Calcul sur les grands nombres et VLSI : application au PGCD, au PGCD étendu et à la distance euclidienne

Rachid Bouraoui

THESE

Présentée par

Rachid BOURAOUI

Pour obtenir le grade de **DOCTEUR**

DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté Ministériel du 30 mars 1992)

(spécialité : **Microélectronique**)

Calcul sur les grands nombres et VLSI : Application au PGCD, au PGCD étendu et à la distance euclidienne

---

Date de la soutenance : 15 janvier 1993

Composition du Jury :

Messieurs	Guy MAZARE	Président
	Hamid BESSALAH	Rapporteur
	Alain GUYOT	
	Jean-Michel MULLER	Rapporteur
	Jean-Louis ROCH	

Thèse préparée au sein du Laboratoire TIMA/INPG

A mes Parents,
Et à mes frères et sœurs de cœur.

' Dans l'interêt de la clarté, il m'a paru inévitable de me répéter souvent, sans me soucier le moins du monde de donner à mon exposé une forme élégante ; j'ai consciencieusement suivi l'avis du théoricien génial L. Boltzmann, de laisser le souci d'élégance aux tailleurs et aux cordonniers. '

Albert EINSTEIN

' La théorie sans la pratique est de l'utopie,
La pratique sans la théorie est de la superstition '

Alain GUYOT

REMERCIEMENTS

Je tiens à remercier M. Guy MAZARE, Professeur à l'ENSIMAG, qui a bien voulu me faire l'honneur de présider le jury de cette thèse. Ainsi que Messieurs Jean-Michel MULLER, Chargé de Recherche au CNRS et Hamid BESSALAH, Maître de Recherche et directeur du CDTA, qui ont assumé la tâche de rapporteur ainsi qu'à M. Jean-Louis ROCH, Maître de Conférence à l'ENSIMAG, qui a accepté d'être membre de mon jury.

Mes remerciements vont à M. Alain GUYOT, Maître de Conférence à l'ENSIMAG, qui m'a suivi tout au long de ce travail et à M. Bernard COURTOIS, Directeur de Recherche au CNRS et directeur du laboratoire TIMA, pour m'avoir accueilli dans l'équipe d'Architecture des Ordinateurs.

Ouvrant le chapitre des bénévoles honteusement exploités, je n'oublierais pas mes lecteurs Isabelle AMIELH, Dominique BRAME, Nathalie REVOL, Ali SKAF, André VACHER, Serge VIDAL. Ne pouvant citer ici tous ceux qui m'ont aidé au laboratoire TIMA, notamment Mohamed AICHOUCHI, Hakim BEDERR, Omar KEBICHI, Michael NICOLAIDIS, Kevin O'BRIEN... ainsi que dans les autres laboratoires de l'IMAG, qu'ils trouvent ici l'expression de ma gratitude...

Enfin, je remercie ceux qui ont eu la patience de m'aider et de me soutenir, au cours de ces dernières années en particulier M. Essaïd BOURAOUI et sa femme .

RESUME

Dans le cadre de cette thèse nous avons étudié l'implantation des algorithmes de l'arithmétique "en ligne". En particulier, la réalisation de deux circuits destinés aux applications exigeant une précision infinie est exposée. En effet, dans de nombreux domaines tels que la génération de nombres aléatoires, cryptographie, calcul formel, arithmétique exacte, réduction de fraction en précision infinie, calcul modulaire, traitement d'images..., les opérateurs classiques manquent d'efficacité. Face à ce type de problèmes, un remède peut être apporté par le calcul en ligne selon lequel les calculs sont faits en introduisant les opérandes en série chiffre à chiffre en notation redondante. Nous obtenons ainsi un haut degré de parallélisme et une précision variable linéairement.

Le premier circuit présenté implante un algorithme de PGCD nommé EUCLIDE offrant, d'après les simulations, le meilleur compromis coût matériel/performance. Il donne également les coefficients de Bezout. Ce circuit est appelé à résoudre les problèmes liés au temps de calcul du PGCD par les méthodes classiques rencontrés dans beaucoup d'applications.

Une deuxième application montre la possibilité de fusionner des opérateurs en ligne afin d'obtenir un opérateur complexe. L'exemple traité dans cette thèse est celui de la distance euclidienne : $Z = \sqrt{X^2 + Y^2}$ utilisée, entre autres, pour la résolution du moindre carré des systèmes linéaires.

Mots-clés

PGCD, PGCD étendu, Notation redondante, Précision infinie, Calcul en ligne, Distance euclidienne, Conception VLSI

ABSTRACT

In this thesis we have studied the implementation of on-line arithmetic algorithms. In particular, the design of two VLSI circuits destined for high-precision applications is shown. In fact, in numerous fields, such as pseudorandom number generation, cryptography, computer algebra, exact arithmetic, reduction of fractions with infinite precision, modular computing, image processing..., classical operators are inefficient. A solution to this type of problem can be provided through on-line arithmetic by which computations are performed by introducing operands serially digit-by-digit in a redundant notation.

The first circuit presented implements a GCD algorithm known as EUCLIDE providing, according to simulations, the best hardware cost/performance compromise. It also provides the Bezout coefficients. This circuit is to be used in order to solve problems related to the classical GCD method found in many applications.

A second application shows the possibility of cascading on-line operators in order to build up new ones. The example reported in this thesis is that of the on-line computation of the euclidean distance : $Z = \sqrt{X^2 + Y^2}$ used, among other things, in the least-square resolution of linear systems.

Keywords

GCD, Extended GCD, Redundant notation, High precision, On-line computation, Euclidean distance, VLSI design

UNIVERSITE
JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. MEDECINE
GRENOBLE 1

Président de l'Université :

M. NEMOZ Alain

ANNEE UNIVERSITAIRE 1990 - 1991

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ERE CLASSE

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARVIEU Robert	Physique Nucléaire I.S.N.
AURIAULT Jean Louis	Mécanique
BARRA Jean René	Statistiques - Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean Paul	Mathématiques Pures
BILLET Jean	Géographie
BLANCHI Jean Pierre	A.P.S.
BOEHLER Jean Paul	Mécanique
BOITET Christian	Informatique et Mathématiques Appliquées
BORNAREL Jean	
BRUANDET Jean François	Physique
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEMAILLY Jean Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean Pierre	Mécanique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences nucléaires
GILLARD Roland	Mathématiques
GUITTON Jacques	Chimie

../ ...

BATIMENT ADMINISTRATIF, 621, AVENUE CENTRALE, DOMAINE UNIVERSITAIRE SAINT-MARTIN-D'HERES - GIERES - B.P 53 X - 38041 GRENOBLE CEDEX
TELEPHONE : (33) 76 51 46 00 - TELEX : 980 134 F - TELECOPIE : (33) 76 51 48 48

HERAULT Jeanny	Physique
HICTER Pierre	Chimie
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures
JOSELEAU Jean Paul	Biochimie
KAHANE André	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LOISEAUX Jean Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
NEMOZ Alain	Physique
OMONT Alain	Astrophysique
PELMONT Jean	Biochimie
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RICHARD Jean Marc	
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie C.E.R.M.A.V.
ROBERT Jean Bernard	
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VALLADE Marcel	Physique
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie
VIDAL Micheal	

PROFESSEURS DE 2EME CLASSE

APPARU Marcel	Chimie
ARMAND Gilbert	Géographie
ARNAUD Hubert	Géologie
ARTRU Marie Christine	Physique
ATTANE Pierre	Mécanique
BARATE Robert	Sciences Nucléaires
BARET Paul	Chimie
BARGE Jean	Mathématiques
BARLET Roger	Chimie
BERTIN José	Mathématiques
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORRIONE Dominique	Automatique informatique
BOULON Marc	Mécanique
BOUTRON Claude	Glaciologie
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUGAL Gérard	Biologie
CAMPILLO Michel	Géophysique
CAVAILLE Jean Yves	Chimie
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COTTET Georges-Henri	Modélisation, calcul scientifique, statis.
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique
DAVID Jean	Géographie
DEROUARD Jacques	Physique
DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
DUPUY Claude	Chimie
DURAND Mireille	Sciences Nucléaires
FONTECAVE Marc	Chimie
FOURNIER Jean Marc	Physique
GASPARD François	Physique
GIDON Maurice	Géologie
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GOURC Jean Pierre	Mécanique
GUIGO Maryse	Géographie
GUMUCHIAN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HAMMOU Abdelkader	Chimie
HERBIN Jacky	Géographie
HERINO Roland	Physique
HERZOG Michel	Biologie
JARDON Pierre	Chimie
JUTTEN Christian	Physique
KERCKHOVE Claude	Géologie
KOSAREW Siegmund	Math. fondamentales et appliquées
KLINGER Jurgen	Glaciologie
LAURENT Christine	Mathématiques
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MERCHEZ Fernand	Physique
MILAS Michel	Chimie
MOREL Alain	Géographie
MORIN Pierre	Physique
NGUYEN HUY Xuong-	Informatique
OUDET Bruno	Mathématiques Appliquées

../ ...

PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Claude	Biologie
PIBOULE Michel	Géologie
PORTESEIL Jean Louis	Physique
PUECH Laurent	Physique
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
ROBERT Claudine	Didactique des disciplines scientifiques
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
SAJOT Gérard	Physique
SARROT REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
TEMPERVILLE André	Mécanique
TISSUT Michel	Biologie
TOURNIER Evelyne	Informatique et Mathématiques appliquées
VALLADE Marcel	Physique
VALLON Michel	Glaciologie
VICAT Jean	Physique
VINCENS Maurice	Chimie
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie
WITOMSKI Patrick	

PRESIDENT DE L'INSTITUT
Monsieur Maurice RENAUD

Année 1991/1992

PROFESSEURS DES UNIVERSITES

BARIBAUD	Michel	ENSERG
BARRAUD	Alain	ENSIEG
BAUDELET	Bernard	ENSPG
BAUDIN	Gérard	UFR PGP
BEAUFILS	Jean-Pierre	ENSIEG/ILL
BOIS	Philippe	ENSHMG
BONNET	Guy	ENSPG
BOUVIER	Gérard	ENSERG
BRISSONNEAU	Pierre	ENSIEG
BRUNET	Yves	CUEFA
CAILLERIE	Denis	ENSHMG
CAVAIGNAC	Jean-François	ENSPG
CHARTIER	Germain	ENSPG
CHENEVIER	Pierre	ENSERG
CHERADAME	Hervé	UFR/PGP
CHERUY	Arlette	ENSIEG
CHOYET	Alain	ENSERG
COGNET	Gérard	ENSHMG
COLINET	Catherine	ENSEEG
COMMAULT	Christian	ENSIEG
CORNUT	Bruno	ENSIEG
COULOMB	Jean-Louis	ENSIEG
CROWLEY	James	ENSIMAG
DALARD	Francis	ENSEEG
DARVE	Félix	ENSHMG
DELLA DORA	Jean	ENSIMAG
DEPEY	Maurice	ENSERG
DEPORTES	Jacques	ENSPG
DEROO	Daniel	ENSEEG
DESRE	Pierre	ENSEEG
DIARD	Jean-Paul	ENSEEG
DOLMAZON	Jean-Marc	ENSERG
DURAND	Francis	ENSEEG
DURAND	Jean-Louis	ENSPG
FAUTRELLE	Yves	ENSHMG
FOGGIA	Albert	ENSIEG
FONLUPT	Jean	ENSIMAG
FOULARD	Claude	ENSIEG
GALERIE	Alain	ENSEEG
GANDINI	Alessandro	UFR/PGP
GAUBERT	Claude	ENSPG
GENTIL	Pierre	ENSERG
GENTIL	Sylviane	ENSIEG
GREVEN	Hélène	CUEFA
GUERIN	Bernard	ENSERG
GUYOT	Pierre	ENSEEG
IVANES	Marcel	ENSIEG
JALLUT	Christian	ENSEEG

JANOT	Marie-Thérese	ENSERG
JAUSSAUD	Pierre	ENSIEG
JOST	Rémy	ENSPG
JOUBERT	Jean-Claude	ENSPG
JOURDAIN	Geneviève	ENSIEG
KUENY	Jean-Louis	ENSHMG
LACHENAL	Dominique	UFR PGP
LACOUME	Jean-Louis	ENSIEG
LADET	Pierre	ENSIEG
LESIEUR	Marcel	ENSHMG
LESPINARD	Georges	ENSHMG
LIENARD	Joël	ENSIEG
LONGEQUEUE	Jean-Pierre	ENSPG
LORET	Benjamin	ENSHMG
LOUCHET	François	ENSEEG
LUCAZEAU	Guy	ENSEEG
LUX	Augustin	ENSIMAG
MASSE	Philippe	ENSIEG
MASSELOT	Christian	ENSIEG
MAZARE	Guy	ENSIMAG
MICHEL	Gérard	ENSIMAG
MOHR	Roger	ENSIMAG
MOREAU	René	ENSHMG
MORET	Roger	ENSIEG
MOSSIERE	Jacques	ENSIMAG
OBLED	Charles	ENSHMG
OZIL	Patrick	ENSEEG
PANANAKAKIS	Georges	ENSERG
PAULEAU	Yves	ENSEEG
PERRET	Robert	ENSIEG
PIAU	Jean-Michel	ENSHMG
PIC	Etienne	ENSERG
PLATEAU	Brigitte	ENSIMAG
POUPOT	Christian	ENSERG
RAMEAU	Jean-Jacques	ENSEEG
REINISCH	Raymond	ENSPG
RENAUD	Maurice	UFR/PGP
ROBERT	François	ENSIMAG
ROSSIGNOL	Michel	ENSPG
ROYE	Daniel	ENSIEG
SABONNADIERE	Jean-Claude	ENSIEG
SAGUET	Pierre	ENSERG
SAUCIER	Gabriéle	ENSIMAG
SCHLENKER	Claire	ENSPG
SCHLENKER	Michel	ENSPG
SILVY	Jacques	UFR/PGP
SIRIEYS	Pierre	ENSHMG
SOHM	Jean-Claude	ENSEEG
SOLER	Jean-Louis	ENSIMAG
SOUQUET	Jean-Louis	ENSEEG
TICHKIEWITCH	Serge	ENSHMG
TROMPETTE	Philippe	ENSHMG
TRYSTRAM	Denis	ENSGI
VEILLON	Gérard	ENSIMAG
VERJUS	Jean-Pierre	ENSIMAG
VINCENT	Henri	ENSPG
ZADWORNY	François	ENSERG

SITUATION PARTICULIERE

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSERG	BLIMAN	Samuel	Mutation	
ENSPG	BLOCH	Daniel	Recteur	21/12/93
ENSIMAG	LATOMBE	Jean-Claude	Détachement	01/05/93
ENSHMG	PIERRARD	Jean-Marie	Disponible	

RETRAITE

ENSEEG	BONNETAIN	Lucien

DIRECTEURS DE RECHERCHE CNRS

ABELLO	Louis
ALDEBERT	Pierre
ALEMANY	Antoine
ALLIBERT	Colette
ALLIBERT	Michel
ANSARA	Ibrahim
ARMAND	Michel
AUDIER	Marc
AUGOYARD	Jean-François
AVIGNON	Michel
BERNARD	Claude
BINDER	Gilbert
BLAISING	Jean-Jacques
BONNET	Roland
BORNARD	Guy
BOUCHERLE	Jean-Xavier
CAILLET	Marcel
CARRE	René
CHASSERY	Jean-Marc
CHATILLON	Christian
CIBERT	Joël
CLERMONT	Jean-Robert
COURTOIS	Bernard
CRIQUI	Patrick
CRISTOLOVEANU	Sorin
DAVID	René
DION	Jean-Michel
DOUSSIERE	Jacques
DRIOLE	Jean
DUCHET	Pierre
DUGARD	Luc
DURAND	Robert
ESCUDIER	Pierre
EUSTATHOPOULOS	Nicolas
FINON	Dominique
FRUCHARD	Robert
GARNIER	Marcel
GIROD	Jacques
GLANGEAUD	François
GUELIN	Pierre
HOPFINGER	Emil
JORRAND	Philippe
JOUD	Jean-Charles
KAMARINOS	Georges
KLEITZ	Michel
KOFMAN	Walter
LACROIX	Claudine
LANDAU	Ioan
LAULHERE	Jean-Pierre
LEGRAND	Michel
LEJEUNE	Gérard
LEPROVOST	Christian
MADAR	Roland
MARTIN	Jean-Marie
MERMET	Jean
MEUNIER	Gérard
MICHEL	Jean-Marie
NAYROLLES	Bernard
PASTUREL	Alain
PEUZIN	Jean-Claude
PHAM	Antoine
PIAU	Monique
PIQUE	Jean-Paul
POINSIGNON	Christiane
PREJEAN	Jean-Jacques
RENOUARD	Dominique
SENATEUR	Jean-Pierre
SIFAKIS	Joseph
SIMON	Jean-Paul
SUERY	Michel
TEODOSIU	Christian
VACHAUD	Georges
VAUCLIN	Michel
WACK	Bernard
YAYARI	Ali-Reza
YONNET	Jean-Paul

PERSONNES AYANT OBTENU LE DIPLOME

D'HABILITATION A DIRIGER DES RECHERCHES

BALESTRA	Francis
BALME	Louis
BECKER	Monique
BIGEON	Jean
BINDER	Zdeneck
BOE	Louis-Jean
CANUDAS DE WIT	Carlos
CHOLLET	Jean-Pierre
COEY	Jean-Pierre
CORNUEJOLS	Gerard
COURNIL	Michel
CRASTES DE PAULET	Michel
DALLERY	Yves
DESCOTES-GENON	Bernard
DUGARD	Luc
DURAND	Madeleine
FERRIEUX	Jean-Paul
FEUILLET	René
FREIN	Yannick
GAUTHIER	Jean-Paul
GHIBAUDO	Gérard
GUILLEMOT	Nadine
GUYOT	Alain
HAMAR	Sylviane
HAMAR	Roger
HORAUD	Patrice
JACQUET	Paul
LATOMBE	Claudine
LE HUY	Hoang
LE GORREC	Bernard
LOZANO-LEAL	Rogelio
MACOVSCHI	Mihaïl
MAHEY	Philippe
METAIS	Olivier
MONMUSSON-PICQ	Georgette
MORY	Mathieu
MULLER	Jean
MULLER	Jean-Michel
NGUYEN TRONG	Bernadette
NIEZ	Jean-Jacques
PERRIER	Pascal
PLA	Fernand
RECHENMANN	François
ROGNON	Jean-Pierre
ROUGER	Jean
ROUX	Jean-Claude
TCHUENT	Maurice

PERSONNES AYANT OBTENU LE DIPLOME

DE DOCTEUR D'ETAT INPG

ABDEL-RAZEK	Adel
AKSAS	Haris
ALLA	Hassane
AMER	Ahmed
ANCELLE	Bernard
ANGENIEUX	Gilbert
ATMANI	Hamid
AYEDI	Hassine Feri
A.BADR	Osman
BACHIR	Aziz
BALANZAT	Emmanuel
BALTER	Roland
BARDEL	Robert
BARRAL	Gérard
BAUDON	Yves
BAUSSAND	Patrick
BEAUX	Jacques
BEGUINOT	Jean
BELLISSENT née FUNEZ	Marie-Claire
BELLON	Catherine
BEN RAIS	Abdejettah
BERGER-SABBATEL	Gilles
BERNACHE-ASSOLANT	Didier
BEROYAL	Abderrahmane
BERTHOD	Jacques
BILLARD	Dominique
BLANC épouse FOULETIER	Mireille
BOCHU	Bernard
BOJO	Gilles
BOKSENBAUM	Claude
BOLOPION	Alain
BONNARD	Bernard
BORRIONE	Dominique
BOUCHACOURT	Michel
BRINI	Jean
BRION	Bernard
CAIRE	Jean-Pierre
CAMEL	Denis
CAPERAN	Philippe
CAPLAIN	Michel
CAPOLINO	Gérard
CASPI	Paul
CHAN-TUNG	Nam
CHASSANDE	Jean-Pierre
CHATAIN	Dominique
CHEHIKIAN	Alain
CHIRAMELLA	Yves
CHILO	Jean
CHUPIN	Jean-Claude
COLONNA	Jean-François
COMITI	Jacques
CORDET	Christian
COUDURIER	Lucien

COUTAZ	Jean-Louis
CREUTIN	Jean-Dominique
DAO	Trongtich
DARONDEAU	Philippe
DAVID	Bertrand
DE LA SEN	Manuel
DELACHAUME	Jean-Claude
DENAT	André
DESCHIZEAUX née CHERUY	Marie-Noëlle
DIJON	Jean
DOREMUS	Pierre
DUPEUX	Michel
EL ADHAM	Karim
EL OMAR	Fevaz
EL-HENNAWY	Adel
ETAY	Jacqueline
FABRE épouse MAXIMOVITCH	Suzanne
FAURE-BONTE épouse MARET	Mireille
FAVIER	Denis
FAVIER	Jean-Jacques
FELIACHI	Movloud
FERYAL	Haj Hassan
FLANDRIN	Patrick
FOREST	Bernard
FORESTIER	Michel
FOSTER	Panayotis
FRANC	Jean-Pierre
GADELLE	Patrice
GARDAN	Yvon
GENIN	Jacques
GERVASON	Georges
GILORMINI	Pierre
GINOUX	Jean-Louis
GOUMIRI	Louis
GROC	Bernard
GROSJEAN	André
GUEDON	Jean-Yves
GUERIN	Jean-Claude
GUESSOUS	Anas
GUIBOUD-RIBAUD	Serge
HALBWACHS	Nicolas
HAMMOURI	Hassan
HEDEIROS SILIVEIRA	Hamilton
HERAULT	Jeanny
HONER	Claude
HUECKEL	Tomasz
IGNAT	Michel
ILIADIS	Athanasios
JANIN	Gérard
JERRAYA	Ahmed Amine
JUTTEN	Christian
KAHIL	Hassan
KHUONGQUANG	Dong
KILLIS	Andreas
KONE	Ali
LABEAU	Michel
LACAZE	Alain
LACROIX	Jean-Claude
LANG	Jean-Claude
LATHUILLERE	Chantal

LATY	Pierre
LAUGIER	Christian
LE CADRE	Jean-Pierre
LE GARDEYR	René
LE NEST	Jean-François
LE THIESSE	Jean-Claude
LEMAIGNAN	Clement
LEMUET	Daniel
LEYEQUE	Jean-Luc
LONDICHE	Henry
L'HERITIER	Philippe
MAGNIN	Thierry
MAISON	François
MAMWI	Abdullah
MANTEL épouse SIEBERT	Elisabeth
MARCON	Guy
MARTINEZ	Francis
MARTIN-GARIN	Lionel
MASSE	Dominique
MAZER	Emmanuel
MERCKEL	Gérard
MEUNIER	Jean
MILI	Ali
MOALLA	Mohamed
MODE	Jean-Michel
MONLLOR	Christian
MONTELLA	Claude
MORET	Frédéric
MRAYATI	Mohammed
M'SAAD	Mohammed
M'SIRDI	Kouider Nace
NEPOMIASTCHY	Pierre
NGUYEN	Trong Khoi
NGUYEN-XUAN-DANG	Michel
ORANIER	Bernard
ORTEGA MARTINEZ	Roméo
PAIDASSI	Serge
PASSERONE	Alberto
PEGON	Pierre
PIJOLAT	Christophe
POGGI	Yves
POIGNET	Jean-Claude
PONS	Michel
POU	Tong Eok
RAFINEJAD	Paiviz
RAGAIE	Marie Fikri
RAHAL	Salah
RAMA SEABRA SANTOS	Fernando
RAVAINE	Denis
RAZBAN-HAGHIGHI	Tchanguiz
RAZZOUK	Micham
REGAZZONI	Gilles
RIQUET	Jean-Pierre
ROBACH	Chantal
ROBERT	Yves
ROGEZ	Jacques
ROHMER	Jean
ROUSSEL	Claude
SAAD	Abdallah
SAAD	Youcef

SABRY	Mohamed Nabi
SALON	Marie-Christine
SAUBAT épouse MARCUS	Bernadette
SCHMITT	Jean-Hubert
SCHOELLKOPF	Jean-Pierre
SCHOLL	Michel
SCHOLL	Pierre-Claude
SCHOULER	Edmond
SCHWARTZ	Jean-Luc
SEGUIN	Jean
SIWY	Jacques
SKALLI	Abdellatif
SKALLI HOUSSEYNI	Abdelali
SOUCHON	Alain
SUETRY	Jean
TALLAJ	Nizar
TEDJAR	Farouk
TEDJINI	Smail
TEYSSANDIER	Francis
THEVENODFOSSE	Pascale
TMAR	Mohamed
TRIOLLIER	Michel
TUFFELIT	Denis
TZIRITAS	Georges
VALLIN	Didier
VELAZCO	Raoul
VERDILLON	André
VERMANDE	Alain
VIKTOROVITCH	Pierre
VITRANT	Guy
WEISS	François
YAZAMI	Rachid

Table des matières

INTRODUCTION

Introduction

Ce document rend compte de travaux effectuée au sein de l'équipe d'Architecture des Ordinateurs du laboratoire TIMA. Les préoccupations actuelles de cette équipe de recherche peuvent se présenter comme suit :

- méthodologies architecturales pour le test avancé des systèmes VLSI (ARCHIMEDES),
- conception de circuits "semi-custom" à très haute sécurité (FASED),
- génération de test et de conception en vue de la testabilité,
- intégration à très grande échelle d'une unité de contrôle pour les systèmes de sécurité critique,
- synthèse d'architecture pour les systèmes intégrés complexes (ASCIS),
- Arithmétique Rapide sur Calculateurs (ARC).

L'équipe Architecture des Ordinateurs compte près de trente personnes dont douze sont spécialisées dans le test et la sécurité de fonctionnement, dix en synthèse et huit en conception des circuits intégrés. Cette thèse s'est déroulée au sein de l'équipe de conception. Cette équipe a commencé ses activités de conception par la réalisation de circuits de calcul. Dans le cas du projet FELIN [COG 87] au cours duquel a été conçu un circuit pour le calcul de fonctions élémentaires usuelles en grande précision qui faisaient défaut aux bibliothèques de calculs disponibles sur ordinateur. Parmi les fonctions élémentaires de base que calculait le circuit FELIN, nous pouvons citer : sinus, cosinus, arctangente, exponentielle, logarithme, racine carrée. D'autres fonctions non primitives ont également été réalisées : x^y, $1/x$, x^2, $\log_2$, $\log_{10}$, sh, ch, th, argth, argch, argsh, arcsin, arccos.

Un deuxième projet a succédé à FELIN : OCAPI (Opérateur de Calcul en Précision Illimitée) pendant lequel l'équipe s'est intéressée au calcul de l'addition, de la soustraction, de la division, du maximum et du minimum de deux grands nombres (dont la taille peut atteindre 2048 chiffres binaires) [GHM 89, GUK 91], et aussi de deux nombres en précision infinie. Le terme "précision infinie" signifie que l'on peut travailler avec autant de bits ou chiffres significatifs que l'on veut ! La transmission des nombres se fait en série chiffre après chiffre. Le calcul se fait donc en ligne et nous obtenons un chiffre du résultat pour chaque nouveau chiffre des opérandes. Les avantages des opérateurs en ligne sont reconnus dans le traitement du signal [GHP 89, GUM 91, GUY 92].

Un troisième projet : le projet EUCLIDE, auquel j'ai participé, a démarré ensuite. Son but est la réalisation d'un nouveau circuit pour le calcul du PGCD (Plus Grand Commun Diviseur)

de deux grands nombres. A ce jour, il n'existe aucun algorithme de PGCD qui ne calcule pas (au moins implicitement) les coefficients de Bezout [ROC 90] associés. Or, il est essentiel de construire un circuit pour des entiers de taille finie (inférieur à n_0 bits) qui accélère le calcul du PGCD de deux nombres quelconques de n bits. Dans l'état actuel des connaissances, seul le calcul des coefficients de Bezout et du PGCD permet l'extension de n_0 à n, grâce aux algorithmes de Schönhage [SCH 71] ou de Lehmer [ROC 90]. Ainsi, le meilleur algorithme séquentiel connu [SCH 71] permet le calcul du PGCD de deux entiers de n chiffres en $O(n\log^2 n \log\log n)$: il est basé sur deux calculs en séquence de PGCD étendu, avec coefficients de Bezout, de nombre de n/2 chiffres.

Ce projet vise à couvrir les insuffisances des calculateurs classiques lors du calcul du PGCD de deux grands nombres. En effet, dans les calculs exacts, près de 80% du temps de l'Unité Centrale (UC) est passé dans un tel calcul [DAR 85]. Face à ces difficultés, nous avons pensé à implanter les algorithmes de calcul du PGCD de grands nombres en matériel et à utiliser par la suite des circuits intégrés qui seront certainement beaucoup plus rapides que l'exécution de logiciels spécifiques. Avant l'implantation de ces algorithmes, ces derniers doivent être simulés pour pouvoir suivre de près leur fonctionnement, les modifier et les optimiser du point de vue performance et coût en matériel, ce qui est beaucoup plus "souple" à réaliser en programmant qu'en construisant des circuits dont nous ignorons les performances optimales.

Remarquons que l'on peut penser que le circuit OCAPI, déjà réalisé, serait utile pour le calcul du PGCD en grande précision, mais il faut toutefois rappeler que les entrées de ce circuit sont introduites en série et que nous obtenons un bit à chaque nombre du résultat par cycle machine. Par ailleurs également, il faut noter que les algorithmes de calcul du PGCD que nous allons exposer sont itératifs (EUCLIDE, STEIN, ...) et par suite, les sorties de l'itération i -1 seront les entrées de l'itération i. Il suffit donc d'imaginer le temps pendant lequel nous attendons à chaque itération de l'algorithme pour pouvoir enfin réitérer le processus avec des nombres de taille 2048 bits !!! C'est en se basant sur ces remarques très importantes que nous avons décidé de concevoir un autre circuit, différent d'OCAPI, pour calculer le PGCD de deux entiers. Ce circuit tiendra compte de la dépendance entre les opérandes et les traitera en parallèle, mais ne donnera un résultat que lorsque toutes les opérandes seront présentes.

Cette thèse présente deux opérateurs de calcul en précision illimitée. Le présent document est organisé en sept chapitres.

Le premier chapitre présente les généralités sur le calcul en précision infinie et la conception de circuits VLSI.

Le chapitre II présente le projet EUCLIDE. Il s'agit de l'étude des algorithmes de calcul du PGCD de deux petits nombres, ceux réalisables par les calculateurs arithmétiques existants, et

de l'étude des principes du circuit OCAPI pour constater son inefficacité pour le calcul du PGCD de deux entiers et avoir une idée sur le traitement des grands nombres de taille variant entre 100 et 2048 bits.

Le chapitre III présente le développement des algorithmes. Il s'agit de la modification des algorithmes déjà existants, de leur adaptation aux grands nombres et de leur optimisation.

Le chapitre IV présente l'architecture. Il s'agit de réaliser une architecture pour chacune des approches et variantes d'une même approche algorithmique.

Le chapitre V présente la simulation et le choix de l'approche à implanter. Il s'agit de la simulation des différentes approches, donc des différents circuits, l'étude de leur complexité (performances et coût en matériel), de leur comparaison et du choix de la meilleure approche à implanter.

Le chapitre VI présente la conception du circuit réalisant l'algorithme du PGCD [BGK 91, BGR 90, BGU 92, GUY 91], constitué de quatre parties, une partie opérative 128 bits utilisant une notation redondante, une partie opérative 4 bits ou tête utilisant une notation complément à deux, une partie opérative 12 bits et une pile LIFO pour stocker les opérations de calcul du PGCD. La première est réalisée en dessin au micron afin d'obtenir une grande densité, alors que les trois dernières sont réalisées à partir de cellules standard de la bibliothèque ES2.

Cette même démarche nous a également servi à concevoir un nouveau circuit [BGW 92, BGW 93], décrit dans le dernier chapitre. Les algorithmes des opérateurs en ligne pour la multiplication, le carré, la division et la racine carrée présentés dans [GHM 89, GUK 91] peuvent être enchainés en un seul bloc pour calculer en ligne la distance euclidienne : $Z = \sqrt{X^2 + Y^2}$ utilisant la notation redondante. Cela entrainerait un faible temps de propagation et une réduction de 20% en nombre de transistors par rapport à ceux réalisés avec les opérateurs en ligne séparés. L'opérateur de la distance euclidienne peut servir à la résolution du moindre carré des systèmes linéaires tel que le filtre de Kalman [STC 90]. L'effort de conception concernant ce circuit est divisé en trois parties. La première partie résume le travail effectué pour développer, vérifier et modifier l'algorithme qui contient la base de la structure de l'opérateur. Dans la deuxième partie, les circuits de l'opérateur seront développés et simulés. La dernière portera sur la conception proprement dite d'un circuit de l'opérateur composé d'une partie opérative 32 bits dessinée au micron et d'une partie contrôle réalisée à partir de cellules standard de la bibliothèque ES2.

CHAPITRE I

GENERALITES

I.1. CALCUL EN PRECISION INFINIE

I.1.1. Introduction

I.I.2. Principes de la représentation redondante des nombres

I.I.3. Etude d'un exemple d'opérateur de calcul en précision illimitée : OCAPI

I.2. CONCEPTION DE CIRCUITS VLSI

I.3. CONCLUSION

I.1. CALCUL EN PRECISION INFINIE

I.1.1. Introduction

Dans certains domaines, tels que la cryptographie, l'intelligence artificielle (Prolog 3), la conception assistée par ordinateur, le calcul formel, le calcul modulaire, la génération de nombres aléatoires, etc..., le besoin de calculs exacts se fait sentir. Le calcul exact, c'est à dire sans troncature, entraîne un accroissement rapide du nombre de chiffres significatifs à chaque opération.

Cependant, les processeurs qui existent actuellement travaillent en général suivant le format IEEE 754, avec des nombres limités à 23 bits ou à 52 bits de mantisse respectivement pour le format court et le format étendu. En utilisant ces processeurs pour le calcul en très grande précision, les temps de calculs deviennent très importants à cause de la propagation de retenue qui empêche de paralléliser les calculs. Dans ce cas, il est nécessaire d'utiliser un autre système, une autre méthode pour améliorer la vitesse de calcul.

Un moyen pour résoudre ce problème est d'utiliser des représentations redondantes des nombres car celles-ci évitent la propagation de retenues lors des additions. Avizienis, en 1961 [AVI 61], a proposé une représentation redondante des nombres pour l'arithmétique parallèle. Aujourd'hui, quelques algorithmes de calculs basés sur ces représentations redondantes ont été développés. Les grands nombres ne peuvent être transmis qu'en série, chiffre à chiffre (chaque chiffre est représenté sur un certain nombres) car les transmettre en parallèle exigerait un nombre de connexions trop importante. Parmi ceux-ci, on peut citer les algorithmes en ligne (on-line) de division et multiplication introduits par M.D. Ercegovac et K.S. Trivedi [ERT 77]. Plus récemment, en juin 1989, le groupe de conception des circuits intégrés au laboratoire TIMA de Grenoble [GHM 89, GUK 91] a conçu un opérateur d'addition, de soustraction, de multiplication, de calcul du maximum et du minimum de deux grands nombres et qui se base sur les algorithmes en ligne appelés OCAPI.

Cet effort s'est poursuivi par la conception de deux nouveaux circuits, EUCLIDE pour le calcul du PGCD de deux entiers [BGR 90, BGK 91, GUY 91, BGU 92] et la distance euclidienne en ligne [BGW 92, BGW 93]. Plusieurs domaines exigent ces calculs en précision infinie.

I.I.2. Principes de la représentation redondante des nombres

Il s'agit là d'une classe de systèmes d'écriture des nombres qui englobe la notation usuelle de position. Pour la présenter, nous allons nous intéresser à l'écriture en base B de nombres

entiers avec des chiffres n'appartenant pas forcément à {0, 1, ..., B-1}. Nous convenons de noter les chiffres négatifs à l'aide d'une barre, afin d'éviter tout risque de confusion entre le signe "-" de l'écriture des nombres négatifs et le signe "-" de la soustraction. Par exemple, si nous avons choisi d'écrire en base 10 avec cinq chiffres appartenant à :

$\{\bar{5}, \bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5\}$
nous pourrons écrire,
$4610 = (05\bar{4}10)_{10}$
puisque,
$5.10^3 + (-4).10^2 + 1.10^1 + 0.10^0 = 4610$
De même,
$9999 = (1000\bar{1})_{10}$
puisque,
$1.10^4 + (-1).10^0 = 9999$

Cette représentation permet d'effectuer des additions exemptes de toute propagation de retenue, c'est à dire en un temps indépendant de la taille du nombre redondant. Nous allons présenter maintenant l'algorithme d'addition pour les nombres écrits dans un système redondant (chiffres signés). Soit D, un nombre redondant en base B, défini comme suit :

$$D = \sum_{i=0}^{n-1} d_i B^i \qquad \forall i,\ d_i \in \{-m, ..., 0, m\}$$

avec,

$m \in \{(B+1)/2, B-1\}$ si B est impair
$m \in \{(B+2)/2, B-1\}$ si B est pair

L'algorithme présenté par Avizienis [AVI 61], ne permet l'opération d'addition de nombres écrits dans un tel système que dans l'hypothèse où la base est supérieure à 2, puisque la condition exigée est :

$$m \leq B - 1$$
$$2m \geq B + 1$$

Il est évident que la deuxième condition n'est pas satisfaite pour $B = 2$ et $m = 1$.

La base choisie dans notre application est B = 2. Les algorithmes présentés dans [CYR 78] et [HOP 85], donnent dans ce cas la solution. La description suivante, celle de [CYR 78], citée aussi dans [MUL 89], est beaucoup plus facile à implanter que celle proposée par Avizienis. Supposons que nous désirons additionner deux nombres A et B qui s'écrivent respectivement $[a_{n-1}a_{n-2}...a_0]_{CS2}$ et $[b_{n-1}b_{n-2}...b_0]_{CS2}$ dans la notation "chiffres signés" en base 2, et considérons des quantités $C = [c_{n-1}c_{n-2}...c_0]_{CS2}$ et $S = [s_{n-1}s_{n-2}...s_0]_{CS2}$, définies de manière non unique par :

$$a_i + b_i = 2c_i + s_i \quad \text{avec} \quad c_i \text{ et } s_i \in \{\bar{1}, 0, 1\}$$

Il est évident que la somme de A et B est égale à 2C + S. La somme écrite en "chiffres signés" sous la forme $Z = [z_n z_{n-1}...z_0]_{CS2}$ s'obtient en effectuant pour tout i :

$$z_i = s_i + c_{i-1}, \quad \text{avec les conventions} \quad s_n = c_{-1} = 0$$

Pour que les valeurs de z_i appartiennent à $\{\bar{1}, 0, 1\}$ il suffit que :

$$(s_i, c_{i-1}) \notin \{(\bar{1}, \bar{1}), (1, 1)\}$$

La table d'addition suivante montre comment choisir les valeurs de c_i et s_i pour obtenir les valeurs de z_i adéquates.

a_i	b_i	condition	s_i	c_i
0	0	aucune	0	0
1	$\bar{1}$	aucune	0	0
$\bar{1}$	1	aucune	0	0
1	1	aucune	0	1
$\bar{1}$	$\bar{1}$	aucune	0	$\bar{1}$
1	0	$a_{i-1} + b_{i-1} > 0$	$\bar{1}$	1
		$a_{i-1} + b_{i-1} \leq 0$	1	0
0	1	$a_{i-1} + b_{i-1} > 0$	$\bar{1}$	1
		$a_{i-1} + b_{i-1} \leq 0$	1	0
$\bar{1}$	0	$a_{i-1} + b_{i-1} > 0$	$\bar{1}$	0
		$a_{i-1} + b_{i-1} \leq 0$	1	$\bar{1}$
0	$\bar{1}$	$a_{i-1} + b_{i-1} > 0$	$\bar{1}$	0
		$a_{i-1} + b_{i-1} \leq 0$	1	$\bar{1}$

Table I.1 : Addition en binaire redondant ou en "chiffres signés"

Puisque le calcul des couples (c_i, s_i) nécessite seulement la connaissance de a_{i-1}, b_{i-1}, a_i et b_i, ce calcul peut se faire de manière totalement parallèle, sans aucune propagation de retenue, et en temps indépendant de la taille des opérandes.

L'exemple suivant montre l'addition des nombres 179 et 105 écrits en "chiffres signés" :

$[\,1\ 1\ 0\ \bar{1}\ 0\ 1\ 0\ \bar{1}\,]_{CS2}$	A (=179)
$[\,1\ 0\ \bar{1}\ 1\ \bar{1}\ 0\ 1\ \bar{1}\,]_{CS2}$	B (= 105)
$0\ 1\ \bar{1}\ 0\ \bar{1}\ \bar{1}\ 1\ 0$	S = 22
$1\ 0\ 0\ 0\ 0\ 1\ 0\ \bar{1}$	C = 131
$1\ 0\ 1\ \bar{1}\ 0\ 0\ \bar{1}\ 0\ 0$	Z = 284 = 179 + 105 = 22 + 2x131

Comme nous travaillons avec des chiffres dans $\{\bar{1}, 0, 1\}$, chaque chiffre est codé sur deux bits (le premier pour le bit positif et le deuxième pour le bit négatif). Lorsque nous additionnons ces deux bits en tenant compte de leur signe, nous récupérons une valeur en nombre redondant), d'où le codage adopté :

1 est codé 10
0 est codé 00 ou 11
$\bar{1}$ est codé 01

Un exemple de circuit pour illustrer l'intérêt du système redondant pour le calcul en grande précision est présenté dans le paragraphe suivant.

I.I.3. Etude d'un exemple d'opérateur de calcul en précision illimitée : circuits OCAPI

Il s'agit du projet d'un opérateur de calcul d'addition, de division, de soustraction et de multiplication en grande précision. Il consiste à regrouper jusqu'à 16 boîtiers OCAPI sur une carte, contrôlée par un transputer comme l'indique la figure I.1.

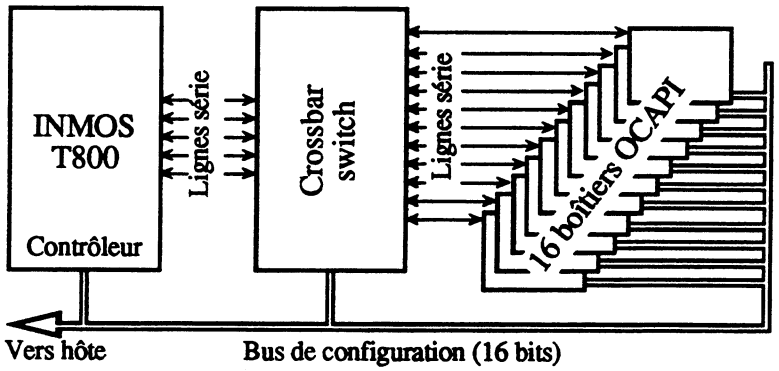


Figure I.1 : carte OCAPI

Avec les opérateurs en ligne, nous pouvons exécuter simultanément des opérations dépendantes qui, avec des opérateurs ordinaires, devraient s'exécuter séquentiellement. Notons que deux opérations ne sont dépendantes que si le résultat de l'une est opérande de l'autre.

Un boîtier OCAPI est en fait un tableau de 2048 opérateurs élémentaires configurables pour l'addition, la soustraction, la multiplication et la division. La figure I.2 représente l'organisation interne d'un tel boîtier, chaque circuit OCAPI contenant une vaste partie opérative de 2048 bits avec un décaleur et un additionneur/soustracteur, ainsi qu'une partie contrôle spécifique aux opérations.

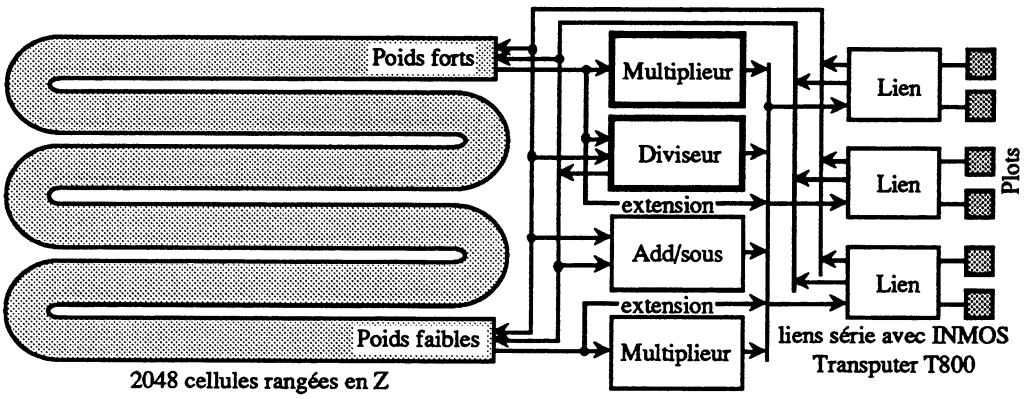


Figure I.2 : organisation interne du boîtier OCAPI

Cet opérateur a été conçu pour pouvoir travailler avec des nombres redondantes en base deux et transmises en série dans les deux sens, poids forts ou poids faibles en tête [GUK 91]. La possibilité de travailler poids forts en tête est nécessaire au processus de division, car le résultat d'une division ne s'obtient que poids fort en tête. Le temps de calcul est en général le temps de transmission du résultat, ou du plus grand résultat intermédiaire.

Pour mieux illustrer le fonctionnement de ce boîtier, nous proposons le schéma (figure I.3) de calcul de la somme de deux rationnels :

$$\frac{A}{A'} + \frac{B}{B'} = \frac{(A*B' + B*A')}{A'*B'}$$

Le résultat est donné en rationnel pour éviter les quotients infinis (tels que 2/3). Nous notons C la somme A*B´ + B*A´et C´ le produit A´*B´. A gauche sont présentés A, A´, B, B´, C et C´ dans la mémoire du transputer, au centre le crossbar et à droite les boîtiers OCAPI.

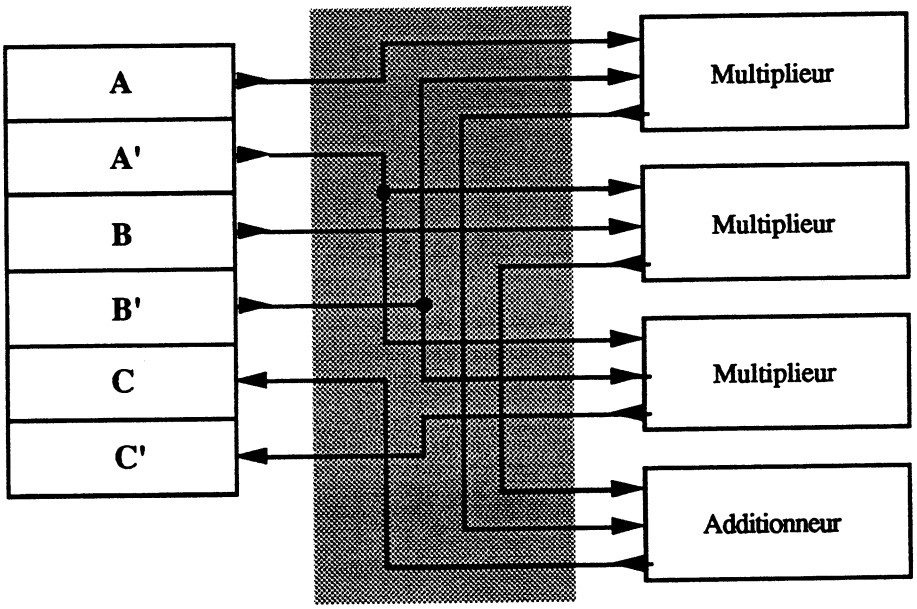


Figure I.3 : schéma de calcul de la somme A/A´ + B/B´ avec les boîtiers OCAPI

Les variables A, A´, B et B´ sont lues chiffre à chiffre dans la mémoire du transputer, qui reçoit le résultat en même temps. Les résultats intermédiaires A*B´, B*A´, A*B´ + B*A´ et A´*B´ sortent d'un boîtier OCAPI pour entrer dans un autre. Le flot de données est régulé par les demandes des circuits, un boîtier cessant ses calculs lorsque le résultat n'est pas demandé ou qu'un, au moins, des opérandes demandés n'est pas disponible. La transmission des demandes n'étant pas instantanée et les chiffres étant transmis par petits paquets, chaque boîtier dispose d'une petite file d'attente des opérandes et du résultat.

Ceci pourrait nous faire venir une idée à l'esprit, celle de se demander si nous ne pouvons pas utiliser les boîtiers OCAPI déjà réalisés pour calculer le PGCD de deux grands entiers et ainsi éviter la conception d'un nouveau circuit.

L'opération de modulo, ou reste de la division entière du plus grand **PG** par le plus petit **pp**, utilisée pour le calcul du PGCD n'est pas exécutable en ligne. En effet, nous connaissons le reste de la division, lorsque sont connus tous les chiffres à la fois du diviseur et du dividende, et que par conséquent le quotient est connu. Les algorithmes de calcul du PGCD que nous allons exposer sont itératifs (EUCLIDE, STEIN, ...) et par suite, les sorties de l'itération i - 1 seront les entrées de l'itération i. Il suffit donc d'imaginer le temps pendant lequel nous attendons à chaque itération de l'algorithme pour pouvoir enfin réitérer le processus avec des nombres de taille 2048 bits !!! Donc, le calcul du PGCD de deux très grands nombres demande soit un très grand nombre de boîtiers OCAPI, soit de nombreux stockages et transferts des résultats intermédiaires (figure I.4) et par suite, nous ne pouvons pas tirer avantage du parallélisme de la carte.

