Un micro-processeur 32 bits

Sam Zoghaib et Thierry Martinez

28 janvier 2009

Table des matières

1 Architecture

1.1 Représentation des valeurs

La longueur du mot est fixée à 32 bits. Les valeurs sont représentées avec la convention petit-boutin. Dans le cas des entiers relatifs, un entier négatif est

obtenu par complémentation à deux de sa valeur absolue.

1.2 Registres

Les registres sont tous sur un mot.

$r_0, \ldots, r_{15}$	00 ... 0F	Registres généraux
sp	10	Adresse du sommet de pile
i_0, i_1, i_2	11 ... 13	Registres d'entrées
o_0, o_1, o_2	14 ... 16	Registres de sorties
pc	17	Adresse de la prochaine instruction exécutée

1.3 Mémoires

Les adresses sont sur un mot et correspondent à une abscisse exprimée en nombre de mots sur une demi-droite positive partagée de la manière suivante :

- Les adresses de 0x0000 à 0x0FFF sont réservées à la ROM.
- Les adresses supérieures (à partir de 0x1000) sont disponibles pour la RAM.

Au niveau du code compilé, les lectures en RAM et en ROM s'effectuent de la même manière.

Le drapeau f est levé lorsqu'une opération non conforme est effectuée, en particulier lors d'une tentative de lecture ou d'écriture en dehors du segment adressable, ou encore lors d'une tentative d'écriture en ROM.

Le mot à l'adresse 0x0FFF (plus haute adresse en ROM) contient $\max_{\mathsf{RAM}}$, la plus haute adresse accessible (en RAM). $1 + \max_{\mathsf{RAM}}$ est chargée dans sp avant l'exécution de la première instruction.

La ROM et la RAM ont un seul port de lecture, et la RAM a un seul port d'écriture ; au cours d'un cycle, la lecture précède l'écriture.

1.4 Instructions

Toute instruction peut être codée selon deux formats :

Format court (un mot)

1 bit	5 bits	2 bits
Bit de format : 0	Code d'opération	Mode d'adressage

5 bits	3 bits	16 bits
Registre	Mode d'opération	Opérande

Format long (deux mots)

1 bit	5 bits	2 bits
Bit de format : 1	Code d'opération	Mode d'adressage

5 bits	3 bits	1 bit	15 bits	32 bits
Registre	Mode d'opération	Mode de décalage	Décalage	Opérande

Les deux bits de Mode d'adressage donnent le sens de Opérande :

- Si le bit de poids fort est à 1, le *contenu de l'opérande* est le contenu du registre désigné par les 5 bits de poids faible de Opérande. Sinon, le *contenu de l'opérande* est Opérande elle-même.

– Si le bit de poids faible est à 1, la *valeur de l'opérande* est la valeur stockée à l'adresse pointée par le *contenu de l'opérande*. Sinon, la *valeur de l'opérande* est le *contenu de l'opérande* lui-même.

Si Mode d'adressage est égal à 0b01, c'est-à-dire si Opérande est une adresse, celle-ci est relative si l'instruction est codée au format court, absolue si l'instruction est codée au format long.

Lorsque l'instruction est codée au format long, si le bit de poids faible de Mode d'adressage est à 1, alors l'adresse est décalée du nombre de *mots* spécifié par la *valeur du décalage*. Si, de plus, Mode de décalage est à 1, les 5 de poids faible du premier octet de Décalage spécifient le registre dans lequel la *valeur du décalage* est stockée. Sinon, la *valeur du décalage* utilisée est Décalage lui-même. La *valeur du décalage* est interprétée comme un entier relatif.

Dans la suite, sauf mention contraire, nous désignerons par « opérande de l'instruction » la *valeur de l'opérande*.

La signification du champ Mode d'opération est décrite au paragraphe 1.6, page 3.

1.5 Unité arithmétique et logique

Les calculs logiques sont effectués bit-à-bit sur 32 bits : non, et, ou, ou exclusif, non simulanéité, égalité, décalages à gauche et à droite avec pertes, rotations à gauche et à droite.

Les entiers sont représentés sur 32 bits.

Les calculs arithmétiques suivants sont proposés : incrémentation, complément à deux, valeur absolue, addition, soustraction et multiplication.

Les comparaisons arithmétiques suivantes sont proposées : égal, plus petit, plus petit ou égal plus grand, plus grand ou égal.

Deux drapeaux d'états, z et o, mémorisent respectivement si le résultat des dernières opérations ou comparaisons est nul ou vrai respectivement, et si ces opérations ont conduit à un dépassement de capacité.

Les drapeaux d'états sont initialement baissés. Les opérations ou comparaisons ne peuvent que lever le drapeau o : s'il est déjà levé au début d'une opération ou comparaison, il le reste.

