Projet Génie Logiciel

Documentation Extension

Bull-Gamma 3

Groupe GL01

Arnaud Baumann
Aurélien Torchet
José Maillard
Laurent Milhade
Raphaël Lambert

Table des matières

1 Principe de fonctionnement du Calculateur **2**

1.1 Historique du calculateur 2

1.2 Description du calculateur 2

2 Principe des instructions Bull-Gamma **4**

3 Implémentation de la gestion du programme **9**

3.1 BullProgram 9

3.2 Les sauts conditionnel dans le programme 9

3.3 Utilisation de registres 10

3.4 Entrées / Sorties 10

4 Implémentation de macros **11**

4.1 Opérations arithmétiques décimales 11

4.2 Opérations arithmétiques binaires 11

4.3 Opérations arithmétiques flottantes 12

4.4 Manipulation des mémoires 13

4.5 Manipulation des valeurs de mémoires 13

4.6 Comparaisons 13

4.7 Autres macros 14

5 Calcul binaire **14**

5.1 Calcul Flottant 14

5.2 Calcul Entier 17

5.3 Conversions 18

6 Tests Réalisés **18**

7 Bibliographie **19**

8 Remerciement **19**

1 Principe de fonctionnement du Calculateur

1.1 Historique du calculateur

Le calculateur Bull-Gamma 3 est un calculateur des années 50. Il a été créé par la compagnie Bull et succède au Bull-Gamma 2 construit en 1951. Ce calculateur fonctionne à l'aide de diodes et de lignes à retard, c'est un véritable ordinosaure.

Le calculateur sur laquelle nous avons travaillé est en fait une extension du Gamma 3 : le Gamma E.T. (Extension Tambour) qui lui permettait d'enregistrer des programmes. Cette version a été installé dans les locaux d'un département scientifique Grenoblois dirigé par Jean Kuntzmann en 1957, à la demande de Louis Bolliet. Ce fût le deuxième ordinateur grenoblois.

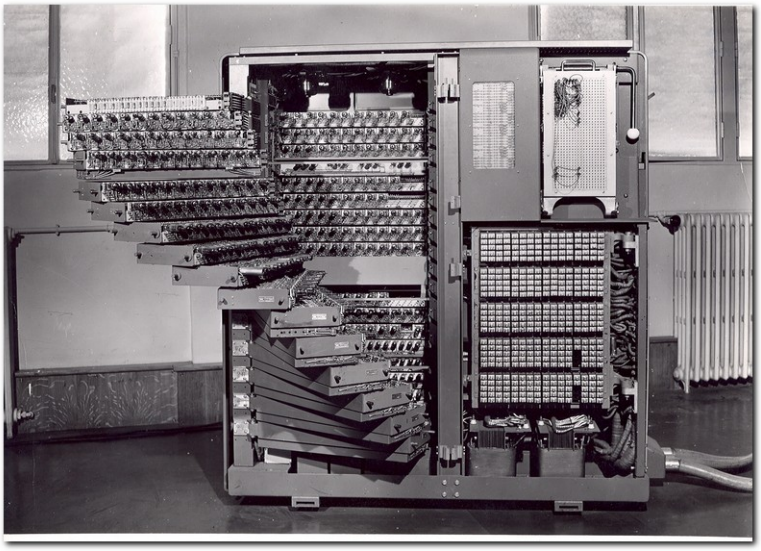

FIGURE 1 – Calculateur Bull-Gamma

1.2 Description du calculateur

Le calculateur est composé de différents éléments :

- Des mémoires utilisées comme des registres.
- Un ensemble de mécanismes qui reçoit les instructions et les exécutes (processeur).
- Diverses mémoires reservées comme les registres de comparaison utilisés en IMA.
- Une mémoire de stockage, le tambour. C'est l'équivalent d'un disque dur.

Les Mémoires Le Bull-Gamma 3 possèdes des mémoires sur 48 bits divisés en 12 mots hexadécimaux. Ces mémoire sont partagées en trois types :

— **Les mémoires banales**. Elles sont au nombre de 7 (sans compter M0 qui est reliée à M1). Deux d'entre elles, M1 et M2 servent à effectuer les opérations. M1 comporte un additionneur-soustracteur séquentiel et M2 peut être ajoutée à M1 pour effectuer des multiplications et division sur 96 bits (deux mémoires). Les autres mémoires M3 à M7 sont des mémoires normales.

— **Deux types de mémoires banales commutées**. Ces mémoires sont numérotées de 8 à 15 et sont au nombre de 8. Elles sont "circulantes", c'est à dire qu'on peut changer d'octade. Elles permettent d'avoir 8 octades et il faut effectuer une instruction pour accéder à l'une d'entre elles en particulier. Ces 8 octades sont regroupées par deux pour former quatre groupes. Les trois premiers sont exécutables, on utilisera le terme de série lorsqu'on parlera d'un accès en exécution. Le dernier correspond aux entrées et aux sorties.

Les instructions Les instructions sont codées sur 4 mots hexadécimaux. Ces 4 mots sont

— le Type d'Opération (TO)

— l'Adresse (AD)

— l'Ordre de Début (OD)

— l'Ordre de Fin (OF)

Mémoires réservées Elles permettent d'effectuer des opérations en fonction de leur état.