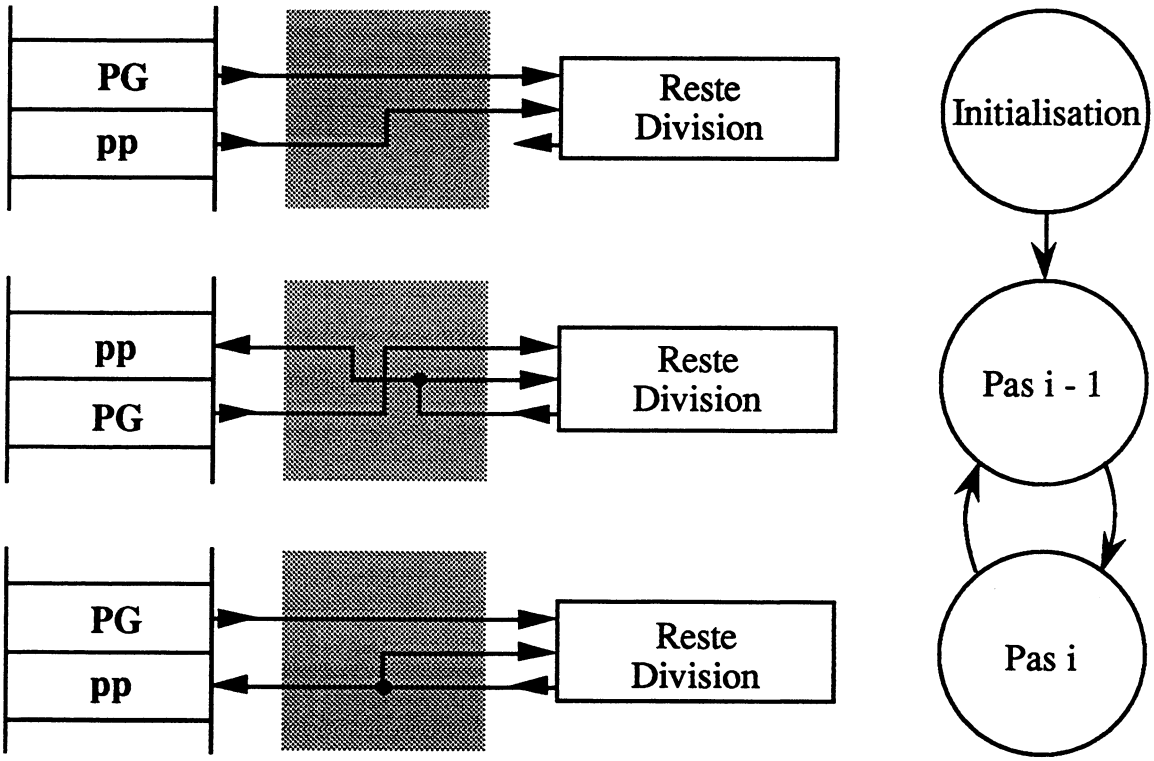


Figure I.4 : OCAPI est inefficace pour le calcul du PGCD

Dans le paragraphe suivant, nous présentons la démarche générale de conception de circuits VLSI adoptée en particulier au deux circuits de calcul du PGCD et de la distance euclidienne.

I.2. CONCEPTION DE CIRCUITS VLSI

La conception des circuits VLSI a pour but la **génération** des masques. Elle couvre deux domaines : **logique** et **topologique** (figure I.5).

La conception logique consiste à traduire en un schéma logique une spécification fonctionnelle donnée pour un circuit.

La conception topologique, consiste à concevoir l'image de l'implantation physique d'un schéma logique sur la pastille de silicium.

L'une des étapes, importante dans le cycle de conception, est la construction du **plan de masse** d'un circuit. Le plan de masse reflète l'implantation physique des blocs qui constituent un circuit.

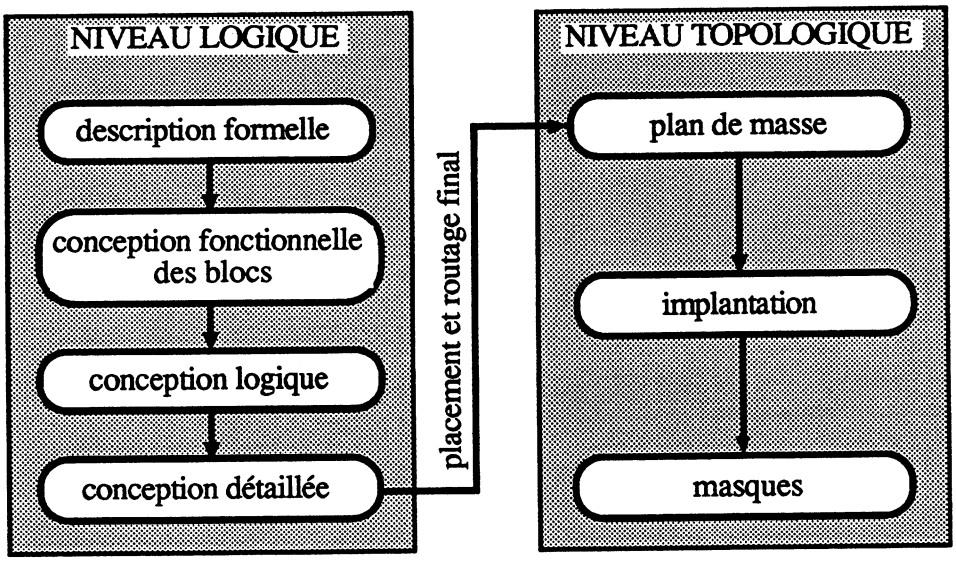


Figure I.5 : schéma de conception conventionnel

La démarche de conception peut être divisée en trois étapes : approche descendante ou décomposition, implantation et approche ascendante ou composition (figure I.6).

Approche descendante

C'est une démarche cartésienne de découpage d'un bloc exécutant une fonction en un ensemble de blocs exécutant chacun une fonction plus simple, mais coopérants pour exécuter la fonction précédente, et d'itérations du découpage jusqu'à ce que les fonctions soient suffisamment simples pour que nous puissons en entreprendre l'implantation. Au cours de cette étape est fait un choix de style qui conditionne :

- le nombre de transistors requis pour exécuter la fonction,
- la densité (transistors par surface) prévisible,
- la déformabilité d'un bloc,
- la facilité d'interconnexions,
- l'effort de dessin.

Pour connaître tous les termes compromis, il est nécessaire de connaître au plus tôt le plan de masse du circuit.

Implantation

Elle consiste à passer d'une représentation variable de la fonction de chaque bloc (schéma logique, schéma à transistors, contenu binaire d'un PLA ou d'une ROM, etc ...) à une représentation unique des masques pour tout le circuit (dessin au micron, au lambda, stick diagramme, ...) et ceci pour chaque bloc.

Approche ascendante

Elle consiste à passer de la représentation du masque de chaque bloc à la représentation des masques du circuit complet.

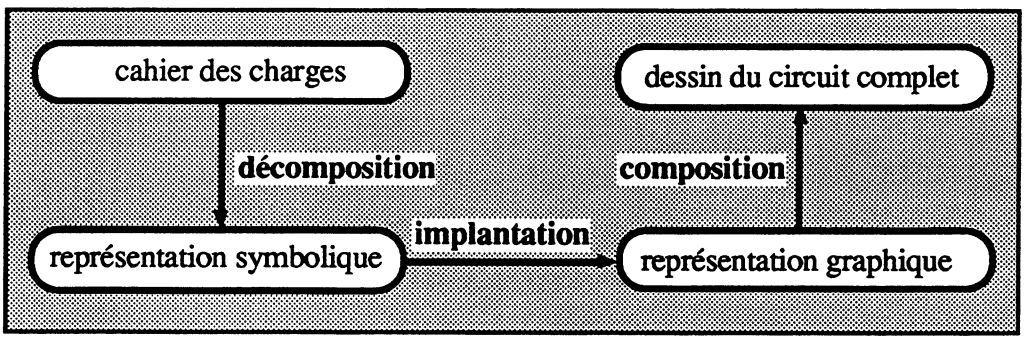


Figure I.6 : démarche de conception

Ces trois étapes, idéalement séquentielles, sont polluées par de fréquents retours en arrière ou au contraire des avances prématurées. En effet, l'implantation d'un bloc (c'est à dire son dessin) peut remettre en cause sa fonction (respectivement sa forme) et par effet de domino celle de tous les blocs qui lui sont connectés (respectivement les blocs voisins).

Afin d'anticiper le problème de composition, il est nécessaire d'estimer la taille et la forme de chaque bloc. Ces estimations sont successivement raffinées.

Cette démarche de conception permet d'assurer, avant de commencer la conception détaillée d'un niveau plus bas dans la hiérarchie, une bonne estimation de l'exigence (taille, surface et position) de module supérieur.

Le résultat d'estimation devient à la fois un guide et une contrainte géométrique pour l'implantation et la composition finale.

I.3. CONCLUSION

Les algorithmes de calcul en ligne fournissent un chiffre du résultat pour chaque nouveau chiffre des opérandes. Certains autres algorithmes ont besoin de connaître tous les chiffres des opérandes avant de commencer à calculer. Enfin, les algorithmes intermédiaires mènent le calcul pour chaque nouveau chiffre fourni, mais ne produisent de résultat que lorsque tous les chiffres des opérandes sont présents. EUCLIDE est un circuit de ce genre.

Le circuit EUCLIDE sera alors conçu autrement que le circuit OCAPI, nous gardons néanmoins la même structure interne des données, c'est à dire la forme en Z (serpentin) des 2048 cellules. Il en va de même pour le circuit de calcul de la distance euclidienne. Dans les deux cas, il s'agit de circuits spécialisés qui appliquent les principes de l'arithmétique en ligne pour fonctionner en précision infinie. Le schéma en blocs de ces deux circuits est montré dans la figure I.7.

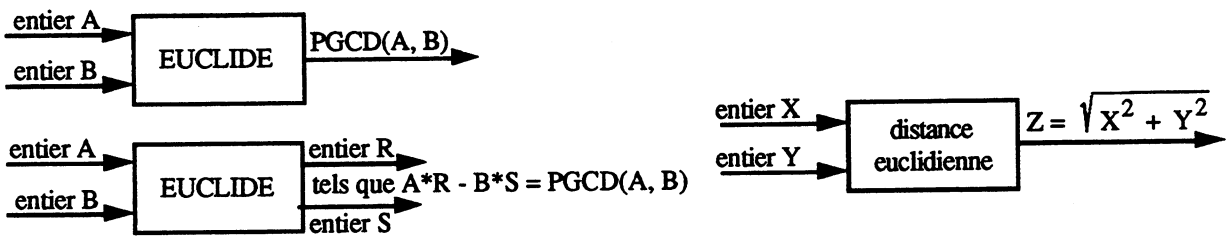


Figure I.7 : vue externe des circuits EUCLIDE et distance euclidienne

Pour chacun des circuits, nous utilisons le schéma conventionnel de conception montré en figure I.5. Les détails de cette démarche feront l'objet des chapitres suivants.

CHAPITRE II

PROJET EUCLIDE

II.1. INTRODUCTION

II.2. ETUDE DES ALGORITHMES EXISTANTS POUR LE CALCUL DU PGCD DE DEUX PETITS NOMBRES

II.3. ADDITION ET COMPARAISON DE TRES GRANDS NOMBRES

II.4. CONCLUSION

II.1. INTRODUCTION

Le calcul du PGCD est la pierre angulaire du calcul formel où les calculs sont faits avec des rationnels. Pour diminuer à la fois l'encombrement en mémoire de très grands nombres et le temps de calcul, et surtout pouvoir tester l'égalité de deux rationnels, un algorithme de simplification est déclenché périodiquement, lui-même gros consommateur de temps de calcul : le Plus Grand Commun Diviseur.

Le calcul du PGCD intervient, en effet, directement dans la réduction des fractions en précision infinie mais aussi dans les calculs modulaires dans les corps finis ($\mathbb{Z}/p\mathbb{Z}$, p premier). En effet, le calcul de l'inverse d'un élément x de $\mathbb{Z}/p\mathbb{Z}$est obtenu par le calcul des coefficients de Bezout associés au calcul du PGCD de x et p. Ce dernier type de calcul est notamment utilisé dans de nombreux algorithmes probabilistes, qui s'intéressent à donner les résultats d'un problème avec une probabilité d'échec non nulle [KAR 87, AHO 90]. Les calculs dans des corps finis sont aussi utilisés pour des problèmes dans les entiers (ou les rationnels), lorsque la solution cherchée est bornée en fonction des entrées : le calcul est effectué alors en parallèle modulo plusieurs p_i (premiers entre eux ou premiers). La remontée vers la solution exacte peut être effectuée par le théorème des restes chinois. Cette technique fournit des algorithmes performants pour de nombreux problèmes, comme par exemple l'inversion de matrice dans $\mathbb{Q}$ [VIL 88]. Ce problème est connu sous le nom de "calcul du PGCD étendu de deux entiers" et l'équation vérifiée par A, B, R et S est dite "identité de Bezout". Soient A et B deux entiers donnés : il existe R et S tel que A*R - B*S = PGCD(A, B). De plus, il existe un couple unique (R, S) tel que R < B et S < A [ROC 89].

Signalons que l'instabilité de nombreux problèmes, dont la plupart des calculs de valeurs propres, oblige à se tourner vers les calculs exacts. Pour exemple trivial, la moindre perturbation d'une matrice non inversible peut la rendre inversible, car dans tout voisinage d'une matrice non inversible, il existe une matrice inversible. Les problèmes liés aux erreurs d'arrondi deviennent critiques lorsque nous désirons calculer, par exemple, la multiplicité d'une valeur propre, et par suite la dimension de l'espace propre associé : ce problème intervient pourtant de manière directe dans la résolution d'équations différentielles [HIS 74].

Une autre remarque importante qu'il ne faut pas oublier est que, lors des calculs exacts, 80% du temps est passé à calculer des PGCD [DAR 85] ; l'impossiblité sous MACSYMA (système de calcul formel utilisant la précision infinie) de calculer la forme d'Hermite d'une matrice carrée d'ordre 6 à coefficients polynômiaux de degré trois est significative, ce problème ayant pourtant des applications importantes en automatique [SIE 89].

Il s'avère donc essentiel d'accélérer les calculs de base : la motivation du circuit EUCLIDE est de fournir un outil matériel adapté au calcul rapide en précision infinie et en multiprécision, l'implantation du circuit EUCLIDE est basée sur le choix d'un bon compromis entre performance et coût en matériel.

EUCLIDE sera, avec ses avantages, un support matériel essentiel au même titre que le sont les unités vectorielles pipelinées pour les calculs d'algèbre linéaire en flottant et permettra ainsi le traitement de certains problèmes de l'ingénieur [DEL 88] : formes normales de matrices denses à coefficients entiers, ou polynômiaux [SIE 89], bases de Gröbner ou bases standard [SEN 90] et certains problèmes d'équations différentielles [HIS 74]...

Le but du projet EUCLIDE sera d'évaluer la complexité de circuits intégrés réalisant tout ou partie des calculs de PGCD et des coefficients de Bezout sur des grands entiers et de déterminer laquelle des différentes méthodes pour l'implantation est la meilleure du point de vue performance et coût en matériel.

II.2. ETUDE DES ALGORITHMES EXISTANTS POUR LE CALCUL DU PGCD DE DEUX PETITS NOMBRES

L'algorithme qu'EUCLIDE proposa au III^{ème} siècle avant Jésus-Christ était vraisemblablement connu 200 ans plus tôt [KNU 81]. En 1967, J. STEIN [STE 67] dériva une version de cet algorithme en arithmétique binaire sans division. Cette idée est au centre de l'algorithme de BRENT et KUNG que nous détaillerons plus tard. D.H. LEHMER [LEH 38] proposa en 1938 une méthode où seuls les premiers chiffres de poids forts étaient testés, comme pour la division d'OCAPI, rendant ainsi possible le calcul du PGCD sur de très grands nombres. La méthode de LEHMER a permis à SCHONHAGE de construire en 1971 le meilleur algorithme séquentiel aujourd'hui connu. L'implantation de ces algorithmes et leur comparaison sont discutées dans [ROC 89].

"Qu'est ce que le PGCD de deux entiers"

Une réponse à cette question pourrait nous faciliter le suivi des algorithmes de calcul du Plus Grand Commun Diviseur (PGCD) de deux entiers, que nous présentons au cours de cette section. Soient deux entiers U et V, non nuls, $PGCD\ (U, V)$ est le plus grand entier qui divise à la fois U *et* V. Cette définition a un sens puisque si $U \neq 0$ alors aucun entier plus grand que $|U|$ ne pourrait diviser U. D'autre part, l'entier 1 divise U et V, il est donc raisonnable de parler de l'existence d'un plus grand entier qui divise U et V. Par contre, si U et V sont tous les deux

égaux à zéro, et comme tout entier divise 0, la règle précédente n'est plus valable. Par convention, PGCD(0, 0) = 0. La définition du PGCD de deux entiers implique les trois propriétés suivantes :

(1) $PGCD(U, V) = PGCD(V, U)$
(2) $PGCD(U, V) = PGCD(-U, V)$
(3) $PGCD(U, 0) = U$

Une première méthode de calcul du PGCD de deux entiers est déduite du théorème fondamental de l'arithmétique : quel que soit l'entier positif U, il peut s'écrire de manière unique sous la forme :

$$U = \prod_{\text{p premier}} p^{u_p}$$

où les exposants u_p sont définis comme entiers non négatifs et un nombre fini de ces exposants est nul, p est la suite de tous les nombres premiers.

Alors, si U et V sont deux entiers positifs ainsi dècomposés en facteurs premiers, et en se basant sur les propriétés (1), (2) et (3), nous déduisons que :

$$\text{PGCD}(U, V) = \prod_{\text{p premier}} p^{\min(u_p, v_p)} \quad \text{où} \quad U = \prod_{\text{p premier}} p^{u_p} \quad \text{et} \quad V = \prod_{\text{p premier}} p^{v_p}$$

Cette méthode théorique n'a aucun intérêt pratique puisqu'elle nécessite à chaque fois de factoriser les deux nombres U et V, et qu'il n'existe pas de méthode rapide pour le faire. Néanmoins, elle prouve clairement que le PGCD est défini dans un anneau factoriel, même si il n'est pas euclidien (i.e possède une division euclidienne). Heureusement, une deuxième méthode, découverte au troisième siècle avant Jésus Christ a survécu jusqu'à nos jours : c'est l'algorithme que nous appelons "grand-père" de tous les autres, celui d'EUCLIDE. En remplaçant les mots qu'il avait utilisés par ceux des mathématiques modernes, le paragraphe suivant résume ce qu'EUCLIDE a écrit :

" *Soit deux entiers positifs, trouver leur Plus Grand Commun Diviseur?*

Soit A et B ces deux entiers. Si B divise A, alors B est le PGCD(A, B) puisqu'il se divise lui même, et c'est le plus grand diviseur puisqu'il n'existe pas un entier plus grand que B qui

divise B. Mais si B ne divise pas A, alors il faut itérer l'opération de soustraction du plus petit des deux nombres du plus grand jusqu'à obtenir deux nombres tels que l'un est un multiple de l'autre. Maintenant, désignons par E le reste de la division de A par B (A mod B), par F le reste de la division de B par E que l'on suppose diviseur de E. F divise E, E divise B - F, F divise aussi B - F et F divise F donc, F divise B. D'autre part, B divise A - E, donc F aussi, or F divise E, donc F divise A.
conclusion : F est un diviseur commun de A et B. Est ce qu'il est le plus grand?

Supposons que non, alors il existerait un G, plus grand que F, qui divise A et B. Nous obtenons : B divise A - E, alors G divise A - E, G divise A, donc G divise E. D'autre part, E divise B - F, donc G aussi, or G divise B, donc G divise F.
conclusion : G > F et G divise F, impossible !, donc F est le PGCD de A et B "

Ainsi, EUCLIDE avait suggéré deux manières pour le calcul du PGCD. La première, qui manipulait les restes de division des deux nombres, est représentée par l'algorithme suivant :

Algorithme d'EUCLIDE
fonction PGCD (A, B) ; (* on suppose que A ≥ B > 0 *)
début
    tant que B ≠ 0 **faire**
        $\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} B \\ A \bmod B \end{pmatrix}$
    PGCD := A ;
fin (* PGCD *) .

Cet algorithme utilise les deux règles de réécriture :

a) PGCD(A, 0) = A	car 0 est multiple de tout entier
b) PGCD(A, B) = PGCD(B, A mod B)	reste de la division entière de A par B

Et comme A mod B est plus petit que A et B, l'algorithme ci-dessus converge ; il part du couple $(A_0, B_0) = (A, B)$ pour converger vers $(A_n, B_n) = (PGCD(A, B), 0)$. Son principal inconvénient est l'utilisation de l'opération de modulo qui est très coûteuse, surtout avec les grands nombres.

Nous ne négligeons pas la deuxième méthode d'EUCLIDE [KNU 81], celle qui itère la soustraction du plus petit du plus grand des deux nombres et qui exprime les divisions sous forme de soustractions ou décalages qui sont beaucoup moins coûteux et que nous savons

réaliser avec du matériel. Malheureusement, cet algorithme demande bien plus de pas que le précédent. L'algorithme d'EUCLIDE devient :

```
fonction PGCD (A, B) ; (* on suppose A ou B impair, A ≥ B, 2^n ≤ A < 2^(n+1) *)
début
    tant que B ≠ 0 faire
        si B < 2^n alors      (A, B) := (A, B*2)
        sinon si A ≥ B alors  (A, B) := (B, (A-B)*2)    (* 2^n ≤ B ≤ A < 2^(n+1) *)
              sinon           (A, B) := (A, (B-A)*2) ;
    tant que A mod 2 = 0 faire A := A div 2; PGCD := A ;
fin (* PGCD *) .
```

Plus récemment, l'algorithme binaire [STE 67] a également permis le calcul du PGCD sans division, uniquement avec des soustractions et des décalages dans la représentation en base deux des entiers. Il est basé sur cinq propriétés qui sont vérifiées d'une manière évidente :

PGCD(U, 0) = PGCD(U)
PGCD(U, V) = PGCD(U - V, V)
PGCD(U, V) = 2 * PGCD(U / 2, V / 2) si U ≡ V ≡ 0 [2]
PGCD(U, V) = PGCD(U / 2, V) si U ≡ V+1 ≡ 0 [2]
PGCD(U, V) = PGCD(U, V / 2) si U+1 ≡ V ≡ 0 [2]

L'algorithme de STEIN montré ci-dessous débute par les poids faibles.

```
fonction PGCD (A, B) ; (* on suppose A impair *)
début
    tant que B ≠ 0 faire
        si B mod 2 = 0 alors      (A, B) := (A, B div 2)      (* B est pair *)
        sinon si A ≥ B alors      (A, B) := (B, (A-B) div 2)  (* A et B sont impairs *)
              sinon               (A, B) := (A, (B-A) div 2);
    PGCD := A ;
fin (* PGCD *)
```

En comparant les algorithmes d'EUCLIDE et de STEIN, qui sont de même ordre, nous pouvons noter une seule différence entre les deux qui se manifeste dans la manière d'examiner les opérandes. En effet EUCLIDE teste les bits de poids fort ($B < 2^n$) alors que STEIN s'intéresse aux bits de poids faible en premier lieu (Bmod2). Bmod2 s'obtient par le bit de poids faible de B.

II.3. ADDITION ET COMPARAISON DE TRES GRANDS NOMBRES

Comme dans OCAPI, nous nous intéressons à des nombres de 600 chiffres décimaux, c'est-à-dire 2048 bits. La multiplication ou la division de ces nombres par deux ne pose pas de problème, par contre l'addition/soustraction ne peut se faire qu'en série ou dans un système redondant pour éviter un temps de propagation de retenue prohibitif.

Nous allons explorer ces deux approches : série et redondant. De toute façon dans ces deux cas, la comparaison de deux entiers (variable ou constante) se ramène à une propagation de retenue puisque ceci consiste à ramener le signe à un emplacement bien défini pour pouvoir le tester. Cette opération serait alors longue et doit être évitée. Nous la remplaçons toujours par des prédicteurs (ou estimateurs) plus ou moins fidèles, telle qu'une approximation du nombre de chiffres des entiers, complétée par quelques chiffres poids forts.

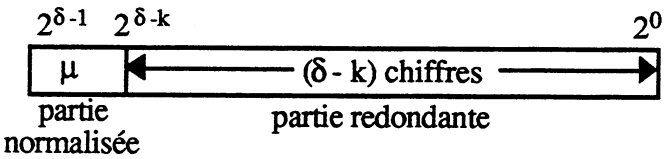
Dans OCAPI, la connaissance exacte du nombre de chiffres du diviseur et du reste partiel associée à la connaissance de quelques chiffres de poids forts de ces nombres, (quatre en l'occurrence), assure la convergence de l'algorithme de division.

Naturellement, les algorithmes précités sont à adapter aux prédicteurs et à leur manque de fidélité, néanmoins ils permettent la manipulation des grands nombres, coûteuse, par celle de petits estimateurs $b_0 \neq 0$ ou B mod $2 \neq 0$ indiquent que B est impair et, en notation ordinaire $b_n = 1$ indique que $B \geq 2^n$.

Nous notons $b_{n...m}$ la chaîne b_n, b_{n+1}, ..., b_m ($n > m$). "B div 2", "B div 4", et "B * 2" ne sont que des notations commodes pour B décalé d'une position, de deux positions vers les poids faibles ou d'une position vers les poids forts, et ne sous-entendent ni multiplication ni division. De même "A * B $\geq$ 0" indique simplement que A et B sont de même signe. L'algorithme de STEIN modifié utilisant les estimateurs se présente comme suit :

```
fonction PGCD (A, B, δa, δb) ;
(* on suppose A impair, δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombres de bits de A et de B) *)
début
  tant que δb > 0 faire
  début
    si B mod 2 = 0 alors                    (* B pair et A impair *)
    début δb := δb - 1 ;
      (A)   (  A      )
      (B) := ( B div 2 )                    (* décaler B *)
    fin
    sinon                                   (* A et B sont tous deux impairs *)
      si δa > δb alors                      (* A > B *)
        début échange (δa, δb) ;            (* échanger les prédicteurs *)
          (A)   (     B      )
          (B) := ( (A+B) div 2 )            (* addition *)
        fin
      sinon
        début δb := δb - 1 ;
          (A)   (     A      )
          (B) := ( (B-A) div 2 )            (* soustraction *)
        fin ;
  fin
  si A ≥ 0 alors PGCD := A sinon PGCD := - A ;
fin (* PGCD *) .
```

Quant à l'algorithme d'EUCLIDE, son adaptation aux grands nombres ne se contente pas des deux estimateurs, taille de A (δ_a) et taille de B (δ_b), mais examine aussi les k premiers bits de poids fort des nombres appellés tête de A (μ_a) et tête de B (μ_b). L'algorithme d'EUCLIDE traitant les grands nombres utilise la représentation redondante des nombres, un nombre a la structure suivante :



La partie normalisée composée de k bits et un bit de signe (notation en complément à deux) de poids fort, représente la tête du nombre. Dans notre système redondant, le changement de signe d'un nombre devient tout simplement un échange des deux bits de chaque chiffre du nombre. b_n, le bit de signe le plus significatif de B, prend la valeur zéro si $|B| < 2^k$. La

normalisation force le premier bit du poids fort de la tête à 1. L'algorithme d'EUCLIDE utilisant les estimateurs et la tête devient alors :

```
fonction PGCD (A, B, k, δa, δb) ;
(* δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombres de bits de A et de B) *)
début
  tant que δb > 0 faire    (* B ≠ 0 *)
  début
    si abs (μb) < 2^(k-1) alors
    début
      si ((δa > δb) ou ((δa = δb) et (abs (μa) > 2^(k-1) ))) et ( bn * μb) = 2^(k-1) - 1 alors
      début μb := μb + bn ; bn := - bn fin
      sinon début μb:= μb + μb + bn ; bn..1:= bn-1..0 ; δb:= δb - 1 fin (* normalisé *)
    fin
    sinon début
      si (δa > δb) ou ( (δa = δb) et (abs(μa) > abs (μb) ) ) alors    (* A > B *)
      début échange (δa, δb) ;
      si ( μa * μb ) < 0 alors (A/B) := ( B / (A+B) *2 ) sinon (A/B) := ( B / (B-A) *2 ) fin
      sinon si ( μa * μb ) < 0 alors (A/B) := ( A / (A+B) *2 ) sinon (A/B) := ( A / (B-A) *2 );
    fin;
    δb := δb - 1 ;
  fin ;
  PGCD := A ;
fin.
```

II.4. CONCLUSION

A la lumière de ce qui précède, et en se basant sur le fait que les opérations d'addition et de soustraction de deux grands nombres ne se font qu'en ligne (approche série) ou en redondant sans propagation de retenue (approche parallèle), nous veillons à varier ces deux méthodes en procédant par plusieurs tactiques qui donnent naissance à plusieurs approches que nous résumons en :

Approche parallèle avec les poids faibles en tête
Approche parallèle avec les poids forts en tête
Approche série avec les poids faibles en tête
Approche série avec les poids forts en tête

Pour chacune de ces approches, nous avons à évaluer la complexité du circuit correspondant qui est de la forme montrée en figure II.1, pour l'approche parallèle, puisque nous ne pouvons

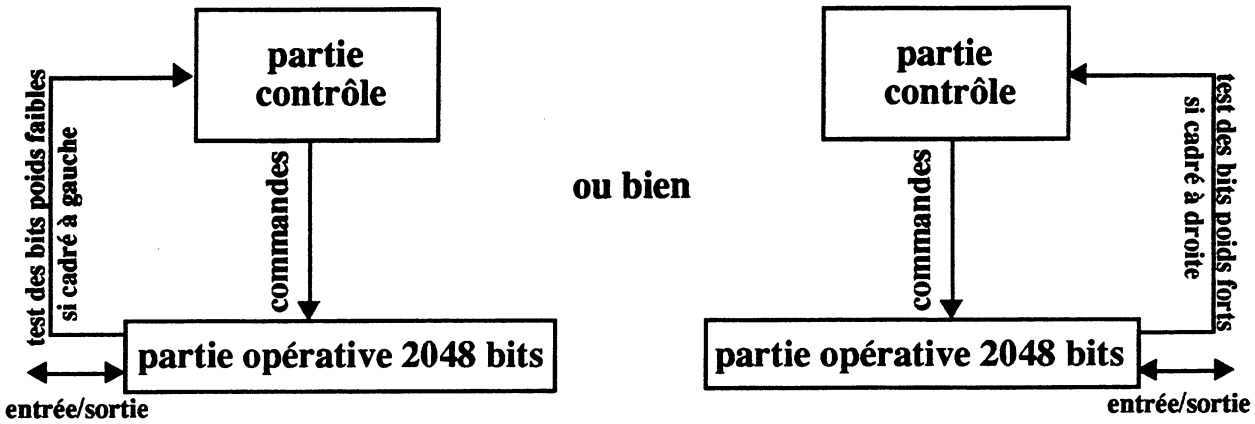


Figure II.1 : choix pour un circuit parallèle

pas connaître à la fois, les bits de poids faible et de poids fort d'un très grand nombre, ou de la forme de la figure II.2 pour l'approche série.

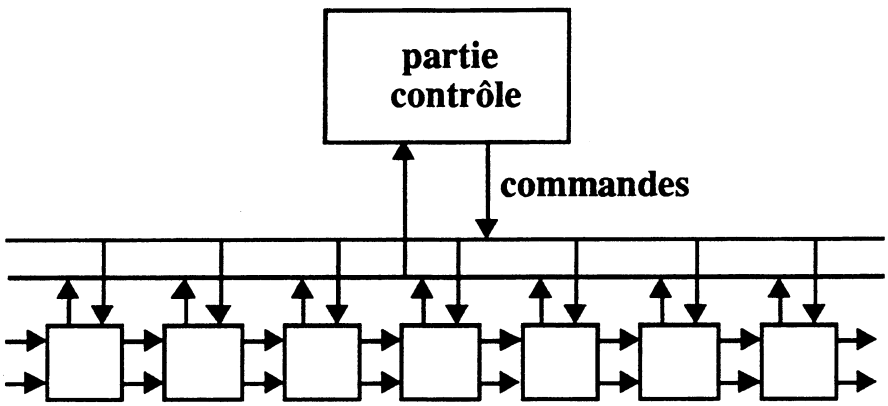


Figure II.2 : choix pour un circuit série

Quant au calcul des coefficients de Bezout R et S, il nous suffit d'exécuter en sens inverse les opérations de l'algorithme déjà utilisé pour le calcul du PGCD comme nous allons voir en détail plus loin.

Pour l'approche parallèle, nous avons besoin d'une pile LIFO pour mémoriser les opérations déjà exécutées sur les deux nombres A et B. Le processus est résumé par le schéma montré dans la figure II.3.

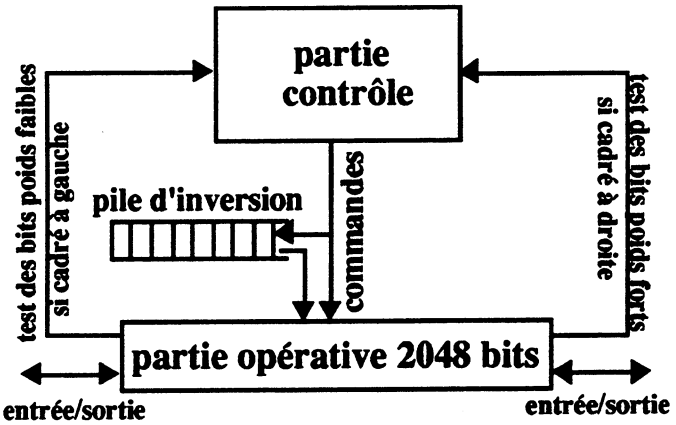


Figure II.3 : principes du circuit EUCLIDE pour le calcul du PGCD et des coefficients de Bezout R et S

Par contre, en série, nous essayons de ne pas utiliser de pile et de nous contenter du vecteur systolique linéaire déjà programmé pour le calcul du PGCD.

CHAPITRE III

ALGORITHMES

III.1. INTRODUCTION

III.2. ALGORITHMES POUR LES TRES GRANDS NOMBRES

III.3. EXPERIMENTATION DES ALGORITHMES

III.4. ALGORITHMES AVEC TEST DES POIDS FAIBLES

III.4.1. Approche parallèle
III.4.2. Approche série

III.5. ALGORITHMES AVEC TEST DES POIDS FORTS

III.6. CONCLUSION

III.1. INTRODUCTION

Rappelons que le but de ce projet est de concevoir un circuit pour le calcul du PGCD de deux grands nombres dont l'implantation vise un bon compromis vitesse/surface de silicium. Pour cela, nous allons considérer plusieurs variantes plus ou moins performantes de chaque algorithme, les simuler, prendre des mesures, évaluer la complexité et les comparer afin de choisir la meilleure.

C'est la méthode de conception que nous utilisons pour déduire la meilleure tactique pour le calcul du PGCD de deux grands nombres. Quatre exemples avec test des poids forts et cinq avec test des poids faibles sont présentés au cours de ce chapitre.

III.2. ALGORITHMES POUR LES TRES GRANDS NOMBRES

Si les affectations dans les algorithmes présentés dans le chapitre II §II.3 sont écrites :

$$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} B \\ (B - q*A)*2^p \end{pmatrix}$$

alors les opérations dans les algorithmes d'EUCLIDE et de STEIN sont des cas particuliers avec $q \in \{0, 1\}$ et $p \in \{-1, +1\}$. En ajustant q et p à d'autres valeurs, il a été possible de concevoir de nouveaux algorithmes. Dans ce chapitre nous nous limitons aux domaines $q \in \{-2, -1, 0, 1, 2, 4\}$ et $p \in \{-3, -2, -1, 1, 2\}$ car les domaines plus grands conduisent à des réalisations inutilement complexes.

Ces algorithmes sont basés sur la multiplication/division par une puissance de deux, sur l'addition/soustraction et sur des tests. Les très grands nombres entiers, de taille 2048 bits peuvent être additionnés sans propagation de retenue soit en mode série, soit en utilisant la notation redondante.

Le coût en transistors d'un décalage est approximativement dix à vingt fois moins que celui d'une addition. Nous présentons donc les algorithmes avec seulement une addition/soustraction par cycle machine et différents décalages.

Pour les tests, $b_0 \neq 0$ indique que B est impair, et en notation normale ou en notation redondante normalisée $b_n \neq 0$ indique que $|B| \geq 2^n$. $B \neq 0$ et $A \geq B$ ne peuvent être testés sans effectuer de propagation de retenue et sont remplacés par des estimateurs.

III.3. EXPERIMENTATION DES ALGORITHMES

Plusieurs approches montrées dans la figure III.1 ont été simulées sur le même échantillon de 200 entiers de 2000 bits [BGR 90, CHE 90].

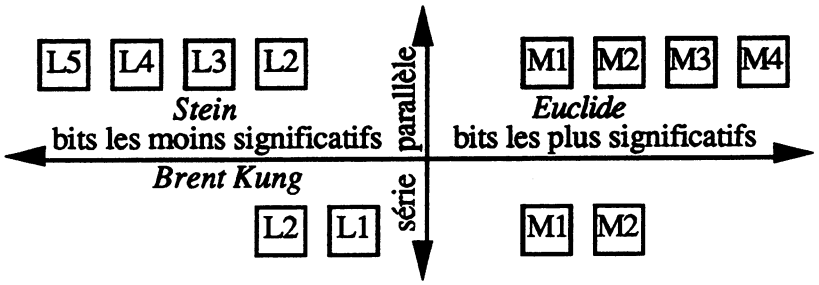


Figure III.1: différentes approches de simulations

Dans ce chapitre nous présentons un algorithme de calcul du PGCD proposé par STEIN [STE 67, KNU 81], dans lequel nous gagnons deux bits de poids faible pour la plupart des cycles d'horloges, et un seul le reste du temps. La variante L3 peut être facilement modifiée pour gagner deux bits de poids faible à chaque cycle d'horloge (variante L4), ou trois bits de poids faible pour la plupart des cycles d'horloges et deux bits pour le reste (variante L5) avec une addition ou une soustraction par cycle.

BRENT et KUNG [BKU 83] ont développé un algorithme systolique (variante notée L2), basé sur les travaux de STEIN, qui remplace les comparaisons des nombres par des comparaisons des estimateurs (nombre de bits) et proposent une implantation VLSI [BKU 83]. Le remplacement du contrôle distribué de la version systolique par une autre semi-centralisée aboutit à une implantation avec un très faible nombre de transistors par bit. Mais, les modifications de cette approche série pour le calcul du PGCD étendu sont compliquées.

PURDY [KNU 81, PUR 83] a proposé un algorithme parallèle (variante L2) sans propagation de retenue utilisant des estimateurs de Brent qui fournit un nombre impair en représentation chiffres signés dont le chiffre de poids faible est égal à 1 (ou $\bar{1}$).

YUN et ZHANG [YUZ 86] ont étendu l'algorithme (variante L3) pour le calcul des coefficients de Bezout. Nous avons pu le modifier dans ce chapitre pour ajuster en une seule fois R et S au lieu de le faire à chaque cycle d'horloge, ce qui empêche de calculer A et B dans le sens inverse et économise aussi bien le matériel (transistors) que les opérations (transistors et/ou cycles d'horloge). Mais l'approche de STEIN utilisée est loin d'être efficace pour les très grands nombres entiers pour deux raisons :

- La première est le manque de précision des estimateurs proposés par BRENT pour le calcul du PGCD, qui parfois induit en erreur le circuit sur le choix d'une opération (addition/soustraction qui produirait le résultat avec un minimum de chiffres) ou une affectation (à A ou B, lequel sera le plus grand). La perte des chiffres les plus significatifs contrebalance le gain des chiffres les moins significatifs,
- La seconde raison est le besoin de correction [KNU 81, YUZ 86] pour le calcul du PGCD étendu.

III.4. ALGORITHMES AVEC TEST DES POIDS FAIBLES

La table III.1 résume la classification des variantes de l'algorithme du PGCD avec test des poids faibles. Chaque approche est spécifiée par un domaine de p et q ainsi que par la manière de traiter les nombres, en série ou en parallèle. L'addition/soustraction et le décalage sont alors utilisés.

≠	q (quotient)	p (positions)	chiffres examinés	mode	
	-1, +1	0	aucun		
L1	-1, 0, +1	-1	b_1	série	
L2	-1, 0, +1	-1	a_1,b_1	série/parallèle	note 1
L3	-1, 0, +1	-1, -2	a_0,a_1,b_0,b_1	parallèle	note 2
L4	-1, 0, +1, +2	-2	a_0,a_1,b_0,b_1	parallèle	note 3
L5	-2,-1,0,+1,+2,+4	-2, -3	a_0,a_1,b_0,b_1,b_2	parallèle	

note 1 : Variante proposée par BRENT-KUNG [BKU 83]
note 2 : Variante de YUN-ZHANG [YUZ 86]
note 3 : Variante détaillée dans ce chapitre

Table III.1 : variantes avec examen des chiffres de poids faibles

III.4.1. Approche parallèle

Dans ce paragraphe nous examinons les variantes basées sur une approche parallèle avec test des chiffres les moins significatifs, soient L3 de STEIN, L4 et L5. En effet, les variantes L3 et L4 écrites en pseudo-Pascal sont similaires, elles diffèrent seulement d'une ligne : dans le cas $(b_0 = 0) \wedge (b_1 \neq 0)$, la variante L3 exécute B := B div 2 tandis que la variante L4 exécute B := (B + 2 * A) div 4. Nous les présentons en un seul algorithme :

```
fonction PGCD (A, B, δa, δb) ;                    (* variantes L3 et L4 *)
(* on suppose A impair, δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombres de bits de A et B) *)
début
  tant que δb > 0 faire    (* B ≠ 0 *)
  début
    Si b0 = 0 alors
    début
      si b1 = 0 alors début B := (B + 0) div 4; δb := δb - 2 fin     (*décalage (Déc1)*)
         sinon début B := (B + 0) div 2 ; δb := δb - 1 fin           (*VL3*) variante L3
         sinon début B := (B + 2*A) div 4 ; δb := δb - 2 fin         (*VL4*) variante L4
    fin
    sinon si  δa ≥ δb alors
    début
      échanger (δa, δb) ; (* échanger *)
      si (a0 + b0 = 0 et a1+ b1 = 0) ou (a0 + b0 ≠ 0 et abs (a1+ b1) = 1) alors
      (A/B) := (  B  )                                              (*addition puis décalage*)
               ( (A+B) div 4 )
      sinon (A/B) := (  B  )                                        (*soustraction puis décalage*)
                     ( (B-A) div 4 )
    fin
      sinon
      si (a0 + b0 = 0 et a1 + b1 = 0) ou (a0 + b0 ≠ 0 et abs (a1 + b1) = 1) alors
      (A/B) := (  A  )                                              (*addition puis décalage*)
               ( (A+B) div 4 )
      sinon (A/B) := (  A  );                                       (*soustraction puis décalage*)
                     ( (B-A) div 4 )
    δb := max (δa, δb) - 2 ;
  fin ;
  PGCD := A ;
fin.
```

Soit A_0 et B_0 deux valeurs initiales de l'algorithme. Pendant le calcul du PGCD, (A_n, B_n) converge vers $(PGCD(A_0, B_0), 0) = (H, 0)$ et, à chaque étape, une opération de la troisième colonne (calcul du PGCD) de la table III.2 est exécutée. Nous mémorisons chaque opération exécutée dans une pile LIFO.

La colonne inverse de la table III.2 donne des opérations de la valeur précédente (A_{n-1}, B_{n-1}) à partir de la valeur courante (A_n, B_n). Donc, à partir de la limite de $(A_n, B_n) = (H, 0)$ et en exécutant les séquences d'opérations qui se trouvent dans la pile LIFO, nous obtenons la valeur initiale (A_0, B_0). Par contre, en partant de $(A_n, B_n) = (1, 0)$ et en exécutant le même algorithme en sens inverse nous obtenons $(A_0 / H, B_0 / H)$.

	type	Calcul du PGCD	inverse	préserve A*R-B*S =H	préserve A*R-B*S =H*2^n	suite n
Déc1	1	B := B div 4	B := B * 4	S := S div 4	R := R * 4	n:=n+2
VL3	2	B:=B div2	B := B*2	S := S div 2	R := R * 2	n:=n+1
VL4	2	B:=(B+2*A) div 4	B := B*4-A*2	R := R - S div 2 S := S div 4	R := R*4 - S*2	n:=n+2
Echange	3	$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$	$\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$	n
Add	4	B:=(B+A)div 4	B := 4*B-A	R := R -S div4 S :=S div 4	R := R * 4 - S	n:=n+2
Soust	5	B:=(B-A)div 4	B := 4*B+A	R := R + S div4 S := S div 4	R := R * 4 + S	n:=n+2
Valeurs init.		A et B	A := 1 , B := 0	R := 1 , S := 1		n := 0

Table III.2 : opérations de calcul du PGCD et du PGCD étendu pour les variantes L3 et L4

En partant de $(R_n, S_n) = (1, 1)$, et en exécutant la cinquième colonne de la table III.2, nous obtenons (R_0, S_0) tel que :

$$A_0 * R_0 - B_0 * S_0 = H$$

Malheureusement, il n'est pas toujours possible d'exécuter les opérations de la cinquième colonne, car elles exigent parfois que S soit divisible par deux ou quatre. Nous les remplaçons par les opérations de la sixième colonne, n sera mis à jour dans la colonne suivante entraînant la relation :

$$A_0 * R_0 - B_0 * S_0 = H * 2^n$$

A_0, B_0 et H sont impairs et $H * 2^n$ est pair si $n > 0$. Mais R_0 et S_0 sont tous les deux pairs ou impairs si $n > 0$. S'ils sont pairs, nous les divisons par deux : $R_0 := R_0 / 2$, $S_0 := S_0 / 2$. sinon, $R_0 + B_0$ et $S_0 + A_0$ sont pairs et préservent la relation :

$$A_0 * (R_0 + B_0) - B_0 * (S_0 + A_0) = A_0 * R_0 - B_0 * S_0 = H * 2^n$$

Nous exécutons $R_0 := (R_0 + B_0) / 2$ ainsi que $S_0 := (S_0 + A_0) / 2$, c'est à dire que nous effectuons un décalage à droite d'une position vers les poids faibles. A chaque étape n est décrémenté ; nous repétons ceci jusqu'a ce que n = 0, à la fin nous obtenons :

$$A_0 * R_0 - B_0 * S_0 = H * 2^0 = H$$

La table III.2 est exécutée pour la variante L3, en éliminant la ligne VL4 et pour la variante L4, en éliminant la ligne VL3. Notons que dans la variante L4, n reste toujours pair. L'algorithme de calcul des coefficients de Bezout pour les variantes L3 et L4 est la suivante :

procédure PGCD étendu (R, S) ; (* variantes L3 et L4 *)
(* trouver R et S tel que A * R - B * S = PGCD(A, B) *)
R:= 1 ; S := 1; (* initialiser *)
Pour i := n **jusqu'à** 1 **par pas de** -1 **faire**
début
  cas type [i] **de**
    1: **début** R := R * 4 ; n:=n+2 **fin** ; (*Decalage 1*)
    2: **début** R := R * 2 ; n:=n+1 **fin** ; (*VL3*) *Variante L3*
    2: **début** R := R * 4 - S * 2 ; n:=n+2 **fin** ; (*VL4 *) *Variante L4*
    3: **début** $\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$ **fin** ; (* échanger *)
    4: **début** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} R*4 \\ S+R \end{pmatrix}$; n:=n+2 **fin** ; (* addition *)
    5: **début** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} R*4 \\ S-R \end{pmatrix}$; n:=n+2 **fin** ; (* soustraction *)
  fin ;
fin
tant que n > 1 **faire** (* Soient A et B les valeurs initiales A_0 et B_0 sauvegardées *)
début
  n := n - 2;

  si $r_0 = 0$ **alors début si** $r_1=0$ **alors** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} R \text{ div } 4 \\ S \text{ div } 4 \end{pmatrix}$ **sinon** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} (R+B*2) \text{ div } 4 \\ (S+A*2) \text{ div } 4 \end{pmatrix}$ **fin**

  sinon début si ($a_0 + r_0 = 0$ **et** $a_1 + r_1 = 0$) **ou** ($a_0 + r_0 \neq 0$ **et abs** ($a_1 + r_1$) = 1) **alors**

  $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} (R+B) \text{ div } 4 \\ (S+A) \text{ div } 4 \end{pmatrix}$ **sinon** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} (R-B) \text{ div } 4 \\ (S-A) \text{ div } 4 \end{pmatrix}$ **fin** ;

si n=1**alors si** $r_0 = 0$ **alors** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} R \text{ div2} \\ S \text{ div } 2 \end{pmatrix}$ **sinon** $\begin{pmatrix} R \\ S \end{pmatrix} := \begin{pmatrix} (R+B) \text{ div } 2 \\ (S+A) \text{ div } 2 \end{pmatrix}$ *inutile pour L4*
fin.

Dans la variante L5 suivante, nous traitons les trois chiffres de poids le plus faible :

- Si seul le moins significatif est nul, nous forçons les deux autres à zéro,
- Si seuls les deux les moins significatifs sont nuls, nous forçons le troisième à zéro,

et nous les supprimons en effectuant un décalage à droite de trois chiffres pour recommencer ensuite.

Forcer les chiffres de B à zéro en ajoutant ou soustrayant A, pourrait augmenter la taille de B (δ_b) d'un nombre de chiffres supérieur à celui des chiffres forcés à zéro. Ceci dépend essentiellement des signes de A et de B et de leur différence de tailles $\delta_b - \delta_a$. Les simulations ont montré qu'il valait mieux prendre deux zéros de B et échanger ensuite A et B que forcer un chiffre supplémentaire à zéro. Si $\delta_b < \delta_a + 2$, il vaut mieux en prendre trois car un décalage d'une position doit de toute façon être inclus. Pour simplifier, les sommes pondérées des trois chiffres les moins significatifs $a_2a_1a_0$ et $b_2b_1b_0$ de A et B sont appelées respectivement μ_a et

μ_b. Par exemple le booléen $(\mu_a + \mu_b) \bmod 4 = 0$ est équivalent à $(a_0 + b_0 = 0) \wedge (a_1 + b_1 = 0) \vee (a_0 + b_0 \neq 0) \wedge (\text{abs}(a_1 + b_1) = 1)$. L'algorithme de calcul du PGCD pour la variante L5 est donné ci-dessous :

```
fonction PGCD (A, B, δa, δb) ;      (* variante L5 *)
(* on suppose A et B impairs, δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombre de bits de A et de B) *)
début
  tant que δb > 0 faire     (* B ≠ 0 *)
  début
  μa := a0 + 2 * a1 + 4 * a2 ; μb := b0 + 2 * b1 + 4 * b2 ;
  si μb mod 8 = 0 alors début  B := B div 8;  δb := δb - 3 fin
  sinon si μb mod 4 = 0 alors début
     si δb ≥ δa + 2 alors début B := (B + 4*A) div 8; δb := δb - 3 fin
           sinon début B := B div 4 ; δb := δb - 2 fin fin
  sinon si μb mod 2 = 0 alors début
     si (μa + μa + μb ) mod 8 = 0
     alors  B := (B + 2 * A) div 8 sinon B := (B - 2 * A) div 8; δb := δb - 3 fin
  sinon
     si  δa ≥ δb alors début échanger (δa, δb ) ;  (* échanger *)
        si  (μa + μb ) mod 8 = 0 alors début (A, B) := (B, (A+B) div 8 ); δb := δb - 3 fin
        sinon si (μa - μb ) mod 8 = 0 alors début (A, B) := (B, (B-A) div 8 ); δb := δb - 3 fin
        sinon si (μa + μb ) mod 4 = 0 alors début (A, B) := (B, (A+B) div 4 ); δb := δb - 2 fin
        sinon  début (A, B) := (B, (B-A) div 4); δb := δb - 2 fin
     fin sinon
        si  (μa + μb ) mod 8 = 0 alors début (A, B) := (A, (A+B) div 8 ); δb := δb - 3 fin
        sinon si (μa - μb ) mod 8 = 0 alors début (A, B) := (A, (B-A) div 8 ); δb := δb - 3 fin
        sinon si (μa + μb ) mod 4 = 0 alors début (A, B) := (A, (A+B) div 4 ); δb := δb - 2 fin
        sinon  début (A, B) := (A, (B-A) div 4) ; δb := δb - 2 fin;
     fin;
  fin;
  PGCD := A;
fin PGCD;
```

De même que précédemment, la table III.3 montre les opérations qui calculent R et S tels que : $A_0 * R_0 - B_0 * S_0 = H * 2^n$. La simplification de R et S est très faible.