1.6 Modes d'opération

Pour toutes les instructions, le Mode d'opération définit trois attributs, a, b et c, énumérés du bit de poids fort au bit de poids faible, qui influent sur le comportement de l'instruction :

– Lorsque l'instruction n'est ni une instruction arithmétique ni une instruction logique, les attributs ont la signification suivante :

a	**Baissé :** les drapeaux z et o sont inchangés. f est levé en cas d'opération non valide, inchangé sinon. **Levé :** les drapeaux z et o sont baissés à la fin de l'exécution. f est levé en cas d'opération non valide, baissé sinon.

b **baissé**	c **baissé**	l'instruction est toujours exécutée
b **levé**	c **baissé**	l'instruction est exécutée seulement si z est levé
b **baissé**	c **levé**	l'instruction est exécutée seulement si o est levé
b **levé**	c **levé**	l'instruction est exécutée seulement si f est levé

– Pour une instruction arithmétique ou logique, et pour IGET, les attributs ont la signification suivante :

a	**Baissé :** les entiers sont naturels. **Levé :** les entiers sont relatifs.
b	**Baissé :** le drapeau z est écrasé à chaque opération et comparaison. **Levé :** le drapeau z est combiné au nouveau résultat selon c.
c	**Baissé :** la combinaison est disjonctive, $z \vee z_0 \to z$. **Levé :** la combinaison est conjonctive, $z \wedge z_0 \to z$. *(où z_0 est le résultat de l'opération en cours)*

2 Jeu d'instructions natif

NOP	00	Ne fait rien

Contrôle de l'exécution

JMP	01	Saute à l'adresse indiquée par l'opérande[1]

Accès à la mémoire

STO	02	Copie le registre à l'adresse indiquée par l'opérande[1]
LDA	03	Copie l'opérande dans le registre

Gestion de la pile

PUSH	04	Empile l'opérande
POP	05	Dépile dans le registre Si le registre est *sp*, la valeur dépilée est oubliée.
PUSHJ	06	Empile *pc* et saute à l'adresse indiquée par l'opérande[1]

Opérations unaires arithmétiques

INCR	08	Met dans le registre le résultat de l'incrémentation appliquée sur l'opérande
MIN	09	... du complément à deux ...
ABS	0A	... du passage à la valeur absolue ...

Opérations unaires logiques

NOT	0B	... du non bit-à-dit ...
LSH	0C	... du décalage à gauche avec perte ...
RSH	0D	... du décalage à droite avec perte ...
LROT	0E	... de la rotation à gauche ...
RROT	0F	... de la rotation à droite ...

Opérations binaires arithmétiques

ADD	10	... de l'addition appliquée sur l'opérande et sur l'ancienne valeur du registre
SUB	11	... de la différence ... ($v_{\text{registre}} - v_{\text{opérande}}$)
MUL	12	... de la multiplication ...

[1] L'adresse est interprétée comme relative à l'adresse de l'instruction courante si l'instruction est courte et l'opérande est constante (mode d'adressage `0b00`), absolue sinon.

Opérations binaires logiques

AND	13	... du et bit-à-dit ...
OR	14	... du ou bit-à-dit ...
XOR	15	... du ou exclusif bit-à-dit ...
NAND	16	... de la non-simulanéité bit-à-dit ...
NOR	17	... de la nullité simultanée bit-à-dit ...
NXOR	18	... de l'égalité bit-à-dit ...

Comparaisons arithmétiques

EQ	19	Opérande égale à la valeur du registre ?
NEQ	1A	... différente ... ?
LE	1B	... inférieure ou égale ... ?
LT	1C	... strictement inférieure ... ?
GE	1D	... supérieure ou égale ... ?
GT	1E	... strictement supérieure ... ?

3 Interface et programme assembleur

Le programme terminal simule le fonctionnement du micro-processeur qu'on suppose monté sur une carte hôte disposant des caractéristiques suivantes :

- un ensemble d'afficheurs sept segments adapté à l'affichage d'une horloge
- un signal d'horloge (indépendante de l'horloge interne du microprocesseur) qui bat la seconde
- un interfaçage texte ASCII qui peut faire penser à un clavier et une imprimante

3.1 Horloge

3.1.1 Matériel

Les sorties correspondant au registre o_1 et aux douze premiers bits du registre o_2 sont reliés à six afficheurs sept segments $a_0, a_1, \ldots, a_5$, et à deux séparateurs d_0 et d_1, de la manière suivante :

o_1						o_2	
7 bits	7 bits	1 bit	7 bits	7 bits	1 bit	7 bits	7 bits
a_0	a_1	d_0	a_2	a_3	d_1	a_4	a_5

Concrètement, $a_0, a_1, \ldots, a_5$, sont supposés alignés de gauche à droite, d_0 séparant a_1 et a_2, d_1 séparant a_3 et a_4. Chaque afficheur sept segments est supposé câblé de la manière suivante, en notant $b_0, b_1, \ldots, b_6$ les septs bits de données, du poids faible au poids fort.

$$\begin{array}{ccc} & b_6 & \\ b_4 & & b_5 \\ & b_3 & \\ b_1 & & b_2 \\ & b_0 & \end{array}$$

Toutes les secondes, le bit de poids faible de l'entrée correspondant au registre i_1 est levé à 1 s'il ne l'était pas déjà (s'il l'était, cela signifie qu'une seconde est

passée inaperçue). Ce bit est baissé lorsque le bit de poids fort de la sortie o_2 est à 1.