— MS1 : mémoire de signe

— MD : mémoire de décalage

— MC : mémoire de comparaison

— RNL1|2 : mémoires de saut

— NL : mémoire de numéro de ligne, correspond à PC

Le tambour Il est constitué de huit seizaine de 16 pistes chacunes, elles-même constituées de 8 blocs. La taille de ces blocs correspond à un groupe de mémoires banales commutées. En chargeant un bloc dans un groupe on peut donc charger une page de 48 instructions (2 octades avec des mémoires de 3 fois 4 mots hexadécimaux)

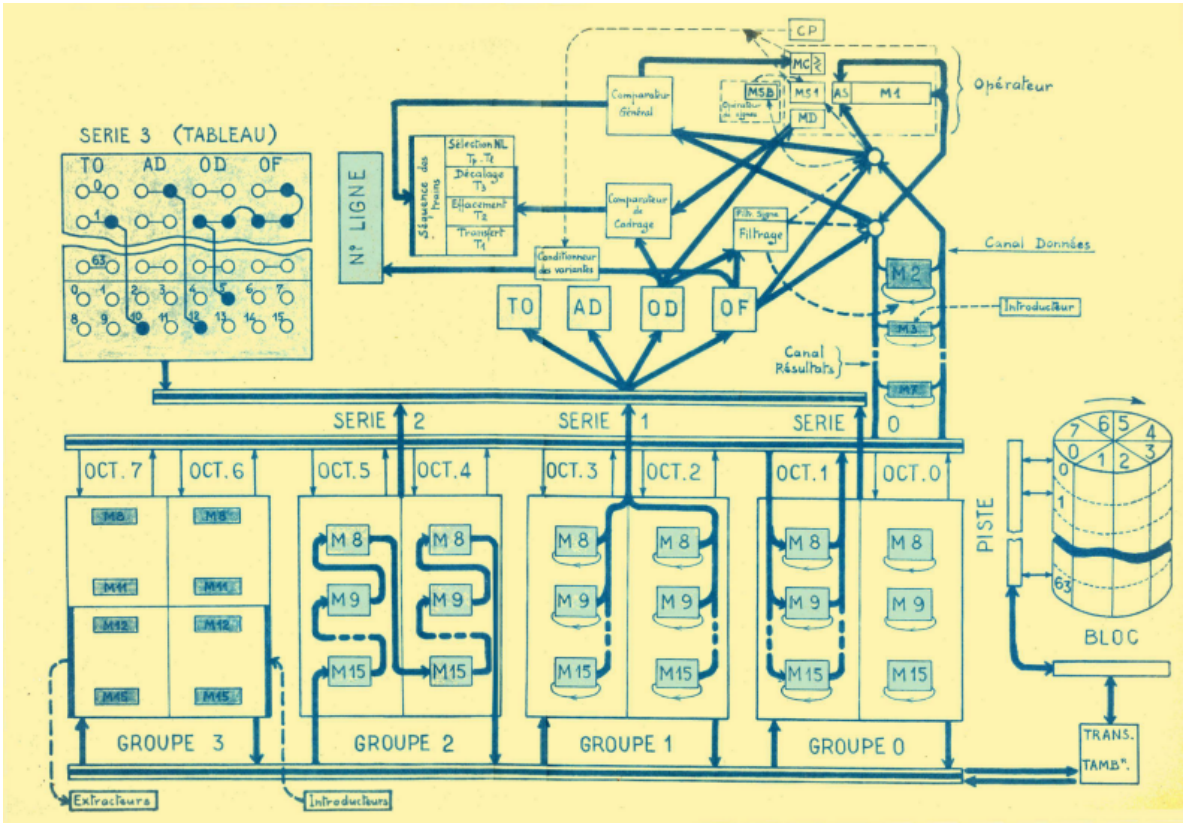

FIGURE 2 – Structure du calculateur

2 Principe des instructions Bull-Gamma

Les instructions Bull-Gamma sont donc composées de 4 nombres hexadécimaux : TO, AD, OD, OF. Dans ces instructions, la seul constante est que TO définie le type de l'opération. AD, OD et OF précisent ensuite cette opération. Par exemple, lorsqu'il s'agit d'instructions de type calcul, AD représente, la plupart du temps, le numéro de la mémoire que l'on veut utiliser, et OD et OF peuvent correspondre respectivement à l'indice de début et de fin de la partie que l'on va utilisée dans cette mémoire.