Calcul du PGCD	inverse	préserve $A * R - B * S = H$	préserve $A*R-B*S = H*2^n$	next n
B := B div 8	B := B * 8	S := S div 8	R := R * 8	n:=n+3
B:=B div 4	B := B * 4	S := S div 4	R := R * 4	n:=n+2
B:=B div 8 + A div2	B := 8 * B - 4 * A	R := R - S div 4 S := S div 8	R := R * 8 - 4 * S	n:=n+3
$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$	$\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$	n
B:=B div 8 + A div4	B := 8 * B - 2 * A	R := R -S div 4 S := S div 8	R := 8 * R - 2 * S	n:=n+3
B:=B div 8 - A div 4	B := 8 * B+ 2 * A	R := R + S div 4 S := S div 8	R := 8 * R + 2 * S	n:=n+3
B:= (B + A) div 8	B := 8 * B - A	R := R -S div 8 S :=S div 8	R := 8 * R - S	n:=n+3
B:= (B - A) div 8	B := 8 * B + A	R := R +S div 8 S :=S div 8	R := 8 * R + S	n:=n+3
B:= (B + A) div 4	B := 4 * B - A	R := R -S div 4 S :=S div 4	R := 4 * R - S	n:=n+2
B:= (B - A) div 4	B := 4 * B + A	R := R + S div 4 S := S div 4	R := 4 * R + S	n:=n+2
A et B	A := 1, B := 0	R := 1, S := 1		n := 0

Table III.3 : opérations de calcul du PGCD et du PGCD étendu pour la variante L5

III.4.2. Approche série

Dans cette approche, les entiers positifs peuvent être donnés sous la forme suivante :

$$A = \sum_{i=0}^{n-1} a_i 2^i \quad \text{et} \quad B = \sum_{i=0}^{m-1} b_i 2^i$$

avec $a_i, b_i \in \{0, 1\}$. Ils sont reçus en série à partir des bits les moins significatifs et l'algorithme de STEIN modifié devient :

```
fonction PGCD (A, B, δa, δb) ;      (* variante L1 *)
(* on suppose A impair, δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombre de bits de A et de B) *)
début
  tant que δb > 0 faire    (* B ≠ 0 *)
  début
    si B mod 2 = 0 alors                          (* B pair et A impair *)
      début δb := δb - 1 ;
        (A)   (  A      )
        (B) := ( B div 2 )                        (* décaler B *)
      fin
    sinon                                         (* A et B sont tous deux impairs *)
      si δa > δb alors                            (* A > B *)
        début échanger (δa, δb) ;                 (* échanger les prédicteurs *)
          (A)   (     B       )
          (B) := ( (A+B) div 2 )                  (* addition *)
        fin
      sinon
        début δb := δb - 1 ;
          (A)   (     A       )
          (B) := ( (B-A) div 2 )                  (* soustraction *)
        fin ;
  fin
  si A ≥ 0 alors PGCD := A sinon PGCD := - A ;
fin (* PGCD *).
```

A chaque fois, A et B sont ajoutés ou retranchés, deviennent pairs, donc la retenue de poids 2^1 est toujours à 1, tandis que la retenue d'entrée de poids 2^0 est soit 0 pour l'addition, soit 1 pour la soustraction. Nous ignorons a_0 et b_0 car a_0 est toujours à 1, b_0 est initialisé à 1, remis à 0 après B := B + A ou B := B - A, et prend la valeur de b_1 après B := B div 2.

Les algorithmes avec test des poids faibles ont été programmé en langage Pascal et simulé [BGR 90, CHE 90]. Nous avons obtenu pour ces algorithmes une complexité en temps de l'ordre O(n) pour des entiers à n bits.

III.5. ALGORITHMES AVEC TEST DES POIDS FORTS

La table III.4 résume la classification des variantes de l'algorithme du PGCD avec test des poids forts. Les nombres de bits δ_a et δ_b seuls sont insuffisants pour la comparaison. Par exemple A = $1\bar{1}\bar{1}\bar{1}\bar{1}$ est plus petit que B = 10 bien que $\delta_a = 5$ et $\delta_b = 2$. Donc, en plus de δ_a et δ_b, nous ajoutons les k premiers bits normalisés μ_a et μ_b de A et B.

≠	q (quotient)	p (positions)	chiffres examinés de A et B	mode
M1	-1, 0, +1	1	3	série/parallèle
M2	-1, 0, +1	1 , 2	4	parallèle
M3	-1, 0, +1	1	3	parallèle
M4	-1, 0, +1	1 , 2	4	parallèle

Table III.4 : variantes avec examen des chiffres de poids forts

La normalisation force le premier chiffre de μ_a et μ_b à être non nul, et tous les k chiffres à avoir le même signe (le chiffre zéro a deux signes), μ_a et $\mu_b \in [-2^k+1..-2^{k-1}] \cup [2^{k-1}..2^k-1]$. En augmentant la valeur de k, nous obtenons plus de chiffres dans μ_a et μ_b, des prédicteurs plus précis et une convergence plus rapide de l'algorithme du PGCD en nombre de cycles d'horloges. Mais, comme il y a une propagation de la retenue, ces cycles sont plus longs. Nous propageons donc la retenue sur trois bits, car ceci est le minimum permettant d'exécuter une autre opération intéressante qui est la division, et nous étudions également le cas de quatre bits. Quand μ_a et μ_b sont normalisés, A et B sont dits semi-normalisés et nous obtenons :

$$(|\mu_a| -1)\, 2^{\delta_a - k} < |A| < (|\mu_a| +1)\, 2^{\delta_a - k} \quad \text{et} \quad (|\mu_b| -1)\, 2^{\delta_b - k} < |B| < (|\mu_b| +1)\, 2^{\delta_b - k}$$

Lorsque A et B sont semi-normalisés, les valeurs $|\mu_b| = 2^{k-1}$ et $|\mu_a| = 2^k - 1$ méritent une certaine attention pendant le déroulement de l'algorithme. Prenons comme exemple k = 3, $B = 100\bar{1}0$ ($\delta_b = 5$, $\mu_b = 4$) et A = 1110 ($\delta_a = 4$, $\mu_a = 7$). Si B := B - (A*2δb-δa) est exécuté, le résultat sera $B = \bar{1}0010$, qui est aussi semi-normalisé et négatif, mais l'opération suivante sera B:= B + (A*2δb-δa) et nous retrouvons la valeur précédente ; l'algorithme ne se terminera jamais. En réécrivant B avant d'échanger A et B, l'algorithme évitera à μ_a de prendre les valeurs 7 ou -7 (sur 3 bits). En d'autres termes, $|\mu_a| = 7$ et $|\mu_b| = 4$ peuvent nous ramener à $|\mu_b| = 4$ qui est déjà normalisé, mais la prochaine étape donnera $|\mu_b| \leq 1$. Cette boucle infinie similaire est présentée et traitée dans [KOM 83]. Quand on fait l'addition/soustraction, une retenue appartenant à $\{\bar{1}, 0, 1\}$ doit être propagée de la partie redondante des nombres à la partie non redondante.

La table III.5 montre la nouvelle rangée de μ_b à partir de la valeur μ_a et la précédente valeur de μ_b après B := B - A, les valeurs en gras ne seront pas prises et celles en italique de la colonne $\mu_a = 7$ évitées. Une table symétrique peut être dressée pour B := B + A.

$\mu_b \setminus \mu_a$	4	5	6	7
4	-1, +1	-2, 0	-3, -1	*-4, -2*
5	0, +2	-1, +1	-2, 0	*-3, -1*
6	+1, +3	0, +2	-1, +1	*-2, 0*
7	+2, +4	+1, +3	0, +2	*-1, +1*

Table III.5

Le programme suivant présenté en pseudo-Pascal est la description comportementale du circuit de calcul du PGCD. Les variantes M1 et M3 utilisent k = 3 et les variantes M2 et M4 utilisent k = 4, chiffres pour la partie normalisée. Ce programme manipule donc 4 entiers : δ_a et δ_b avec $\lceil \log_2 (\log_2$ (valeur maximum de A ou B)) $\rceil$ bits, μ_a et μ_b avec k bits (sans signe) plus un bit de signe. b_n, le bit de signe le plus significatif de B, prend la valeur zéro si $|B| < 2^k$. Les résultats de simulations sont montrés dans l'Annexe 1.

```
fonction PGCD (A, B, k, δa, δb) ; (* variantes M1 et M2 *)
(* δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombre de bits de A et de B) *)
début
  tant que δb > 0 faire     (* B ≠ 0 *)
  début
    si abs (μb) < 2^(k-1) alors
    début
      si ((δa > δb) ou ((δa = δb) et (abs (μa) > 2^(k-1) ))) et ( bn * μb) = 2^(k-1) -1 alors
      début μb := μb + bn ; bn := - bn fin
      sinon début μb := μb + μb + bn ; bn,...,1 := bn-1,..,0; δb := δb - 1 fin(*normaliser*)
    fin
    sinon début
      si (δa > δb) ou ((δa = δb) et (abs (μa) > abs (μb))) alors    (* A > B *)
      début échanger (δa, δb);
      si ( μa * μb ) < 0 alors (A/B) := ( B / (A+B) *2 ) sinon (A/B) := ( B / (B-A) *2 ) fin
      sinon si ( μa * μb ) < 0 alors (A/B) := ( A / (A+B) *2 ) sinon (A/B) := ( A / (B-A) *2 );
    fin ;
    δb := δb - 1 ;
  fin ;
  PGCD := A ;
fin.
```

La variante suivante (M3 et M4) de l'algorithme du PGCD utilise un décalage vers les poids forts de deux positions, en plus du décalage d'une position utilisée dans la variante précédente. Dans la nouvelle variante, nous essayons de tirer un avantage de ce décalage chaque fois qu'il est possible. Autrement, les deux variantes sont similaires.

```
fonction PGCD (A, B, k, δa, δb) ;          (* variantes M3 et M4 *)
(* δa = ⌈log2 A⌉, δb = ⌈log2 B⌉ (nombre de bits de A et de B) *)
début
  tant que δb > 0 faire     (* B ≠ 0 *)
  début
    si abs (μb) < 2^(k-1) alors
    début
      si ((δa > δb) ou ((δa = δb) et (abs (μa) > 2^(k-1) ))) et ( bn * μb) = 2^(k-1) - 1 alors
      début μb := μb + bn ; bn := - bn fin
      sinon si abs (μb) < 2^(k-2) alors
      début μb := 4 * μb + 2 * bn + bn-1 ; bn..2 := bn-2..0 ; δb := δb - 2 fin
      sinon début μb := μb + μb + bn ; bn..1 := bn-1..0 ; δb := δb - 1 fin (*normaliser*)
    fin
    sinon début
      si (δa > δb) ou ((δa = δb) et (abs (μa) > abs (μb))) alors   (* A > B *)
      début échanger (δa, δb) ; échanger (A, B) fin ;   (* échanger *)
      si abs ( μa + μb ) < 2^(k-2) alors début B:= (B + A) * 4 ; δb := δb - 2 fin
      sinon si abs ( μa - μb ) < 2^(k-2) alors début B:= (B - A) * 4 ; δb := δb - 2 fin
      sinon si abs ( μa + μb ) < 2^(k-1) alors début B:= (B + A) * 2 ; δb := δb - 1 fin
      sinon début B:= (B - A) * 2 ; δb := δb - 1 fin ;
    fin;
  fin ;
  PGCD := A ;
fin.
```

Maintenant, si nous voulons calculer (R, S) tel que $A_0 * R - B_0 * S = H = PGCD(A_0, B_0)$, nous utilisons la table III.6 pour exécuter la séquence inverse des opérations qui donne H.

	type	calcul du PGCD	inverse	preserve A*R-B*S
Normaliser	1	B := B * 2	B := B div 2	S := S * 2
Echanger	2	$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$	$\begin{pmatrix} R \\ S \end{pmatrix} = \begin{pmatrix} -S \\ -R \end{pmatrix}$
Add	3	B := (B + A) * 2	B := B div 2 - A	R := R - S * 2 S := S * 2
Soust	4	B := (B - A) * 2	B := B div 2 + A	R := R + S * 2 S := S * 2
Valeurs init.		A et B	A := 1, B :=0	R := 1, S := 1

Table III.6 : opérations de calcul du PGCD et du PGCD étendu pour les variantes M1 et M2

L'algorithme produit directement le résultat et comme nous n'avons pas besoin de correction, les valeurs initiales de A_0 et B_0, n'ont pas à être sauvegardées comme dans la version avec test des poids faibles.

A partir de $(A_n, B_n) = (H, 0)$ et en exécutant la colonne du milieu, nous retournons à la valeur (A_0, B_0). Rappelons que, lors de calcul du PGCD, B_n a été multiplié par deux, donc sa valeur finale est paire. Ici, B_n est divisée chaque fois par deux. A partir de $(A_n, B_n) = (1, 0)$ et, comme B_n / H est pair également, nous obtenons donc à la fin $(A_0 / H, B_0 / H)$. Les deux algorithmes de calcul des coefficients de Bezout et de la simplification des fractions, qui sont pratiquement identiques, sont donnés ci-dessous.

```
procédure PGCD étendu (R, S) ;
(* trouver R et S tel que A * R - B * S = PGCD(A, B) *)
R:= 1 ; S := 1 ;                          (* initialiser *)
Pour i := nbetape à 1 par pas de -1 faire
début
    cas type [ i ] de
        1: S := S * 2 ;                   (* normaliser *)
        2: (R)  = ( -S ) ;                (* échanger *)
           (S)    ( -R )
        3: (R)  := ( (R-S) *2 ) ;         (* soustraction *)
           (S)     (  S * 2   )
        4: (R)  := ( (R+S) *2 ) ;         (* addition *)
           (S)     (  S * 2   )
    fin ;
fin.
```

Les algorithmes avec test des poids forts ont été programmé en langage Pascal et simulé [BGR 90, CHE 90]. Nous avons obtenu pour ces algorithmes une complexité en temps de l'ordre O(n) pour des entiers à n bits.

```
procédure simplifier la fraction(A, B) ;
(* trouver A / PGCD(A, B) et B / PGCD(A, B) *)
A:= 1 ; B := 0 ;                          (* initialiser *)
Pour i := nbetape à 1 par pas de -1 faire
début
    cas type [ i ] de
        1: B := B div 2 ;                 (* normaliser *)
        2: (A) = (B) ;                    (* échanger *)
           (B)   (A)
        3: B := B div 2 - A ;             (* soustraction *)
        4: B := B div 2 + A ;             (* addition *)
    fin ;
fin.
```

III.6. CONCLUSION

Lorsque nous sommes passés du domaine des petits entiers à celui des grands entiers, nous avons effectué quelques modifications aux algorithmes de calcul du PGCD (table III.7). En effet, ces derniers ont six variables (A, B, μ_a, μ_b, δ_a, δ_b) pour les très grands nombres et utilisent des opérateurs en notation redondante. Cependant les algorithmes restent complexes et heuristiques, comparés à ceux utilisés pour l'arithmétique des petits nombres (chapitre II §II.2) qui sont simples et déterministes et utilisent seulement deux variables (A, B) dans un système non redondant. Néanmoins pour une simple opération comme l'addition, nous obtenons un temps de propagation en ligne énorme dû à la propagation de retenue.

Petits entiers	Grands entiers
2 variables	6 variables
propagation de retenue	pas de propagation
algorithme simple	algorithme complexe
algorithme déterministe	algorithme heuristique

Table III.7 : conséquences sur les algorithmes

Concernant les très grands nombres, quatre approches ont été détaillées, deux avec test des poids forts et deux avec test des poids faibles, en parallèle ou en série. Nous avons essayé d'obtenir plusieurs variantes pour chaque approche, en manipulant les quotients ainsi que le nombre de positions de décalage (tables III.1 et III.4).

La convergence des algorithmes n'a pas été démontrée mais seulement tenté sur un grand nombre d'échantillons. Cependant, toutes les opérations effectuées dans ces algorithmes concernent la valeur du PGCD. Nous pouvons donc ainsi dire que chaque fois que les algorithmes convergent, ils donnent bien le PGCD.

La complexité en temps de chacun des algorithmes détaillés dans ce chapitre, de calcul du PGCD de deux entiers à n bits est évaluée à O(n). En effet, dans les variantes où nous utilisons des additionneurs d'un bit signé à quatre entrées et deux sorties, la complexité en temps est estimée à O(n).

Ces algorithmes ont été présentés avec seulement une addition/soustraction par cycle machine et plusieurs décalages différents. Les tests ont été remplacés par des estimateurs, car nous manipulons de très grands nombres.

Pour chacune des variantes décrites dans ce chapitre, nous allons proposer une architecture qui fera l'objet du chapitre suivant.

CHAPITRE IV

ARCHITECTURES

IV.1. INTRODUCTION

IV.2. PRINCIPE DE L'ADDITIONNEUR PARALLELE : CFA

IV.3. ARCHITECTURES AVEC TEST DES POIDS FAIBLES

IV.3.1. Approche parallèle
IV.3.2. Approche série

IV.4. ARCHITECTURES AVEC TEST DES POIDS FORTS

IV.4.1. Approche parallèle
IV.4.2. Approche série

IV.5. CONCLUSION

IV.1. INTRODUCTION

Dans ce chapitre, nous présentons les architectures des différents algorithmes développés dans le chapitre III. Nous essayons à chaque fois qu'il est possible d'obtenir des tranches de bits qui exécutent des opérations simples telles que l'addition/soustraction, le décalage et l'échange. Pour chaque architecture, nous essayons d'évaluer le nombre de transistors nécessaires pour traiter un bit.

IV.2. PRINCIPE DE L'ADDITIONNEUR REDONDANT

Au cours de cette section, nous essayons de présenter le fonctionnement de cet additionneur utilisé pour chacune des approches parallèles. Son schéma externe est représenté à la figure IV.1. Nous obtenons un additionneur de deux nombres de trois chiffres signés chacun écrits en notation redondante.

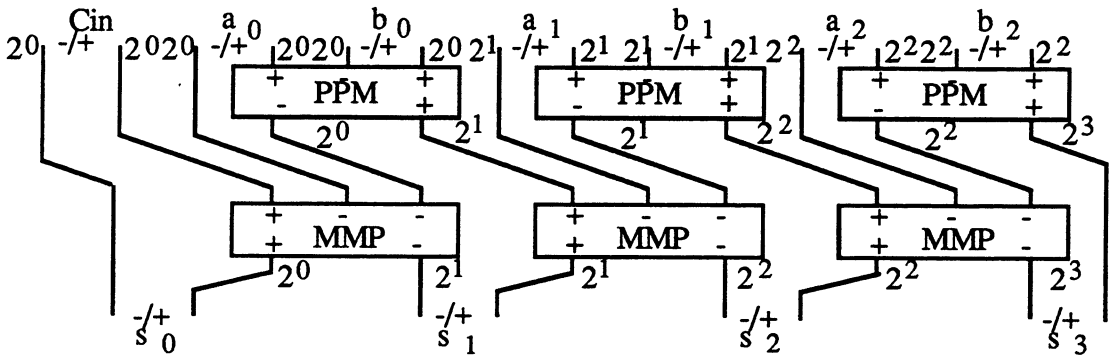


Figure IV.1 : schéma externe d'un additionneur redondant

Remarquons que le parallélisme réalisé pour ce type d'additionneur n'est pas total, en effet, l'addition ou la soustraction se réalise avec une propagation locale de la retenue qui influe au plus sur le résultat du bit suivant comme l'illustre la figure II.1.

Si nous examinons la structure de cette unité arithmétique, nous pouvons distinguer la boîte PPM (Plus Plus Moins) et la boîte MMP (Moins Moins Plus) ; il s'agit d'un additionneur qui a trois entrées, D, E et F et deux sorties, C_{out} (retenue) et S_{out}. La sortie du PPM est D + E - F et celle du MMP est F - D - E, quelques soient les signes affectés aux entrées et aux sorties, la retenue est de poids 2, et la somme est de poids 1.

La configuration montrée dans la figure IV.2 est celle de la cellule PPM.

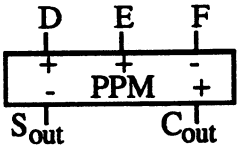


Figure IV.2 : schéma externe de la cellule PPM

Nous avons la relation suivante,

$$D + E - F = (2*C_{out}) - S_{out}$$

Par contre, la configuration montrée dans la figure IV.3 est celle du MMP.

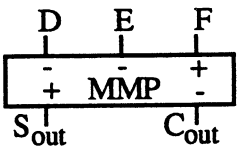


Figure IV.3 : schéma externe d'un MMP

La relation devient,

$$F - D - E = S_{out} - 2*C_{out}$$

Les équations logiques d'un PPM ou MMP sont,

$$S_{out} = D \oplus E \oplus F$$

$$C_{out} = (D \wedge E) \vee (D \wedge \bar{F}) \vee (E \wedge \bar{F})$$

IV.3. ARCHITECTURES AVEC TEST DES POIDS FAIBLES

IV.3.1. Approche parallèle

Dans l'approche parallèle, les deux premiers nombres sont envoyés en série dans les registres à décalage en notation redondante.

Soit un nombre A donné par $\sum_{i=0}^{n-1} a_i 2^i$, $a_i \in \{\bar{1}, 0, 1\}$. Avec cette notation, il est facile d'effectuer une addition ou une soustraction (en changeant le signe d'un nombre). Ce qui revient à échanger seulement les deux bits de chacun de ces chiffres et ne demande pas d'extension de signe.

La structure d'une partie de l'opération est une description fonctionnelle du circuit de calcul du PGCD. Le chemin de données de la variante L4 est montré dans la figure IV.4. Notons que chaque fil représente deux bits.

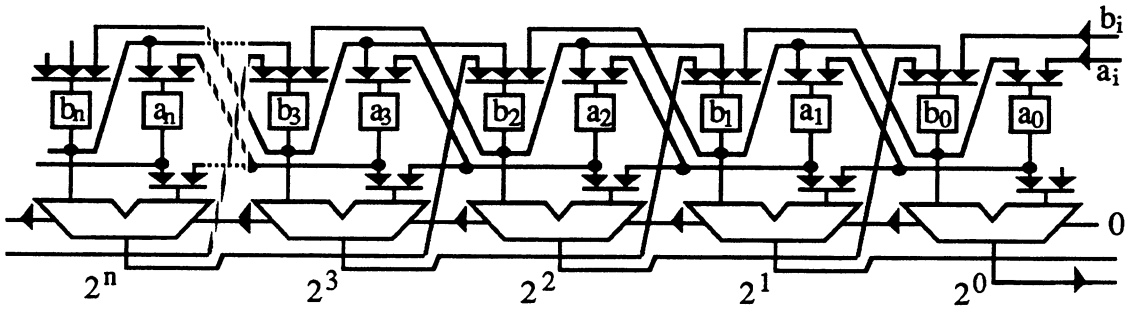


Figure IV.4 : chemin de données de la variante L4

L'implantation des variantes L2, L3, L4 et L5 nécessitent respectivement 158, 200, 216 et 222 transistors par chiffre traité pour exécuter chaque opération de calcul du PGCD. Dans le cas du PGCD étendu, nous obtenons respectivement 202, 268, 276 et 296 transistors.

IV.3.2. Approche série

Sur des entiers transmis bit à bit poids faible en tête, diviser B par deux ou décaler à droite (figure IV.5) est simple à réaliser en retardant seulement d'un cycle d'horloge, mais en même temps A serait divisé par deux, il suffit donc de mémoriser la valeur de A dans une bascule pour pouvoir récupérer sa valeur initiale.

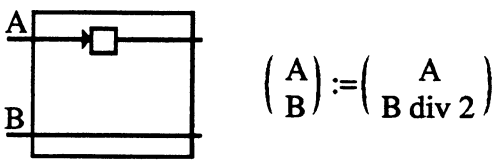
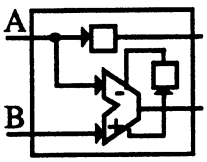


Figure IV.5 : décalage à droite

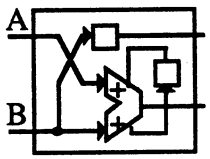
Soustraire A de B (figure IV.6) exige en plus une bascule pour mémoriser la retenue. Cette bascule est initialisée à 1, puis mise à jour pour chaque nouveau bit. En effet, $B-A = B+\bar{A}+1$.



$$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} A \\ (B-A) \text{ div } 2 \end{pmatrix}$$

Figure IV.6 : soustraction et décalage à droite

Enfin, échanger deux nombres en-ligne nécessite seulement le croisement des deux fils (figure IV.7). Cette opération est exécutée en même temps que l'addition. Cette dernière est semblable à la soustraction et la bascule doit être initialisée à 0.



$$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} B \\ (A+B) \text{ div} 2 \end{pmatrix}$$

Figure IV.7 : échange et addition décalée à droite

Les bits d'entrées sont fournis en série à un vecteur de cellules programmables (figure IV.8). Une partie de contrôle programme chaque cellule séquentiellement de gauche à droite à travers deux registres de 2 bits appelés registres de contrôle, en exécutant une des opérations représentées ci-dessus.

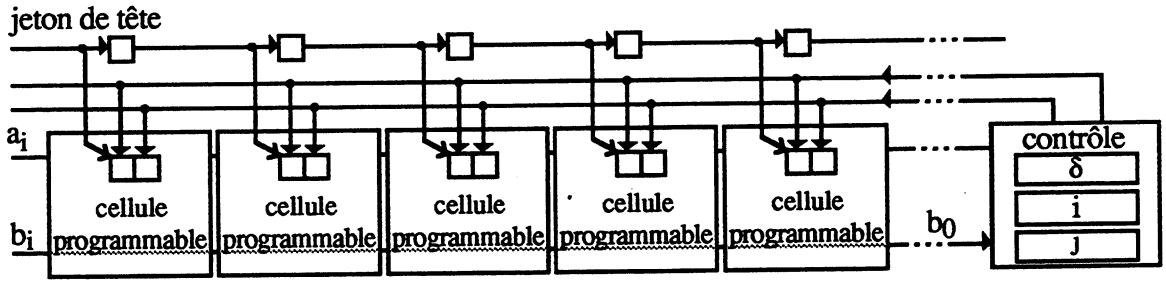


Figure IV.8 : vecteurs de cellules programmables par la partie contrôle

L'algorithme de la partie contrôle (à droite du dessin ci-dessus) se résume à :

```
fonction PGCD (A, B, δa, δb );
début
  i := δa + δb + 1 ; (* initialiser les sommes de nombre de bits *)
  lire le premier bit b0 de B; positionner le jeton de tête à gauche;
  tant que i ≠ 0 faire
  début
    ok := ( b0 = 0 );
      si δb > 0 alors
        début
          programmer la cellule courante à faire une addition (code 01);
          si ok alors échange (δa, δb ) sinon i := i - 1;
        fin
      sinon
          début
          programmer la cellule courante à faire une soustraction (code 11);
          i := i - 1;
        fin ;
    si ok alors j := i div 2;
     δb := δb - 1;
    chercher les bits suivants de A et de B ;
    décaler le jeton de tête de façon que la cellule courante soit la suivante ;
  fin ; (* maintenant B est 0 et A est le PGCD *)
  forcer la cellule courante à faire une addition ; (* B := A *)
  tant que j ≠ 0 faire
      début j := j - 1;
        émettre b0 ; (* émettre le bit suivant du PGCD *)
      fin ;
fin (* PGCD *)
```

La figure IV.9 montre l'exemple d'un vecteur de cellules après 4 cycles d'horloge. Notons que la cellule de décalage est la même qu'une cellule libre.

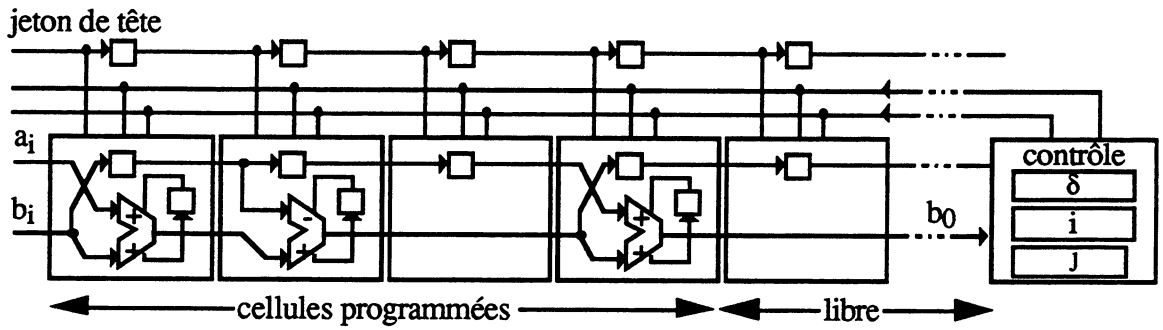


Figure IV.9 : exemple de cellules programmées par la partie contrôle

Pour chaque cycle d'horloge, b_0 est testé par la partie de contrôle du circuit afin de programmer la cellule suivante. Le choix judicieux du code de programmation de la cellule de décalage permet le contrôle local du chargement du registre de contrôle. La boucle de rétro-

action partie contrôle-partie opérative (figure IV.10) est supprimée, la partie contrôle propose une action, et la partie opérative l'exécute ou non.

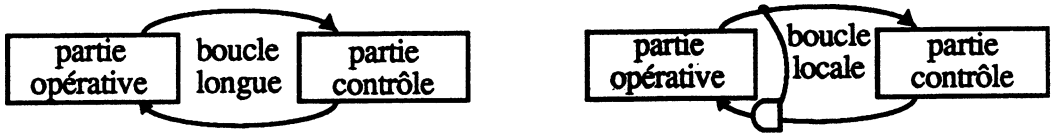


Figure IV.10 : la boucle longue est remplacée par la boucle locale

Maintenant, nous évaluons le nombre de transistors pour chaque cellule fonctionnelle de la variante L1 dont le schéma est présenté dans la figure IV.11.

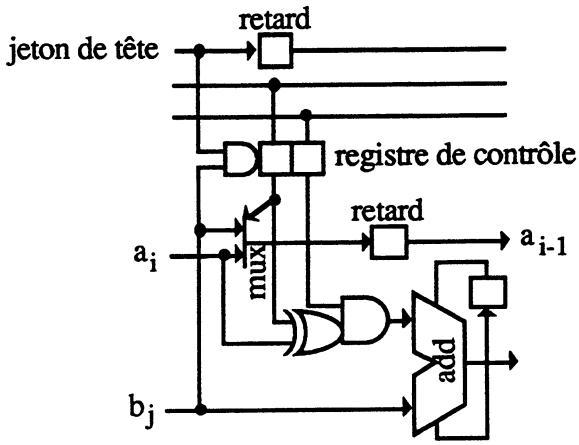


Figure IV.11 : cellule fonctionnelle de la variante L1

Pièce	instances	transistors (règles CMOS)	
additionneur	1	30	additionneur rapide
porte Et	1	4	
porte Ou-Exclusif	1	6	
bascule	2	10	maître-esclave statique
bascule	2	5	latch statique (contrôle)
interrupteur	5	1	pour reset, preset, select
TOTAL		**75** transistors par tranche de bit	

Le circuit décrit à la figure IV.13 ne fonctionne pas pour les très grands nombres. Cela est dû au retard des portes accumulé le long des cellules, ralentissant ainsi le cycle d'horloge. Dans ce cas, nous n'avons pas de propagation de la retenue mais une propagation du résultat. En introduisant toutes les 8 cellules fonctionnelles une cellule de restauration (une barrière

temporelle) ne contenant que des bascules maîtres-esclaves (figure IV.12), nous éliminons la propagation du résultat, changeant le circuit combinatoire en un circuit pipeline. Pendant le parcours, nous perdons un cycle d'horloge toutes les 8 cellules fonctionnelles. Cette cellule de restauration sert aussi de "voiture-balai", recyclant les cellules inutilisées. Pour cela, nous introduisons un "jeton de queue" délimitant la partie active du circuit. Cette cellule de restauration est utilisée dans le calcul du PGCD pour économiser le matériel. Nous verrons ultérieurement que ceci ne fonctionne malheureusement pas pour le calcul des coefficients de Bezout.

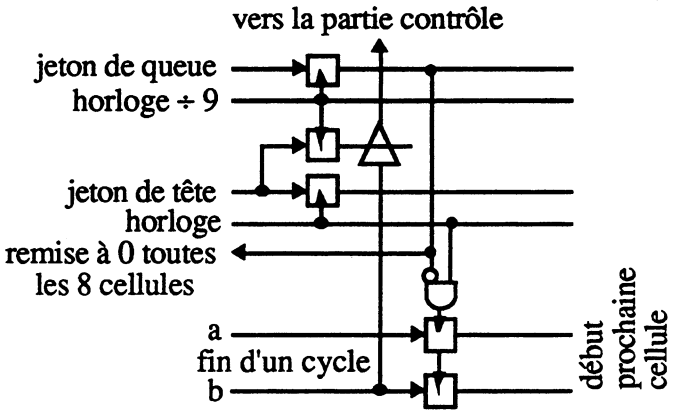


Figure IV.12 : cellule de restauration

Le nombre de cycles permettant de trouver le PGCD dépend des valeurs des paramètres, le maximum étant $2*(\delta_a + \delta_b)$ cycles. Une nouvelle cellule est programmable à chaque cycle d'horloge.

Nous appelons "cellule morte" une cellule dont les entrées et sorties resteront stables jusqu'à ce que l'algorithme soit terminé. Dans le cas contraire, il s'agit d'une "cellule active". Nous comptons n cellules actives, car chaque résultat intermédiaire peut être calculé avec n bits. Toutes les cellules mortes peuvent être fonctionnellement remplacées par un registre de 2 bits permettant une extension de signe de A et B. Ce registre est localisé dans une cellule de restauration qui est une cellule intermédiaire placée toutes les 8 cellules fonctionnelles. Un jeton de queue est positionné par le dernier bit des opérandes, déclenchant le recyclage de 8 cellules fonctionnelles mortes en cellules oisives en mettant le registre de contrôle à 0.

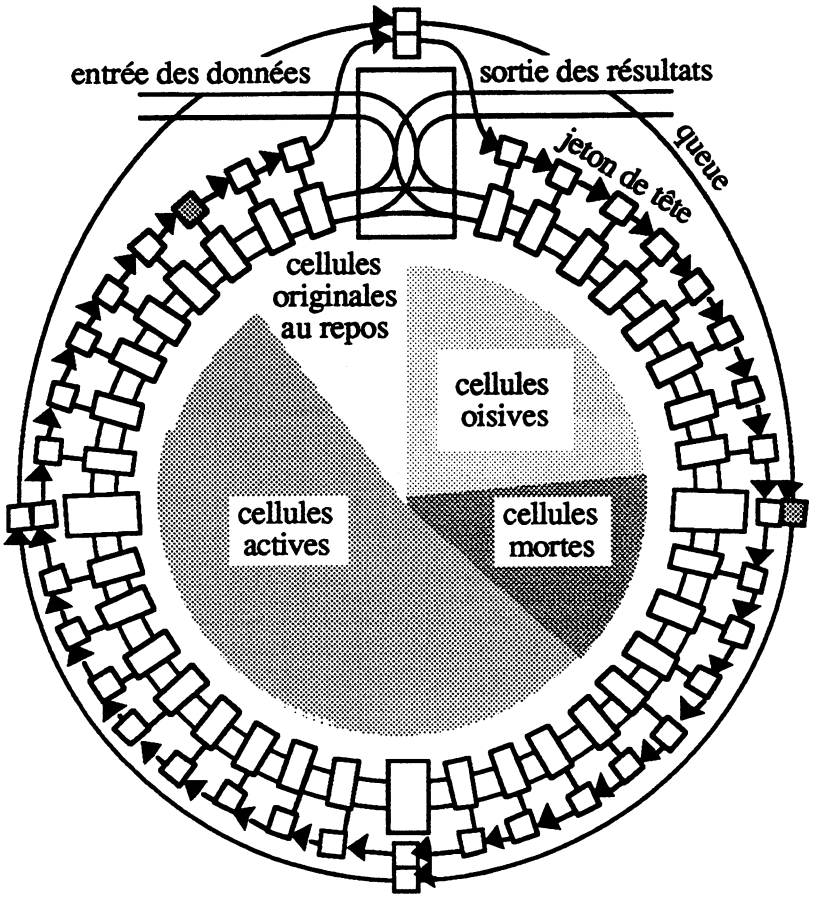


Figure IV.13 : circuit de calcul du PGCD en mode série avec test des poids faibles

Le calcul du PGCD étendu en mode série avec test des poids faibles est possible, mais coûte cher du point de vue matériel. Pour le réaliser :

- nous utilisons les opérations stockées dans les registres de contrôle au lieu d'une pile. Dans ce cas, nous n'avons pas besoin de la partie de contrôle, mais seulement du registre de contrôle qui doit être programmable,
- nous abandonnons l'algorithme de recyclage, afin de conserver le registre de contrôle,
- nous utilisons deux additionneurs supplémentaires d'un bit par cellule pour calculer R + B et S + A, afin de les conserver toujours dans le même ordre,
- nous utilisons deux additionneurs/soustracteurs supplémentaires par cellule pour calculer 2*B ± A, pour retrouver la séquence des valeurs de A et B, ainsi que 2*(R + B) ± (S + A).

En partant des valeurs initiales A, B, R et S qui sont respectivement H, 0, 1, 1 et en exécutant les opérations dans le sens inverse donnant le PGCD (table IV.1), nous aboutissons au calcul des coefficients de Bezout R et S. L'implantation de la variante L1 nécessite 75 transistors par chiffre traité pour exécuter chaque opération de calcul du PGCD. Dans le cas du PGCD étendu, nous obtenons 526 transistors.

Calcul du PGCD	préserve la relation (A * R - B * S) = H
$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} A \\ B \text{ div } 2 \end{pmatrix}$	$\begin{pmatrix} A \\ B \\ R \\ S \end{pmatrix} := \begin{pmatrix} A \text{ div} 2 \\ B \\ R \\ S \text{ div} 2 \end{pmatrix}$
$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} B \\ (A+B) \text{ div } 2 \end{pmatrix}$	$\begin{pmatrix} A \\ B \\ R \\ S \end{pmatrix} := \begin{pmatrix} B \text{div} 2 \\ B-A \text{ div} 2 \\ (S+A) \text{ div} 2 \\ (R+B)-(S+A) \text{div} 2 \end{pmatrix}$
$\begin{pmatrix} A \\ B \end{pmatrix} := \begin{pmatrix} A \\ (B-A) \text{ div } 2 \end{pmatrix}$	$\begin{pmatrix} A \\ B \\ R \\ S \end{pmatrix} := \begin{pmatrix} B \text{div} 2 \\ B+A \text{ div} 2 \\ (R+B)+(S+A) \text{div} 2 \\ (S+A) \text{ div} 2 \end{pmatrix}$

Table IV.1 : opérations calculant le PGCD et le PGCD étendu

BRENT et KUNG [BKU 83] ont proposé une variante qui est légèrement plus rapide que la précédente, et qui possède trois sortes de cellules et teste a_1 ainsi que b_1. L'algorithme de la partie contrôle de la variante L2 est donné ci-dessous.

```
fonction PGCD (A, B, δa, δb ) ;      (* variante L2 *)
début
   tant que δb ≥0 faire
   début
        si b0 = 0 alors
              programmer la cellule courante à faire un décalage
        sinon si δa < δb et b1 ≠ a1 alors
              programmer la cellule courante à faire un échange-addition
        sinon si δa < δb et b1 = a1 alors
              programmer la cellule courante à faire un échange-soustraction
        sinon si δa ≥ δb et b1 ≠ a1 alors
              programmer la cellule courante à faire une addition
        sinon si δa ≥ δb et b1 = a1 alors
              programmer la cellule courante à faire une soustraction ;
        si δa < δb alors début échange(δa, δb); échange (b1 = a1) fin;
        δb:= δb - 1; b0 := b1;
        chercher les bits suivants de A et de B ;
   fin ; (* maintenant B est 0 et A est le PGCD *)
fin (* PGCD *) .
```

Il est facile d'implanter une version série des variantes L3 ou L4, mais elles sont très complexes. L'algorithme de la partie contrôle de la variante L4 se résume à :

fonction PGCD (A, B, δ_a, δ_b) ; (* variante L4 *)
début
  tant que $\delta_b \geq 0$ **faire**
  début
    si $b_0 = 0$ **et** $b_1 = 0$ **alors**
      programmer la cellule courante à faire un décalage
    si $b_0 = 0$ **et** $b_1 \neq 0$ **alors**
      programmer la cellule courante à faire un décalage-addition
    sinon si $\delta_a < \delta_b$ **et** $b_1 \neq a_1$ **alors**
      programmer la cellule courante à faire un échange-addition
    sinon si $\delta_a < \delta_b$ **et** $b_1 = a_1$ **alors**
      programmer la cellule courante à faire un échange-soustraction
    sinon si $\delta_a \geq \delta_b$ **et** $b_1 \neq a_1$ **alors**
      programmer la cellule courante à faire une addition
    sinon si $\delta_a \geq \delta_b$ **et** $b_1 = a_1$ **alors**
      programmer la cellule courante à faire une soustraction ;
    si $\delta_a < \delta_b$ **alors début** échange (δ_a, δ_b); échange ($b_1 = a_1$) **fin;**
    $\delta_b := \delta_b - 1$; $b_0 := b_1$;
    chercher les bits suivants de A et de B ;
  fin ; (* maintenant B est 0 et A est le PGCD *)
fin (* PGCD *) .

Nous avons présenté dans la figure IV.14 les différentes cellules programmables de la variante L4.

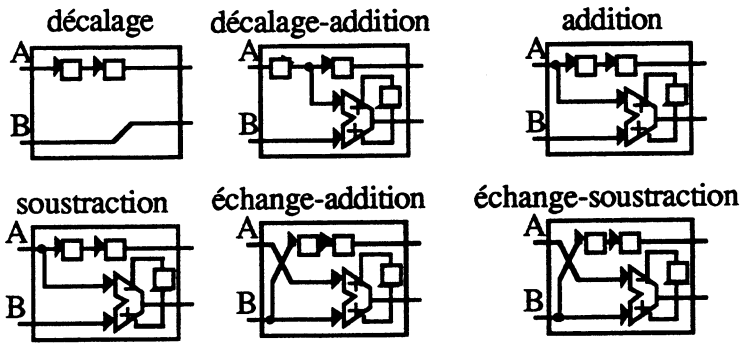


Figure IV.14 : cellules programmables de la variante L4

IV.4. ARCHITECTURES AVEC TEST DES POIDS FORTS

IV.4.1. Approche parallèle

Le chemin de données de calcul du PGCD et du PGCD étendu de l'algorithme M1 et M2, de la division et de la simplification décrit dans le chapitre III §III.5, est montré dans la figure IV.15. La partie de contrôle n'est pas détaillée dans ce chapitre. Dans cette approche la relation B := B div 2 ± A est faite par B := (B ± 2*A) div 2.

Le chemin de données des variantes M3 et M4 peut être facilement déduit à partir de la figure IV.15 en permettant de charger b_i par la sortie de la tranche de bit s_{i-2} ou s_{i+2}, au lieu de s_{i-1} ou s_{i+1} et en stockant les 3 bits de codage des opérations de calcul du PGCD dans la pile LIFO.

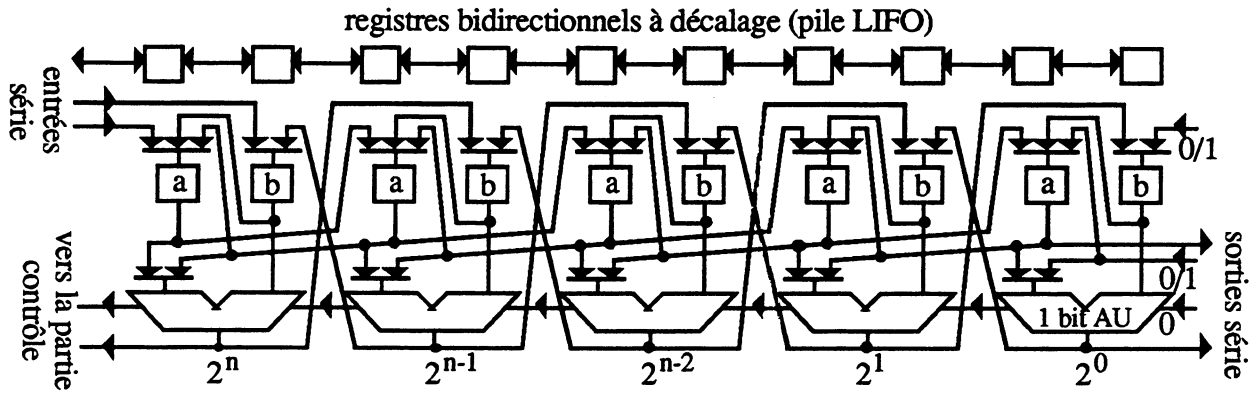


Figure IV.15 : chemin de données des variantes M1 et M2, de la division et de la simplification

La figure IV.16 illustre les différentes étapes de calcul du PGCD et du PGCD étendu.

- La première étape initialise tous les registres à zéro,
- Lors de la deuxième étape, nous décalons les entrées en série par les poids faibles dans les registres A et B,
- Durant la troisième étape, nous calculons le PGCD de ces deux nombres en exécutant les opérations décrites par l'algorithme M1 ou M2 qui, au fur et à mesure sont stockées dans une pile LIFO. Ensuite, le résultat H de ce calcul est décalé en série par les poids faibles et reçu en sortie,
- Pendant la quatrième étape, nous réinitialisons à 1 les bits de poids faible des registres A et B qui seront utilisés pour le calcul des coefficients de Bezout R et S (PGCD étendu),
- En cinquième étape nous calculons R et S en exécutant les opérations qui se trouvent dans la pile LIFO dans le sens inverse,

- Enfin, la dernière étape consiste à récupérer les résultats décalés par les poids faibles tout en exécutant la deuxième étape pour les prochaines entrées des opérandes.

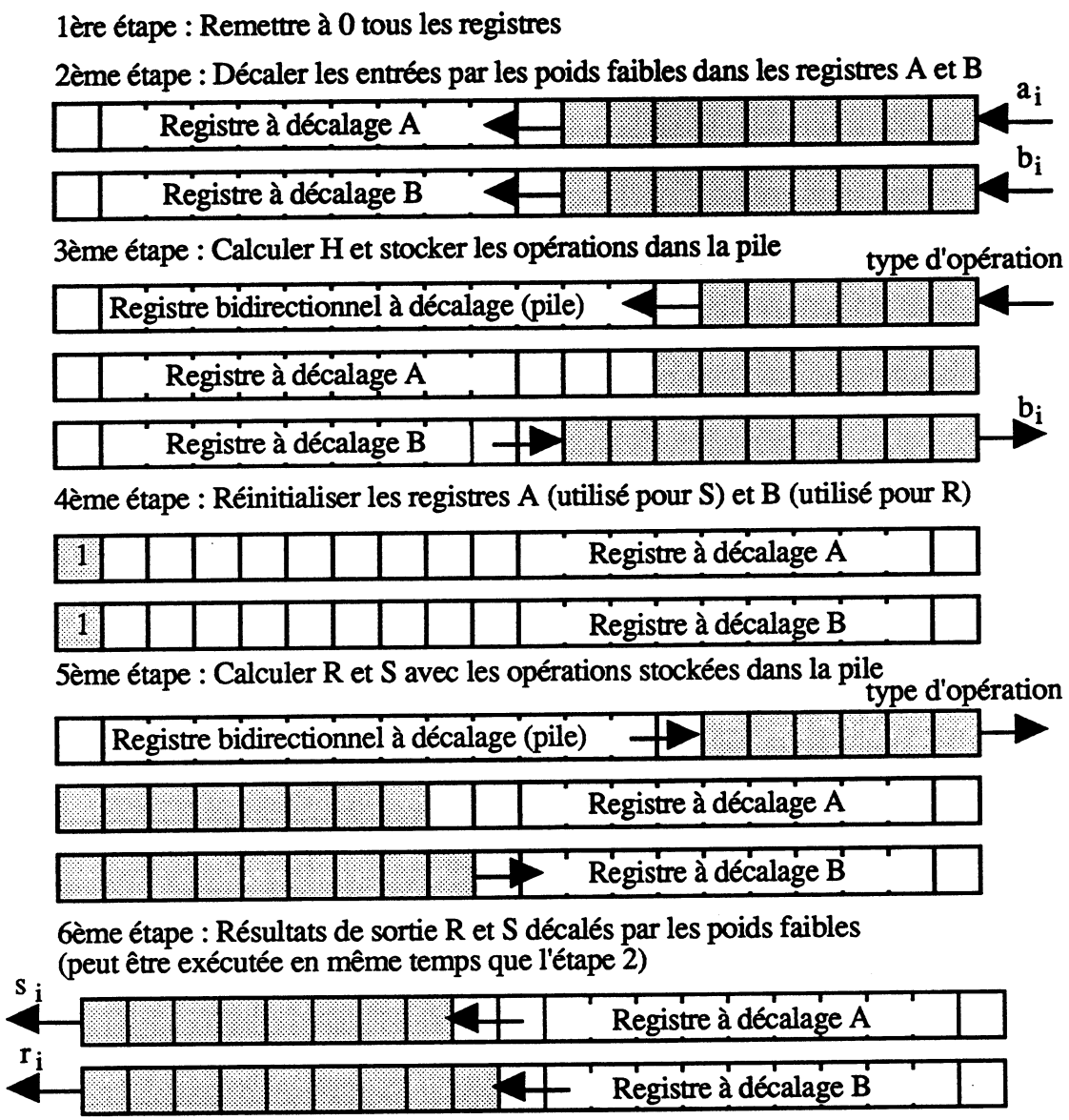


Figure IV.16 : différentes étapes de calcul du PGCD et du PGCD étendu

L'Unité Arithmétique 1 bit montrée à la figure IV.17 est constituée de trois blocs, le multiplieur dont les sorties sont b_i, $-b_i$ ou 0 et deux blocs, PPM et MMP (identiques), qui forment un additionneur en notation redondante.