3.1.2 Logiciel

Pour dessiner un chiffre sur un afficheur, le programme assembleur utilise une ressource qui contient la « police » à utiliser : il s'agit d'une séquence de dix octets contigüs en mémoire, les septs premiers bits de chaque octet spécifiant les segments à éclairer, le huitième étant ignoré.

Une sous-routine permet d'extraire de cette ressource le « glyphe » correspondant au chiffre placé dans r_0, et de mettre celui-ci dans r_2 : l'accès à la mémoire étant effectué par mot de quatre bits, une division euclidienne spécialisée est programmée pour diviser r_0 par 4, le quotient indiquant le mot de la ressource à lire, le reste l'octet intéressant parmi les quatre lus.

Pour éviter les scintillements, r_14 et r_15 servent de tampon aux registres o_1 et o_2. Pour éviter un décompte cahotique, le contenu des tampons est copié dans les registres de sorties à chaque battement d'horloge, et non dès la fin de la préparation du tampon suivant.

À chaque afficheur est associé une sous-routine pour charger le glyphe stocké dans r_2 au sein des tampons, sans modifier le reste de l'affichage.

Le code de l'horloge n'écrit jamais dans la mémoire. En particulier, si l'horloge est chargée est ROM, elle peut fonctionner sans RAM.

3.2 Interface textuelle

3.2.1 Matériel

L'interface textuelle utilise un protocole rudimentaire sur 8 bits d'entrées et 4 bits de contrôle, il s'agit des 12 premiers bits de i_0 et de o_0. Les huit bits de poids faible correspondent au code ASCII du caractère à transmettre ; les quatre bits suivants indiquent respectivement un nouveau caractère (POST), un envoi en cours (SEND), un accusé de réception (RECEIVED), une requête d'interruption de la transmission (STOP). Le protocole est donc symétrique (en fait, presque symétique, en raison du paragraphe suivant).

La communication est alternée : SEND ne peut être simultanément levé dans i_0 et o_0. De plus, de chaque côté, pour chaque paquet de données envoyé, un paquet de données est attendu en réponse. Ainsi, quand SEND se baisse d'un côté, il ne se relèvera pas tant que le SEND de l'autre côté ne se sera pas levé puis rebaissé. C'est le microprocesseur qui « parle » en premier, puis le terminal.

Le protocole est synchrone : pour chaque caractère posté, un accusé de réception est attendu avant de poster un autre caractère. Soit A le côté où SEND est levé, B l'autre côté. Pour transmettre un caractère, A le prépare dans les huit premiers bits et lève POST, puis attend que B lève RECEIVED. B maintient RECEIVED levé tant que A ne baisse pas POST. A attend que B baisse RECEIVED avant de transmettre un autre caractère.

3.2.2 Logiciel

terminal affiche les caractères reçues sur la sortie standard. Lorsque c'est à lui de parler, il attend la saisie d'une ligne sur l'entrée standard avant de l'envoyer (sans le caractère de fin de ligne). Lorsque l'entrée commence par #load$_\sqcup$,

c'est le fichier dont le nom suit sur la ligne qui est envoyé. Lorsqu'une ligne vide est entrée, le caractère nul est envoyé. La simulation du microprocesseur est lente : sur les ordinateurs récents, la montre peut s'exécuter en temps réel, mais les transmissions textuelles prennent du temps. Lorsque l'envoi d'une ligne ou d'un fichier est long, le terminal affiche un indicateur de progression pendant la transmission : cet indicateur est géré par le logiciel terminal, non par le microprocesseur.

Le programme en ROM charge les caractères reçues dans la RAM, en commençant à l'adresse 0x1000 (plus basse adresse en RAM), et en concaténant les entrées successives, le début de chaque entrée étant aligné sur le mot qui succède immédiatement la fin de l'entrée précédente. Lorsqu'une entrée n'est composée que du simple caractère nul, le programme fait un appel (PUSHJ) à la sous-routine 0x1000 : au retour, le programme réaffiche un invite et recommence à écrire les entrées à partir de l'adresse 0x1000.

