination AAL unit (See Section 8.4.4), using a round-robin mechanism.

Chapter 10

The PVM Platform

Early in Chapter 9, I mentioned the need to simulate the network prototype directly under a workload which is dynamically generated while running a real distributed application. This is simply because it is more realistic.
Provided that conventional VLSI design tools do not allow to meet this objective, and a workload synthesized from the traces of the execution of a real distributed application over a parallel machine is not truly realistic for the simulation of a different interconnection network, it became nessary to first find a programming interface allowing to write and run distributed application, and implement it over the network prototype, in order to allow the processes from applications plug themselves in the input and output ports of the latter, and communicate each with other via it.
The Parallel Virtual Machine (PVM) [42, 43] developed at the Oak Ridge National Laboratory (ORNL) under the auspices of the Faculty Research Program of Oak Ridge Associated Universities, and the Message Passing Interface (MPI) [44] developed at the University of Tennessee Knoxville, are two emerging distributed Application Programming Interface (API) standard proposals.
They both allow the use of a collection of scalar, vectorial, multiprocessor, or even special-purpose computers interconnected by one or more networks, to be use as a single computational resource. They both also provide C and Fortran libraries allowing to run fully general Multiple Instruction Multiple Data (MIMD) applications, as well as those written in the more restricted style of Single Program Multiple Data (SPMD).

10.1 PVM

The PVM system consists of two sides:

- the User Library
- the Daemon

10.1.1 The User Library

This side of the PVM system is a message passing library. It provides not only routines that deal with groups of tasks such as barrier synchronization, but also and particularly routines that allow PVM tasks to communicate with each other.
Under PVM, tasks are identified by their names and instance numbers, whereas messages are discriminated by their identification numbers.
To receive a message, the destination PVM task:

1. examines its receive buffer to see if the message has arrived and is ready for reading, by a call to either a blocking receive using the pvm_recv() routine, or a non-blocking receive using the pvm_probe() and pvm_recv() routines,
2. unpacks the message from its receive buffer, when the latter is ready for reading, by a number of calls to the pvm_unpk*() or pvm_get*() routines.

On the other hand, to send a message the source PVM task:

1. initializes a send buffer by a call to the pvm_initsend() or pvm_mkbuf() routine,
2. packs the message into the initialized buffer using a number of calls to the pvm_pk*() or pvm_put*() routines,
3. sends the packed message by a call to the pvm_send() or pvm_cast() (multicasting) routine.

In addition, a message may be sent in one of the two following modes:

- PVM Advise. In this mode, the message is directly sent through a TCP connection established between the source and destination PVM tasks.
- PVM Normal. In this mode, the source PVM task first sends the message to the local PVM daemon process via a TCP connection. Then, the latter forwards it to the remote PVM daemon process via a UDP connection. Finally, the remote PVM daemon process forwards the message to the destination PVM task via a TCP connection.
  The UDP connections between the PVM daemon processes are established during the virtual machine setup. Whereas, each PVM task create a TCP connection with its local PVM daemon process, when it enrols into the PVM system.

Therefore, the PVM Advise mode only requires one TCP connection, whereas the PVM Normal mode requires two TCP and two UDP connections, for a bi-directional communication between two PVM tasks (See Figure 10.1). The

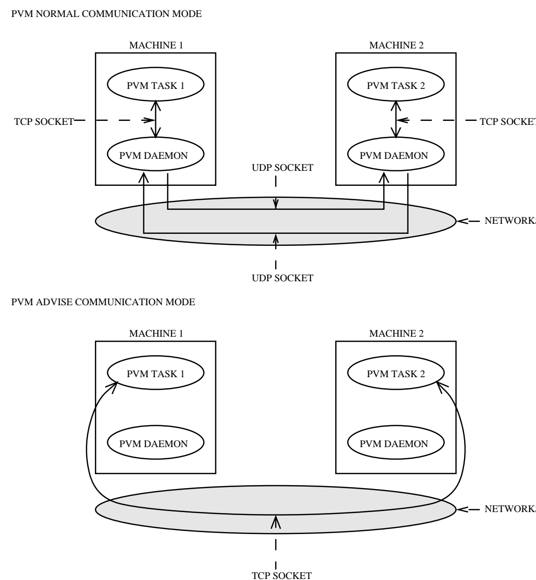


Figure 10.1: PVM Advise and Normal Communication Modes

PVM Advise mode provides a more efficient communication path than the PVM Normal mode. However, the drawback of the PVM Advise mode is the

small number of simultaneous direct TCP connections sustainable by some UNIX systems, which makes their use unscalable.

10.1.2 The Daemon Side

This side of the PVM system, allows a heterogeneous network of serial, parallel, and vectorial computers to be used as a single parallel and distributed-memory computer.
Under PVM, a per-user distributed environment must be set up before running PVM applications. Indeed, every user may customize its virtual machine by providing a configuration file to the system. This file contains the list of computers that make up the user's virtual machine, and set of setup information.
A PVM daemon process runs on each of the virtual machine participating computers, and is used to exchange network configuration information.

10.2 MPI

The MPI system has been designed to efficiently implement the message passing paradigm on a wide range of computers, especially parallel and distributed computers.
MPI makes use of the most attractive features of a number of existing message passing systems such as CHIMP [45, 46], PVM [47, 48], Chameleon [49], PICL [50], p4 [52, 53], and Zipcode [54, 55]. It has been also strongly influenced by PARMACS [56, 57], Intel's NX/2 [58], Express [59], nCUBE's Vertex [60], and work at the IBM T. J. Watson Research Center [61, 62].
Under MPI, processes are organized into ordered groups. And, every process is always identified by its relative rank in a group.
There are two communication types within a process group:

- point-to-point, where only two processes are involved,
- collective, where all processes are involved.

Only point-to-point communications are allowed between processes from disjoint groups.
Point-to-point and collective communications in a group happen within two

separate contexts. And, point-to-point communications between two disjoint groups, happen within a context that is different from the two local contexts used in each of these groups.
This way, the collective, local point-to-point, and remote point-to-point communications related to a same group never interfere. Indeed, messages are always received within the context they were sent.
Likewise, contexts prevent communications from two different groups from interfering.
Every group is associated with its point-to-point and collective communication contexts in a communicator. There are two kinds of communicators:

- Intracommunicators for intracommunications. An intracommunication is either a point-to-point or a collective communication between processes within a single group.

- Intercommunicators for intercommunications. An intercommunication is a point-to-point communication between two processes from two different groups. The group containing the process that initiates an intercommunication operation is called the "local group", that is, the sender in a send and the receiver in a receive. The group containing the destination process is called the "remote group".

Intracommunications and intercommunications use (rank, communicator) pairs to identify the destination processes. In such a pair, the rank component indicates the rank of the destination process within the unique group (respectively "remote group") of the communicator indicated by the communicator component, if the latter is an intracommunicator (respectively intercommunicator).
In both intracommunications and intercommunications, processes communicate with one another through explicit messages. Basically, messages are sent and received via calls to MPI send, receive, and broadcast communication primitives.

10.2.1 The MPI Send Communication Primitives

The MPI send primitives are used to send messages, only in point-to-point communications. There are two types of MPI send primitives:

- blocking

- nonblocking

A call to a blocking send does not return until the message has been safely stored away, either directly into the matching receive buffer or into a temporary system buffer, so that the calling process is free to access and overwrite the send buffer.
On the other hand, a call to a nonblocking send initiates a send operation, but does not complete it. Such a call returns before the message was copied out of the send buffer. A separate send call is needed to complete the initiated send.
Both blocking and nonblocking sends may use the following four communication modes:

- Standard. In this mode, MPI may not buffer an outgoing message either due to a lack of available buffer space, or for performance reasons. In this case, a send call will not return until the matching receive has been posted, and the message has been moved to the receive buffer. Thus, the standard mode send is nonlocal. That is, its successful completion may depend on the occurrence of a matching receive.

- Buffered. In this mode, MPI copies every outgoing message into a temporary system buffer. In this case, a buffered mode send may start and/or complete before the matching receive is posted. A send executed in this mode is local. Indeed, its successful completion does not require any communication with another user process.

- Synchronous. In this mode, there is normally no message buffering. A synchronous mode send may start before the matching receive is posted. But, it will successfully complete when the matching receive is posted, and has started to receive the message. In this way, a send executed in this mode is nonlocal.

- Ready. In this mode, a send may be started only if the matching receive is already posted. So, a send executed in this mode is nonlocal.

Every call to an MPI send is of the form:

mpi_*send(sbuf, count, datatype, dst, tag, comm)

Such a call specifies the send buffer in the calling process' memory by the first three parameters. That is, the specified send buffer consists of *count* successive elements of the type indicated by *datatype*, starting at the address *sbuf*.
The basic types that can be indicated by datatype are presented in Table 10.1. The use of one of these types leads to the specification of a send buffer, that

MPI datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_CHARACTER		CHARACTER(1)
MPI_SHORT	signed short int	
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_INT	signed int	
MPI_UNSIGNED	unsigned int	
MPI_INTERGER		INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	
MPI_REAL		REAL
MPI_DOUBLE	double	
MPI_DOUBLE_PRECISION		DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	
MPI_COMPLEX		COMPLEX
MPI_LOGICAL		LOGICAL
MPI_BYTE		
MPI_PACKED		

Table 10.1: Basic MPI Datatypes

is contiguous and contains a sequence of elements of the same type.
These basic types are not suitable when one wants to send a message that contains data of different types, and when one wants to send noncontiguous data (e.g. a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site, and unpack it at the receiver site. Unfortunately, this solution has the disadvantage to require additional memory-to-memory copies at both sites.
To avoid these additional memory-to-memory copies, data must be collected directly from the site where they reside. To achieve that, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. Such buffers are specified by replacing the basic types indicated by datatype with derived types.
A *derived* (also *moregeneral*) type is constructed from basic types. Indeed,

a general type consists of a number of $(basictype, displacement)$ pairs. A sequence of such pairs are called *typemap*. And, the sequence of the basic types in a *typemap* is called *typesignature*.
Let

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

be a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Therefore,

$$Typesig = \{type_0, ..., type_{n-1}\}$$

is the associated type signature.
This *typemap*, together with a base address $sbuf$, specifies a communication buffer that consists of n elements, where the i-th element is at the address $sbuf + disp_i$ and is of type $type_i$. A message assembled from such a communication buffer consists of n elements of the type defined by $Typesig$.
In addition to the data, a message carries additional information that allows to identify and selectively receive it. This information is called the *message envelope*, and consists of the following fields:

- source
- destination
- tag
- communicator

The message source is implicitly determined by the identity of the calling process. The remaining fields are specified by the last three parameters. The dst (respectively tag) specifies the message destination (respectively tag), whereas the comm argument specifies the communicator used to send the message.

10.2.2 The MPI Receive Communication Primitives

The MPI receive primitives are used to receive messages, only in point-to-point communications. There are two types of MPI receive primitives:

- blocking
- nonblocking

A call to a blocking receive returns only after the receive buffer contains the incoming message. On the other hand, A call to a nonblocking receive initiates a receive operation, but does not complete it. Such a call returns before the message is stored into the receive buffer. A separate receive call is needed to complete the initiated receive.
Unlike MPI sends, neither blocking nor nonblocking MPI receives do not use the standard, buffered, synchronous, and ready communication modes described in Section 10.2.1.
In addition, blocking receives can match with nonblocking sends, and vice-versa.
Every call to an MPI receive is of the form:

mpi_recv(rbuf, count, datatype, src, tag, comm, status)

Such a call specifies the receive buffer in the calling process' memory, where a selected message is stored, by the first three parameters. That is, the specified receive buffer consists of *count* successive elements of the type indicated by *datatype*, starting at the address $rbuf$.
Both *basic* and *derived* types (See Section 10.2.1) are allowed for *datatype.*
The selection of a message by a receive call is based upon the content of that message envelope. Indeed, a message can be received by a receive call if its envelope matches the *src*, *tag*, and *comm* parameters specified by the call.

10.2.3 The MPI Broadcast Communication Primitive

The MPI broadcast communication primitive is used to send and receive messages, only in collective communications. The syntax of a broadcast call is:

mpi_bcast(buf, count, datatype, root, comm)

Such a call broadcasts a message from the process with rank *root* to all the processes of the group indicated by *comm*, itself included. The broadcast primitive must be called by all the processes of the group using the same arguments for *comm*, and *root*.
The first three parameters specify a send or receive buffer in the calling process' memory, depending on whether this calling process is the broadcast root or not.
Both *basic* and *derived* types (See Section 10.2.1) are allowed for *datatype.*

10.3 The Platform Itself

Although MPI provides a wider communication primitives set than PVM, and even if MPI has the *derived* data type which allows to transfer noncontiguous data in straight, hence reducing the number of memory-to-memory copies during communication operations, whereas PVM does not, I implemented a PVM platform simply because MPI was not officially released when I started my Ph.D. late 1993.

The objective is to enable the processes of PVM applications to plug themselves in a process, which emulates the interconnection network implementation under test, and communicate each with another via it.

On one hand, we saw in Section 9.3.2.1, that the processes of PVM applications and the network process communicate by means of shared variables.

That is, the network process first creates a shared memory segment, including variables implementing the 32-bit bus of the host machine and few control signals between the host machine and an AAL unit, for each of its input/output port at the setup time.

Then, the processes of PVM applications bind themselves to the input/-output ports of the network process by getting the corresponding shared memory segments.

On the other hand, the standard communication primitives of the PVM user library copy data from the memories of the calling processes into UDP or TCP sockets, hence causing communications to happen via Unix or Ethernet, rather than the network process.

In this context, it was sufficient to rewrite these communication primitives in a way to have them copy data from the memories of the calling processes into the shared memory segments.

Table 10.2 gives the synopsis of the C communication primitives that I rewrote.

Their syntax and semantic were defined to be consistent with that of their PVM counterparts. This is to achieve a minimum of changes, of the syntactic nature essentially, over already existing PVM applications in order to run them over the platform.

However, the buffers used to send and receive messages are located in the AAL units rather than the memories of the calling processes. This is to avoid the additional memory-to-memory copies that are otherwise required, as it is the case with PVM.

Communication Primitives Synopsis
atminitsend(dev) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ - initializes the send buffer of the AAL unit indicated by *dev*
atmputn*type***(dev, np, cnt)** **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ *type* ***np;** **int cnt;** - packs *cnt* successive elements, of the type indicated by *type*, from the calling process' - memory starting at the address *np* into the send buffer of the AAL unit indicated by *dev* - *type* must be int, or short, or long, or float, or dfloat, or cplx, or dcplx
atmputbytes(dev, np, cnt) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** **int cnt;** - packs *cnt* successive characters from the calling process' memory starting at - the address *np* into the send buffer of the AAL unit indicated by *dev*
atmputstring(dev, np) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** - packs the character string from the calling process' memory starting at the address *np* - into the send buffer of the AAL unit indicated by *dev*
atmgetn*type***(dev, np, cnt)** **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ *type* ***np;** **int cnt;** - unpacks *cnt* successive elements of the type indicated by *type* - from the receive buffer of the AAL unit indicated by *dev*, - and stores them to the calling process' memory starting at the address *np* - *type* must be int, or short, or long, or float, or dfloat, or cplx, or dcplx
atmgetbytes(dev, np, cnt) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** **int cnt;** - unpacks *cnt* successive characters from the receive buffer of the AAL unit indicated - by *dev*, and stores them to the calling process' memory starting at the address *np*
atmgetstring(dev, np) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **char *np;** - unpacks a character string from the receive buffer of the AAL unit indicated by *dev*, - and stores it into the calling process' memory starting at the address *np*
atmsnd(dev, prcstab, proc, inum, type) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **struct prcsdescstr *prcstab;** /* reference to the process description table */ **char proc[3];** /* destination proces' 3-length name. */ **int inum;** /* destination process' instance number */ **int type;** /* message type */ - grants to the AAL unit indicated by *dev*, the permission to sends the last message packed into - its send buffer to any process whose name, instance number, and expected message type match - the last three parameters of the call. Note that, $type \geq 0$
atmrcv(dev, type) **struct shmst *dev;** /* reference to the shared memory segment of the network I/O port to which the calling process is bound to */ **int type;** /* message type */ - selects a message on the receive buffer of the AAL unit indicated by *dev*, whose PVM header - matches the name, instance number, and expected message type specified by the calling process - This routine is blocking (i.e. it does not return until it selects a matching message)

Table 10.2: C Communication Primitives Synopsis

Like in the PVM Advise mode, the rewritten routines provide direct accesses to the send and receive buffers. Calls to these routines send commands, along with data eventually, directly to the AAL units, hence bypassing the local PVM daemons. The commands tell the AAL units which operations to perform (See the description of AAL unit's controller "STATE0" component in Section 9.3.2.2).

A message send operation, with the rewritten routines, starts with a call to the *atminitsend*() routine. Then, it proceeds with a number of calls to the *atmput*∗() routines. Finally, it ends with a call to the *atmsnd*() routine.

On the other hand, a message receive operation, with the rewritten routines, starts with a call to the *atmrcv*() routine. Then, it ends with a number of calls to the *atmget*∗() routines.

The sequence of the calls of any message send or receive operation must be atomic. This is to provide a mutual exclusion accesses in read and/or write to the AAL units, that prohibit the interference of different message reads and writes on the same AAL unit.

A way to implement these atomic call sequences, is to lock (respectively unlock) the AAL units at the beginning (respectively end) of every message send or receive operation.

Regarding of the AAL units locking, every call to the *atminitsend*() routine which always starts a message send operation, or to the *atmrcv*() routine which always starts a message receive operation, blocks the calling process until the AAL unit, to which that process is bound with, is unlocked. Then, it locks it, and lets the calling process proceeds.

Regarding of the AAL units unlocking, things are slightly different. On the one hand, every message receive operation ends with a variable number of calls to the *atmget*∗() routines. However, the return of the last *atmget*∗() call indicates that the received message has been safely stored into the memory of the calling process. Then, the latter may unlock the AAL unit to which it is bound with, by running the $dev{-}{>}rlcked = 0$ statement, providing that *dev* points to the memory segment shared by the calling and network processes.

On the other hand, every message send operation ends with a call to the *atmsnd*() routine. However, a call to this routine may return before the AAL unit, that is bound with the calling process, completes the segmentation of the message that has been just entrusted to it. In this case, it is that AAL unit which unlocks itself after it is done with the aforementioned segmentation, by setting the $dev{-}{>}wlcked$ signal to 0, providing that *dev*

points to the memory segment shared by the calling and network processes. The rewritten routines has been added into the crunch.c PVM file, to be included into the libpvm.a PVM user library. They may also be compiled out of PVM to provide a separate user library that allows non PVM processes to communicate via the network prototype.

Chapter 11

Tests and Experiments

This chapter presents the simulations that I performed to test the operation of the target network prototype presented in Chapter 3, and the platform presented in Chapter 4. Next, it presents the evaluation of few performances of this network prototype. Finally, it presents the evaluation of the impact of the contentions within this network prototype over the execution on the platform of a distributed application.

11.1 Network Prototype Operation Tests

This section presents the tests of:

- a Primary Packet Distributor (PPD) unit
- a Secondary Packet Distributor (SPD) unit
- an upstream SPD and downstream PPD unit tandem
- a Bus Matrix Switch (BMX)
- an AAL unit

These tests were performed off the PVM platform. This is due to the following facts:

- In the four former cases, the execution of a real distributed application is not necessary to test the operation of the units under test.

- In the latter case, although the execution of a real distributed application is used, the network pototype is simulated cycle per cycle in order to close monitor of the target AAL units. This is useful for debugging purposes.

Therefore, a different simulation platform was built for each of the aforementioned test case.
The simulation platforms for the tests of a PPD unit, an SPD unit, an SPD-PPD tandem, and a BMX switch, use a classical approach. Indeed, the corresponding C simulation programs that I wrote take two arguments on their command lines. The first argument indicates the input and output stimuli file, whereas the second argument specifies the traces file. Figure 11.1 presents the stimuli and traces file formats.

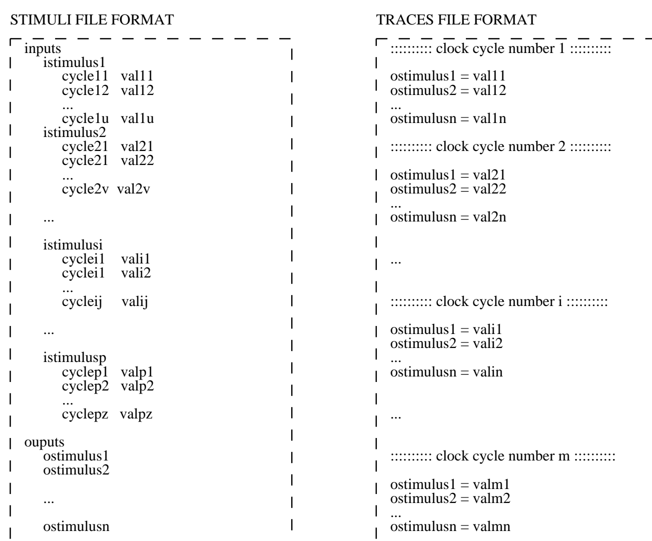


Figure 11.1: Stimuli and Traces Files Formats

Initially, a simulation program reads the stimuli file and builds the corresponding input and output stimuli chained lists, whose structures are presented in Figure 11.2.
An input stimulus is a data structure with four fields including:

- its name

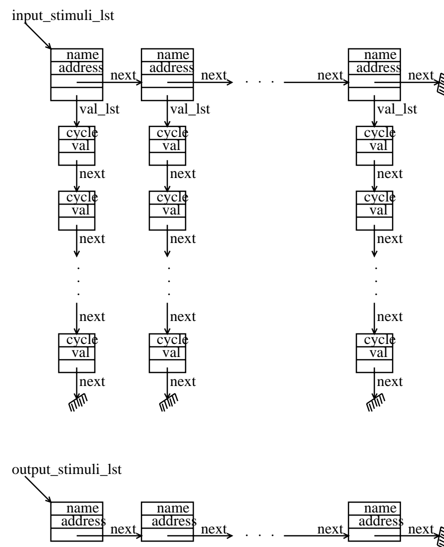


Figure 11.2: Input and Output Stimuli Chained Lists Structures

- its address in the memory of the simulation test program
- a pointer to the next input stimulus
- the chained list of its successive values

An input stimulus' value is a data structure with three fields including:

- the clock cycle number at which it must be assigned
- the value itself
- a pointer to the next input stimulus' value

An output stimulus is a data structure with three fields including:

- its name
- its address in the memory of the simulation test program
- a pointer to the next output stimulus

Next, that simulation program enters in an infinite loop, numbering its iterations. At each iteration:

- It first goes through the input stimuli chained list. And, for each input stimulus, it goes through its values chained list until it reaches either the tail or a value whose assignation time is greater or equal to that iteration number. In the latter case, it assigns the value to the current input stimulus in the equality case.
- Next, it runs the programs that simulates the behavior of the units under test during one network clock cycle.
- Finally, it first prints a short message including the current iteration number on the screen and into the traces file. Then, it goes through the the output stimuli chained list, and prints the current value of every output stimulus on the screen and into the traces file.

Users may break the execution of the simulation programs by typing CTRL-C on the keyboard.
The simulation platform of the tests of the AAL units uses a different approach. It consists of:

- the network prototype,
- a non PVM source process plugged in an input port (hence a source AAL unit) of the network prototype,
- a non PVM destination process plugged in an output port (hence a destination AAL unit) of the network prototype.

A program simulating the network prototype is required to allow the source and destination processes to communicate each with other via the network prototype, hence generating on-line the workload of the source and destination AAL units.
The *aal34_schd_tst* program that I wrote to achieve that, is an infinite loop interacting with the user. At each iteration, it first display a message prompting the user to indicate whether or not he wants to run an extra iteration. If he indicates that he does not want an extra iteration, then the program ends. Otherwise, the program first simulates the behavior of the network prototype during one clock cycle, then it prints a short message including

the current iteration number, followed by the current values of the source and destination AAL units' registers specified in the program by the user.
A simulation test of a source and destination AAL unit pair is started by running the *aal34_schd_tst* program, a source process, and a destination process, in this order.
During this simulation test, the source process sends messages to the destination process via the network prototype. As soon as the destination process receives these messages, it prints their contents and their receipt clock cycle numbers on the screen. This is to provide a preliminary indication on the correctness of not only the AAL units operation, but also the network prototype access PVM routines at the application level.
Further more, the *aal34_schd_tst* program provides detail indications on the AAL units operation. While simulating the network prototype (See Section 9.4), it monitors the source AAL unit to provide details on how AAL units:

- segment messages from application processes into ATM cells on their send buffers
- transmit ATM cells from their send buffers
- discard ATM cells from their send buffers upon receipts of acknowledgments.

And, it monitors the destination AAL unit to provide details on how AAL units:

- receive ATM cells from ATM switches into their receive buffers
- reassemble ATM cells from their receive buffers into messages
- serve the reassembled messages to application processes

The *aal34_schd_tst* program monitors the source and destination AAL units, by printing the successive contents of their internal registers into a traces file, whose format is shown in Figure 11.1.

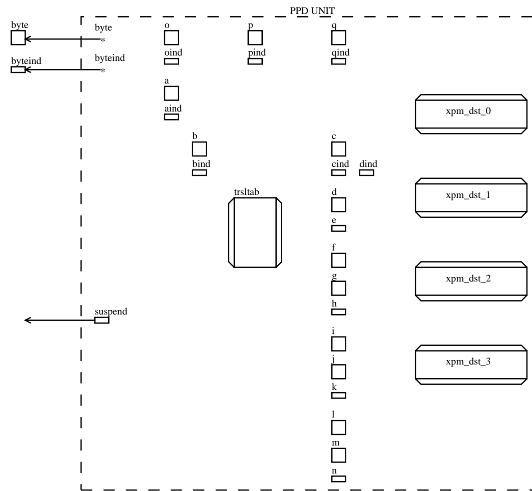


Figure 11.3: PPD Unit Test Platform

11.1.1 PPD Unit Simulation Tests

Figure 11.3 indicates the resources that are used to build the simulation platform of the tests of a PPD unit.
This platform consists of:

- an asynchronous transmission link (See Section 9.1), here implemented by the *byte* and *byteind* signals
- a target PPD unit

On such a platform, the cells written over the asynchronous transmission link are read and stored, if possible, into the destination queues specified in their headers, by the target PPD unit.
The simulation program for this platform is called: *ppdtst*. Therefore, a simulation test of a PPD unit is started by running the command line:

ppdtst stimuli_file traces_file

The different stimuli files that I passed to the *ppdtst* program correspond each to a particular test:

- I used the stimuli file shown in Appendix F.1 for the test of the header translation and successful routing of ATM cells.
  This file indicates that an ATM cell arrives at the target PPD unit at clock cycle number 1 for a broadcast routing to all the destination queues, whereas another ATM cell arrives at it at clock cycle number 58 for a point-to-point routing to destination queue number 2.
  Extracts from the traces file resulting from this simulation are shown in Appendix F.2. They show the states of the target PPD unit's destination queues, as well as its registers.

  - During the clock cycles number 1 to 5, the target PPD unit translates the VPI field of the first incoming ATM cell's header, and stores the new VPI into all its destination queues.
  - During the clock cycles number 6 to 56, it runs as a three stages pipeline operator, with the o p and q registers implementing the input middle and output stages respectively, to write the remaining bytes of this first incoming ATM cell into all its destination queues.
  - During the clock cycles number 58 to 113, it repeats an identical header translation and successful routing process for the second incoming ATM cell.

- I used the stimuli file shown in Appendix F.3 for the test of the header translation and routing suspension of ATM cells.
  This file indicates that an ATM cell arrives at the target PPD unit at clock cycle number 1 for a broadcast routing to all its destination queues, and another ATM cell arrives at it at clock cycle number 6 for a point-to-point routing to destination queue number 3.
  During this simulation, the target PPD unit's destination queue number 3 does not have enough space to store an additional ATM cell
  Extracts from the traces file resulting from this simulation are shown in Appendix F.4. They show the states of the target PPD unit's destination queues, as well as its registers.

  - During the clock cycles number 1 to 3, the target PPD unit translates the VPI field of the first incoming ATM cell.

- During the clock cycle number 4, it generates a suspension signal rather than writing the new VPI into all its destination queues.
- During the clock cycles number 4 and 5, it discards not only the new VPI, but also the third, fourth, and fifth bytes of this first incoming ATM cell. This is why its *count* register is set to *zero* during the clock cycle number 5.
- During the clock cycles number 6 to 10, it repeats an identical header translation and routing suspension process for the second incoming ATM cell.

