Description of mono and multi-cycles MIPS

Pascal Garcia

21 juillet 2022

Table des matières

Chapitre 1

Introduction

Nous allons réaliser dans ce cours, un sous-ensemble de l'architecture *MIPS* 32 bits. Nous verrons tout d'abord, au chapitre 2, une architecture mono-cycle dans laquelle toutes les instructions s'exécuteront en un cycle d'horloge. Après avoir vu les limitations de cette approche simple, nous réaliserons, au chapitre 3, une implémentation multi-cycles où différentes instructions pourront avoir des temps d'exécution différents. Nous ajouterons aussi à cette nouvelle implémentation la gestion des interruptions et des exceptions et une gestion de la mémoire virtuelle. Ces deux premiers chapitres sont une synthèse de certains chapitres (essentiellement le chapitre 5) du livre [1]. Nous avons ajouté par rapport à ce livre la gestion d'une partie du coprocesseur 0 et la gestion de la mémoire virtuelle.

Chapitre 2

Architecture *MIPS* mono-cycle

Nous allons réaliser une architecture simple où toutes les instructions s'exécuteront en un cycle. De plus, le code et les données seront placés dans des mémoires séparées.

2.1 Modèle de programmation

2.1.1 Espace d'adressage

Le bus d'adresse de la machine *MIPS* que nous étudions est de 32 bits et l'unité élémentaire de mémorisation est l'octet. On peut donc accéder à 2^{32} octets. En fait, pour cette implémentation nous aurons deux mémoires séparées : une pour les instructions et une pour les données. Nous aurons donc accès à 2^{32} octets pour chacune des mémoires.

2.1.2 Registres

Nous présentons dans la table 2.1 les 32 registres que comporte l'unité centrale de cette architecture. La première colonne de cette table indique le nom utilisé en assembleur pour décrire ce registre, la deuxième colonne donne le numéro du registre et la dernière indique l'usage qui en est généralement fait.

2.1.3 Jeu d'instructions

Nous allons étudier un sous-ensemble du jeux d'instructions de l'architecture *MIPS* 32 bits, afin de pouvoir réaliser l'architecture de cet ordinateur. Les instructions que nous allons étudier seront de trois types :

- les instructions de référence à la mémoire *load word* (`lw`) et *save word* (`sw`) ;
- les instructions arithmétiques et logiques `add`, `sub`, `and`, `or` et `slt` ;
- les instructions de saut *branch equal* (`beq`) et *jump* (`j`).

Nous allons maintenant décrire les différentes instructions et donner leur format. Notons que toutes les instructions sont codées sur 32 bits.

- `lw $s1, 100($s2)`
  Soit a la valeur du registre $s2$ plus 100. Cette instruction charge le mot de 32 bits commençant à l'adresse a dans le registre `$s1`. L'instruction `lw rt, v(rs)` est codée en langage machine :

100011	rs	rt	v
6 bits	5 bits	5 bits	16 bits

Nom	Numéro	Usage
$zero	0	la constante zéro
$at	1	réservé pour l'assembleur
$v0	2	évaluation d'une expression et résultat d'une fonction
$v1	3	évaluation d'une expression et résultat d'une fonction
$a0	4	premier argument d'une fonction
$a1	5	deuxième argument d'une fonction
$a2	6	troisième argument d'une fonction
$a3	7	quatrième argument d'une fonction
$t0	8	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t1	9	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t2	10	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t3	11	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t4	12	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t5	13	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t6	14	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t7	15	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$s0	16	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s1	17	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s2	18	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s3	19	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s4	20	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s5	21	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s6	22	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$s7	23	temporaire et sauvegardé (est préservé lors de l'appel d'une fonction)
$t8	24	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$t9	25	temporaire (n'est pas préservé lors de l'appel d'une fonction)
$k0	26	réservé pour le système d'exploitation
$k1	27	réservé pour le système d'exploitation
$gp	28	pointeur vers la zone des variables globales
$sp	29	pointeur de pile
$fp	30	pointeur vers la zone *frame pointer*
$ra	31	adresse de retour d'une fonction

TABLE 2.1 – Les 32 registres de la machine *MIPS*

Notons que `v` est sur 16 bits et peut donc représenter des déplacements de $-2^{15} = -32768$ à $2^{15} - 1 = 32767$

- `sw $s1, 100($s2)`
  Soit a la valeur du registre $s2$ plus 100. Cette instruction sauvegarde le mot de 32 bits contenu dans le registre `$s1` à l'adresse a. L'instruction `sw rt, v(rs)` est codée en langage machine :

101011	rs	rt	v
6 bits	5 bits	5 bits	16 bits

- `add $s1, $s2, $s3`
  Cette instruction place dans le registre `$s1` la somme du registre `$s2` et `$s3`. L'instruction `add rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  Notons les 6 premiers bits (qui représentent l'*opcode*) des instructions arithmétiques et logiques sont toujours à 0. Ce qui différencie ces différentes instructions est la valeur des 6 derniers bits de l'instruction (qui se nomme *funct*).