Il a été choisi de n'implémenter en java que les instructions qui nous serviraient pour atteindre notre objectif primaire d'extension. Ces instructions sont implémentées par des sous-classes d'une classe abstraite `BullInstruction`. Un peu plus d'informations sur cette classe seront données par la suite. Pour le moment, savoir qu'elle contient quatre attributs correspondants à TO, AD, OD et OF est suffisant. Il y a en tout 16 types d'instructions. Voici ces types, ainsi que nos implémentations d'instructions qui correspondent :

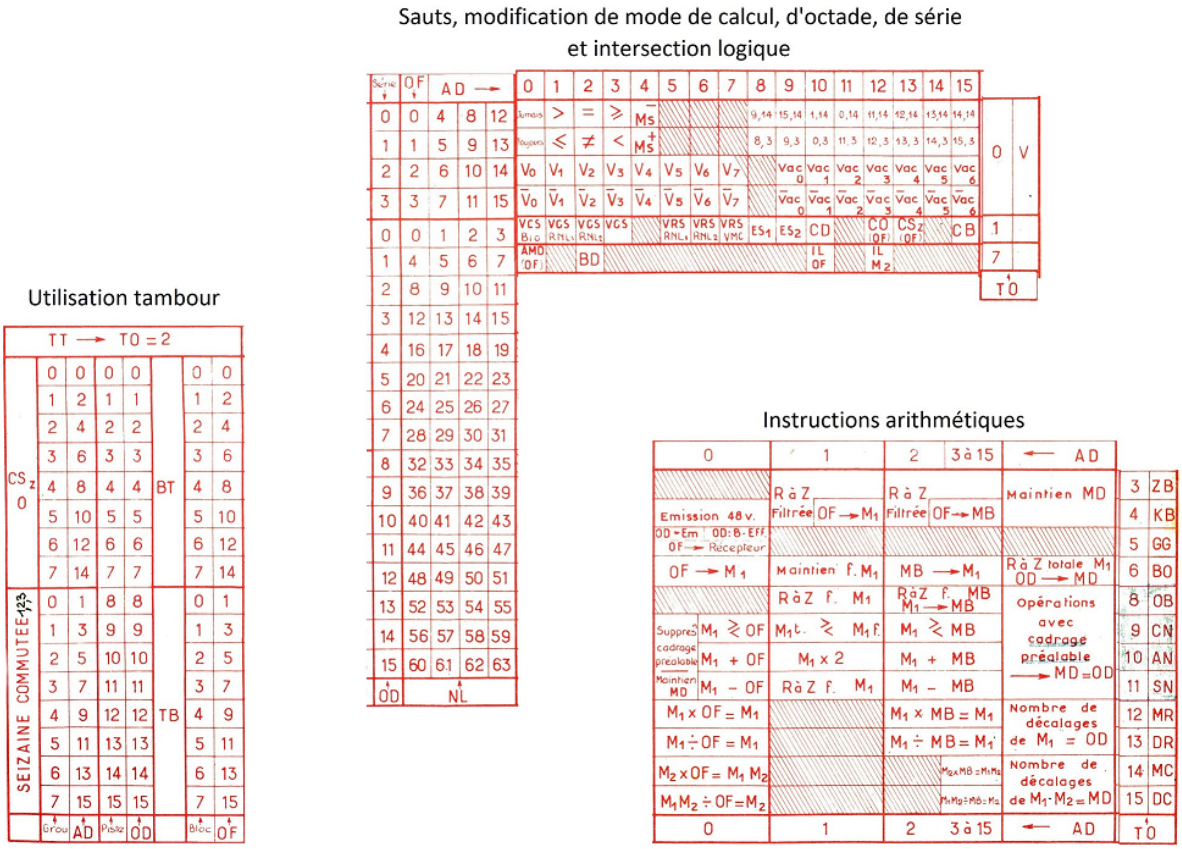

FIGURE 3 – Tableau des instructions

Opération 0

Saut conditionnel à un numéro de ligne

— `ConditionalBranch(int type, int nLine)` : *type* correspond au numéro de la condition à vérifier pour sauter, *nLine* correspond au numéro de ligne voulu → calcul de AD, OD et OF en fonction ces paramètres.

— `Vide()` : instruction vide.

Opération 1

Permet selon la valeur de AD de passer en calcul décimal, en calcul binaire, de sauter dans un groupe spécifique à une ligne donnée (VCS), de changer d'octade ou de seizaine utilisée pour les instructions qui utilise des mémoires banales.

— `JumpSpecificLocation()` : Saut direct à la ligne 0 de la série 0 (Seul accès à la série 0)

— `ChangeOctad(int num)` : Permet de changer l'octade utilisée lors d'interaction avec les octades commutées (CO)

— `ChangeSeizaine(int num)` : Permet de changer la seizaine utilisée lors d'une interaction avec le (CS)

— `BinaryMode()` : Passe le calculateur en mode calcul binaire (CB)

— `DecimalMode()` : Passe le calculateur en mode calcul décimal (CD)

Opération 2

Permet de charger un groupe dans un bloc du tambour et inversement.

— `LoadPage(int numBloc, int numPiste)` : Charge dans le groupe 1 la page situé en *numBloc*, *numPiste* de la seizaine actuellement utilisée en accès.

Opération 3

Effectue une mise à zéro filtrée (entre OD et OF) d'une mémoire.

— `Clear(int MB,int OD, int OF)` : remet à 0 la valeur du registre MB entre OD et OF.

— `ClearM1()` : met à zéro le registre M1.

Opération 4

Permet de placer une valeur constante dans une mémoire.

— `SetConst(int AD, int OD, int OF)` : Ajoute la constante OF dans le registre AD avec un décalage OD.