- I used the stimuli file shown in Appendix F.5 for the test of the chain of the header translation and routing suspension, and the header translation and successful routing of ATM cells.
  This file indicates that an ATM cell arrives at the target PPD unit at clock cycle number 1 for a broadcast routing to all its destination queues, and another ATM cell arrives at it at clock cycle number 6 for a point-to-point routing to destination queue number 2.
  During this simulation, only the target PPD unit's destination queue number 3 does not have enough space to store an additional ATM cell Extracts from the traces file resulting from this simulation are shown in Appendix F.4. They show the states of the target PPD unit's destination queues, as well as its registers.

  - During the clock cycles number 1 to 3, the target PPD unit translates the VPI field of the first incoming ATM cell.
  - During the clock cycle number 4, it generates a suspension signal rather than writing the new VPI into all its destination queues.
  - During the clock cycles number 4 and 5, it discards not only the new VPI, but also the third, fourth, and fifth bytes of this first incoming ATM cell. This is why its *count* register is set to *zero* during the clock cycle number 5.
  - During the clock cycles number 6 to 10, it translates the VPI field of the second incoming ATM cell's header, and stores the new VPI into destination queue number 2.
  - During the clock cycles number 11 to 61, it runs as a three stages pipeline operator, with the o p and q registers implementing the

input middle and output stages respectively, to write the remaining bytes of this second incoming ATM cell into destination queue number 2.

11.1.2 SPD Unit Simulation Tests

Figure 11.4 indicates the resources that are used to build the simulation platform of the tests of an SPD unit.

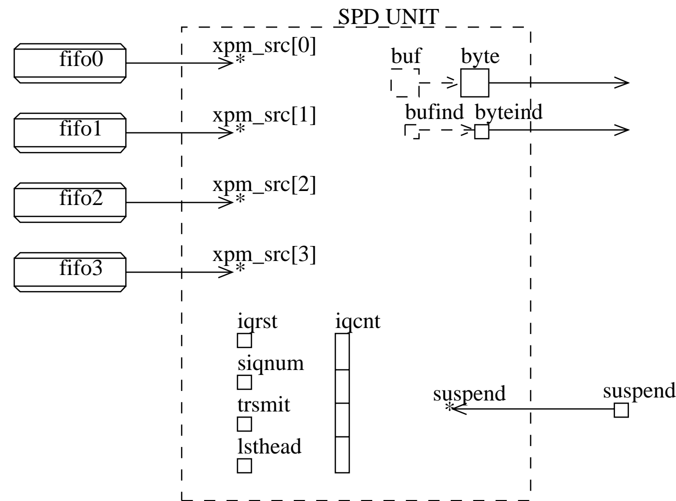


Figure 11.4: SPD Unit Test Platform

This platform consists of:

- four input queues
- a suspend signal
- a target SPD unit

On such a platform, the cells buffered into the input queues, are extracted by the target SPD unit and transmitted over its outgoing link. The target SPD unit stops the current transmission upon the receipt of a suspend signal.
The simulation program for this platform is called: *spdtst*. Therefore, a simulation test of an SPD unit is started by running the command line:

spdtst stimuli_file traces_file

The different stimuli files that I passed to the *spdtst* program correspond each to a particular test:

- I used the stimuli file shown in Appendix G.1 for the test of the successful transmission of ATM cells.
  This file indicates the arrival of a first ATM cell at the input queue number 0 starting from clock cycle number 1, second ATM cell at the input queue number 0 starting from the clock cycle number 54, third ATM cell at the input queue number 2 starting from the clock cycle number 60, and fourth ATM cell at the input queue number 1 starting from the clock cycle number 108. This stimuli file triggers the simulation of the chain of two consecutive ATM cell transmissions from the same input queue, then the switch to an ATM cell transmission from a second, then third input queue.
  Extracts from the traces file resulting from this simulation are shown in Appendix G.2. From the clock cycle number 2 to 54, it shows the transmission of the input queue number 0 head cell. Then, from the clock cycle number 55 to 107, it shows the transmission of the input queue number 0 new head cell. Next, from the clock cycle number 108 to 160, it shows the transmission of the input queue number 2 head cell. Finally, from the clock cycle number 161 to 213, it shows the transmission of the input queue number 1 head cell.

- I used the stimuli file shown in Appendix G.3 for the test of the suspension of the ATM cell transmissions.
  This file indicates that an ATM cell arrives at the input queue number 0 at the clock cycle number 1, and two others arrive simultaneously at the input queues number 1 and 2 at the clock cycle number 8. In addition, the target SPD unit will suspend its current transmission during the clock cycles number 7, 13, 19, and 25.
  The traces file resulting from this simulation is shown in Appendix G.4. When the target SPD unit suspends its current transmission at the beginning of the clock cycle number 7, only the input queue number 0 is non-empty. Therefore, starting from the clock cycle number 8, it proceeds with the transmission of the head block of that same input queue. But, when it suspends the latter transmission at the beginning

of the clock cycle number 13, the input queue number 1 is non-empty. Consequently, starting from the clock cycle number 14, it proceeds with the transmission of the head cell of the input queue number 1. Likewise, it proceeds with the transmission of the head cell from the input queue number 2 starting the clock cycle number 20, after have suspend its previous transmission at the clock cycle number 19.

- I used the stimuli file shown in Appendix G.5 for the test of the transmission suspension and successful transmission chain.
  This file indicates that an ATM cell arrives at the input queue number 0 starting from the clock cycle number 1, and two others arrive simultaneously at the input queues number 1 and 2 starting from the clock cycle number 8. It also indicates that the target SPD unit will suspend its transmissions during the clock cycles number 7 and 66.
  Extracts of the traces file resulting from this simulation are shown in Appendix G.6. After having suspended the transmission of the head cell of the input queue number 0 during the clock cycle number 7, the SPD unit under test proceeds with the transmission the latter cell starting from the clock cycle number 8, since the input queue number 0 was its only non-empty input queue during the clock cycle number 7. Next, it chains with the transmission of the head cell of the input queue number 1 starting from the clock cycle number 61. After having suspended the latter transmission during the clock cycle number 66, it proceeds with the transmission of the head cell of the input queue number 2 starting from the clock cycle number 67, since this input queue was non-empty during the clock cycle number 66.

11.1.3 SPD-PPD Link Simulation Test

Figure 11.5 indicates the resources that are used to build the simulation platform of the operation tests of an upstream SPD unit and a downstream PPD unit tandem.
This platform consists of:

- four input queues
- a target upstream SPD unit

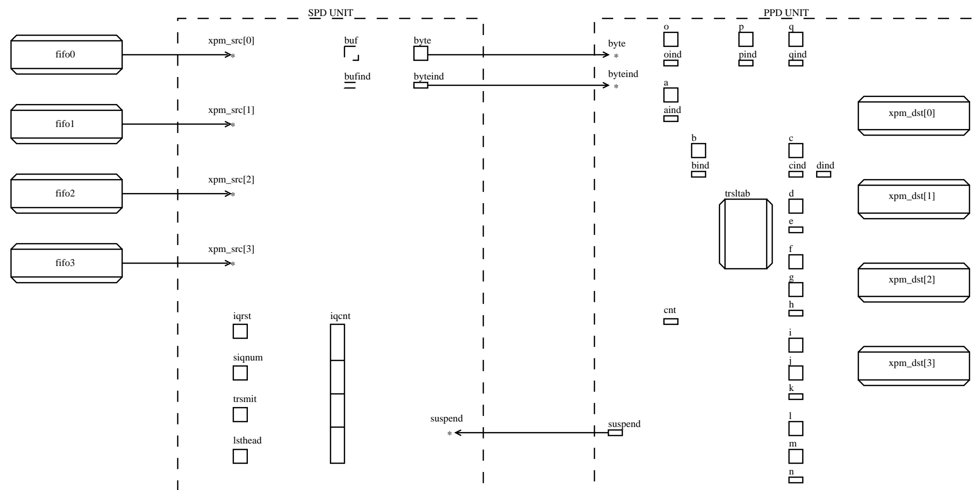


Figure 11.5: SPD-PPD Link Test Platform

- a target downstream PPD unit

On such a platform, the cells buffered into the input queues, are extracted by the target SPD unit and transmitted to the target PPD unit over their interconnection link. The target SPD unit stops the current transmission upon the receipt of a suspend signal from the target PPD unit.
The simulation program for this platform is called: *spdppdtst*. Therefore, a simulation test of such a tandem is started by running the command line:

spdppdtst stimuli_file traces_file

I used the stimuli file shown in Appendix H.1 for the test of the following cases:

- The successful transfer of ATM cells from the target SPD unit' input queues into the target PPD unit' output queues.

- The suspension of the ATM cells transfer from the target SPD unit' input queues into the target PPD unit' output queues.

- The chain of the two previous cases.

This file indicates that, an ATM cell to be routed to the target PPD unit's output queue number 2, arrives into the target SPD unit's input queue number 0 starting from clock cycle number 1. It also indicates that, another ATM cell to be routed to the target PPD unit's output queue number 0, arrives into the target SPD unit's input queue number 2 starting from clock cycle number 1. Finally, it indicates that another ATM cell is to be broadcast to all the target PPD unit's output queues.
Extracts from the traces file resulting from this simulation test are shown in Appendix H.2. The target SPD and PPD units quite transfer the head cell of the former's input queue number 0, into the latter's output queue number 2, starting from clock cycle number 2 until clock cycle number 58. This transfer saturated the aforementioned output queue. Hence, after the target SPD and PPD unit tandem starts the broadcast transfer of the head cell of the target SPD unit's input queue number 2 to all the target PPD unit's output queues during the clock cycle number 55, it has to stop this transfer during the clock cycle number 60. Next, the tandem proceeds with the point-to-point transfer of the head cell of the target SPD unit's input queue number 1 into the target PPD unit's output queue number 0. This transfer starts during the clock cycle number 61, and successfully completes during the clock cycle number 113. While the target PPD unit's output queue number 2 remains saturated, the tandem will keep on starting the aforementioned broadcast transfer and stoping it 6 clock cycles later.

11.1.4 BMX Switch Simulation Tests

Figure 11.6 indicates the resources that are used to build the simulation platform of the operation tests of a BMX switch.
This platform consists of:

- four asynchronous links (See Section 9.1), implemented by four (byte, byteind) variable pairs
- four suspend signals
- four target PPD units
- four target SPD units

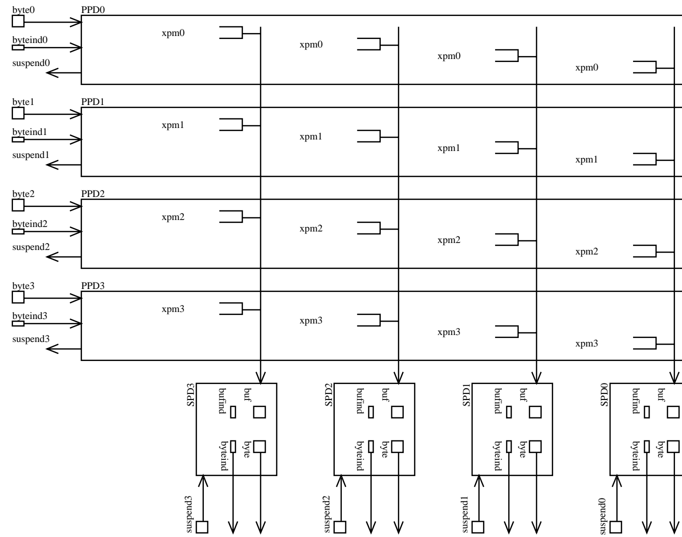


Figure 11.6: BMX Switch Test Platform

On such a platform, the target PPD units read incoming cells from the asynchronous links, translate their headers, and either bufferize them to the specified destination queues if thereis enough free space in them, or generate a congestion indication signal otherwise. On the other hand, the target SPD units extract cells from the destination queues and transmit them over their output transmission link. They stop their current transmissions upon receipts of suspend signals. The simulation program for this platform is called: *bmxtst*. Therefore, a simulation test of an SPD unit is started by running the command line:

bmxtst stimuli_file traces_file

I used the stimuli file shown in Appendix I.1 for the test of the routing of two cells that arrive simultaneously at the input ports of a BMX switch.
This file indicates that both cells arrive at the input ports of the switch under test during the clock cycle number 1. The cell that arrives at the input port number 1, is to be routed to the output port number 1. And, the cell that arrives at the input port number 2, is to be broadcast to all the output ports. During this simulation test, I did not monitor the internal registers of the target PPD and SPD units. I simply scanned their input and output links.

Extracts from the traces file resulting from this simulation test are presented in Appendix I.2. As expected, the cell which arrived at the input port number 1 (respectively 2) during the clock cycle number 1, is transmitted from the output port number 1 (respectively 0, 2, and 3), starting from clock cycle number 6 until clock cycle number 58. Next, the cell which arrived at the input port number 2 during the clock cycle number 1, is transmitted from the output port number 1, starting from clock cycle number 59, until clock cycle number 111.
This simulation test quite confirms, that every BMX switch of the network prototype introduces a 6 clock cycles latency.

11.1.5 AAL Unit Simulation Tests

Figure 11.7 describes the configuration of the simulation platform that I used to test the operation of a source and destination AAL units pair.

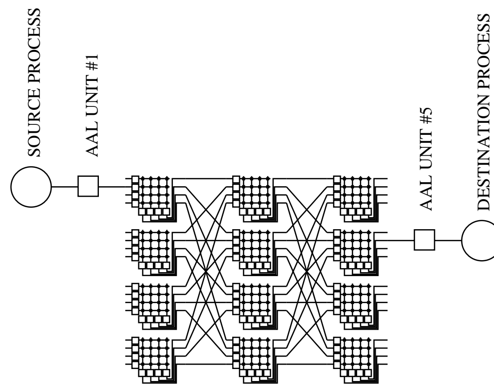


Figure 11.7: AAL Units Test Platform

During this simulation test, the source process bound to the network prototype input port (hence AAL unit) number 1, sends a first message including an integer, then a second message including an integer and a real to the destination process bound to the network prototype output port (hence AAL unit) number 5, via the network prototype.
The source program used on this platform is called *snd*. An extract of its C code is shown next:

```
/* Source Progam C Code Extract */
...
int val1 = 285;
int val2 = 544;
float val3 = 1297.456;
...
(void)atminitsend(dev);
(void)atmputnint(dev, &val1, 1);
(void)atmsnd(dev, prcstab, "rcv", -1, 1);

(void)atminitsend(dev);
(void)atmputnint(dev, &val2, 1);
(void)atmputnfloat(dev, &val3, 1);
(void)atmsnd(dev, prcstab, "rcv", 1, 1);
...
```

And, the destination program used on this platform is called *rcv*. An extract of its C code is shown next:

```
/* Destination Program C Code Extract */
...
int x;
float y;
...
(void)atmrcv(dev, prcstab, 1);
(void)atmgetnint(dev, &x, 1);
dev->rlcked = 0;

printf("\n(cycle = %u, x1 = %d)\n\n", *cycle, x);

(void)atmrcv(dev, prcstab, 1);
(void)atmgetnint(dev, &x, 1);
(void)atmgetnfloat(dev, &y, 1);
dev->rlcked = 0;

printf("\n(cycle = %u, x2 = %d, x3 = %f)\n\n", *cycle, x, y);
...
```

Figure 11.8 provides a screen display history of this simulation test. The values printed on the screen by the destination process *rcv*, are exactly those sent by the source process *snd*.

```
cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~> aal34_schd_tst

:::::::::: clock cycle number 0 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

:::::::::: clock cycle number 1 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

...

:::::::::: clock cycle number i-1 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

:::::::::: clock cycle number i ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

:::::::::: clock cycle number i+1 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

...

:::::::::: clock cycle number 1056 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 1

:::::::::: clock cycle number 1057 ::::::::::
do you wish to run an extra network prototype cycle 1(yes)/0(no)? 0

cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~>
```

```
cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~> snd
cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~>
```

```
cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~> rcv

(cycle = 211, x1 = 285)

(cycle = 376, x2 = 544, x3 = 1297.456)

cime701::/rech/lsr/ondoa/pvm/atm/aal34 ~>
```

Figure 11.8: An AAL Unit Simulation Test's Screen Display History

Extracts of the traces file resulting from this simulation test are shown Appendix J. For example, they indicate that:

- during the clock cycle number 2, the AAL unit number 1 (that is the source AAL unit) begins the segmentation of the first message,
- during the clock cycle number 59, the AAL unit number 1 ends the segmentation of the first message,
- during the clock cycle number 62, the AAL unit number 1 begins the sending of the first message cell, and the segmentation of the second message,

- during the clock cycle number 81, the AAL unit number 5 (that is the destination AAL unit) begins the receipt of the first message cell,
- during the clock cycle number 115, the AAL unit number 1 ends the sending of the first message cell,
- during the clock cycle number 133, the AAL unit number 5 ends the receipt of the first message cell, and begins the reassembly of the first message,
- during the clock cycle number 147, the AAL unit number 1 ends the segmentation of the second message,
- during the clock cycle number 149, the AAL unit number 5 begins the sending of an acknowledge receipt (ACK),
- during the clock cycle number 150, the AAL unit number 1 begins the sending of an ACK,
- during the clock cycle number 156, the AAL unit number 5 ends the reassembly of the first message, and turns it ready to be served to application processes,
- during the clock cycle number 202, the AAL unit number 5 ends the sending of an ACK,
- during the clock cycle number 203, the AAL unit number 1 ends the sending of an ACK,
- during the clock cycle number 206, the AAL unit number 1 begins the sending of the second message cell,
- during the clock cycle number 225, the AAL unit number 5 begins the receipt of the second message cell,
- during the clock cycle number 259, the AAL unit number 1 ends the sending of the second message cell,
- during the clock cycle number 277, the AAL unit number 5 ends the receipt of the second message cell, and begins the reassembly of the second message,

- during the clock cycle number 295, the AAL unit number 5 begins the sending of an ACK,
- during the clock cycle number 302, the AAL unit number 5 ends the reassembly of the second message, and turns it ready to be serve to application processes,
- during the clock cycle number 348, the AAL unit number 5 ends the sending of an ACK,
- during the clock cycle number 390, the AAL unit number 1 indicates, that it has received the ACK corresponding the second message cell,
- during the clock cycle number 394, the AAL unit number 1 marks the second message cell as to be freed,
- during the clock cycle number 395, the AAL unit number 1 frees the second message cell,
- during the clock cycle number 1056, the AAL unit number 1 is waiting for a last ACK corresponding to the first message cell.

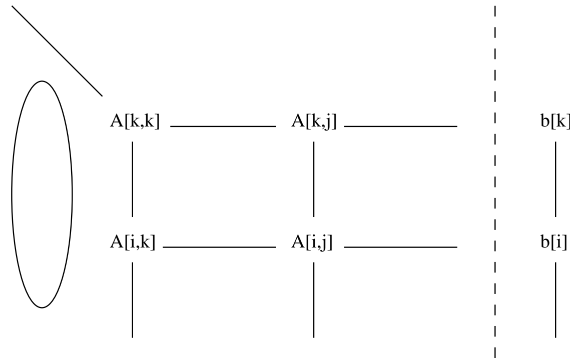
11.2 PVM Platform Operation Tests

Yet, the AAL units simulation test involves the entire platform. However, the communication scheme yielded by the (*snd*, *rcv*) processes pair used by this simulation test, is extremely simple. Indeed, the two messages that crossed the network prototype could not really interfere within it.
It therefore became necessary to run applications with more complex communication schemes, to check whether or not the network prototype properly handles several interprocess communications at a time. Running some of such applications on the PVM platform allows also to check if the network prototype access routines properly operate the mutual exclusion read and/or write accesses to the AAL units.
In order to achieve these objectives, I evaluated two well known distributed applications in the *Linear Algebra* field of interest:

- Linear Equation System Distributed Resolution
- Matrix Distributed Multiplication

11.2.1 Linear Equation System Distributed Resolution

The issue in solving a *Linear Equation System*, is to find, if any, an x vector of size p verifying the equation $Ax{=}b$, where A is an mxp matrix, and b, a vector of size m, both resulting the system to solve.
Gauss Jordan, whose sequential pseudocode is presented in Figure 11.9, is one of the algorithm most widely used to solve such systems. It has two



Let A be an NxN Matrix, and x and b be two N-size vectors.

```
for (k=0; k<N-2, k++)
    for (i=k+1; i<=N-1; i++)
        for (j=k; j<=N-1; j++)
            A[i,j] = A[i,j] - (A[i,k]*A[k,j])/A[k,k];
        endfor
        b[i] = b[i] - (A[i,k]*b[k])/A[k,k];
    endfor
endfor

for (k=N-1; k>=0; k--)
    x[k] = (b[k] - A[k,k+1]*x[k+1] - ... - A[k,N-1]*x[N-1])/A[k,k];
endfor
```

Figure 11.9: Linear Equation System Resolution Gauss Jordan Sequential Pseudocode

phases:

- triangulation
- result calculation

During the *triangulation* phase, the $A[i,k]$, $A[k,j]$ (respectively $b[k]$), and $A[k,k]$ components are required for the transformation of the $A[i,j]$ (respec-

tively $b[i]$) component, where 0≤k≤m-2, and k+1≤i≤m-1, and k≤j≤m-1, and the $A[k,k]$ component is called *pivot*.
During the *result calculation* phase, the result x vector's components are calculated from $x[m-1]$ down to $x[0]$. The $b[k]$, $A[k,k]$ components, and all the $x[j]$ and $A[k,j]$ components for k+1≤j≤m-1, are required for the calculation of the $x[k]$ component, where 0≤k≤m-1.
For the platform tests, I restricted the resolution to systems with m=p=N, and hence whose resulting Ab systems are Nx(N+1) matrices.
The distributed resolution application consists of:

- a *startup* process, whose C code is shown in Appendix K.1
- a set of *resolution* processes, whose C code is shown in Appendix K.2

First, the *startup* process reads the initial Ab system from the input/output file which is passed to it in argument on its command line, and determine the size N of the Ab system.
Next, it reads the initial Ab system again. This time, for each Ab system's component read, it starts a new *resolution* process, and passes it the read component along with the N's value via the network prototype. This way, it starts as many *resolution* processes as there are components in the Ab system, that is N*(N+1).
This *Linear Equation System Distributed Resolution* is a *fine grain* application. Indeed, the basic message units exchanged between its processes are Ab system's components. This is to generate a greater number of interprocess communications during its execution.
When the *resolution* processes are done with the *triangulation* phase, they send their current states to the *startup* process, via the network prototype. Then, the latter process appends the triangulated Ab system into the input/output file.
And, when the *resolution* processes corresponding to the Ab system's rightmost column are done with the *result calculation* phase, they send the result x vector's computed components to the *startup* process, via the network prototype. Then, the latter process appends the result x vector to the input/output file, which has been containing until then, the initial and triangulated Ab systems.
Figure 11.10 presents the input/output file formats before and after the execution of this resolution application.

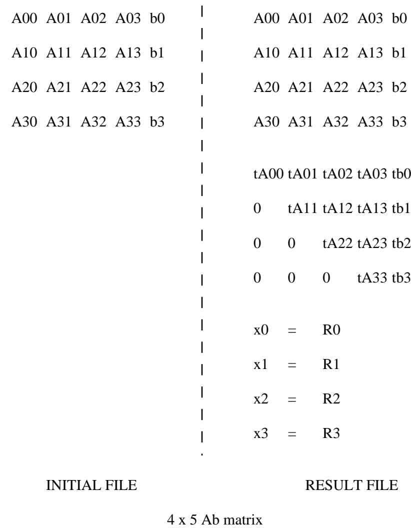


Figure 11.10: Linear Equation System Distributed Resolution Input/Output File Formats

The C program that run the *resolution* processes is written in the SPMD style. That is, the *resolution* processes run the same C code. But, they may not behave identically, since they are each working on different data.
All the *resolution* processes cooperate in order to collectively solve the system. However, all of them are not involved in all the steps of the resolution. During the *triangulation* phase, all the *resolution* processes participating in the ith step of the resolution, will participate in the $(i+1)$th step, except those from the *pivot* row and column. In addition, all the *resolution* processes involved in a given step of the resolution must transform their internal states, except those belonging to the *pivot* row. To proceed in doing so, they must each receive three internal states from:

- the *pivot* process
- the process belonging to both its column and *pivot* row
- the process belonging to both its row and *pivot* column

Moreover, the *resolution* processes handle the case where they encounter a *null pivot*, by seeking, on the *pivot* process column, for the first *resolution*

process under the *pivot* process and whose internal state is not null. If such a *resolution* process is found, then it becomes the current *pivot* process. Otherwise, the *resolution* processes merely proceed with the next step.
During the *result calculation* phase, only the *resolution* processes, corresponding the *Ab* system's rightmost column, calculate, in turn, the result x vector's components. To calculate its result x vector's component, each of these *resolution* processes receives:

- the result x vector's components previously calculated, from the *resolution* processes belonging to its column, and below it
- the internal states of the *resolution* processes belonging to its row, and above the top left and bottom right corners diagonal

Figure 11.11 illustrates the communication schemes between the *resolution* processes during the resolution of a 3x4 *Ab* system. This example assumes that, any *null pivot* has not been encountered during the *triangulation* phase. Appendix K.3 shows the input/output files from the resolution of

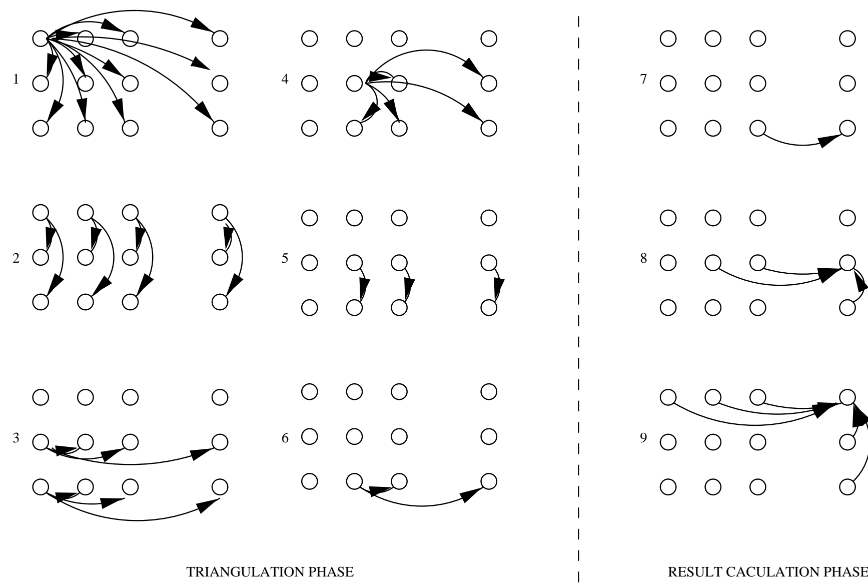


Figure 11.11: Linear Equation System Distributed Resolution Communication Schemes Example

four different Ab systems. The resolution of the 4x5 Ab system lasted 3 minutes and 37 seconds.

11.2.2 Matrix Distributed Multiplication

The issue here is to calculate the C MxN matrix resulting from the multiplication of the A MxP and B PxN matrices.
Figure 11.12 presents the matrix multiplication standard sequential algorithm. The C matrix's component $C[i,j]$ results from the *vectorial multiplication* of the A matrix's row number i and the B matrix's column number j, where $0 \leq i \leq$M-1 and $0 \leq j \leq$N-1. That is,

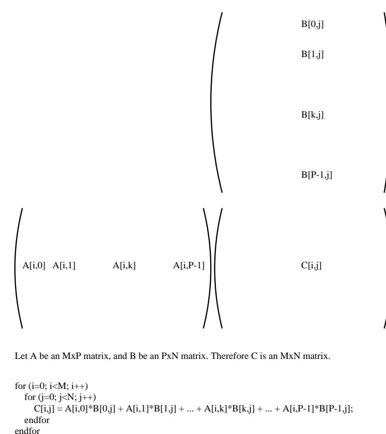


Figure 11.12: Matrix Multiplication Sequential Pseudocode

$$C[i,j] = \sum_{k=0}^{P-1} A[i,k] * B[k,j]$$

The matrix distributed multiplication application consists of:

- a startup process, whose C code is shown in Appendix K.4
- a set of multiplication processes, whose C code is shown in Appendix K.5

First, the *startup* process reads the A and B matrices to be multiplied from the input/output file which is passed to it in argument on its command line, and determine the values of M, N, and P.
Next, it reads the A and B matrices again. This time, for each A or B matrix's component read, it starts a new *multiplication* process, and passes it the read component along with the value of M, N, and P via the network prototype. This way, it starts as many *multiplication* processes as there are components in both A and B matrices, that is, $M{*}P{+}P{*}N$.
This *Matrix Distributed Multiplication* is a *fine grain* application. Indeed, the basic message units exchanged between its processes are matrices' components. This is to generate a greater number of interprocess communications during its execution.
The *startup* process gradually collects the result C matrix's components and appends them to the input/output file, as they are computed.
Figure 11.13 shows the input/output file formats before and after the execution of this application.