- `sub $s1, $s2, $s3`
  Cette instruction place dans le registre `$s1` la différence entre le registre `$s2` et le registre `$s3` (`$s2` - `$s3`). L'instruction `sub rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	100010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  Notons les 6 premiers bits (qui représentent l'*opcode*) des instructions arithmétiques et logiques sont toujours à 0. Ce qui différencie ces différentes instructions est la valeur des 6 derniers bits de l'instruction (qui se nomme *funct*).

- `and $s1, $s2, $s3`
  Cette instruction place dans le registre `$s1` le *et logique bit à bit* des registres `$s2` et `$s3` (`$s2` $\wedge$ `$s3`). L'instruction `and rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	100100
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `or $s1, $s2, $s3`
  Cette instruction place dans le registre `$s1` le *ou logique bit à bit* des registres `$s2` et `$s3` (`$s2` $\vee$ `$s3`). L'instruction `or rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	100101
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `slt $s1, $s2, $s3`
  Si (`$s2` $<$ `$s3`) alors `$s1` $= 1$ sinon `$s1` $= 0$ L'instruction `slt rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	101010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `beq $0, $1, v`
  Cette instruction permet de faire un saut conditionel relativement à la valeur du compteur ordinal (`PC`). Si `$s0` = `$s1` la prochaine instruction sera cherchée en `PC` + 4 + $v \times 4$ sinon elle sera cherchée en `PC` + 4.
  Par exemple, soit le programme suivant :

```
8:  add $1,$2,$3
12: beq $0,$1,2
16: ...
20: ...
24: ...
28: ...
```

  Notons que les adresses des instructions vont de 4 en 4 car une instruction est sur 32 bits. À la fin de l'exécution de l'instruction `beq $0,$1,2`, si `$s0` = `$s1` on aura `PC` $= 12 + 4 + 4 * 2 = 24$ sinon on aura `PC` = 16.
  L'instruction `beq rs, rt, v` est codée en langage machine :

000100	rs	rt	v
6 bits	5 bits	5 bits	16 bits

- `j v`
  Cette instruction permet de faire un saut inconditionel à l'adresse `v` (nous allons voir que c'est un peu plus compliqué). L'instruction `j v` est codée en langage machine :

000010	v
6 bits	26 bits

  On voit que la valeur de saut `v` est codée sur 26 bits. Or, une adresse est sur 32 bits. La valeur `v` est en fait multipliée par 4 et les 4 bits de poids forts manquant sont les 4 bits de poids forts de `PC` + 4.
  Soit le programme suivant :

```
Loop: 80000: ...
      80004: ...
      80008: ...
      80012: ...
      80016: ...
      80020: j Loop
      80024: ...
```

  L'instruction `j Loop` (qui permet de revenir à l'adresse `80000`) est écrite de manière abusive, car dans l'instruction `j Loop`, `Loop` ne vaut pas `80000`. Néanmoins, on écrit de cette manière pour que ce soit plus lisible. En fait, il faudrait écrire `j 20000` pour faire le saut correctement. En effet, la valeur de `PC` est de 80020 donc celle de `PC` + 4 est de 80024. Les 4 bits de poids forts de 80024 (sur 32 bits) sont à zéro et donc le saut est effectué à l'adresse $20000 * 4$ sur 28 bits auxquels ont ajoute les 4 bits de poids forts à zéro. On obtient donc bien l'adresse 80000 sur 32 bits.

2.1.4 Exemple de programme

Nous allons présenter un petit programme en langage C, puis sa traduction en assembleur et en langage machine.

```
if (i != j) f = g + h;
else f = g - h;
```

Nous supposons que les variables `f`, `g`, `h`, `i` et `j` sont représentées respectivement par les registres `$s0`, `$s1`, `$s2`, `$s3` et `$s4`. De plus, nous supposons que le programme commence à l'adresse `0`. La traduction en assembleur de ce programme est la suivante :

```
       0: beq $s3, $s4, Else
       4: add $s0, $s1, $s2
       8: j Exit
Else: 12: sub $s0, $s1, $s2
Exit: 16:
```

Enfin, la traduction en langage machine donne le programme suivant :

```
 0: 000100 10011 10100 0000000000000010
 4: 000000 10001 10010 10000 00000 100000
 8: 000010 00000000000000000000000100
12: 000000 10001 10010 10000 00000 100010
16:
```

2.1.5 Exercice

Traduire en assembleur puis en langage machine le programme suivant :

```
A[300] = h + A[300]
```

On supposera que l'adresse de `A` est dans le registre `$t1` et la valeur de `h` dans le registre `$s2`.