— `SetConstM1(int shift, int constant)` : ajoute la constante comprise entre 0 et 15 au décalage (compris entre 0 et 11) souhaité dans la mémoire M1.

— `SetConstMB(int numMemory, int shift, int constant)` : ajoute la constante comprise entre 0 et 15 au décalage (compris entre 0 et 11) souhaité dans la mémoire numMemory.

Opération 6

Place la valeur d'une mémoire spécifiée dans la mémoire M1, ou effectue une remise à zéro filtrée de M1.

— `Store(int register, int OD, int OF)` : stocke les mots compris entre OD et OF de la mémoire M1 dans la mémoire register.

Opération 7

Permet d'effectuer des intersections logiques entre deux nombres.

— `interConst()` : Effectue une intersection logique entre la mémoire M1 et une constante comprise entre 0 et 15. Le registre M1 contenant 12 mots de 4 bits (valeurs entre 0 et 15), on effecture l'intersection avec chaque mot de M1.

— `InterM2()` : effectue une instersaction logique de la mémoire M2 avec la mémoire M1

Opération 8

Place la valeur de M1 dans une mémoire spécifiée.

— `Load(int register, int OD, int OF)` : charge les mots compris entre OD et OF (OD<OF) de la mémoire register (comprise entre 2 et 7) dans la mémoire M1.

Opération 9

Effectue une comparaison entre un registre et M1.

— `Comparison(int AD, int OD, int OF)` : Compare le contenu entre OD et OF du registre AD au registre M1.

Opération 10

Effectue une addition

— `Add( int AD, int OD, int OF)` : Ajoute le contenu situé entre OD et OF de la mémoire AD à la mémoire M1.

— `AddMemTOM1(int register)` : Ajoute le contenu de la mémoire register à la mémoire M1.

— `AddConstM1(int shift, int constant)` : Ajoute un constante comprise entre 0 et 15 avec un décalage égal à shift.

— `ShiftLeft()` : multiplier par deux le contenu de M1.

Opération 11

Effectue une soustraction

— `Sub( int AD, int OD, int OF)` : Soustrait le contenu situé entre OD et OF de la mémoire AD à la mémoire M1.

— `SubConstM1(int decalage, int constant)` : Soustrait la constante constant avec un décalage décalage à la mémoire M1.

— `SubMemToM1(int register)` : soustrait le contenu de la mémoire M1 par la mémoire register.

Opération 12

Effectue une multiplication réduite

— `MultMB(int register)` : Multiplie la mémoire M1 par la mémoire MB.

Opération 13

Effectue une division réduite

— `DivMB(int register)` : divise le contenu de la mémoire M1 par la mémoire register.

Opération 14

Effectue une multiplication complète

— `CompleteMult(int register)` : Effectue une multiplication complete de M1M2 par le registre register.

Opération 15

Effectue une division complète

— `CompleteDiv(int register)` : Effectue une division complete de M1M2 par le registre register.

3 Implémentation de la gestion du programme

3.1 BullProgram

Pour obtenir en sortie une suite d'instruction utilisable par l'émulateur du Bull Gamma il nous fallait un gestionnaire d'instruction ou elles seraient stockées les une après les autres et avec lequel on pourrait les écrire comme il faut. Nous avons donc choisi d'utiliser un objet proche de celui que l'on avait utilisé pour la compilation en IMA, `IMAProgram`. `BullProgram` s'en inspire donc grandement.

La gestion de l'écriture d'une instruction en hexadécimal, comme demandé par l'émulateur, étant géré par l'instruction elle même à travers une méthode de `BullInstruction`, l'intêret de la nouvelle classe n'est pas là. En effet, elle gère les sauts à la page suivante du programme en utilisant les instructions `ChangeSeizaine(...)` et `LoadPage(...)` quand il le faut et où il faut. Par soucis d'optimisation on charge la nouvelle page dans la série actuelle et on doit donc être à la dernière instruction possible pour bien recommencer à la première ligne de la nouvelle page. De plus, `BullProgram` aide à la gestion des sauts demandés par le programme.

3.2 Les sauts conditionnel dans le programme

Un saut du programme vers une autre page du programme côute cher et est compliqué. Il faut obligatoirement charger une page depuis le tambour (opération la plus longue) mais il faut aussi passer sur une autre série car sinon le chargement écrase la page et le numéro de ligne a de grandes chances d'être mauvais. On se retrouve donc a devoir aller sur une autre série pour sauter à un endroit.

Il nous faut une "adresse" de saut. Les sauts sur un label n'existant pas, il va falloir calculer cette adresse. De plus, on ne peut pas stocker toutes les adresses de saut à un endroit. On a donc décidé d'utiliser le bloc 0 pour stocker les 3 instructions nécessaires aux sauts inter-pages. Ces instructions devront être modifiées depuis le programme puis le programme devra y sauter.

Il ne faut donc sous aucun pretexte écrire sur ces instructions pré-faite d'une mauvaise manière. Ainsi, on décide de **réserver l'octade 0 a des valeurs utilisées de façon externe, le groupe 1 pour être la série par défaut (utilisée pour exécuter les pages du programme) et l'octade 1 et le groupe 2 pour servir de pseudo pile**.