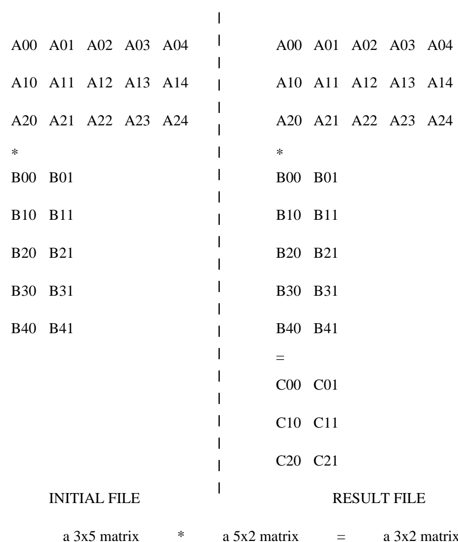


Figure 11.13: Matrix Distributed Multiplication Input/Output File Formats

The *multiplication* processes work together to collectively multiply the A and B matrices. They achieve the multiplication in as many steps as they are columns in the B matrix, that is in N steps.

During the step number j, they compute the result C matrix's column number j, that is all the $C[i,j]$ components, where $0 \leq i \leq$M-1 and $0 \leq j \leq$N-1. More specifically, during this step:

1. every *multiplication* process corresponding to a component of the B matrix's column number j sends its internal state to all those corresponding to the components of the A matrix's column, whose number equals that of the B matrix's row to which its corresponding component belongs to

2. every *multiplication* process corresponding to an A matrix's component computes a partial multiplication with the received state and its internal state, and sends the result to the one whose corresponding component belongs to the A matrix's both leftmost column, and row to which its corresponding component belongs to

3. every *multiplication* process whose corresponding component belongs to the A matrix's leftmost column, sums up all the partial multiplication it received, and sends the result to the *startup* process

Figure 11.14 illustrates the steps and communication schemes during the multiplication of 3x2 and 2x3 matrices. Appendix K.6 shows the input/output files of three matrix multiplications. The multiplication of the 3x4 and 4x2 matrices lasted 2 minutes and 52 seconds.

11.3 Prototype Performance Measurements

After having carried out the PVM platform operation tests presented in the previous section, I found interesting to evaluate the performances of the prototype.
Solely, the question that one may ask is: do the performances of such a software prototype somewhat have an hardware significance?
In order to bring an answer to this issue, I evaluated the performances of the prototype by theoretically and experimentally measuring the following performance metrics:

- r_{max} (*maximum achievable throughput*): the maximum achievable throughput which is obtained from experiments by transmitting very

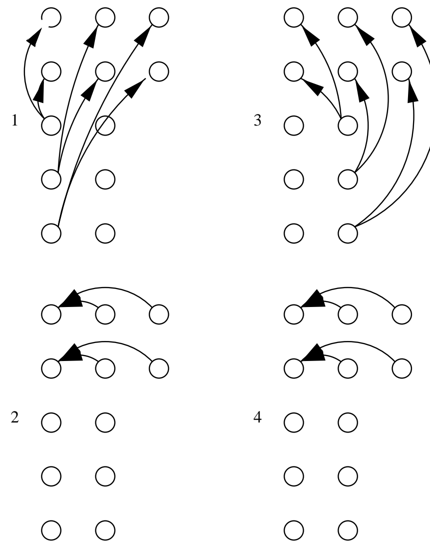


Figure 11.14: Matrix Distributed Multiplication Steps and Communication Schemes

large messages.
This is an important measure for applications requiring large volumes of data transmissions

- $n_{1/2}$ (*half performance length*): the message size needed to achieve half that of the maximum achievable throughput.
This measure provides a reference point for when half the maximum achievable throughput can be expected

- t_0 (*startup latency*): the time required to send a message of 4 bytes.
This is an important measure when transmitting short messages

These *end-to-end communication characteristics* has been defined researchers at the University of Minnesota Minneapolis, while investigating distributed network computing over local ATM networks [64]. And, they were measured on four hardware platforms, while studying the performances of PVM over local ATM networks [65]:

- PVM/ATM (AAL 3/4)

- PVM/ATM (AAL 5)
- PVM/TCP/ATM
- PVM/TCP/Ethernet

I compared my measurement results with those from the University of Minnesota Minneapolis' PVM/ATM (AAL 3/4) platform, with PVM running in the *Direct Route* (also *Advise*) mode, since my platform has the same configuration.

11.3.1 Theoretical Performances of the Platform

An anatomy of a message transfer at the network level indicates, that it starts with controller *STATE0* of the source AAL unit, which processes, first an *atminitsend* command to initiate the *send buffer*, next a series of *atmput** commands to pack the message to be sent into the *send buffer*, and finally an *atmsnd* command to buffer the message PVM header.
Next, controller *STATE1* of the source AAL unit packs the buffered message PVM header into the first cell of the message. Thereafter, it goes through the cells list of the message to be sent, and sets up their SAR headers.
Next, the trio made up with controllers *STATE2* *STATE3* and *STATE4*, goes through the cells list of the message to be sent, and sets up their ATM headers.
Next, controller *STATE5* of the source AAL unit transmits the cells of the message to be sent.
When the message arrives at the destination AAL unit, controller *STATE6* of that AAL unit stores the incoming cells into the *receive buffer*, and controller *STATE7* of that AAL unit reassembles them to the original message.
If N is the number of bytes of the message to be sent, then:

- Ndiv4 is the number 32-bit message chunks to be transfer over the host machine's local bus
- $\lceil \frac{Nmod4}{N} \rceil$ indicates, when it equals 1, that the last message chunk to be transferred over the host machine's local bus, is of size less than 32 bits. In this case, Nmod4 gives the number of bytes of that chunk

- $\lceil\frac{N+20}{36}\rceil$ is the number of cells required to pack the message to be sent into the *send buffer*.

Therefore,

- controller *STATE0* at the source AAL unit requires
  - 3 clock cycles to process the *atminitsend* command
  - $(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(5+((Nmod4)-1)*4+2)$, that is $(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4))$ clock cycles to process the series of *atmput** commands
  - 3+15 clock cycles to process the *atmsnd* command
- controller *STATE1* at the source AAL unit requires $7+(\lceil\frac{N+20}{36}\rceil-1)*4+2$, that is $5+\lceil\frac{N+20}{36}\rceil*4$ clock cycles to set up the SAR headers of the cells of the message to be sent
- the trio, at the source AAL unit, made up with controllers *STATE2* *STATE3* and *STATE4* requires $\lceil\frac{N+20}{36}\rceil*6$ clock cycles to set up the ATM headers of the cells of the message to be sent
- controller *STATE5* at the source AAL unit transmits the first byte of the outgoing message after $3 + (Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4)) + 3+15 + 5+\lceil\frac{N+20}{36}\rceil*4 + \lceil\frac{N+20}{36}\rceil*6 + 3$, that is $29+\lceil\frac{N+20}{36}\rceil*10+(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4))$
- controller *STATE6* at the destination AAL unit receives the first byte of the message being transmitted after $29+\lceil\frac{N+20}{36}\rceil*10+(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4))+18$, that is $47+\lceil\frac{N+20}{36}\rceil*10+(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4))$ clock cycles.
  5 clock cycles later, controller *STATE7* may start the reassembly of that message. And it completes it in $23+(\lceil\frac{N+20}{36}\rceil>1?58:0)+ (\lceil\frac{N+20}{36}\rceil>2?(\lceil\frac{N+20}{36}\rceil-2)*61:0)$ clock cycles.
  Therefore, the destination AAL unit is ready to serve the message after $47+\lceil\frac{N+20}{36}\rceil*10+(Ndiv4)*19+\lceil\frac{Nmod4}{N}\rceil*(3+4*(Nmod4))+ 23+(\lceil\frac{N+20}{36}\rceil>1?58:0)+ (\lceil\frac{N+20}{36}\rceil>2?(\lceil\frac{N+20}{36}\rceil-2)*61:0)$, that is,

$123+\lceil \frac{N+20}{36} \rceil$*10+(Ndiv4)*19+$\lceil \frac{Nmod4}{N} \rceil$*(3+4*(Nmod4))+
($\lceil \frac{N+20}{36} \rceil$>1?58:0)+($\lceil \frac{N+20}{36} \rceil$>2?($\lceil \frac{N+20}{36} \rceil$-2)*61:0) clock cycles.

- Now, controllers *STATE0* and *STATE8* at the destination AAL unit requires 15+7 clock cycles to locate the message in the *receive buffer*, upon the information from the *atmrcv* command.
  Next, controller *STATE0* at the destination AAL unit requires
  (Ndiv4)*19+$\lceil \frac{Nmod4}{N} \rceil$*(3+4*(Nmod4)) clock cycles
  to process the series of *atmget** commands used to unpack the message from the *receive buffer*

In summary, the transfer of a message of size N bytes lasts
123+22+$\lceil \frac{N+20}{36} \rceil$*10+2*((Ndiv4)*19+$\lceil \frac{Nmod4}{N} \rceil$*(3+4*(Nmod4)))+
($\lceil \frac{N+20}{36} \rceil$>1?58:0)+($\lceil \frac{N+20}{36} \rceil$>2?($\lceil \frac{N+20}{36} \rceil$-2)*61:0), that is
145+$\lceil \frac{N+20}{36} \rceil$*10+(Ndiv4)*38+$\lceil \frac{Nmod4}{N} \rceil$*(6+8*(Nmod4))+
($\lceil \frac{N+20}{36} \rceil$>1?58:0)+($\lceil \frac{N+20}{36} \rceil$>2?($\lceil \frac{N+20}{36} \rceil$-2)*61:0) clock cycles
Under the *Gnuplot* UNIX interactive plotting program, the function
*f(x)=145+10*ceil((x+20)/36)+38*floor(x/4)+ceil(mod(x,4)/x)*(6+8*mod(x,4))+*
*(ceil((x+20)/36)>1?58:0)+(ceil((x+20)/36)>2?(ceil((x+20)/36)-2)*61:0)*
where *mod(x,y)=x-y*floor(x/y)*,
returns the *theoretical one-way delay* of the transfer of a message of size x.

11.3.2 Experimental Performances of the Prototype

11.3.2.1 Echo Program

As mentioned in [64, 65], an *echo* program is used to measure an *end-to-end communication latency* between two machines. I used a slightly different version of that *echo* program to measure the *end-to-end communication delay* between two processes that communicate via the network prototype.
In my *echo* program version, the *source* process sends a message of size $4*msg_size$ bytes to the destination process.
After the source process has been granted the permission to write to the *send buffer* of the AAL unit to which its bound with, and before it starts to pack the message to be sent into it, it stores the current value of the *cycle* shared variable into the *cycle*1 shared variable.
After the *destination* process has completely moved the received message into its local memory, it reads the current values of the *cycle*1 and *cycle*

shared variables into the *cycle*2 and *cycle*3 local variables respectively.
Then, the *one way delay* of the last message transfer between the source and destination processes is measured by subtracting *cycle2* from *cycle3*.
The *cycle* variable is shared by the network, source, and destination processes, whereas the *cycle*1 variable is shared only by the source and destination processes.
Recall that, the *cycle* shared variable is incremented by one at every iteration of the *network* process loop (also clock cycle), by the latter. Hence on my platform, the delay measurements are carried out in terms of elapse network clock cycles.
On my platform, I chose to measure a *one way* rather than *round trip* delay for two main reasons:

1. It is possible. Indeed, the use of shared variables obviates the very tough issue of the synchronization of clocks from two different machines.

2. It allows to drastically reduce the simulation time, specially during the transfer of large messages.

In addition, conversely to University of Minnesota Minneapolis' *echo* program, my *echo* program does not perform a *statistical* measurement. Indeed, it does not repeat N times a message transfer and its delay measurement. The simulation time required in that case is quite long.
Figure 11.15 shows an overview of my *echo* program, and the C code extracts of the *source* and *destination* processes, as well as the way they compute the one way delay of a message transfer.

11.3.2.2 Measurement Results

Table 11.1 shows, the experimental and theoretical one-way message transfer delay versus the message size.
Figure 11.16 shows plots from Table 11.1.
The expected curves are straight lines. However, there is a slight disruption into the experimental points alignment.
Nevertheless, the resulting cloud of points has the shape of a straight line. And, the equation $Y{=}a{*}X{+}b$ with $a = \frac{\sum_{i=0}^{13}(X_i-\bar{X})(Y_i-\bar{Y})}{\sum_{i=0}^{13}(X_i-\bar{X})^2}$ and $b = \bar{Y} - a * \bar{X}$, returns $Y{=}17.356792{*}X{+}11068.489258$ as the experimental one-way delay

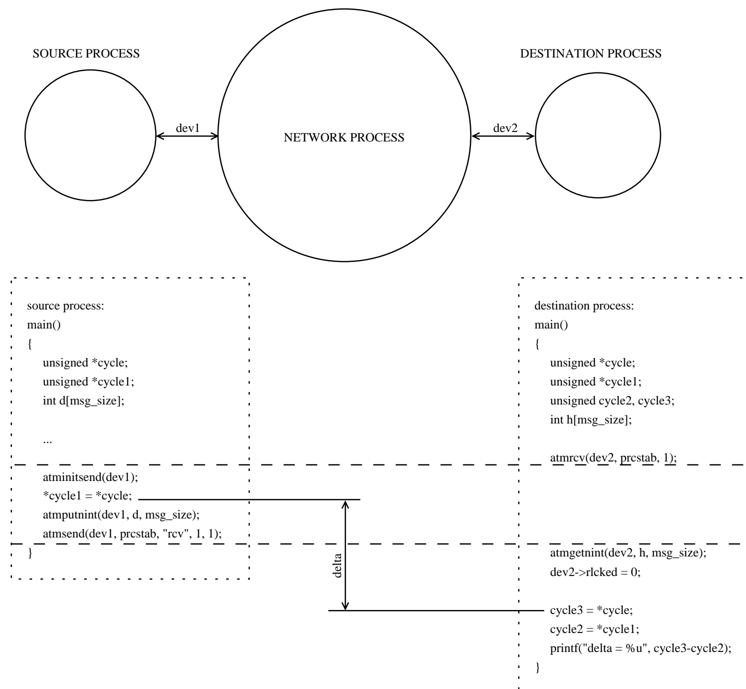


Figure 11.15: Echo Program Overview

message size (bytes)	experimental delay (cycles)	theoretical delay (cycles)
1000	19460	11640
5000	94410	57521
10000	190441	114890
15000	281160	172259
20000	370003	229628
25000	460233	286926
30000	534713	344295
35000	620481	401664
40000	685766	459033
45000	758368	516331
50000	863335	573771
55000	964429	631140
60000	1069154	688509
65000	1157703	745878

Table 11.1: One-Way Delay

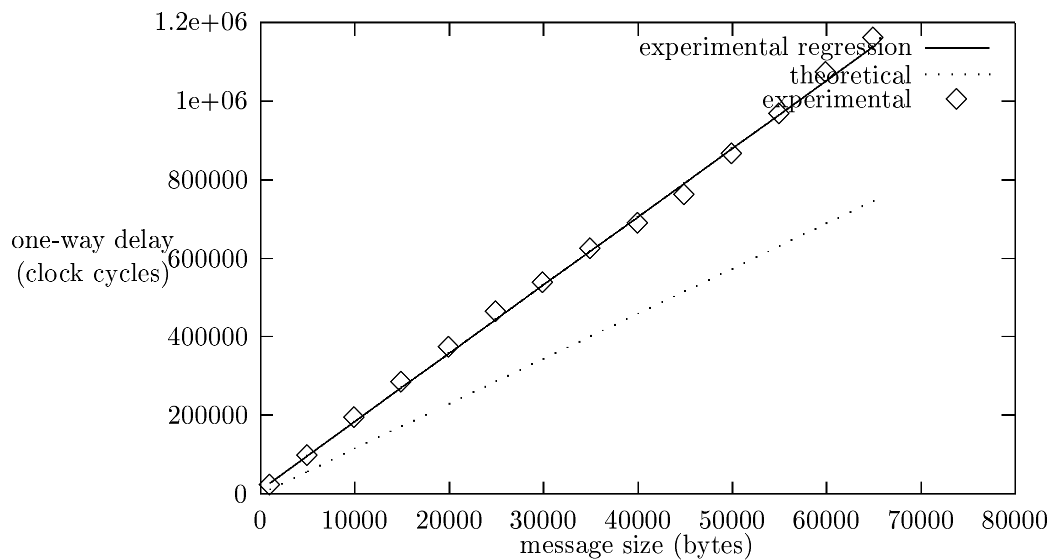


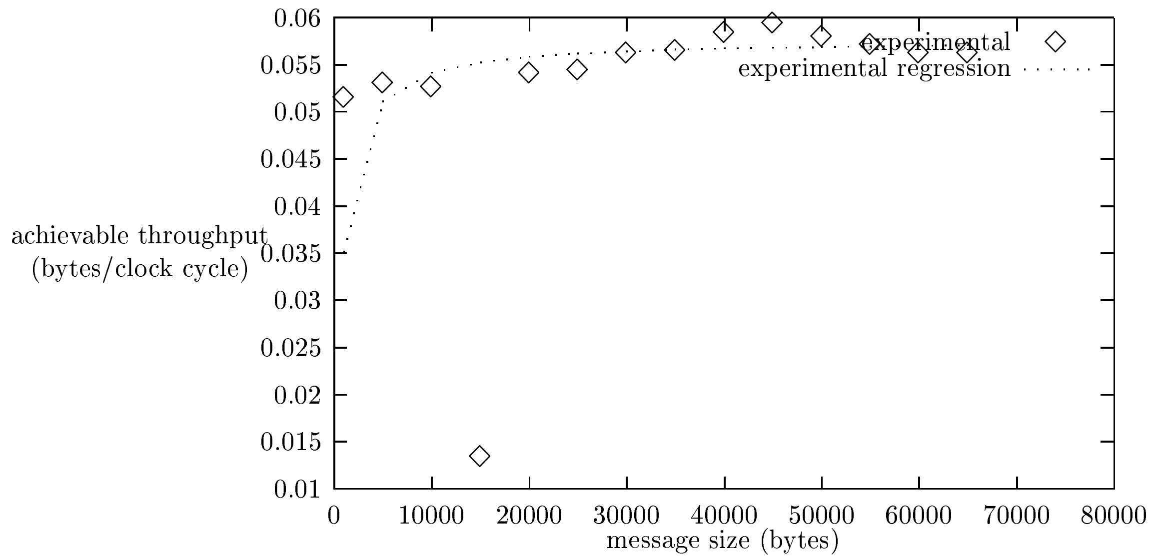
Figure 11.16: Direct mode: One-way Delay versus message size

regression line.
The disruption on the *experimental* curve, and the deviation between the *regression* and *theoretical* lines may result from the following fact:
At the beginning, our laboratory server was a mono-processor machine running UNIX Operating System (OS). Hence, the synchronization between the network and application processes were designed assuming that, all the processes run on the same processor and their schedule is based on a *round-robin* mechanism.
When the experimental measurement were carried out, a second processor had been added to it, and then, it were running SOLARIS OS.
Now, SOLARIS can guarantee a processes schedule based on a *round-robin* mechanism only on each processor and not both at a time.
Therefore, when a process from an application packs (respectively unpacks) a message into a send (respectively from a receive) buffer in several passes (this is the case for messages whose length is greater than 32 bits), the network process may loop more than once between two consecutives passes, since both processes do not run on the same processor.
By the way, this experiment was performed on a *SUNW, SPARCstation-20.* It lasted an entire week-end day.
A transfer throughput is calculated by dividing a message size by its corresponding *one-way* delay. Figure 11.17 shows the achievable throughput versus the message size.
My platform achieves an *experimental* (respectively *experimental regression*, *theoretical*) maximum throughput of 0.0593379467 (respectively 0.0570546, 0.0871456189) (bytes/cycle).
Table 11.2 shows the *one-way delay* and *achievable throughput* measurements in terms of r_{max}, $n_{1/2}$, and t_0 performance metrics.

measurement type	r_{max} (bytes/cycle)	$n_{1/2}$ (bytes)	t_0 (cycles)
experimental regression	0.0570546	625	11346
theoretical	0.0871456189	12	193

Table 11.2: Direct Route

Since the network prototype's transmission links are 8-bit lines, a bandwidth of 155.52 (respectively 622.08) Mbits/sec implies a network clock frequency



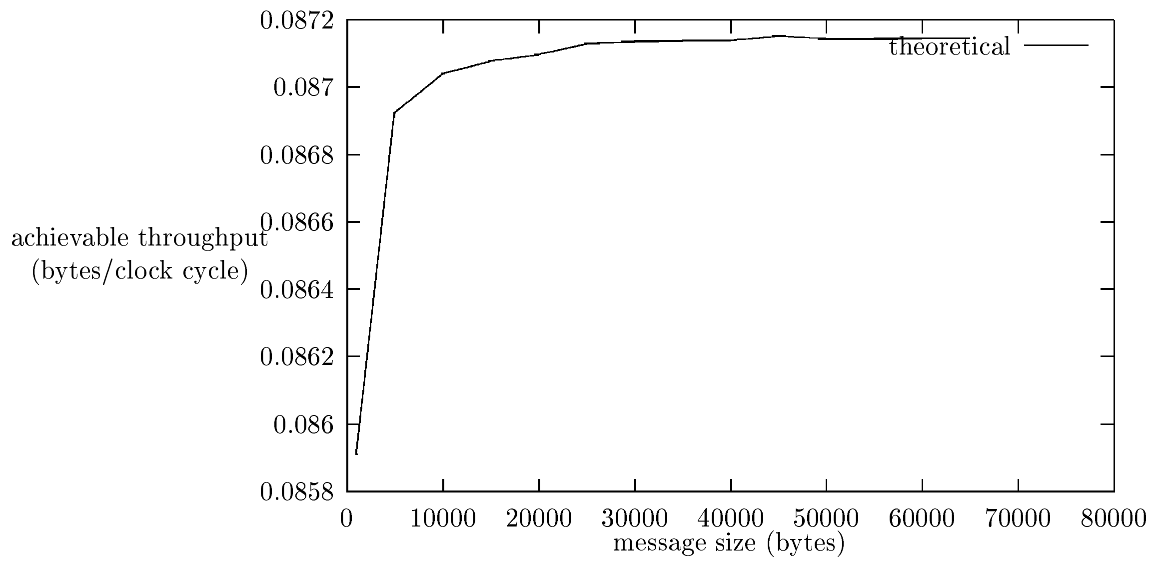


Figure 11.17: Direct mode: Achievable Throughput versus message size

of $\frac{155.52}{8}$ (respectively $\frac{622.08}{8}$) Mbits/sec or MHz. Therefore, the *bytes/cycle* unit can be converted in 155.52 (respectively 622.08) Mbits/sec.
Upon this *bytes/cycle* unit conversions, Table 11.3 gives the r_{max} values in Mbits/sec.

measurement type	r_{max} (Mbits/sec) at 155.52 Mbits/sec	r_{max} (Mbits/sec) at 622.08 Mbits/sec
experimental regression	8.873131392	35.492525568
theoretical	13.5528866513	54.2115466053

Table 11.3: r_{max} values in Mbits/sec

The ASX-100 local ATM switch [66] is based on a 2.4 Gbits/sec switch fabric. I will then base my network prototype's BMX switches on 622.08 Mbits/sec switch fabrics.
In this context, while the overhead from PVM in *Direct Route* mode limits the maximum throughput of the University of Minnesota Minneapolis' PVM/ATM platform to 27.202 Mbits/sec [65], my ATM/Software ATM platform achieves an *experimental* (respectively *theoretical* maximum throughput of 35.492 (respectively 54.211) Mbits/sec.

11.4 Contentions Impact over The execution of an Application

This section present an evaluation the impact of the contentions within the network prototype over the execution, on the PVM platform, of the distributed resolution of the linear equation system shown in the example number 4 of Appendix K.3.
Figure 11.18 illustrates the way this evaluation is achieved.
First, I ran the resolution on the PVM platform. During this resolution, whenever a resolution process (See Section 11.2.1) is granted a permission to pack a message into the send buffer it is bound to, it sends a send operation message including:

- the name of the host machine over which it is running

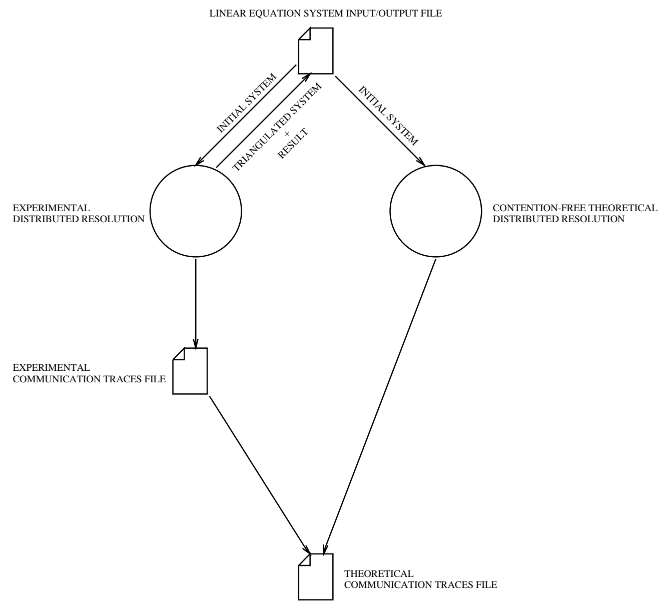


Figure 11.18: Network Prototype Contentions Impact over an Application Execution

- the network clock cycle at which it starts to pack the message
- the send operation indication
- the message tag
- its name and instance number
- the destination process' name and instance number

to its local PVM daemon.
And whenever a resolution process completes to unpack a message from the receive buffer to which it is bound to, it sends a receive operation message including:

- the name of the host machine over which it is running
- the network clock cycle at which it ends to unpack the message
- the message tag
- its name and instance number

to its local PVM daemon.
Gradually, the master PVM daemon collects all the these messages from the local PVM daemons and prints them on its standard output and into the experimental communication traces file.
This is how this resolution returned the experimental communication traces file shown in Appendix K.3.1. Note that, in accordance with the PVM communication protocol, matching send and receive operations are identified according to the message tags, the source and destination process names and instance numbers.
Next, I manually ran the same resolution in accordance with the steps presented in Section 11.2.1, and produced the resulting theoretical communication traces file shown in Appendix K.3.2 assuming that the network prototype is contention-free, and applying the following rules:

- The message reception order is maintained inside and between processes bound to the same input/output port of the nerwork prototype, in accordance with the experimental communication traces file.

- The numbers of clock cycles elapsed between consecutive send, send and receive operations are maintained inside processes, in accordance with the experimental communication traces file.
- Only the message receipt clock cycles are changed, providing the transfer of an integer, that is a 4-byte message, through the network prototype is 193 clock cycles.
- The minimum number of clock cycles elapsed between two consecutive receive operations over an output port of the network prototype is 40 clock cycles. It is the time required to simply unpack an integer from a receive buffer.

Finally, I select the receive operation perfomed by the resolution process instance number 4, while receiving the internal value of the resolution process instance number 0 with a message tag of 20, since it is the last inter-resolution process communication operation during the resolution itself.
The experimental communication traces file indicates that this operation took place during the clock cycle number 14228 (See Appendix K.3.1), whereas the theoretical communication traces file indicates the it took place during the clock cycle number 9211 (See Appendix K.3.2).
Therefore, it appears that the contentions within the network prototype lengthened the distributed resolution of the linear equation system shown in the example number 4 in the Appendix K.3 of 5107 clock cycles, that is nearly 36% of the overall resolution time.

11.5 Tests and Measurements Conclusion

The simulation tests presented in this chapter indicate that the network prototype and PVM platform described in this dissertation are operational.
On the other hand, the evaluation of the performances of the network prototype discussed in this chapter, indicates that the experimental measurements are consistent with both the theoretical measurements derived directly from the architecture of the network prototype, and the measurements over the PVM/ATM(AAL3/4) hardware platform performed at the University of Minnesota Minneapolis.
From this evaluation, it appears that it theoretically takes 193 clock cycles

to transfer a 4-byte message via the network prototype, and the transfer delay of the network prototype linearly increases as the message size. In addition, the throughput achieved by the network prototype sharply increases for small message sizes. And, for higher message sizes, it quickly saturates to achieve an approximative maximum value of 0.06 bytes/cycle, that is, 8,9 Mbits/sec for a network speed of 155.52 Mbits/sec or 35,5 Mbits/sec for a network speed of 622.08 Mbits/sec.
Finally, the comparison presented in this chapter, between an execution of the distributed resolution of a particular linear equation system on the PVM platform, and a theoretical execution of the same resolution assuming that the network prototype is contention-free, indicates that the contentions in the bosom of the network prototype lengthened the execution of the resolution of an extra delay, which is nearly 36% of the overall execution time.