2.2 Architecture

L'architecture que nous allons présenter dans ce chapitre décodera et exécutera chaque instruction sur un seul cycle d'horloge. Pour ce faire, nous aurons besoin de deux mémoires : une pour les instructions et une pour les données.
Nous allons tout d'abord présenter les différents éléments de l'unité de traitement et les liens entre eux, puis l'unité de contrôle et enfin nous parlerons des performances de cette architecture.

2.2.1 Unité de traitement

La figure 2.1 présente les blocs principaux de l'unité de traitement. Nous allons détailler chacun de ces éléments.

Compteur ordinal (`PC`)

Le compteur ordinal est un registre de 32 bits qui permet de stocker l'adresse de la prochaine instruction à décoder et exécuter. L'entrée de ce registre est stockée à chaque front montant de l'horloge.

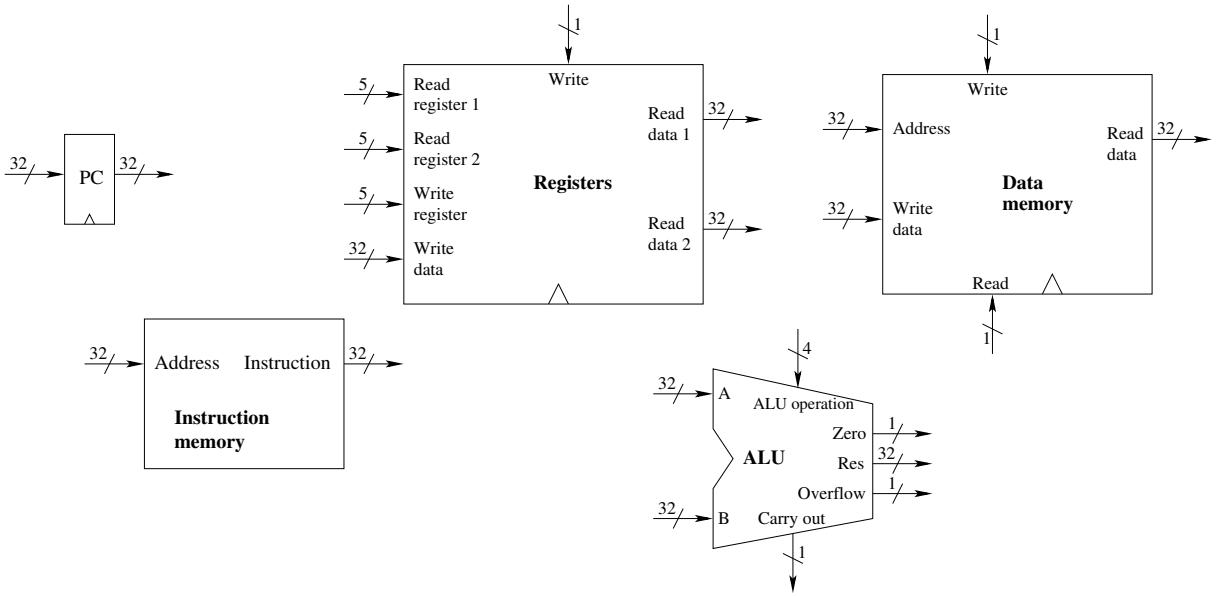


FIGURE 2.1 – Blocs principaux de l'unité de traitement

Mémoire des instructions (*Instruction memory*)

La mémoire des instructions contiendra le programme. Elle met en sortie les 32 bits commençant à l'adresse spécifiée sur la broche `Address`. Le contenu élémentaire de stockage de cette mémoire est l'octet, mais on peut y lire des données de 8, 16 ou 32 bits [1]. On y lira ici toujours des valeurs sur 32 bits.

Mémoire des données (*Data memory*)

La mémoire des données contiendra les données du programme. La mémoire met sur sa sortie `Read data` les 32 bits de la mémoire commençant à l'adresse `Address` si la broche `Read` est à `1`. Si la broche `Write` est à `1`, au front montant de l'horloge, la donnée en entrée sur la broche `Write data` est copiée dans la mémoire à partir de l'adresse `Address`. Le comportement est indéfini si `Read` et `Write` sont tous les deux à `1`.

Banc de registres (*Registers*)

Le banc de registre va contenir les 32 registres de la machine *MIPS*. La sortie `Read data 1` a pour valeur le contenu du registre dont le numéro est donné sur la broche `Read register 1`. La sortie `Read data 2` a pour valeur le contenu du registre dont le numéro est donné sur la broche `Read register 2`. La valeur présente sur l'entrée `Write data` sera copiée dans le registre de numéro `Write register` au top d'horloge si l'entrée `Write` est à `1`.

Exercice 1 : réaliser le banc de registres.