3.3 Utilisation

Le terminal n'effectue qu'une encapsulation du code Caml généré à partir de la *net-list*. Celle-ci spécifie que la ROM doit être initialisé à partir du fichier bios.rom. Ainsi, si nous souhaitons tester directement l'horloge, nous devons mettre dans ce fichier le code assemblé de l'horloge.

```
$ ./asm clock
$ ./rom clock.o bios.rom
$ ./terminal
```

Le programme horloge utilise les registres r_5, r_6, r_7, r_8, r_9, r_10 pour mémoriser le chiffre affiché dans chaque afficheur sept segments, et commence à initialiser ces registres à partir des 24 premiers bits du registre r_13, découpés en six paquets de quatre : aucune vérification de borne n'est effectuée, l'heure peut donc éventuellement être initialisée avec une heure incorrecte, comme 53 :87 :91 ! Tous les registres étant à 0 lors de l'initialisation du microprocesseur, l'heure commence à être décomptée par défaut à partir de 00 :00 :00.

L'interface textuelle fournit un outil pour régler le début de l'horloge à une heure arbitraire. Il s'agit du programme set_time.asm : ce programme lit les deux mots qui succèdent immédiatement son code et suppose qu'il s'agit d'une chaîne de caractères ASCII de la forme hh :mm :ss, où les : peuvent en réalité être des séparateurs quelconques. Il charge dans r_13 la valeur correspondante.

Ainsi, en assemblant set_time.asm et clock.asm de façon à ce qu'ils puissent être chargés en RAM (à l'adresse 0x1000, ce que fait l'interface textuelle), nous pouvons les utiliser avec l'interface textuelle et régler l'heure à volonté.

Attention : l'interface textuelle est excessivement lente à exécuter.

```
$./asm bios
$./rom bios
$./asm set_time --base 0x1000
$./asm clock --base 0x1000
$./terminal
READY
>#load set_time.o
>23:58:30
```

```
>

FINISHED
>#load clock.o
>
```

3.4 Déboguage

Afin de faciliter la mise au point des programmes assembleurs, le microprocesseur présente à l'extérieur le contenu du registre *pc* ainsi que le code d'opération de l'instruction en cours d'exécution. Lorsque la ligne #debug est saisie dans le terminal, celle-ci n'est pas envoyée au microprocesseur, mais le terminal active le déboguage (ou le désactive s'il était déjà actif). Lorsque le déboguage est activé, à chaque exécution d'un cycle de *net-list*, la valeur du *pc* courant et le nom de l'instruction en cours d'exécution est affiché, et une saisie de la part de l'utilisateur est attendue. Si la ligne saisie est vide, le déboguage continue l'exécution pas-à-pas ; si une valeur numérique est spécifiée, le déboguage pas-à-pas reprendra dès que *pc* sera égal à cette valeur (ceci permet de placer un point d'arrêt en 0x1000 afin de déboguer le programme chargé et non l'interface textuelle) ; si un texte non numérique est entré, le déboguage est désactivé.

4 Implémentation

4.1 Modifications de la syntaxe des *net-lists* d'entrées

Quelques améliorations apportées au programme netlist ont facilité la conception du microprocesseur. Voici les plus importantes :

- Les tableaux peuvent être multi-dimensionnels. Lorsqu'un tableau est utilisé comme partie variable des paramètres d'un modèle, une seule dimension doit être variable, et le nombre de paramètres passés dans la partie variable lors d'un appel à ce modèle devra être un multiple du produit des autres dimensions. Par exemple, soit le modèle exemple(a[1 .. 3, 1 .. n]), alors exemple devra être appelé avec un nombre de paramètres divisible par 3. Les tableaux sont étendus à plat en parcourant les dimensions de gauche à droite ([1,2] précède [2,1]).
- Les ROM et les RAM prennent un paramètre supplémentaire qui permet d'activer ou non la lecture (ou l'écriture). Elles peuvent toutes deux être éventuellement initialisées à partir d'un fichier, et le contenu de ce fichier peut être lié à la *net-list* à la compilation, ou chargé à chaque exécution.
- Les tables ne sont donc plus syntaxiquement équivalentes aux ROM : il n'y a pas de paramètres d'activation, et dans le cas d'une initialisation à partir d'un fichier, le contenu de ce fichier ne peut être lié qu'à la compilation.

4.2 Unité arithmétique et logique

L'unité arithmétique et logique est définie au sein du fichier alu.net par un ensemble de modèles de circuits opérant sur des entiers binaires de longueur quelconque.

– Pour les modèles de circuits à une seule opérande, la partie variable estun tableau à une dimension : la longueur des entiers binaires manipulés est égale au nombre de paramètres de la partie variable.
– Pour les modèles de circuits à deux opérandes, c'est une matrice dont la première dimension est 2 et la seconde est variable : le nombre de paramètres de la partie variable doit donc être pair, $2m$, la longueur des entiers binaires manipulés est alors m, la première opérande est constitué par les m premiers paramètres de la partie variable, la seconde par les m derniers.