Chapter 12

Conclusion and Perspectives

This dissertation has presented the architecture of the ATM-based interconnection network that I designed during my Ph.D. works.
The underlying transmission system of this network is extremely simple. It is of type asynchronous, with the transmission links organized into recurrent bytes. This is for the network to best deal with a byte-wise transfer of cells.
The Physical Layer (PL) directly above the tansmission system only performs pure byte-wise transmissions of cells.
The ATM layer above the physical layer consists of 4x4 switches. The size 4, along with a suitable interconnection topology, allows to achieve an interconnection network with yet an interesting number of input and output ports and a reduced number of switches. These switches are of type Bus Matrix Switch (BMX). Hence, they achieve the same performance as the classical output queueing-based switches without resorting to run faster than the transmission links. They have been interconnected in a way to achieve a 3-stage interconnection topology, with 4 switches on each stage. The resulting interconnection network is a rearrangeable clos network with 16 input and output ports.
The ATM Adaptation Layer (AAL) directly above the ATM layer is a lightened version of the AAL3/4. Indeed, its Segmentation And Reassembly (SAR) sublayer only performs the segmentation and reassembly of messages, and the processing of cell errors, cell losses and insertions, whereas, its Convergence Sublayer (CS) only performs the multiplexing and demultiplexing of messages.
Moreover, the AAL layer has been designed in a way to allow a seamless

implementation of PVM over this network.
On the other hand, the physical layer, the switches, and the AAL cards that I designed only implement functions of the User Plane. Hence, this network does not deal with neither the establishment and release of connections, nor its proper management.
This partly explain why sole a connectionless data service is provided above this network. Indeed, communications via this network use the unleased Permanent Virtual Connections (PVC) established at the setup time of this network. And, the congestion control within this network uses an hop by hop flow control mechanism based on a back-pressure transmission suspension signal, to throttle a source hop, that is an upstream AAL card or SPD unit, into suspending its current transmission of the bytes from an outgoing cell when the destination hop, that is a downstream PPD unit or AAL card, is not able to buffer the incoming cell entirely into a destination queue. A traffic control is not really implemented in this network. Indeed, there is neither any source policing nor any source shaping at the access points to this network. Sources transmit cells at their proper rates until there is not any more free space left available in the send buffers to which they are bound.
The architecture of this network has been described in the C programming language at the Register Transfer Level (RTL), hence achieving an ATM network C prototype.
A C prototype were achieved in order to simulate the architecture of the aforementioned ATM network directly under the workloads dynamically generated while running real distributed applications. This is to avoid the use of neither any workload generation analytical model, since it is difficult to find whether or not such a workload reflects a given reality, nor any workload derived from the traces of the execution of a real distributed application over a parallel machine, since the interconnection networks have an impact over the schedule of the communications of a running application, and since the interconnection network of that parallel machine and the network C prototype are different.
Classical VLSI design tools, such as VHDL, Verilog, Compass, Cadence, etc., do not allow to meet this objective. Then, it became necessary to select an Application Programming Interface (API) and to implement it over the network C prototype.
In this context, this dissertation has presented two emerging APIs: the Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). Although

MPI is right now viewed as being more mainstream than PVM, I implemented a PVM platform, simply because MPI was not officially released when I started my Ph.D. late 1993. The PVM platform has been presented in this dissertation.

The simulation tests presented in this dissertation indicate that the network C prototype and PVM platform are operational.

An evaluaton of the performances of the network C prototype has been presented in this dissertation. It indicates that the experimental measurement results are consistent with the treoretical measurement results directly derived from the architecture of the network C prototype, and the results of the measurements performed over a PVM/ATM(AAL3/4) hardware platform at the University of Minnesota Minneapolis.

In summary, it theoretically takes 193 clock cycles to transfer a 4-byte message via the network C prototype, and the transfer delay of this network linearly increases as the message size. In addition, the throughput achieved by this network sharply increases for small message sizes, whereas for higher message sizes it quickly saturates to achieve an approximative maximum value of 0.06 bytes/cycle, that is 8.9 Mbits/sec for a network speed of 155.52 Mbits/sec or 35.5 Mbits/sec for a network speed of 622.08 Mbits/sec.

Finally, this dissertation has presented a comparison between an execution of the distributed resolution of a particular linear equation system on the PVM platform and a theoretical execution of the same resolution assuming that the network C prototype is contention-free.

It shows up that the contentions within the network C prototype lengthened the resolution of an extra delay, which is nearly 36% of the overall resolution time.

Now, the PVM platform needs to be fully exploited. It might be used to run PVM distributed applications other than the linear equation system distributed resolution and matrix distributed multiplication presented in this dissertation.

In addition, it might be interesting to investigate several other ATM network architectures. In order to achieve that, one might either keep the transmission system, physical layer, switches, and AAL cards of the network C prototype presented in this dissertation, and only study different interconnection topologies, or study partially or totally different ATM network architectures. In the latter study case, switches from the industry might be described in C at the register transfer level and used to setup a different ATM network C

prototype.
Finally, It might be interesting to achieve the hardware prototype of the ATM network that I designed. The performances of the hardware prototype should be consistent with that of the C prototype.
The thesis of this dissertation is that the PVM platform is a promising tool for our investigations in ATM-based interconnection networks.

Part III

Appendixes

Appendix A

PPD Units' C Code and Data Structures

This appendix presents the C Data Structures that describes the PPD units' resources, and the C code that simulates the PPD units architecture (See Figure 9.2 in Section 9.2.1.1) during one network clock cycle.

A.1 PPD Units' C Data Structures

```
struct xpm
{ short head, tail, size;
  char mem[424];
}
struct hdrtrsltab
{ unsigned short wadr, wcmd;
  char datain[7];
  char mem[1792];
}
struct ppd
{ char *byte;
  short *byteind;
  char a, b, c, d, f, g, i, j, l, m, o, p, q;
  short aind, bind, cind, dind, oind, pind, qind;
  short e, h, k, n, count, suspend;
  struct hdrtrsltab trsltab;
  struct xpm xpm_dst[4];
}
```

A.2 PPD Units' C Code

```
void ppd_scheduler(curppd)
struct ppd *curppd;
{ char rbuf[7], fifo0, fifo1, fifo2, fifo3;
  unsigned short r, s, t, u, x, y, z;
  unsigned short radr, dstflg, vp0, vp1, vp2, vp3;

  if (curppd->trsltab.wcmd)
```

```
  for (curppd->trsltab.wcmd=u=0; u<7; u++)
    curppd->trsltab.mem[u+curppd->trsltab.wadr*7]=
    curppd->trsltab.datain[u];
if ((curppd->e && curppd->xpm_dst[0].size>=424-5) ||
    (curppd->h && curppd->xpm_dst[1].size>=424-5) ||
    (curppd->k && curppd->xpm_dst[2].size>=424-5) ||
    (curppd->n && curppd->xpm_dst[3].size>=424-5)
   ) s=1;
else s=0;
if (curppd->cind && s)
{ r=1;
  curppd->e=curppd->h=curppd->k=curppd->n=0;
}
else
{ r=0;
  if (curppd->cind || curppd->dind || curppd->qind)
  { if (curppd->qind) fifo0=fifo1=fifo2=fifo3=curppd->q;
    else
      if (curppd->cind)
      { fifo0=curppd->c; fifo1=curppd->f; fifo2=curppd->i;
        fifo3=curppd->l;
      }
      else
      { fifo0=curppd->d; fifo1=curppd->g; fifo2=curppd->j;
        fifo3=curppd->m;
      }
    if (curppd->e)
    { curppd->xpm_dst[0].mem[curppd->xpm_dst[0].tail]=fifo0;
      curppd->xpm_dst[0].tail=
      (curppd->xpm_dst[0].tail+1)%424;
      curppd->xpm_dst[0].size++;
    }
    if (curppd->h)
    { curppd->xpm_dst[1].mem[curppd->xpm_dst[1].tail]=fifo1;
      curppd->xpm_dst[1].tail=
      (curppd->xpm_dst[1].tail+1)%424;
      curppd->xpm_dst[1].size++;
    }
    if (curppd->k)
    { curppd->xpm_dst[2].mem[curppd->xpm_dst[2].tail]=fifo2;
      curppd->xpm_dst[2].tail=
      (curppd->xpm_dst[2].tail+1)%424;
      curppd->xpm_dst[2].size++;
    }
    if (curppd->n)
    { curppd->xpm_dst[3].mem[curppd->xpm_dst[3].tail]=fifo3;
      curppd->xpm_dst[3].tail=
      (curppd->xpm_dst[3].tail+1)%424;
      curppd->xpm_dst[3].size++;
    }
  }
}
if (curppd->bind)
{ ((char*)&radr)[0]=curppd->a;
  ((char*)&radr)[1]=curppd->b;
  t=(radr>>4)*7;
  for (u=0; u<7; u++) rbuf[u]=curppd->trsltab.mem[t+u];
  ((char*)&dstflg)[1]=((char*)&vp0)[1]=rbuf[6];
  ((char*)&dstflg)[0]=((char*)&vp0)[0]=rbuf[5];
  vp0=(vp0&0xfff0)|(radr&0xf);
  ((char*)&vp1)[1]=rbuf[4];
  ((char*)&vp1)[0]=rbuf[3];
  vp1=(vp1<<4)|(radr&0xf);
  ((char*)&vp2)[1]=rbuf[3];
  ((char*)&vp2)[0]=rbuf[2];
  vp2=(vp2&0xfff0)|(radr&0xf);
  ((char*)&vp3)[1]=rbuf[1];
  ((char*)&vp3)[0]=rbuf[0];
  vp3=(vp3<<4)|(radr&0xf);
  curppd->c=((char*)&vp0)[0];
  curppd->d=((char*)&vp0)[1];
  curppd->e=(dstflg&1?1:0);
  curppd->f=((char*)&vp1)[0];
  curppd->g=((char*)&vp1)[1];
  curppd->h=(dstflg&2?1:0);
```

```
    curppd->i=((char*)&vp2)[0];
    curppd->j=((char*)&vp2)[1];
    curppd->k=(dstflg&4?1:0);
    curppd->l=((char*)&vp3)[0];
    curppd->m=((char*)&vp3)[1];
    curppd->n=(dstflg&8?1:0);
  }
  curppd->dind=(r ? 0 : curppd->cind);
  curppd->cind=curppd->bind;
  curppd->q=curppd->p;
  curppd->qind=curppd->pind;
  curppd->p=curppd->o;
  curppd->pind=(curppd->suspend || r ? 0 : curppd->oind);
  if (*curppd->byteind)
    if (!curppd->count)
    { curppd->bind=curppd->oind=0;
      curppd->aind=1;
      curppd->a=*curppd->byte;
      curppd->count=(curppd->count+1)%53;
    }
    else
      if (curppd->count==1)
      { curppd->aind=curppd->oind=0;
        curppd->bind=1;
        curppd->b=*curppd->byte;
        curppd->count=(curppd->count+1)%53;
      }
      else
      { curppd->aind=curppd->bind=0;
        if (curppd->suspend) curppd->oind=curppd->count=0;
        else
        { curppd->oind=1;
          curppd->o=*curppd->byte;
          curppd->count=(curppd->count+1)%53;
        }
      }
  else curppd->aind=curppd->bind=curppd->oind=0;
  curppd->suspend=r;
}
```

Appendix B

SPD Units' C Code and Data Structures

This appendix presents the C Data Structures that describes the SPD units' resources, and the C code that simulates the SPD units architecture (See Figure 9.5 in Section 9.2.1.2) during one network clock cycle.

B.1 SPD Units' C Data Structures

```
struct spd
{ char buf, byte;
  short bufind, byteind;
  short trsmit, siqnum, iqrst, lsthead;
  short *suspend;
  short iqcnt[4];
  struct xpm *xpm_src[4];
}
```

B.2 SPD Units' C Code

```
void spd_scheduler(curspd)
struct spd *curspd;
{ unsigned short i;

  curspd->byte=curspd->buf;
  curspd->byteind=curspd->bufind;
  if (*curspd->suspend)
  { curspd->bufind=0;
    curspd->xpm_src[curspd->siqnum]->head=curspd->lsthead;
    curspd->xpm_src[curspd->siqnum]->size+=5;
    curspd->iqcnt[curspd->siqnum]=0;
    curspd->trsmit=0;
  }
  if (curspd->trsmit)
  { curspd->bufind=1;
    curspd->buf=curspd->xpm_src[curspd->siqnum]->
    mem[curspd->xpm_src[curspd->siqnum]->head];
```

```
    if (!curspd->iqcnt[curspd->siqnum])
      curspd->lsthead=curspd->xpm_src[curspd->siqnum]->head;
    curspd->xpm_src[curspd->siqnum]->head=
    (curspd->xpm_src[curspd->siqnum]->head+1)%424;
    curspd->xpm_src[curspd->siqnum]->size--;
    if (curspd->iqcnt[curspd->siqnum]==52) curspd->trsmit=0;
    curspd->iqcnt[curspd->siqnum]=
    (curspd->iqcnt[curspd->siqnum]+1)%53;
  }
  else curspd->bufind=0;
  if (!curspd->trsmit)
    for (i=(curspd->siqnum+1)%4; ; i=(i+1)%4)
    { if (curspd->xpm_src[i]->size)
      { curspd->iqrst=curspd->siqnum=i;
        curspd->trsmit=1;
        break;
      }
      if (i==curspd->siqnum) break;
    }
}
```

Appendix C

Network Prototype's Interconnection Topology, and Predefined Virtual Connections

This appendix shows the PPD and SPD units interconnection algorithm that achieves the network prototype's topology presented in Figure 9.7 in Section 9.2.2, and the establishment algorithm of a set of predefined virtual connections.

C.1 PPD and SPD units Interconnection Algorithm

```
for (i=0; i<48; i+=4)
  if (i<32)
  {
    l = 16*(i/16 + 1) + 4*(i%4) + (i%16)/4
    /* note that, l increases as i increases.   */
    /* Then, since 0<=i<=31, therefore 0<=l<=47 */
    the SPD of rank i is connected to the PPD of rank l.
    if (i<16)
    {
      /* note that, 0<=i<=15 */
      the PPD of rank i is assigned to
      the input port number i
      of the interconnection network, of the prototype.
    }
  }
  else
  {
    /* note that, 32<=i<=47 */
    the SPD of rank i is assigned to
    the output port number i-32
    of the interconnection network, of the prototype.
  }
```

C.2 Establishment Algorithm of Predefined Virtual Connections

```
...
unsigned buf[2];
/* buf is used to synthesize PPD routing tables entries */
unsigned short ppdcnxnum[48];
...
for (i=0; i<48; i++) ppdcnxnum[i]=0;
/* ppdcnxnum[i] indicates the next PPD[i] routing table */
/* entry to be reserved.                                */
for(src=0; src<16; src++)
/* src is the rank of the network source input port */
{ i = src - src%4;

  /* regarding point-to-point communications */

  for(dst=0; dst<16; dst++)
  /* dst is the rank of the network destination output port */
  { j = dst/4;
    q = dst%4;
    for(k=0; k<4; k++)
    /* i+k is the rank of the network input stage output SPD */
    { l = 16*(i/16 + 1) + 4*k + (i%16)/4;
      /* l is the rank of the network middle stage input PPD */
      m = l - l%4;
      /* m+j is the rank of the network middle stage */
      /* output SPD                                  */
      n = 16*(m/16 + 1) + 4*j + (m%16)/4;
      /* n is the rank of the network output stage input PPD */
      p = n - n%4;
      /* p+q is the rank of the network output stage */
      /* output SPD                                  */
      a = ppdcnxnum[n]++;
      /* a is the virtual path identifier to used between  */
      /* PPD[l] and PPD[n] for the path currently computed */
      if (!q) buf[1] = 1;
      else if (1==q) buf[1] = 2;
           else if (2==q) buf[1] = 4;
                else buf[1] = 8;
      buf[0] = 0;
      for(u=0; u<7; u++)
        ppd[n].trsltab.mem[u+7*a] = ((char*)buf + 1)[u];
      b = ppdcnxnum[l]++;
      /* b is the virtual path identifier to used between    */
      /* PPD[src] and PPD[l] for the path currently computed */
      if (!j)
      { buf[0] = 0;
        buf[1] = (a << 4) | 1;
      }
      else if (1==j)
           { buf[0] = 0;
             buf[1] = a;
             buf[1] = (buf[1] << 16) | 2;
           }
           else if (2==j)
                { buf[0] = buf[1] = a;
                  buf[0] >>= 4;
                  buf[1] = (buf[1] << 28) | 4;
                }
                else
                { buf[0] = a << 8;
                  buf[1] = 8;
                }
      for(u=0; u<7; u++)
        ppd[l].trsltab.mem[u+7*b] = ((char*)buf + 1)[u];
      c = ppdcnxnum[src]++;
      /* c is the virtual path identifier to used at the  */
      /* network access for the path currently computed   */
      if (!k)
      { buf[0] = 0;
        buf[1] = (b << 4) | 1;
```

```
    }
    else if (1==k)
         { buf[0] = 0;
           buf[1] = b;
           buf[1] = (buf[1] << 16) | 2;
         }
         else if (2==k)
              { buf[0] = buf[1] = b;
                buf[0] >>= 4;
                buf[1] = (buf[1] << 28) | 4;
              }
              else
              { buf[0] = b << 8;
                buf[1] = 8;
              }
    for(u=0; u<7; u++)
      ppd[src].trsltab.mem[u+7*c] = ((char*)buf + 1)[u];
    aal34[src].vpoto[dst][k] = c;
  }
}

/* regarding broadcast communications */

for(k=0; k<4; k++)
/* i+k is the rank of the network input stage output SPD */
{ l = 16*(i/16 + 1) + 4*k + (i%16)/4;
  /* l is the rank of the network middle stage input PPD */
  m = l - l%4;
  for(j=0; j<4; j++)
  /* m+j is the rank of the network middle stage */
  /* output SPD                                   */
  { n = 16*(m/16 + 1) + 4*j + (m%16)/4;
    /* n is the rank of the network output stage */
    /* input PPD                                 */
    a = ppdcnxnum[n]++;
    /* a is the virtual path identifier to used between  */
    /* PPD[l] and PPD[n] for the path currently computed */
    buf[0] = 0;
    buf[1] = 15;
    for(u=0; u<7; u++)
      ppd[n].trsltab.mem[u+7*a] = ((char*)buf + 1)[u];
    if (!j) b = a;
    else if (1==j) c = a;
         else if (2==j) d = a;
              else e = a;
  }
  a = ppdcnxnum[l]++;
  /* a is the virtual path identifier to used between  */
  /* PPD[src] and PPD[l] for the path currently computed */
  buf[0] = (e << 8) | (d >> 4);
  buf[1] = (d << 12) | c;
  buf[1] = (buf[1] << 16) | (b << 4) | 15;
  for(u=0; u<7; u++)
    ppd[l].trsltab.mem[u+7*a] = ((char*)buf + 1)[u];
  b = ppdcnxnum[src]++;
  /* b is the virtual path identifier to used at the  */
  /* network access for the path currently computed   */
  if (!k)
  { buf[0] = 0;
    buf[1] = (a << 4) | 1;
  }
  else if (1==k)
       { buf[0] = 0;
         buf[1] = a;
         buf[1] = (buf[1] << 16) | 2;
       }
       else if (2==k)
            { buf[0] = buf[1] = a;
              buf[0] >>= 4;
              buf[1] = (buf[1] << 28) | 4;
            }
            else
            { buf[0] = a << 8;
              buf[1] = 8;
            }
```

```
    for(u=0; u<7; u++)
      ppd[src].trsltab.mem[u+7*b] = ((char*)buf + 1)[u];
    aa134[src].vpota[k] = b;
  }
}
```

Appendix D

CRC Code Generation and Verification

This appendix presents the CRC code generation algorithm used by source AAL units when transmitting cells, and the CRC code verification algorithm used by destination AAL units when receiving cells (See Figures 9.12 and 9.13 in Section 9.3.1).

D.1 CRC Code Generation Algorithm

```
...
unsigned gen=0x11021;
unsigned crc;
char cell[53];
/* cell being transmitted */
short byternk;
/* rank of the cell's byte currently transmitted */
short i;
...
if (!byternk) crc=0;
else
  if (byternk<3) ((char*)&crc)[byternk]=cell[byternk+4];
  else
    if (byternk<49)
    { if (byternk<47) ((char*)&crc)[3]=cellbyternk+4];
      for (i=0; i<8; i++)
      { crc<<=1;
        if (crc & 0x1000000) crc=(crc&~gen)|(~crc&gen);
      }
    }
    else
      if (byternk<51)
      /* at this point,                              */
      /* byternk==49 or byternk==50, and             */
      /* ((char*)&crc)[1] concatenated to ((char*)&crc)[2] */
it
      /* contains the computed CRC code.             */
        cell[byternk+2]=((char*)&crc)[byternk-48];
...
```

D.2 CRC Code Verification Algorithm

```
...
unsigned gen=0x11021;
unsigned crc;
unsigned short orgcrc;
/* the original CRC code */
char cell[53];
/* cell being received */
char icbyte;
/* the cell's byte currently received */
short byternk;
/* rank of the cell's byte currently received */
short i;
...
if (byternk<4);
else
  if (byternk==4) crc=0;
  else
    if (byternk<7)
      cell[byternk]=((char*)&crc)[byternk-4]=icbyte;
    else
      if (byternk<51)
      { cell[byternk]=((char*)&crc)[3]=icbyte;
        for (i=0; i<8; i++)
        { crc<<=1;
          if (crc & 0x1000000) crc=(crc&~gen)|(~crc&gen);
        }
      }
      else
        if (byternk==51)
        { for (i=0; i<8; i++)
          { crc<<=1;
            if (crc & 0x1000000) crc=(crc&~gen)|(~crc&gen);
          }
          ((char*)&orgcrc)[0]=icbyte;
        }
        else
        { for (i=0; i<8; i++)
          { crc<<=1;
            if (crc & 0x1000000) crc=(crc&~gen)|(~crc&gen);
          }
          ((char*)&orgcrc)[1]=icbyte;
        /* at this point,                                 */
        /* orgcrc contains the original CRC code, and     */
        /* ((char*)&crc)[1] concatenated to ((char*)&crc)[2] */
        /* contains the recomputed CRC code.              */
          ...
        }
...
```

Appendix E

AAL Units' C Data Structures, and Finite State Machine Graphs

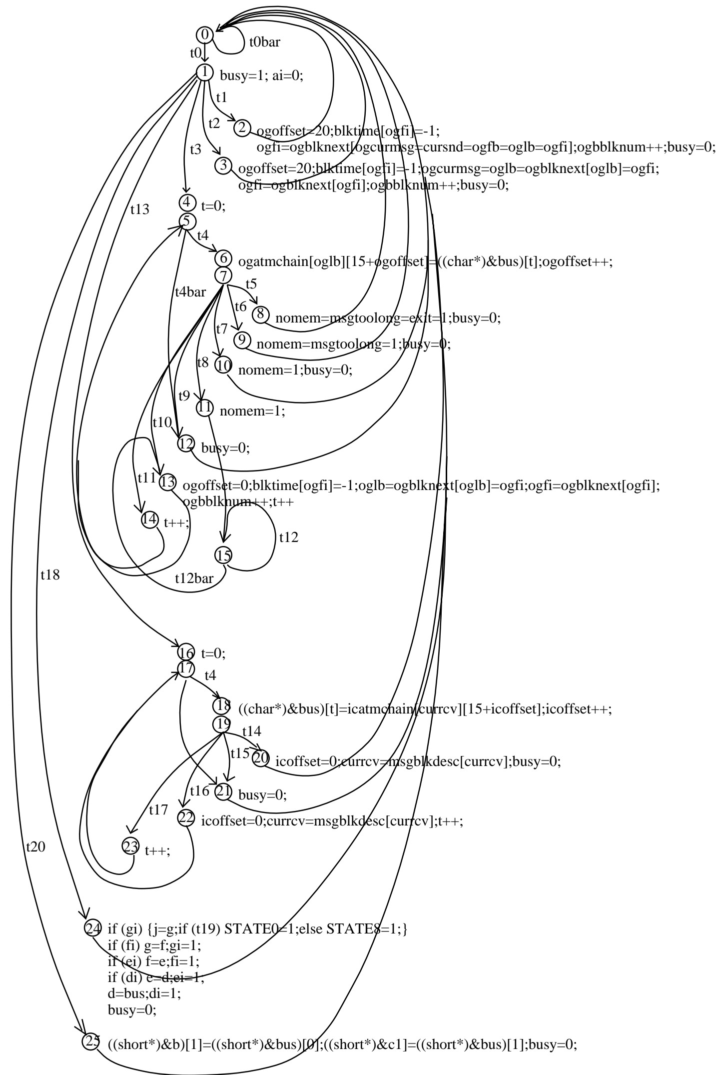
This appendix presents the C Data Structures that describes the AAL units' resources (See Figure 9.14 in Section 9.3.2), and the graphs of the AAL units architecture's finite state machines.