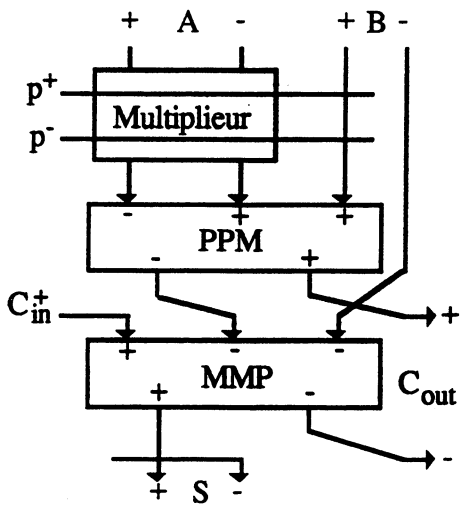


Figure IV.17 : Unité Arithmétique avec des opérandes en notation redondante

Nous allons maintenant évaluer le nombre de transistors pour chaque tranche de cellules de la variante M1 et M2 dont le schéma est présenté dans la figure IV.15.

Pièce	instances	transistors (règles CMOS)	
bloc PPM	2	20	additionneur
multiplieur	1	16	
multiplexeur	3	8	
inverseur	1	2	
bascule	7	10	maître-esclave
interrupteur	4	1	pour la remise à 0
TOTAL		**156** transistors par tranche de bit	

Les implantations des variantes M1, M2 et M3, M4 nécessite respectivement 138 et 156 transistors par chiffre traité pour exécuter chaque opération de calcul du PGCD. Dans le cas du PGCD étendu, nous obtenons respectivement 156 et 160 transistors.

IV.4.2. Approche série

Les algorithmes M1 et M2 sont simples à implanter en mode série. Nous proposons la cellule programmable de la variante M1 (figure IV.18), qui exécute la normalisation, échange-addition, échange-soustraction, addition ou soustraction, et à chaque fois le résultat est multiplié par 2.

Le registre de contrôle de chaque cellule doit avoir trois bits. Les blocs D sont des bascules maître-esclave, qui avant de commencer le calcul devraient toutes être remises à 0 (0 a deux représentations possibles 00 ou 11).

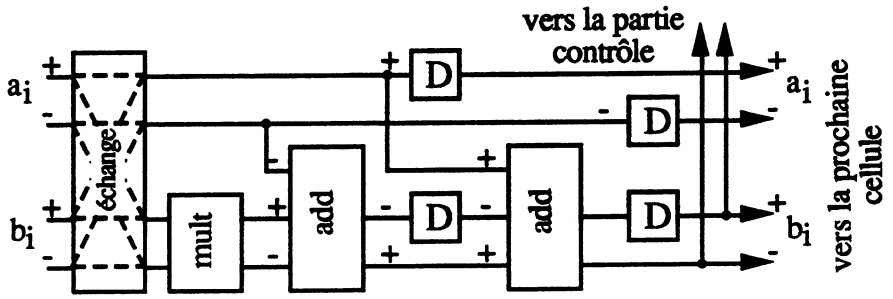


Figure IV.18 : cellule programmable de la variante M1

Malheureusement, il n'est pas possible de propager b_n à travers les cellules jusqu'à la partie contrôle. Cette dernière recoderait la cellule en changeant les valeurs des bascules. Il est donc nécessaire de propager b_n à travers les multiplexeurs, ralentissant ainsi le cycle d'horloge. Le calcul du PGCD étendu exige des cellules supplémentaires dupliquées.

L'implantation des variantes M1 et M2 nécessite 192 transistors par chiffre traité pour exécuter chaque opération de calcul du PGCD. Dans le cas du PGCD étendu, nous obtenons 344 transistors.

IV.5. CONCLUSION

Nous avons réalisé une architecture de l'approche parallèle avec les poids forts en tête. Nous appliquons aussi les principes de Bezout par les poids forts pour calculer les coefficients de Bezout R et S qui vérifient R * A - S * B = PGCD(A, B) ; ceux par les poids faibles sont aussi performants.

Nous avons également conçu une architecture de l'approche série avec test des poids faibles et des poids forts. Les coefficients de Bezout sont compliqués et coûteux à calculer.

Le choix de la meilleure tactique du point de vue performance et coût en matériel sera évoqué dans le prochain chapitre. La méthode est classique : nous établissons un tableau comparatif des différentes méthodes simulées et nous en déduisons celle qui obtient le meilleur compromis vitesse/surface de silicium.

CHAPITRE V

SIMULATION ET CHOIX DE L'APPROCHE A IMPLANTER

V.1. INTRODUCTION

V.2. SIMULATION ET CHOIX DE L'APPROCHE A IMPLANTER

V.3. CONCLUSION

V.1. INTRODUCTION

Dans ce chapitre, nous présentons le rapport coût/performance des algorithmes décrits dans les chapitres III et IV. Pour chaque algorithme de calcul du PGCD et du PGCD étendu, les simulations ont été effectuées avec un même ensemble de 200 échantillons de 2000 bits. Chaque échantillon est généré arbitrairement de façon à être impair (chiffre le moins significatif différent de zéro) et normalisé (chiffre le plus significatif différent de zéro).

Les performances sont données par le rapport du nombre de bits traités au nombre de cycles d'horloge simulés.

Pour chaque algorithme, une architecture décrite dans le chapitre IV a été proposée, soit en série, soit en parallèle, soit parfois les deux. Cela nous a permis d'obtenir le chemin des données, le coût de chaque opération en terme de nombre de cycles d'horloge, et le coût d'un bit en terme de nombre de transistors.

Dans le calcul du PGCD et du PGCD étendu, le coût d'addition/soustraction est d'un cycle d'horloge. L'opération d'échange est exécutée en parallèle avec l'addition/soustraction et ne nécessite pas de cycle supplémentaire. De même concernant les décalages. Le schéma du matériel d'addition/soustraction est le même que celui montré dans [GHM 89].

L'avantage d'utiliser la distributivité du décalage sur l'addition/soustraction entraîne un gain en terme de matériel d'un petit nombre de transistors. Par exemple pour les relations
B := 8 * B + 2 * A et B := 4 * B + A présentées dans la variante L5.

V.2. SIMULATION ET CHOIX DE L'APPROCHE A IMPLANTER

Dans les variantes L2, L3, L4 et L5 (test des premiers bits de poids faible), les nombres sont traitées en parallèle tandis que dans L1, elles sont traitées en série. Pour chaque variante, nous avons évalué le coût en nombre de cycles d'horloges par chiffre ainsi que le nombre de transistors par chiffre. De même pour les variantes M1, M2, M3 et M4 (test des premiers bits de poids fort) où les nombres sont manipulées en parallèle, M1 et M2 pouvant également être exécutées avec des nombres en série. La table V.1 résume les résultats de simulation pour les différentes variantes de calcul du PGCD et du PGCD étendu.

coût \ variantes			premiers bits du poids faible					premiers bits du poids fort			
			L1	L2	L3	L4	L5	M1	M2	M3	M4
cycles d'horloges par chiffre		PGCD	1,80	1,76	1,36	1,12	1,09	1,09	1,08	0,97	0,96
		PGCD Etendu	2,63	4,74	4,06	3,44	3,37	2,18	2,16	1,95	1,90
transistors par chiffre	parallèle	PGCD		158	200	216	222	138	138	156	156
		PGCD Etendu		202	268	276	296	156	156	160	160
	série	PGCD	75					192	192		
		PGCD Etendu	526					344	344		

Table V.1 : résultats de simulation

Dans l'approche série (figure V.1), l'architecture de la variante L1 pour le calcul du PGCD a une faible vitesse ainsi qu'une faible complexité. Par contre, dans le cas contraire, la variante M1 a une architecture plus complexe avec une grande vitesse. Entre ces deux extrêmes, nous obtenons des variantes de l'approche parallèle, parmi lesquelles les algorithmes M3 ou M4 qui présentent un meilleur compromis avec une architecture de complexité modérée et une grande vitesse.

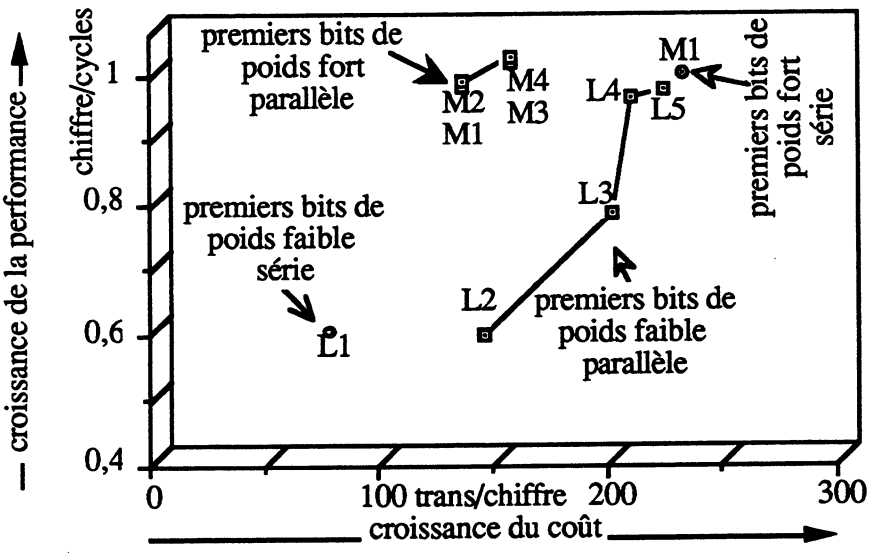


Figure V.1 : vitesse et complexité des différentes variantes pour le calcul du PGCD

La figure V.2 confirme que les versions parallèles qui testent les chiffres les plus significatifs sont également meilleures pour le calcul du PGCD suivi du calcul des coefficients de Bezout pour le calcul du PGCD étendu.

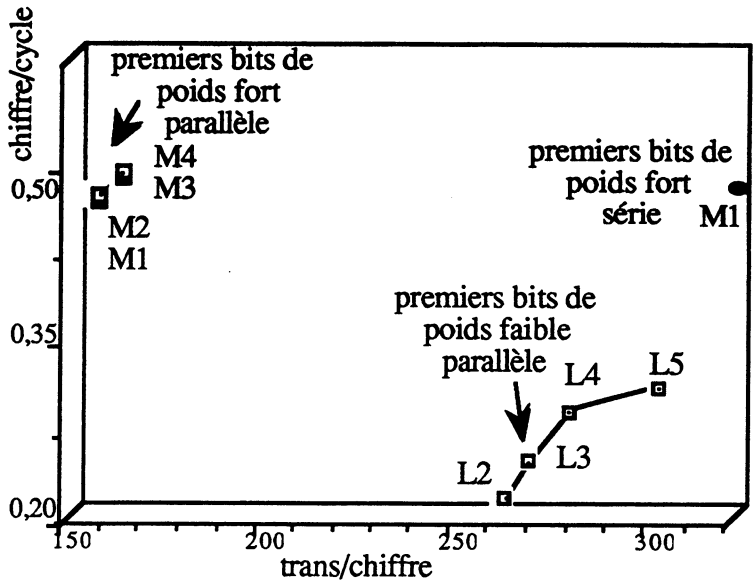


Figure V.2 : vitesse et complexité des différentes variantes pour le calcul du PGCD suivi des coefficients de Bezout

Le choix de l'algorithme à implanter s'est donc porté sur la variante M3 qui nécessite un cycle d'horloge pour chaque opération de calcul du PGCD et deux cycles pour le calcul des coefficients de Bezout. Le coût matériel pour chaque tranche de cellules est de 156 transistors. Le chemin de données des opérandes est constitué d'une partie en notation redondante et d'une partie en notation en complément à deux (3 bits plus un bit de signe). De plus, les opérations d'addition, soustraction, décalage et comparaison peuvent être exécutées dans les deux notations différentes.

V.3. CONCLUSION

Bien que le gain potentiel en terme matériel de l'approche d'EUCLIDE [KNU 81] (test des chiffres les plus significatifs) soit très faible, l'estimateur de BRENT utilisée par cette approche pour les très grands nombres est suffisamment précis. La version la moins chère en terme de matériel pour le calcul du PGCD est donc la variante M3 qui est aussi rapide que la variante L5, cette dernière étant beaucoup plus chère. Pour le calcul du PGCD étendu, l'approche d'EUCLIDE est à la fois plus rapide et moins chère en matériel. Notre choix s'est porté sur la

variante M3 qui présente le meilleur compromis coût/performance. Les détails du circuit implantant cette approche feront l'objet du chapitre suivant.

CHAPITRE VI

CIRCUIT PGCD

VI.1. INTRODUCTION

Le choix de la méthode à implanter étant l'approche parallèle avec poids forts en tête, nous allons à présent effectuer l'étude du circuit conçu afin de calculer le PGCD et les coefficients de Bezout de nombres pouvant atteindre 2048 bits et plus. Ce circuit est constitué :

1- d'une partie opérative utilisant la notation redondante, mais dont la tête utilise la notation en complément à deux,

2- d'une seconde partie opérative pour comparer le nombre de bits de A et de B utilisant la notation binaire non redondante,

3- d'une pile LIFO contenant des opérations assurant le calcul des coefficients de Bezout.

Un autre circuit présentant la partie contrôle pour gérer le circuit PGCD, ainsi que d'autres circuits, sera implanté sur une carte destinée au traitement du signal. Cette partie contrôle ne sera pas décrite dans ce document mais qui fera prochainement l'objet d'un sujet de thèse.

VI.2. ETUDE DE LA PARTIE OPERATIVE 2048 BITS

Cette partie (figure VI.1), étant conçue pour traiter en parallèle de très grands nombres, occupe une surface importante, ce chemin de données n'étant en fait qu'un assemblage de 2048 cellules toutes identiques, une cellule par bit traité ou tranche de bit.

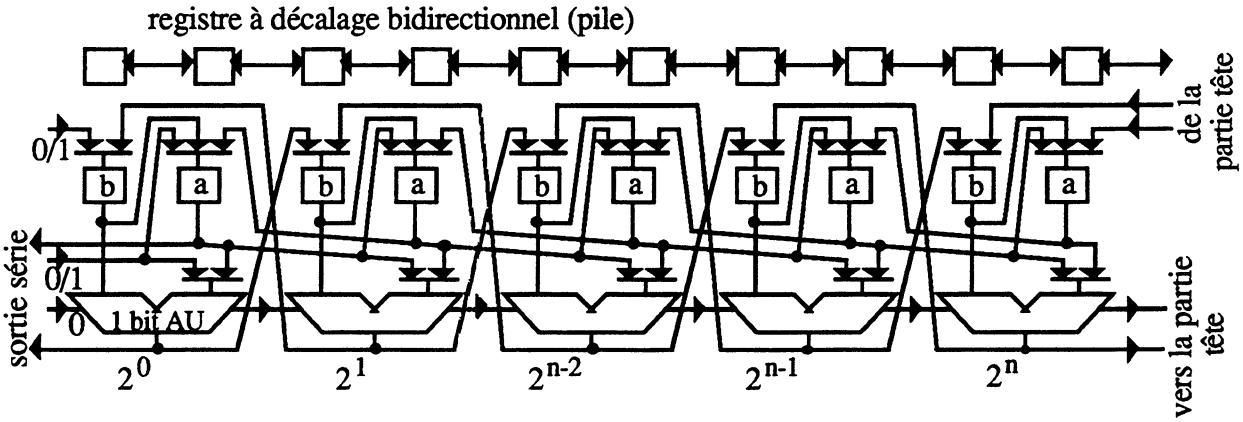


Figure VI.1 : partie opérative 2048 bits

VI.2.1. Blocs constitutifs d'une tranche de bit

Nous présentons dans cette section les différents blocs formant une tranche de bit réalisée en dessin au micron, composée de registres maître-esclave, d'un multiplieur, d'un additionneur 1 bit redondant et de multiplexeurs. Tous les transistors ont la taille minimale W/L = 4µm/1.6µm pour la technologie 1.5 µm.

VI.2.1.1. Registre ou point mémoire

Chaque chiffre en notation redondante représente deux bits. Si nous voulons sélectionner et mémoriser deux opérandes A et B venant de droite ou de gauche (figure VI.1), nous devons obtenir dans une tranche de bit quatre multiplexeurs et quatre points mémoires. La figure VI.2 montre le schéma électrique du deux points mémoire de B avec deux multiplexeurs en entrées. La sélection de l'entrée se fait par une commande φ1bg ou φ1bd et la mémorisation par φ2. La commande reset est utilisée pour la remise à zéro du registre.

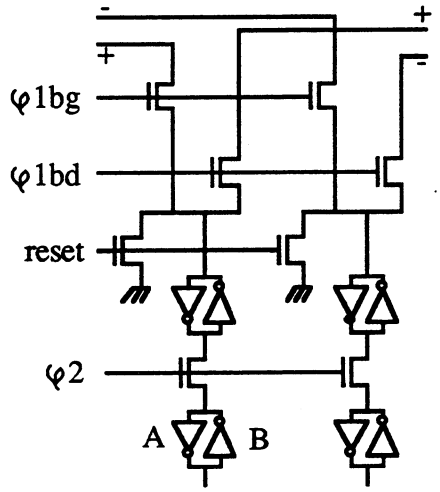


Figure VI.2 : schéma électrique du deux points mémoire de B

Pour ne pas obtenir le problème d'écriture d'information, les inverseurs A ont leur transistors de taille minimale W/L = 4µm/1.6µm par contre les dimensions des inverseurs B sont W/L = 2µm/5.6µm.

VI.2.1.2. Multiplieur

En notation redondante, pour obtenir l'opposé d'un nombre, il suffit de complémenter les deux bits constituant chaque chiffre ou encore de les permuter. Ceci peut être réalisé en utilisant

l'entrée d'un multiplieur de deux chiffres redondants (figure VI.3) en connectant l'autre à une commande p (p^+, p^-).

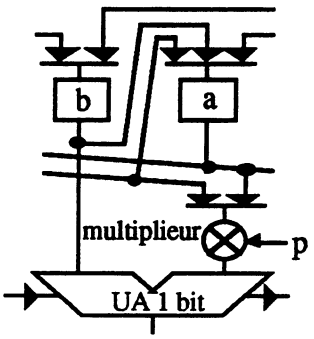
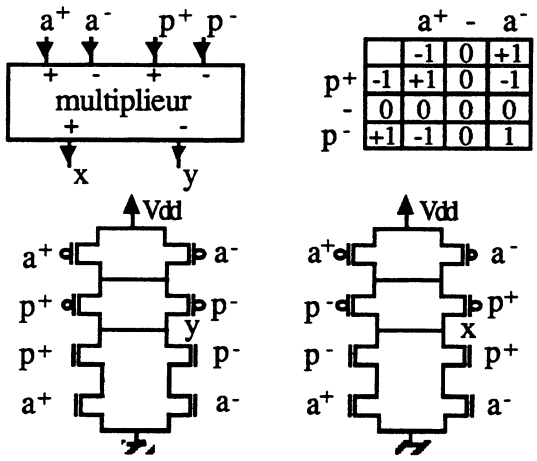


Figure VI.3 : connection du multiplieur avec l'UA 1 bit

Dans l'algorithme de calcul du PGCD, les opérations que doit effectuer l'UA sont (B + A), (B - A) et (B + 0) car les multiplications (respectivement divisions) par deux ne sont que de simples décalages à gauche (respectivement à droite). La figure VI.4 montre le schéma électrique ainsi que la table de vérité du multiplieur. En fonction des commandes p+ et p-, nous obtenons en sortie du multiplieur soit A (addition B + A), soit - A (soustraction B - A), soit 0 (opération B + 0).



	a^+	-	a^-
	-1	0	+1
p^+	-1	+1	0	-1
-	0	0	0	0
p^-	+1	-1	0	1

Figure VI.4 : schéma électrique et table de vérité du multiplieur

VI.2.1.3. PPM et additionneur redondant d'un bit

L'additionneur redondant permet d'éviter la propagation de retenues. Le traitement des nombres par les poids forts ou les poids faibles devient possible. Cet additionneur est montré à

la figure VI.5 pour additionner deux chiffres signés. La manière avec laquelle ce type de nombre est additionné est donnée à la table I.1 présentée dans le chapitre I §I.I.2.

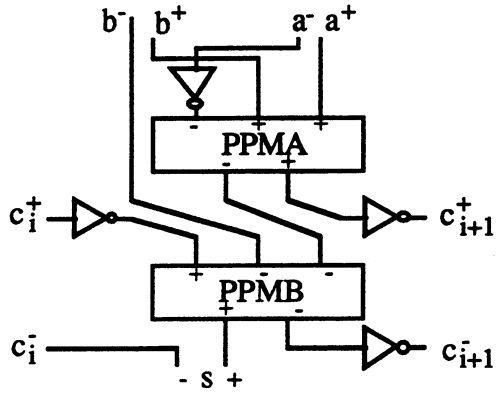
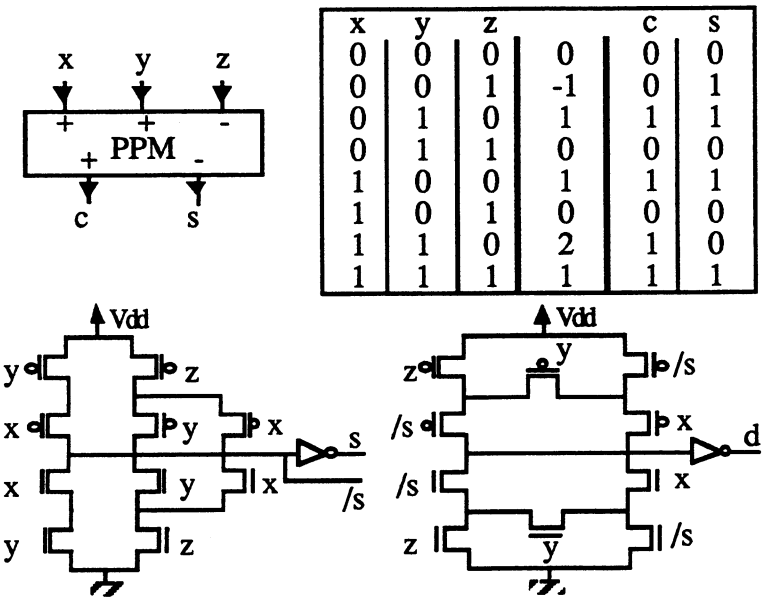


Figure VI.5 : additionneur redondant d'un bit

Cependant, il est possible d'éliminer les inverseurs du milieu de la figure, car quand il y a plus d'un bit, les inverseurs s'annulent. Deux PPM ont donc été conçus : l'un, PPMA, sans inverseur en sortie et l'autre PPMB sans inverseur en entrée. Cela conduit à une faible quantité de transistors. Cependant, pour que l'opération soit correcte, il est nécessaire de connecter en première entrée c^+ à 5V et la dernière sortie à un inverseur, avec la sortie finale prise à partir de l'inverseur. La figure VI.6 montre la table de vérité et le schéma électrique de l'additionneur PPM constituant l'additionneur redondant d'un bit.



x	y	z		c	s
0	0	0	0	0	0
0	0	1	-1	0	1
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	2	1	0
1	1	1	1	1	1

Figure VI.6 : schéma électrique et la table de vérité du PPM

VI.2.2. Plan de masse et routage d'une tranche de bit

Nous présentons le plan de masse et routage (figure VI.7) permettant d'obtenir une meilleure disposition des différents blocs constituant une tranche de bit (figure VI.1). Afin de simplifier au maximum le routage entre ces blocs, nous avons disposé en haut quatre multiplexeurs et quatre registres maître-esclave, au milieu deux multiplexeurs et un multiplieur, et en bas un additionneur redondant constitué du PPMA et du PPMB montés en cascade.

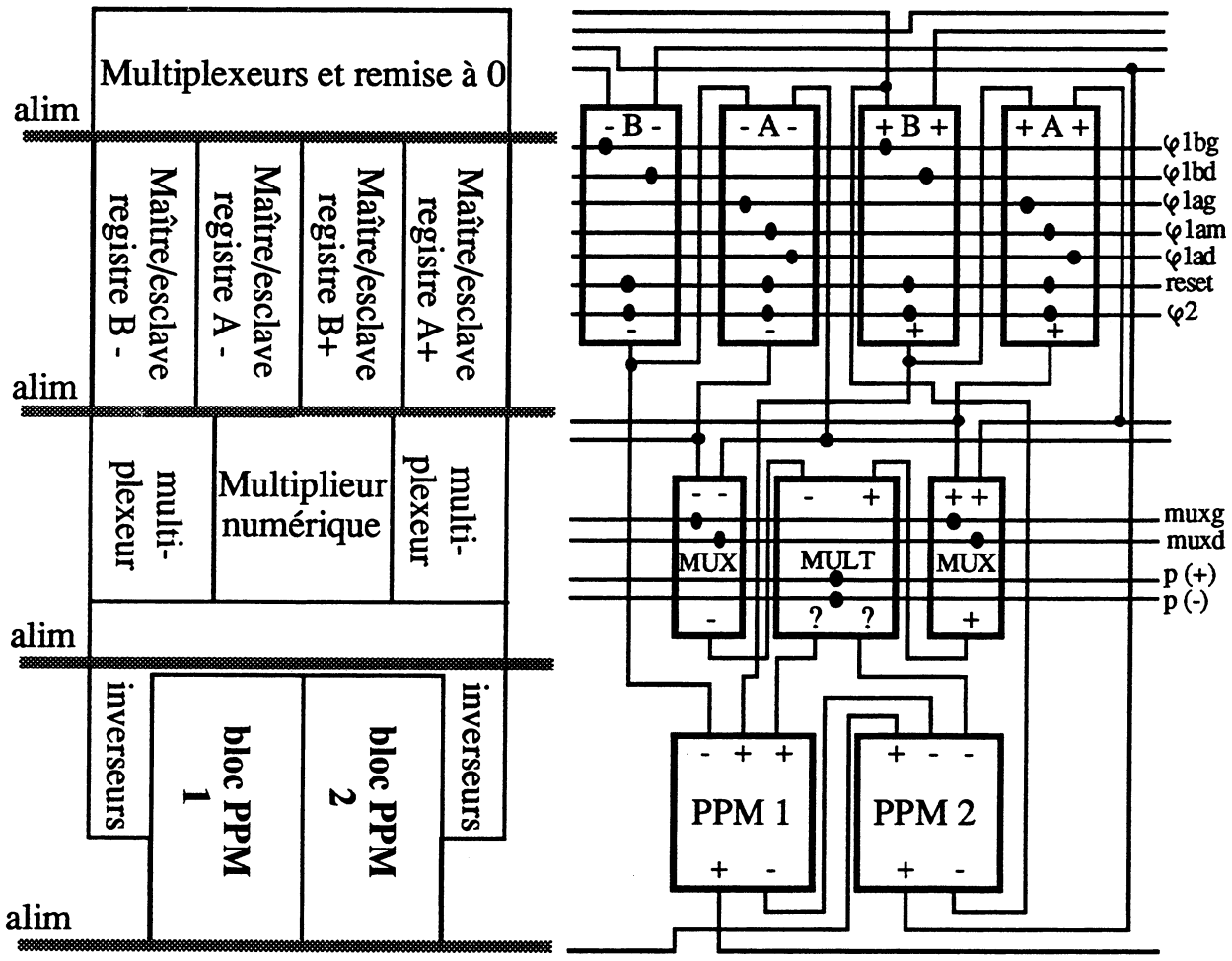


Figure VI.7 : plan de masse et routage d'une tranche de bit

VI.2.3. Réalisation d'une tranche de bit

La figure VI.8 représente une tranche de bit réalisée en dessin au micron. Tous les transistors ont la taille minimale W/L = 4µm/1.6µm.

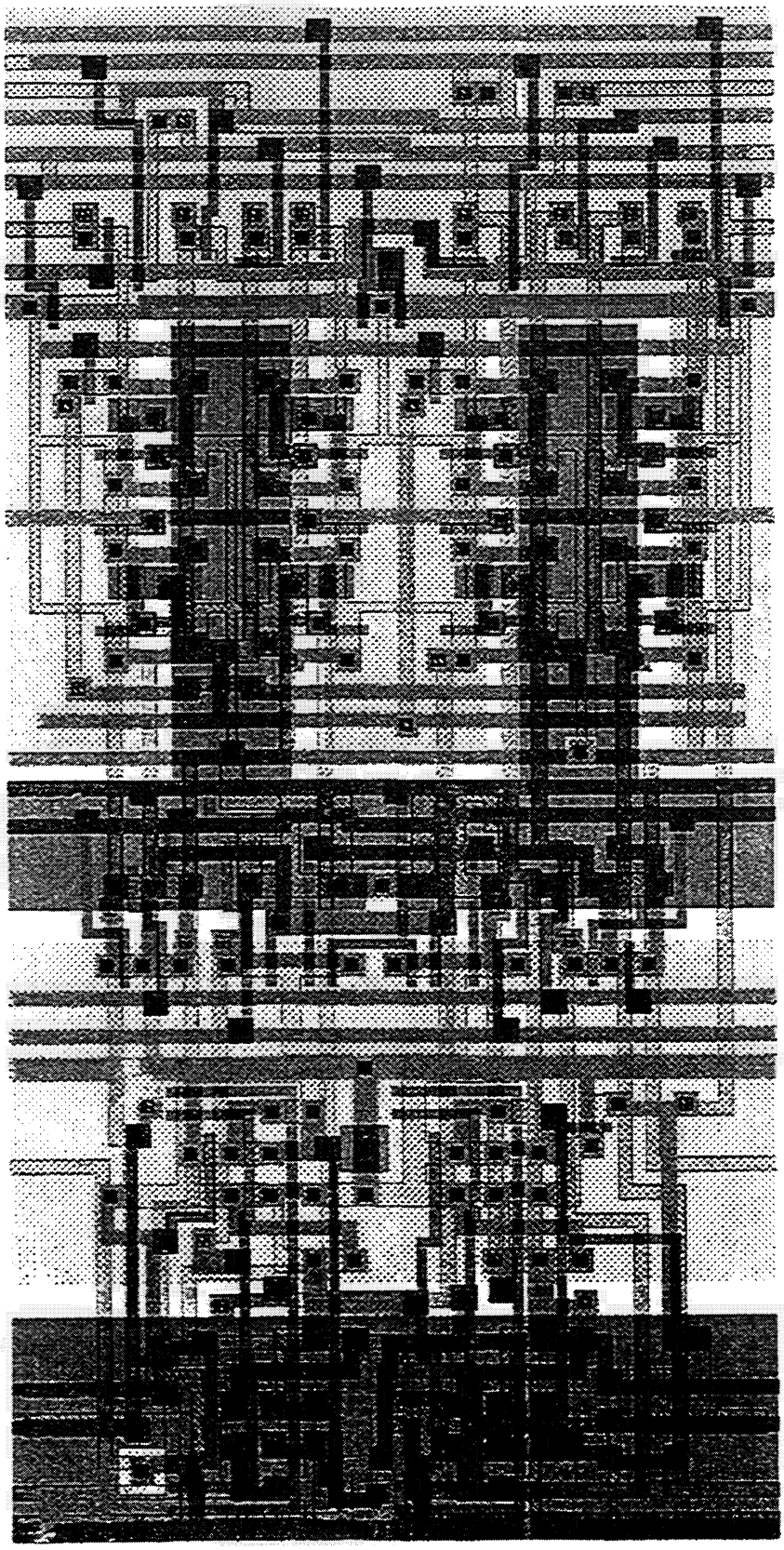


Figure VI.8 : layout d'une tranche de bit

VI.3. ETUDE DE LA PARTIE OPERATIVE 4 BITS OU TETE

L'approche que nous avons adoptée, test des poids forts, nécessite une représentation semi-redondante des nombres dont la tête, formée de quatre bits de poids le plus fort, utilise la notation en complément à deux.

L'intérêt de la notation en complément à deux est que l'on ne s'occupe pas des signes ; nous additionnons les nombres, bit de signe compris, comme s'ils étaient non signés. Pour des raisons de normalisation, le premier bit de μ_a et μ_b doit être non nul. Mais, comme il y a une propagation de la retenue pour normaliser les μ, chaque cycle de l'algorithme dure plus longtemps. Ainsi, k étant fixé à 3, μ_a et μ_b doivent toujours être compris dans l'intervalle [- 7,- 4] U [4 , 7]. Pour certaines valeurs de μ_a et μ_b, l'algorithme d'EUCLIDE peut ne pas converger, ces valeurs correspondant à abs(μ_a) = 7 et abs(μ_b) = 4. A chaque cycle du PGCD, les valeurs μ_a et μ_b sont testées.

La tête du circuit doit réaliser les mêmes opérations que la partie redondante (figure VI.9), mais elle doit, en plus, traiter des bits dans les deux notations. Par exemple, lorsque l'opération (B+A)*2 ⇒B est exécutée, nous voyons bien que le résultat de la dernière cellule de la partie redondante est décalé vers la tête. En fait, d'autres valeurs sont aussi décalées, telles que B lors de l'exécution de l'algorithme du PGCD, et A lors de l'exécution de l'algorithme de Bezout et la retenue r.

Le circuit formant la tête montré dans la figure VI.10 a été réalisé à partir de cellules de la bibliothèque ES2.

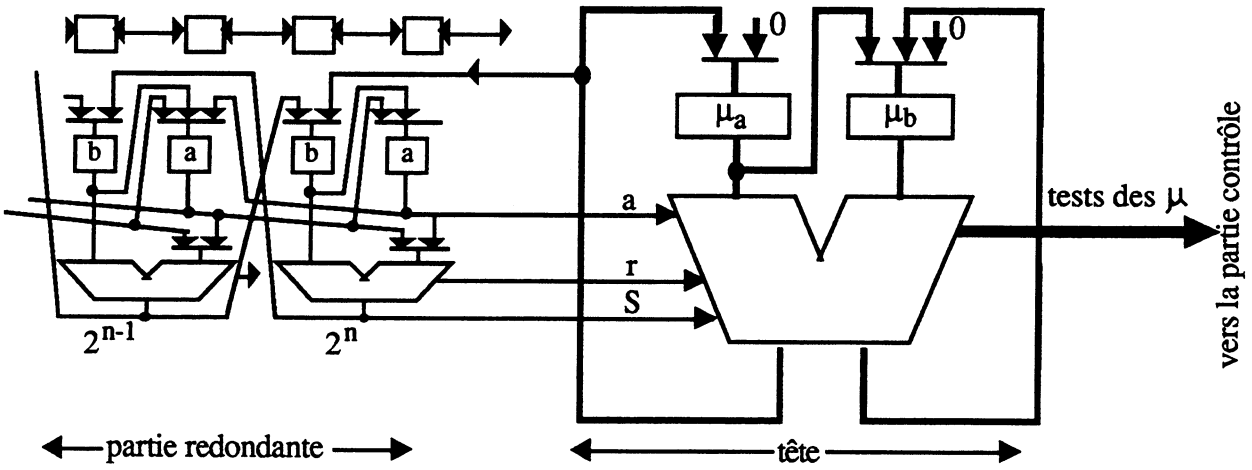


Figure VI.9 : partie redondante et tête

Nous avons tracé des traits en gras pour indiquer qu'ils concernent tous les bits des μ. La figure VI.10 présente le schéma structurel de la tête. Cette dernière est constituée d'additionneurs, de multiplexeurs, de bascules D et de portes logiques.

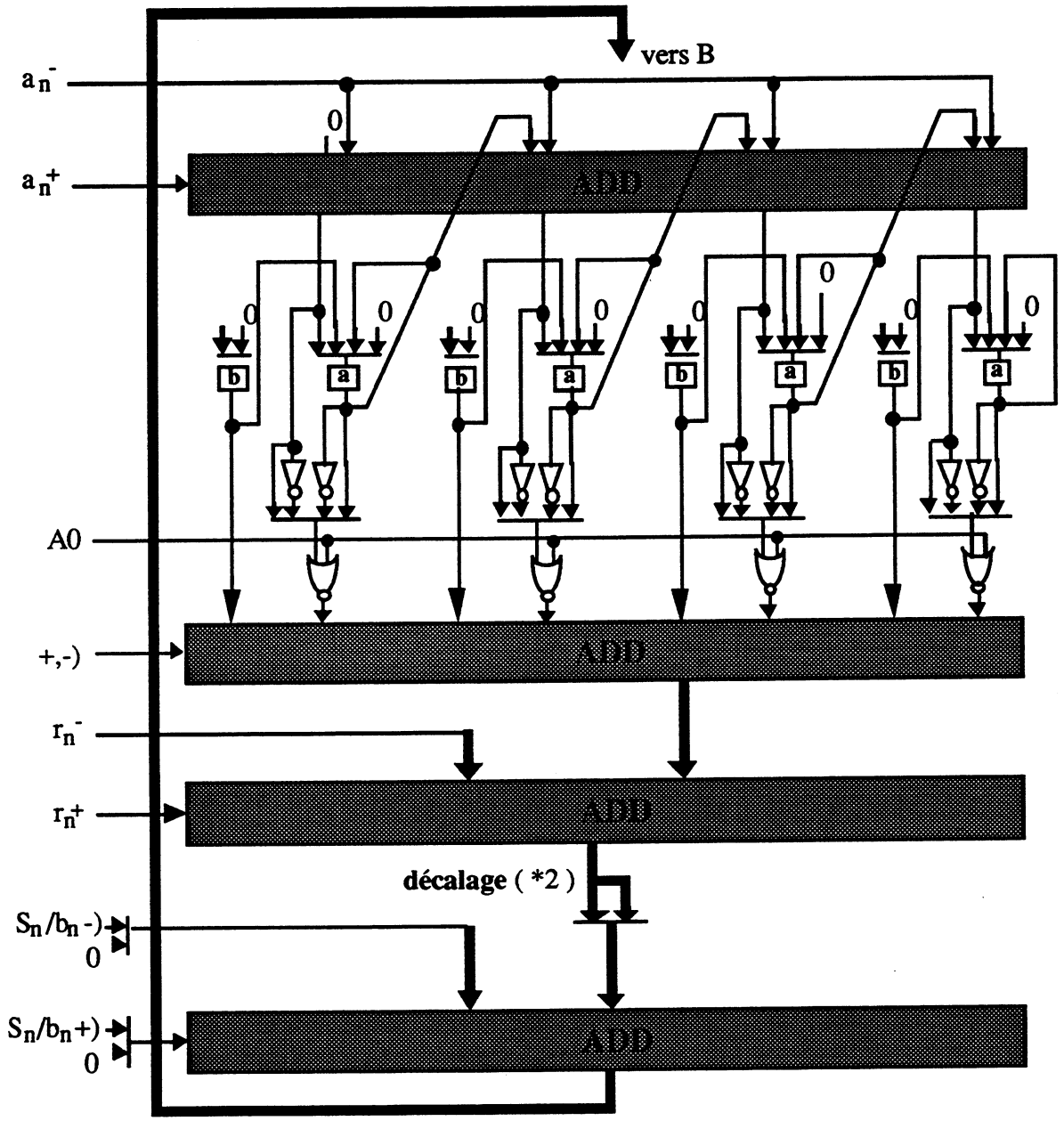
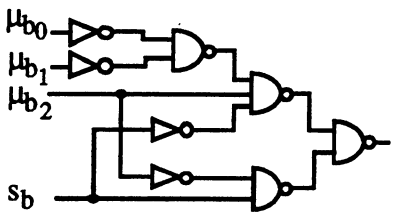


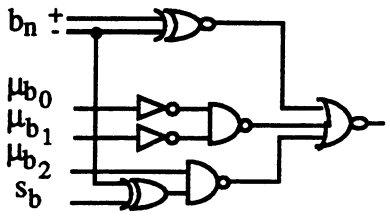
Figure VI.10 : structure de la partie opérative 4 bits ou tête

A chaque cycle du PGCD, les valeurs de μ_a et μ_b sont testées, $(b_n * \mu_b) = 4$, abs $(\mu_b) > 4$, $\mu_a * \mu_b < 0$ et abs (μ_a) > abs (μ_b) . Les schémas (figure VI.11) ont été proposés pour chacune de ces conditions.

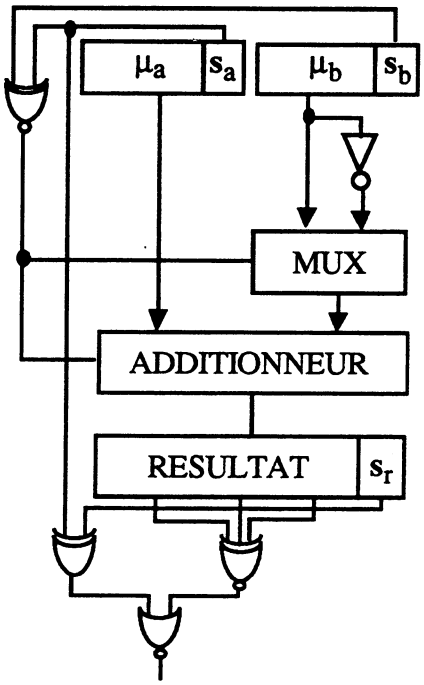


abs (μ_b) > 4

$\mu_a * \mu_b < 0$



$b_n * \mu_b = 4$



abs (μ_a) > abs (μ_b)

s_a	s_b	opération	$\mu_{b_0}+\mu_{b_1}+\mu_{b_2}$	s_r	$\lvert\mu_a\rvert > \lvert\mu_b\rvert$
0	0	-	0	quelconque	non
0	1	+			
1	0	+			
1	1	-			
0	0	-	1	0	oui
				1	non
0	1	+		0	oui
				1	non
1	0	+		0	non
				1	oui
1	1	-		0	non
				1	oui

Figure VI.11 : tests sur les μ

VI.4. ETUDE DE LA PARTIE OPERATIVE 12 BITS

Le déroulement de l'algorithme du PGCD ne peut se faire sans la connaissance, après chaque cycle, des valeurs δ_a et δ_b car de nombreux tests et opérations concernent les tailles des nombres A et B : $\delta_b > 0$, $\delta_a > \delta_b$, $\delta_a = \delta_b$, $\delta_b = \delta_b - 1$, $\delta_b = \delta_b - 2$ et échange (δ_a, δ_b). Pour les réaliser, nous avons conçu une deuxième partie opérative sur 12 bits puisque la taille maximale des nombres A et B est 2048, qui s'écrit 100000000000 en notation binaire.

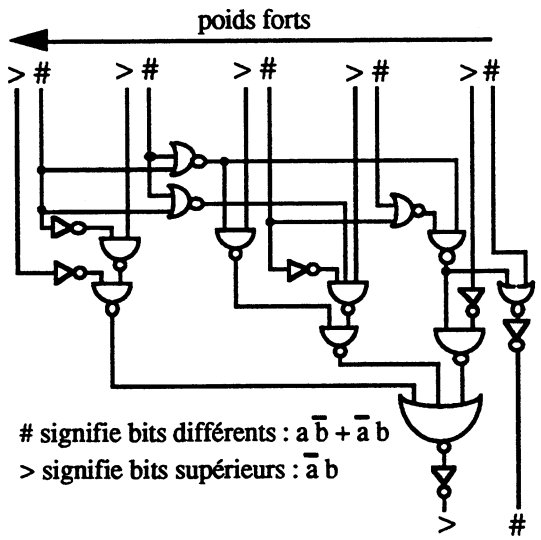
La partie opérative 12 bits (figure VI.15) est constituée :

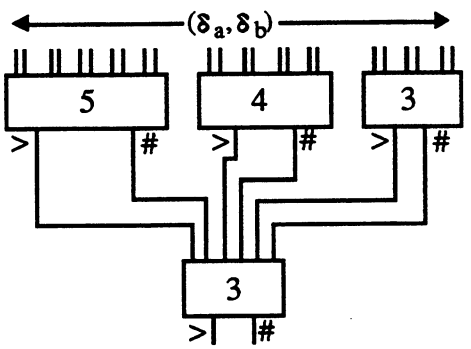
- d'un registre N qui contient initialement la valeur du nombre de chiffres maximum (2048) permettant d'assurer, en le decrémentant jusqu'à la valeur zéro que toutes les données sont entrées en série dans les registres A, B, δ_a et δ_b. La valeur zéro est alors obtenue à la sortie du comparateur ($\delta_b > 0$),

- d'un compteur de cycles dont la valeur du nombre de cycles comptés après l'exécution de l'algorithme du PGCD sera utilisée pour la recherche des coefficients de Bezout,

- de deux comparateurs 12 bits. L'un décomposé en trois parties de tailles décroissantes pour permettre d'obtenir un comparateur rapide (la plus petite partie recevant les poids faibles et la plus grosse les poids forts) exécutant le test $\delta_a > \delta_b$ et $\delta_a = \delta_b$ (figure VI.12) ; et l'autre (figure VI.13) exécutent le test $\delta_b > 0$,

- d'un décrémenteur rapide 12 bits (figure VI.14) dont la valeur du nombre de bits δ_b est décrémentée de -1 ou -2 pendant l'exécution de l'algorithme d'EUCLIDE et le résultat du PGCD est donné lorsque $\delta_b = 0$.





Le comparateur de gauche est donné sur 5 bits. Nous obtenons des comparateurs sur 3 ou 4 bits en le dégradant. L'ensemble forme un comparateur 12 bits.

Figure VI.12 : comparateur rapide 12 bits ($\delta_a > \delta_b$ et $\delta_a = \delta_b$)

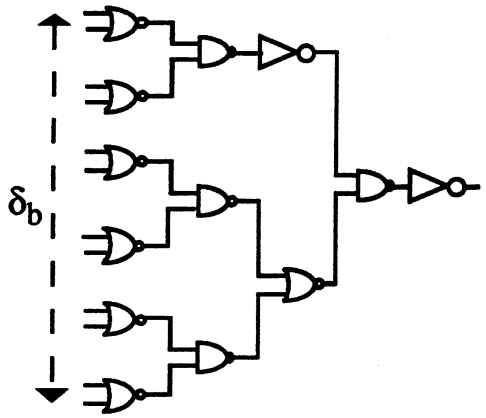
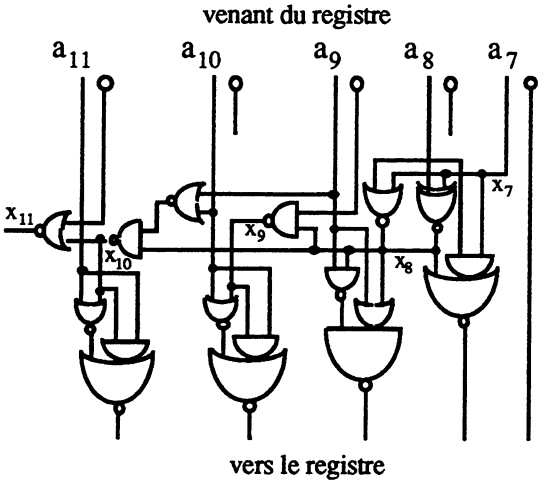


Figure VI.13 : comparateur 12 bits $\delta_b > 0$



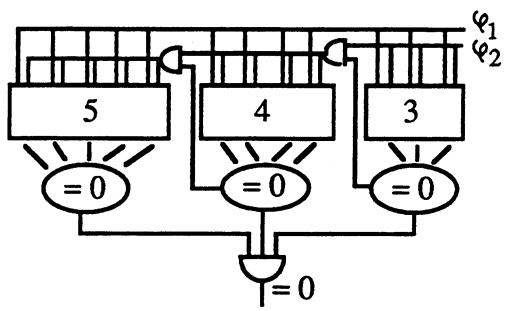
$x_7 = \overline{a_7 = 0}$

$x_8 = a_7, a_8 = 0$

$x_9 = \overline{a_7, a_8, a_9 = 0}$

$x_{10} = \overline{a_7, a_8, a_9, a_{10} = 0}$

$x_{11} = a_7, a_8, a_9, a_{10}, a_{11} = 0$



Le décrémenteur de gauche est donné sur 5 bits, les autres 4 et 3 bits s'en déduisent par simplification. x_{11} est connecté vers l'horloge des suivants pour les 4 et 3 bits. L'ensemble forme un décrémenteur 12 bits. L'horloge φ_2 n'est transmise vers les poids forts que si les poids faibles sont zéro. φ_1 est toujours transmise.

Figure VI.14 : décrémenteur rapide 12 bits

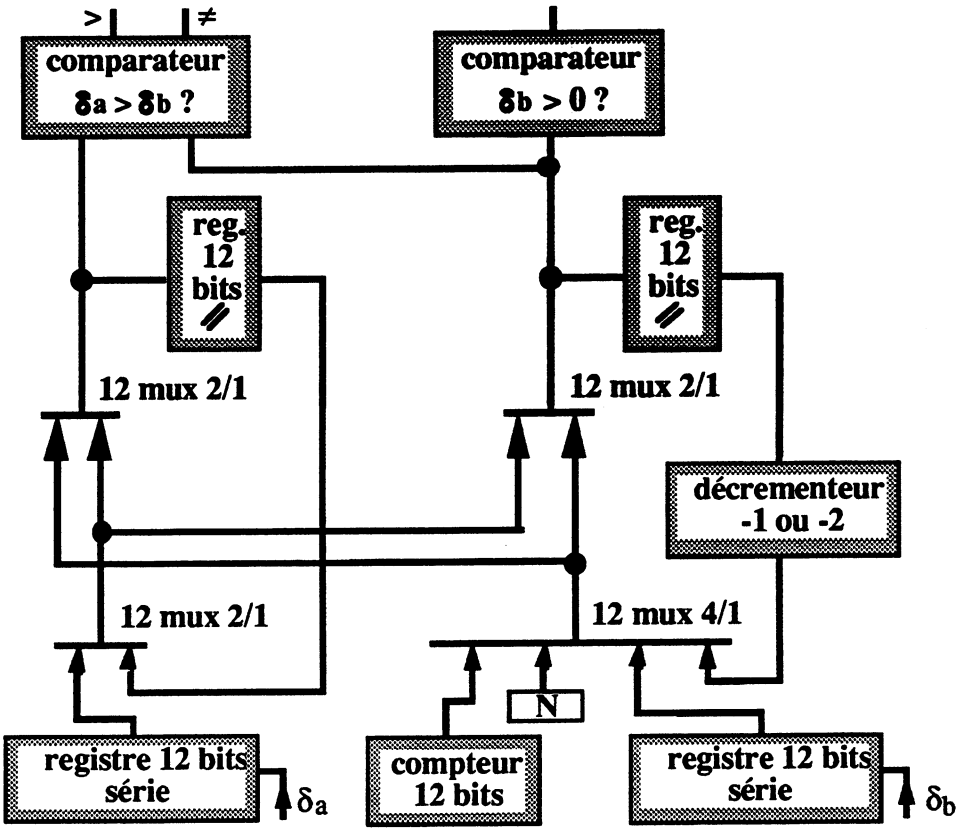


Figure VI.15 : partie opérative 12 bits

VI.5. ETUDE DE LA PILE LIFO

Une pile LIFO est nécessaire pour stocker les résultats des opérations effectuées lors du calcul du PGCD que sont l'addition, la soustraction, le décalage, l'échange et la normalisation. Nous comptons au maximum 2048 opérations qui sont codées sur 3 bits. Pour calculer les coefficients de Bezout, il suffit d'exécuter les opérations stockées dans la pile dans le sens inverse.