La plupart des modèles de circuits prennent de plus un paramètre de flux supplémentaire r, placé avant la partie variable, qui spécifie si les entiers sont à considérer comme des entiers naturels (si r est à 0) ou relatifs (si r est à 1).

4.2.1 Addition

L'additionneur de von Neumann permet de calculer $A + B$ et $A + B + 1$ en temps parallèle logarithmique. Les entiers binaires renvoyés sont écrits avec un bit de plus que les opérandes, le calcul est donc effectué sans dépassement de capacité.

```
fAdd(a[0 .. 1, 0 .. n]) = (s[0 .. n + 1], s'[0 .. n + 1]) {
  if n = 0 {
    s[0] = a[0, 0] xor a[1, 0]
    s'[0] = ~ s[0]
    s[1] = a[0, 0] & a[1, 0]
    s'[1] = a[0, 0] | a[1, 0] }
  else
    for m = (n + 1) / 2 {
      (u[0 .. m], u'[0 .. m]) = fAdd(a[0 .. 1, 0 .. m - 1])
      (v[0 .. n - m + 1], v'[0 .. n - m + 1]) = fAdd(a[0 .. 1, m .. n])
      for i = 0 .. m - 1 {
        s[i] = u[i]
        s'[i] = u'[i] }
      for i = 0 .. n - m + 1 {
        s[i + m] = mux(u[m], v'[i], v[i])
        s'[i + m] = mux(u'[m], v'[i], v[i]) } } }
```

Le modèle de circuits réalisant l'addition s'en déduit presque directement. Pour les entiers naturels, il y a dépassement de capacité lorsque le bit excédentaire renvoyé par l'additionneur de von Neumann est à 1. Pour les entiers relatifs, nous observons que :

– La somme de deux entiers de signe contraire ne donne jamais lieu à un dépassement de capacité : le résultat est du même signe que l'opérande avec la plus grande valeur absolue, mais la valeur absolue du résultat est inférieure à celle-ci.
– La somme de deux entiers de même signe donne un entier de même signe : il y a dépassement de capacité lorsque l'écriture du résultat est de signe différent.

```
add(r, a[0 .. 1, 0 .. n]) = (s[0 .. n], o) {
  (s[0 .. n], o', s'[0 .. n + 1]) = fAdd(a[0 .. 1, 0 .. n])
  o = mux(r, (a[0, n] = a[1, n]) & (a[0, n] xor s[n]), o') }
```

4.2.2 Comparaisons

Pour l'égalité, il suffit de comparer chaque bit deux à deux. Pour avoir un modèle de circuits en temps parallèle logarithmique, nous effectuons cette comparaison par dichotomie.

```
eq(a[0 .. 1, 0 .. n]) = r {
  if n = 0
    r = a[0, 0] = a[1, 0]
  else
    for m = (n + 1) / 2 {
      r1 = eq(a[0 .. 1, 0 .. m - 1])
      r2 = eq(a[0 .. 1, m .. n])
      r = r1 & r2 } }
```

Les autres comparaisons sont dissymétriques par rapport à l'écriture du nombre. Nous discutons à la fois l'égalité et la stricte infériorité (par exemple) à chaque niveau de dichotomie. Nous coupons les deux nombres en deux et nous comparons séparemment les deux parties de poids faibles et les deux parties de poids forts. L'hérédité s'obtient simplement en propageant soit la stricte infériorité des parties de poids fort le cas échéant, soit le résultat de la comparaison les parties de poids faibles, en cas d'égalité des parties de poids fort.

Ainsi, le modèle de circuits suivant détermine en temps parallèle logarithmique si les deux entiers naturels passés en paramètre sont égaux ou si le premier est strictement inférieur au second.

```
compareN(a[0 .. 1, 0 .. n]) = (e, l) {
  if n = 0 {
    e = a[0, 0] = a[1, 0]
    l = a[0, 0] < a[1, 0]
  } else
    for m = (n + 1) / 2 {
      (e1, l1) = compareN(a[0 .. 1, 0 .. m - 1])
      (e2, l2) = compareN(a[0 .. 1, m .. n])
      e = e1 & e2
      l = mux(e2, l1, l2) } }
```

Comparer deux entiers relatifs commence déjà par comparer leurs signes : si ce sont les mêmes, nous les comparons comme s'il s'agissait d'entiers naturels. Dans l'implémentation, nous comparons séparemment le bit de poids fort, pour éviter de le comparer deux fois.

```
comparaison(r, a[0 .. 1, 0 .. n]) = (e, l) {
  (e', l') = compareN(a[0 .. 1, 0 .. n - 1])
  e = e' & (a[0, n] = a[1, n])
  l = (r xor (a[0, n] < a[1, n])) | l'}
```