E.1 AAL Units' C Data Structures

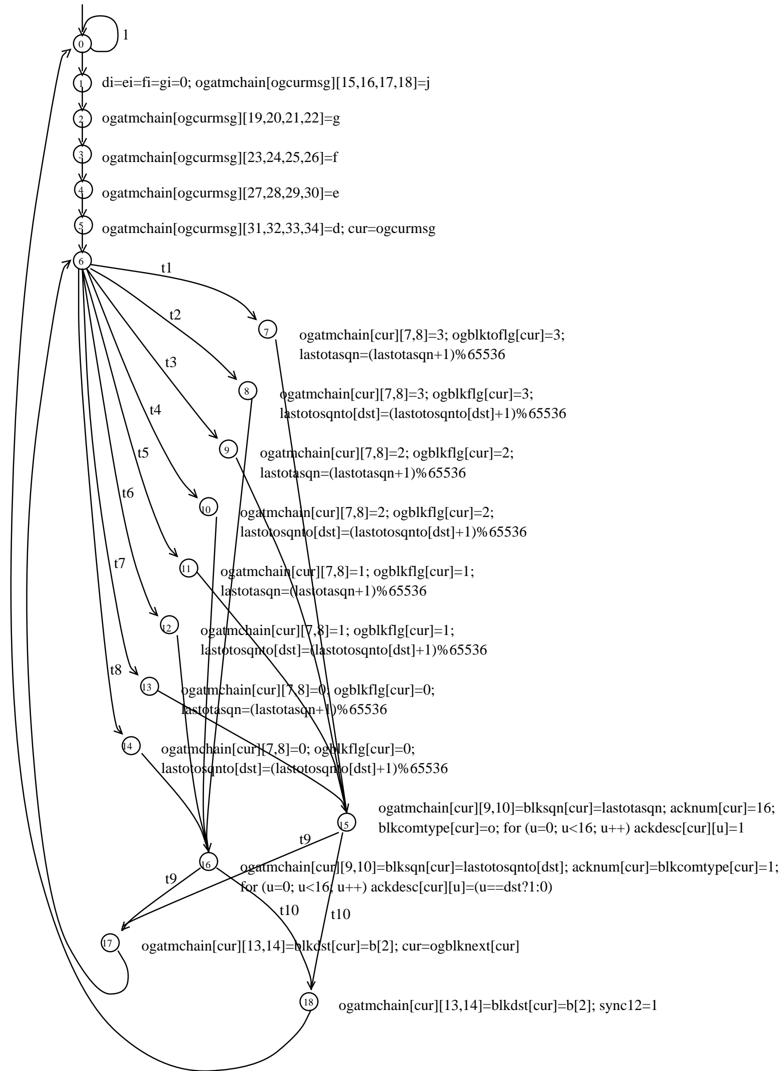
```
struct shmst
{ int bus;
  short busi, cmd, exit, msgtoolong, nomem, busy;
  short read, readi, rlcked, wlcked;
}
struct aal3_4
{ struct shmst *dev;

  char buf, ogbyte;
  short bufi, ogbytei, *ogsuspend;

  char *icbyte;
  short *icbytei, icsuspend;

  char ogatmchain[1887][53];
  int blktime[1887];
  unsigned short ogblknext[1887], blksqn[1887], blkflg[1887];
  unsigned short blkcomtype[1887], ogblktofr[1887];
  unsigned short blkdst[1887], acknum[1887], ackdesc[1887][16];
  unsigned short lastotosqnto[16], lastotasqn, ogbblknum;
  unsigned short ogmsgnum, ogoffset, ogfi, ogli, ogfb, oglb;
  unsigned short ogcurb, ogpreb, ogcurmsg, cursnd, presnd;

  char icatmchain[1887][53];
  unsigned short icblknext[1887], msgblkdesc[1887];
  unsigned short blkinmsg[1887], icblktofr[1887];
  unsigned short lastotosqnfrm[16], lastotasqnfrm[16];
  unsigned short msgnumfrm[16], fimsg[16], limsg[16];
```

```
  unsigned short fbmsg[16], lbmsg[16], curindex[16];
  unsigned short msgdesctab[16][32];

  struct
  { short dst;
    unsigned short complete, hdblk, tlblk;
    unsigned timeout;
    int name[3], inum, id, tab[10];
  } msglsttab[16][32];

  unsigned short icbblknum, icblkinmsgnum, icoffset, msgtype;
  unsigned short flg, sqn, srcpn, comtype, icfi, icli, icfb;
  unsigned short iclb, iccurb, icpreb, cb, currcv;
  short dstpn, dstv;

  char ackbuf[53];

  int b, c, d, e, f, g, j;
  unsigned short bi, ci, di, ei, fi, gi, hd, hi;

  unsigned short state0, t;

  unsigned short state1, cur, u, sync12;

  unsigned short state2, sync23;

  unsigned short state3, vp4, vp4i, vp7, vp7d, vp7i, k;
  unsigned short vpota[4], vpoto[16][4];

  unsigned short state4;
  char h[5];

  unsigned short state5, ig, v;
  unsigned crcg;
  int rtime;

  unsigned short state6, count, iv, rescrc;
  unsigned crcv;

  unsigned short state7, sync73, sync75, ackhdr;

  unsigned short state8, p, q, r;

  unsigned state9, tt, vt, ut;

  unsigned state10, tf, uf, vf, bf, yf, kf, lf, wf;
}
```

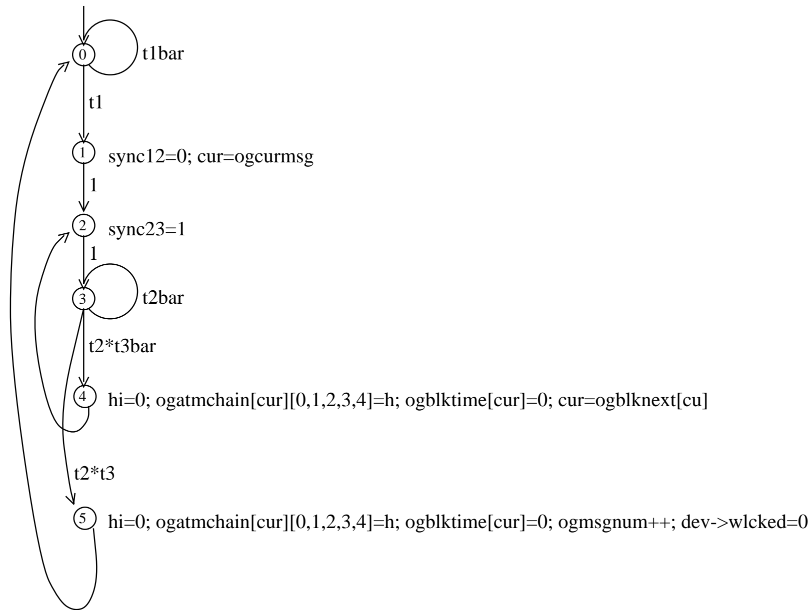
E.2 AAL Units' Finite State Machine 0



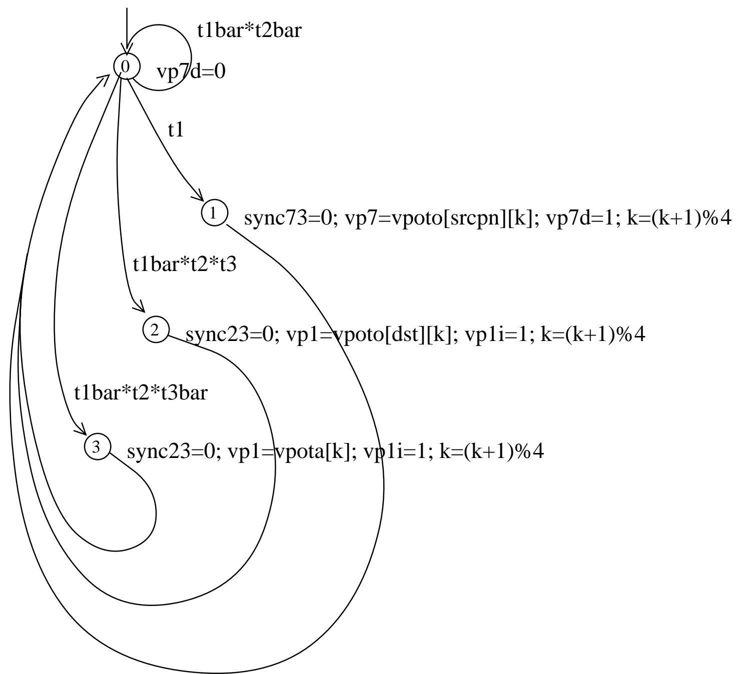
E.3 AAL Units' Finite State Machine 1



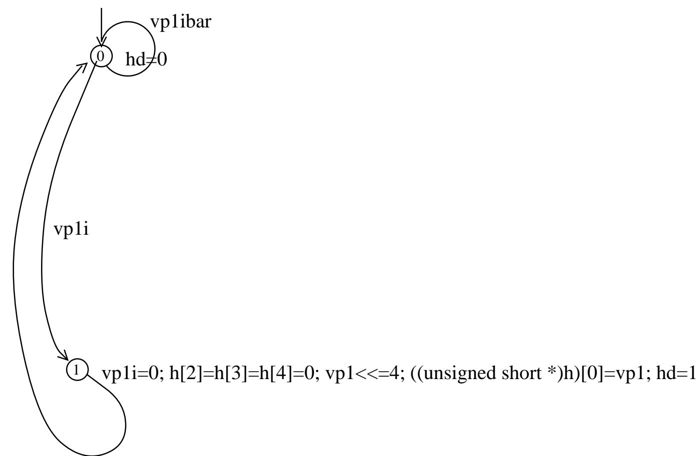
E.4 AAL Units' Finite State Machine 2



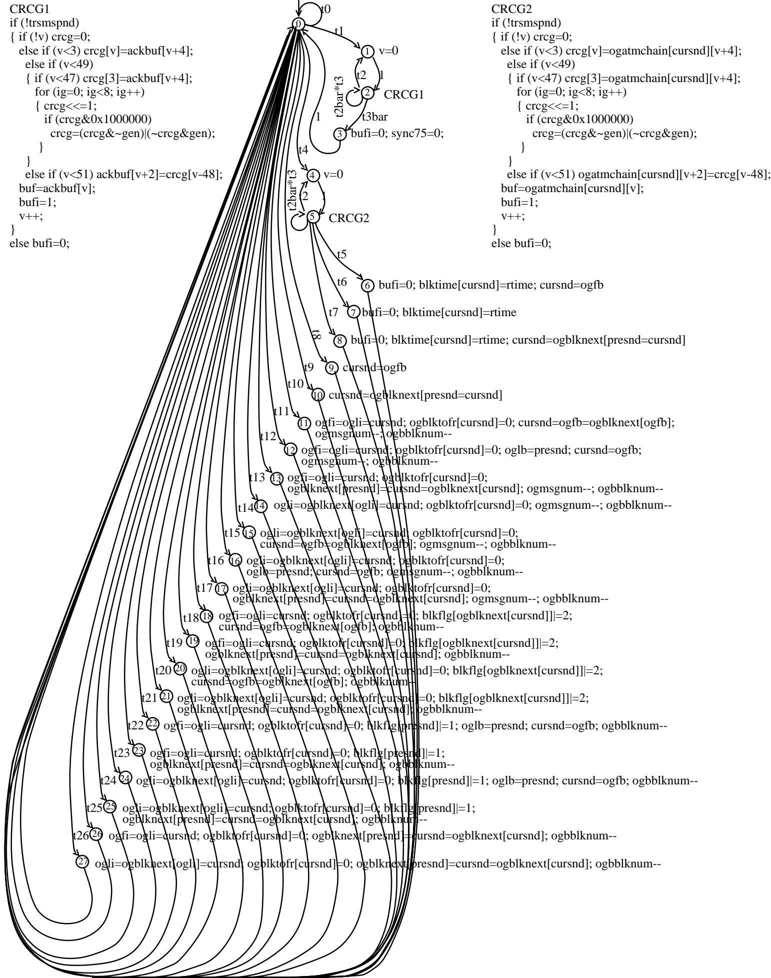
E.5 AAL Units' Finite State Machine 3



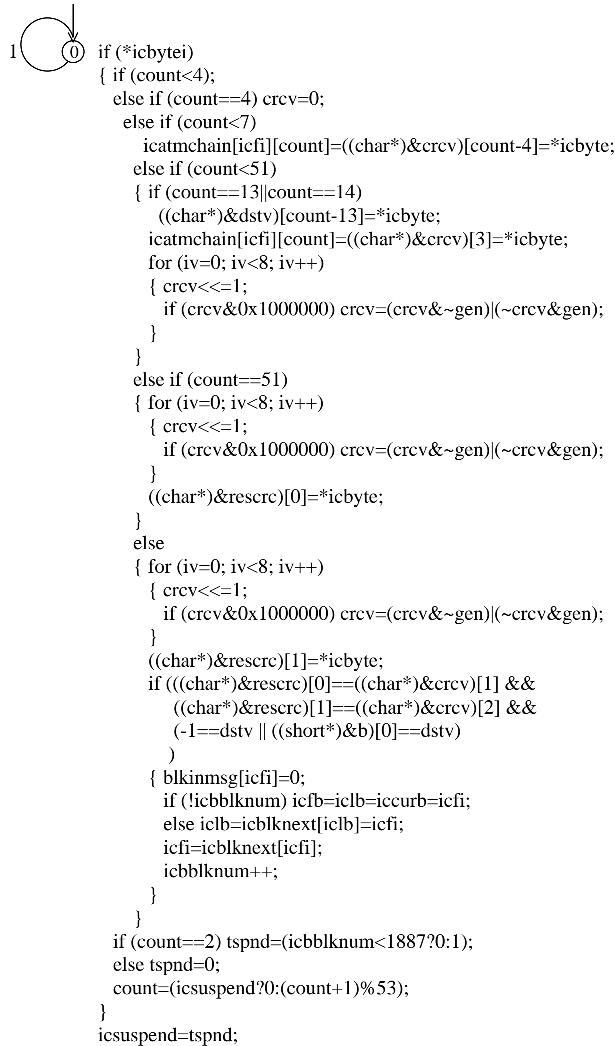
E.6 AAL Units' Finite State Machine 4



E.7 AAL Units' Finite State Machine 5

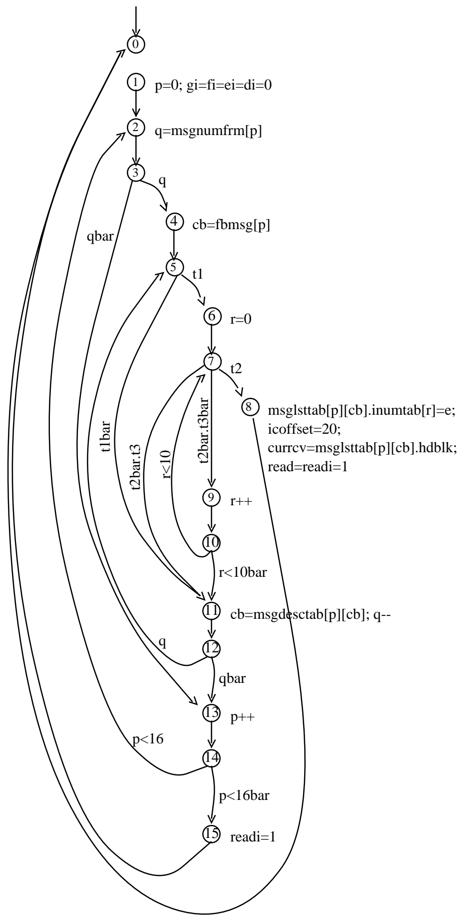


E.8 AAL Units' Finite State Machine 6

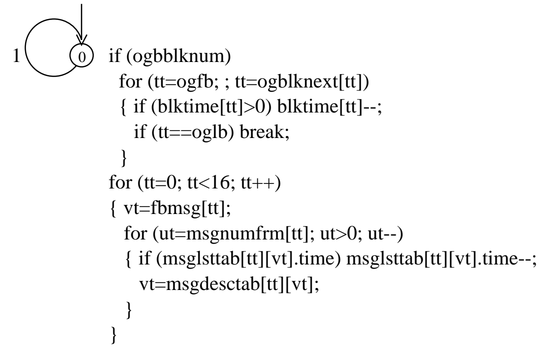


E.9 AAL Units' Finite State Machine 7

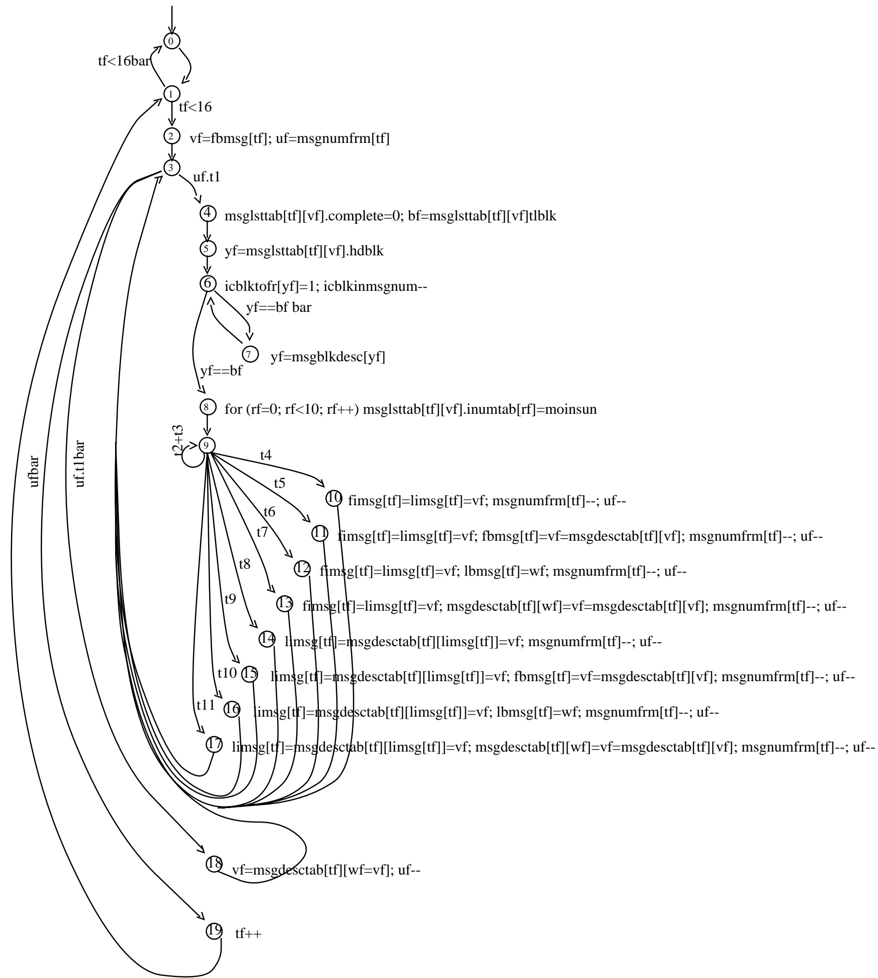
E.10 AAL Units' Finite State Machine 8



E.11 AAL Units' Finite State Machine 9



E.12 AAL Units' Finite State Machine 10



E.13 AAL Units' C Code Skeleton

```
void aal34_scheduler(curaal34)
struct aal3_4 *curaal34;
{ unsigned short tspnd, ind;
  curaal34->ogbyte=curaal34->buf;
  curaal34->ogbytei=curaal34->bufi;
  switch(curaal34->state5)
  { case  0 : ...
    ...
    case 27 : ...
  }
  curaal34->hi=curaal34->hd;
  switch(curaal34->state4)
  { case  0 : ...
    case  1 : ...
  }
  curaal34->vp7i=curaal34->vp7d;
  switch(curaal34->state3)
  { case  0 : ...
    ...
    case  3 : ...
  }
  switch(curaal34->state2)
  { case  0 : ...
    ...
    case  5 : ...
  }
  switch(curaal34->state1)
  { case  0 : ...
    ...
    case 18 : ...
  }
  switch(curaal34->state10)
  { case  0 : ...
    ...
    case 19 : ...
  }
  switch(curaal34->state9)
  { case  0 : ...
  }
  switch(curaal34->state8)
  { case  0 : ...
    ...
    case 15 : ...
  }
  switch(curaal34->state7)
  { case  0 : ...
    ...
    case 46 : ...
  }
  switch(curaal34->state6)
  { case  0 : ...
  }
  switch(curaal34->state0)
  { case  0 : ...
    ...
    case 25 : ...
  }
}
```

Appendix F

PPD Unit Simulation Tests

This appendix presents the stimuli and trace files of three PPD unit simulation tests (See Section 11.1.1).

F.1 Header Translation and Successful Routing of ATM Cells: Stimuli File

```
inputs

  byte
        1 01f2030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
        58 063738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a
  byteind
        1 1
        54 0
        58 1
        111 0
  tstppd.trsltab.wadr
        1 1f
        2 63
  tstppd.trsltab.datain
        1 0708506304102f
        2 00003630000004

outputs

  tstppd.trslreg
  tstppd.xpm_dst
```

F.2 Header Translation and Successful Routing of ATM Cells: Traces File

```
::::::::: clock cycle number 1 :::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 1f
tstppd.trsltab.datain = 7850634102f

byte = 1
byteind = 1

tstppd.(a, b,    o, p, q) = (1, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (0, 0, 0)

::::::::: clock cycle number 2 :::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 63
tstppd.trsltab.datain = 003630004

byte = f2
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (0, 0, 0)

::::::::: clock cycle number 3 :::::::::

byte = 3
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    3, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (0, 0, 0)

::::::::: clock cycle number 4 :::::::::

byte = 4
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    4, 3, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 1, 0)
tstppd.xpmdst[0].(size, head, tail) = (1, 0, 1)
tstppd.xpmdst[0].mem[0] = 10
tstppd.xpmdst[1].(size, head, tail) = (1, 0, 1)
```

```
tstppd.xpmdst[1].mem[0] = 30
tstppd.xpmdst[2].(size, head, tail) = (1, 0, 1)
tstppd.xpmdst[2].mem[0] = 50
tstppd.xpmdst[3].(size, head, tail) = (1, 0, 1)
tstppd.xpmdst[3].mem[0] = 70

:::::::::: clock cycle number 5 ::::::::::

byte = 5
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    5, 4, 3)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 5
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (2, 0, 2)
tstppd.xpmdst[0].mem[1] = 22
tstppd.xpmdst[1].(size, head, tail) = (2, 0, 2)
tstppd.xpmdst[1].mem[1] = 42
tstppd.xpmdst[2].(size, head, tail) = (2, 0, 2)
tstppd.xpmdst[2].mem[1] = 62
tstppd.xpmdst[3].(size, head, tail) = (2, 0, 2)
tstppd.xpmdst[3].mem[1] = 82

:::::::::: clock cycle number 6 ::::::::::

byte = 6
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    6, 5, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 6
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (3, 0, 3)
tstppd.xpmdst[0].mem[2] = 3
tstppd.xpmdst[1].(size, head, tail) = (3, 0, 3)
tstppd.xpmdst[1].mem[2] = 3
tstppd.xpmdst[2].(size, head, tail) = (3, 0, 3)
tstppd.xpmdst[2].mem[2] = 3
tstppd.xpmdst[3].(size, head, tail) = (3, 0, 3)
tstppd.xpmdst[3].mem[2] = 3



...



:::::::::: clock cycle number 53 ::::::::::

byte = 35
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    35, 34, 33)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (50, 0, 50)
tstppd.xpmdst[0].mem[49] = 32
tstppd.xpmdst[1].(size, head, tail) = (50, 0, 50)
tstppd.xpmdst[1].mem[49] = 32
tstppd.xpmdst[2].(size, head, tail) = (50, 0, 50)
tstppd.xpmdst[2].mem[49] = 32
tstppd.xpmdst[3].(size, head, tail) = (50, 0, 50)
tstppd.xpmdst[3].mem[49] = 32

:::::::::: clock cycle number 54 ::::::::::

byte = 35
byteind = 0

tstppd.(a, b,    o, p, q) = (1, f2,    35, 35, 34)
```

```
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (51, 0, 51)
tstppd.xpmdst[0].mem[50] = 33
tstppd.xpmdst[1].(size, head, tail) = (51, 0, 51)
tstppd.xpmdst[1].mem[50] = 33
tstppd.xpmdst[2].(size, head, tail) = (51, 0, 51)
tstppd.xpmdst[2].mem[50] = 33
tstppd.xpmdst[3].(size, head, tail) = (51, 0, 51)
tstppd.xpmdst[3].mem[50] = 33

:::::::::: clock cycle number 55 ::::::::::

byte = 35
byteind = 0

tstppd.(a, b,    o, p, q) = (1, f2,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (52, 0, 52)
tstppd.xpmdst[0].mem[51] = 34
tstppd.xpmdst[1].(size, head, tail) = (52, 0, 52)
tstppd.xpmdst[1].mem[51] = 34
tstppd.xpmdst[2].(size, head, tail) = (52, 0, 52)
tstppd.xpmdst[2].mem[51] = 34
tstppd.xpmdst[3].(size, head, tail) = (52, 0, 52)
tstppd.xpmdst[3].mem[51] = 34

:::::::::: clock cycle number 56 ::::::::::

byte = 35
byteind = 0

tstppd.(a, b,    o, p, q) = (1, f2,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 35
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 57 ::::::::::

byte = 35
byteind = 0

tstppd.(a, b,    o, p, q) = (1, f2,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 35
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 58 ::::::::::

byte = 6
byteind = 1
```

```
tstppd.(a, b,    o, p, q) = (6, f2,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 35
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 59 ::::::::::

byte = 37
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 35
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 60 ::::::::::

byte = 38
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    38, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 35
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 61 ::::::::::

byte = 39
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    39, 38, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 1, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (54, 0, 54)
tstppd.xpmdst[2].mem[53] = 36
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 62 ::::::::::

byte = 3a
byteind = 1
```

```
tstppd.(a, b,    o, p, q) = (6, 37,    3a, 39, 38)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 5
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (55, 0, 55)
tstppd.xpmdst[2].mem[54] = 37
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 63 ::::::::::

byte = 3b
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    3b, 3a, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 6
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (56, 0, 56)
tstppd.xpmdst[2].mem[55] = 38
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35



...



:::::::::: clock cycle number 110 ::::::::::

byte = 6a
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 69, 68)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (103, 0, 103)
tstppd.xpmdst[2].mem[102] = 67
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 111 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 69)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (104, 0, 104)
tstppd.xpmdst[2].mem[103] = 68
```

```
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 112 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (105, 0, 105)
tstppd.xpmdst[2].mem[104] = 69
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 113 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (106, 0, 106)
tstppd.xpmdst[2].mem[105] = 6a
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35

:::::::::: clock cycle number 114 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[0].mem[52] = 35
tstppd.xpmdst[1].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[1].mem[52] = 35
tstppd.xpmdst[2].(size, head, tail) = (106, 0, 106)
tstppd.xpmdst[2].mem[105] = 6a
tstppd.xpmdst[3].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[3].mem[52] = 35
```

F.3 Header Translation and Routing Suspension of ATM Cells: Stimuli File

```
inputs
```

```
  byte
          1 01f2030405
          6 063738393a
  byteind
          1 1
          11 0
  tstppd.trsltab.wadr
          1 1f
          2 63
  tstppd.trsltab.datain
          1 0708506304102f
          2 00003630000008
tstppd.xpm_dst[3].tail
        1 419
tstppd.xpm_dst[3].size
        1 419

outputs

  tstppd.trslreg
  tstppd.xpm_dst
```

F.4 Header Translation and Routing Suspension of ATM Cells: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 1f
tstppd.trsltab.datain = 7850634102f

byte = 1
byteind = 1

tstppd.(a, b,    o, p, q) = (1, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 2 ::::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 63
tstppd.trsltab.datain = 003630008

byte = f2
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0
```

```
:::::::::: clock cycle number 3 ::::::::::

byte = 3
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    3, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 4 ::::::::::

byte = 4
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    4, 3, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 1)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 5 ::::::::::

byte = 5
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    4, 4, 3)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 6 ::::::::::

byte = 6
byteind = 1

tstppd.(a, b,    o, p, q) = (6, f2,    4, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 7 ::::::::::

byte = 37
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    4, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
```

```
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 8 ::::::::::

byte = 38
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    38, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 0,    0, 7, 1)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 9 ::::::::::

byte = 39
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    39, 38, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 0,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 1)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 10 ::::::::::

byte = 3a
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    39, 39, 38)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 0,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 11 ::::::::::

byte = 3a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    39, 39, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 0,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0
```

F.5 Chain of the Header Translation and Routing Suspension, Header Translation and Successful Routing of ATM Cells: Stimuli File

```
inputs

byte
        1 01f2030405
        6 063738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a
byteind
        1 1
        59 0
tstppd.trsltab.wadr
        1 1f
        2 63
tstppd.trsltab.datain
        1 0708506304102f
        2 00003630000004
tstppd.xpm_dst[3].tail
        1 419
tstppd.xpm_dst[3].size
        1 419

outputs

  tstppd.trslreg
  tstppd.xpm_dst
```

F.6 Chain of the Header Translation and Routing Suspension, Header Translation and Successful Routing of ATM Cells: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 1f
tstppd.trsltab.datain = 7850634102f

byte = 1
byteind = 1

tstppd.(a, b,    o, p, q) = (1, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
```

```
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 2 ::::::::::

tstppd.trsltab.wcmd = 1
tstppd.trsltab.wadr = 63
tstppd.trsltab.datain = 003630004

byte = f2
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 3 ::::::::::

byte = 3
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    3, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 1,    30, 42, 1,    50, 62, 1,    70, 82, 1)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 4 ::::::::::

byte = 4
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    4, 3, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 1)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 5 ::::::::::

byte = 5
byteind = 1

tstppd.(a, b,    o, p, q) = (1, f2,    4, 4, 3)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 6 ::::::::::

byte = 6
byteind = 1
```

```
tstppd.(a, b,    o, p, q) = (6, f2,    4, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 7 ::::::::::

byte = 37
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    4, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 22, 0,    30, 42, 0,    50, 62, 0,    70, 82, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 8 ::::::::::

byte = 38
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    38, 4, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (1, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 9 ::::::::::

byte = 39
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    39, 38, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 1, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (1, 0, 1)
tstppd.xpmdst[2].mem[0] = 36
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 10 ::::::::::

byte = 3a
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    3a, 39, 38)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 5
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (2, 0, 2)
tstppd.xpmdst[2].mem[1] = 37
```

```
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 11 ::::::::::

byte = 3b
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    3b, 3a, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 6
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (3, 0, 3)
tstppd.xpmdst[2].mem[2] = 38
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0



...



:::::::::: clock cycle number 58 ::::::::::

byte = 6a
byteind = 1

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 69, 68)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (50, 0, 50)
tstppd.xpmdst[2].mem[49] = 67
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 59 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 69)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (51, 0, 51)
tstppd.xpmdst[2].mem[50] = 68
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 60 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (52, 0, 52)
tstppd.xpmdst[2].mem[51] = 69
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
```

```
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 61 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 6a
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0

:::::::::: clock cycle number 62 ::::::::::

byte = 6a
byteind = 0

tstppd.(a, b,    o, p, q) = (6, 37,    6a, 6a, 6a)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 7, 0,    0, 7, 0,    36, 37, 1,    0, 7, 0)
tstppd.(cind, dind, suspend) = (0, 0, 0)
tstppd.xpmdst[0].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[1].(size, head, tail) = (0, 0, 0)
tstppd.xpmdst[2].(size, head, tail) = (53, 0, 53)
tstppd.xpmdst[2].mem[52] = 6a
tstppd.xpmdst[3].(size, head, tail) = (419, 0, 419)
tstppd.xpmdst[3].mem[418] = 0
```

Appendix G

SPD Unit Simulation Tests

This appendix presents the stimuli and trace files of three SPD unit simulation tests (See Section 11.1.2).