Unité arithmétique et logique (`ALU`)

La table 2.2 décrit la valeur calculée sur la sortie `Res` en fonction des entrées `A`, `B` et `ALU operation`. La sortie `Zero` est à `1` ssi `Res` est à `0`, la sortie `Carry out` est à `1` ssi il y a une retenue (notons

1. c'est le même principe que les mémoires entrelacées que nous avons déjà étudiées

`ALU operation`	Fonction
`0000`	$A \wedge B$
`0001`	$A \vee B$
`0010`	$A + B$
`0110`	$A - B$
`0111`	si $A < B$ alors 1 sinon 0
`1100`	$\neg(A \vee B)$

TABLE 2.2 – Fonctionnement de l'unité arithmétique et logique

que cette broche n'aura de signification que si l'on effectue une addition, une soustraction). La sortie `Overflow` est à `1` ssi il y a un débordement dans le calcul effectué par l'unité arithmétique et logique (notons que cette broche n'aura de signification que s'il on effectue une addition, une soustraction ou une comparaison). La table 2.3 indique comment se calcule un débordement en complément à deux.

Opération	A	B	Il y a débordement ssi le résultat est :
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

TABLE 2.3 – Détection d'un débordement

Exercice 2 : réaliser l'unité arithmétique et logique.

Connexions entre les différents éléments de l'unité de traitement

Exercice 3 : ajouter les connexions et les composants nécessaires pour mettre à jour le compteur ordinal `PC`.

Exercice 4 : ajouter les connexions et les composants nécessaires pour relier la mémoire des instructions au banc de registres.

Exercice 5 : ajouter les connexions et les composants nécessaires pour relier la mémoire des instructions et le banc de registres à l'unité arithmétique et logique.

Exercice 6 : relier l'unité arithmétique et logique à la mémoire des données.

Exercice 7 : relier l'unité arithmétique et logique et la mémoire des données au banc de registres.

2.2.2 Unité de contrôle

Contrôle de l'unité arithmétique et logique

Pour simplifier la réalisation de l'unité de contrôle, nous allons tout d'abord réaliser une sous-unité de contrôle qui gèrera l'entrée `ALU operation` de l'unité de traitement. Comme indiqué dans la figure 2.2, cette sous-unité, que l'on nommera *ALU control*, prend en entrée les bits 5 à 0 de

l'instruction (ces bits correspondent au champ `funct` d'une instruction arithmétique ou logique) et une sortie de l'unité de contrôle `ALUOp` permettant d'indiquer le type de l'instruction (instruction de sauvegarde/chargement, instruction de saut conditionel, ou instruction arithmétique ou logique). La table 2.4 présente la relation entre la partie `funct` de l'instruction, la sortie `ALUOp` de l'unité de contrôle et l'entrée `ALU operation` de l'unité arithmétique et logique.

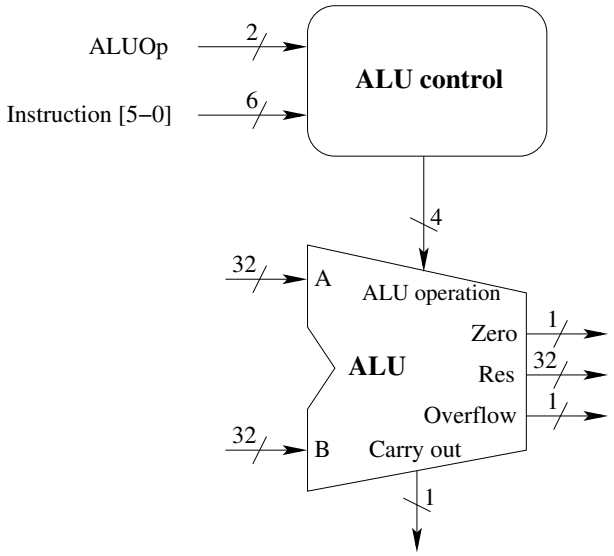


FIGURE 2.2 – Sous-unité de contrôle permettant de gérer l'unité arithmétique et logique

Opcode	ALUOp	Instruction	Champs `funct`	action de l'`ALU`	`ALU operation`
0x23	00	lw	XXXXXX	+	0010
0x2B	00	sw	XXXXXX	+	0010
0x04	01	beq	XXXXXX	−	0110
0	10	add	100000	+	0010
0	10	sub	100010	−	0110
0	10	and	100100	$\wedge$	0000
0	10	or	100101	$\vee$	0001
0	10	slt	101010	set on less than	0111

TABLE 2.4 – Les valeurs des différents bits de l'entrée *ALU operation* de l'unité arithmétique et logique dépendent des bits de contrôle *ALUOp* et du champ *funct*

Exercice 8 : réaliser la sous-unité de contrôle `ALU control`.

Contrôle principal

La table 2.5 résume les lignes de contrôle que nous avons élaboré lors de la création de l'unité de traitement. Ces lignes de contrôle et les lignes `ALUOp` de la sous-unité de traitement ne dépendent que de l'opcode de l'instruction.