Il suffit de spécialiser ce modèle de circuits pour obtenir les comparateurs habituels. Par exemple, pour le comparateur strictement plus petit :

```
lt(r, a[0 .. 1, 0 .. n]) = o {
  (e, l) = comparaison(r, a[0 .. 1, 0 .. n])
  o = l }
```

4.2.3 Opérations bit-à-bit

Il suffit de distribuer l'opération sur chaque bit du mot. Ainsi, pour distribuer un opérateur binaire op sur un mot, nous pouvons utiliser le modèle de circuits suivant :

```
bitwise(op : circuit, a[0 .. 1, 0 .. n]) = s[0 .. n] {
  for i = 0 .. n
    s[i] = op(a[0, i], a[1, i]) }
```

Il suffit ensuite de spécialiser ce modèle de circuits. Par exemple, pour le et bit-à-bit :

```
band(a[0 .. 1, 0 .. n]) = s[0 .. n] {
  s[0 .. n] = bitwise(&, a[0 .. 1, 0 .. n]) }
```

4.2.4 Décalages avec perte et rotations

Ce sont de simples recâblages, en temps parallèle constant.

Les décalages avec perte renvoient un dépassement de capacité lors d'une perte.

Pour les entiers relatifs, tout se passe comme si l'opérande était une opérande naturelle sur n bits, et le bit de signe (le $n+1$ème bit) est inchangé.

4.2.5 Soustraction et entiers relatifs

La valeur absolue d'un entier relatif sur $n+1$ bits peut éventuellement nécessiter $n+1$ bits : en effet, $|-2^n| = 2^n$. Le modèle de circuits suivant ne produit ainsi jamais de dépassement de capacité.

```
abs(a[0 .. n]) = s[0 .. n] {
  s'[0 .. n] = bnot(a[0 .. n])
  (s''[0 .. n], o) = incr(false, s'[0 .. n])
  for i = 0 .. n
    s[i] = mux(a[n], s''[i], a[i]) }
```

Le modèle de circuits suivant réalise l'opération inverse : à partir d'un entier naturel, il renvoie un entier relatif ayant même valeur absolue, positif lorsque *neg* est à 0, négatif lorsque *neg* est à 1. Lors de ce passage, il y a un dépassement de capacité si la valeur absolue est supérieure ou égale à 2^n pour un nombre positif, strictement supérieure à 2^n pour un nombre négatif.

```
sgn(neg, a[0 .. n]) = (s[0 .. n], o) {
  s'[0 .. n] = bnot(a[0 .. n])
  (s''[0 .. n], o') = incr(true, s'[0 .. n])
  for i = 0 .. n
    s[i] = mux(neg, s''[i], a[i])
  o = mux(neg, a[n] & share(|, a[0 .. n - 1]), a[n]) }
```

La soustraction s'obtient par la formule $A - B = A + \neg B + 1$. Nous utilisons le second membre du couple renvoyé par l'additionneur de von Neumann pour le calcul de $A + \neg B$ afin d'économiser l'incrémentation.

Pour les entiers naturels, il y a dépassement de capacité lorsque $A < B$.

Pour les entiers relatifs, il y a dépassement de capacité lorsque A et B sont de signes contraires et que le signe de A et celui du résultat sont différents.

```
sub(r, a[0 .. 1, 0 .. n]) = (s[0 .. n], o) {
  b'[0 .. n] = bnot(a[1, 0 .. n])
  (s'[0 .. n + 1], s[0 .. n], o') = fAdd(a[0, 0 .. n], b'[0 .. n])
  o = mux(r, (a[0, n] xor a[1, n]) & (a[0, n] xor s[n]),
    lt(false, a[0 .. 1, 0 .. n])) }
```

4.2.6 Multiplication

Nous effectuons des additions sans retenue en temps parallèle constant afin d'obtenir un multiplicateur en temps parallèle logarithmique.

```
csAdd(r0, s0, a[0 .. 3, 0 .. n]) = (s[0 .. n + 1], s'[0 .. n + 1]) {
  f[0] = s0
  r[0] = r0
  for i = 0 .. n {
    (e[i], f[i + 1]) = fulladd(a[0, i], a[1, i], a[2, i])
    (s[i], s'[i + 1]) = fulladd(a[3, i], e[i], f[i]) }
  s'[0] = r[0]
  s[n + 1] = f[n + 1] }
```

Pour sommer un ensemble d'entiers écrits en binaire mou, on applique l'addition ci-dessus par dichotomie. Le tableau des paramètres est de dimension variable en la longueur des nombres et en le nombre de termes dans la somme, mais il ne peut y avoir en pratique qu'une seule dimension variable. Nous regroupons donc les deux dimensions en une seule, et nous précisons une des deux dimensions explicitement dans un paramètre entier supplémentaire : l'autre dimension s'en déduira par division. Le nombre de termes est variable, il est plus simple de laisser le compilateur le déterminer, alors que le nombre de bits est constant : c'est donc cette dernière dimension qui est précisée explicitement.