L'algorithme de la table III.6 du chapitre III exige d'adresser toutes les cellules de la RAM ; l'ordre d'adressage des cellules n'est pas significatif (l'algorithme est fonctionnel, et donc indépendant de l'implantation topologique de la RAM). Cependant, nous adressons la RAM pour stocker les différentes opérations dans un ordre aléatoire jusqu'à la fin de calcul du PGCD. A ce moment là, nous commençons à adresser cette même RAM dans l'ordre inverse du précédent pour le calcul des coefficients de Bezout.

Etant donné que l'ordre d'adressage importe peu, nous utilisons comme générateur d'adresse un LFSR (registres à décalage à rebouclage linéaire) Up/Down [NIC 85] qui est plus compact qu'un compteur Up/Down.

Un LFSR Up/Down à Ou-Exclusif externe et de longueur n peut être réalisé comme suit :

- Durant la fonction Up, le circuit est configuré comme un simple LFSR à Ou-Exclusif externe et de longueur n,
- Durant la fonction Down, la direction de décalage est inversée (fonction Up : décalage à droite, fonction Down : décalage à gauche). La boucle de retour de la fonction Down est dérivée de la boucle de retour de la fonction Up de la façon suivante :

a- Si la sortie de la cellule C_k est utilisée dans la boucle de retour de la fonction Up, alors la sortie de la cellule C_m ($m = (k+1) \bmod n$) est utilisée dans la boucle de retour de la fonction Down,

b- La sortie de la cellule utilisée dans la boucle de retour de la fonction Down est utilisée comme entrée de la $n^{ième}$ cellule.

Un exemple d'un LFSR Up/Down de longueur 7 utilisé dans le prototype du circuit PGCD est présenté figure VI.16. Ce LFSR Up/Down est dérivé d'un LFSR normal (configuration Up) correspondant au polynôme primitif caractéristique $x^7 + x^3 + 1$. Le fait que le polynôme caractéristique soit primitif, permet de générer tous les états sauf l'état 000...0. Cet dernier état est également généré en insérant une porte NOR avec (n - 1) entrées dans la boucle de retour Up et Down. Une pile LIFO est composée d'une mémoire RAM (128 mots * 3 bits) et d'un LFSR Up/Down de longueur 7.

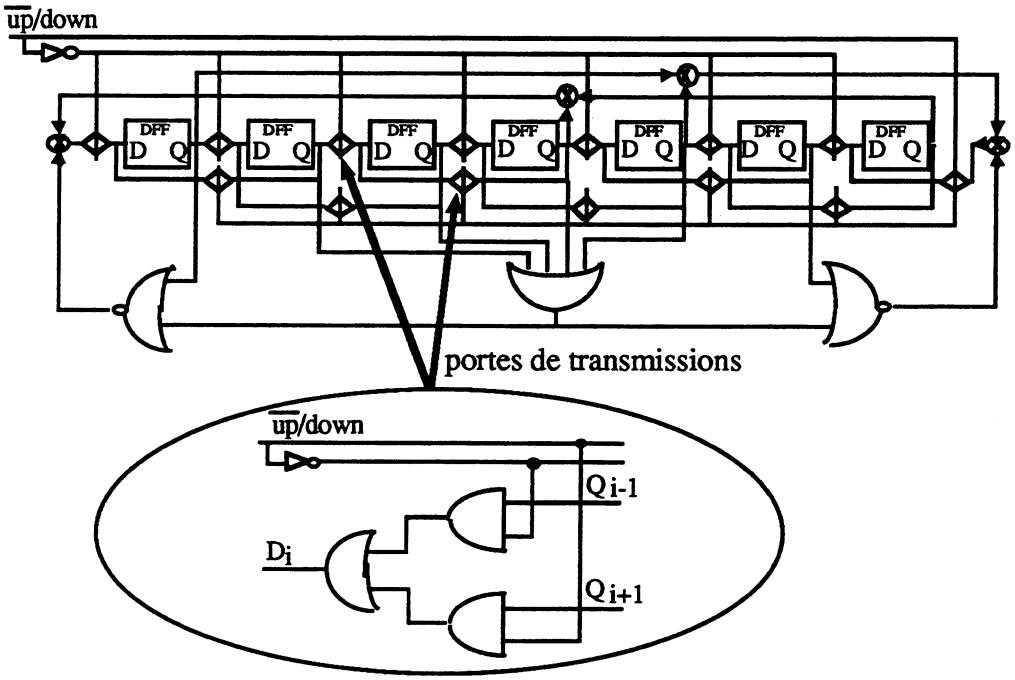


Figure VI.16 : LFSR Up/Down de longueur 7

Le schéma de la pile LIFO (figure VI.17) a été réalisé à partir de cellules standard.

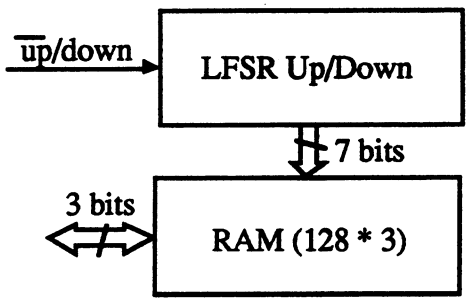


Figure VI.17 : pile LIFO

VI.6. REALISATION DU CIRCUIT PGCD

Le circuit réalisé (figure VI.18) comporte cinq parties [BGK 91, BGU 92] :

1- Un chemin de données 2048 bits en notation redondante composé de tranches de bits identiques contenant 156 transistors par bit. Les opérations exécutées dans chaque tranche sont (B + A), (B - A), (B + 0), un simple décalage à gauche, et échange (A, B). L'unité arithmétique 1 bit de chaque tranche traite les nombres A et B en utilisant la notation redondante. Ceci a pour résultat une sortie rapide sans propagation de retenue et avec un coût matériel faible.

2- Un chemin de données 4 bits, connu sous le nom de "tête" utilise la notation complément à deux et, après normalisation, le premier bit de μ_a et μ_b est toujours différent de zéro. La tête doit pouvoir exécuter les mêmes opérations que le chemin de données de 2048 bits en notation redondante mais devrait aussi être capable de traiter les nombres dans les deux notations.

3- Une pile LIFO pour stocker les séquences d'opérations de calcul du PGCD. Le couple (R, S) est simple à obtenir en exécutant les opérations correspondantes dans le sens inverse.

4- Un chemin de données 12 bits détermine la continuité de l'algorithme de calcul du PGCD. Cet algorithme est incapable de continuer sans la connaissance, après chaque cycle, des valeurs de δ_a et δ_b, car les valeurs de test et les opérations dépendent de la taille des nombres A et B.

5- Une partie contrôle, qui est une machine d'états finis, implantée pour séquencer l'algorithme.

La première partie est réalisée en dessin au micron afin d'obtenir une grande densité, et les quatre dernières parties sont réalisées en cellules standard.

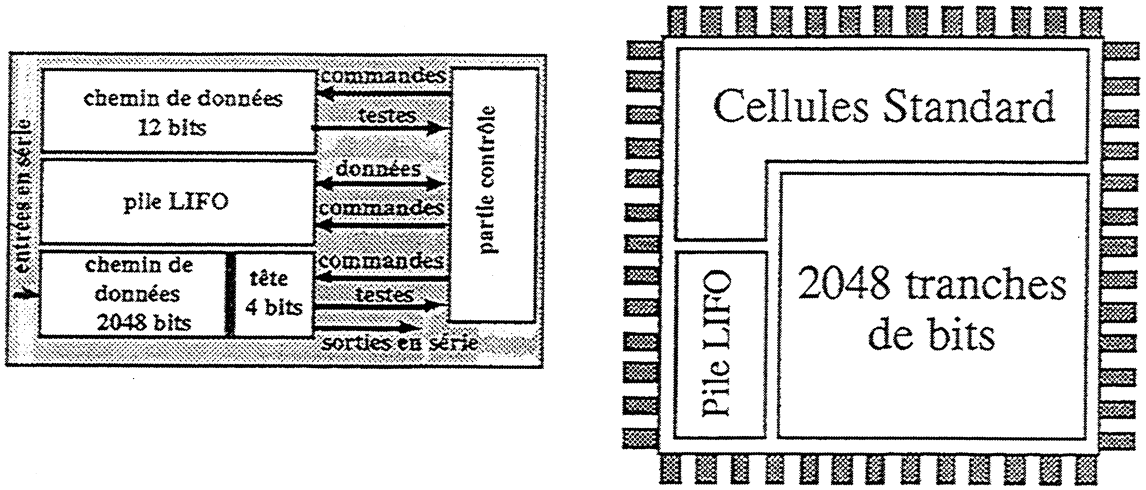


Figure VI.18 : description du circuit

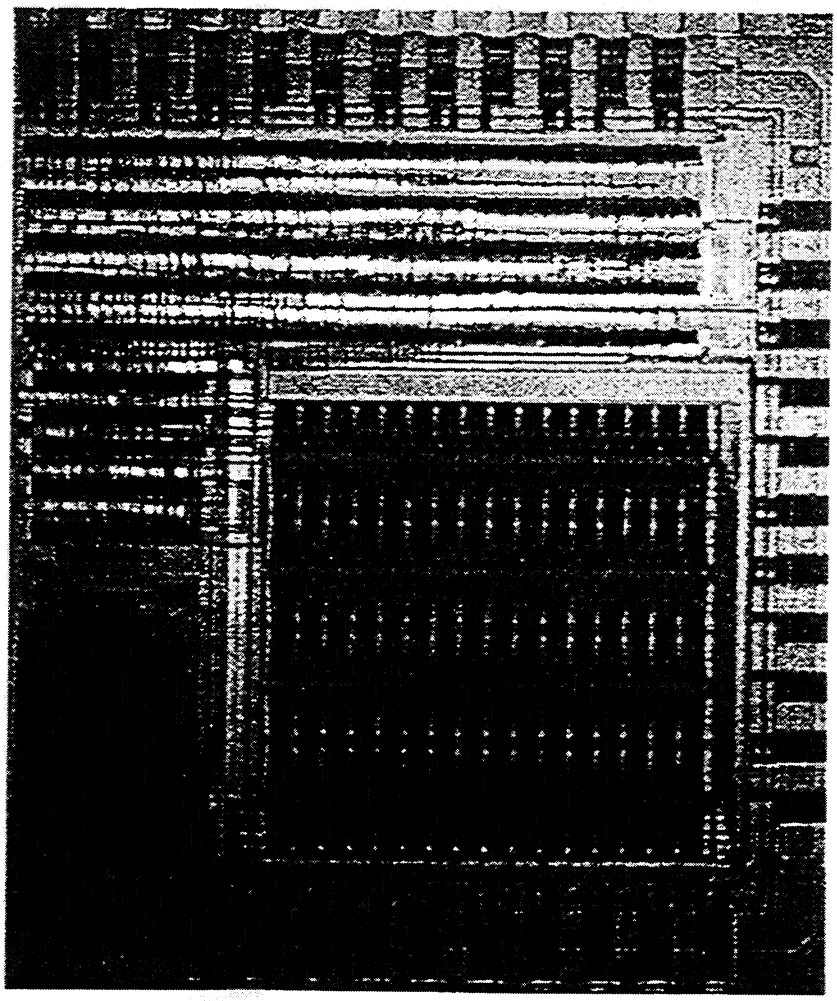


Figure VI.19 : photo du circuit

En utilisant une technologie 1.5 µm, 2 couches de métal et une très grande régularité, nous obtenons pour le circuit 2048 bits une complexité de l'ordre de 700.000 transistors avec une densité de 5000 transistors/mm^2, ce qui donne une surface totale de 140 mm^2. Un prototype du circuit de 128 bits, occupant une surface de 21 mm^2 est réalisé au Service Eurochip. La simulation a montré que la fréquence maximale de travail est limitée à 25 Mhz. Au retour de la fabrication du circuit intégré (figure VI.19), nous avons testé ce dernier, en utilisant une série des nombres d'entrées, par exemple, ceux utilisés dans la simulation du circuit (Annexe 1). Ce dernier ne fonctionne pas pour des raisons de forte consommation.

VI.7. CONCLUSION

Le circuit de calcul du PGCD et des coefficients de Bezout a été réalisé à partir d'un algorithme, l'approche parallèle avec test des poids forts, choisi par simulation parmi d'autres approches afin d'obtenir le meilleur compromis performance/coût en matériel. La performance est donnée par la manière de produire le résultat sous forme de nombre de bits en un cycle d'horloge, le coût est évalué en nombre de transistors pour traiter un bit dans le chemin de données.

Grâce aux nouvelles technologies submicroniques, nous pouvons espérer réaliser ultérieurement, le circuit 2048 bits sur une surface de 30 mm^2 environ.

CHAPITRE VII

CONCEPTION D'UN PROCESSEUR EUCLIDIEN EN LIGNE

VII.1. INTRODUCTION

Le deuxième circuit, décrit dans ce chapitre, implante le calcul en ligne de la distance euclidienne : $Z = \sqrt{X^2 + Y^2}$ en utilisant un système redondant de représentation des nombres (le même que dans les chapitres précédents). Nous rappelons qu'à chaque coup d'horloge et en faisant abstraction du délai initial, nous obtenons un bit du résultat. En d'autres termes, il est possible de donner des résultats sur 600 chiffres décimaux en environ 2048 coups d'horloge!

Ce circuit a été conçu en tranches de bits pour faciliter la duplication afin de varier la précision. C'est le nombre de tranches de bits qui détermine la précision de calcul de la fonction. Cette fonction a été implantée en utilisant deux circuits élévateurs au carré, un additionneur redondant et un extracteur de racine carrée. Cependant, il est possible de concevoir un circuit qui implante les opérateurs les plus efficaces pour ces opérations.

Ce chapitre est divisé en trois parties. La première partie résume le travail effectué pour développer, vérifier et modifier l'algorithme contenant la base de la structure de l'opérateur. La deuxième partie présente le développement et la simulation des circuits de l'opérateur. La dernière partie concerne la conception proprement dite d'un circuit de l'opérateur, et le résultat du test après fabrication.

Pour la réalisation présentée, le circuit se compose de 32 tranches dessinées au micron. Les nombres d'entrées et de sorties en série se présentent sous la forme de chiffres binaires signés.

VII.2. ALGORITHMES

La système utilisé pour la représentation des nombres est la notation redondante introduite dans le chapitre I §I.I.2. Cette notation est fondamentale pour les différents circuits de calcul de l'opérateur euclidien.

VII.2.1. Elévateur au carré

L'entrée et la sortie de ce circuit sont des nombres en chiffres signés. Les deux chiffres binaires constituant chaque chiffre signé arrivent en parallèle. Le registre-SP (Série-Parallèle) et le multiplieur-VD (Vecteur par Digit) sont également introduits.

Soit A défini par $\sum_{i=1}^{n} a_i*2^{-i}$ et $A_j = \sum_{i=1}^{j} a_i*2^{-i}$ et définissons :

$$\widetilde{A_0} = A_0 = 0$$
$$\widetilde{A_j} = A_j + \bar{a}_j * 2^{-(j+1)} = A_{j-1} + a_j * 2^{-j} + \bar{a}_j * 2^{-(j+1)}$$

La valeur de $\widetilde{A_j}$ est mémorisée dans un registre. L'opération est exécutée en mettant a_j dans la $j^{ième}$ position et $\bar{a}_j$ dans la $(j + 1)^{ième}$ position d'un registre qui mémorise A_{j-1}. Ce registre est connu sous le nom du registre-Série Parallèle (registre-SP). Il est chargé en série et ses contenus sont utilisés en parallèle sous la forme d'un vecteur.

Dans la figure VII.1, le fonctionnement du registre-SP est illustré. Le registre est chargé par les chiffres signés les plus significatifs. Il est aussi possible de charger le registre en premier par les chiffres signés les moins significatifs.

La récursion suivante génère les produits partiels de l'opération carré du nombre A :

$$\Delta A^2_0 = 0$$
$$\Delta A^2_j = 2 * (\Delta A^2_{j-1} + \widetilde{A_j} * a_j)$$

La valeur de $(\widetilde{A_j} * a_j)$ est calculée en multipliant les contenus du registre-SP par le chiffre a_j.

Le multiplieur-VD est composé de séries de multiplieurs élémentaires présentés dans le chapitre VI §VI.2.1.2.

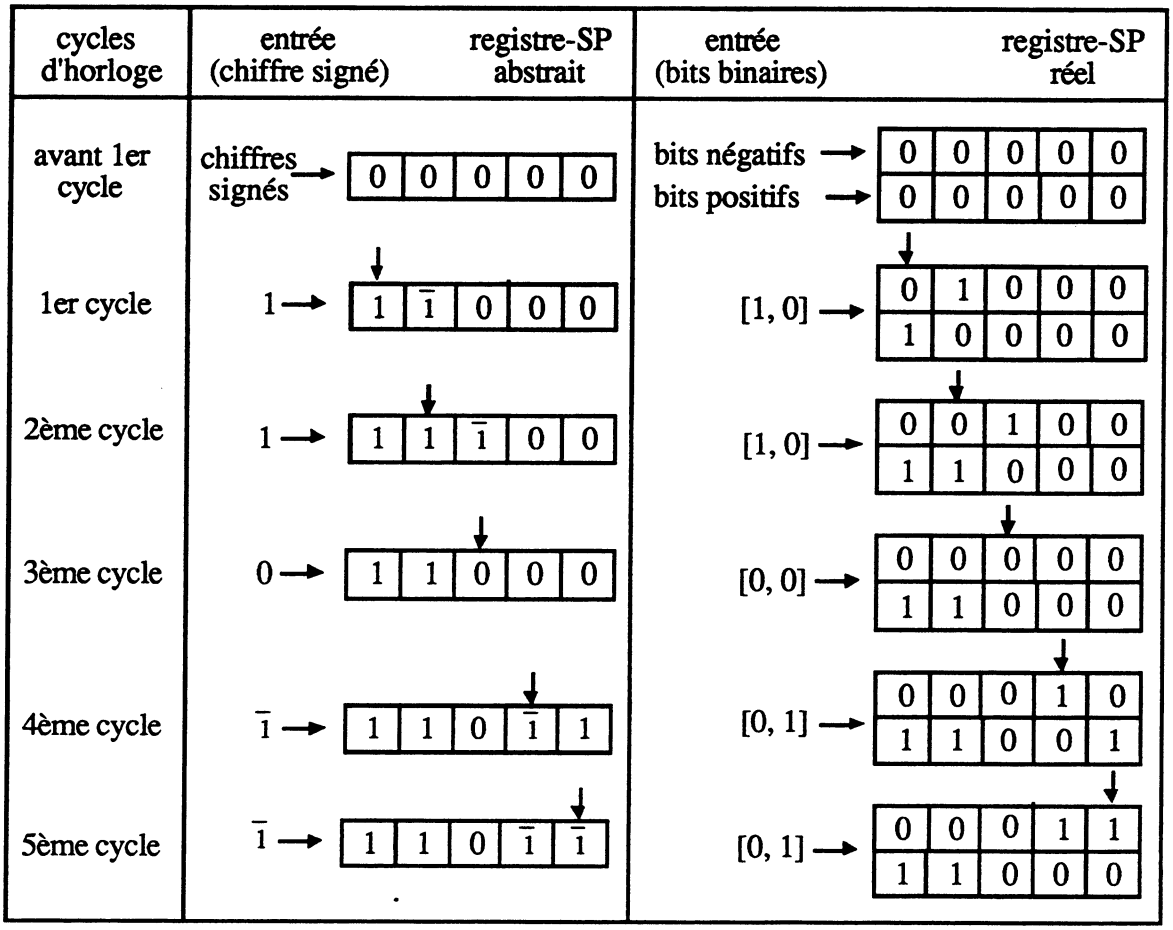


Figure VII.1 : le chargement d'un registre-SP

Le multiplieur-VD est composé d'un seul bloc. Il est réalisé en multipliant a_j par tous les chiffres signés formant le vecteur (figure VII.2).

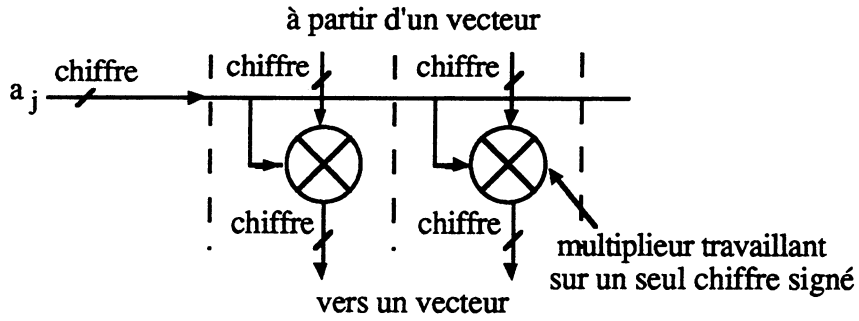


Figure VII.2 : multiplieur-VD pour un vecteur à deux chiffres signés

Le résultat de l'opération carré est formé dans un accumulateur. L'accumulateur construit le résultat complet dans un registre en ajoutant le produit partiel de l'entrée à la valeur qui, à

présent, existe dans le registre. Le résultat de cette addition sera remis, au retour, dans ce registre.

A chaque cycle d'horloge, le registre est décalé à gauche d'une place de telle façon qu'il est possible de sortir du chiffre du poids fort d'un résultat. Le chiffre le plus significatif de l'accumulateur est un résultat de sortie pour l'additionneur redondant.

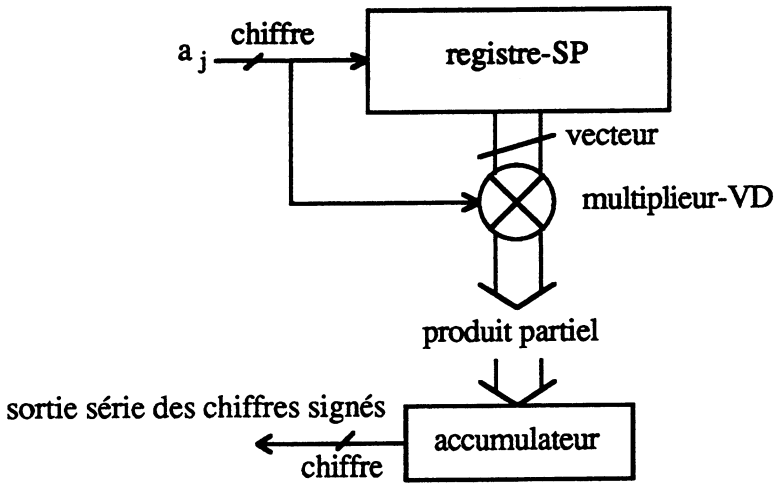


Figure VII.3 : élévateur au carré en ligne

L'élévateur au carré en ligne est montré figure VII.3 sous forme de blocs. Son fonctionnement général peut être mieux illustré en utilisant un exemple (voir tableau ci-après). Le fait qu'un chiffre du résultat ne soit pas prêt au début du premier cycle implique que ce circuit a un retard en ligne d'un cycle d'horloge. Le calcul sera fait en premier, par le chiffre le plus significatif, ce qui sera l'approche adoptée dans l'opérateur euclidien. Cependant, il est aussi facile d'adopter une approche par le chiffre le moins significatif.

Le registre-SP qui est utilisé dans l'élévateur au carré en ligne doit contenir une location de plus que la taille maximale de l'entrée du nombre pour obtenir le résultat correct ; c'est à dire (n + 1) locations si le plus grand nombre a n chiffres signés. Le poids du chiffre le plus significatif du résultat peut être déterminé à partir de 2^{2n} où n est le nombre de chiffres signés significatifs dans le nombre d'entrée.

$A = (13)^2 = (1\ 1\ 0\ 1)^2$

cycles d'horloge	entrée série	multiplieur -VD	contenu du registre-SP	produits partiels	sortie série chiffre signé	binaires	poids
0	0		00000	00000	-	-	-
1	1	x	$1\bar{1}000$ =	$1\bar{1}000$			
				$\mathbf{1}\bar{1}0000$	**1**	[1, 0]	128
2	1	x	$11\bar{1}00$ =	$11\bar{1}00$			
				$\mathbf{0}1\bar{1}000$	**0**	[0, 0]	64
3	0	x	11000 =	00000			
				$\mathbf{1}\bar{1}00000$	**1**	[1, 0]	32
4	1	x	$1101\bar{1}$ =	$1101\bar{1}$			
				$\mathbf{0}101\bar{1}$	**0**	[0, 0]	16
5				**1**	**1**	[1, 0]	8
6				**0**	**0**	[0, 0]	4
7				**1**	**1**	[1, 0]	2
8				$\bar{1}$	$\bar{1}$	[0, 1]	1

$1*128 + 1*32 + 1*8 + 1*2 + -1*1 = 169$

VII.2.2. Extracteur de racine carrée

L'entrée et la sortie de cet opérateur (figure VII.4) se font en série, chiffre de poids fort en premier. Il est nécessaire que le nombre d'entrées soit positif. Il doit aussi être normalisé. Cela exige d'assurer que le premier chiffre signé est 1. Ce dernier sera utilisé pour déterminer le poids des chiffres du résultat.

Cet opérateur fonctionne en respectant la relation suivante :

$$(z_i^2 - a_i) < \varepsilon$$

où z_i et $a_i \in \{ \bar{1}, 0, 1 \}$ et $\varepsilon \propto 2^{-i}$. Cela implique que $Z = \sqrt{A}$, où A est le nombre d'entrée et Z le résultat final. Cette technique est seulement valide si A est transmis chiffre de poids fort en premier.

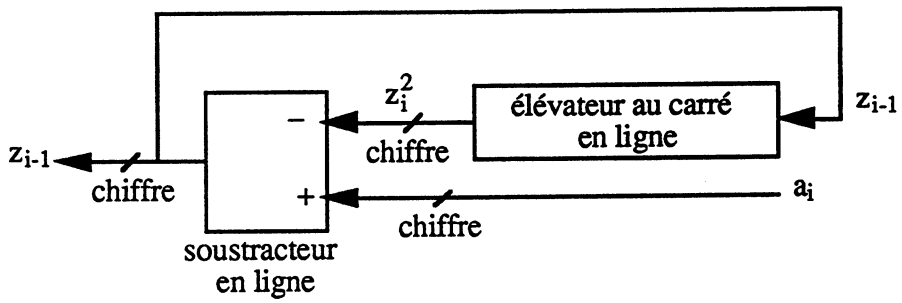


Figure VII.4 : extracteur de racine carrée

La valeur du résultat de chacun des chiffres signés est calculée par un soustracteur de la manière suivante :

si	$z_i^2 = a_i$	alors	$z_{i-1} = 0$
si	$z_i^2 < a_i$	alors	$z_{i-1} = 1$
si	$z_i^2 > a_i$	alors	$z_{i-1} = \bar{1}$

Le chiffre du résultat est utilisé par un élévateur au carré en ligne pour générer la valeur suivante de z_i^2 pour le soustracteur. C'est un processus itératif qui maintient la relation :

$$(z_i^2 - a_i) < \varepsilon$$

VII.2.3. Opérateur euclidien

L'opérateur euclidien (figure VII.5) accepte deux nombres en série, X et Y, composés de chiffres signés et génère la distance euclidienne de ces nombres. Le résultat est une sortie série de chiffres signés. Il est aussi normalisé. L'opérateur travaille chiffre de poids fort en premier pour le calcul de la racine carrée.

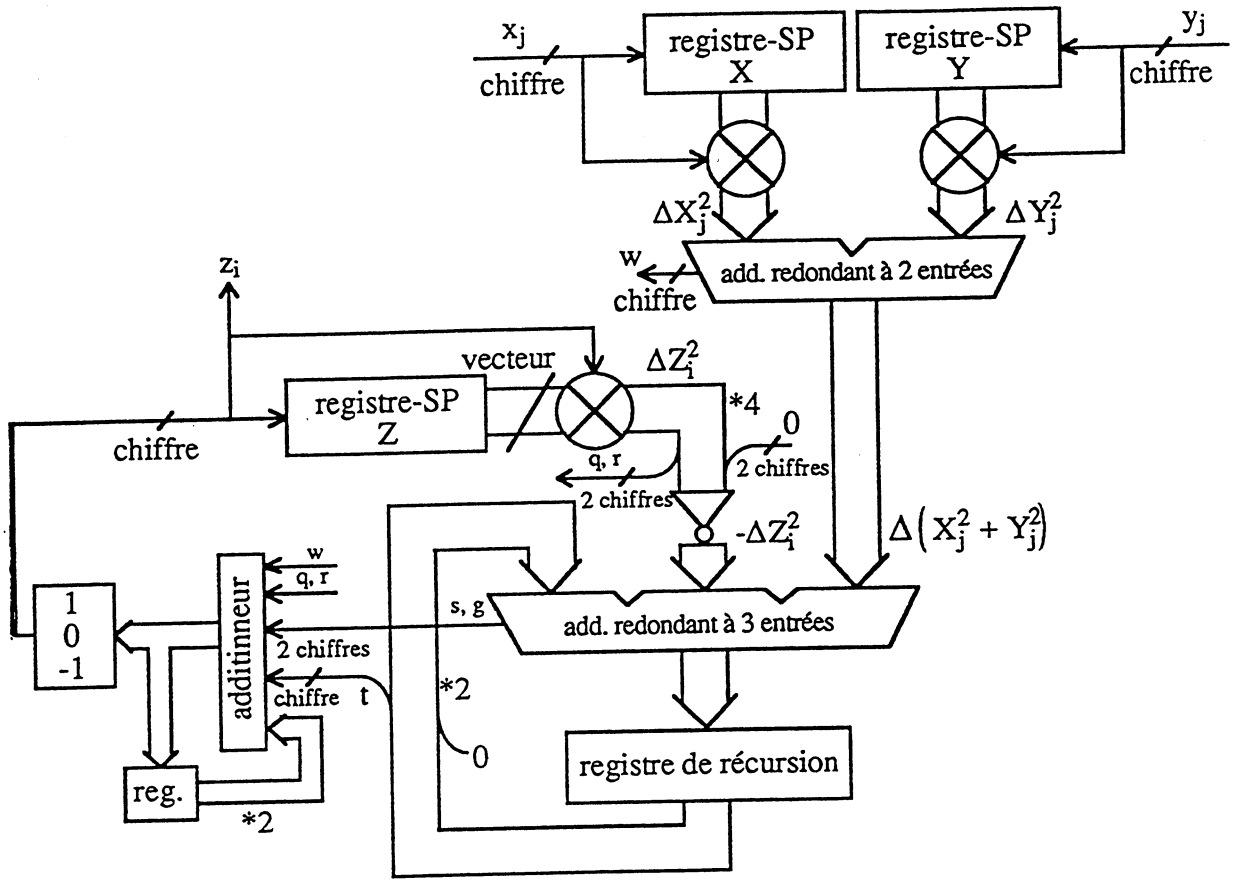


Figure VII.5 : opérateur euclidien

Le calcul commence avec la formation de ΔX^2_j et ΔY^2_j qui sont des produits partiels obtenus en multipliant les contenus des registres-SP de X et Y par les chiffres signés de x_j et y_j respectivement. Ces valeurs sont ajoutées pour donner $\Delta(X^2_j + Y^2_j)$ qui forme une évaluation partielle de $X^2 + Y^2$. L'étape suivante sera d'utiliser cette valeur partielle comme une entrée pour un processus itératif qui déterminera le résultat Z.

L'objet du processus itératif est de produire une valeur pour z_i, le prochain chiffre signé du résultat, à chaque cycle d'horloge, en respectant la relation suivante :

$$\left|\Delta Z^2 - \Delta(X^2 + Y^2)\right| < k*2^{-i}$$

où k est légèrement plus grand que 1. Cela signifie qu'avec la limite de précision : $Z = \sqrt{X^2 + Y^2}$.

Un indice différent est utilisé pour Z car le poids de ses chiffres signés ne sera pas le même que ceux de X et Y durant n'importe quel cycle d'horloge.

Le processus itératif commence en ajoutant $\Delta(X^2_j + Y^2_j)$, ainsi que le résultat précédent de l'addition - ΔZ^2_i . La dernière opérande est multipliée par 2 (un décalage à gauche d'une location) pour la maintenir au même poids que les autres opérandes. Cela fournit un chiffre signé appelé t.

L'opérande - ΔZ^2_i est formée en décalant deux locations ΔZ^2_i à gauche et le chiffre signé qui n'a pas été décalé en sortie sera négatif. ΔZ^2_i est obtenue de la même manière que ΔX^2_j et ΔY^2_j. Les deux chiffres signés qui ont été décalés en sortie deviennent q et r.

Le résultat de l'addition produira deux retenues et une nouvelle valeur est stockée dans le registre de récursion. Les deux retenues s et g, sont des chiffres signés. Leurs valeurs serviront à indiquer si $\Delta Z^2 - \Delta(X^2 + Y^2)$ est positif, négatif, ou nul.

Le registre de récursion contient deux registres à décalage maître-esclave qui décalent le résultat à gauche d'une location et le retardent au début du prochain cycle d'horloge.

L'étape suivante du processus itératif détermine la valeur de l'erreur dans $\Delta Z^2 - \Delta(X^2 + Y^2)$ en ajoutant w qui est la retenue à partir de l'étage le plus significatif du premier additionneur, s, g, t, -q, -r et la valeur qui précède cette addition. En faisant ce calcul, en tenant compte des poids et des chiffres signés, nous aboutissons à l'expression suivante :

valeur d'erreur = valeur précédente d'erreur*2 + t + (w+g+s)*2 - q*4 -r*2

Finalement, la valeur de l'erreur est utilisée pour déterminer z_i, où z_i est $\bar{1}$, 0, ou 1 selon que la valeur d'erreur est respectivement négative, nulle ou positive.

Cette manière d'obtenir le prochain chiffre signé du résultat fonctionne correctement tant qu'il y a une cohérence dans l'opération sur le chiffre signé de même poids, et que la valeur de l'erreur calculée par l'expression ci-dessus est bornée. Si la valeur de l'erreur devient non bornée, cela indique que l'exactitude de la condition $|\Delta Z^2 - \Delta(X^2 + Y^2)| < k*2^{-i}$ n'est pas maintenue par le processus itératif, et le résultat est donc invalide.

La cohérence de l'opérateur utilise une architecture en tranche de bit qui contient :

- une location en chiffre signé de chacun des registres-SP de X, Y et Z,
- un multiplieur-VD pour chaque registre-SP,
- une tranche de bit de l'additionneur redondant pour $\Delta X^2_j + \Delta Y^2_j$,
- deux tranches de bit de l'additionneur redondant pour l'additionneur à trois entrées,
- une location d'une bascule maître-esclave de type D pour le registre de récursion.

Le nombre de transistors par bit de l'opérateur euclidien est donné ci dessous.

Pièce	transistors (règles CMOS)	
registre-SP	14	
multiplieur-VD	16	
additionneur redondant à 2 entrées	40	
additionneur redondant à 3 entrées	80	
registre	24	maître-esclave statique

Nous avons gagné 48 transistors par bit sur un total de 282, ce qui représente un gain de 17% par rapport à ceux réalisés pour le même calcul avec les opérateurs en ligne de multiplication, carré, division et racine carrée. Le délai a été réduit à 1 au lieu de 7 au dépend de la péroide [GHP 89, GHM 89, GUK 91].

Le décalage dans ΔZ^2_i peut être fait en passant les paires de fils à travers la tranche de bit. Cette tranche peut être maintenant dupliquée autant de fois que nécessaire pour obtenir un registre-SP de Z assez large, un nombre de chiffres signés exigés pour obtenir un résultat exact.

Dans le paragraphe suivant, l'opérateur euclidien est décrit sous sa forme finale qui traduit le programme en langage C (Annexe 2).

VII.3. CIRCUITS DE L'OPERATEUR EUCLIDIEN

VII.3.1. Opérateurs

Dans ce paragraphe, nous présentons les différents opérateurs constituant le circuit du calcul euclidien. L'additionneur redondant et le multiplieur ont été décrits dans le chapitre VI §VI.2.1.2 et §VI.2.1.3.

VII.3.1.1. Registre-SP

La spécification de ce circuit tient dans le fait que chaque location est capable d'être remise à zéro, de stocker un chiffre signé et de stocker la valeur négative d'un chiffre signé d'entrée. Ce circuit est montré figure VII.4.

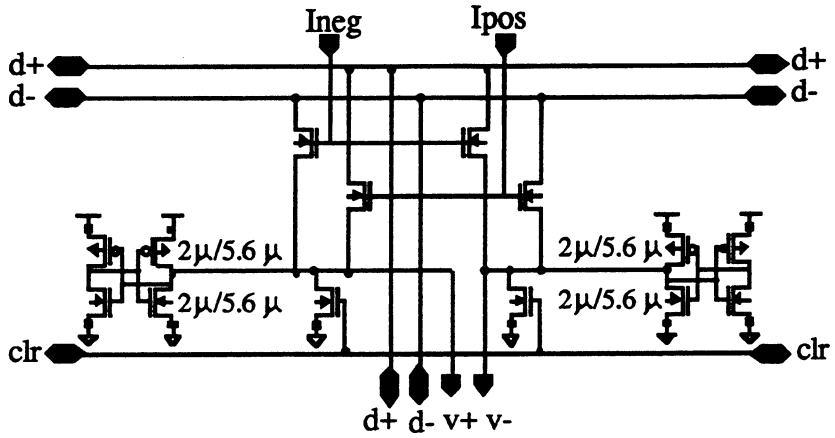


Figure VII.4 : circuit du registre-SP (w/l de tous les transistors non déclarés est 4µm/1.6µm)

L'entrée du chiffre signé a ses bits positif et négatif placés sur d+ et d- respectivement. Cette valeur est répartie à la location courante du registre-SP par les lignes appelées d+ et d- qui parcourent le circuit de gauche à droite le circuit.

Le chiffre signé est aussi fourni au multiplieur-VD avec les valeurs stockées dans le registre-SP, v+ et v- se trouvent en bas du circuit. Ils constituent parmi les chiffres signés ceux qui vont compléter un vecteur stocké dans le registre-SP.

Le registre-SP stocke les deux bits d'un chiffre signé dans les deux paires d'inverseurs tête bêche. Un des inverseurs dans chaque paire est plus faible que l'autre. Cette configuration facilite l'écriture dans les éléments mémoires.

Ces éléments mémoires sont statiques, cela signifie qu'une fois la valeur placée sur les inverseurs tête bêche, elle reste mémorisée jusqu'à ce que la prochaine valeur vienne la changer. En cas contraire un élément mémoire dynamique exige d'avoir son contenu rafraîchi à raison de quelques millisecondes. Les dimensions (w/l) des transistors dans les inverseurs faibles et gros sont respectivement 2µm/5.6µm et 4µm/1.6µm.

La sortie de l'élément mémoire est prise à partir du même point que l'entrée. Cela ne pose pas de problème car c'est la seule sortie qui aille à l'entrée de la logique combinatoire et la distance d'interconnexion est courte dans le dessin au micron final.

Il y a trois lignes de contrôle dans le registre-SP, chaque ligne est active quand nous lui appliquons un 1 logique. Les deux premières lignes de contrôle **Ipos** et **Ineg** contrôlent le transfert éventuel de d+ et d- vers v+ et v-. **Ipos** activé impose respectivement d+ sur v+ et d- sur sur v-, **Ineg** d+ sur v- et d- sur v+.

La troisième ligne de contrôle, **clr**, est utilisée pour mettre les contenus de deux éléments mémoires à zéro. Ceci est réalisable en effectuant, par l'intermédiaire d'un transistor NMOS, une connexion à partir de l'entrée d'un élément mémoire.

VII.3.1.2. Registre de récursion

L'objectif du registre de récursion est de décaler la valeur qui apparait sur les entrées d'une location de gauche et de retarder son apparition sur les sorties jusqu'au début du prochain cycle d'horloge. Ceci est réalisé (figure VII.5) en utilisant deux lignes pour décaler le chiffre signé d'entrée à la prochaine location du poids fort et les deux bascules maître-esclave de type D valident l'entrée sur la sortie au début du prochain cycle d'horloge.

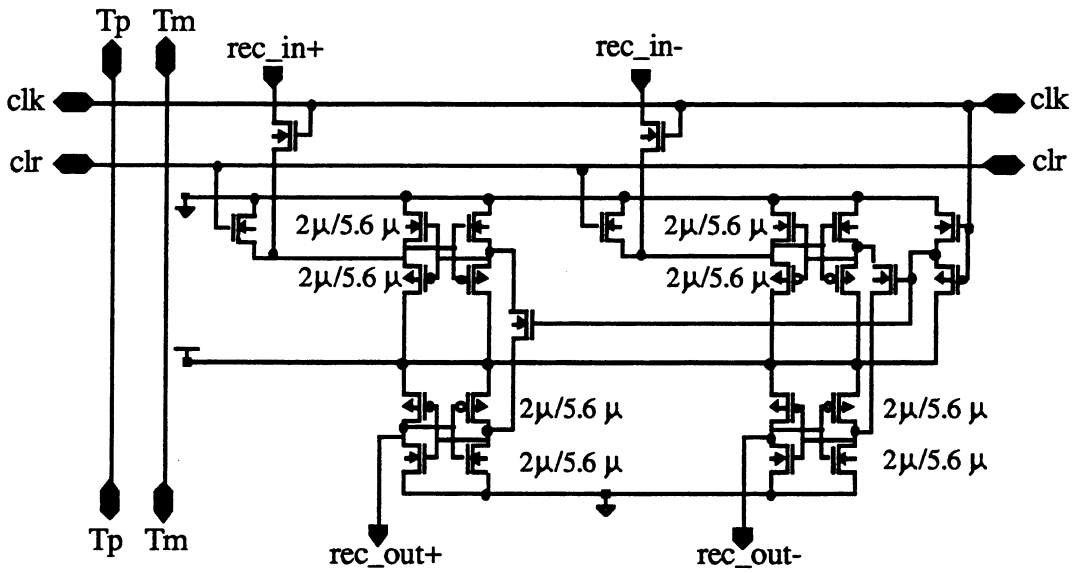


Figure VII.5 : circuit du registre de récursion
(w/l de tous les transistors non déclarés est 4µm/1.6µm)

VII.3.1.3. Tranche de bit

Les symboles des opérateurs additionneurs redondants, registre-SP, multiplieur-VD et registre de récursion, ont été assemblés et connectés (figure VII.6) afin d'implanter la description de l'opérateur euclidien donnée par la fonction **main** du programme 1 écrit en langage C (Annexe 2).

A l'intérieur de la tranche, toutes les données passent du haut vers le bas. Le chemin de données commence au registre-SP et se termine au registre de récursion où sont assemblées les données.

Toutes les données inter-tranches passent horizontalement. Les broches qui ont été utilisées dans la figure VII.6 sont appelées de façon à ce que tous les noms ayant le suffixe dp ou + indiquent le bit "plus" du chiffre signé et tous ceux ayant le suffixe dm ou - indiquent le bit "moins" du chiffre signé. Ils ont été positionnés de telle manière que si deux tranches sont aboutées, alors le passage de données entre elles est correct.

Le décalage à gauche de deux places est accompli en passant le chiffre signé - $\Delta Z2_i$ de la présente tranche de bit à la suivante, qui à son tour le passe à celle d'après par le chemin **Thru_1** et **Thru_2**.

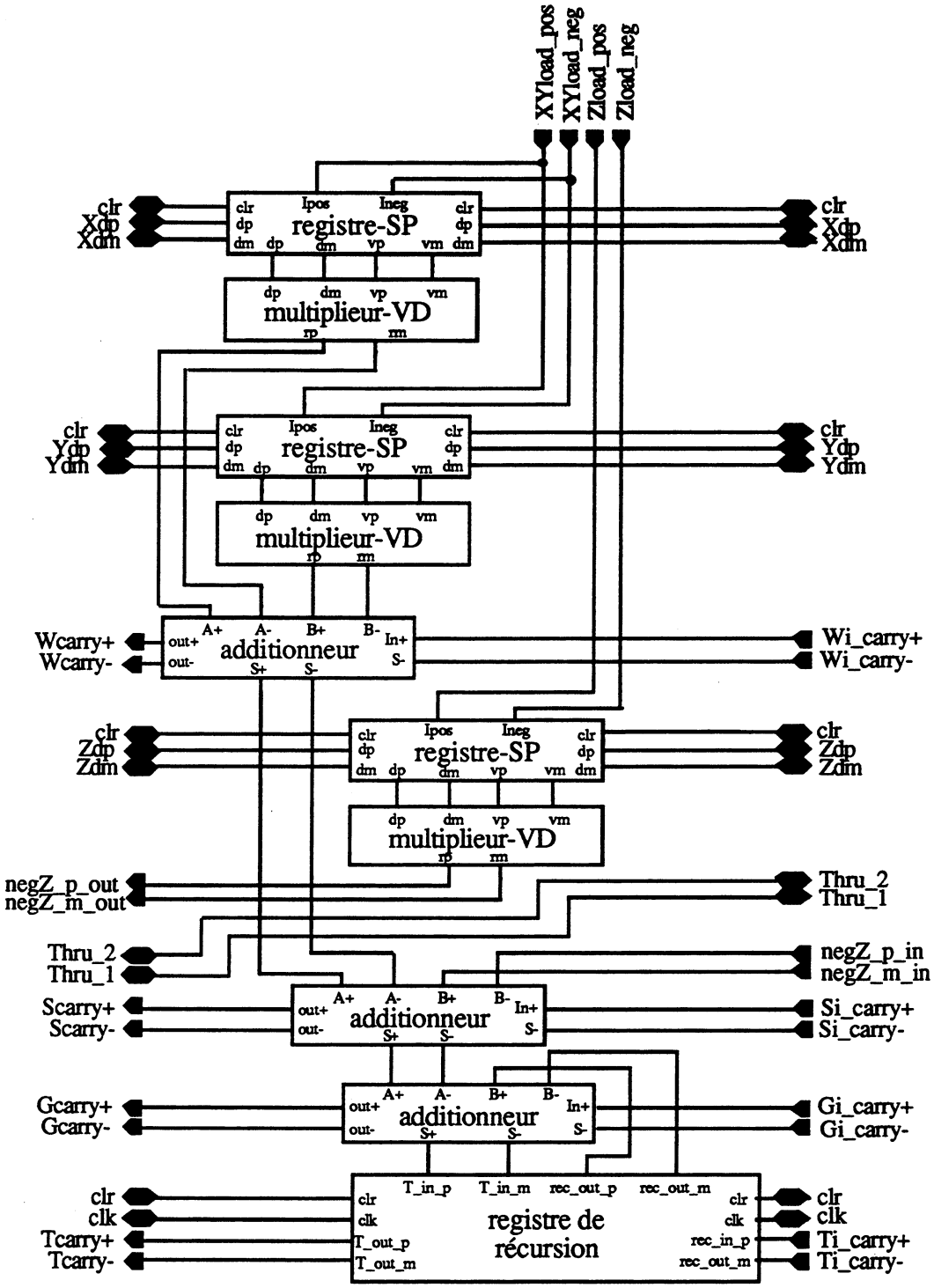


Figure VII.6 : schéma de la tranche de bit

Dans la tranche de bit du poids le plus fort, qui forme la tranche du poids fort d'un circuit, **[negZ_p_out, negZ_m_out]** est le chiffre signé q et **[Thru_2, Thru_1]** est le chiffre signé

r. La négation de - $\Delta Z2_i$ est achevée en changeant les lignes qui partent de la retenue positive et négative du chiffre signé dans l'additionneur du milieu de la tranche. Concernant le chargement des registres-SP, nous avons choisi, pour permettre un contrôle maximal de la tranche de bit à cette étage, que l'accès direct soit donné aux broches **Ipos** et **Ineg** des registres-SP via respectivement **XYload_pos** et **Zload_pos**, et **XYload_neg** et **Zload_neg**.

XYload_pos et **XYload_neg** contrôlent les registres-SP pour les données X et Y. Elles sont liées ensemble car ces registres doivent toujours charger les données en même temps. **Zload_pos et Zload_neg** contrôlent le registre-SP de Z.

Signalons que les chiffres signés ainsi produits (w, q, r, s, g et t) nous ont permis de vérifier le bon fonctionnement de la tranche de bits moyennant la comparaison avec ceux générés par le programme 2 (Annexe 2).

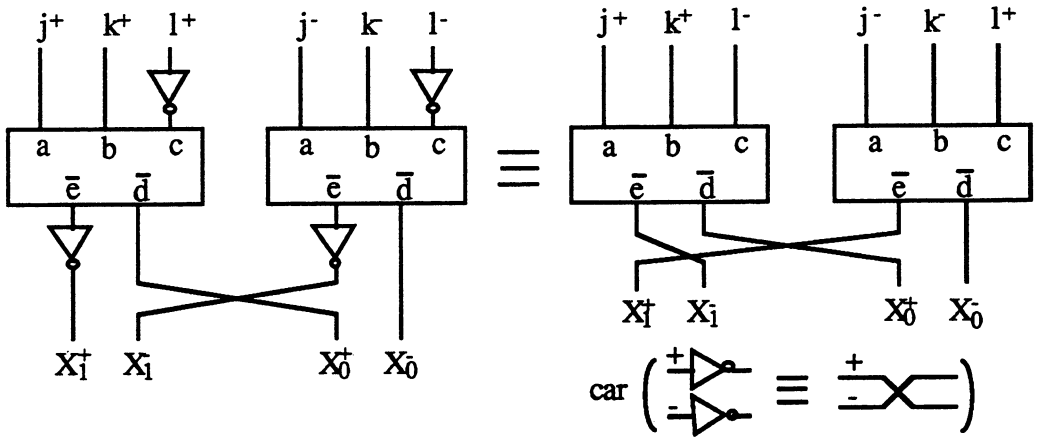


Figure VII.7 : deux variantes de l'opérateur j+k-1 utilisant deux PPM

VII.3.1.4. Circuits d'évaluation et de boucle de retour

Les circuits d'évaluation et de la boucle de retour sont utilisés pour déterminer la valeur de chaque chiffre signé, z_i, du résultat à partir des chiffres signés w, q, r, s, g et t. Le circuit d'évaluation implante l'expression de la **valeur d'erreur** décrite dans le paragraphe VII.2.3. Cette expression inclut l'addition et la soustraction de chiffres signés de différents poids. Remarquons que, de même que pour l'additionneur redondant, un circuit pour calculer j+ k-1 pourrait être réalisé en utilisant deux PPM, où j, k et 1 sont des chiffres signés. Le circuit pour l'opérateur j+k-1 est montré dans la figure VII.7.