G.1 Successful Transmission of ATM Cells: Stimuli File

```
inputs

  fifo0
          1 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
          54 4142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f707172737475
  fifo1
          108 8182838485868788898a8b8c8d8e8f909192939495969798999a9b9c9d9e9fa0a1a2a3a4a5a6a7a8a9aaabacadaeafb0b1b2b3b4b5
  fifo2
          60 c1c2c3c4c5c6c7c8c9cacbcccdcecfd0d1d2d3d4d5d6d7d8d9dadbdcdddedfe0e1e2e3e4e5e6e7e8e9eaebecedeeeff0f1f2f3f4f5

outputs

  tstspd.xpm_src
  *tstspd.suspend
  tstspd.reg
  tstspd.buf
  tstspd.byte
  tstspd.bufind
  tstspd.byteind
```

G.2 Successful Transmission of ATM Cells: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
```

```
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (0, 0)
tstspd.(bufind, byteind) = (0, 0)

:::::::::: clock cycle number 2 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (1, 2, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (1, 0)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 3 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (2, 3, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (2, 1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 4 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (3, 4, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)
tstspd.(buf, byte) = (3, 2)
tstspd.(bufind, byteind) = (1, 1)



...



:::::::::: clock cycle number 52 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (51, 52, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (51, 0, 0, 0)
tstspd.(buf, byte) = (33, 32)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 53 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (52, 53, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (52, 0, 0, 0)
tstspd.(buf, byte) = (34, 33)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 54 ::::::::::
```

```
tstspd.xpm_src[0]->(head, tail, size) = (53, 54, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (35, 34)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 55 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (54, 55, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 53, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (41, 35)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 56 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (55, 56, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 53, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (42, 41)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 57 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (56, 57, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 53, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)
tstspd.(buf, byte) = (43, 42)
tstspd.(bufind, byteind) = (1, 1)



...



:::::::::: clock cycle number 105 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (104, 105, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 46, 46)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 53, 1)
tstspd.iqcnt[0,1,2,3] = (51, 0, 0, 0)
tstspd.(buf, byte) = (73, 72)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 106 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (105, 106, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 47, 47)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 53, 1)
tstspd.iqcnt[0,1,2,3] = (52, 0, 0, 0)
tstspd.(buf, byte) = (74, 73)
tstspd.(bufind, byteind) = (1, 1)
```

```
:::::::::: clock cycle number 107 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 48, 48)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 53, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (75, 74)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 108 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[2]->(head, tail, size) = (1, 49, 48)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 1, 0)
tstspd.(buf, byte) = (c1, 75)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 109 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 2, 2)
tstspd.xpm_src[2]->(head, tail, size) = (2, 50, 48)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 2, 0)
tstspd.(buf, byte) = (c2, c1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 110 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 3, 3)
tstspd.xpm_src[2]->(head, tail, size) = (3, 51, 48)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 3, 0)
tstspd.(buf, byte) = (c3, c2)
tstspd.(bufind, byteind) = (1, 1)



...



:::::::::: clock cycle number 158 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 51, 51)
tstspd.xpm_src[2]->(head, tail, size) = (51, 53, 2)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 51, 0)
tstspd.(buf, byte) = (f3, f2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 159 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 52, 52)
tstspd.xpm_src[2]->(head, tail, size) = (52, 53, 1)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
```

```
tstspd.iqcnt[0,1,2,3] = (0, 0, 52, 0)
tstspd.(buf, byte) = (f4, f3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 160 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (f5, f4)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 161 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (1, 53, 52)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 1, 0, 0)
tstspd.(buf, byte) = (81, f5)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 162 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (2, 53, 51)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 2, 0, 0)
tstspd.(buf, byte) = (82, 81)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 163 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (3, 53, 50)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 3, 0, 0)
tstspd.(buf, byte) = (83, 82)
tstspd.(bufind, byteind) = (1, 1)



...



:::::::::: clock cycle number 211 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (51, 53, 2)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 51, 0, 0)
tstspd.(buf, byte) = (b3, b2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 212 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (52, 53, 1)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
```

```
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 52, 0, 0)
tstspd.(buf, byte) = (b4, b3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 213 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 0)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (b5, b4)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 214 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 0)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (b5, b5)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 215 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (106, 106, 0)
tstspd.xpm_src[1]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 0)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (b5, b5)
tstspd.(bufind, byteind) = (0, 0)
```

G.3 ATM Cell Transmissions Suspension: Stimuli File

```
inputs

  fifo0
          1 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
  fifo1
          8 8182838485868788898a8b8c8d8e8f909192939495969798999a9b9c9d9e9fa0a1a2a3a4a5a6a7a8a9aaabacadaeafb0b1b2b3b4b5
  fifo2
          8 c1c2c3c4c5c6c7c8c9cacbcccdcecfd0d1d2d3d4d5d6d7d8d9dadbdcdddedfe0e1e2e3e4e5e6e7e8e9eaebecedeeeff0f1f2f3f4f5
  suspend
          7 1
          8 0
          13 1
          14 0
          19 1
          20 0
          25 1
          26 0

outputs

  tstspd.xpm_src
  *tstspd.suspend
  tstspd.reg
```

```
tstspd.buf
tstspd.byte
tstspd.bufind
tstspd.byteind
```

G.4 ATM Cell Transmissions Suspension: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (0, 0)
tstspd.(bufind, byteind) = (0, 0)

:::::::::: clock cycle number 2 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (1, 2, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (1, 0)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 3 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (2, 3, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (2, 1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 4 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (3, 4, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)
tstspd.(buf, byte) = (3, 2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 5 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (4, 5, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (4, 0, 0, 0)
tstspd.(buf, byte) = (4, 3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 6 ::::::::::
```

```
tstspd.xpm_src[0]->(head, tail, size) = (5, 6, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (5, 0, 0, 0)
tstspd.(buf, byte) = (5, 4)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 7 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 7, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (5, 5)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 8 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (1, 8, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[2]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (1, 5)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 9 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (2, 9, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 2, 2)
tstspd.xpm_src[2]->(head, tail, size) = (0, 2, 2)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (2, 1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 10 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (3, 10, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 3, 3)
tstspd.xpm_src[2]->(head, tail, size) = (0, 3, 3)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)
tstspd.(buf, byte) = (3, 2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 11 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (4, 11, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 4, 4)
tstspd.xpm_src[2]->(head, tail, size) = (0, 4, 4)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (4, 0, 0, 0)
tstspd.(buf, byte) = (4, 3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 12 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (5, 12, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 5, 5)
```

```
tstspd.xpm_src[2]->(head, tail, size) = (0, 5, 5)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (5, 0, 0, 0)
tstspd.(buf, byte) = (5, 4)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 13 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 13, 13)
tstspd.xpm_src[1]->(head, tail, size) = (0, 6, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 6, 6)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (5, 5)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 14 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 14, 14)
tstspd.xpm_src[1]->(head, tail, size) = (1, 7, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 7, 7)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 1, 0, 0)
tstspd.(buf, byte) = (81, 5)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 15 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 15, 15)
tstspd.xpm_src[1]->(head, tail, size) = (2, 8, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 8, 8)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 2, 0, 0)
tstspd.(buf, byte) = (82, 81)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 16 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 16, 16)
tstspd.xpm_src[1]->(head, tail, size) = (3, 9, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 9, 9)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 3, 0, 0)
tstspd.(buf, byte) = (83, 82)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 17 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 17, 17)
tstspd.xpm_src[1]->(head, tail, size) = (4, 10, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 10, 10)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 4, 0, 0)
tstspd.(buf, byte) = (84, 83)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 18 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 18, 18)
tstspd.xpm_src[1]->(head, tail, size) = (5, 11, 6)
tstspd.xpm_src[2]->(head, tail, size) = (0, 11, 11)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
```

```
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 5, 0, 0)
tstspd.(buf, byte) = (85, 84)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 19 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 19, 19)
tstspd.xpm_src[1]->(head, tail, size) = (0, 12, 12)
tstspd.xpm_src[2]->(head, tail, size) = (0, 12, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (85, 85)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 20 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 20, 20)
tstspd.xpm_src[1]->(head, tail, size) = (0, 13, 13)
tstspd.xpm_src[2]->(head, tail, size) = (1, 13, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 1, 0)
tstspd.(buf, byte) = (c1, 85)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 21 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 21, 21)
tstspd.xpm_src[1]->(head, tail, size) = (0, 14, 14)
tstspd.xpm_src[2]->(head, tail, size) = (2, 14, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 2, 0)
tstspd.(buf, byte) = (c2, c1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 22 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 22, 22)
tstspd.xpm_src[1]->(head, tail, size) = (0, 15, 15)
tstspd.xpm_src[2]->(head, tail, size) = (3, 15, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 3, 0)
tstspd.(buf, byte) = (c3, c2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 23 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 23, 23)
tstspd.xpm_src[1]->(head, tail, size) = (0, 16, 16)
tstspd.xpm_src[2]->(head, tail, size) = (4, 16, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 4, 0)
tstspd.(buf, byte) = (c4, c3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 24 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 24, 24)
tstspd.xpm_src[1]->(head, tail, size) = (0, 17, 17)
tstspd.xpm_src[2]->(head, tail, size) = (5, 17, 12)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 5, 0)
tstspd.(buf, byte) = (c5, c4)
```

```
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 25 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 25, 25)
tstspd.xpm_src[1]->(head, tail, size) = (0, 18, 18)
tstspd.xpm_src[2]->(head, tail, size) = (0, 18, 18)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (c5, c5)
tstspd.(bufind, byteind) = (0, 1)
```

G.5 Transmission Suspension and Successful Transmission Chain: Stimuli File

```
inputs

  fifo0
          1 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
  fifo1
          8 8182838485868788898a8b8c8d8e8f909192939495969798999a9b9c9d9e9fa0a1a2a3a4a5a6a7a8a9aaabacadaeafb0b1b2b3b4b5
  fifo2
          8 c1c2c3c4c5c6c7c8c9cacbcccdcecfd0d1d2d3d4d5d6d7d8d9dadbdcdddedfe0e1e2e3e4e5e6e7e8e9eaebecedeeeff0f1f2f3f4f5
  suspend
          7 1
          8 0
          66 1
          67 0

outputs

  tstspd.xpm_src
  *tstspd.suspend
  tstspd.reg
  tstspd.buf
  tstspd.byte
  tstspd.bufind
  tstspd.byteind
```

G.6 Transmission Suspension and Successful Transmission Chain: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (0, 0)
tstspd.(bufind, byteind) = (0, 0)
```

```
:::::::::: clock cycle number 2 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (1, 2, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (1, 0)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 3 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (2, 3, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (2, 1)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 4 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (3, 4, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)
tstspd.(buf, byte) = (3, 2)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 5 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (4, 5, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (4, 0, 0, 0)
tstspd.(buf, byte) = (4, 3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 6 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (5, 6, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (5, 0, 0, 0)
tstspd.(buf, byte) = (5, 4)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 7 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (0, 7, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[2]->(head, tail, size) = (0, 0, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (5, 5)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 8 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (1, 8, 7)
```

```
tstspd.xpm_src[1]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[2]->(head, tail, size) = (0, 1, 1)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)
tstspd.(buf, byte) = (1, 5)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 9 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (2, 9, 7)
tstspd.xpm_src[1]->(head, tail, size) = (0, 2, 2)
tstspd.xpm_src[2]->(head, tail, size) = (0, 2, 2)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)
tstspd.(buf, byte) = (2, 1)
tstspd.(bufind, byteind) = (1, 1)



...



:::::::::: clock cycle number 59 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (52, 53, 1)
tstspd.xpm_src[1]->(head, tail, size) = (0, 52, 52)
tstspd.xpm_src[2]->(head, tail, size) = (0, 52, 52)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (52, 0, 0, 0)
tstspd.(buf, byte) = (34, 33)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 60 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (35, 34)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 61 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (1, 53, 52)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 1, 0, 0)
tstspd.(buf, byte) = (81, 35)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 62 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (2, 53, 51)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 2, 0, 0)
tstspd.(buf, byte) = (82, 81)
tstspd.(bufind, byteind) = (1, 1)
```

```
:::::::::: clock cycle number 63 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (3, 53, 50)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 3, 0, 0)
tstspd.(buf, byte) = (83, 82)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 64 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (4, 53, 49)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 4, 0, 0)
tstspd.(buf, byte) = (84, 83)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 65 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (5, 53, 48)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 5, 0, 0)
tstspd.(buf, byte) = (85, 84)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 66 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 1
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (85, 85)
tstspd.(bufind, byteind) = (0, 1)

:::::::::: clock cycle number 67 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (1, 53, 52)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 1, 0)
tstspd.(buf, byte) = (c1, 85)
tstspd.(bufind, byteind) = (1, 0)

:::::::::: clock cycle number 68 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (2, 53, 51)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 2, 0)
tstspd.(buf, byte) = (c2, c1)
tstspd.(bufind, byteind) = (1, 1)



...
```

```
:::::::::: clock cycle number 118 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (52, 53, 1)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 52, 0)
tstspd.(buf, byte) = (f4, f3)
tstspd.(bufind, byteind) = (1, 1)

:::::::::: clock cycle number 119 ::::::::::

tstspd.xpm_src[0]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[1]->(head, tail, size) = (0, 53, 53)
tstspd.xpm_src[2]->(head, tail, size) = (53, 53, 0)
tstspd.xpm_src[3]->(head, tail, size) = (0, 0, 0)
suspend = 0
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)
tstspd.(buf, byte) = (f5, f4)
tstspd.(bufind, byteind) = (1, 1)
```

Appendix H

SPD PPD Unit Link Simulation Tests

This appendix presents the stimuli and trace files of the simulation test of the upstream SPD and downstream PPD unit links (See Section 11.1.3).

H.1 SPD PPD Unit Link Simulation Test: Stimuli File

```
inputs

  fifo0
         1 0432030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
  fifo1
         58 00124344454647484 94a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f707172737475
  fifo2
         2 039738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a
  tstppd.trsltab.wadr
         1 43
         2 39
         58 1
  tstppd.trsltab.datain
         1 00000100000004
         2 0708506304102f
         58 00000000004141
  tstppd.xpm_dst[2].tail
         1 371
  tstppd.xpm_dst[2].size
         1 371

outputs

  tstspd.xpm_src
  tstspd.reg
  tstspd.byte
  tstppd.reg
  tstppd.xpm_dst
```

H.2 SPD PPD Unit Link Simulation Test: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (0, 1, 1,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)

tstspd.(buf, byte) = (0, 0)
tstspd.(bufind, byteind) = (0, 0)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 371, 371,    0, 0, 0)
tstppd.xpmdst[2].mem[370] = 0

:::::::::: clock cycle number 2 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (1, 2, 1,    0, 0, 0,    0, 1, 1,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (1, 0, 0, 0)

tstspd.(buf, byte) = (4, 0)
tstspd.(bufind, byteind) = (1, 0)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 371, 371,    0, 0, 0)
tstppd.xpmdst[2].mem[370] = 0

:::::::::: clock cycle number 3 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (2, 3, 1,    0, 0, 0,    0, 2, 2,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (2, 0, 0, 0)

tstspd.(buf, byte) = (32, 4)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 0,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 371, 371,    0, 0, 0)
tstppd.xpmdst[2].mem[370] = 0

:::::::::: clock cycle number 4 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (3, 4, 1,    0, 0, 0,    0, 3, 3,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (3, 0, 0, 0)

tstspd.(buf, byte) = (3, 32)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    0, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 0, 0,    0, 0, 0,    0, 0, 0,    0, 0, 0)
tstppd.(cind, dind) = (0, 0)
```

```
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 371, 371,    0, 0, 0)
tstppd.xpmdst[2].mem[370] = 0

:::::::::: clock cycle number 5 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (4, 5, 1,    0, 0, 0,    0, 4, 4,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (4, 0, 0, 0)

tstspd.(buf, byte) = (4, 3)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    3, 0, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (1, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 371, 371,    0, 0, 0)
tstppd.xpmdst[2].mem[370] = 0

:::::::::: clock cycle number 6 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (5, 6, 1,    0, 0, 0,    0, 5, 5,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (5, 0, 0, 0)

tstspd.(buf, byte) = (5, 4)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    4, 3, 0)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 1)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 372, 372,    0, 0, 0)
tstppd.xpmdst[2].mem[371] = 1

:::::::::: clock cycle number 7 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (6, 7, 1,    0, 0, 0,    0, 6, 6,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (6, 0, 0, 0)

tstspd.(buf, byte) = (6, 5)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    5, 4, 3)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 5
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 373, 373,    0, 0, 0)
tstppd.xpmdst[2].mem[372] = 2

:::::::::: clock cycle number 8 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (7, 8, 1,    0, 0, 0,    0, 7, 7,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (0, 0, 1)
tstspd.iqcnt[0,1,2,3] = (7, 0, 0, 0)

tstspd.(buf, byte) = (7, 6)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    6, 5, 4)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 6
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 374, 374,    0, 0, 0)
tstppd.xpmdst[2].mem[373] = 3
```

```
...


:::::::::: clock cycle number 54 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 0, 0,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)

tstspd.(buf, byte) = (35, 34)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    34, 33, 32)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 52
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 420, 420,    0, 0, 0)
tstppd.xpmdst[2].mem[419] = 31

:::::::::: clock cycle number 55 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 0, 0,    1, 53, 52,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 1, 0)

tstspd.(buf, byte) = (3, 35)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (4, 32,    35, 34, 33)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 421, 421,    0, 0, 0)
tstppd.xpmdst[2].mem[420] = 32

:::::::::: clock cycle number 56 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 0, 0,    2, 53, 51,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 2, 0)

tstspd.(buf, byte) = (97, 3)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 32,    35, 35, 34)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 1, 1)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 422, 422,    0, 0, 0)
tstppd.xpmdst[2].mem[421] = 33

:::::::::: clock cycle number 57 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 0, 0,    3, 53, 50,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 3, 0)

tstspd.(buf, byte) = (38, 97)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    35, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 1)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (0, 2, 0,    0, 2, 0,    1, 2, 1,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
```

```
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 423, 423,    0, 0, 0)
tstppd.xpmdst[2].mem[422] = 34

:::::::::: clock cycle number 58 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 1, 1,    4, 53, 49,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 4, 0)

tstspd.(buf, byte) = (39, 38)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    38, 35, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 1,    30, 47, 1,    50, 67, 1,    70, 87, 1)
tstppd.(cind, dind) = (1, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 59 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 2, 2,    5, 53, 48,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 5, 0)

tstspd.(buf, byte) = (3a, 39)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 1

tstppd.(a, b,    o, p, q) = (3, 97,    39, 38, 35)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 60 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    0, 3, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)

tstspd.(buf, byte) = (3a, 3a)
tstspd.(bufind, byteind) = (0, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    39, 39, 38)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 61 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    1, 4, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 1, 0, 0)

tstspd.(buf, byte) = (0, 3a)
tstspd.(bufind, byteind) = (1, 0)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    39, 39, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35
```

```
:::::::::: clock cycle number 62 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    2, 5, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 2, 0, 0)

tstspd.(buf, byte) = (12, 0)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 97,    39, 39, 39)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 0, 0)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 63 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    3, 6, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 3, 0, 0)

tstspd.(buf, byte) = (43, 12)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    39, 39, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 0)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 64 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    4, 7, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 4, 0, 0)

tstspd.(buf, byte) = (44, 43)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    43, 39, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (1, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 0, 0,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 65 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    5, 8, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 5, 0, 0)

tstspd.(buf, byte) = (45, 44)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    44, 43, 39)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 0)
tstppd.count = 4
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 1)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 1, 1,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[0] = 41
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 66 ::::::::::
```

```
tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    6, 9, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 6, 0, 0)

tstspd.(buf, byte) = (46, 45)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    45, 44, 43)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 5
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 2, 2,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[1] = 42
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 67 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    7, 10, 3,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (1, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 7, 0, 0)

tstspd.(buf, byte) = (47, 46)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    46, 45, 44)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 6
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 3, 3,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[2] = 43
tstppd.xpmdst[2].mem[423] = 35



...



:::::::::: clock cycle number 113 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)

tstspd.(buf, byte) = (75, 74)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    74, 73, 72)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 52
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 49, 49,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[48] = 71
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 114 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    1, 53, 52,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 1, 0)

tstspd.(buf, byte) = (3, 75)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (0, 12,    75, 74, 73)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 1, 1)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
```

```
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 50, 50,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[49] = 72
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 115 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    2, 53, 51,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 2, 0)

tstspd.(buf, byte) = (97, 3)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 12,    75, 75, 74)
tstppd.(aind, bind,    oind, pind, qind) = (1, 0,    0, 1, 1)
tstppd.count = 1
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 51, 51,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[50] = 73
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 116 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    3, 53, 50,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 3, 0)

tstspd.(buf, byte) = (38, 97)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    75, 75, 75)
tstppd.(aind, bind,    oind, pind, qind) = (0, 1,    0, 0, 1)
tstppd.count = 2
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (41, 42, 1,    0, 2, 0,    0, 2, 0,    0, 2, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 52, 52,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[51] = 74
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 117 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    4, 53, 49,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 4, 0)

tstspd.(buf, byte) = (39, 38)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    38, 75, 75)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 3
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 1,    30, 47, 1,    50, 67, 1,    70, 87, 1)
tstppd.(cind, dind) = (1, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 53, 53,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[52] = 75
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 118 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    5, 53, 48,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 5, 0)

tstspd.(buf, byte) = (3a, 39)
tstspd.(bufind, byteind) = (1, 1)
tstppd.suspend = 1

tstppd.(a, b,    o, p, q) = (3, 97,    39, 38, 75)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    1, 0, 0)
tstppd.count = 4
```

```
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 53, 53,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[52] = 75
tstppd.xpmdst[2].mem[423] = 35

:::::::::: clock cycle number 119 ::::::::::

tstspd.xpm_src[0, 1, 2, 3]->(head, tail, size) = (53, 53, 0,    53, 53, 0,    0, 53, 53,    0, 0, 0)
tstspd.(siqnum, lsthead, trsmit) = (2, 0, 1)
tstspd.iqcnt[0,1,2,3] = (0, 0, 0, 0)

tstspd.(buf, byte) = (3a, 3a)
tstspd.(bufind, byteind) = (0, 1)
tstppd.suspend = 0

tstppd.(a, b,    o, p, q) = (3, 97,    39, 39, 38)
tstppd.(aind, bind,    oind, pind, qind) = (0, 0,    0, 0, 0)
tstppd.count = 0
tstppd.(c, d, e,    f, g, h,    i, j, k,    l, m, n) = (10, 27, 0,    30, 47, 0,    50, 67, 0,    70, 87, 0)
tstppd.(cind, dind) = (0, 0)
tstppd.xpm_dst[0, 1, 2, 3].(head, tail, size) = (0, 53, 53,    0, 0, 0,    0, 0, 424,    0, 0, 0)
tstppd.xpmdst[0].mem[52] = 75
tstppd.xpmdst[2].mem[423] = 35
```

Appendix I

BMX Switch Simulation Test

This appendix presents the stimuli and trace files of the BMX switch simulation test (See Section 11.1.4).

I.1 BMX Switch Simulation Test: Stimuli File

```
inputs

  byte1
        1 0162030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435
  byteind1
        1 1
        54 0
  byte2
        1 0112434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f707172737475
  byteind2
        1 1
        54 0
  tstppd.trsltab.datain1
        1 00000000100002
  tstppd.trsltab.wadr1
        1 16
  tstppd.trsltab.wcmd1
        1 1
        2 0
  tstppd.trsltab.datain2
        1 04d44c44b44a4f
  tstppd.trsltab.wadr2
        1 11
  tstppd.trsltab.wcmd2
        1 1
        2 0

outputs

  byte
  tstppd.suspend
  tstspd.byte
  suspend
```

I.2 BMX Switch Simulation Test: Traces File

```
:::::::::: clock cycle number 1 ::::::::::

byte[0,1,2,3] = (0, 1, 1, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (0, 0,    0, 0,    0, 0,    0, 0)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 2 ::::::::::

byte[0,1,2,3] = (0, 62, 12, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (0, 0,    0, 0,    0, 0,    0, 0)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 3 ::::::::::

byte[0,1,2,3] = (0, 3, 43, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (0, 0,    0, 0,    0, 0,    0, 0)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 4 ::::::::::

byte[0,1,2,3] = (0, 4, 44, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (0, 0,    0, 0,    0, 0,    0, 0)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 5 ::::::::::

byte[0,1,2,3] = (0, 5, 45, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (0, 0,    0, 0,    0, 0,    0, 0)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 6 ::::::::::

byte[0,1,2,3] = (0, 6, 46, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (4a, 0,    1, 0,    4c, 0,    4d, 0)
tstspd[0,1,2,3].(bufind, byteind) = (1, 0,    1, 0,    1, 0,    1, 0)
```

```
suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 7 ::::::::::

byte[0,1,2,3] = (0, 7, 47, 0)
byteind[0,1,2,3] = (0, 1, 1, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (42, 4a,    2, 1,    42, 4c,    42, 4d)
tstspd[0,1,2,3].(bufind, byteind) = (1, 1,    1, 1,    1, 1,    1, 1)

suspend[0,1,2,3] = (0, 0, 0, 0)



...



:::::::::: clock cycle number 57 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (74, 73,    34, 33,    74, 73,    74, 73)
tstspd[0,1,2,3].(bufind, byteind) = (1, 1,    1, 1,    1, 1,    1, 1)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 58 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 74,    35, 34,    75, 74,    75, 74)
tstspd[0,1,2,3].(bufind, byteind) = (1, 1,    1, 1,    1, 1,    1, 1)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 59 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    4b, 35,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 1,    1, 1,    0, 1,    0, 1)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 60 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    42, 4b,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    1, 1,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)



...
```

```
:::::::::: clock cycle number 110 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    74, 73,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    1, 1,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 111 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    75, 74,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    1, 1,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 112 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    75, 75,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 1,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 113 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    75, 75,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 114 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    75, 75,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)

:::::::::: clock cycle number 115 ::::::::::

byte[0,1,2,3] = (0, 35, 75, 0)
byteind[0,1,2,3] = (0, 0, 0, 0)

tstppd[0,1,2,3].suspend = (0, 0, 0, 0)

tstspd[0,1,2,3].(buf, byte) = (75, 75,    75, 75,    75, 75,    75, 75)
tstspd[0,1,2,3].(bufind, byteind) = (0, 0,    0, 0,    0, 0,    0, 0)

suspend[0,1,2,3] = (0, 0, 0, 0)
```

Appendix J

AAL Units Simulation Test: Traces File

This appendix presents the traces file of the AAL units simulation test described in Section 11.1.5.

```
:::::::::: clock cycle number 0 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (0, 0, 0, 0, 0, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 1 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (0, 0, 0, 0, 0, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 2 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (0, 0, 0, 0, 0, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (-1, 0, 0, 0, 0)
```

```
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 3 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (0, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 59 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (0, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (-1, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 14, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 60 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 61 ::::::::::
```

```
aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 62 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 13, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 63 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 1, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (4, 0)
aal34[1].(bufi, ogbytei) = (1, 0)

aal34[5].icbyte = 0
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 14, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 81 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (5, 0, 0, 0, 0, 5, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (72, 0)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 0
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

```
:::::::::: clock cycle number 82 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (6, 0, 0, 0, 0, 5, 0, 0, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 72)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 0
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 1
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 115 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (0, 0, 65535, 16, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 4a)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 1
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 34
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 13, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 116 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 1)

aal34[5].icbyte = 0
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 35
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 14, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 133 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)
```

```
aal34[5].icbyte = 78
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 52
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 0, 0, 0, 0, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 134 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (-1, 0, 0, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 147 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 5, 0, 0, 0, 0, 26, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (-1, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 27, 13, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 148 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 27, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 1, 0, 29, 14, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 149 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 1, 0, 29, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
```

```
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 30, 2, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 150 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 30, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (78, 78)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 32, 3, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 156 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 41, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (0, 0)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 42, 14, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (0, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 157 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 42, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (1, 0)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 45, 2, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

```
...

:::::::::: clock cycle number 202 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (be, 0)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (17, 0, 0, 0, 0, 3, 0, 0, 0, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 203 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (22, be)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 204 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (22, 22)
aal34[1].(bufi, ogbytei) = (0, 1)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 205 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (22, 22)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (23, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

```
:::::::::: clock cycle number 206 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 18)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (22, 22)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 207 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (1, 22)
aal34[1].(bufi, ogbytei) = (1, 0)

aal34[5].icbyte = 78
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 225 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 5, 0, 2, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (72, 0)
aal34[1].(bufi, ogbytei) = (1, 1)

aal34[5].icbyte = 0
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 259 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 6, 0, 16, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (0, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, df)
aal34[1].(bufi, ogbytei) = (1, 1)
```

```
aal34[5].icbyte = 1
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 34
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 260 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399855, 0, 65535, 15, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 1)

aal34[5].icbyte = 0
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 35
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 13, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

...