Les valeurs des modifications nécessaire doivent être calculées depuis java avant d'inserer les instructions nécessaire à ces modifications. Le saut conditionnel devient bien plus qu'une instruction. Il se transforme en 8 instructions à générer *aprs* une passe de l'arbre provenant du code deca. On distingue, de plus, les sauts inter-page des sauts dans la même page (simple changement de `NL`) et il faut donc connaître l'adresse de départ et d'arrivée d'un saut.

La classe `Labellisable` dont `BullInstruction` hérite, permet cela mais, il est au

final plus pratique d'utiliser le `BullProgram` avec des labels non liées aux instructions. (Si l'extension est poursuivie, cette classe mère disparaîtra) Une fois que l'on a toutes les informations nécessaires, on modifie les 8 instructions reservées dans le `BullProgram` grâce à l'ajout, au préalable, d'un objet `ReservedJump`

```
ChangeOctad(0));
SetConstMB(8, 0, nSeiz));
SetConstMB(8, 5, nPiste));
SetConstMB(8, 4, 2*nBloc + 1));
SetConstMB(8, 8, (nLine%4)*4 + 1));
SetConstMB(8, 9, nLine/4));
ConditionalBranch(invType, reservedLines[7].getLine()+1));
JumpSpecificLocation());
```

FIGURE 4 – Exemple d'instruction de saut inter-page

3.3 Utilisation de registres

Les memoires des octades 1, 4 et 5 sont utilisées comme registres de calcul et pile. La classe `Registers` fournit des outils pour faciliter l'accès à ces registres. Elle permet actuellement, pour un numéro de registre de 0 à 23, d'en générer l'octade et le numéro de mémoire. Il faut donc changer d'octade explicitement dans les macros. A terme elle pourra contenir une methode retournant une suite d'instructions permettant de préparer les mémoires, permettant ainsi une gestion de pile sur le tambour transparente.

3.4 Entrées / Sorties

Les octades 6 et 7 sont utilisées pour les entrées sorties. L'émulateur n'implémentant pas les entrées sorties, il a été décidé de ne pas gérer les entrées (ReadInt ou ReadFloat). En revanche, les sorties sont écrites dans l'octade 7 comme suit :

— La sortie d'erreur réserve la mémoire M8

— Les 7 autres mémoires sont utilisées comme sortie standard. Il est possible d'y écrire des entiers et des flottant stockés dans les registres. Une fois la dernière mémoire utilisée, on réécrit dans la première.

La classe `Registers` généralise également l'accès à ces mémoires

4 Implémentation de macros

Afin de simplifier les opérations et d'optimiser la réutilisation du code, des macros ont été créees. Ces macros sont des listes d'instructions. Toutes les macro possèdent la super Classe `AbstractMacro`, qui possède un attribut `ArrayList<BullInstruction>`. C'est dans cette liste que sont ajoutées les instructions. Dans le programme principal, on pourra alors selon les besoins, ajouter des macros, ou ajouter des instructions simples.

Ces macros manipulent pour la grande majorité les mémoires des octades (mémoire banale commutée), et non les mémoire banales, qui sont reservées à stocker des résultats intermédiaires de calculs. Il est spécifié alors que **les macros n'utilisent pas les mémoire banales commutées**.

4.1 Opérations arithmétiques décimales

Ces opérations se font sur des entiers en représentation décimale. Ces macros comportent peu d'instructions car le calculateur gère bien le calcul décimal.

- `MacroAdd(int reg1, int reg2, int destReg)` : Additionne deux mémoire reg1 et reg2 entre elles et stocke le résultat dans la mémoire destreg.
- `MacroSub(int reg1, int reg2, int destReg)` :Soustrait la mémoire reg1 par la mémoire reg2 et stocke le résultat dans la mémoire destreg.
- `MacroMult(int nb, int reg1, int reg2, int destReg)` : Multiplie deux mémoire reg1 et reg2 entre elles et stocke le résultat dans la mémoire destreg.
- `MacroMultC(int reg1, int reg2, int destReg)` : Multiplie deux mémoire à laide d'une mutiplication complete reg1 et reg2 entre elles et stocke le résultat dans la mémoire destreg.
- `MacroDiv(int nb, int reg1, int reg2, int destReg)` : Division euclidienne de la mémoire reg1 par la mémoire reg2 et stocke le résultat dans la mémoire destreg.
- `MacroDivC(int reg1, int reg2, int destReg)` : Divise deux mémoire à laide d'une division complete reg1 et reg2 entre elles et stocke le résultat dans la mémoire destreg.

4.2 Opérations arithmétiques binaires

Ces opérations se font sur des entiers en représentation binaires. Ces macros comportent beaucoup d'instruction car le Bull Gamma ne gère pas automatiquement le calcul binaire.

— `AbstractBinaryIntAddition(int regLeft, int regRight, int regDest)` : Effectue une addition ou soustraction binaire de deux entiers entre les deux registre regLeft et regRight et stocke le résultat dans regDest. La Classe est abstraite et est implémentée dans FloatAddition et FloatSubtraction.