Après avoir réarrangé l'expression, le circuit est présenté dans la figure VII.8. En plus des différents éléments d'addition du circuit, il y a quatre paires de bascules maître-esclave de type D qui sont utilisées pour stocker **la valeur d'erreur** jusqu'au début du cycle d'horloge suivant. Il arrive qu'un additionneur redondant fournisse une valeur dont le bit de poids le plus fort est à 1, par exemple 5 sous la forme 1 0 $\bar{1}$ $\bar{1}$. La broche de sortie, R0-, qui est toujours à 0V, a été omis de ce circuit pour pouvoir donner deux noms à un nœud en connectant l'entrée R0- à 0V du circuit de la boucle de retour.

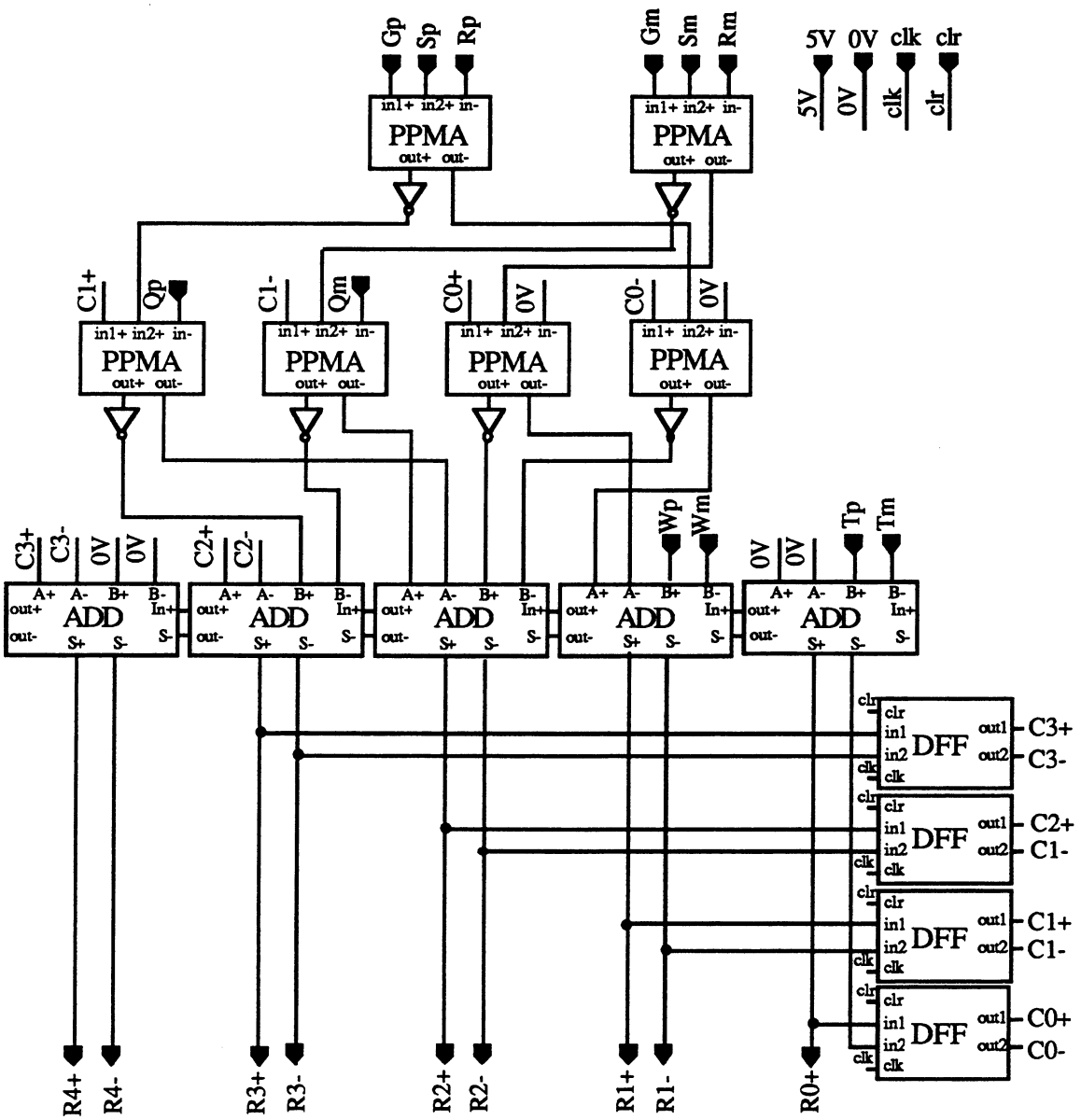


Figure VII.8 : circuit d'évaluation

Le circuit de la boucle de retour (figure VII.9) prend les cinq chiffres signés du résultat produit par le circuit d'évaluation et donne en sortie un chiffre signé, qui est la valeur du prochain chiffre signé du résultat. La sortie du chiffre signé est 1, 0 ou $\bar{1}$ dépendant de **la valeur d'erreur** qui est respectivement positive, nulle ou négative.

La technique utilisée pour déterminer la valeur est de trouver dans le nombre d'entrées le chiffre signé de poids le plus fort différent de zéro pour le faire traverser jusqu'à la sortie. S'il n'y a pas de chiffre signé différent de zéro dans le résultat, alors la sortie est zéro. Le circuit de la boucle de retour implante cette technique en testant en entrée chaque chiffre signé pour déterminer s'il est différent de zéro, et ce pour le transmettre en sortie, seulement lorsqu'il n'y a pas un chiffre signé de poids supérieur différent de zéro.

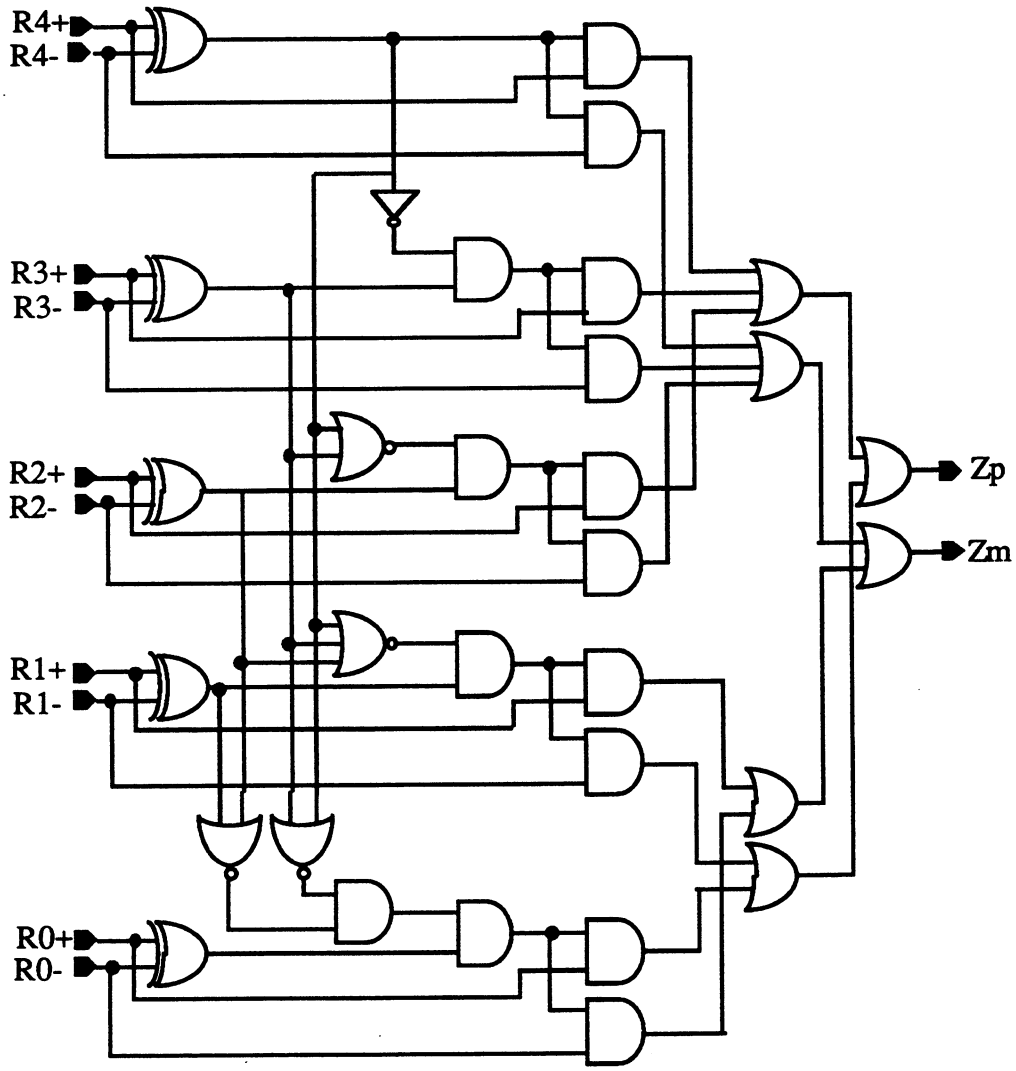


Figure VII.9 : circuit de boucle de retour

Ces circuits ont été initialement vérifiés en écrivant les descriptions au niveau portes de ceux-ci pour être utilisés comme des fonctions dans le programme 2 (Annexe 2).

Le programme a alors été exécuté sur plusieurs paires d'entrées. Les résultats générés par le programme étaient identiques à ceux générés par la simulation algorithmique.

VII.3.2. Circuit de contrôle

Maintenant que la tranche de bit, le circuit d'évaluation et le circuit de la boucle de retour ont été conçus et simulés, il est possible de concevoir un matériel qui sera chargé de contrôler le circuit de l'opérateur euclidien.

La méthode utilisée est la même que celle utilisée pour les circuits multiplieur et diviseur [KUS 89]. Cela implique de passer un bit de contrôle, ou un jeton, à partir d'une tranche de bit vers la suivante au début de chaque cycle d'horloge. Dans ce cas, l'opérateur euclidien a deux jetons pour indiquer quand nous pouvons charger les registres-SP de X et Y, et quand nous pouvons charger le registre-SP de Z. Le circuit permettant de faire passer le jeton entre les tranches sera décrit en premier lieu, puis nous nous penchons sur la logique utilisée pour générer le jeton.

VII.3.2.1. Circuit de contrôle de charge

Le circuit de contrôle de charge est le même que celui du registre de récursion (figure VII.5). Dans ce circuit, une des bascules est utilisée pour contrôler les registres-SP de X et Y, l'autre le registre-SP de Z. Dans les deux cas le fonctionnement est le même. La bascule reçoit un jeton à 1 logique et affirme en même temps le signal à trois endroits différents.

Le premier endroit est **Ipos** propre au registre-SP dans la tranche de bit qui fait partie du circuit. Le deuxième endroit est une entrée **Ineg** propre au registre-SP dans la tranche de bit suivante. Il n'est pas possible d'utiliser ce signal comme une entrée de la prochaine bascule du contrôle de charge pour le troisième endroit car nous aurons un problème pour monter la bascule du poids fort dans le circuit. En effet, dans la tranche de bit du poids fort, les entrées **Ineg** pour tous les registres-SP doivent être connectées à 0 logique car elles ne font pas partie du processus de calcul. Si le signal pour les entrées **Ineg** était pris à partir de l'entrée des bascules, nous aurons alors un conflit intéressant. Il n'est pas possible de connecter l'entrée à 0

et d'introduire un jeton. Les deux circuits de contrôle de charge du poids fort sont montrés dans la figure VII.10. Cela conduisant à une tranche de bit montrée dans la figure VII.11.

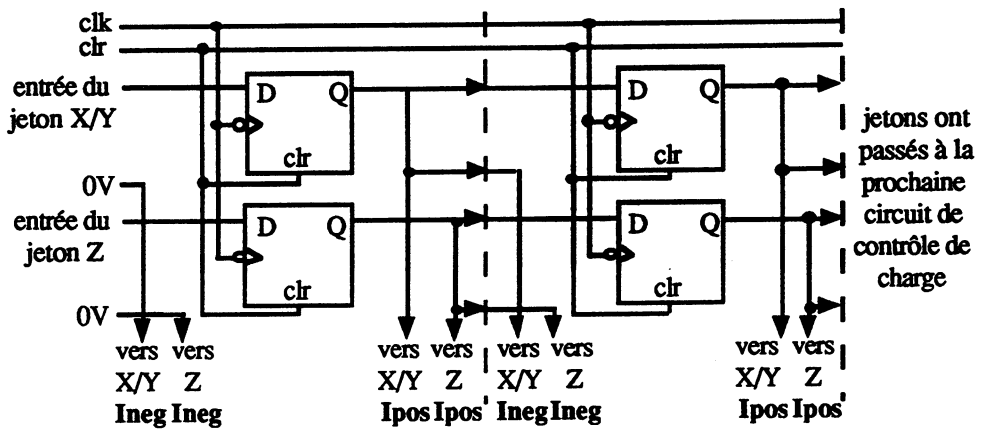


Figure VII.10 : schéma de deux circuits de contrôle de charge du poids fort

Lorsque le circuit de contrôle est conçu, il est clair que, si le jeton commence son parcours à une place correcte pour les registres-SP de X et Y, il est nécessaire de supprimer le circuit de contrôle de charge à partir des trois premières instances de poids fort de la tranche de bit. Ceci ressemble à une séquence, en raison du fait que le premier chiffre signé de X et/ou Y doit charger dans une position de trois locations du poids faible que du poids fort de Z (§2 dans l'Annexe 2). Au retour nous pouvons construire des locations de registres-SP de X et Y et leurs multiplieurs-VD dans ces tranches redondantes.

C'est pour cela qu'une seconde tranche de bit a été conçue (figure VII.12). Elle se compose d'une location du registre-SP, d'un multiplieur-VD, de trois additionneurs et d'une location du registre de récursion. En haut de l'additionneur, ces entrées sont reliées à 5V, pour deux raisons. La première est qu'il est nécessaire de tenir le chiffre signé w, utilisé pour déterminer le prochain résultat du chiffre, compatible avec les autres chiffres signés. Cela signifie que w sera toujours maintenu égal à 0 en chiffre signé. La seconde raison est de fournir un 0 en chiffre signé pour les entrées A de l'additionneur du milieu. Les entrées de l'additionneur du haut ont été connectées à 1 logique qui serait leur valeur si le registre-SP et le multiplieur-VD étaient connectés, et si le registre-SP contenait un 0 en chiffre signé présenté par [0, 0].

Trois instances de cette tranche sont maintenant utilisées pour activer le chemin de données de l'opérateur euclidien.

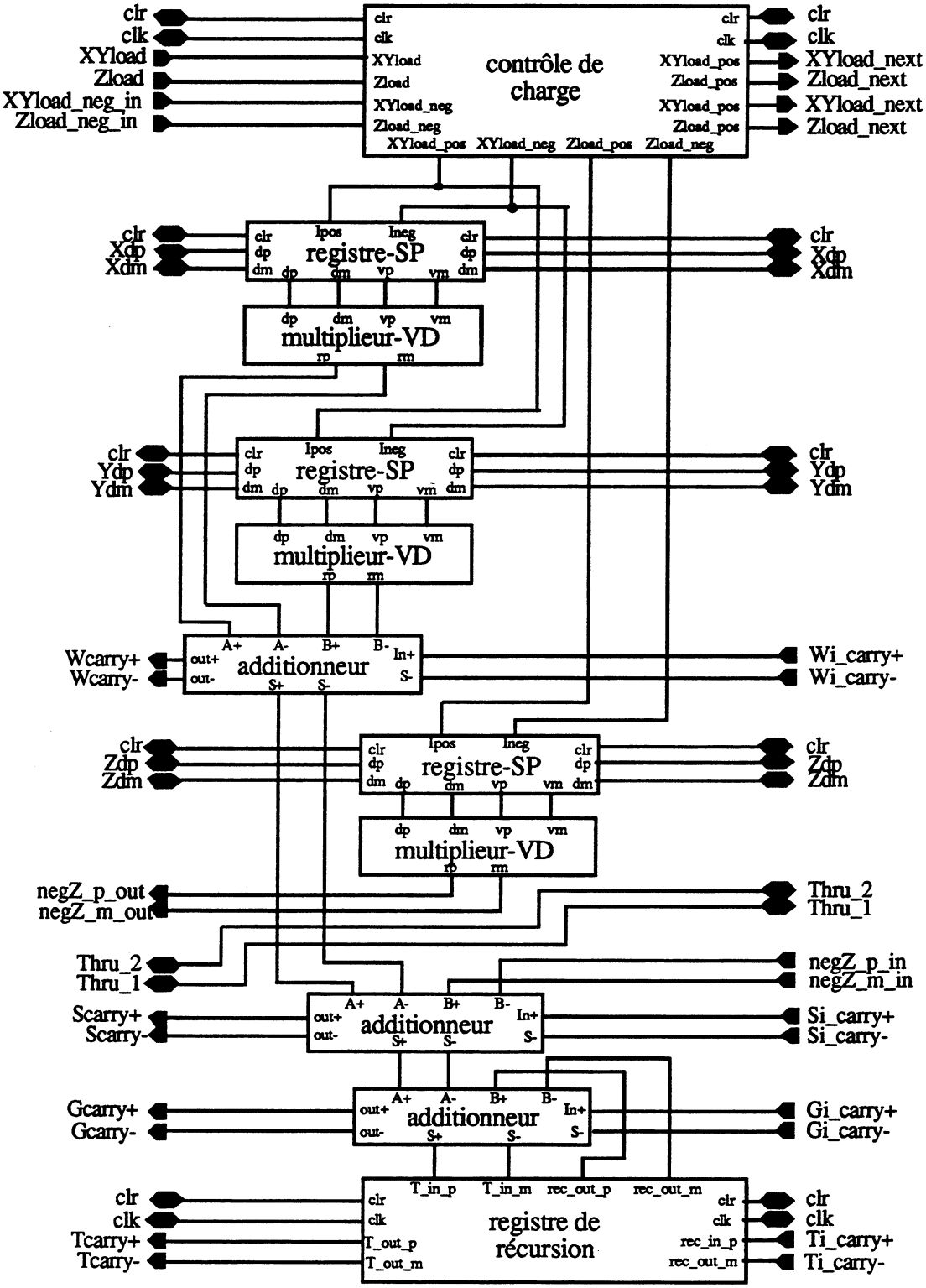


Figure VII.11 : tranche de bit avec contrôle de charge

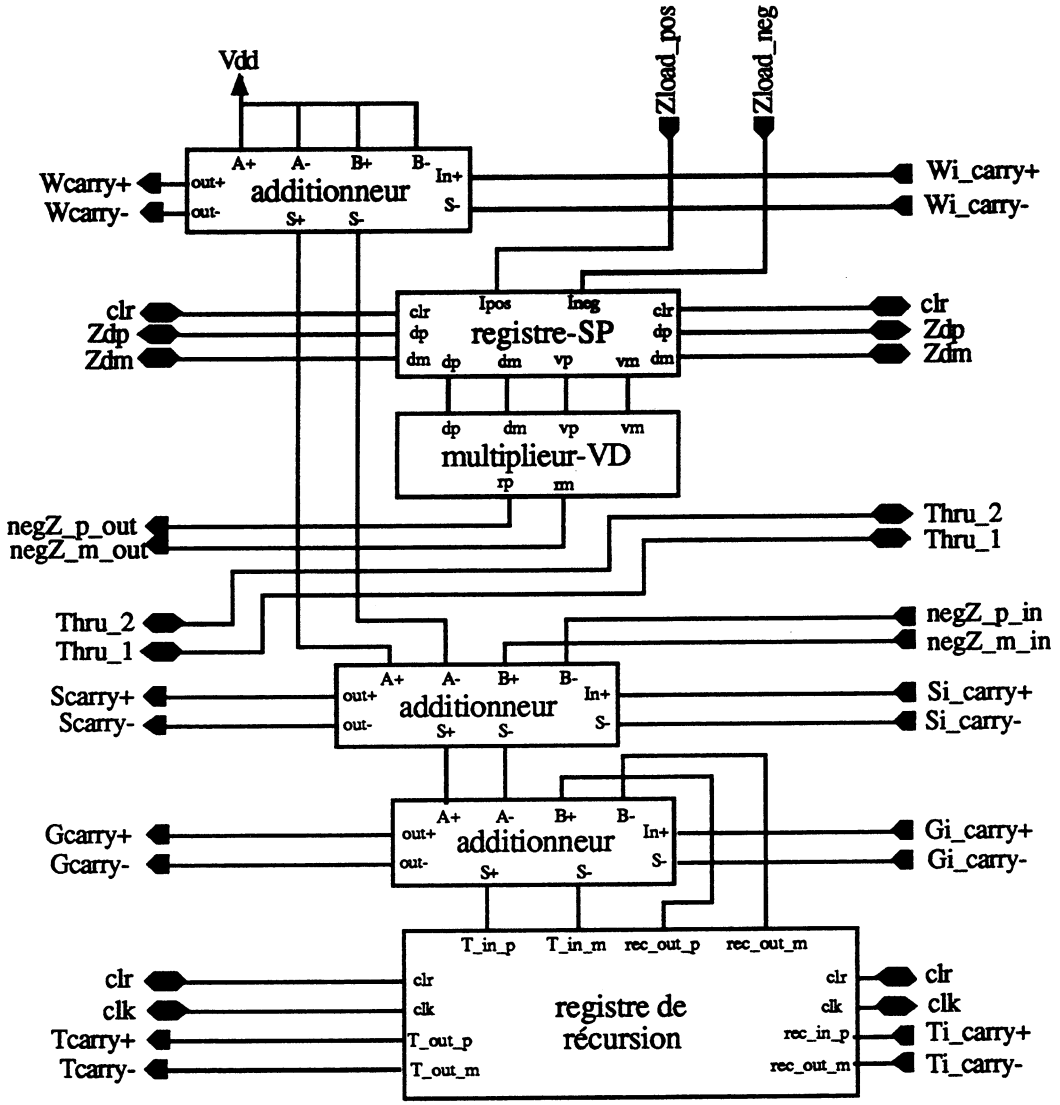


Figure VII.12 : petite tranche de bit

VII.3.2.2. Circuits de séquencement

Il y a deux circuits de séquencement, un pour les registres-SP de X et Y (figure VII.13) appelé XY_ipstage, et un pour le registre-SP de Z (figure VII.15) appelé Z_ipstage. Ces circuits synchronisent les données d'entrées et génèrent des jetons pour les circuits de contrôle de charge.

La synchronisation de données est accomplie en utilisant des bascules maître-esclave déclenchées sur le front descendant. Il y en a quatre dans le XY_ipstage pour les chiffres signés d'entrées de X et Y et deux dans le Z_ipstage pour le chiffre signé de Z.

Cette technique réalise que les entrées pour les registres-SP sont maintenues valides pour un cycle d'horloge complet. La génération de jeton de charge sera décrite maintenant.

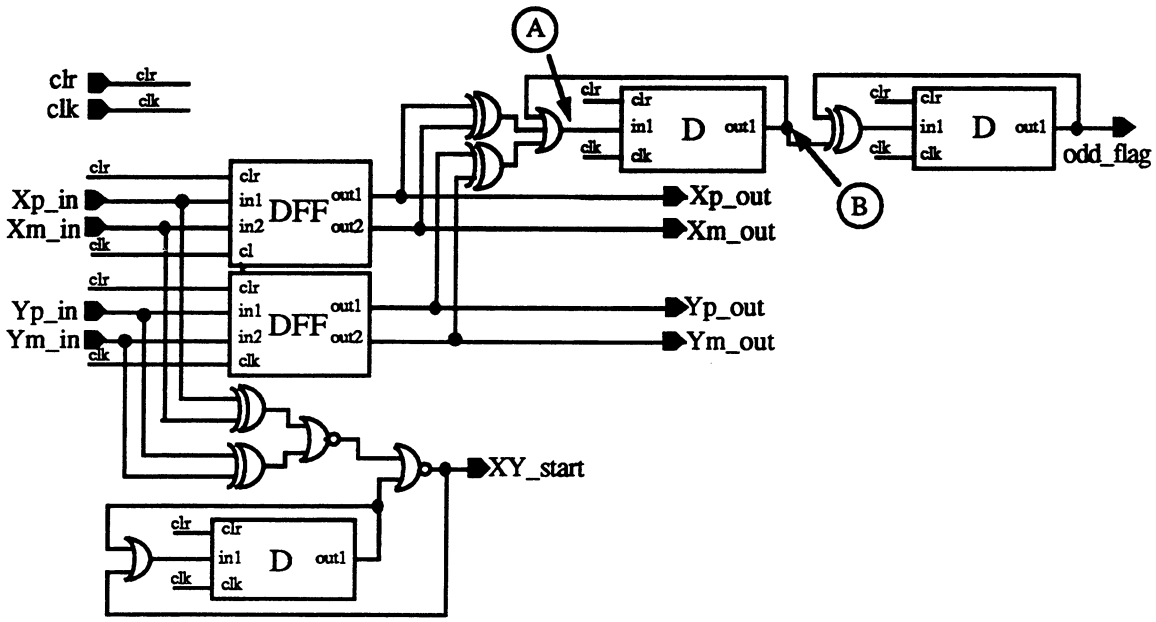


Figure VII.13 : circuit pour XY_ipstage

Le XY_ipstage contient deux circuits. Le premier est une logique qui génère le jeton pour les circuits de contrôles de charges X et Y. Ce circuit donne le signal **XY_start** quand le premier chiffre signé différent de zéro devient valide sur les entrées **Xp_in** et **Xm_in**, ou sur les entrées **Yp_in** et **Ym_in** après que l'opérateur euclidien ait été remis à zéro.

Le signal **XY_start** est alors injecté dans le premier circuit de contrôle de charge et la validation du chiffre signé est chargé. Un chiffre signé différent de zéro peut être détecté en utilisant une porte Ou-Exclusive, la sortie est 1 si un chiffre signé +1 ([1, 0]) ou $\bar{1}$ ([0, 1]) est placé sur les entrées.

Les paires d'entrées qui sont examinées sont celles qui forment les entrées pour les bascules de synchronisation. Cela signifie que le signal **XY_start** est seulement valide quand la donnée d'entrée est valide mais comme cette dernière est valide au début de chaque cycle d'horloge, cela ne pose pas de problème.

Le signal **XY_start** est aussi utilisé pour activer un drapeau en réalisant un second jeton qui ne peut pas être généré. Il a été fait en injectant le signal **XY_start** dans une bascule de type D

qui a une porte OU sur l'entrée.

Une des entrées de la porte OU est connectée à la sortie de la bascule. Donc, lorsqu'un 1 a été injecté dans le latch ; il retiendra cette valeur jusqu'à ce que **clr** soit activé. Le drapeau est alors envoyé dans une porte NOR avec les trois portes, qui détectent les chiffres signés différents de zéro, pour produire le signal **XY_start.**

Le second circuit dans XY_ipstage génère un signal appelé **odd_flag** en indiquant si le nombre de cycles d'horloges qui ont eu lieu depuis le chargement du premier chiffre signé est pair ou impair.

Ce circuit fonctionne en détectant lorsque le premier chiffre signé différent de zéro apparait sur les sorties des circuits de synchronisation et active alors un drapeau qui met une bascule maître-esclave de type T à basculer. Le drapeau est conçu de la même manière que le drapeau utilisé pour le signal **XY_start.**

La bascule de type T utilise une bascule maître-esclave de type D avec une porte Ou-Exclusive sur l'entrée. Une des entréee pour la porte Ou-Exclusive est connectée à la sortie de la bascule et l'autre forme d'entrée pour la bascule de type T. La sortie de cette bascule change d'état, ou bascule quand l'entrée est 1 et reste dans son état présent si l'entrée est 0. Tout changement d'état est produit sur le front descendant de l'horloge.

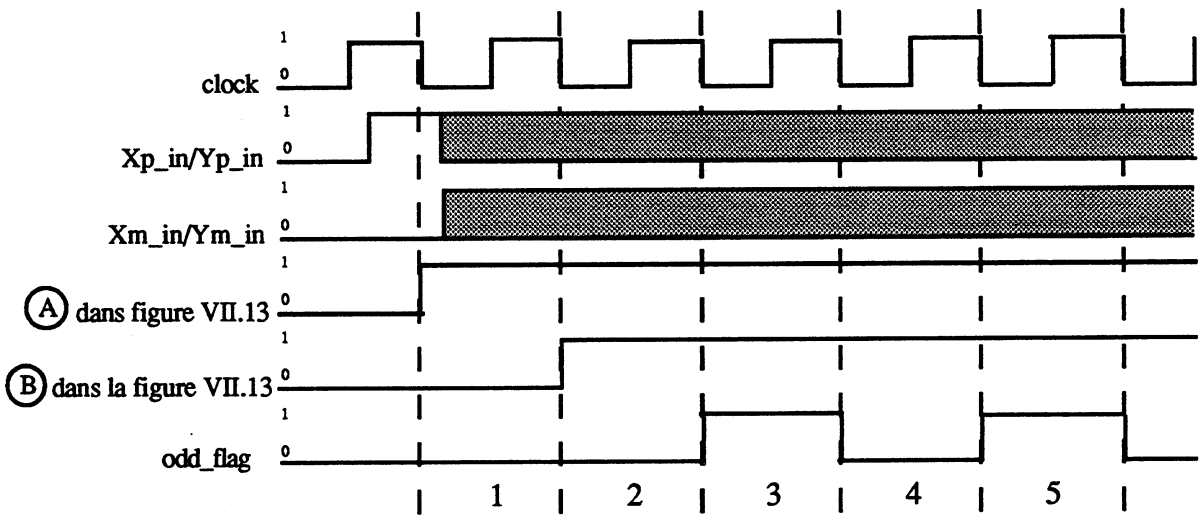


Figure VII.14 : diagramme de temps pour **odd_flag** (le temps de retard n'est pas inclu)

A partir du diagramme de temps (figure VII.14), nous pouvons voir que la valeur de **odd_flag** durant la période d'horloge appelée 1 n'est pas en logique 1 ce qui sera le cas si **odd_flag** a indiqué tous les impairs appelés périodes. Cependant, ce n'est pas un problème si la seule valeur de **odd_flag** à partir de la période 2 sera exigé d'être plus loin à cause du retard en ligne dans l'algorithme de l'opérateur euclidien.

Le signal **odd_flag** traverse sur le circuit Z_ipstage. Le signal est utilisé pour gouverner la location dans laquelle le premier chiffre signé de Z du résultat est chargé par le programme 2 (Annexe 2).

Le circuit fonctionne en déterminant quand le premier chiffre signé de Z différent de zéro apparait après que l'opérateur euclidien ait été remis à zéro, et en combinant cette connaissance avec la valeur de **odd_flag** de telle façon qu'un jeton soit placé à l'entrée pour corriger le circuit de contrôle du registre-SP de Z.

La valeur du chiffre signé est testée en entrée pour la synchronisation des bascules qui est nécessaire pour savoir si le chiffre signé est zéro ou différent de zéro avant le début de cycle d'horloge suivant.

La raison en est que, si le chiffre signé est différent de zéro, il sera chargé dans la location propre du registre-SP de Z sur le prochain front descendant de l'horloge. Le résultat de ce test est alors envoyé dans une porte AND avec un drapeau, qui a été conçu de la même manière que précédemment, pour arrêter la génération d'un second jeton de Z. Si l'entrée du chiffre signé n'est pas égale à zéro et le drapeau n'a pas été activé, alors une logique 1 est produite et envoyée dans une porte AND avec odd_flag et $\overline{\text{odd\_flag}}$ séparément. Cela cause un 1 qui apparait à l'entrée de la bascule de contrôle de correction.

Finalement, le circuit produit le signal **Zload**, qui est équivalent à **XY_start**. Ce signal est connecté aux entrées de Z du circuit de contrôle de charge de la grande tranche de bit du poids fort.

Ces blocs complets sont nécessaires pour assembler l'opérateur euclidien.

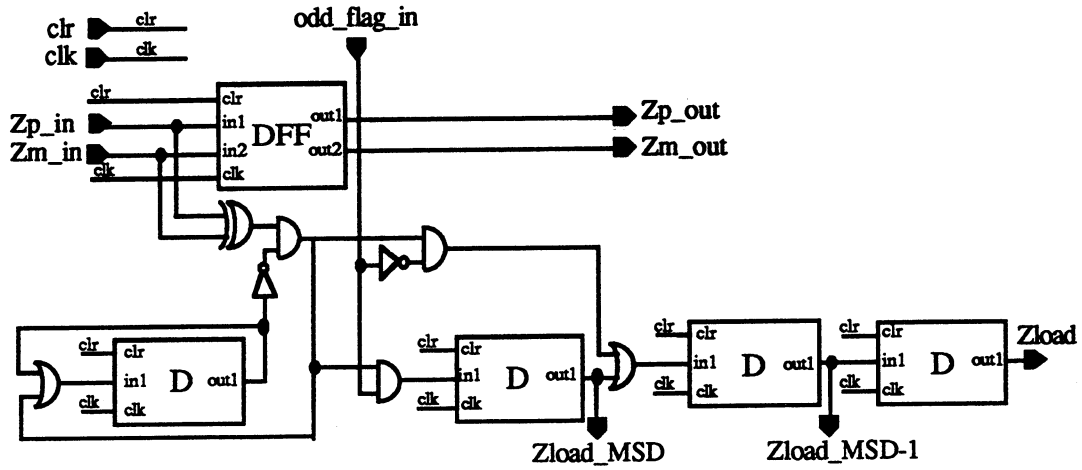


Figure VII.15 : circuit pour le Z_ipstage

VII.4. REALISATION DE L'OPERATEUR EUCLIDIEN

La description du circuit de l'opérateur euclidien complet [BGW 92, BGW 93] est montrée dans la figure VII.16. Il contient 3 instances de la petite tranche, 32 instances de la grande tranche, le circuit d'évaluation, le circuit de boucle de retour et tous les circuits de contrôle.

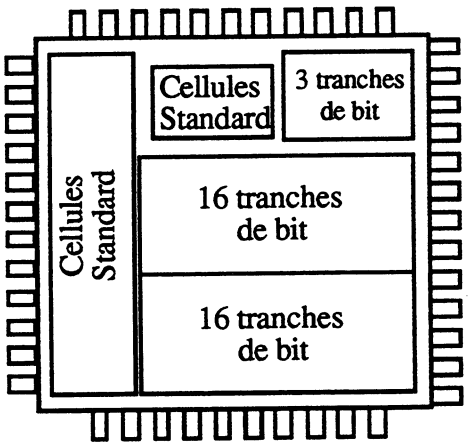


Figure VII.16 : description du circuit

Cette combinaison des composants est une implantation complète du programme 2 (Annexe 2). Le circuit a été simulé en utilisant SILOS avec les paires d'entrées des nombres 3 et 4, 5 et 12, 7 et 17, 23 et 23 et 25 et 25.

Les 32 instances permettentt au circuit de calculer les résultats à 9 chiffres décimaux significatifs de précision. Les 8 buffers sont inclus pour assurer un courant suffisant pour

conduire les lignes d'horloges, clear, Xp, Xm, Yp, Ym, Zp et Zm qui s'étendent tout le long du circuit

Dans chaque cas, la simulation procède de la même façon. D'abord, l'opérateur est remis à zéro en mettant un 1 sur l'entrée clear tandis que toutes les autres entrées sont à 0. Ensuite, l'horloge est activée et des valeurs sont placées sur les entrées. Les entrées des nombres de X et Y sont Xp et Xm, et Yp et Ym. Les valeurs sont mises sur ces entrées à partir du milieu d'un cycle d'horloge jusqu'au milieu du cycle suivant. Cela assure qu'un front descendant se produit, bien que les valeurs soient valides.

Les exemples de bit des résultats de simulation sont vérifiés en les comparant avec la sortie du programme machine pour les cinq paires de nombres d'entrées mentionnées ci-dessus. Dans l'Annexe B, les résultats de simulations pour les nombres 3 et 4 cités ci-dessus sont montrés et indiquent la manière selon laquelle la sortie du programme machine sera interprêtée.

Les résultats de simulation ont été analysés pour déterminer une estimation de la période minimale d'horloge qui peut être utilisée par le circuit. Le temps de propagation maximal observé pour l'ensemble des cinq résultats est 94 ns.

Le temps de propagation maximal de ce circuit est le temps que prend le chiffre signé du résultat pour devenir valide aux entrées du circuit de synchronisation du Z_ipstage après le début du présent cycle d'horloge.

Le test du circuit comporte essentiellement des plots de sortie qui activent tous les points externes de contrôle raisonnables aux tranches de bit. Ces plots ont été nommés avec le préfixe test.

De plus 10 plots d'entrées, nommés avec le préfixe force, et les portes Ou-Exclusive ont été ajoutées pour permettre le contrôle externe du circuit qui produira une panne de collage dans chaque partie du circuit de contrôle. Une panne de collage logique survient si l'entrée ou la sortie d'une porte ou d'une bascule reste collée de façon permanente à la valeur logique 0 ou 1. C'est généralement la manifestation électrique d'une panne physique qui surgit durant la fabrication du circuit intégré. Ces 10 plots d'entrées seront liés à 0V durant le fonctionnement normal du circuit.

Au retour de la fabrication du circuit intégré (figure VI.17), nous avons testé ce dernier, en utilisant une série des nombres d'entrées, par exemple, ceux utilisés dans la simulation du circuit. Les cinq paires de nombres ci-dessus indiquent que le circuit fonctionne en grande partie avec une fréquence entre 10 et 15 Mhz.

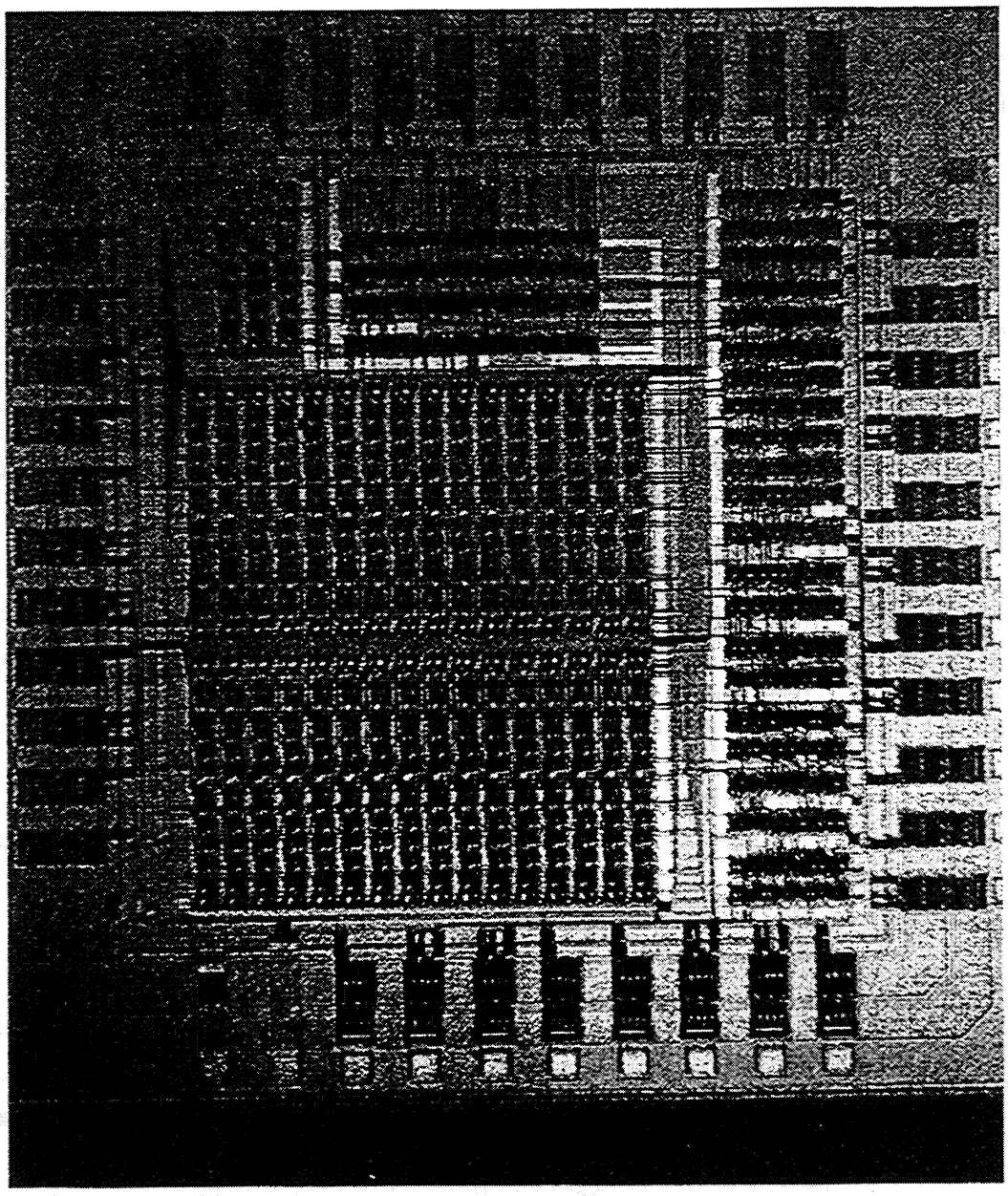


Figure VII.17 : photo du circuit

VII.5. CONCLUSION

Les algorithmes des opérateurs en ligne pour la multiplication, le carré, la division et la racine carrée décrits dans [GHP 89, GHM 89, GUK 91] peuvent être enchainer en un seul bloc pour calculer en ligne la distance euclidienne : $Z = \sqrt{X^2 + Y^2}$ utilisant la notation redondante. Cela entraine un faible temps de propagation et une réduction de 20% en nombre de transistors par rapport à ceux réalisés avec les opérateurs en ligne séparés.

La cohérence de l'opérateur utilise une architecture en tranche de bit. Cette dernière est dupliquée autant de fois que nécessaire pour obtenir un nombres de chiffres signés suffisant pour obtenir un résultat exact.

En utilisant une technologie 1,5 μm, 2 couches de métal et une très grande régularité, nous obtenons pour le circuit 32 bits une densité de 4500 tr/mm^2, occupant une surface de 18 mm^2.

Le test utilisant les paires d'entrées des nombres cités ci-dessus, a montré que quatre sur cinq circuits fabriqués par Service Eurochip fonctionnent correctement avec une fréquence entre 10 et 15 Mhz.

L'opérateur de la distance euclidienne est appelé à la résolution du moindre carrée des systèmes linéaires tel que le filtre Kalman [STC 90].

CONCLUSION

Conclusion

Les deux prototypes présentés dans ce document ont été conçus pour répondre au besoin de calcul exact avec une très grande précision et à grande vitesse. Ils peuvent faire chacun des opérations de décalage, de soustraction et d'addition qui sont les opérations de base pour le calcul du PGCD et la distance euclidienne de deux grands nombres. L'opération de décalage ne posait pas de problème lorsque nous avons utilisé les grands nombres. Par contre, le problème qui aurait pu apparaître dans l'opération d'addition ou de soustraction était la propagation de la retenue dans toute la chaîne de bits. Ceci peut provoquer le retard du résultat sortant. Les derniers bits des opérandes voient se propager leurs retenues jusqu'aux premiers bits du résultat, ce qui est plus grave lorsque nous travaillons avec beaucoup de chiffres (vers les 2048 bits et plus). Nous avons prévu deux solutions pour surmonter ce problème de retenue : additionner soit en ligne (approche série), soit en redondant (approche parallèle) sans propagation de retenue. Après l'étude des différentes approches, il est évident que la dernière solution est la meilleure, du point de vue performance et coût en matériel. Les deux processeurs travaillent donc en nombre redondant et plus précisément avec les poids fort en tête (arithmétique en ligne).

Les résultats de simulations de ces deux prototypes ont montré que le circuit PGCD de taille 128 bits devrait fonctionner correctement à une fréquence maximale comprise entre 20 et 25 Mhz, et le circuit distance euclidienne de 32 bits aux alentours de 20 Mhz. Après la fabrication de cinq circuits pour chacun des opérateurs et leur test, l'opérateur norme euclidienne donne des résultats exacts avec une fréquence entre 10 et 15 Mhz. Cela est différent pour le circuit EUCLIDE. Ces deux opérateurs peuvent être conçus pour obtenir une taille de 2048 bits pour le besoin de calcul en très grande précision. Le champ d'application du circuit EUCLIDE intervient directement dans la réduction des fractions en précision infinie, les calculs modulaires et la cryptographie [DAR 85, KOM 83, LEH 38]. Celui de la distance euclidienne peut être appliqué à la résolution du moindre carré des systèmes linéaires tel que les filtres de Kalman [STC 90] utilisés dans le traitement du signal [GHP 89, GUM 91, GUY 92].

Ces deux circuits sont conçus pour être placé à côté d'un microprocesseur et connecté sur le bus de la mémoire afin de réduire notablement le temps nécessaire pour les calculs de très grande précision. La taille des mots mémoires et le nombre des plots d'un circuit intégré étant limités, les grands nombres de n chiffres doivent être transférés en série, avec un temps O(n).

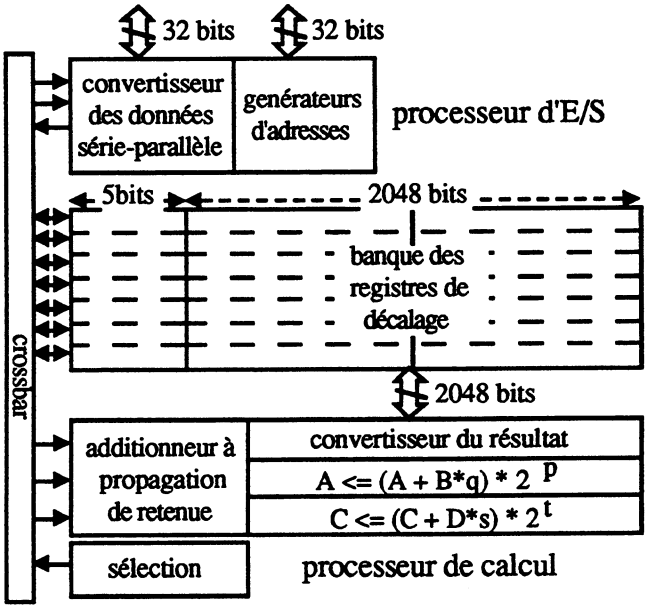
Le processeur d'entrées/sorties agit comme un accès mémoire directe (DMA), il scrute les bus pour l'adresse et la valeur du premier mot de l'opérande. Il demande ensuite le bus et engendre les séquences d'adresses du mot suivant [COG 87].

Chacun des registres de la banque est accessible aussi bien en série qu'en parallèle. Le bit du poids le plus fort est le plus à droite. Un accès en parallèle et six en série peuvent avoir lieu simultanément.

Un réseau complet (crossbar) permet de charger et stocker en série les registres pendant l'exécution des calculs sur les autres en mode pipeline, ou de contourner la banque en mode en ligne. Pour le PGCD étendu, trois des registres sont utilisés comme une pile pour enregistrer la séquence d'opérations.

Deux décaleurs additionneurs/soustracteurs, chacun avec deux registres (nombres redondants), composent le processeur de calcul. Pour la multiplication, la division [GHM 89], la racine carrée [OKE 82], ou la distance euclidienne [BGW 92, BGW 93], ils sont utilisés ensemble en mode en ligne ou séparés en mode série/parallèle pour effectuer deux opérations différentes.

Dans la figure ci dessous, A, B, C, D représentent des registres pour les grands nombres, q, $s \in \{-1, 1\}$ et r, $t \in \{0, 1, 2\}$. Les registres B et D peuvent être chargés en série de la gauche vers la droite [GHM 89]. Une paire de registres convertit "à la volée" les résultats série de la notation redondante à la notation usuelle (-1 ou 0 pour le bit de signe, 0 ou 1 ailleurs) [ELA 87] et à sa représentation sans 0 (seulement 1 et -1). Cette dernière représentation est utile pour la triangularisation des matrices avec la méthode de rotation de Givens parce qu'elle ne change pas le facteur de normalisation [ELA 90].



coprocesseur pour les très grands nombres

Les avantages de cette architecture sont :

1- d'exécuter les calculs en parallèle avec l'introduction des nombres (calcul en ligne) ; mais aussi de pipeliner le chargement, l'exécution et le stockage des résultats [COG 87],

2- de minimiser l'occupation du bus externe en stockant les résultats intermédiaires et les variables locales dans la banque de registres intégrée au circuit,

3- d'éviter les opérations à précision multiple en tirant avantage des très grands opérateurs. La notation redondante interne raccourcit le chemin de propagation de la retenue [GHM 89, MER 59]. Le coprocesseur doit exploiter la transmission parallèle des variables internes avec la banque de registres et la transmission série des variables externes,

4- d'exécuter l'addition, soustraction, multiplication, division, racine carrée [GHM 89, GUK 91, OKE 82], PGCD, PGCD étendu [BGK 91, BGU 92], la distance euclidienne [BGW 92, BGW 93] et fournir aussi un support pour les nombres rationnels [KOM 83] et les matrices,

5- l'utilisation optimale du silicium puisque nous pouvons utiliser le même chemin des données pour effectuer en parallèle de nombreuses opérations sur des opérandes courtes et/ou des opérandes transmises en parallèle.