:::::::::: clock cycle number 277 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 10, 0, 1, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399838, 0, 65535, 15, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 1
aal34[5].icsuspend = 0
aal34[5].count = 52
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 14, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 1, 0, 0, 1, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 278 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399981, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

...
```

```
:::::::::: clock cycle number 295 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 9, 0, 16, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399964, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 31, 3, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 2, 0, 1, 2, 1886, 1)
aal34[5].(blkinmsg[1], icblktofr[1]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 302 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399957, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 42, 15, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (1, 2, 0, 1, 2, 1886, 1)
aal34[5].(blkinmsg[1], icblktofr[1]) = (0, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 303 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 9, 0, 16, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399956, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 2, 0, 45, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 1)
aal34[5].(blkinmsg[1], icblktofr[1]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 348 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399767, 0, 65535, 13, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)
```

```
aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (19, 0, 0, 0, 0, 3, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)


...


:::::::::: clock cycle number 390 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399869, 1, 5, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 391 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 9, 0, 8, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399868, 1, 5, 0, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 392 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 9, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399723, 0, 65535, 13, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 393 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 10, 0, 16, 0, 0, 1)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399722, 0, 65535, 13, 0)
```

```
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 394 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 2)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399865, 1, 5, 0, 1)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 395 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 16, 0, 16, 0, 0, 3)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (2, 2, 0, 1, 2, 1886, 1)
aal34[1].(blktime[1], blkcomtype[1], blkdst[1], acknum[1], ogblktofr[1]) = (399864, 1, 5, 0, 1)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

:::::::::: clock cycle number 396 ::::::::::

aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 2, 1, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399719, 0, 65535, 13, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)



...



:::::::::: clock cycle number 1056 ::::::::::
```

```
aal34[1].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 19)
aal34[1].(ogmsgnum, ogbblknum, ogfb, oglb, ogfi, ogli, cursnd) = (1, 1, 0, 0, 2, 1, 0)
aal34[1].(blktime[0], blkcomtype[0], blkdst[0], acknum[0], ogblktofr[0]) = (399059, 0, 65535, 1, 0)
aal34[1].ogsuspend = 0
aal34[1].(buf, ogbyte) = (bd, bd)
aal34[1].(bufi, ogbytei) = (0, 0)

aal34[5].icbyte = bd
aal34[5].icbytei = 0
aal34[5].icsuspend = 0
aal34[5].count = 0
aal34[5].state(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2)
aal34[5].(icblkinmsgnum, icbblknum, icfb, iclb, icfi, icli, iccurb) = (2, 2, 0, 1, 2, 1886, 0)
aal34[5].(blkinmsg[0], icblktofr[0]) = (1, 0)
aal34[5].msgnumfrm(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) = (0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0)


...
```

Appendix K

PVM Platform Operation Tests

This appendix presents the programs' C codes and the input/output files used for the platform operation tests (See Section 11.2).

K.1 Linear Equation System Distributed Resolution: Startup Program C Code

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "../atm/these.h"

void main(argc, argv)
int argc;
char **argv;
{
  FILE *ff;
  int i, n, nsq, inum, msgnum, shmid1, shmid2;
  float current_coeff_val;
  char lbuf[64];
  char ctmp;
  char *carcour, *strptr;

  struct prcsdescstr *prcstab;
  struct shmst *dev;

  if (!(ff = fopen(argv[1], "r")))
  {
    fprintf(stderr, "\ncan't open the file: %s\n", argv[1]);
    exit(1);
  }

  n = 0;

  while (n == 0)
  {
    if (!fgets(lbuf, sizeof(lbuf), ff)) break;
    carcour = lbuf;
    while (*carcour != '\n' && *carcour != EOF)
```

```
    {
      while (*carcour == ' ') carcour++;
      if (*carcour == '\n' || *carcour == '#' || *carcour == EOF) break;
      while (*carcour != ' ' && *carcour != '\n' && *carcour != EOF) carcour++;
      n++;
    }
    if (*carcour == EOF) break;
  }

  fclose(ff);

  if (n == 0)
  {
    fprintf(stderr, "\nthe file %s is empty\n", argv[1]);
    exit(0);
  }

  if (!(ff = fopen(argv[1], "r")))
  {
    fprintf(stderr, "\ncan't open the file: %s\n", argv[1]);
    exit(1);
  }

  --n;

  enroll("srl");

/* recuperation d'un idf sur le segment de memoire partage qui contient la table des processus */

  if ((shmid1 = shmget((key_t)0x11021000, sizeof(struct prcsdescstr), 0)) == -1)
  { perror("Echec de shmget survenu dans srl");
    exit(1);
  }

/* attachement au segment ci-dessus recupere, et recuperation du pointeur sur ce dernier */

  if (((int)prcstab = shmat(shmid1, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans srl");
    exit(1);
  }

/* enregistrement dans la table des processus */

  strcpy(prcstab->tab[prcstab->nb].name, "srl");
  prcstab->tab[prcstab->nb++].inum = 0;

/*
   recuperation d'un idf sur le segment de memoire partage qui interface les processus de l'application
   avec le processus reseau porte reseau d'E/S numero 0
*/

  if ((shmid2 = shmget((key_t)0x11021002, sizeof(struct shmst), 0)) == -1)
  { perror("Echec de shmget survenu dans srl");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)dev = shmat(shmid2, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans srl");
    exit(1);
  }

  while (fgets(lbuf, sizeof(lbuf), ff))
  {
    carcour = lbuf;
    while (*carcour != '\n' && *carcour != EOF)
    {
      while (*carcour == ' ') carcour++;
      if (*carcour == '\n' || *carcour == '#' || *carcour == EOF) break;
      strptr = carcour;
      while (*carcour != ' ' && *carcour != '\n' && *carcour != EOF) carcour++;
      ctmp = *carcour;
      *carcour = 0;
      current_coeff_val = (float)atoi(strptr);
```

```
      *carcour = ctmp;

/* enregitrement du processus a initier dans la table des processus */

      strcpy(prcstab->tab[prcstab->nb].name, "srp");
      prcstab->tab[prcstab->nb].inum = prcstab->nb++ - 1;

      inum = initiate("srp", NULL);

      (void)atminitsend(dev);
      (void)atmputnint(dev, &n, 1);
      (void)atmputnfloat(dev, &current_coeff_val, 1);
      (void)atmsnd(dev, prcstab, "srp", inum, inum);
    }
    if (*carcour == EOF) break;
  }

  fclose(ff);

  for (i=0; i<prcstab->nb; i++)
    printf("(<%s, %d>, %d)\n", prcstab->tab[i].name, prcstab->tab[i].inum, i%16);

  if (!(ff = fopen(argv[1], "a+")))
  {
    fprintf(stderr, "\ncan't append the file: %s\n", argv[1]);
    leave();
    exit(1);
  }

  nsq = n * (n + 1);

  fputs("\n", ff);

  for (i = 0; i < nsq; i++)
  {
    (void)atmrcv(dev, i);
    (void)atmgetnfloat(dev, &current_coeff_val, 1);
    dev->rlcked = 0;
    if ((i + 1) % (n + 1) == 0) fprintf(ff, "  %f\n", current_coeff_val);
    else fprintf(ff, "%f  ", current_coeff_val);
  }

  for (i = 1; i <= n; i++)
  {
    msgnum = i * (n + 1) - 1;
    (void)atmrcv(dev, msgnum);
    (void)atmgetnfloat(dev, &current_coeff_val, 1);
    dev->rlcked = 0;
    fprintf(ff, "\nx%d = %f", i, current_coeff_val);
  }

  fclose(ff);

  leave();
}
```

K.2 Linear Equation System Distributed Resolution: Resolution Program C Code

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "../atm/these.h"
```

```
void main()
{
  int n, nsq, my_inum, pvt_line, pvt_inum, xl, i, msgtype, shmid;
  float my_val, pvt_col, pvt_row, pvt_val, res;
  char lbuf[65];

  struct prcsdescstr *prcstab;
  struct shmst *dev;
  unsigned *cycle;

/* recuperation d'un idf du segment de memoire partage qui contient la table des processus */

  if ((shmid = shmget((key_t)0x11021000, sizeof(struct prcsdescstr), 0)) == -1)
  { perror("Echec de shmget survenu dans srp");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)prcstab = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans srp");
    exit(1);
  }

/* recuperation d'un idf du segment de memoire partage qui contient la variable cycle */

  if ((shmid = shmget((key_t)0x11021001, sizeof(unsigned), 0)) == -1)
  { perror("Echec de shmget survenu dans srp");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)cycle = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans srp");
    exit(1);
  }

  my_inum = enroll("srp");

  for (i=0; i<prcstab->nb; i++)
    if (!strcmp(prcstab->tab[i].name, "srp") && prcstab->tab[i].inum==my_inum)
      break;

/*
   recuperation de l'idf sur le segment de memoire partage qui interface les processus de l'application
   avec le processus reseau par la porte reseau d'E/S numero (i mod 16).
*/

  if ((shmid = shmget((key_t)0x11021002 + i%16, sizeof(struct shmst), 0)) == -1)
  { perror("Echec de shmget survenu dans srp");
    exit(1);
  }

/* attachement au segment ci_dessus recupere */

  if (((int)dev = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans srp");
    exit(1);
  }
  (void)atmrcv(dev, my_inum);
  (void)atmgetnint(dev, &n, 1);
  (void)atmgetnfloat(dev, &my_val, 1);
  dev->rlcked = 0;
  nsq = n * (n + 1);

  for (pvt_line = 1; pvt_line <= n - 1; pvt_line++)
  {
    pvt_inum = (pvt_line - 1) * (n + 2);
    if (my_inum == pvt_inum)
    {
      (void)atminitsend(dev);
      (void)atmputnfloat(dev, &my_val, 1);

      sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp: -1> \n", *cycle, my_inum, my_inum);
```

```
  write(1, lbuf, strlen(lbuf));

  (void)atmsnd(dev, prcstab, "srp", -1, my_inum);
  if (my_val == 0)
  {
    for (i = 1; i <= n - pvt_line; i++)
    {
      msgtype = pvt_inum + i * (n + 1);
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &my_val, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

      if (my_val != 0) break;
    }
    if (i > n - pvt_line) continue;
  }
  (void)atminitsend(dev);
  (void)atmputnfloat(dev, &my_val, 1);

  sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp: -1> \n", *cycle, my_inum, my_inum);
  write(1, lbuf, strlen(lbuf));

  (void)atmsnd(dev, prcstab, "srp", -1, my_inum);
}
if (pvt_inum < my_inum && my_inum <= pvt_line * (n + 1) - 1)
{
  (void)atmrcv(dev, pvt_inum);
  (void)atmgetnfloat(dev, &pvt_val, 1);
  dev->rlcked = 0;

  sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, pvt_inum, my_inum);
  write(1, lbuf, strlen(lbuf));

  if (pvt_val == 0)
  {
    for (i = 1; i <= n - pvt_line; i++)
    {
      msgtype = pvt_inum + i * (n + 1);
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &pvt_val, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

      if (pvt_val != 0) break;
    }
    if (i > n - pvt_line) continue;
    (void)atminitsend(dev);
    (void)atmputnfloat(dev, &my_val, 1);
    msgtype = my_inum + i * (n + 1);

    sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp:%3d> \n", *cycle, my_inum, my_inum, msgtype);
    write(1, lbuf, strlen(lbuf));

    (void)atmsnd(dev, prcstab, "srp", msgtype, my_inum);
    (void)atmrcv(dev, msgtype);
    (void)atmgetnfloat(dev, &my_val, 1);
    dev->rlcked = 0;

    sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
    write(1, lbuf, strlen(lbuf));

  }
  (void)atminitsend(dev);
  (void)atmputnfloat(dev, &my_val, 1);

  sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp: -1> \n", *cycle, my_inum, my_inum);
  write(1, lbuf, strlen(lbuf));

  (void)atmsnd(dev, prcstab, "srp", -1, my_inum);
}
```

```
for (xl = 1; xl <= n - pvt_line; xl++)
  if (pvt_inum + xl * (n + 1) <= my_inum && my_inum <= (pvt_line + xl) * (n + 1) - 1) break;
if (xl <= n - pvt_line)
{
  (void)atmrcv(dev, pvt_inum);
  (void)atmgetnfloat(dev, &pvt_val, 1);
  dev->rlcked = 0;

  sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, pvt_inum, my_inum);
  write(1, lbuf, strlen(lbuf));

  if (pvt_val == 0)
  {
    for (i = 1; i <= n - pvt_line; i++)
    {
      msgtype = pvt_inum + i * (n + 1);
      if (my_inum == msgtype)
      {
        (void)atminitsend(dev);
        (void)atmputnfloat(dev, &my_val, 1);

        sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp: -1> \n", *cycle, my_inum, my_inum);
        write(1, lbuf, strlen(lbuf));

        (void)atmsnd(dev, prcstab, "srp", -1, my_inum);
        pvt_val = my_val;
      }
      else
      {
        (void)atmrcv(dev, msgtype);
        (void)atmgetnfloat(dev, &pvt_val, 1);
        dev->rlcked = 0;

        sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
        write(1, lbuf, strlen(lbuf));

      }
      if (pvt_val != 0) break;
    }
    if (i > n - pvt_line) continue;
    if (my_inum == msgtype) my_val = 0;
    if (msgtype < my_inum && my_inum <= (pvt_line + i) * (n + 1) - 1)
    {
      (void)atminitsend(dev);
      (void)atmputnfloat(dev, &my_val, 1);
      msgtype = my_inum - i * (n + 1);

      sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp:%3d> \n", *cycle, my_inum, my_inum, msgtype);
      write(1, lbuf, strlen(lbuf));

      (void)atmsnd(dev, prcstab, "srp", msgtype, my_inum);
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &my_val, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

    }
  }
  msgtype = my_inum - xl * (n + 1);
  (void)atmrcv(dev, msgtype);
  (void)atmgetnfloat(dev, &pvt_row, 1);
  dev->rlcked = 0;

  sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
  write(1, lbuf, strlen(lbuf));

  if (pvt_inum + xl * (n + 1) == my_inum)
  {
    pvt_col = my_val;
    (void)atminitsend(dev);
    (void)atmputnfloat(dev, &my_val, 1);

    sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp: -1> \n", *cycle, my_inum, my_inum);
```

```
      write(1, lbuf, strlen(lbuf));

      (void)atmsnd(dev, prcstab, "srp", -1, my_inum);
    }
    else
    {
      msgtype = pvt_inum + xl * (n + 1);
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &pvt_col, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

    }
    my_val = my_val - pvt_col * pvt_row / pvt_val;
  }
}
(void)atminitsend(dev);
(void)atmputnfloat(dev, &my_val, 1);
(void)atmsnd(dev, prcstab, "srl", 0, my_inum);

for (pvt_line = n; pvt_line > 0; --pvt_line)
{
  pvt_inum = pvt_line * (n + 1) - 1;
  xl = n - pvt_line;
  if (pvt_inum - (xl + 1) <= my_inum && my_inum < pvt_inum)
  {
    (void)atminitsend(dev);
    (void)atmputnfloat(dev, &my_val, 1);
    msgtype = my_inum + nsq;

    sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp:%3d> \n", *cycle, msgtype, my_inum, pvt_inum);
    write(1, lbuf, strlen(lbuf));

    (void)atmsnd(dev, prcstab, "srp", pvt_inum, msgtype);
    continue;
  }
  for (i = 1; i <= xl; i++)
    if (my_inum == pvt_inum + i * (n + 1)) break;
  if (i <= xl)
  {
    (void)atminitsend(dev);
    (void)atmputnfloat(dev, &res, 1);
    msgtype = my_inum + nsq;

    sprintf(lbuf, "<%14u cycle><snd><%3d><      srp:%3d><      srp:%3d> \n", *cycle, msgtype, my_inum, pvt_inum);
    write(1, lbuf, strlen(lbuf));

    (void)atmsnd(dev, prcstab, "srp", pvt_inum, msgtype);
    continue;
  }
  if (my_inum == pvt_inum)
  {
    res = 0;
    for (i = 1; i <= xl; i++)
    {
      msgtype = pvt_inum - (xl + 1) + i + nsq;
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &pvt_row, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

      msgtype = pvt_inum + i * (n + 1) + nsq;
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &pvt_col, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

      res -= pvt_row * pvt_col;
    }
```

```
      res += my_val;
      msgtype = pvt_inum - (xl + 1) + nsq;
      (void)atmrcv(dev, msgtype);
      (void)atmgetnfloat(dev, &pvt_row, 1);
      dev->rlcked = 0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      srp:%3d> \n", *cycle, msgtype, my_inum);
      write(1, lbuf, strlen(lbuf));

      if (res == 0 && pvt_row == 0) res = 1;
      else
        if (res != 0 && pvt_row == 0)
        {
          fputs("\nsolution impossible\n", stderr);
          leave();
          exit(1);
        }
        else res /= pvt_row;
    }
  }
  for (i = 1; i <= n; i++)
    if (my_inum == i * (n + 1) - 1) break;
  if (i <= n)
  {
    (void)atminitsend(dev);
    (void)atmputnfloat(dev, &res, 1);
    (void)atmsnd(dev, prcstab, "srl", 0, my_inum);
  }

  leave();

}
```

K.3 Linear Equation System Distributed Resolution: I/O File Examples

```
:::::::::: example number 1 ::::::::::

1 9 2  1
1 20 15  1
3 1 7  1

1.000000  9.000000  2.000000    1.000000
0.000000  11.000000  13.000000    0.000000
0.000000  0.000000  31.727272    -2.000000

x1 = 0.455587
x2 = 0.074499
x3 = -0.063037


:::::::::: example number 2 ::::::::::

0 9 2  1
1 20 15  1
3 1 7  1
1.000000  20.000000  15.000000    1.000000
0.000000  9.000000  2.000000    1.000000
0.000000  0.000000  -24.888889    4.555555

x1 = 0.709821
x2 = 0.151786
x3 = -0.183036


:::::::::: example number 3 ::::::::::

0 9 2  1
```

```
0 20 15  1
6 1 7  1
6.000000  1.000000  7.000000    1.000000
0.000000  20.000000  15.000000    1.000000
0.000000  0.000000  -4.750000    0.550000

x1 = 0.278947
x2 = 0.136842
x3 = -0.115789


:::::::::: example number 4 ::::::::::

1 2 3 4 1
5 6 7 8 1
9 10 18 17 1
13 14 15 19 1

1.000000  2.000000  3.000000  4.000000    1.000000
0.000000  -4.000000  -8.000000  -12.000000    -4.000000
0.000000  0.000000  7.000000  5.000000    0.000000
0.000000  0.000000  0.000000  3.000000    0.000000

x1 = -1.000000
x2 = 1.000000
x3 = 0.000000
x4 = 0.000000
```

K.3.1 Traces File from the Experimental Resolution of the Example 4

```
<cime701><          4548  cycle><snd><  0><      srp:  0><      srp: -1>
<cime701><          4789  cycle><rcv><  0><         :   ><      srp: 18>
<cime701><          4789  cycle><rcv><  0><         :   ><      srp: 15>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp: 13>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp: 14>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp: 12>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp: 11>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp: 10>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  9>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  8>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  7>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  6>
<cime701><          4836  cycle><snd><  0><      srp:  0><      srp: -1>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  5>
<cime701><          4836  cycle><rcv><  0><         :   ><      srp:  4>
<cime701><          4924  cycle><snd><  4><      srp:  4><      srp: -1>
<cime701><          4924  cycle><rcv><  0><         :   ><      srp:  2>
<cime701><          4924  cycle><rcv><  0><         :   ><      srp:  1>
<cime701><          4965  cycle><snd><  2><      srp:  2><      srp: -1>
<cime701><          4997  cycle><rcv><  0><         :   ><      srp: 19>
<cime701><          5056  cycle><rcv><  0><         :   ><      srp: 17>
<cime701><          5088  cycle><rcv><  0><         :   ><      srp: 16>
<cime701><          5088  cycle><snd><  1><      srp:  1><      srp: -1>
<cime701><          5120  cycle><rcv><  0><         :   ><      srp: 10>
<cime701><          5120  cycle><rcv><  0><         :   ><      srp:  3>
<cime701><          5151  cycle><snd>< 10><      srp: 10><      srp: -1>
<cime701><          5174  cycle><rcv><  0><         :   ><      srp:  5>
<cime701><          5216  cycle><snd><  5><      srp:  5><      srp: -1>
<cime701><          5238  cycle><rcv><  0><         :   ><      srp: 15>
<cime701><          5272  cycle><snd>< 20><      srp:  0><      srp:  4>
<cime701><          5288  cycle><snd>< 15><      srp: 15><      srp: -1>
<cime701><          5320  cycle><rcv><  4><         :   ><      srp: 19>
<cime701><          5342  cycle><rcv><  4><         :   ><      srp: 14>
<cime701><          5342  cycle><rcv><  2><         :   ><      srp: 12>
<cime701><          5342  cycle><rcv><  2><         :   ><      srp:  7>
<cime701><          5342  cycle><rcv><  4><         :   ><      srp:  9>
<cime701><          5360  cycle><rcv><  2><         :   ><      srp: 17>
<cime701><          5360  cycle><snd><  3><      srp:  3><      srp: -1>
<cime701><          5383  cycle><snd>< 22><      srp:  2><      srp:  4>
<cime701><          5497  cycle><rcv><  1><         :   ><      srp:  6>
```

```
<cime701><          5497 cycle><rcv>< 10><         :   ><      srp: 14>
<cime701><          5497 cycle><rcv>< 10><         :   ><      srp: 12>
<cime701><          5497 cycle><rcv><  1><         :   ><      srp: 11>
<cime701><          5553 cycle><snd>< 21><      srp:  1><      srp:  4>
<cime701><          5664 cycle><rcv><  5><         :   ><      srp:  6>
<cime701><          5669 cycle><rcv>< 10><         :   ><      srp: 11>
<cime701><          5684 cycle><rcv><  5><         :   ><      srp:  9>
<cime701><          5689 cycle><rcv><  5><         :   ><      srp:  7>
<cime701><          5689 cycle><snd><  6><      srp:  6><      srp: -1>
<cime701><          5710 cycle><rcv>< 15><         :   ><      srp: 19>
<cime701><          5742 cycle><rcv><  3><         :   ><      srp: 13>
<cime701><          5742 cycle><snd>< 23><      srp:  3><      srp:  4>
<cime701><          5762 cycle><rcv><  3><         :   ><      srp:  8>
<cime701><          5865 cycle><snd><  6><      srp:  6><      srp: -1>
<cime701><          5926 cycle><rcv><  5><         :   ><      srp:  8>
<cime701><          5962 cycle><rcv><  3><         :   ><      srp: 18>
<cime701><          5962 cycle><rcv>< 10><         :   ><      srp: 13>
<cime701><          6033 cycle><rcv><  6><         :   ><      srp: 14>
<cime701><          6033 cycle><rcv><  6><         :   ><      srp: 12>
<cime701><          6033 cycle><rcv><  6><         :   ><      srp: 11>
<cime701><          6107 cycle><rcv><  6><         :   ><      srp:  8>
<cime701><          6113 cycle><rcv><  6><         :   ><      srp: 13>
<cime701><          6148 cycle><snd><  8><      srp:  8><      srp: -1>
<cime701><          6148 cycle><snd>< 26><      srp:  6><      srp:  9>
<cime701><          6148 cycle><rcv>< 21><         :   ><      srp:  4>
<cime701><          6195 cycle><rcv><  6><         :   ><      srp: 19>
<cime701><          6195 cycle><rcv><  6><         :   ><      srp:  9>
<cime701><          6216 cycle><rcv><  6><         :   ><      srp: 11>
<cime701><          6216 cycle><rcv><  6><         :   ><      srp:  7>
<cime701><          6259 cycle><snd>< 11><      srp: 11><      srp: -1>
<cime701><          6259 cycle><snd><  9><      srp:  9><      srp: -1>
<cime701><          6259 cycle><snd><  7><      srp:  7><      srp: -1>
<cime701><          6423 cycle><snd>< 28><      srp:  8><      srp:  9>
<cime701><          6474 cycle><rcv><  8><         :   ><      srp: 13>
<cime701><          6520 cycle><rcv>< 15><         :   ><      srp: 18>
<cime701><          6520 cycle><snd>< 27><      srp:  7><      srp:  9>
<cime701><          6616 cycle><rcv><  7><         :   ><      srp: 12>
<cime701><          6634 cycle><rcv><  9><         :   ><      srp: 14>
<cime701><          6685 cycle><rcv>< 11><         :   ><      srp: 13>
<cime701><          6724 cycle><rcv><  6><         :   ><      srp: 18>
<cime701><          6729 cycle><rcv>< 15><         :   ><      srp: 17>
<cime701><          6799 cycle><rcv><  9><         :   ><      srp: 19>
<cime701><          6818 cycle><rcv>< 11><         :   ><      srp: 12>
<cime701><          6846 cycle><rcv>< 11><         :   ><      srp: 14>
<cime701><          6865 cycle><snd>< 12><      srp: 12><      srp: -1>
<cime701><          6899 cycle><rcv><  6><         :   ><      srp: 17>
<cime701><          6921 cycle><rcv><  8><         :   ><      srp: 18>
<cime701><          6978 cycle><rcv><  1><         :   ><      srp: 16>
<cime701><          7010 cycle><snd>< 12><      srp: 12><      srp: -1>
<cime701><          7061 cycle><rcv><  7><         :   ><      srp: 17>
<cime701><          7243 cycle><rcv>< 27><         :   ><      srp:  9>
<cime701><          7276 cycle><rcv>< 12><         :   ><      srp: 14>
<cime701><          7294 cycle><rcv>< 12><         :   ><      srp: 13>
<cime701><          7294 cycle><snd>< 32><      srp: 12><      srp: 14>
<cime701><          7348 cycle><snd>< 14><      srp: 14><      srp: -1>
<cime701><          7348 cycle><snd>< 13><      srp: 13><      srp: -1>
<cime701><          7564 cycle><rcv>< 15><         :   ><      srp: 16>
<cime701><          7608 cycle><snd>< 33><      srp: 13><      srp: 14>
<cime701><          7727 cycle><rcv><  6><         :   ><      srp: 16>
<cime701><          7911 cycle><rcv><  6><         :   ><      srp: 16>
<cime701><          7971 cycle><snd>< 16><      srp: 16><      srp: -1>
<cime701><          8337 cycle><rcv>< 33><         :   ><      srp: 14>
<cime701><          8375 cycle><rcv>< 16><         :   ><      srp: 18>
<cime701><          8515 cycle><rcv>< 16><         :   ><      srp: 19>
<cime701><          8515 cycle><rcv>< 16><         :   ><      srp: 17>
<cime701><          8567 cycle><rcv>< 12><         :   ><      srp: 18>
<cime701><          8723 cycle><rcv>< 12><         :   ><      srp: 19>
<cime701><          8748 cycle><rcv>< 12><         :   ><      srp: 17>
<cime701><          8811 cycle><rcv>< 13><         :   ><      srp: 18>
<cime701><          8969 cycle><rcv>< 14><         :   ><      srp: 19>
<cime701><          8985 cycle><rcv>< 12><         :   ><      srp: 17>
<cime701><          9039 cycle><snd>< 17><      srp: 17><      srp: -1>
<cime701><          9436 cycle><rcv>< 17><         :   ><      srp: 19>
<cime701><          9493 cycle><rcv>< 17><         :   ><      srp: 18>
<cime701><          9695 cycle><snd>< 38><      srp: 18><      srp: 19>
```

```
<cime701><        10371 cycle><rcv>< 38><        :   ><      srp: 19>
<cime701><        10434 cycle><snd>< 39><     srp: 19><      srp: 14>
<cime701><        10613 cycle><snd>< 39><     srp: 19><      srp:  9>
<cime701><        10754 cycle><snd>< 39><     srp: 19><      srp:  4>
<cime701><        10800 cycle><rcv>< 39><        :   ><      srp: 14>
<cime701><        11031 cycle><rcv>< 32><        :   ><      srp: 14>
<cime701><        11079 cycle><snd>< 34><     srp: 14><      srp:  9>
<cime701><        11275 cycle><snd>< 34><     srp: 14><      srp:  4>
<cime701><        11793 cycle><rcv>< 34><        :   ><      srp:  9>
<cime701><        12087 cycle><rcv>< 28><        :   ><      srp:  9>
<cime701><        12301 cycle><rcv>< 39><        :   ><      srp:  9>
<cime701><        12514 cycle><rcv>< 26><        :   ><      srp:  9>
<cime701><        12602 cycle><snd>< 29><     srp:  9><      srp:  4>
<cime701><        13045 cycle><rcv>< 29><        :   ><      srp:  4>
<cime701><        13273 cycle><rcv>< 22><        :   ><      srp:  4>
<cime701><        13586 cycle><rcv>< 34><        :   ><      srp:  4>
<cime701><        13767 cycle><rcv>< 23><        :   ><      srp:  4>
<cime701><        14003 cycle><rcv>< 39><        :   ><      srp:  4>
<cime701><        14228 cycle><rcv>< 20><        :   ><      srp:  4>
```

K.3.2 Traces File from the Theoretical Resolution of the Example 4