Pour simplifier la suite, nous recâblons t en un tableau p (opération gratuite du point de vue électronique) pour retrouver le tableau à trois dimensions avec deux dimensions variables que nous voulions au départ.

Il y a dépassement lorsque l'une des additions sans retenue présente un bit excédentaire non nul : nous le vérifions par un arbre de *ou*.

```
dichoAdd(n : int, t[1 .. u, 0 .. 1]) = (s[0 .. n], o) {
  for nb = u / (n + 1)
    if nb = 1
      (s[0 .. n], o) = add(false, t[1 .. n + 1, 0], t[1 .. n + 1, 1])
    else {
      for i = 0 .. nb - 1 {
        p[i, 0 .. n, 0] = t[i * (n + 1) + 1 .. i * (n + 1) + 1 + n, 0]
        p[i, 0 .. n, 1] = t[i * (n + 1) + 1 .. i * (n + 1) + 1 + n, 1] }
      for i = 0 .. nb - 1 {
        m[i, 0, 0 .. n] = p[i, 0 .. n, 0]
        m[i, 1, 0 .. n] = p[i, 0 .. n, 1] }
      for l = nb / 2 - 1 {
```

```
    for i = 0 .. l
      m'[i, 0 .. 1, 0 .. n + 1] =
        csAdd(0, 0, m[2 * i .. 2 * i + 1, 0 .. 1, 0 .. n])
    for i = 0 .. l {
      p'[i, 0 .. n, 0] = m'[i, 0, 0 .. n]
      p'[i, 0 .. n, 1] = m'[i, 1, 0 .. n] }
    if nb mod 2 = 0
      (s[0 .. n], o') = dichoAdd(n, p'[0 .. l, 0 .. n, 0 .. 1])
    else
      (s[0 .. n], o') =
        dichoAdd(n, p'[0 .. l, 0 .. n, 0 .. 1], p[nb - 1, 0 .. n, 0 .. 1])
    o = o' | share(|, m'[0 .. l, 0 .. 1, n + 1]) } } }
```

Nous en déduisons le modèle de circuits suivant pour réaliser la multiplication de deux entiers naturels :

```
natmul(a[0 .. 1, 0 .. n]) = (s[0 .. n], o) {
  for i = 0 .. n, l = i / 2, d = i mod 2 {
    for j = 0 .. i - 1
      p[l, 2 * j + d] = 0
    for j = 0 .. n
      p[l, 2 * (i + j) + d] = a[1, i] & a[0, j]
    for j = i + n + 1 .. 2 * n
      p[l, 2 * j + d] = 0 }
  (s[0..n], o') = dichoAdd(n, p[0 .. n / 2, 0 .. 2 * n + 1])
  o = o' | share(|, p[0 .. n / 2, 2 * (n + 1) .. 4 * n + 1]) }
```

Pour multiplier deux entiers relatifs, il suffit de multiplier leur valeur absolue, puis d'appliquer la règle des signes.

```
mul(r, a[0 .. 1, 0 .. n]) = (s[0 .. n], o) {
  (s0[0 .. n], o0) = natmul(a[0 .. 1, 0 .. n])
  a'[0, 0 .. n] = abs(a[0, 0 .. n])
  a'[1, 0 .. n] = abs(a[1, 0 .. n])
  (s1[0 .. n], o1) = natmul(a'[0 .. 1, 0 .. n])
  (s2[0 .. n], o2) = sgn(a[0, n] xor a[1, n], s1[0 .. n])
  for i = 0 .. n
    s[i] = mux(r, s2[i], s0[i])
  o = mux(r, o1 | o2, o0) }
```

Pour des raisons d'efficacité, le circuit ci-dessus n'est pas inclus dans la *net-list* du microprocesseur : celui-ci effectue toujours une multiplication d'entiers naturels, au risque d'erreurs dans les dépassements de capacité.

4.3 La partie contrôle

Les spécifications de la mémoire limitent la vitesse d'exécution à la lecture d'un mot mémoire par cycle : nous effectuons en effet une lecture mémoire par cycle, nous ne pouvons donc globalement pas réduire le nombre de cycles nécessaires à l'exécution du programme assembleur (sauf cas particuliers : si par

exemple, une instruction prend comme opérande le contenu de sa propre adresse mémoire, nous lirons deux fois cette adresse alors qu'une seule lecture suffisait).

La partie contrôle est un automate : à chaque état, l'automate reçoit un mot lu en mémoire, ainsi que quelques informations supplémentaires transmises par l'état précédent (notamment les valeurs des registres). Après traitement de ces données, l'automate peut éventuellement écrire un mot en RAM, puis spécifie l'état suivant, ainsi que le prochain mot à lire en mémoire.