— `BinaryIntAddition(int regLeft, int regRight, int regDest)` : Effectue une addition entière en repésentation binaire entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `BinaryIntSubtraction(int regLeft, int regRight, int regDest)` : Effectue une soustraction entière en repésentation binaire entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `BinaryIntMultiplication (int regLeft, int regRight, int regDest)` : Effectue une multiplication entière en repésentation binaire entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `BinaryIntDivision (int regLeft, int regRight, int regDest)` : Effectue une division entière en repésentation binaire du registre regLeft par le registre regRight et stocke le résultat dans regDest.

4.3 Opérations arithmétiques flottantes

— `AbstractFloatAddition(int regLeft, int regRight, int regDest)` : Effectue une addition ou soustraction flottante entre les deux registre regLeft et regRight et stocke le résultat dans regDest. La Classe est abstraite et est implémentée dans FloatAddition et FloatSubtraction.

— `FloatAddition(int regLeft, int regRight, int regDest)` : Effectue une addition flottante entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `FloatSubstraction(int regLeft, int regRight, int regDest)` :Effectue une soustraction flottante entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `FloatMultiplication (int regLeft, int regRight, int regDest)` :Effectue une mutltiplication flottante entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `FloatDivision (int regLeft, int regRight, int regDest)` :Effectue une division flottante entre les deux registre regLeft et regRight et stocke le résultat dans regDest.

— `FloatUnaryMinus (int register, int destReg)` : Change le signe du nombre dans le registre register et stocke le resultat dans le registre destReg

4.4 Manipulation des mémoires

— `MacroLoad(int register, int octade,int OD, int OF )` : charge les mots compris entre OD et OF (OD<OF) de la mémoire register (comprise entre 2 et 15), de l'octade octade dans la mémoire M1.

— `MacroStore(int register, int OD, int OF)` : stocke les mots compris entre OD et OF (OD<OF) de la mémoire M1 (comprise entre 2 et 15), de l'octade octade dans la mémoire register.

— `MoveReg(int reg1, int reg2)` : déplace la valeur de la mémoire reg1 dans la mémoire reg2. Ces registres sont ceux des octades (mémoire banale commutée).

— `Move(int reg1, int reg2, int M1Dest)` : déplace la valeur de la mémoire reg1 dans la mémoire reg2. On déplace préalablement la valeur de M1 dans le registre M1Dest. Ces registres ne sont pas ceux des octades (mémoire banale).

— `Move(int reg1, int reg2)` : déplace la valeur de la mémoire reg1 dans la mémoire reg2.Ces registres ne sont pas ceux des octades (mémoire banale).

4.5 Manipulation des valeurs de mémoires

— `MacroSetM1(int number)` : met un entier number en représentation décimale dans le registre M1. Passe automatiquement en calcul décimal.

— `MacroSetReg(int register, int number)` : met un entier number en représentation décimale dans le registre register. Passe automatiquement en calcul décimal.

— `MacroSetBinaryIntM1(int number)` : met un entier number en représentation binaire dans le registre M1. Passe automatiquement en calcul binaire.

— `MacroSetBinaryReg(int number, int register)` : met un entier number en représentation binaire dans le registre register. Passe automatiquement en calcul binaire.

— `MacroSetFloatM1(float number)` :met un flottant number (représentation binaire) dans le registre M1. Passe automatiquement en calcul binaire.

— `MacroSetReg(int register, int number)` : met flottant number (représentation binaire) dans le registre register. Passe automatiquement en calcul Binaire.

4.6 Comparaisons

— `MacroComp(int reg1, int reg2)` : Compare les nombres entiers en représentation décimale contenus dans deux registres. Le registre modifié est le registre spécial de comparaison.

— `BinaryIntComparison(int regLeft, int regRight)` : Compare les nombres entiers en représentation binaire contenus dans deux registres. Le registre modifié est le registre spécial de comparaison.

— `FloatComparison(int regLeft, int regRight)` : Compare les nombres flottans en représentation binaire contenus dans deux registres. Le registre modifié est le registre spécial de comparaison.

4.7 Autres macros

— `Print (int out, int register)` : copie le registre register dans un registre de sortie pour un print.

— `ShiftRight(int dest)` : décale la valeur de M1 vers la droite. Les trois bits les plus à gauche devraient être à zéro pour ne pas les perdre.

— `WriteError()` : incrémente le registre d'erreur.

5 Calcul binaire

5.1 Calcul Flottant

Les nombres flottants sont représentés sur 48 bits. Le bit de poids fort est utilisé pour le signe, 8 bits sont utilisés pour l'exposant, et les 39 bits de poids fort sont réservés à la mantisse.
Cette représentation est proche de la norme IEEE 754 simple précision (32 bits), seulement la mantisse diffère et permet une plus grande précision.

Seuls les nombres normalisés sont considérés. Zéro est représenté par une mantisse égale à zéro. Afin de faciliter les algorithmes sous Bull Gamma, il a été choisi de ne pas omettre le 1 le plus significatif de la mantisse.