Exercice 9 : réaliser l'unité de contrôle.

Ligne de contrôle	Effet quand la valeur est à `0`	Effet quand la valeur est à `1`
`RegDst`	le numéro du registre destination vient du champ `rt` (bits `20:16`)	le numéro du registre destination vient du champ `rd` (bits `15:11`)
`RegWrite`	aucun	le registre de numéro `Write register` est écrit avec la donnée `Write data`
`ALUSrc`	la deuxième opérande (`B`) de l'`ALU` provient de la sortie `Read data 2` du banc de registre	la deuxième opérande (`B`) de l'`ALU` provient des 16 bits de poids faibles de l'instruction qui ont été étendu (en prenant en compte le signe) sur 32 bits
`Branch`	le `PC` est remplacé par la sortie de l'additioneur qui calcule la valeur `PC` + 4	le `PC` est remplacé par la sortie de l'additioneur qui calcul l'adresse de saut
`Jump`	le `PC` est calculée en fonction de la ligne de contrôle `Branch`	le `PC` est remplacé par l'adresse de saut calculée par la combinaison des 26 bits de poids faibles de l'instruction multipliés par 4 et des 4 bits de poids fort de `PC` + 4
`MemRead`	aucun	Les 32 bits de la mémoire des données commençant à l'adresse désignée par `Address` sont placés sur la sortie `Read data`
`MemWrite`	aucun	Les 32 bits de la mémoire des données commençant à l'adresse désignée par `Address` sont remplacés par l'entrée `Write data`
`MemToReg`	La valeur sur la broche `Write data` du banc de registres vient de l'`ALU`	La valeur sur la broche `Write data` du banc de registres provient de la mémoire des données

TABLE 2.5 – L'effet de chacune des lignes de contrôle (en dehors des lignes $ALUOp$)

2.2.3 Performances

Nous allons supposer que les temps de fonctionnement des blocs principaux de l'unité de traitement sont les suivants :

- les mémoires des instructions et des données : 200 picosecondes (ps) ;
- l'unité arithmétique et les additionneurs : 100 ps ;
- le banc de registre (en lecture ou en écriture) : 50 ps.

On suppose aussi que les multiplexeurs, l'unité et la sous-unité de contrôle, les accès au compteur ordinal, les unités d'extension de signes et les câbles n'ont pas de délai. On supposera enfin que dans un programme, on a en moyenne : 25% d'instruction de chargement (`lw`), 10% de sauvegarde (`sw`), 45% d'instructions arithmétiques et logiques (`add`, `sub`, `and`, `or` et `slt`), 15% d'instructions de branchement conditionel (`beq`) et 5% d'instructions de saut inconditionel (`j`).

Exercice 10 : Calculer le temps moyen d'exécution d'une instruction pour les deux implémentations suivantes :

1. l'implémentation que nous venons de réaliser où chaque instruction dure un cycle d'horloge d'une durée fixée ;
2. une implémentation où chaque instruction s'exécute en un cycle d'horloge dont la durée est adaptée à l'instruction en cours d'exécution (cette approche n'est pas pratique, mais permettra de voir ce que l'on perd lorsque chaque instruction doit avoir la même durée d'exécution).

Chapitre 3

Architecture *MIPS* multi-cycles

Nous allons réaliser une implémentation de l'architecture *MIPS* dans laquelle chaque instruction sera décomposée en une série d'étapes, chacune prenant un cycle d'horloge. Nous verrons que procéder ainsi permet de réutiliser plusieurs fois le même bloc de l'unité de traitement lors de l'exécution d'une instruction. De plus, comme nous l'avions remarqué au chapitre précédent, les performances seront meilleures que dans le cas de l'implémentation où chaque instruction était exécutée en un seul cycle, car le temps d'exécution sera dépendant de l'instruction en cours et non de l'instruction la plus longue.

Nous allons aussi ajouter la gestion des interruptions, un mode utilisateur et superviseur et un mécanisme de protection de la mémoire. Nous aurons ainsi une architecture assez complète.

Dans la machine *MIPS*, il existe plusieurs coprocesseurs. Le coprocesseur numéro 1 s'occupe des calculs en nombre flottant. Nous n'avons pas implémenté ce coprocesseur dans le chapitre précédent et nous ne l'implémenterons pas non plus ici. Par contre, nous allons implémenter une partie du coprocesseur 0 qui permet de gérer les interruptions et les exceptions [1], le mode utilisateur et superviseur et la gestion de la mémoire. Notons que la gestion de la mémoire sera différente de celle utilisée dans l'architecture *MIPS*, car nous n'utiliserons pas la pagination.

3.1 Modèle de programmation

3.1.1 Espace d'adressage

Dans l'implémentation multi-cycles, les instructions et les données seront dans la même mémoire. Nous aurons donc accès à 2^{32} octets.