Les registres sont donc tous directement accessibles et directement affectables : il est donc plus rapide de lire une valeur dans un registre, ce qui peut se faire au cours d'un cycle, que de lire une valeur en mémoire, ce qui nécessite d'attendre le cycle suivant.

L'automate comporte cinq états, s_0, s_1, s_2, s_3, s_4. Trois registres (de *net-list*) pouvait donc suffire pour conserver l'état en cours, mais nous en utilisons un par état : ainsi, une seule valeur binaire permet de discuter si nous nous trouvons dans un cas particulier, et incrémenter l'état consiste en un simple décalagage.

s_0 et s_1 ne servent qu'à l'initialisation : s_0, état initial et commande la lecture de 0x0FFF, plus haute adresse en ROM, qui est supposée contenir la plus haute adresse en RAM, et passe à s_1. s_1 stocke la valeur lue dans *sp*, commande la lecture de 0x0000, première instruction à exécuter, et passe à s_2.

L'état s_2 décode l'instruction et l'exécute si elle est au format court et si l'opérande n'est pas une d'indirection. Si elle est au format long, nous passons à s_3 en commandant la lecture du second mot, sinon, si l'opérande est une indirection, nous passons à s_4 en commandant la lecture de l'adresse pointée.

L'état s_3 exécute l'instruction au format long si l'opérande n'est pas une indirection, sinon passe à s_4 en commandant la lecture de l'adresse pointée.

L'état s_4 exécute l'instruction avec la valeur lue à l'adresse que pointait l'opérande.

Après l'exécution de l'instruction, nous repassons en s_2 en commandant la lecture de la valeur pointée par le nouveau *pc*.

Ainsi, une instruction au format court sans indirection est exécutée en un cycle, une instruction au format long sans indirection ou au format court avec indirection est exécutée en deux cycles, une instruction au format long avec indirection est exécutée en trois cycles.

Donc écrire en RAM à un emplacement dont l'adresse relativement au pc est exprimable sur 16 bits est aussi rapide que de charger une valeur dans un registre, c'est-à-dire un cycle de simulation.

Pour décoder les instructions, nous utilisons le circuit suivant :

```
decodeinstructions(partone, parttwo, readoperand, read[0 .. 31]) =
  (long, common[0 .. 14], operand[0 .. 31], offset[0 .. 15])
{
  long = read[0]
  common[0 .. 14] = multimux(partone, read[1 .. 15], data[0 .. 14])
  read16[0 .. 31] = padding(31, read[16 .. 31])
  readplace[0 .. 31] = multimux(partone, read16[0 .. 31], read[0 .. 31])
  operand[0 .. 31] = multimux(readoperand, data[31 .. 62], readplace[0 .. 31])
  offset[0 .. 15] = multimux(partone, 0[16], data[15 .. 30])
  hipart[0 .. 15] = multimux(partone, read[16 .. 31], data[15 .. 30])
  data[0 .. 62] = Z(common[0 .. 14], hipart[0 .. 15], operand[0 .. 31]) }
```

partone, parttwo et readoperand correspondent respectivement aux états s_2, s_3 et s_4. Nous mémorisons dans data les valeurs lues au cours des états précédents.

common correspond à la partie commune des deux formats d'instructions.

Si l'opérande est sur 16 bits, on la considère comme un entier relatif qu'on étend sur 32 bits : notons que ceci est effectué indépendamment de l'instruction, en particulier même si l'instruction en cours d'exécution est une instruction de l'ALU portant sur une opérande naturelle.

À chaque cycle, nous regénérons la table des registres. execution indique si l'exécution a bien lieu dans l'état courant (c'est-à-dire si nous n'avons pas besoin d'effectuer une lecture supplémentaire en mémoire avant). writepoinded indique s'il faut écrire dans un registre quelconque, et alors pointedregister est un tableau qui vaut vrai à l'indice du registre à écrire, faux partout ailleurs. spvalue contient l'adresse à écrire spécifiquement dans sp (ce qui peut se faire simultanément à l'écriture dans un autre registre, avec POP).

```
  for i = 0 .. 23
    if i = 16 {
      pointedregistervalue[i, 0 .. 31] =
        multimux(execution & writepointed & pointedregister[i],
          newregistervalue[0 .. 31], registers[i, 0 .. 31])
      newspvalue[0 .. 31] =
multimux(execution & (pop | push | pushj),
          spvalue[0 .. 31], pointedregistervalue[i, 0 .. 31])
      modifiedregisters[i, 0 .. 31] =
        multimux(states[1], initsp[0 .. 31], newspvalue[0 .. 31]) }
    else
      modifiedregisters[i, 0 .. 31] =
        multimux(
          execution & writepointed & pointedregister[i],
          newregistervalue[0 .. 31], registers[i, 0 .. 31])
```