Quelques examples de représentations flottantes

0 : $\phi|\phi\phi\phi\ \phi\phi\phi\phi\ \phi|$000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1 : 0|011 1111 1|100 0000 0000 0000 0000 0000 0000 0000 0000 0000
-5.5 : 1|100 0000 1|101 1000 0000 0000 0000 0000 0000 0000 0000 0000

Operations flottantes

Comme présenté en section 4, les opérations flottantes suivantes ont été implémentées :

— `FloatMultiplication`

— `FloatDivision`

— `FloatAddition`

— `FloatSubtraction`

— `FloatComparison`

— `FloatUnaryMinus`

Le principe général de ces algorithmes consiste à séparer le sign, l'exposant et la mantisse pour effectuer les calculs nécessaires sur chacun d'eux, puis de recomposer le résultat.

Détail de la multiplication flottante

L'algoritme de la multiplication est détaillé ci-dessous. Il prend en entrée deux nombres flottants normalisés et retourne un produit également normalisé.

Sur Bull Gamma, la multiplication s'effectue sur deux memoires de 48 bits, la multiplication des mantisses ne génère donc pas d'erreur.

Le produit des mantisses est tronqué au $38^{\text{ème}}$ bit après le premier bit à un et devient la mantisse du résultat. Avec ce raisonnement, et en sachant la virgule des mantisses à multiplier à droite du premier bit significatif, il faut au plus corriger de 1 l'exposant résultat obtenu par la somme des exposants.
Si l'exposant dépasse sa valeur maximal, un erreur est signalée.

Le signe est simplement le produit des signes.

Précision sur la soustration flottante

Afin de factoriser le code, une soustration est obtenue en effectuant une addition avec l'opérande de gauche et le résultat du moins unaire appliqué à l'opérande de droite.

Liste des erreurs gérées

Une erreur incrémente de 1 la valeur du registre dédié aux erreurs. Les erreurs suivantes sont signalées lors de l'execution d'une opération flottante :

— Le résultat est plus grand que le flottant maximum représentable, en valeur absolue

— Le résultat est plus petit que le flottant minimum représentable, en valeur absolue

— La division par zéro

Une amélioration possible pourrait consister à attribuer certain bits du registre d'erreur à chaque erreur afin de savoir laquelle a été levée.