3.1.2 Registres

En plus des 32 registres du processeur, le coprocesseur 0 ajoute 32 nouveaux registres dont certains ont un usage particulier.

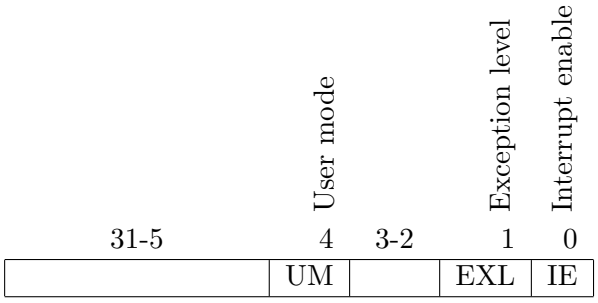
- Le registre numéro 0 contient l'adresse de base qui sera ajoutée aux adresses en mode utilisateur. Ceci va nous permettre d'isoler chaque processus et de faire en sorte que chaque programme puisse commencer à l'adresse `0x00000000` (virtuellement). Une adresse `A` générée par l'unité centrale en mode utilisateur, sera donc transformée en : `valeur du registre 0`

1. Dans la terminologie *MIPS*, on parle d'interruption pour un évènement extérieur au processeur (lié à un périphérique par exemple) et d'exception pour un évènement interne au processeur (division par zéro par exemple).

+ `A`, avant d'être envoyée sur le bus d'adresse. En mode superviseur, la valeur de ce registre est ignorée et les adresses générées par l'unité centrale sont directement mises sur le bus d'adresse. Rappelons que cette gestion de la mémoire n'est pas celle utilisée dans l'architecture *MIPS*.

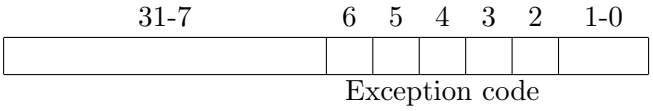
- Le registre numéro 1 contient l'adresse maximale autorisée pour un processus donné en mode utilisateur. En mode utilisateur, l'adresse générée par l'unité centrale est d'abord traduite (en ajoutant le contenu du registre 0) et ensuite comparée à la valeur du registre 1. Si l'adresse traduite est strictement supérieure à celle-ci, une exception est générée. En mode superviseur, ce registre est ignoré.

- Le registre numéro 12, appelé *Status register* (pour registre d'état), permet de savoir si on se trouve en mode utilisateur ou superviseur et si les exceptions ou interruptions sont masquées ou non. Une partie de ce registre (celle qui nous sera utile) est décrite ci-dessous :



  La signification des différents champs est la suivante :
  - ⋆ le bit 4 (`UM`) est à `0` si le processeur est en mode superviseur et il est à `1` si le processeur est en mode utilisateur ;
  - ⋆ le bit 1 (`EXL`) est mis à `1` (automatiquement) lors de la prise en compte d'une exception ou interruption. Cela a pour effet de mettre le processeur en mode superviseur (sans modifier la valeur de `UM`), et de ne pas prendre en compte de nouvelles exceptions ou interruptions ;
  - ⋆ le bit 0 (`IE`) est à `0` si les demandes d'interruptions (arrivant sur la broche `Int` du processeur) sont masquées. Elle est à `1` sinon. Si `EXL` est à `1`, les interruptions sont masquées même si `IE` est à `1`.

- Le registre numéro 13, appelé *Cause register* (pour registre de cause), permet de connaître la cause d'une exception ou interruption. Une partie de ce registre (celle qui nous sera utile) est décrite ci-dessous :



  Les bits 6 à 2 contiennent un nombre indiquant le type d'exception ou interruption qui vient de se produire. Les différentes valeurs qui nous intéressent sont décrites dans la table 3.1.

- Le registre numéro 14, appelé `EPC`, permet de sauvegarder la valeur du compteur ordinal (`PC`) lors de la prise en compte d'une exception ou interruption.

Valeur du champ `Exception code`	Nom	Description
0	`Int`	interruption provenant d'un contrôleur de périphérique
4	`AdEL`	l'adresse est en dehors de la zone allouée pour un *load* ou le chargement d'une instruction (mode utilisateur)
5	`AdES`	l'adresse est en dehors de la zone allouée pour un *store* (mode utilisateur)
8	`Sys`	on vient d'exécuter l'instruction `syscall`
10	`RI`	l'instruction n'est pas valide (instruction non reconnue) ou n'est pas autorisée (en mode utilisateur)
12	`Ov`	un débordement s'est produit pour une instruction arithmétique

TABLE 3.1 – Les différentes valeurs du champ *Exception code*

Dans notre implémentation, tous les autres registres du coprocesseur 0 n'auront aucune signification particulière.