```
<cime701><         4548 cycle><snd><  0><     srp:  0><      srp: -1>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  1>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  4>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  5>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  6>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  7>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  8>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp:  9>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 10>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 11>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 12>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 13>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 14>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 15>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 16>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 18>
<cime701><         4741 cycle><rcv><  0><        :   ><      srp: 19>
<cime701><         4829 cycle><snd><  4><     srp:  4><      srp: -1>
<cime701><         4836 cycle><snd><  0><     srp:  0><      srp: -1>
<cime701><         4864 cycle><rcv><  0><        :   ><      srp:  3>
<cime701><         4873 cycle><rcv><  0><        :   ><      srp: 17>
<cime701><         4876 cycle><rcv><  0><        :   ><      srp:  2>
<cime701><         4905 cycle><snd><  1><     srp:  1><      srp: -1>
<cime701><         4917 cycle><snd><  2><     srp:  2><      srp: -1>
<cime701><         5022 cycle><rcv><  4><        :   ><      srp:  9>
<cime701><         5022 cycle><rcv><  0><        :   ><      srp: 14>
<cime701><         5029 cycle><rcv><  0><        :   ><      srp:  5>
<cime701><         5029 cycle><rcv><  0><        :   ><      srp: 10>
<cime701><         5029 cycle><rcv><  0><        :   ><      srp: 15>
<cime701><         5060 cycle><snd>< 10><     srp: 10><      srp: -1>
<cime701><         5064 cycle><rcv><  4><        :   ><      srp: 19>
<cime701><         5071 cycle><snd><  5><     srp:  5><      srp: -1>
<cime701><         5079 cycle><snd>< 15><     srp: 15><      srp: -1>
<cime701><         5098 cycle><rcv><  1><        :   ><      srp:  6>
<cime701><         5098 cycle><rcv><  1><        :   ><      srp: 11>
<cime701><         5104 cycle><snd><  3><     srp:  3><      srp: -1>
<cime701><         5110 cycle><rcv><  2><        :   ><      srp:  7>
<cime701><         5110 cycle><rcv><  2><        :   ><      srp: 12>
<cime701><         5177 cycle><rcv><  2><        :   ><      srp: 17>
<cime701><         5253 cycle><rcv>< 10><        :   ><      srp: 11>
<cime701><         5253 cycle><rcv>< 10><        :   ><      srp: 12>
<cime701><         5253 cycle><rcv>< 10><        :   ><      srp: 14>
<cime701><         5264 cycle><rcv><  5><        :   ><      srp:  6>
<cime701><         5264 cycle><rcv><  5><        :   ><      srp:  7>
<cime701><         5264 cycle><rcv><  5><        :   ><      srp:  9>
<cime701><         5272 cycle><snd>< 20><     srp:  0><      srp:  4>
<cime701><         5289 cycle><snd><  6><     srp:  6><      srp: -1>
<cime701><         5297 cycle><rcv><  3><        :   ><      srp:  8>
```

```
<cime701><         5297 cycle><rcv><  3><         :   ><      srp: 13>
<cime701><         5335 cycle><snd>< 22><      srp:  2><      srp:  4>
<cime701><         5337 cycle><rcv><  5><         :   ><      srp:  8>
<cime701><         5337 cycle><rcv>< 10><         :   ><      srp: 13>
<cime701><         5370 cycle><snd>< 21><      srp:  1><      srp:  4>
<cime701><         5454 cycle><rcv>< 15><         :   ><      srp: 19>
<cime701><         5465 cycle><snd><  6><      srp:  6><      srp: -1>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp:  7>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp:  8>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp:  9>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp: 11>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp: 12>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp: 13>
<cime701><         5482 cycle><rcv><  6><         :   ><      srp: 14>
<cime701><         5486 cycle><snd>< 23><      srp:  3><      srp:  4>
<cime701><         5523 cycle><snd><  8><      srp:  8><      srp: -1>
<cime701><         5525 cycle><snd><  7><      srp:  7><      srp: -1>
<cime701><         5546 cycle><snd><  9><      srp:  9><      srp: -1>
<cime701><         5546 cycle><rcv>< 15><         :   ><      srp: 17>
<cime701><         5563 cycle><rcv>< 21><         :   ><      srp:  4>
<cime701><         5586 cycle><rcv><  6><         :   ><      srp: 17>
<cime701><         5658 cycle><rcv><  6><         :   ><      srp: 11>
<cime701><         5701 cycle><snd>< 11><      srp: 11><      srp: -1>
<cime701><         5716 cycle><rcv><  8><         :   ><      srp: 13>
<cime701><         5718 cycle><rcv><  7><         :   ><      srp: 12>
<cime701><         5718 cycle><rcv><  7><         :   ><      srp: 17>
<cime701><         5739 cycle><rcv><  9><         :   ><      srp: 14>
<cime701><         5748 cycle><snd>< 26><      srp:  6><      srp:  9>
<cime701><         5786 cycle><snd>< 27><      srp:  7><      srp:  9>
<cime701><         5798 cycle><snd>< 28><      srp:  8><      srp:  9>
<cime701><         5894 cycle><rcv>< 11><         :   ><      srp: 12>
<cime701><         5894 cycle><rcv>< 11><         :   ><      srp: 13>
<cime701><         5894 cycle><rcv>< 11><         :   ><      srp: 14>
<cime701><         5914 cycle><rcv><  3><         :   ><      srp: 18>
<cime701><         5939 cycle><rcv><  6><         :   ><      srp: 19>
<cime701><         5941 cycle><snd>< 12><      srp: 12><      srp: -1>
<cime701><         5954 cycle><rcv>< 15><         :   ><      srp: 18>
<cime701><         5979 cycle><rcv><  9><         :   ><      srp: 19>
<cime701><         5980 cycle><rcv>< 27><         :   ><      srp:  9>
<cime701><         5994 cycle><rcv><  6><         :   ><      srp: 18>
<cime701><         6034 cycle><rcv><  8><         :   ><      srp: 18>
<cime701><         6086 cycle><snd>< 12><      srp: 12><      srp: -1>
<cime701><         6134 cycle><rcv>< 12><         :   ><      srp: 13>
<cime701><         6134 cycle><rcv>< 12><         :   ><      srp: 14>
<cime701><         6157 cycle><snd>< 14><      srp: 14><      srp: -1>
<cime701><         6188 cycle><snd>< 13><      srp: 13><      srp: -1>
<cime701><         6370 cycle><snd>< 32><      srp: 12><      srp: 14>
<cime701><         6448 cycle><snd>< 33><      srp: 13><      srp: 14>
<cime701><         6641 cycle><rcv>< 33><         :   ><      srp: 14>
<cime701><         6978 cycle><rcv><  1><         :   ><      srp: 16>
<cime701><         7018 cycle><rcv>< 15><         :   ><      srp: 16>
<cime701><         7058 cycle><rcv><  6><         :   ><      srp: 16>
<cime701><         7098 cycle><rcv><  6><         :   ><      srp: 16>
<cime701><         7158 cycle><snd>< 16><      srp: 16><      srp: -1>
<cime701><         7351 cycle><rcv>< 16><         :   ><      srp: 17>
<cime701><         7351 cycle><rcv>< 16><         :   ><      srp: 18>
<cime701><         7351 cycle><rcv>< 16><         :   ><      srp: 19>
<cime701><         7391 cycle><rcv>< 12><         :   ><      srp: 17>
<cime701><         7391 cycle><rcv>< 12><         :   ><      srp: 18>
<cime701><         7391 cycle><rcv>< 12><         :   ><      srp: 19>
<cime701><         7431 cycle><rcv>< 12><         :   ><      srp: 17>
<cime701><         7431 cycle><rcv>< 13><         :   ><      srp: 18>
<cime701><         7431 cycle><rcv>< 14><         :   ><      srp: 19>
<cime701><         7485 cycle><snd>< 17><      srp: 17><      srp: -1>
<cime701><         7678 cycle><rcv>< 17><         :   ><      srp: 18>
<cime701><         7678 cycle><rcv>< 17><         :   ><      srp: 19>
<cime701><         7880 cycle><snd>< 38><      srp: 18><      srp: 19>
<cime701><         8073 cycle><rcv>< 38><         :   ><      srp: 19>
<cime701><         8136 cycle><snd>< 39><      srp: 19><      srp: 14>
<cime701><         8315 cycle><snd>< 39><      srp: 19><      srp:  9>
<cime701><         8329 cycle><rcv>< 39><         :   ><      srp: 14>
<cime701><         8369 cycle><rcv>< 32><         :   ><      srp: 14>
<cime701><         8417 cycle><rcv>< 29><         :   ><      srp: 14>
<cime701><         8455 cycle><snd>< 39><      srp: 19><      srp:  4>
<cime701><         8610 cycle><rcv>< 34><         :   ><      srp:  9>
```

```
<cime701><         8613 cycle><snd>< 34><      srp: 14><      srp:  4>
<cime701><         8650 cycle><rcv>< 28><         :   ><      srp:  9>
<cime701><         8690 cycle><rcv>< 39><         :   ><      srp:  9>
<cime701><         8730 cycle><rcv>< 26><         :   ><      srp:  9>
<cime701><         8818 cycle><snd>< 29><      srp:  9><      srp:  4>
<cime701><         9011 cycle><rcv>< 29><         :   ><      srp:  4>
<cime701><         9051 cycle><rcv>< 22><         :   ><      srp:  4>
<cime701><         9091 cycle><rcv>< 34><         :   ><      srp:  4>
<cime701><         9131 cycle><rcv>< 23><         :   ><      srp:  4>
<cime701><         9171 cycle><rcv>< 39><         :   ><      srp:  4>
<cime701><         9211 cycle><rcv>< 20><         :   ><      srp:  4>
```

K.4 Matrix Distributed Multiplication: Startup Program C Code

```
#include <stdio.h>
#include <malloc.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "../atm/these.h"

void main(argc, argv)
int argc;
char **argv;
{
  FILE *ff;
  int m, p, n, inum, msgtype, shmid, val, i;
  char lbuf[64];
  char ctmp;
  char *carcour, *strptr;

  char *res;
  struct prcsdescstr *prcstab;
  struct shmst *dev;

  if (!(ff=fopen(argv[1], "r")))
  {
    fprintf(stderr, "\ncan't open the file: %s\n", argv[1]);
    exit(1);
  }

  m=p=n=0;

  while (1)
  {
    if (!fgets(lbuf, sizeof(lbuf), ff)) exit(1);
    carcour=lbuf;
    while (1)
    {
      while (*carcour==' ') carcour++;
      if (*carcour=='\n') break;
      if (*carcour=='*')
        if (!m && !p) exit(0);
        else break;
      while(1)
      {
        while (*carcour!=' ' && *carcour!='\n' && *carcour!=EOF) carcour++;
        if (!m) p++;
        while (*carcour==' ') carcour++;
        if (*carcour=='\n')
        {
          m++;
          break;
        }
      }
```

```
    }
    if (*carcour=='*') break;
  }
  while(1)
  {
    if (!fgets(lbuf, sizeof(lbuf), ff)) exit(1);
    carcour=lbuf;
    while (*carcour==' ') carcour++;
    if (*carcour=='\n') continue;
    if (*carcour==EOF) exit(0);
    while(1)
    {
      while (*carcour!=' ' && *carcour!='\n' && *carcour!=EOF) carcour++;
      n++;
      while (*carcour==' ') carcour++;
      if (*carcour=='\n' || *carcour==EOF) break;
    }
    break;
  }

  fclose(ff);

/* recuperation d'un idf sur le segment de memoire partage qui contient la table des processus */

  if ((shmid = shmget((key_t)0x11021000, sizeof(struct prcsdescstr), 0)) == -1)
  { perror("Echec de shmget survenu dans mml");
    exit(1);
  }

/* attachement au segment ci-dessus recupere, et recuperation du pointeur sur ce dernier */

  if (((int)prcstab = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans mml");
    exit(1);
  }

/* enregistrement dans la table des processus */

  strcpy(prcstab->tab[prcstab->nb].name, "mml");
  prcstab->tab[prcstab->nb++].inum=0;

/*
   recuperation d'un idf sur le segment de memoire partage qui interface les processus de l'application
   avec le processus reseau porte reseau d'E/S numero 0
*/

  if ((shmid = shmget((key_t)0x11021002, sizeof(struct shmst), 0)) == -1)
  { perror("Echec de shmget survenu dans mml");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)dev = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans mml");
    exit(1);
  }

  if (!(ff = fopen(argv[1], "r")))
  {
    fprintf(stderr, "\ncan't open the file: %s\n", argv[1]);
    exit(1);
  }

  enroll("mml");

  while (fgets(lbuf, sizeof(lbuf), ff))
  {
    carcour=lbuf;
    while (*carcour!='\n' && *carcour!='*' && *carcour!=EOF)
    {
      while (*carcour==' ') carcour++;
      if (*carcour=='\n' || *carcour=='*' || *carcour==EOF) break;
      strptr=carcour;
      while (*carcour!=' ' && *carcour!='\n' && *carcour!=EOF) carcour++;
```

```
      ctmp=*carcour;
      *carcour=0;
      val=atoi(strptr);
      *carcour=ctmp;

/* enregitrement du processus initie dans la table des processus */

      strcpy(prcstab->tab[prcstab->nb].name, "mmp");
      prcstab->tab[prcstab->nb].inum = prcstab->nb++ - 1;

      inum=initiate("mmp", NULL);

      (void)atminitsend(dev);
      (void)atmputnint(dev, &m, 1);
      (void)atmputnint(dev, &p, 1);
      (void)atmputnint(dev, &n, 1);
      (void)atmputnint(dev, &val, 1);
      (void)atmsnd(dev, prcstab, "mmp", inum, inum);
    }
    if (*carcour==EOF) break;
  }

  fclose(ff);

  printf("(m, p, n) = (%d, %d, %d)\n", m, p, n);

  for (inum=0; inum<prcstab->nb; inum++)
    printf("(<%s, %d>, %d)\n", prcstab->tab[inum].name, prcstab->tab[inum].inum, inum%16);

  res=(char*)malloc(m*n*sizeof(int));

  for (shmid=0; shmid<n; shmid++)
    for (inum=0; inum<m; inum++)
    {
      atmrcv(dev, inum*p);
      atmgetnint(dev, (int*)res+(inum*n+shmid)*sizeof(int), 1);
      dev->rlcked=0;
    }

  if (!(ff=fopen(argv[1], "a+")))
  {
    fprintf(stderr, "\ncan't append the file: %s\n", argv[1]);
    leave();
    exit(1);
  }

  fputs("=", ff);

  for (inum=0; inum<m*n; inum++)
    if (!(inum%n)) fprintf(ff, "\n%d ", ((int*)res+inum*sizeof(int))[0]);
    else fprintf(ff, "%d ", ((int*)res+inum*sizeof(int))[0]);

  fclose(ff);

  leave();
}
```

K.5 Matrix Distributed Multiplication: Multiplication Program C Code

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "../atm/these.h"
```

```
void main()
{
  int m, p, n, myinum, dstinum, msgtype, shmid;
  int myval, val, res;
  char lbuf[65];

  struct prcsdescstr *prcstab;
  struct shmst *dev;
  unsigned *cycle;

/* recuperation d'un idf du segment de memoire partage qui contient la table des processus */

  if ((shmid = shmget((key_t)0x11021000, sizeof(struct prcsdescstr), 0)) == -1)
  { perror("Echec de shmget survenu dans mmp");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)prcstab = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans mmp");
    exit(1);
  }

/* recuperation d'un idf du segment de memoire partage qui contient la variable cycle */

  if ((shmid = shmget((key_t)0x11021001, sizeof(unsigned), 0)) == -1)
  { perror("Echec de shmget survenu dans mmp");
    exit(1);
  }

/* attachement au segment ci-dessus recupere */

  if (((int)cycle = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans mmp");
    exit(1);
  }

  myinum = enroll("mmp");

  for (dstinum=0; dstinum<prcstab->nb; dstinum++)
    if (!strcmp(prcstab->tab[dstinum].name, "mmp") && prcstab->tab[dstinum].inum==myinum)
      break;

/*
   recuperation de l'idf sur le segment de memoire partage qui interface les processus de l'application
   avec le processus reseau par la porte reseau d'E/S numero (dstinum mod 16).
*/

  if ((shmid = shmget((key_t)0x11021002 + dstinum%16, sizeof(struct shmst), 0)) == -1)
  { perror("Echec de shmget survenu dans mmp");
    exit(1);
  }

/* attachement au segment ci_dessus recupere */

  if (((int)dev = shmat(shmid, 0, 0)) == -1)
  { perror("Echec de shmat survenu dans mmp");
    exit(1);
  }

  (void)atmrcv(dev, myinum);
  (void)atmgetnint(dev, &m, 1);
  (void)atmgetnint(dev, &p, 1);
  (void)atmgetnint(dev, &n, 1);
  (void)atmgetnfloat(dev, &myval, 1);
  dev->rlcked = 0;

  if (myinum<m*p)
    for (shmid=0; shmid<n; shmid++)
    {
      msgtype=n*(myinum%p) + m*p + shmid;
      atmrcv(dev, msgtype);
      atmgetnint(dev, &val, 1);
```

```
      dev->rlcked=0;

      sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      mmp:%3d> \n", *cycle, msgtype, myinum);
      write(1, lbuf, strlen(lbuf));

      res=myval*val;
      if (!(myinum%p))
      {
        for (dstinum=1; dstinum<p; dstinum++)
        {
          msgtype=myinum+dstinum;
          atmrcv(dev, msgtype);
          atmgetnint(dev, &val, 1);
          dev->rlcked=0;

          sprintf(lbuf, "<%14u cycle><rcv><%3d><         :   ><      mmp:%3d> \n", *cycle, msgtype, myinum);
          write(1, lbuf, strlen(lbuf));

          res+=val;
        }
        atminitsend(dev);
        atmputnint(dev, &res, 1);
        atmsnd(dev, prcstab, "mml", 0, myinum);
      }
      else
      {
        dstinum=myinum-(myinum%p);
        atminitsend(dev);
        atmputnint(dev, &res, 1);

        sprintf(lbuf, "<%14u cycle><snd><%3d><      mmp:%3d><      mmp:%3d> \n", *cycle, myinum, myinum, dstinum);
        write(1, lbuf, strlen(lbuf));

        atmsnd(dev, prcstab, "mmp", dstinum, myinum);
      }
    }
  else
  {
    atminitsend(dev);
    atmputnint(dev, &myval, 1);

    sprintf(lbuf, "<%14u cycle><snd><%3d><      mmp:%3d><      mmp: -1> \n", *cycle, myinum, myinum);
    write(1, lbuf, strlen(lbuf));

    atmsnd(dev, prcstab, "mmp", -1, myinum);
  }

  leave();

}
```

K.6 Matrix Distributed Multiplication I/O File Examples

```
:::::::::: example number 1 ::::::::::

2
*
7
=
14


:::::::::: example number 2 ::::::::::

1 2
3 5
*
```

```
1 0
0 1
=
1 2
3 5
```

```
:::::::::: example number 3 ::::::::::

6 4 2 0
1 3 5 7
7 7 7 7
*
1 1
1 1
1 1
1 1
=
12 12
16 16
28 28
```

Part IV

Bibliography

Bibliography

[1] Ellis Horwood
Asynchronous Transfer Mode - Solution for B-ISDN, 2nd Edition, 1993.

[2] Zahir Ebrahim
http://www.noc.lanl.gov/land/project/atm.tutorial.html,
A Brief Tutorial on ATM, March 5, 1992.

[3] Contribution from Japan (D368) CCITT SG XVIII
Adressing and Routing for ATM Connectionless Service, 1989.

[4] Contribution from Japan (D369) CCITT SG XVIII
Cell Transport Method for ATM Connectionless Service, 1989.

[5] J. Crowcroft, I. Wakeman, Z. Wang, D. Sirovica
Is Layering Harmful? IEEE Network, 6(1):20–24, Jan. 1992.

[6] C. T. Lea
What Should Be the Goal for ATM, IEEE Network, Sept. 1992.

[7] Jean Yves Le Boudec
http://lcrwww.epfl.ch/,
ATM Tutorial, May 30, 1991.

[8] P. Joos, W. Verbiest
A Statistical Bandwidth Allocation and Usage Monitoring Algorithm for ATM Networks, ICC'89, Boston, June 1989.

[9] T.-Y. Feng
A Survey of Interconnection Networks, Computer, Vol. 14, pp. 12–27, Dec. 1981.

[10] P. Kermani, L. Kleinrock
Virtual Cut-Through: A new Computer Communication Switching Technique, Computer Networks, Vol. 3, No. 4, Sept. 1979.

[11] D. M. Dias, M. Kumar
Packet Switching in N lg N Multistage Networks, GLOBECOM'84 Conf. Rec., pp 114–120, Nov. 1984.

[12] H. Armbruster
The Flexibility of ATM, Proceedings of the ATM conference, Paris, April 1993.

[13] J. P. Coudreuse, Al.
Special ATM, l'Echo des Recherches No. 144/145, 1991.

[14] Draft Recommendation I.150
B-ISDN ATM Functional Characteristics, CCITT SG XVIII, Report R34, June 1990.

[15] Draft Recommendation I.361
B-ISDN ATM Layer Specification, CCITT SG XVIII, Report R34, June 1990.

[16] Draft Recommendation I.362
B-ISDN AAL Functional Description, CCITT SG XVIII, Report R34, June 1990.

[17] Draft Recommendation I.363
B-ISDN AAL Specification, CCITT SG XVIII, Report R34, June 1990.

[18] Y.-C. Jenq
Performance Analysis of a Packet Switch based on a single-buffered Banyan Network, IEEE J. Select. Areas Commun., vol. SAC-1, pp 1014–1021, Dec. 1983.

[19] M. J. Karol, M. G. Hluchyj, S. P. Morgan
Input Versus Output Queueing on a Space-Division Packet Switch, IEEE Trans. Commun., Vol. COM-35, No 12, pp 1347–1356, Dec. 1987.

[20] C. A. Sunshine Ed.
Computer Network Architectures and Protocols, Plenum Press, 1989.

[21] S. Nojima, E. Tsutsui, H. Fukuda, M. Hascimoto
Integrated Services Packet Network using Bus Matrix Switch, IEEE J. Select. Areas Commun., vol. 10, pp 1284–1292, Oct. 1987.

[22] P. Boyer
A Congestion Control for the ATM, in seventh ITC Seminar, 1990.

[23] A. Gersht, K. J. Lee
A Congestion Control Framework for ATM Networks, Infocom, pp 701–710, 1989.

[24] Request For Comment 1483
Multiprotocol Encapsulation over ATM Adaptation Layer 5, July 1993.

[25] F. Denissen, Al.
The Policing Function in an ATM Network, Int'l Zürich Seminar on Digital Communications, 1990.

[26] E. P. Rathgeb
Comparison of Policing Mechanisms for ATM Networks, Race Workshop, CNET Paris, Oct. 1989.

[27] Andrew S. Tanenbaum
Computer Networks, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1988.

[28] S. G. Akl.
The Design and Analysis of Parallel Algorithms, Prentice-Hall, 1989.

[29] A. S. Tanenbaum
Réseaux-Architectures, Protocoles, Applications, InterEditions, 1990.

[30] SunOS 5.5
Reference Manual Pages of the signal() C Library Function.

[31] M. Decina, T. Toniatti
On Bandwidth Allocation to Bursty Virtual Connections in ATM Networks, ICC'90, 1990.

[32] SunOS 5.5
Reference Manual Pages of the pipe() System Call.

[33] H. Ahmadi, R. Guerin
Bandwidth Allocation in High Speed Networks, ITC Spec. Seminar, Adelaïde, 1989.

[34] SunOS 5.5
Reference Manual Pages of the msgget() System Call.

[35] G. Gallassi, G. Rigolio, L. Fratta
ATM: Bandwidth Assignment and Bandwidth Enforcement Policies, Globecom 89, 1989.

[36] SunOS 5.5
Reference Manual Pages of the shmget() System Call.

[37] Z. Dziong, Al.
Admission Control and Routing in ATM Networks, ITC Spec Seminar, Adelaïde, 1989.

[38] SunOS 5.5
Reference Manual Pages of the semget() System Call.

[39] J. A. S. Monteiro, M. Gerla, L. Fratta
Statistical Multiplexing in ATM Networks, to appear in Performance Evaluation.

[40] R. Händel, M. N. Huber
Integrated Broadband Networks, An Introduction to ATM-based Networks, to be published.

[41] F. K. Hwang
Control Algorithms for Rearrangeable Clos Networks, IEEE Transaction on Communications, Vol. 31, No. 8, August 1983.

[42] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam
A User's Guide to PVM Parallel Virtual Machine (Version 2.3), University of Tennessee Technical Report CS-91-136, July 1991.

[43] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam
PVM3 User's Guide and Reference Manual, Oak Ridge National Laboratory, May 1993.

[44] Message Passing Interface Forum (MPIF)
MPI: A Message-Passing Interface Standard, University of Tennessee Technical Report CS-94-230, April 1994.

[45] Edinburgh Parallel Computing Center, University of Edinburgh
CHIMP Concepts, June 1991.

[46] Edinburgh Parallel Computing Center, University of Edinburgh
CHIMP Version 1.0 Interface, May 1992.

[47] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam
Visualization and debugging in a heterogeneous environment, IEEE Computer, 26(6):88-95, June 1993.

[48] J. Dongarra, A. Geist, R. Manchek, V. Sunderam
Integrated PVM framework supports heterogeous network computing, Computers in Physics, 7(2):166-75, April 1993.

[49] William D. Gropp, B. Smith
Chameleon parallel programming tools users manual, Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[50] G. A. Geist, M. T. Heath, B. W. Peyton, P. H. Worley
A user's guide to PICL: a portable instrumented communication library, Technical Report TM-11616, Oak Ridge National Laboratory, Oct. 1990.

[51] Cray Research
PVM and HeNCE Programmer's Manual, SR-2501 version 3.0.

[52] R. Butler, E. Lusk
User's Guide to the p4 programming system, Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.

[53] R. Butler, E. Lusk
Monitors, messages, and clusters: the p4 parallel programming system,

Journal of Parallel Computing, 1994
to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-O493)

[54] A. Skjellum, A. Leung
Zipcode: a portable multicomputer communication library atop the reactive kernel, In D. W. Walker and Q. F. Stout, editors, Proceedings of the Fifth Distributed Memory Concurrent Computing Conference, pages 767-776, IEEE Press, 1990

[55] A. Skjellum, S. Smith, C. Still, A. Leung, M. Morari
The Zipcode message passing system, Technical report, Lawrence Livemore National Laboratory, Sept. 1992.

[56] L. Bomans, R. Hempel
The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on Intel iPCS/2, Parallel Computing, 15:119-132, 1990.

[57] R. Calkin, R. Hempel, H.-C. Hoppe, P. Wypior
Portable programming with parmacs message-passing library, Parallel Computing, Special issue on message-passing interfaces, to appear.

[58] P. Pierce
The NX/2 operating system, In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384-390, ACM Press, 1988.

[59] Parasoft Corporation, Pasadena, CA.
Express User's Guide, version 3.2.5 edition, 1992.

[60] nCUBE Corporation
nCUBE 2 Programmers Guide, r2.0, Dec. 1990.

[61] V. Bala, S. Kipnis
Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library, IBM T. J. Watson Center Technical Report, Oct. 1992.

[62] V. Bala, S. Kipnis, L. Rudolph, M. Snir
Designing efficient, scalable, and portable collective communication libraries, IBM T. J. Watson Center Technical Report, Oct. 1992.

[63] C. A. Thekkath, H. M. Levy, E. D. Lazowska
Efficient Suport for Multicomputing on ATM Networks, Technical Report TR 93-04-03, Department of Computer Science and Engineering, University of Washington, April 1993.

[64] M. Lin, J. Hsieh, D. Du, J. Thomas, J. MacDonald.
Distributed Network Computing Over Local ATM Networks. IEEE Journal on Selected Areas in Communications, Special Issue of ATM LANs: Implementations and Experiences with an Emerging Technology, Vol. 13, No 4, May 1995.

[65] S. Chang, D. Du, J. Hseih, M. Lin, R. Tsang.
Enhanced PVM Communications over a High-Speed Local Area Network. Paper from the Distributed Multimedia Center & Computer Science Department of the University of Minnesota Minneapolis.

[66] E. Biagioni, E. Coope, R. Samson.
Designing a Practical ATM LAN, *IEEE Network*, pages 32-39, March 1993.