REFERENCES ET BIBLIOGRAPHIE

[AHO 90] A. V. Aho, "Algorithms for Finding Patterns in Strings", H.T.C.S. Vol. A "Algorithms and Complexity", J. Van Leeuwen ed., p. 225-300, Elsevier 1990

[AVI 61] A. Avizenis, "Signed-digit number representation for fast parallel arithmetic", IRE Trans. on Electronic Computers, 10, pp. 389-400, 1961

[BGK 91] R. Bouraoui, A. Guyot and K. Khoumsi, "Prototype of a Circuit for the GCD and Extended GCD of Very Large Numbers", proc. International Conference on Microelectronics, Cairo Egypt, December 1991

[BGR 90] R. Bouraoui, A. Guyot and J.L. Roch, "The best way to build a circuit for the extended GCD of large integers", IMAG research report 1990

[BGU 92] R. Bouraoui and A. Guyot, "A Circuit for GCD and Extended GCD Calculation With Unlimited Precision", ESSCIRC'92, Copenhagen Denmark, September 1992

[BGW 92] R. Bouraoui, A. Guyot and G. Walker, "Design of an on-line Euclidean Processor", Revue Internationale Algérienne des Technologies Avancées, Juin 1992

[BGW 92] R. Bouraoui, A. Guyot and G. Walker, "Circuit Design for On-line Euclidean Norm", The International Conference on Microelectronics, Monastir Tunisia, December 1992

[BGW 93] R. Bouraoui, A. Guyot and G. Walker, "Design of an On-line Euclidean Processor", The Sixth International Conference on VLSI Design, Bombay India, January 1993

[BGW 93] R. Bouraoui, A. Guyot and G. Walker, "On-line Operator for Euclidean Distance Computing", EDAC-EUROASIC'93, Paris France, february 1993

[BKL 83] R.P. Brent, H.T. Kung and F.T. Luk, "Some linear time algorithms for systolic arrays", 9th IFIP World Computer Congress, Masson eds, 1983

[BKU 82] R.P. Brent and H.T. Kung, "A Systolic Algorithm for Integer GCD Computation", Report TR-CS-82-11, Dept. of Comp. Sc. Australian National Univ. December 1982

[BKU 83] R.P. Brent and H.T. Kung, "Systolic VLSI Array for linear-time GCD computation", proc. VLSI'83, Elsevier Science Publishing, North Holland-IFIP, 1983

[BKU 85] R.P. Brent and H.T. Kung, "A systolic algorithm for integer GCD computation", proc. 7th Symposium on Computer Arithmetic, 1985

[BMU 92] J.C. Bajard and J.M. Muller, "A New VLSI Architecture for fast on-line evaluation of power series", International Conference on Signal Processing Applications and Technology, Boston, November 1992

[CHE 90] S. CHEBBI, "Contribution à la conception d'un circuit intégré de calcul du PGCD en précision infinie", rapport de mémoire d'ingénieur, TIMA-INPG, Juillet 1990

[COG 87] M. Cosnard, A. Guyot, B. Hochet, J.M. Muller, H. Ouaouicha, Ph. Paul and E. Zysman, "The FELIN arithmetic coprocessor", proc. 8th Symposium on Computer Arithmetic, 1987

[CYR 78] C. chou and J.E. Robertson, "Logical design of a redundant binary adder", proc. 4th Symposium on Computer Arithmetic, pp. 109-115, October 1978

[DAR 85] J. Davenport and Y. Robert, "VLSI and Computer Algebra: The GCD Example", Dynamical Systems and Cellular Automata Academic Press, London 1985

[DEL 88] J. Della Dora, "Machines Auto-reproductibles: de l'intérêt du parallélisme dans la fabrication du café", Journal Libération, 16 Avril 1988

[DUP 46] A. Dupré, "Sur le nombre de divisions à effectuer pour obtenir le plus grand commun diviseur de deux nombres entiers", Journal de Mathématiques, Tome XI, 1846

[ELA 87] M. Ercegovac and T. Lang, "On-the-fly conversion of Redundant into Conventional Representation", IEEE Transactions on Computers, Vol. C36-7, 1987

[ELA 88] M.D. Ercegovac and T. Lang, "On-line scheme for computing rotation factors", Journal of parallel and distributed computing, Vol. 5 pp. 209-227, 1988, reprinted in IEEE Computer Arithmetic, Vol. II E. Swartzlander ed.

[ELA 90] M. Ercegovac and T. Lang, "Redundant and On-Line CORDIC: Matrix Triangularisation and SVD", IEEE Transactions on Computers, Vol. C39-6, 1990

[ERC 77] M.D. Ercegovac, "An on-line square root algorithm", proc. 4th IEEE Symposium on Computer Arithmetic, 1977

[ERT 77] M. Ercegovac and K.S. Trivedi, "On line algorithms for division and multiplication" IEEE Trans. on Computers, Vol. C26-7 pp. 681-687, July 1977

[GHM 89] A. Guyot, Y. Herreros and J.M. Muller, "JANUS, an on-line multiplier/divider for manipulating large numbers", proc. 9th Symposium on Computer Arithmetic, 1989

[GHP 89] A. Guyot, Y. Herreros, J.M. Muller and G. Privat, "Redundant arithmetic operators in digital signal processing applications", proc. IFIP workshop on parallel architecture on silicon, Grenoble, December 1989

[GUK 91] A. Guyot and Y. Kusumaputri, "OCAPI: A circuit for on line radix 2 high precision arithmetic", proc. VLSI 91, Edinburgh Scotland, August 1991

[GUM 91] A. Guyot and Z.J. Mou, "Optimal organisation of on-line arithmetic operators in signal processing applications", Internal Report, Electronic Department, Telecom Paris University, 1991

[GUY 91] A. Guyot, "OCAPI: Architecture of a VLSI coprocessor for the GCD and the extended GCD of large numbers", proc. 10th Symposium on Computer Arithmetic, 1991

[GUY 92] A. Guyot, "Optimal organization of on-line arithmetic operators in signal processing applications", International Symposium on Signal, Systems and Electronics, Paris, September 1992

[HAW 81] V.C Hamacher and J. William, "A linear time divider array", Canadian Electr. Engineering Journal, Vol. 6, N° 4, 1981

[HIS 74] M.W. Hirsch and S. Smale, "Differential Equations, Dynamical Systems and Linear Algebra", Academic Press, 1974

[HOP 85] M. D'Hoe, M. Pierre, Ph. Deleuze, A. Vandemeulebroeke, P. Jespers, M. Davio and C. Asba, "Cryptography Applications using Signed Binary Arithmetic", ESSCIRC'85, Toulouse 1985

[IRO 79] M.J. Irwin and R.M. Owen, "On-line algorithms for the design of pipelined architecture, proc. 6th IEEE Symposium on Computer Arithmetic, Philadelphia, 1979

[IRO 90] M.J. Irwin and R.M. Owen, "A case for digit serial VLSI signal processors", Journal of VLSI signal processing, n° 1 pp. 321-334, 1990

[KAR 87] R.M. Karp and M.O. Rabbin, "Efficient randomized pattern-matching algorithms", IBM. J. Re. Develop. 31 (2), 249-260, 1987

[KNU 81] D.W. Knuth, "Seminumerical Algorithms" (The Art of Computer Programming) pp. 317-336, Addison-Wesley, Reading, Massachusetts 1981

[KOM 83] P. Kornerup and D.W. Matula, "Finite Precision Rational Arithmetic: An Arithmetic Unit", IEEE Transactions on Computers, Vol. C34-4, 1983

[KUS 89] Y.O. Kusumaputri, "OCAPI, Opérateur de Calcul A Précision Illiimitée", Rapport de D.E.A, Université Joseph Fourier, Grenoble, Juin 1989

[LEH 38] D.H. Lehmer, "Euclid's Algorithm for Large Numbers", AMMMM 45, pp. 227-233, April 1938

[MER 59] G. Metze and J.E. Robertson, "Elimination of carry propagation in digital computer", proc. IFIP conference, 1959, reprinted in IEEE Computer Arithmetic, Vol. 1, E.E. Swartzanlander ed.

[MUL 89] J.M. Muller, "Arithmétique des ordinateurs", Masson-collection ed., 1989

[NIC 85] M. Nicolaidis, "An efficient Built-In Self-Test for functional test of embedded RAMs", proc. 15th International Symposium on Fault tolerant Computing, Ann Arbor, U.S, June 1985

[OKE 82] V. Oklobdzija and M. Ercegovac, "An on-line square root algorithm", IEEE Transactions on Computers, Vol. C31-1, 1982

[PAM 91] S.N. Parikh and D.W. Matula, "A redundant Binary Euclidean GCD Algorithm", proc. 10th Symposium on Computer Arithmetic, 1991

[PUR 83] G. B. Purdy, "A carry-free algorithm for finding the greatest common divisor of two integers", Comp. & Maths. with Appls, Vol. 9 pp. 311-316, 1983

[ROC 89] J.L. Roch, "L'architecture du Système PAC et son Arithmétique Rationnelle", thèse INP Grenoble, Decembre 1989

[ROC 90] J.L. Roch, "The PAC System, general presentation", proc. CAP 90. E. Kaltofeln ed. North Holland. 1990

[SCH 71] A. Schönhage, "Schnelle Berechnung von Kettenbruchentwicklungen", Acta Informatica, Vol. 1, p. 139-144, 1971

[SEN 90] P. Sénéchaud, "Calcul formel et parallélisme : bases de Grobner booléennes, preuve de circuit et parallélisme", thèse INP Grenoble, 1990

[SIE 89] F. Siebert-Roch, "Parallel Algorithm for Hermit, Normal Form of Matrices", thèse INP Grenoble, 1990

[STC 90] R.W. Stewart and R. Chapman, "Fast stable Kalman filter algorithm utilizing the square root", proc. IEEE International conference on Acoustic Speech and Signal Processing, 1990

[STE 67] J. Stein, "Computing GCD without division", J. Comp. Phys. Vol. 1, pp. 397-405, 1967

[TUN 68] C. Tung, "A division algorithm for signed digit arithmetic", IEEE Trans. on Computers. Vol. C-17, 1968

[VIL 88] G. Villard, "Calcul formel et parallélisme : résolution de systèmes linéaires", thèse INP Grenoble, 1988

[YUZ 86] D.Y. Yun and C.H. Zhang, "A fast carry-free algorithm and hardware design for extended integer GCD computation", proc. ACM Symposium on Symbolic Algebraic Computing pp. 82-84, 1986

ANNEXES

Annexe 1 : Résultats de simulations du circuit PGCD

Programme

```
program PGCD;
{approche parallèle avec les poids forts en tête et utilisant bezout avec les}
{ poids forts en tête                                                        }

type tableau = packed array [0..1999] of char;
var val: longint;
var ap, am, bp, bm, cp, cm, xm  : integer;
var gcd,signe, cout, seq, newseq, rapport  :integer;
var i,j,k, ctyp, nbcycle, cyclebezout , x,savetaila,echantillon,save: integer;
var taila, tailb, initaila, initailb, tailr,tails,teta,tetb, oldtail: integer;
var valamod2_16, valbmod2_16, valrmod2_16, valsmod2_16, valgcdmod2_16:integer;
var trp,trm,tsp,tsm:tableau;
var tap,tam,tbp,tbm:tableau;
var typ:packed array[0..3999] of char;{historique des opérations}
var car:char;

printer {trace des resultats établies}, entree {les couples de nombres },
bilan {le fichier resultat tabulé qui nous permettra d'établir nos tracés}:text;


function modulo2_16(t1p,t1m:tableau; taille:integer):integer;

{recupère les 16 derniers bits contenus dans les deux tableaux t1p et t1m}
{et calcule la valeur du nombre redondant modulo 2 à la puissance 16     }

begin
i:=1999;
{recherche du signe du nombre = signe du premier bit significatif de poid}
{fort                                                                    }

while ord(t1p[i]) - ord(t1m[i]) = 0 do i:=i-1;
if ord(t1p[i]) - ord(t1m[i]) = 1 then signe:=1 else signe:=-1;
val:=0;

{application de la formule de recurrence donnant la valeur d'un nombre en}
{redondante modulo 2 à la puissance 16                                   }

if taille >= 15 then
for i:= 2000-taille + 15 downto 2000-taille do
val:=val + val + ord(t1p[i]) - ord(t1m[i])
else
for i:= 1999 downto 2000-taille do
val:=val + val + ord(t1p[i]) - ord(t1m[i]);
if (signe = 1) and (val < 0) then val:=65536 + val
else if (signe=-1) and ( val > 0) then val:= val - 65536;
modulo2_16:=LoWord(val);
end;
```

```
procedure addAB (p:integer); {A*p+B=>B}
begin
am :=0; ap:=0;
if p=1 then
for i:=0 to 1999 do
begin
bp:=ord(tap[i]);
bm:=ord(tam[i]);
cp:=ord(tbp[i]);
cm:=ord(tbm[i]);
xm :=  (bp - bm + cp ) and 1;
tbp[i]:=chr((ap-xm-cm) and 1);
tbm[i]:=chr(am);
am := (-ap + xm + cm +ord(tbp[i]) ) div 2;
ap := (+bp - bm + cp + xm ) div 2;
end;
if p=2 then
for i:=0 to 1999 do
begin
bp:=ord(tap[i]);
bm:=ord(tam[i]);
cm:=ord(tbp[i]);
cp:=ord(tbm[i]);
xm :=  (bp - bm + cp ) and 1;
tbm[i] := chr( (ap - xm - cm ) and 1);
tbp[i] := chr(am);
am := (-ap + xm + cm + ord(tbm[i])) div 2;
ap := (+bp - bm + cp + xm ) div 2;
end;
if p=1 then tetb:=tetb+teta+ap-am;
if p=2 then tetb:=tetb-teta+am-ap;
end; {addAB}

procedure addRS (p:integer); {S*p+R=>R}

begin
am :=0; ap:=0;
if p=1 then
for i:=0 to 1999 do
begin
bp:=ord(tsp[i]);
bm:=ord(tsm[i]);
cp:=ord(trp[i]);
cm:=ord(trm[i]);
xm :=  (bp - bm + cp ) and 1;
trp[i]:=chr((ap-xm-cm) and 1);
trm[i]:=chr(am);
am := (-ap + xm + cm +ord(trp[i]) ) div 2;
ap := (+bp - bm + cp + xm ) div 2;
end else
for i:=0 to 1999 do
begin
bp:=ord(tsp[i]);
```

```
bm:=ord(tsm[i]);
cm:=ord(trp[i]);
cp:=ord(trm[i]);
xm :=  (bp - bm + cp ) and 1;
trm[i] := chr( (ap - xm - cm ) and 1);
trp[i] := chr(am);
am := (-ap + xm + cm + ord(trm[i])) div 2;
ap := (+bp - bm + cp + xm ) div 2;
end;
{contrôle de la propagation de la retenue}
if ap-am <>0 then
begin

   ap:=4*ap+ord(trp[1999])+ord(trp[1999])+ord(trp[1998])
   -(4*am+ord(trm[1999])+ord(trm[1999])+ord(trm[1998]));
   if abs(ap)>3 then
   begin
      {il y'a débordement,donc l'algorithme ne marche pas pour}
      { de tel échantillon! }

   end;

   if ap>=0 then begin
      {il faut réinjecter la retenue et la recoder sur les deux derniers bits}
      {du resultat                                                    }
      trm[1999]:=chr(0); trm[1998]:=chr(0);
      trp[1999]:= chr((ap div 2) and 1);trp[1998]:=chr(ap and 1);
   end else begin
      ap:=-ap;
      trp[1999]:=chr(0); trp[1998]:=chr(0);
      trm[1999]:=chr( (ap div 2) and 1);trm[1998]:=chr(ap and 1);
   end;
end;
if tails > tailr then tailr:=tails;

end; {addRS}

procedure exchAB ;
{échange les nombres A et B}
begin
nbcycle:=nbcycle+1;
for i:=0 to 1999 do
begin
j:=ord(tbp[i]);tbp[i]:=tap[i];tap[i]:=chr(j);
j:=ord(tbm[i]);tbm[i]:=tam[i];tam[i]:=chr(j);
end;
j:=tailb;tailb:=taila;taila:=j;
j:=tetb;tetb:=teta;teta:=j;
end;{exch}

procedure exchRS ;
{échange les valeurs des coefficients de bezout;R et S }
```

```
begin
for i:=0 to 1999 do
begin
j:=ord(tsp[i]);tsp[i]:=trm[i];trm[i]:=chr(j);
j:=ord(tsm[i]);tsm[i]:=trp[i];trp[i]:=chr(j);
end;
j:=tailr;tailr:=tails;tails:=j;
end;{*exchRS*}

procedure shift1R;
{décalage à gauche de R: division par 2 de R : opération inverse de shift1B}
begin

for i := 0 to 1998 do
begin
trp [i] := trp[i+1];
trm [i] := trm[i+1];
end;
trp[1999]:= chr(0);
trm[1999]:= chr(0);
tailr:=tailr + 1;
end {*shift1R*} ;

procedure charger;

{transforme un caractère de ['0'..'9'] U ['A'..'F'] en sa représentation binaire et le}
{charge dans les quatres bits correspendants du tableau tap ou tbp selon le nombre traité}

begin
j:=1995;{C'est le premier bit avec lequel on va commencer à remplir le tableau tap}
read(entree,car);
while car <> '*' do   {Le transformer en binaire sur quatre bits}
begin
if car  <='9' then  x:= ord(car) -ord('0')
          else  x:= ord(car) -ord('A') +10;

{Inscrire les quatres bits dans le tableau tbp puis initialiser le j de nouveau au}
{debut d'un nouveau bloc de quatre bits pour le prochain caractére dela représent-}
{ation hexadécimale du nombre traité                                    }

for i:=1 to 4 do begin  tap[i+j] := tbp[i+j];
                  tbp[i+j]:= chr(x and 1);
                  x:=x div 2 end;
 j:=j-4; read(entree,car);
 end;
 readln(entree);
end; {*charger*}

procedure shift1B;
{décale le nombre B d'une position à droite: multiplication de B par 2:}
{ doubler la valeur de la tête et y injecter la valeur du premier bit  }
{ de poid fort avant de décaler le tout                                 }
```

```
begin
  tetb:=tetb + tetb + ord(tbp[1999]) - ord(tbm[1999]);
  tailb := tailb - 1;
  for i := 1999 downto 1 do
  begin
   tbp [i] := tbp[i-1];
   tbm [i] := tbm[i-1];
  end;
  tbp [0] := chr(0);
  tbm [0] := chr(0);
end;

procedure normalisation;
{transforme un nombre quelconque en représentation redondante en un autre qui}
{est sous la forme normalisée                                    }

begin
while (abs(tetb) < 4) and (tailb > 3) do
begin
nbcycle:=nbcycle+1;
 if ((taila > tailb) or ((taila = tailb ) and (abs(teta) > abs(tetb))))
 and ((ord(tbp[1999])-ord(tbm[1999]))*tetb >= 3) then
 {il faut éviter le cas où la tête vaut 7 qui est une valeur critique}
 begin
  tetb:=tetb + ord(tbp[1999]) - ord(tbm[1999]);
  car:=(tbp[1999]);tbp[1999]:=tbm[1999];tbm[1999]:=car;
 end else
 begin {decaler les zéros jusqu'à le premier chiffre significatif}
    gotoxy(70,2);write('NON') ; shift1B;
    typ[ctyp]:='1'; ctyp:=ctyp+1;
 end;
end;
{si le nombre ainci n'est pas encore normalisé,faire à ce que ça soit!}
if tetb <> 0 then
while (abs(tetb) < 4) and (tailb <= 3) do
begin
tetb := tetb + tetb ;tailb:=tailb-1;
nbcycle:=nbcycle+1;
typ[ctyp] := '1'; ctyp:=ctyp+1;
end
end; {normalisation}


function find_sequence(t1:tableau;t2:tableau;taille:integer): integer;

{compte le nombre de zéros existant dans la plus longue seq de zéros  }
{figurant dans le nombre ayant sa représentation redondante répartie  }
{dans les deux tableaux t1 et t2 et  taille bits significatifs        }

begin
i:=2000 - taille;
seq:=0;
newseq:=0;
```

```
while i<= 1999 do
begin
while ((ord(t1[i])-ord(t2[i])) =0) do
begin
seq:=seq+1;
i:=i+1;
end;
if seq>newseq then newseq:=seq;
seq:=0;i:=i+1;
end;
find_sequence:=newseq;
end;{find_seq}

                {****PROGRAMME  PRINCIPAL****}

begin

                {***INITIALISATION***}

reset(entree,'donnee');
rewrite(bilan,'sortie');
rewrite (printer, 'res');
for i:=2000 to 3999 do typ[i] := '0';
for i:=0 to 1999 do begin tap[i]:=chr(0); tam[i]:=chr(0);
                    tbp[i]:=chr(0); tbm[i]:=chr(0);
                    trp[i]:=chr(0); trm[i]:=chr(0);
                    tsp[i]:=chr(0); tsm[i]:=chr(0);
                     typ[i]:='0';
                end;
writeln('         .....SIMULATION EN COURS.....      ');

                {***LECTURE DU PREMIER NOMBRE ET DE SA TAILLE***}

readln(entree,echantillon);
readln(entree,tailb);initailb:=tailb;oldtail:=initailb;
charger;

    {***LECTURE DU SECOND NOMBRE DU PREMIER COUPLE A TRAITER***}

taila:=tailb; initaila:=taila;
charger;
{calcul de leurs  valeurs modulo 2 à la puissance 16}

valamod2_16:=modulo2_16(tap,tam,taila);
gotoxy(50,1);write('a   ',valamod2_16:5);
valbmod2_16:=modulo2_16(tbp,tbm,tailb);
gotoxy(50,2);write('b   ',valbmod2_16:5);

while initailb <> 0 do {Le dernier couple traité sera = ('*'(0),'*'(0)) }
begin
gotoxy(2,5); write('échantillon  :',echantillon);
gotoxy(2,6);write(' ==========');
gotoxy(6,8);write('taille courante  ',initailb);
```

```
gotoxy(6,10);write('variation de la taille de A et de B:  ');

nbcycle:=0;cyclebezout:=0;
teta:=0; tetb:=0; ctyp:=0 ;

{préparer l'entree de telle façon qu'elle soit traitable par l'approche: les}
{nombres doivent être  normalisés                                          }
exchAB;normalisation;exchAB;normalisation;
ctyp:=0;
taila:=initaila;
tailb:=initailb;

while ((tailb>3) or (abs(tetb)>0)) do
begin
gotoxy(41,10); write(tailb:4,taila:4);gotoxy(55,10);write('tetb ',tetb :4);
if (taila > tailb)  or
((taila = tailb ) and (tailb <=3 ) and (abs(teta) > abs(tetb)))
then
begin{a>b; échanger les deux nombres puisqu'on a choisit dès le debut d'avoir}
    {le cas:  b>a                                                          }

typ[ctyp] := '2'; ctyp:=ctyp+1; {mise à jour de la pile des opérations: typ  }
exchAB ;
end;
if teta*tetb < 0 then
begin

{a et b sont de signe contraire, faire donc l'addition qui diminuera la      }
{valeur du plus grand des deux nombres                                       }

typ[ctyp] := '3' ; ctyp:=ctyp+1; {mise à jour de la pile des opérations: typ }
addAB (1) ;
if tailb >3
then shift1B else  begin tetb := tetb + tetb;tailb:=tailb-1; end;
nbcycle:=nbcycle+1;
end
else
begin

{les deux nombres sont de même signe, soustraire a de b qui dimunira la     }
{valeur du plus grand des deux entiers: le b                                 }

typ[ctyp] := '4' ; ctyp:=ctyp+1;{mise à jour de la pile des opérations: typ}
addAB (2);
if tailb > 3
then shift1B else  begin tetb := tetb + tetb;tailb:=tailb-1; end;
nbcycle:=nbcycle+1;
end;
{dans tous les cas, il faut penser toujours à avoir un résultat normalisé  }
normalisation;savetaila:=taila;
end;

{enregistrer les résultats dans un fichier qu'on pourrait consulter si besoin}
```

```
{*Le PGCD étant calculé,on a le résultat réparti entre les tableaux tap et tam,}

valgcdmod2_16:=teta;
for i:=1 to taila do
valgcdmod2_16:=valgcdmod2_16+valgcdmod2_16+ord(tap[i])-ord(tam[i]);
while valgcdmod2_16 and 1 = 0 do
begin
valgcdmod2_16:=valgcdmod2_16 div 2;
end;
 gotoxy(50,6);write('gcd ',valgcdmod2_16:5);

 ctyp:=ctyp-1;

                {***INITIALISATION DE BEZOUT***}

tailr:=tailb + 1;      trp[2000-tailr]:=chr(1);{premier bit de poid faible}
tails:=taila + 1;      tsp[2000-tails]:=chr(1);{premier bit de poid faible}
gotoxy(6,12);
write('nombre total d''opération: ',ctyp:4);

gotoxy(1,15); for i:=1 to 128 do write('     ');
gotoxy(6,14); writeln('liste des operations faites sur cet échantillon:');

while ctyp >= 0 do
begin

{suivant le type d'opération déjà faite sur a et b pour le calcul de leur }
{pgcd, réagir de telle façon à preserver l'identité de Bezout          }
{on rappèle que la convention utilisée pour numéroter les opérations etait}
{              '1': b <-- b*2                                  }
{              '2': a <-- b; b <-- a;                           }
{              '3': b <-- (a+b)*2                               }
{              '4': a <-- (b-a)*2                               }
{              '2': b <-- b*4;                                 }

 write(typ[ctyp]);
case typ[ctyp] of
'1': begin shift1R; cyclebezout:=cyclebezout+1;end;
'2': begin exchRS;  cyclebezout:=cyclebezout+1;end;
'3': begin shift1R; addRS(2); cyclebezout:=cyclebezout+1 end;
'4': begin shift1R; addRS(1);  cyclebezout:=cyclebezout+1 end;
'5': begin shift1R; shift1R; cyclebezout:=cyclebezout+1 end;
end;
ctyp:=ctyp-1;
end;

{ la sortie du circuit simulé est en représentation redondante, donc il faut }
{ la convertir en binaire; Ce qui n'est pas gratuit! et parsuite, il faut    }
{ calculer son coût qui est proportionnel au nombre de zéros successifs qui  }
{ apparaissent dans la représentation redondante de r et s                   }

if find_sequence(trp,trm,tailr) > find_sequence(tsp,tsm,tails)
then cout:= find_sequence(trp,trm,tailr)
```

```
else cout:= find_sequence(tsp,tsm,tails);

{enregistrer ce resultat }
valrmod2_16:=modulo2_16(trp,trm,tailr);
gotoxy(50,3);write('r  ',valrmod2_16:5);
valsmod2_16:=modulo2_16(tsp,tsm,tails);
gotoxy(50,4);write('s   ',valsmod2_16:5);

{vérification modulo 2 à la puissance 16 des résultats}

val:=(valamod2_16 * valrmod2_16 - valsmod2_16 *valbmod2_16);
if  valgcdmod2_16 = LoWord(val) then
 begin
  gotoxy(55,5);

 end
else
 begin
   gotoxy(55,5);
end;

                {***IMPRIMER UNE LIGNE DU FICHIER BILAN***}

if echantillon=1 then
 write(bilan,initailb:5,chr(9),nbcycle:20,chr(9),cyclebezout:20,chr(9),cout:10)
else
 if initailb = oldtail
   then write(bilan,chr(9),nbcycle:20,chr(9),cyclebezout:20, chr(9),cout:10)
   else
     begin
       oldtail:=initailb;
       writeln(bilan);
       write(bilan,initailb:5,chr(9),nbcycle:20,chr(9),cyclebezout:20, chr(9),cout:10);
     end;

{réinitialiser le tout pour la prochaine itération}

for i:=2000 to 3999 do typ[i] := '0';
for i:=0 to 1999 do begin tap[i]:=chr(0); tam[i]:=chr(0);
                    tbp[i]:=chr(0); tbm[i]:=chr(0);
                    trp[i]:=chr(0); trm[i]:=chr(0);
                    tsp[i]:=chr(0); tsm[i]:=chr(0);
                     typ[i]:='0';
                end;

{***LECTURE DU COUPLE SUIVANT AFFECTE DE LA  TAILLE DES NOMBRES***}

readln(entree,echantillon);
readln(entree,tailb);initailb:=tailb;
charger;
taila:=initailb;
initaila:=taila;
charger;
```

```
{récuperer leur valeur modulo 2 à la puissance 16 avant tout calcul}
valamod2_16:=modulo2_16(tap,tam,taila);
gotoxy(50,1);write('a   ',valamod2_16:5);
valbmod2_16:=modulo2_16(tbp,tbm,tailb);
gotoxy(50,2);write('b   ',valbmod2_16:5);

 if keypressed then initailb:=0;
end;
clearscreen;
gotoxy(5,15);
 writeln('.....SIMULATEUR  CONVERGE.....');

close(entree);
close(bilan);
close(printer);
end.
```

Exemple de calcul du PGCD pour les valeurs d'entrées A = 115510395 et B = 577505775

opérations	A	μa	∂a	B	μb	∂b
init	115510395	6	27	577505775	4	30
sub	115510395	6	27	-346577385	-3	30
add	115510395	6	27	115464195	1	29
shift	115510395	6	27	115464195	7	27
sub	115510395	6	27	-46200	0	27
shift2	115510395	6	27	-46200	0	24
shift2	115510395	6	27	-46200	0	22
shift2	115510395	6	27	-46200	0	20
shift2	115510395	6	27	-46200	-1	18
shift2	115510395	6	27	-46200	-6	16
exch add	-46200	-6	16	20892795	1	27
shift	-46200	-6	16	20892795	5	25
add	-46200	-6	16	-2761605	-1	25
shift2	-46200	-6	16	-2761605	-5	22
sub	-46200	-6	16	195195	0	22
shift2	-46200	-6	16	195195	3	20
shift	-46200	-6	16	195195	6	18
add	-46200	-6	16	10395	0	18
shift2	-46200	-6	16	10395	3	15
shift	-46200	-6	16	10395	5	14
exch add	10395	5	14	-4620	-1	16
shift2	10395	5	14	-4620	-5	13
exch add	-4620	-5	13	1155	1	14
shift2	-4620	-5	13	1155	5	11
exch add	1155	5	11	0	0	13
shift2	1155	5	11	0	0	10
shift2	1155	5	11	0	0	8
shift2	1155	5	11	0	0	6
shift2	1155	5	11	0	0	4
shift2	1155	5	11	0	0	2
shift2	1155	5	11	0	0	0

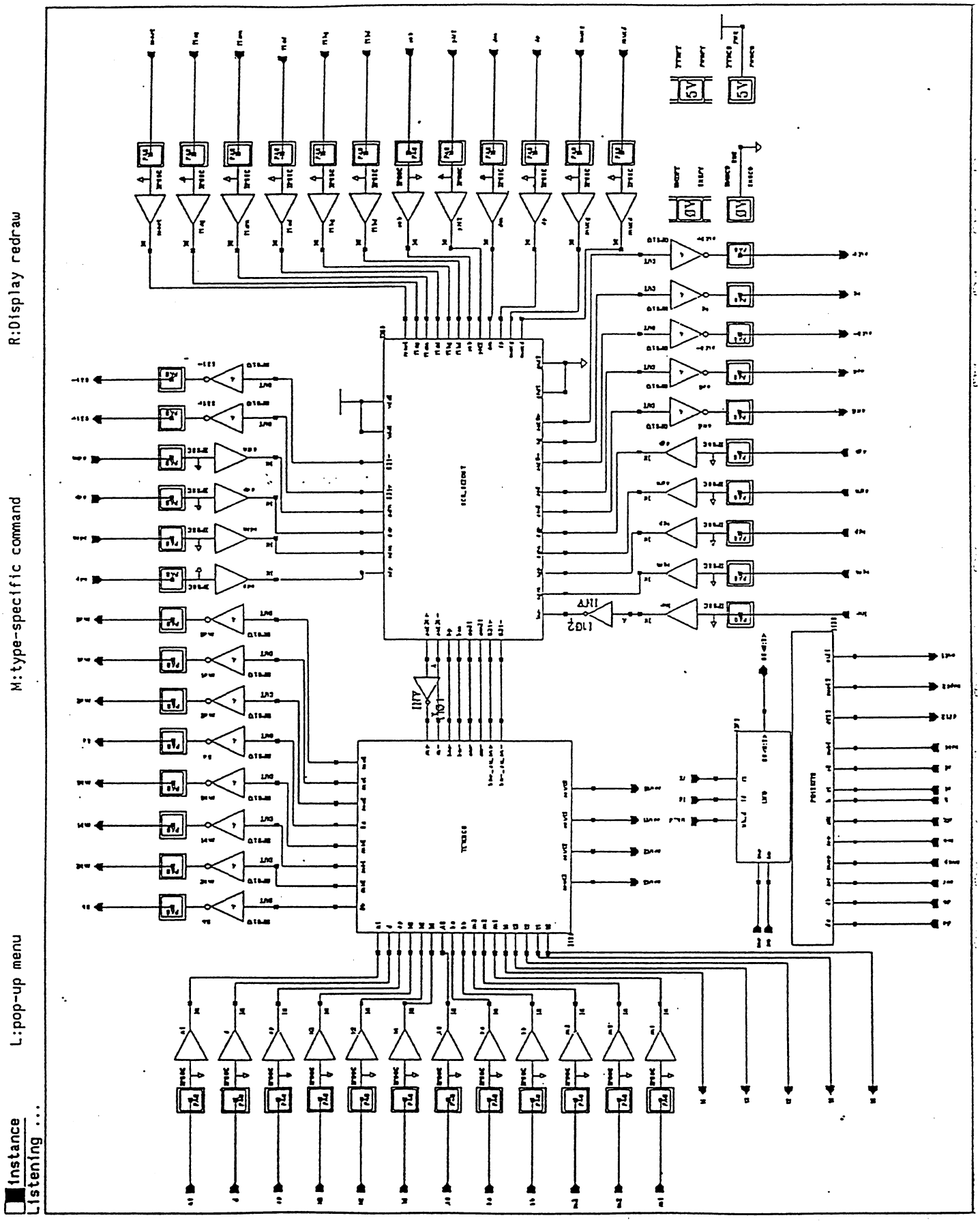
PGCD = 1155 Nombres de bits 11

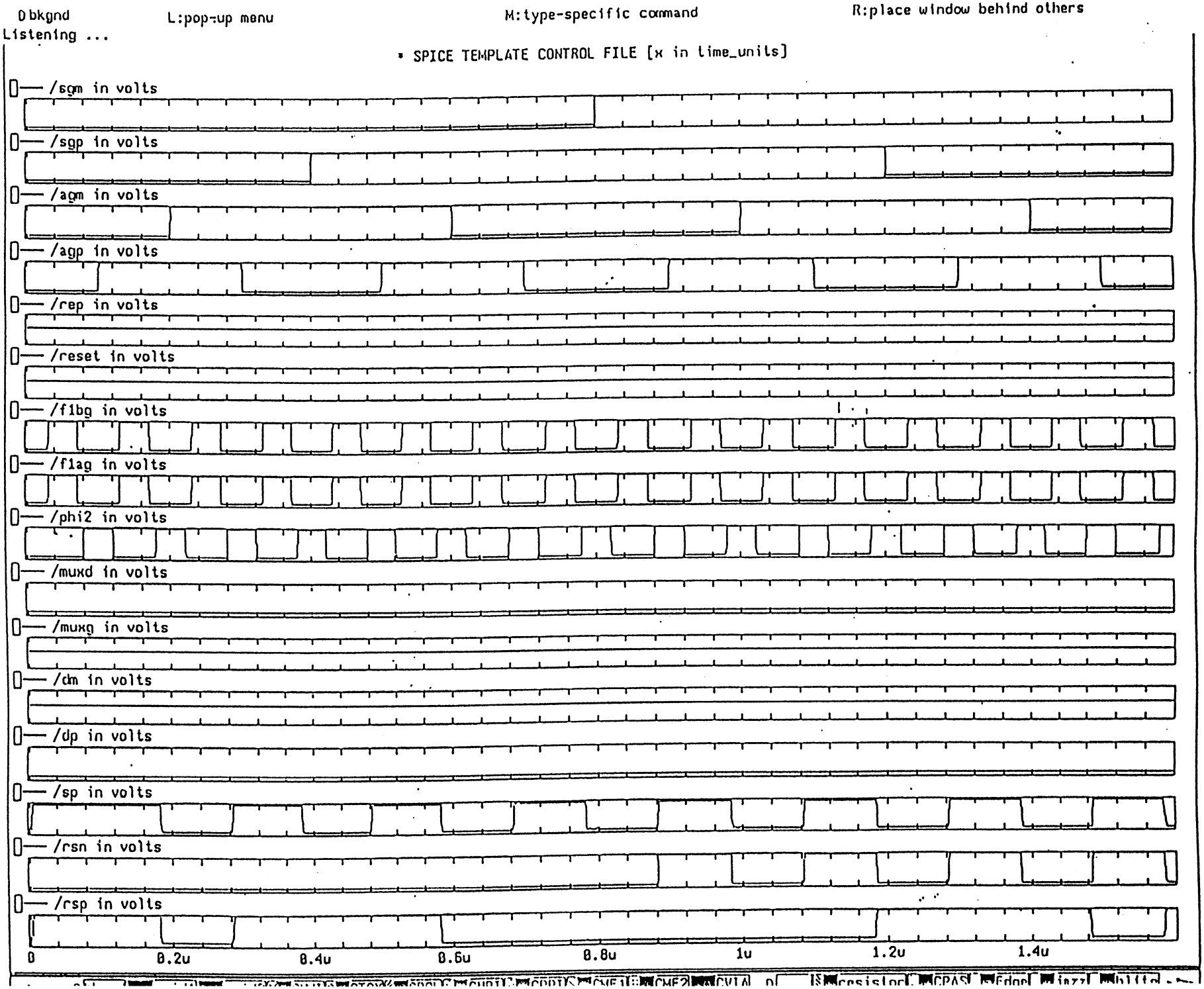
Exemple de calcul du PGCD étendu : R et S tels que R*A - S*B = PGCD(A, B)

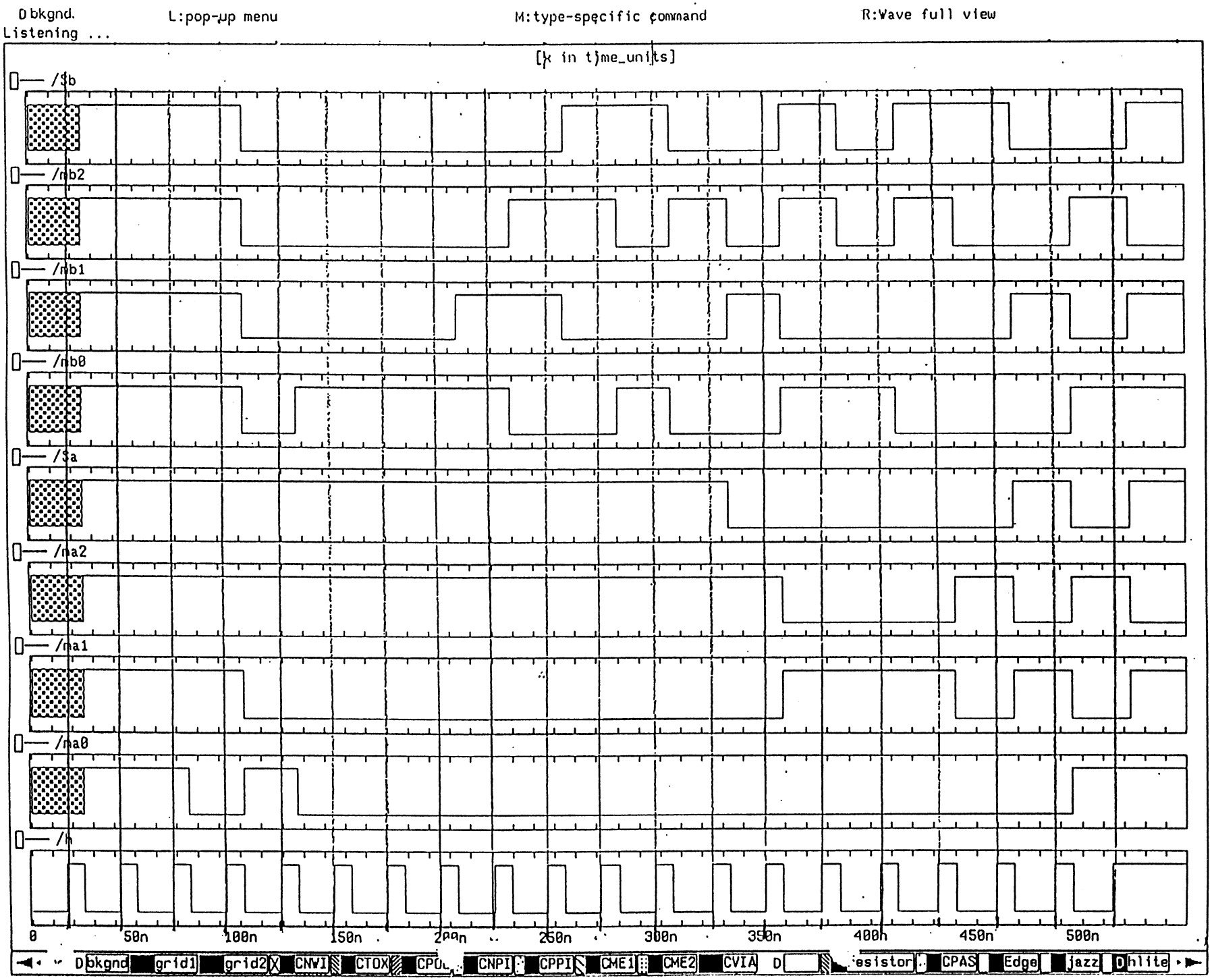
liste de types d'opérations:413114111111111112311311141111311112311123112311123

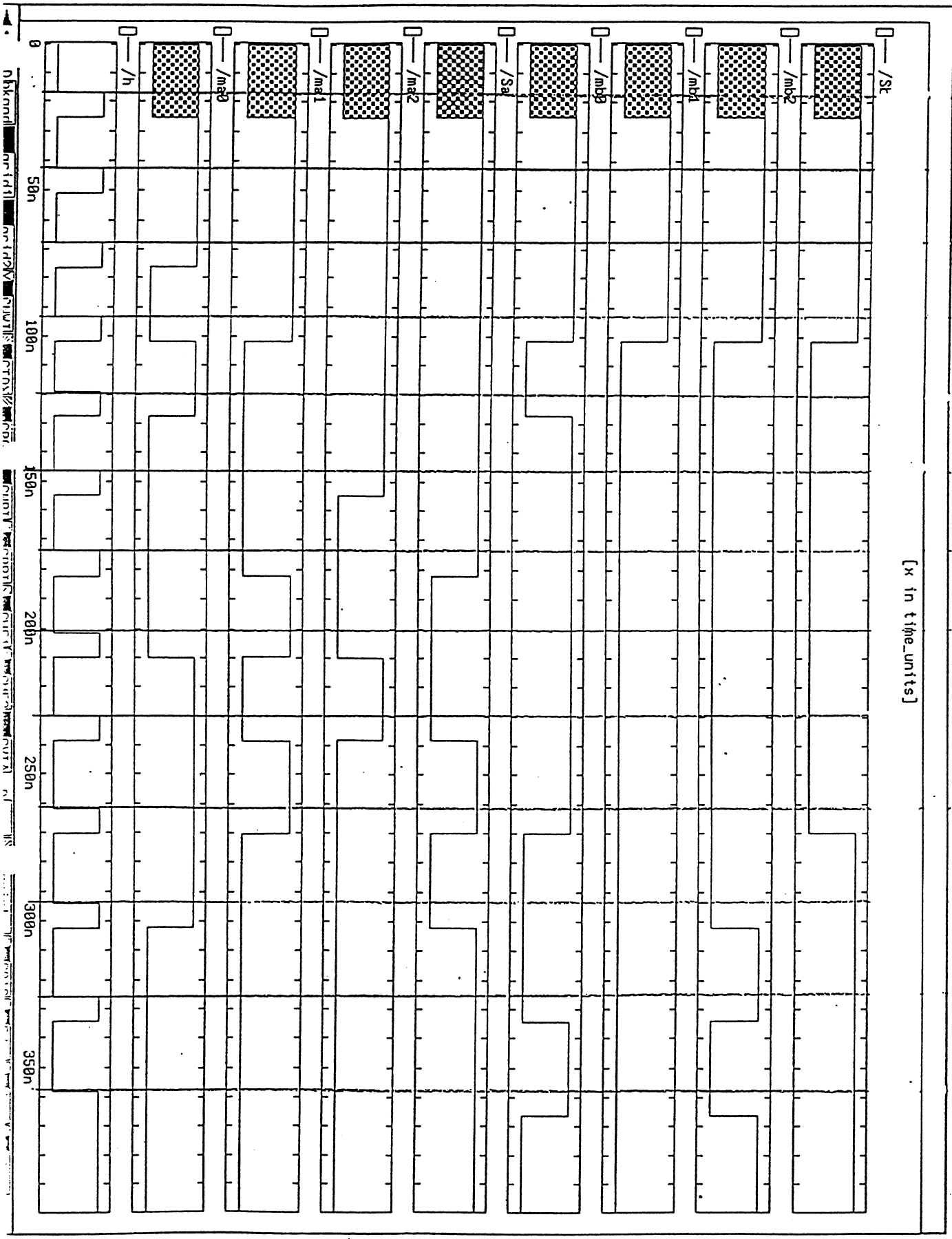
PGCD = 1155

opérations	A	∂a	B	∂b	R	∂r	S	∂s
/	1155	11	0	13	1	13	1	11
46 3	1155	11	-4620	13	5	13	1	11
45 2	-4620	13	1155	11	-1	11	-5	13
44 1	-4620	13	1155	12	-1	12	-5	13
43 1	-4620	13	1155	13	-1	13	-5	13
42 1	-4620	13	1155	14	-1	14	-5	13
41 3	-4620	13	10395	14	-11	14	-5	13
40 2	10395	14	-4620	13	5	13	11	14
39 1	10395	14	-4620	14	5	14	11	14
38 1	10395	14	-4620	15	5	15	11	14
37 1	10395	14	-4620	16	5	16	11	14
36 3	10395	14	-46200	16	49	16	11	14
35 2	-46200	16	10395	14	-11	14	-49	16
34 1	-46200	16	10395	15	-11	15	-49	16
33 1	-46200	16	10395	16	-11	16	-49	16
32 1	-46200	16	10395	17	-11	17	-49	16
31 1	-46200	16	10395	18	-11	18	-49	16
30 3	-46200	16	195195	18	-207	18	-49	16
29 1	-46200	16	195195	19	-207	19	-49	16
28 1	-46200	16	195195	20	-207	20	-49	16
27 1	-46200	16	195195	21	-207	21	-49	16
26 1	-46200	16	195195	22	-207	22	-49	16
25 4	-46200	16	-2761605	22	2929	22	-49	16
24 1	-46200	16	-2761605	23	2929	23	-49	16
23 1	-46200	16	-2761605	24	2929	24	-49	16
22 1	-46200	16	-2761605	25	2929	25	-49	16
21 3	-46200	16	20892795	25	-22159	25	-49	16
20 1	-46200	16	20892795	26	-22159	26	-49	16
19 1	-46200	16	20892795	27	-22159	27	-49	16
18 3	-46200	16	115510395	27	-122511	27	-49	16
17 2	115510395	27	-46200	16	49	16	122511	27
16 1	115510395	27	-46200	17	49	17	122511	27
15 1	115510395	27	-46200	18	49	18	122511	27
14 1	115510395	27	-46200	19	49	19	122511	27
13 1	115510395	27	-46200	20	49	20	122511	27
12 1	115510395	27	-46200	21	49	21	122511	27
11 1	115510395	27	-46200	22	49	22	122511	27
10 1	115510395	27	-46200	23	49	23	122511	27
9 1	115510395	27	-46200	24	49	24	122511	27
8 1	115510395	27	-46200	25	49	25	122511	27
7 1	115510395	27	-46200	26	49	26	122511	27
6 1	115510395	27	-46200	27	49	27	122511	27
5 4	115510395	27	115464195	27	-122462	27	122511	27
4 1	115510395	27	115464195	28	-122462	28	122511	27
3 1	115510395	27	115464195	29	-122462	29	122511	27
2 3	115510395	27	-346577385	29	367582	29	122511	27
1 1	115510395	27	-346577385	30	367582	30	122511	27
0 4	115510395	27	577505775	30	-612506	30	122511	27


R:Display redraw
M:type-specific command
L:pop-up menu
Instance
Listening ...


0 bkgnd
L:pop-up menu
M:type-specific command
R:place window behind others
Listening ...
* SPICE TEMPLATE CONTROL FILE [x in time_units]
/sgm in volts
/sgp in volts
/agm in volts
/agp in volts
/rep in volts
/reset in volts
/f1bg in volts
/f1ag in volts
/phi2 in volts
/muxd in volts
/muxg in volts
/dm in volts
/dp in volts
/sp in volts
/rsn in volts
/rsp in volts
0
0.2u
0.4u
0.6u
0.8u
1u
1.2u
1.4u


D bkgnd.
L:pop-up menu
M:type-specific command
R:Wave full view
Listening ...
[x in time_units]
/$b
/nb2
/nb1
/nb0
/$a
/na2
/na1
/na0
/h
0
50n
100n
150n
200n
250n
300n
350n
400n
450n
500n
D bkgnd grid1 grid2 CNVI CTOX CPOL CNPI CPPI CME1 CME2 CVIA D esistor CPAS Edge jazz Dhlite


D:bkgnd
Listening ...
L:pop-up menu
M:type-specific command
R:Wave set anchor
[x in time_units]
/St
/mb2
/mb1
/mb0
/Sa
/ma2
/ma1
/ma0
/h
0
50n
100n
150n
200n
250n
300n
350n

Annexe 2 : Résultats de simulations de l'opérateur euclidien

1. La description du programme

Le programme 1 (Appendice A) est écrit pour recevoir deux nombres en format décimal et produire un résultat en 9 chiffres signés. Ce programme permet de vérifier une description matérielle. L'entrée n'est pas protégée pour limiter en 6 chiffres signés la taille des nombres d'entrées. Le programme ne vérifie pas non plus qu'au moins l'un des nombres est différent de zéro. Si les deux nombres sont égaux à zéro, alors le programme entre dans une boucle infinie par la condition "flag = 1 OR counter! = 0" qui ne finera jamais. De plus, le résultat exige plusieurs traitements après son édition. Les contenus de la variable Z sont édités en décimal. Cette valeur n'est pas normalisée et il est donc nécessaire de diviser en répétition le nombre décimal par deux jusqu'à ce que le résultat correct soit trouvé. Dans l'exemple ci dessous le résultat exige d'être divisé par 2^4.

Entrer le 1er nombre, X: -5	Entrer le 2nd nombre, Y: 12	
-1	0	Z 1
0	0	Z -1
-1	1	Z 1
0	1	Z 1
0	0	Z -1
0	0	Z 0
0	0	Z 0
0	0	Z 0
0	0	Z 0
		résultat $208 \div 2^4 = 13$

Les différentes fonctions ont été écrites sous la forme d'une description comportementale de chacun des opérateurs présentés dans le chapitre VII §VII.2. Elles ont été vérifiées en écrivant des programmes séparément pour l'addition et multiplication.

2. La solution du probléme $\sqrt{2}$

Dans sa première version, le programme a donné pour certaines entrées, un résultat biaisé d'un facteur multiplicatif de $\sqrt{2}$ [ERT 77]. En entrant les valeurs 5 et 12, nous devons obtenir un résultat 13, au lieu 18,375 (294 ÷ 16). La première cause possible est qu'une quantité

insuffisante de la variable Zcarry est utilisée dans la boucle de retour, ce qui limiterait l'étendu des valeurs que la routine peut sortir. Ceci est modifié, la variable Zcarry est maintenant décalée de 2 places vers les poids forts, fournissant deux chiffress r et q pour la routine de la boucle de retour.