3.1.3 Gestion d'une exception ou d'une interruption

La gestion des demandes d'exceptions ou d'interruptions sont similaires sauf que la demande d'interruption (provenant d'un contrôleur de périphérique) peut être masquée (bit `IE` du registre d'état à `0`). Si le bit `EXL` du registre d'état est à `1`, aucune exception ou interruption ne peut être prise en compte (ceci veut dire qu'une exception ou une interruption est déjà en train d'être traitée). Lorsqu'une exception ou une interruption est prise en compte :

- le bit `EXL` du registre d'état est mis à `1`. Ceci a pour effet de masquer les autres exceptions et demandes d'interruptions qui pourraient avoir lieu, et de passer en mode superviseur ;
- le compteur ordinal (`PC`) est sauvegardé dans le registre `EPC` et on charge le compteur ordinal avec l'adresse du traitant d'exceptions et d'interruptions. Nous supposons ici que cette adresse est `0x00000004` ;
- le traitant va sauvegarder les registres qu'il va utiliser et analyser ce qui a causé son appel grâce à la partie `Exception code` du registre de cause (*Cause register*). Le traitant fait alors les opérations nécessaires pour gérer l'exception ou l'interruption puis restaure les registres et enfin exécute l'instruction `eret` (qui a pour effet de remettre le compteur ordinal à la valeur sauvegardée dans `EPC` et de mettre le bit `EXL` du registre d'état à `0`).

3.1.4 Jeu d'instructions

Nous allons ajouter quatre nouvelles instructions qui vont permettre de réaliser des appels système dans le cadre des systèmes d'exploitation (`syscall`), de sortir d'un traitant d'interruption (`eret`), de déplacer des valeurs du banc de registres du processeur vers celui du coprocesseur 0 (`mtc0`) et de déplacer des valeurs du banc de registres du coprocesseur 0 vers celui du processeur (`mfc0`).

- `syscall`
  Cette instruction permet de générer une exception afin de passer en mode superviseur et de passer la main au système d'exploitation. Le compteur ordinal est sauvegardé dans le registre `EPC`, la cause de l'exception `Sys` est placée dans le registre de cause (*Cause register*) (voir la table 3.1) et le champ `EXL` du registre d'état (*Status register*) est mis à `1`. Le

compteur ordinal est alors chargé avec l'adresse du traitant d'exceptions ou d'interruptions (adresse `0x00000004`). L'instruction `syscall` est codée en langage machine :

0	001100
26 bits	6 bits

- `eret`
  Cette instruction permet de sortir d'un traitant d'exceptions ou d'interruptions. Elle a pour effet de remettre simultanément le compteur ordinal à la valeur sauvegardée dans le registre `EPC` et le bit `EXL` du registre d'état à `0`. L'instruction `eret` est codée en langage machine :

100000	1	0	011000
6 bits	1 bits	19 bits	6 bits

- `mtc0 rd, rt`
  Cette instruction permet de copier la valeur du registre numéro `rt` du banc de registres du processeur vers le registre numéro `rd` du banc de registres du coprocesseur 0. Cette instruction ne peut être utilisée qu'en mode superviseur sinon, l'exception de code `RI` est lancée. L'instruction `mtc0 rd, rt` est codée en langage machine :

010000	00100	rt	rd	0
6 bits	5 bits	5 bits	5 bits	11 bits

- `mfc0 rd, rt`
  Cette instruction permet de copier la valeur du registre numéro `rt` du banc de registres du coprocesseur 0 vers le registre numéro `rd` du banc de registres du processeur. Cette instruction ne peut être utilisée qu'en mode superviseur sinon, l'exception de code `RI` est lancée. L'instruction `mfc0 rd, rt` est codée en langage machine :

010000	0	rt	rd	0
6 bits	5 bits	5 bits	5 bits	11 bits

3.2 Architecture

3.2.1 Unité de traitement

Unité de traitement sans le coprocesseur 0

Le schéma de la figure 3.1 [2] présente une vision de haut niveau de l'unité de traitement de l'architecture multi-cycles sans prendre pour le moment en considération la gestion du coprocesseur 0. Par rapport au chapitre précédent, on peut remarquer les différences suivantes :

- une seule mémoire est utilisée pour les instructions et les données ;
- il y a seulement une unité arithmétique et logique, au lieu d'une `ALU` et deux additionneurs ;
- un ou plusieurs registres sont ajoutés après chaque bloc principal pour mémoriser la sortie de ce bloc qui sera utilisé lors d'un cycle suivant.

Une instruction étant exécutée sur plusieurs cycles, toutes les données étant utilisées lors des cycles suivants de la même instruction doivent être sauvegardées. Ainsi, la position des re-

2. Notons que sur ce schéma, une flèche touchant un fils sans cercle plein à l'intersection indique qu'il faudra mettre un multiplexeur à cet endroit.