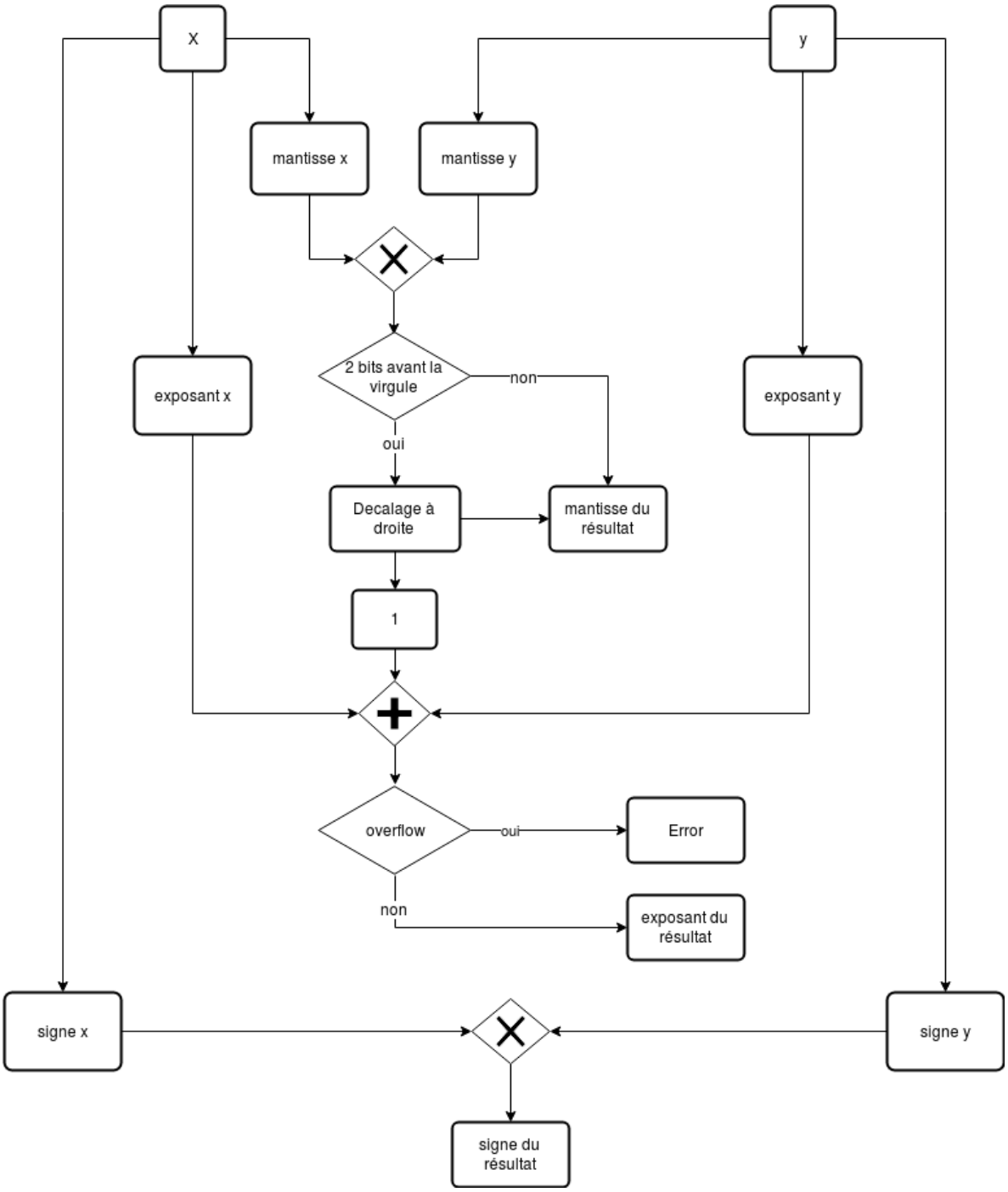

FIGURE 5 – Algorithme de multiplication flottante

5.2 Calcul Entier

Les nombres entiers sont représentés sur 48 bits. Le bit de poids fort est utilisé pour le signe, les 47 autres bits sont utilisés par la valeur absolue du nombre.
Il a été choisi de ne pas utiliser le complément à 2 afin de faciliter les conversions entre entiers et flottant et de permettre une gestion du dépassement lors de l'addition.

Quelques examples de représentations flottantes

0 : ϕ|000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1 : 0|000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
-42 : 1|000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 1010

Operations flottantes

Comme présenté en section 4, les opérations entières suivantes ont été implémentées :

- `BinaryIntMultiplication`
- `BinaryIntDivision`
- `BinaryIntAddition`
- `BinaryIntSubtraction`
- `BinaryIntComparison`
- `BinaryIntUnaryMinus`
- `BinaryIntModulo`

Le principe général de ces algorithmes consiste à séparer le signe et la valeur absolue pour effectuer les calculs nécessaires sur chacun d'eux, puis de recomposer le résultat.

Liste des erreurs gérées

Une erreur incrémente de 1 la valeur du registre dédié aux erreurs. Les erreurs suivantes sont signalées lors de l'execution d'une opération entière :

- Le résultat est plus grand que le flottant maximum représentable, en valeur absolue
- La division par zéro

5.3 Conversions

Conversion d'entier à flottant

La convrsion d'un entier à un flottant est presque immédiate en instructions Bull Gamma grâce aux représentations choisies.
Le signe est inchangé et il suffit décaler la valeur absolue jusqu'à qu'elle corrresponde à une mantisse normalisée.
L'exposant vaut 46 moins le nombre de décalages.

Conversion de float java en flottant

Comme mentionné précédement, dans la représentation java, le premier '1' significatif est omis. Il faut donc l'insérer en $9^{\text{ème}}$ position de la suite de bits de la représentation. Les float java sont codés sur 32 bits, on complète donc ensuite par 15 '0' pour obtenir le flottant sur 48 bits

Conversion d'int java en entier

Les int java étant représentés en complément à 2, on en prend le signe et la valeur absolue et on les concatène pour obtenir une représentation entière pour le Bull Gamma. Les int java sont codés sur 32 bits, il n'y a donc pas de problème de dépassement.

6 Tests Réalisés

Il s'agissait d'abord de prendre en main le calculateur. C'est pourquoi les premiers tests se firent directement sur le simulateur, en ajoutant des instructions à la main. La prise en main a permit de mieux comprendre le jeu d'instructions du calculateur après l'analyse des documents récupérés.

Chaque instruction utile a été implémentée en java. Des tests java ont ensuite été faits sur les différentes macro. Les tests génèrent des instructions pour Bull Gamma dans un fichier TAMBOUR.IMG, qui est a placer dans le dossier où se trouve l'émulateur avant de le lancer. Enfin, lorsque la génération de code fut implémentée, des tests de programmes deca ont été réalisés et des options ont été rajoutés pour faciliter l'utilisation de l'émulateur avec decac. Ces options sont :

— -e pour passer en compilateur pour Bull Gamma E.T.

— -c <path> pour écrire le fichier TAMBOUR.IMG directement dans le dossier de l'émulateur et juste avoir à le lancer.

7 Bibliographie

Tous les documents utilisés sont présents à l'adresse `http://www.aconit.org/histoire/Gamma-3/Articles/`.

Les documents que nous avons le plus utilisé pour notre implémentation, sont :

— la deuxième partie du cours de programmation de 1959 de Louis Bolliet.

— le résumé des instructions sous forme de tableau.

8 Remerciement

Merci à nos encadrants de projet de Génie Logiciel pour leur encadrement et leurs conseils.

Merci à l'association Aconit pour tous les documents apportés et particulièrement à Alain Guyot qui nous a permis de démarrer cette extension.

Merci à Vincent Joguin pour son émulateur-débogueur de Bull Gamma E.T. Après 18 ans, il a réussi à se replonger dans son code et à corriger une un bogue. Nous lui en sommes très reconnaissants.