Ce changement n'est pas suffisant pour résoudre le problème en totalité. Il est devenu nécessaire de déterminer le moment où le compteur devient différent de zéro pour pouvoir charger le premier chiffre dans le registre-SP de X ou Y. Ce qui à son tour indique la position dans le registre-SP de Z pour charger le premier chiffre du résultat. L'algorithme est le suivant :

si (le nombre de cycles d'horloge depuis le chargement du 1er chiffre dans les registres-SP de X et/ou Y est pair) **alors**

charger le chiffre du résultat dans MSD - 1

sinon

charger le chiffre du résultat dans MSD

Un autre paramètre important est de savoir l'endroit du début de chargement des données dans les registres-SP de X et Y. Si le résultat Z contient n chiffres, alors, pour obtenir le fonctionnement correct du circuit, il est nécessaire de commencer à charger les entrées des registres-SP à partir de la $(n - 3)^{ième}$ position. Cela conduit à un retard en ligne entre 3 et 5 cycles d'horloges selon la valeur des variables d'entrées.

Cinq paires de valeurs se sont avérées critiques pour déterminer le bon fonctionnement de l'algorithme. Ces valeurs sont 3 et 4, 5 et 12, 7 et 17, 23 et 23, 25 et 25. Une fois les résultats corrects obtenus pour ces valeurs, deux tests supplémentaires sont effectués. Le premier est d'obtenir les résultats pour toutes les paires des nombres entre 1 et 100 et de vérifier leur exactitude. Le second est de dérouler le programme mais seulement pour vérifier que la valeur de la variable "counter" reste bornée. Cela est fait pour un domaine de 1 à 1000 et a montré que le compteur est compris entre -5 et 5 (décimal).

3. Version au niveau porte de l'opérateur euclidien

Une fois que nous sommes certains que l'algorithme global fonctionne correctement, le programme 1 est modifié en programme 2 (Appendice B). Il a été fait dans le but d'aider à la mise au point d'un circuit (chapitre VII § VII.3) qui a été conçu pour implanter l'opérateur euclidien. Ce deuxième programme permet de modifier l'exactitude du résultat en changeant la valeur de **bt**. La valeur maximale de cette variable est 31.

Appendice A

Programme 1

```
/* Comment:  x++ is the same as x = x + 1, ie increment the contents of x */

#include </usr3/syc/walker/p2c/p2c.h>      /* this contains all the */
                                            /* standard header files */

#define p      0   /* positive bit */
#define m      1   /* negative bit */

#define AND &&      /* means that && is replaced by AND for easy of reading */
#define OR  | |     /* means that || is replaced by OR for easy of reading */

typedef long digit[2];     /* definition of arrays */
typedef digit number[9];


Static long XYrang, Zrang, start, flag, b, odd_flag, counter, result;
Static number X, Y, Z, Xcarry, Ycarry, Zcarry, Xparallel, Yparallel,
             Zparallel, XYsum, XYZsum, Total_sum, negZ, retain;
Static digit g, t, s, w, o, q, r;

/*****************************************************************************/

Static Void resdigit(x)      /* reset function */
long *x;
{
 x[p] = 0;
 x[m] = 0;
}


Static Void reset(x)         /* reset function */
digit *x;
{
 long rang;

 for (rang = 0; rang <= 8; rang++) {
   x[rang][p] = 0;
   x[rang][m] = 0;
  }
}


Static long value(x)  /* returns 1, 0, -1 */
long *x;
{
 return (x[p] - x[m]);
}
```

```
Static Void trace(name, t)   /* permits contents of registers to be followed */
Char *name;
digit *t;
{
 long index;

 printf("%s  ", name);
 for (index = 0; index <= 8; index++)
  printf("%3ld", value(t[8 - index]));
 putchar('\n');
}


Static Void lecture(coded)     /* load data */
digit *coded;
{
 long input, rang, next, binary;

 scanf("%ld", &input);
 next = 1;
 for (rang = 0; rang <= 8; rang++) {
  binary = (input / next) % 2;   /* mod 2 */
  printf("%12ld\n", binary);
  next *= 2;
  if (binary == 0)  {
   coded[rang][p] = 0;
   coded[rang][m] = 0;
  }
  if (binary == 1) {
   coded[rang][p] = 1;
   coded[rang][m] = 0;
  }
  if (binary == -1) {
   coded[rang][p] = 0;
   coded[rang][m] = 1;
  }
 }
}

/*************************************************************************/

Local Void ppm(in1p, in2p, inm, outm, outp)  /* ppm */
long in1p, in2p, inm, *outp, *outm;
{
 *outm = (in1p + in2p + inm) % 2;          /* % = mod */
 *outp = (in1p + in2p - inm + *outm) / 2;   /* / = div */
}

Static Void addition(operand1, operand2, sum, carry)       /* Adder*/
digit *operand1, *operand2, *sum;
long *carry;
{
 long rang;
```

```
 number interim;            /* this is used to pass values between ppm's */

 interim[0][p] = 0;
 sum[0][m] = 0;

 for (rang = 0; rang <= 7; rang++) {
  ppm(operand1[rang][p], operand2[rang][p], operand1[rang][m],
       &interim[rang][m], interim[rang + 1]);
  ppm(operand2[rang][m], interim[rang][m], interim[rang][p],
       sum[rang], &sum[rang + 1][m]);
 }

 ppm(operand1[8][p], operand2[8][p], operand1[8][m],
    &interim[8][m], carry);
 ppm(operand2[8][m], interim[8][m], interim[8][p],
    sum[8], &carry[m]);
}


Static Void SPregister(weight, serial, parallel)  /* loads SPregister*/
long weight;
long *serial;
digit *parallel;
{
 parallel[weight - 1][m] = serial[m];            /* jth position with aj */
 parallel[weight - 1][p] = serial[p];
 if (weight > 1) {
  parallel[weight - 2][m] = serial[p];           /* j-1 with -aj       */
  parallel[weight - 2][p] = serial[m];
 }
}


Static Void recursionreg(input, output, carry)      /* left shifts contents of */
digit *input, *output;                              /* recursion register     */
long *carry;
{
 long rang;

 for (rang = 1; rang <= 8; rang++) {
  output[rang][m] = input[rang - 1][m];
  output[rang][p] = input[rang - 1][p];
 }
 carry[p] = input[8][p];
 carry[m] = input[8][m];
 output[0][m] = 0;
 output[0][p] = 0;
}

Static Void multiplication(multiplier, multiplicant, product)
long *multiplier;
digit *multiplicant, *product;        /* vector by digit multiplication */
{
```

```
  long rang;

  if (multiplier[m] == 0 AND multiplier[p] == 1) {
   for (rang = 0; rang <= 8; rang++) {
     product[rang][p] = multiplicant[rang][p];
     product[rang][m] = multiplicant[rang][m];
   }
   return;
  }

  if (multiplier[m] == 1 AND multiplier[p] == 0) {
   for (rang = 0; rang <= 8; rang++) {
     product[rang][p] = multiplicant[rang][m];
     product[rang][m] = multiplicant[rang][p];
   }
   return;
  }

  for (rang = 0; rang <= 8; rang++) {
    product[rang][p] = 0;
    product[rang][m] = 0;
  }
}


Static Void feedback(count, output)  /* determines output */
long count;
long *output;
{
 if (count == 0) {
   output[p] = 0;
   output[m] = 0;
   return;
 }
 if (count < 0) {
   output[p] = 0;
   output[m] = 1;
 } else {
   output[p] = 1;
   output[m] = 0;
 }
}


main()          /* Main part of program */
{
/* Clear all variables */
 reset(X);
 reset(Y);
 reset(Z);
 reset(Xparallel);
 reset(Yparallel);
 reset(Zparallel);
```

```
reset(Xcarry);
reset(Ycarry);
reset(Zcarry);
reset(XYsum);
reset(XYZsum);
reset(negZ);
reset(Total_sum);
reset(retain);
resdigit(w);
resdigit(g);
resdigit(t);
resdigit(q);
resdigit(r);
resdigit(s);
counter = 0;
start = 0;
Zrang = 1;
XYrang = 1;
flag = 0;
odd_flag = 0;


printf("Enter first number, X:");   /* Read in Numbers */
lecture(X);
printf("Enter second number, Y:");
lecture(Y);

/* Find first non-zero digit in either number */
while ((value(X[5 - start]) == 0) AND (value(Y[5 - start]) == 0))
 start++;

while (XYrang < 9 OR Zrang < 9) {

 if (XYrang < 8 - start) {
  SPregister(7 - XYrang, X[7 - start - XYrang], Xparallel); /* X register */
  multiplication(X[7 - start - XYrang], Xparallel, Xcarry);

  SPregister(7 - XYrang, Y[7 - start - XYrang], Yparallel); /* Y register */
  multiplication(Y[7 - start - XYrang], Yparallel, Ycarry);

  addition(Xcarry, Ycarry, XYsum, w);   /* Add to get sum of 2 squares */
 } else
  reset(XYsum);

 SPregister(10 - Zrang - odd_flag, Z[9 - Zrang], Zparallel); /* Z register */
 multiplication(Z[9 - Zrang], Zparallel, Zcarry);
  for (b = 2; b <= 8; b++) {
   negZ[b][p] = Zcarry[b - 2][m];          /* negate contents of Z reg. */
   negZ[b][m] = Zcarry[b - 2][p];          /* and multiply by 4         */
  }
 memcpy(q, Zcarry[8], sizeof(digit));      /* remove MSB for counter */
 memcpy(r, Zcarry[7], sizeof(digit));      /* remove next MSB        */
 addition(XYsum, negZ, XYZsum, s);         /* add the three numbers */
```

```
  addition(XYZsum, retain, Total_sum, g); /* retain is the contents of the */
  recursionreg(Total_sum, retain, t);     /* recursion register           */

  if (flag == 1 OR counter != 0) {        /* indicates when to start */
   Zrang++;                               /* forming the result      */
   flag = 1;
  }


                                          /* determines the next value of Z */
  counter = counter*2 + value(t) + value(g)*2 + value(s)*2 + value(w)*2
            - value(q)*4 - value(r)*2;
  feedback(counter, Z[9 - Zrang]);

  if ((flag == 0 AND counter != 0 AND XYrang %2 == 0) OR odd_flag == 1)
   odd_flag = 1;

  XYrang++;

 }
 /* output result */
 result = 0;
 for (Zrang = 0; Zrang <= 8; Zrang++)
  {result = result*2 + value(Z[8 - Zrang]);
   printf("Z      %12ld\n", value(Z[8 - Zrang]));
 }
 printf("result%12ld\n", result);
 exit(0);
}
```

Appendice B

Programme 2

```
/* Comment:  x++ is the same as x = x + 1, ie increment the contents of x */

#include </usr3/syc/walker/p2c/p2c.h>      /* this contains all the */
                                            /* standard header files */

#define p      0  /* positive bit */
#define m      1  /* negative bit */

#define AND &&      /* means that && is replaced by AND for easy of reading */
#define OR  | |     /* means that || is replaced by OR for easy of reading */

#define bt 31       /* this line defines the number of digits in the result */

typedef long digit[2];     /* definition of arrays */
typedef digit number[bt];

Static long XYrang, Zrang, start, flag, b, odd_flag, transfer, result;
Static digit g, t, s, w, q, r;
Static number X, Y, Z, Xparallel, Yparallel, Zparallel, Xcarry, Ycarry, Zcarry,
            XYsum, XYZsum, Total_sum, negZ, retain, old_value, new_value;

/*************************************************************************/

Static Void resdigit(x)     /* reset function */
long *x;
{
 x[p] = 0;
 x[m] = 0;
}

Static Void setdigit(x)     /* reset function */
long *x;
{
 x[p] = 1;
 x[m] = 1;
}

Static Void reset(x)        /* reset function */
digit *x;
{
 long rang;

 for (rang = 0; rang <= (bt-1); rang++) {
  x[rang][p] = 0;
  x[rang][m] = 0;
 }
}
```

```
Static Void set(x)        /* set function */
digit *x;
{
 long rang;

 x[0][p] = 0;
 x[0][m] = 0;
 for (rang = 1; rang <= (bt-1); rang++) {
  x[rang][p] = 1;
  x[rang][m] = 1;
 }
}
Static long value(x)  /* returns 1, 0, -1 */
long *x;
{
 return (x[p] - x[m]);
}
Static Void trace(name, t)              /* permits contents of registers to be watched */
Char *name;
digit *t;
{
 long index;

 printf("%s  ", name);
 for (index = 0; index <= (bt-1); index++)
  printf("%3ld", value(t[(bt-1) - index]));
 putchar('\n');
}
Static Void lecture(coded)              /* load data */
digit *coded;
{
 long input, rang, next, binary;

 scanf("%ld", &input);
 next = 1;
 for (rang = 0; rang <= (bt-1); rang++) {
  binary = (input / next) % 2;                  /* mod 2 */
  printf("%12ld\n", binary);
  next *= 2;
  if (binary == 0)  {
   coded[rang][p] = 0;
   coded[rang][m] = 0;
  }
  if (binary == 1)  {
   coded[rang][p] = 1;
   coded[rang][m] = 0;
  }
  if (binary == -1) {
   coded[rang][p] = 0;
   coded[rang][m] = 1;
  }
 }
}
```

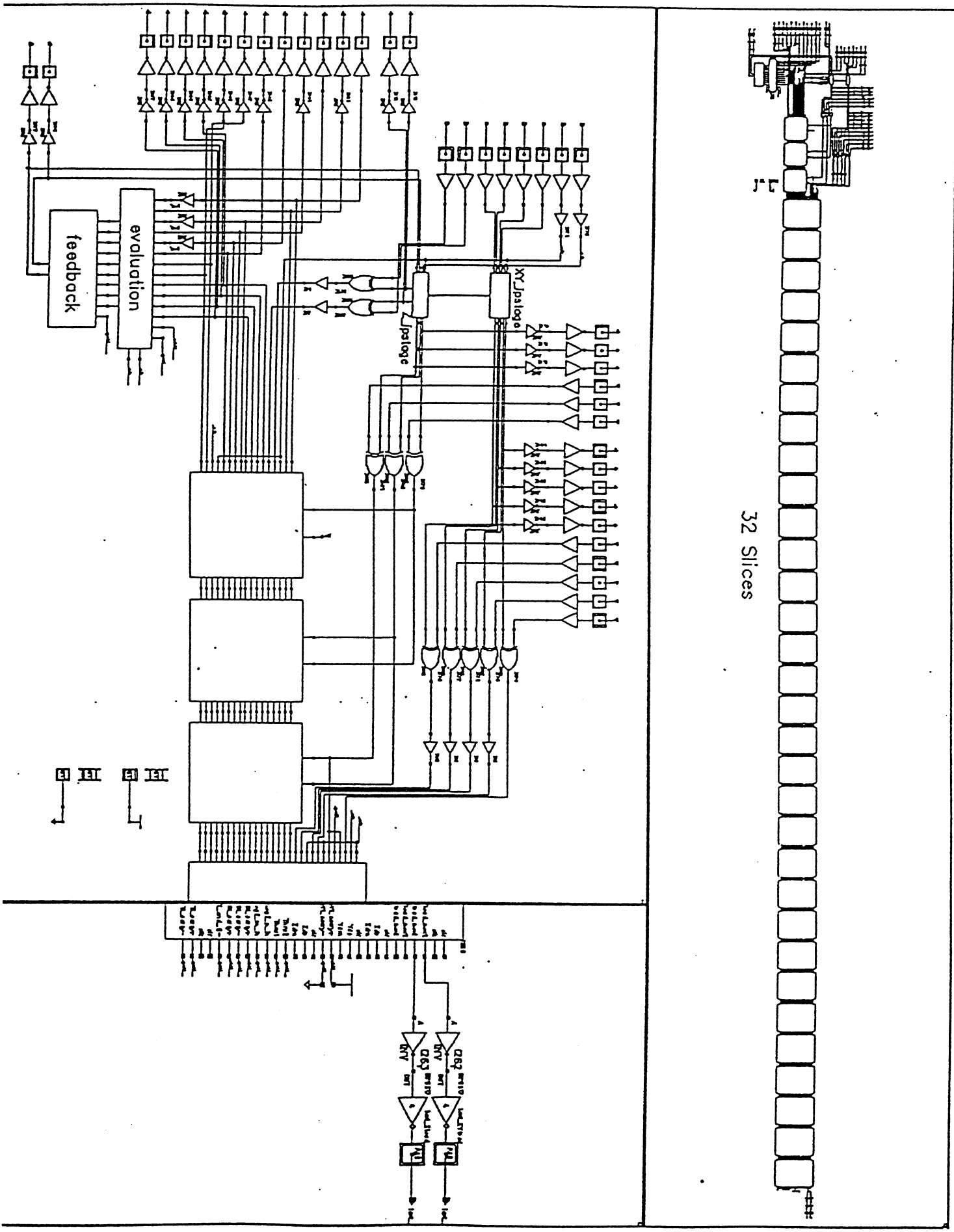
```
/*************************************************************************/

Local Void ppmA(in1p, in2p, inm, outm, outp)      /* ppmA */
long in1p, in2p, inm, *outp, *outm;
{
 *outp = !((in1p AND in2p) OR ((!inm) AND in1p) OR ((!inm) AND in2p));
 *outm = !((*outp AND (in1p OR in2p OR (!inm))) OR (in1p AND in2p AND (!inm)));
}
Local Void ppmB(in1m, in2m, inp, outp, outm)     /* ppmB */
long in1m, in2m, inp, *outm, *outp;
{
 *outm = ((in1m AND in2m) OR (inp AND in1m) OR (inp AND in2m));
 *outp = !(((!*outm) AND (in1m OR in2m OR inp)) OR (in1m AND in2m AND inp));
}
Static Void addition(operand1, operand2, sum, carry)  /* Adder*/
digit *operand1, *operand2, *sum;
long *carry;
{
 long rang;
 number interim;

 interim[0][p] = 1;
 sum[0][m] = 0;
 for (rang = 0; rang <= (bt-2); rang++) {
  ppmA(operand1[rang][p], operand2[rang][p], operand1[rang][m],
       &interim[rang][m], interim[rang + 1]);
  ppmB(interim[rang][m], operand2[rang][m], interim[rang][p],
       sum[rang], &sum[rang + 1][m]);
 }
 ppmA(operand1[bt-1][p], operand2[bt-1][p], operand1[bt-1][m],
    &interim[bt-1][m], carry);
 ppmB(interim[bt-1][m], operand2[bt-1][m], interim[bt-1][p],
    sum[bt-1], &carry[m]);
}
Static Void SPregister(weight, serial, parallel)       /* loads SP-register*/
long weight;
long *serial;
digit *parallel;
{
 parallel[weight][m] = serial[m];
 parallel[weight][p] = serial[p];
 if (weight > 1) {
  parallel[weight - 1][m] = serial[p];
  parallel[weight - 1][p] = serial[m];
 }
}
Static Void recursionreg(input, output, carry) /* alters weight on contents */
digit *input, *output;                  /* of recursive register    */
long *carry;
{
 long rang;
```

```
  for (rang = 1; rang <= (bt-1); rang++) {
   output[rang][m] = input[rang - 1][m];
   output[rang][p] = input[rang - 1][p];
  }
  carry[p] = input[bt-1][p];
  carry[m] = input[bt-1][m];
  output[0][m] = 0;
  output[0][p] = 0;
}

Static Void multiplication(multiplier, multiplicant, product)    /* vector by digit multiplication*/
long *multiplier;
digit *multiplicant, *product;
{
 long rang;

   for (rang = 0; rang <= (bt-1); rang++) {
    product[rang][p] = !((multiplier[p] AND multiplicant[rang][m]) OR (multiplier[m] AND
multiplicant[rang][p]));
    product[rang][m] = !((multiplier[p] AND multiplicant[rang][p]) OR (multiplier[m] AND
multiplicant[rang][m]));
   }
}


/* This function takes the place of the counter. */
/* However, it carries out the same operation.  */
Static Void evaluation(old_counter, w, s, g, t, q, r, new_counter)
long *w, *s, *g, *t, *q, *r;
digit *old_counter, *new_counter;
{
digit a, b, x0, x1, y0, y1;
number interim;

 ppmA(g[p], s[p], r[p], &b[m], a);
 a[p] = !a[p];
 ppmA(g[m], s[m], r[m], b, &a[m]);
 a[m] = !a[m];

 ppmA(old_counter[1][p], a[p], q[p], &x0[m], x1);
 x1[p] = !x1[p];
 ppmA(old_counter[1][m], a[m], q[m], x0, &x1[m]);
 x1[m] = !x1[m];
 ppmA(old_counter[0][p], b[p], 0, &y0[m], y1);
 y1[p] = !y1[p];
 ppmA(old_counter[0][m], b[m], 0, y0, &y1[m]);
 y1[m] = !y1[m];

 ppmA(0, t[p], 0, &interim[1][m], interim[1]);
 ppmA(y0[p], w[p], y0[m], &interim[2][m], interim[2]);
 ppmA(x0[p], y1[p], x0[m], &interim[3][m], interim[3]);
 ppmA(old_counter[2][p], x1[p], old_counter[2][m], &interim[4][m], interim[4]);
 ppmA(old_counter[3][p], 0, old_counter[3][m], &interim[5][m], interim[5]);
```

```
 new_counter[0][m] = 0;
 ppmB(interim[1][m], t[m], 1, new_counter[0], &new_counter[1][m]);
 ppmB(interim[2][m], w[m], interim[1][p], new_counter[1], &new_counter[2][m]);
 ppmB(interim[3][m], y1[m], interim[2][p], new_counter[2], &new_counter[3][m]);
 ppmB(interim[4][m], x1[m], interim[3][p], new_counter[3], &new_counter[4][m]);
 ppmB(interim[5][m], 0, interim[4][p], new_counter[4], &new_counter[5][m]);
}


/* this function is closer to the gate level than the previous version */
Static Void feedback(counter_out, newZ)
digit *counter_out;
long *newZ;
{
 if (value(counter_out[3]) != 0) {
     newZ[p] = counter_out[3][p];
     newZ[m] = counter_out[3][m];
     }
   else {
    if (value(counter_out[2]) != 0) {
        newZ[p] = counter_out[2][p];
        newZ[m] = counter_out[2][m];
        }
      else {
       if (value(counter_out[1]) != 0) {
           newZ[p] = counter_out[1][p];
           newZ[m] = counter_out[1][m];
           }
         else {
          if (value(counter_out[0]) != 0) {
            newZ[p] = counter_out[0][p];
            newZ[m] = counter_out[0][m];
            }
           else {
             newZ[p] = 0;
             newZ[m] = 0;
          }
       }
    }
 }
}

main()          /* Main part of program */
{
/* Clear all variables */
 reset(X); reset(Y); reset(Z);
 reset(Xparallel); reset(Yparallel); reset(Zparallel);
 reset(Xcarry); reset(Ycarry); reset(Zcarry);
 reset(XYsum);
 reset(XYZsum);
 reset(negZ);
 reset(Total_sum);
 reset(retain);
```

```
reset(new_value);
reset(old_value);
resdigit(w);
resdigit(g);
resdigit(t);
resdigit(q);
resdigit(r);
resdigit(s);
start = 0;
Zrang = 1;
XYrang = 1;
flag = 0;
odd_flag = 0;

printf("Enter first number, X:");  /* Read in Numbers */
lecture(X);
printf("Enter second number, Y:");
lecture(Y);

/* Find first non zero digit in either number */
while ((value(X[(bt-1) - start]) == 0) AND (value(Y[(bt-1) - start]) == 0))
 start++;

while (XYrang < bt OR Zrang < bt) {

 if (XYrang < bt + 1 - start) {
  SPregister(bt-3 - XYrang, X[bt - start - XYrang], Xparallel); /* fill X */
  multiplication(X[bt - start - XYrang], Xparallel, Xcarry);  /* register */

  SPregister(bt-3 - XYrang, Y[bt - start - XYrang], Yparallel); /* fill Y */
  multiplication(Y[bt - start - XYrang], Yparallel, Ycarry);  /* register */

  addition(Xcarry, Ycarry, XYsum, w);  /* Add to get sum of 2 squares */
  w[p] = (!w[p]);
 } else {
  set(XYsum);
  setdigit(w);
 }

 SPregister(bt - Zrang - odd_flag, Z[bt - Zrang], Zparallel);  /* fill Z register */
 multiplication(Z[bt - Zrang], Zparallel, Zcarry);
  for (b = 2; b <= (bt-1); b++) {
   negZ[b][p] = Zcarry[b - 2][m];      /* negate contents of Z reg. */
   negZ[b][m] = Zcarry[b - 2][p];      /* and multiply by 4        */
  }
  memcpy(q, Zcarry[bt-1], sizeof(digit));  /* remove MSB for counter */
  memcpy(r, Zcarry[bt-2], sizeof(digit));  /* remove next MSB       */

 addition(XYsum, negZ, XYZsum, s);     /* add the three numbers */
 s[p] = (!s[p]);
 addition(XYZsum, retain, Total_sum, g); /* retain is the contents of the */
 g[p] = (!g[p]);                  /* recursion register          */
 recursionreg(Total_sum, retain, t);
```

```
  if (flag == 1 OR value(Z[bt-Zrang]) != 0) {
    Zrang++;
    flag = 1;
  }

 evaluation(old_value, w, s, g, t, q, r, new_value);
 feedback(new_value, Z[bt - Zrang]);

 /* transfers new_value to old_value */
 for (transfer = 0; transfer <= (bt-1); transfer++) {
  old_value[transfer][p] = new_value[transfer][p];
  old_value[transfer][m] = new_value[transfer][m];
 }

 if ((flag == 0 AND value(Z[bt-Zrang]) != 0 AND XYrang%2 == 0) OR odd_flag == 1)
    odd_flag = 1;

  XYrang++;
 }

 /* output result */
 result = 0;
 for (Zrang = 0; Zrang <= (bt-1); Zrang++)
   {result = result*2 + value(Z[(bt-1) - Zrang]);
    printf("Z       %12ld\n", value(Z[(bt-1) - Zrang]));
 }
 printf("result%12ld\n", result);
 exit(0);
}
```


0 bkgnd
Listening ...
L:pop-up menu
M:type-specific command
R:Configuration crtGridOff
circuit4
0 122.0438 17.9500
feedback
evaluation
XY_lpslogic
lpslogic
32 Slices

Exemple de calcul de la distance euclidienne pour les valeurs d'entrées 3 et 5

Entrée du premier nombre, X:3

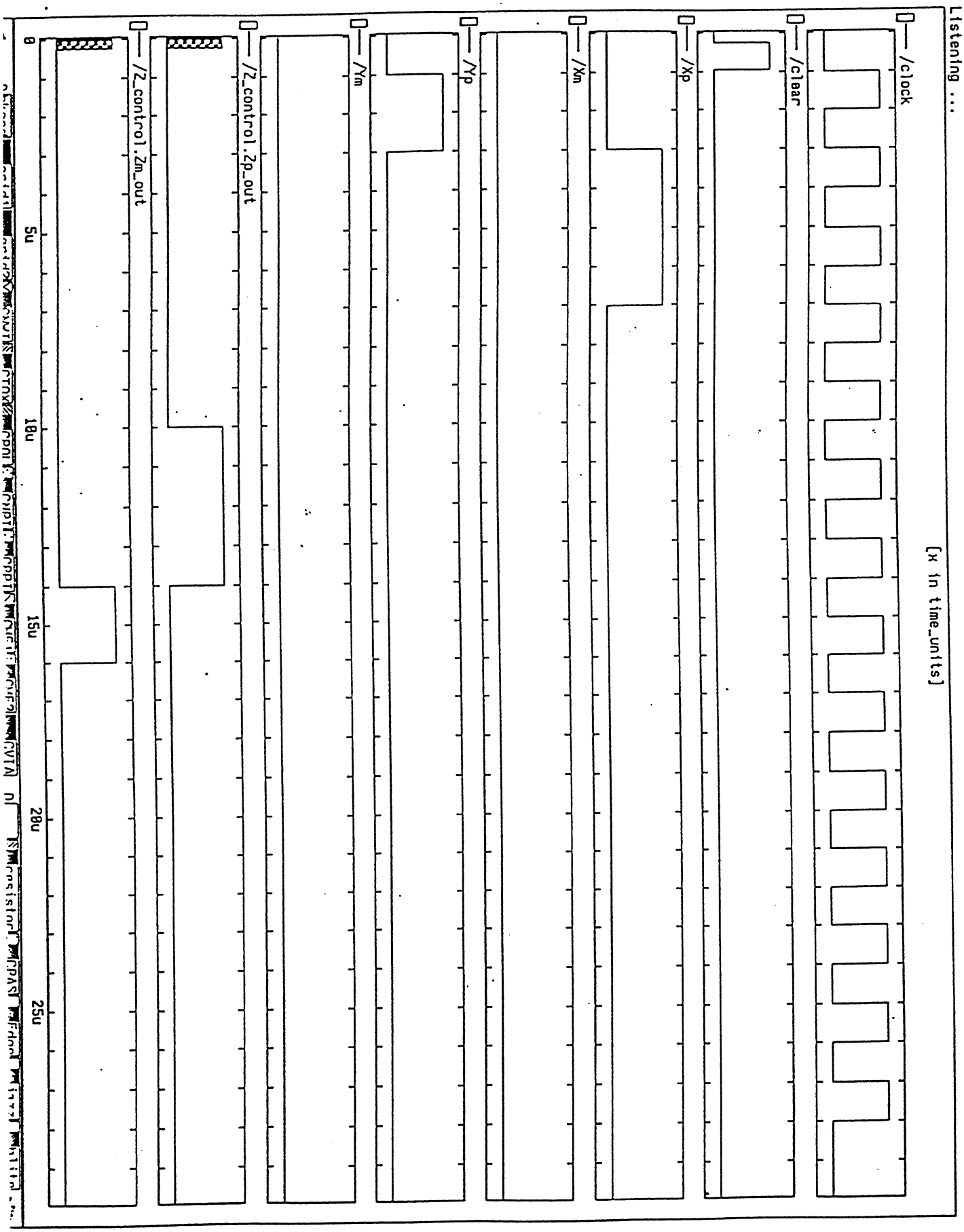
1
1
0
0
0
0
0
0
0

Entrée du second nombre, Y:5

0
0
1
0
0
0
0
0
0

Z	1
Z	1
Z	-1
Z	0
Z	0
Z	0
Z	0
Z	0
Z	0
résultat	320

5 * 64 = 320 Correct.


D:bkgnd
L:pop-up menu
M:type-specific command
R:deskmenu
Listening ...
[x in time_units]
/clock
/clear
/Xp
/Xm
/Yp
/Ym
/Z_control.Zp_out
/Z_control.Zm_out
0
5u
10u
15u
20u
25u

Exemple de calcul de la distance euclidienne pour les valeurs d'entrées 5 et 12

Entrée du premier nombre, X:5

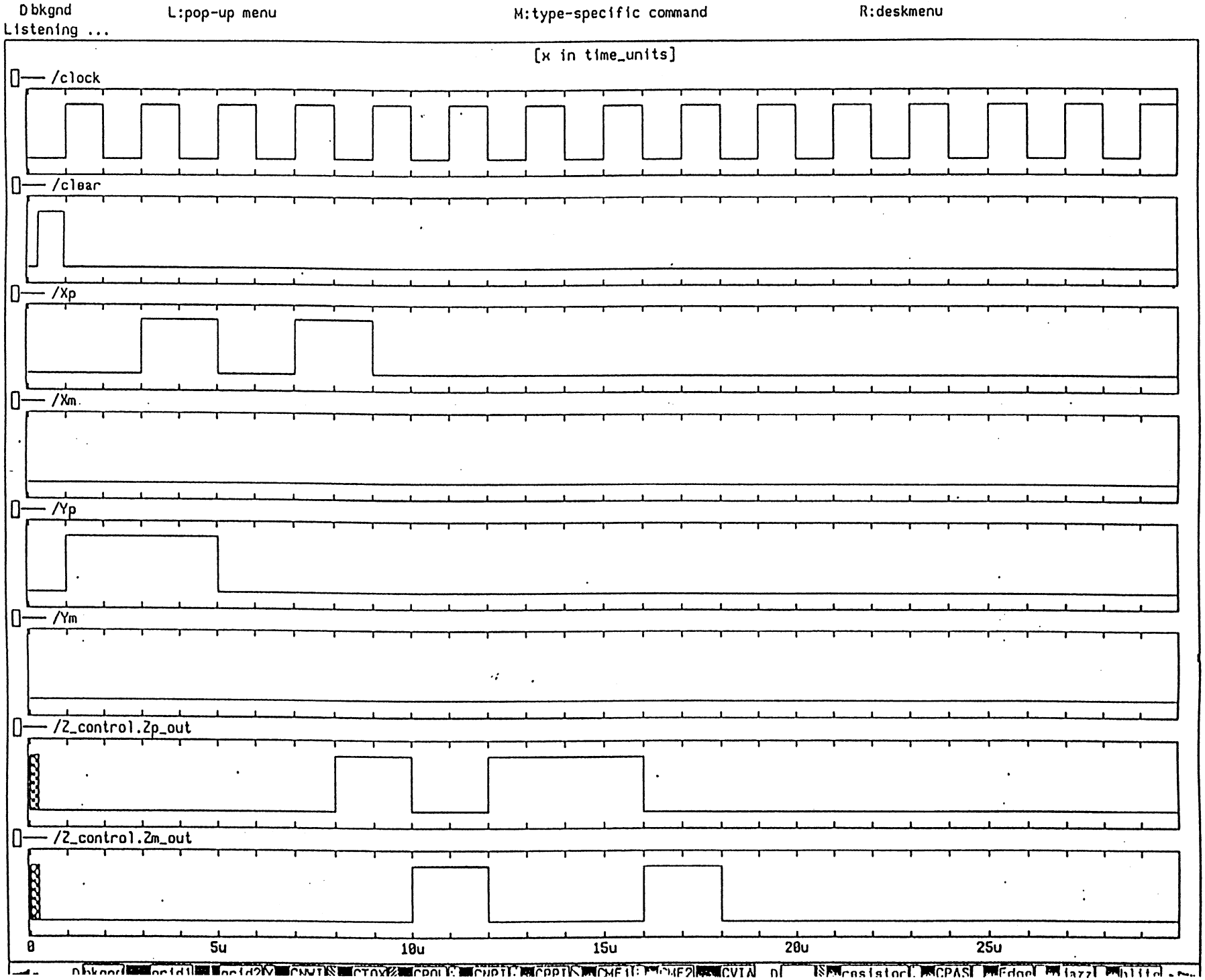
1
0
1
0
0
0
0
0
0

Entrée du second nombre, Y:12

0
0
1
1
0
0
0
0
0

Z	1
Z	-1
Z	1
Z	1
Z	-1
Z	0
Z	0
Z	0
Z	0
résultat	208

13 * 16 = 208 Correct.


Dbkgnd
L:pop-up menu
M:type-specific command
R:deskmenu
Listening ...
[x in time_units]
/clock
/clear
/Xp
/Xm
/Yp
/Ym
/Z_control.Zp_out
/Z_control.Zm_out
0
5u
10u
15u
20u
25u

Exemple de calcul de la distance euclidienne pour les valeurs d'entrées 7 et 17

Entrée du premier nombre, X:7

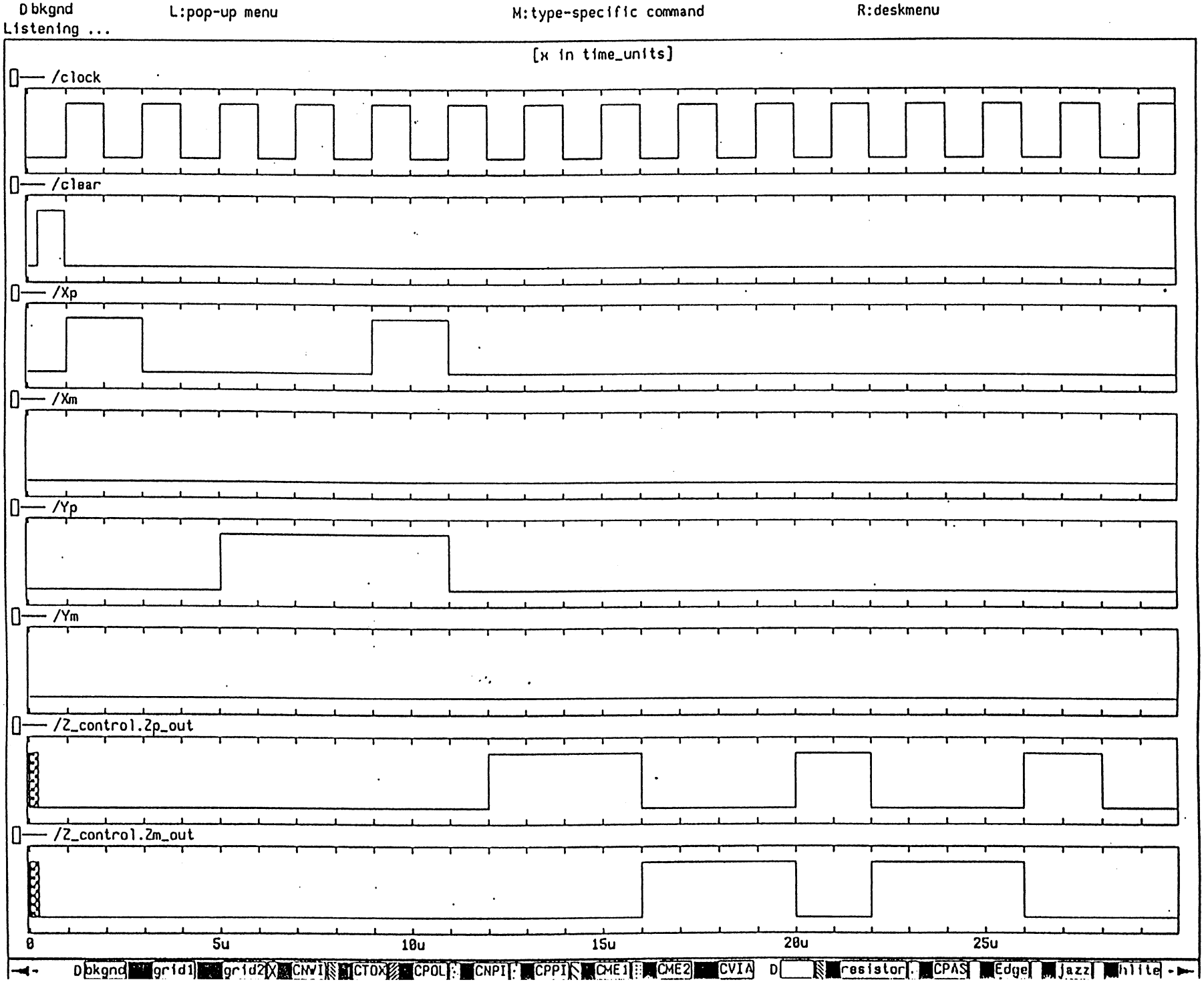
1
1
1
0
0
0
0
0
0

Entrée du second nombre, Y:17

1
0
0
0
1
0
0
0
0

Z 1
Z 1
Z -1
Z -1
Z 1
Z -1
Z -1
Z 1
Z 1
résultat 295

18.4375 * 16 = 295 Correct.


D bkgnd
L:pop-up menu
M:type-specific command
R:deskmenu
Listening ...
[x in time_units]
/clock
/clear
/Xp
/Xm
/Yp
/Ym
/Z_control.Zp_out
/Z_control.Zm_out
0
5u
10u
15u
20u
25u
D bkgnd grid1 grid2 CNWI CTOX CPOL CNPI CPPI CME1 CME2 CVIA D resistor CPAS Edge Jazz hlite

Exemple de calcul de la distance euclidienne pour les valeurs d'entrées 23 et 23

Entrée du premier nombre, X:23

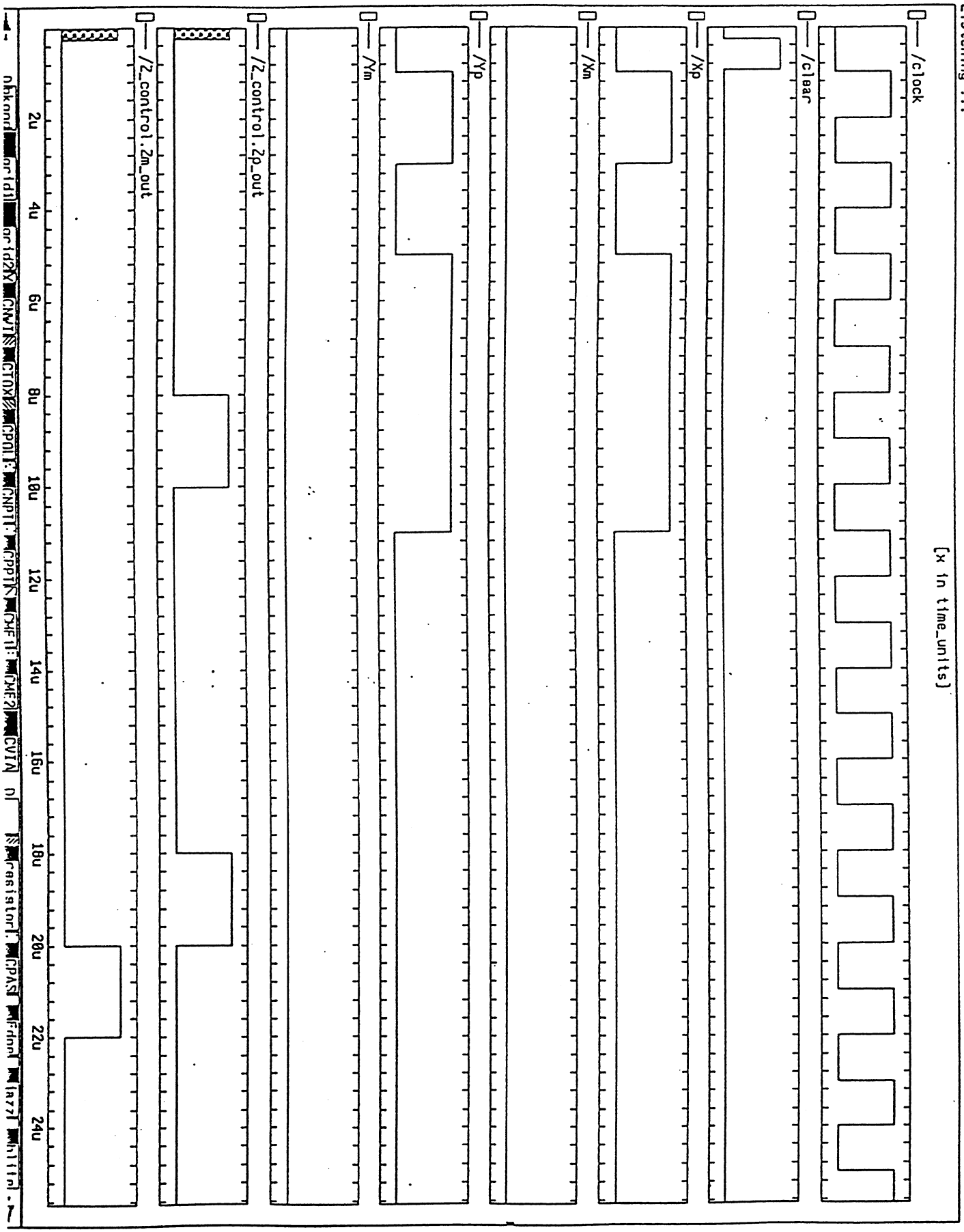
1
1
1
0
1
0
0
0
0

Entrée du second nombre, Y:23

1
1
1
0
1
0
0
0
0

Z	1
Z	0
Z	0
Z	0
Z	1
Z	-1
Z	-1
Z	0
Z	1
résultat	261

32.625 * 8 = 261 Correct.


D:bkgnd
Listening ...
L:pop-up menu
M:type-specific command
R:deskmenu
[x in time_units]
/clock
/clear
/Xp
/Xm
/Yp
/Ym
/Z_control.Zp_out
/Z_control.Zm_out
2u
4u
6u
8u
10u
12u
14u
16u
18u
20u
22u
24u

Exemple de calcul de la distance euclidienne pour les valeurs d'entrées 25 et 25

Entrée du premier nombre, X:25

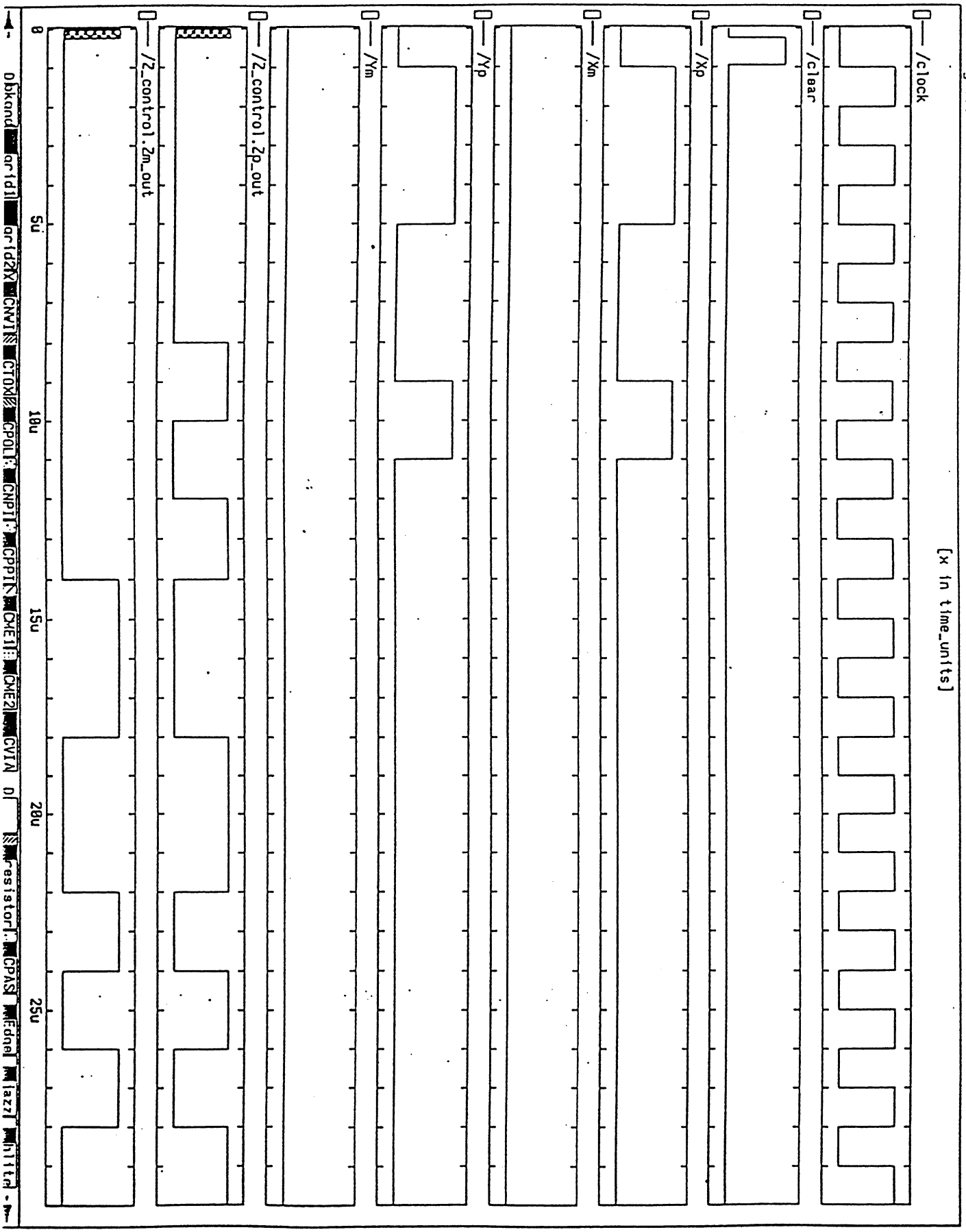
1
0
0
1
1
0
0
0
0

Entrée du second nombre, Y:25

1
0
0
1
1
0
0
0
0

Z 1
Z 0
Z 1
Z -1
Z -1
Z 1
Z 1
Z -1
Z 1
résultat 283

35.375 * 8 = 283 Correct.


Dbkgnd
Listening ...
L:pop-up menu
M:type-specific command
R:Wave full view
[x in time_units]
/clock
/clear
/Xp
/Xm
/Yp
/Ym
/Z_control.Zp_out
/Z_control.Zm_out
0
5u
10u
15u
20u
25u

Résumé

Dans le cadre de cette thèse nous avons étudié l'implantation des algorithmes de l'arithmétique "en ligne". En particulier, la réalisation de deux circuits destinés aux applications exigeant une précision infinie est exposée. En effet, dans de nombreux domaines tels que la génération de nombres aléatoires, cryptographie, calcul formel, arithmétique exacte, réduction de fraction en précision infinie, calcul modulaire, traitement d'images..., les opérateurs classiques manquent d'efficacité. Face à ce type de problèmes, un remède peut être apporté par le calcul en ligne selon lequel les calculs sont faits en introduisant les opérandes en série chiffre à chiffre en notation redondante. Nous obtenons ainsi un haut degré de parallélisme et une précision variable linéairement.

Le premier circuit présenté implante un algorithme de PGCD nommé EUCLIDE offrant, d'après les simulations, le meilleur compromis coût matériel/performance. Il donne également les coefficients de Bezout. Ce circuit est appelé à résoudre les problèmes liés au temps de calcul du PGCD par les méthodes classiques rencontrés dans beaucoup d'applications.

Une deuxième application montre la possibilité de fusionner des opérateurs en ligne afin d'obtenir un opérateur complexe. L'exemple traité dans cette thèse est celui de la distance euclidienne : $Z = \sqrt{X^2 + Y^2}$ utilisée, entre autres, pour la résolution du moindre carré des systèmes linéaires.

Abstract

In this thesis we have studied the implementation of on-line arithmetic algorithms. In particular, the design of two VLSI circuits destined for high-precision applications is shown. In fact, in numerous fields, such as pseudorandom number generation, cryptography, computer algebra, exact arithmetic, reduction of fractions with infinite precision, modular computing, image processing..., classical operators are inefficient. A solution to this type of problem can be provided through on-line arithmetic by which computations are performed by introducing operands serially digit-by-digit in a redundant notation.

The first circuit presented implements a GCD algorithm known as EUCLIDE providing, according to simulations, the best hardware cost/performance compromise. It also provides the Bezout coefficients. This circuit is to be used in order to solve problems related to the classical GCD method found in many applications.

A second application shows the possibility of cascading on-line operators in order to build up new ones. The example reported in this thesis is that of the on-line computation of the euclidean distance : $Z = \sqrt{X^2 + Y^2}$ used, among other things, in the least-square resolution of linear systems.

Mots-clés

PGCD, PGCD étendu, Notation redondante, Précision infinie, Calcul en ligne, Distance euclidienne, Conception VLSI

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 30 mars 1992 relatif aux Etudes doctorales

VU les rapports de présentation de

. Monsieur BESSALAH Hamid
. Monsieur MULLER Jean-Michel

Monsieur BOURAOUI Rachid

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Microélectronique "

Fait à Grenoble, le 6 janvier 1993

/ BV.

Pour le Président de l'INPG
et par délégation
le Directeur de l'Ecole Doctorale
J.L. LACOUME