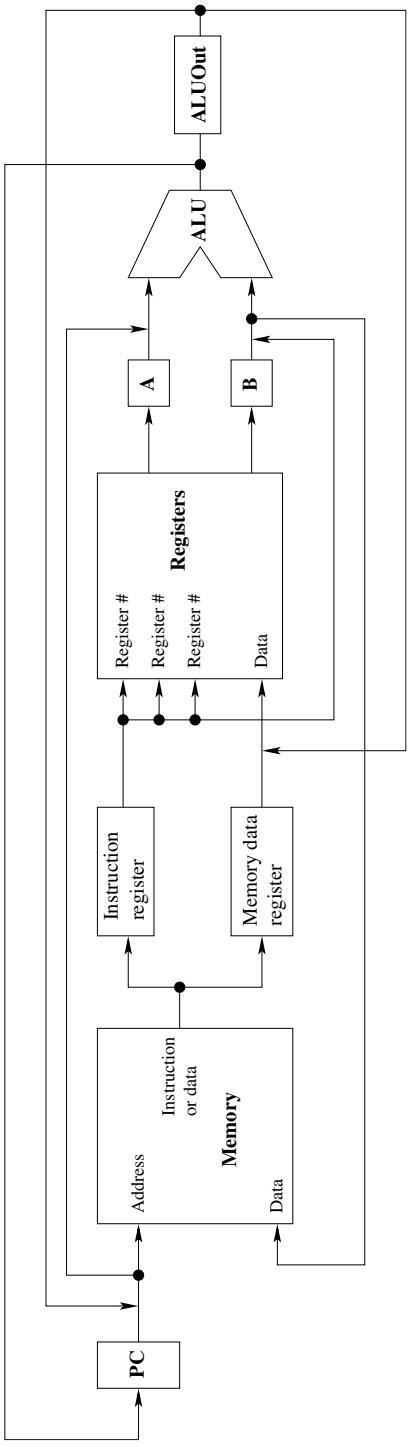


FIGURE 3.1 – Vision de haut niveau de l'unité de traitement de l'architecture multi-cycles

gistres à ajouter est déterminée par deux facteurs : quel bloc de l'unité de traitement tiendra en un cycle et quelles données sont nécessaires dans les cycles suivants d'une même instruction. Dans l'implémentation multi-cycles, nous assumons que le cycle d'horloge est assez grand pour les opérations suivantes : un accès mémoire, un accès au banc de registre (deux lectures ou une écriture), une opération de l'unité arithmétique et logique ou une traduction d'adresse (que l'on verra par la suite). Il faut donc sauvegarder les résultats produits par chacun de ces blocs dans un registre temporaire pour pouvoir les utiliser dans des cycles ultérieurs (pour une même instruction). Les registres temporaires suivants sont donc ajoutés :

- les registres `Instruction register` et `Memory data register` sont ajoutés pour sauvegarder le contenu de la mémoire lors d'une lecture d'une instruction ou d'une donnée respectivement. On verra que ces deux valeurs peuvent être utilisées durant le même cycle ;
- les registres `A` et `B` sont utilisés pour mémoriser les opérandes de l'unité arithmétique et logique lues dans le banc de registres ;
- le registre `ALUOut` sauvegarde la sortie de l'unité arithmétique et logique.

Tous les registres que nous venons d'énoncer excepté le registre `Instruction register`, conservent la même donnée simplement pendant un cycle d'horloge et n'ont donc pas besoin d'une brôche de contrôle indiquant s'il faut ou non charger une donnée au front montant d'horloge (on chargera toujours au front montant). Le registre `Instruction register` conserve l'instruction jusqu'à la fin de l'exécution de l'instruction et nécessite donc une broche de chargement.

Parce que plusieurs blocs de l'unité de traitement sont partagés pour différents usages, il faudra ajouter des multiplexeurs et étendre ceux déjà existants. Par exemple, comme nous remplaçons les deux additionneurs et l'`ALU` de l'implémentation sur un seul cycle par une seule `ALU`, cela veut dire que la nouvelle unité arithmétique et logique devra avoir en entrée toutes celles qui allaient sur les trois différents composants.

Exercice 10 : réaliser l'unité de traitement.

Ajout du coprocesseur 0

Nous allons maintenant intégrer le coprocesseur 0 afin de gérer les modes utilisateur et superviseur, la gestion de la mémoire et les exceptions et les interruptions. Nous disposons du composant représenté dans la figure 3.2. La decription de ce composant est la suivante :

- La sortie `Read data` a pour valeur le contenu du registre dont le numéro est donné sur la broche `Read register` ;
- Le contenu du registre numéro 12 (*Status register*) est sur la sortie `Status` ;
- Le contenu du registre numéro 0 (*Base register*) est sur la sortie `Base` ;
- Le contenu du registre numéro 1 (*Limit register*) est sur la sortie `Limit` ;
- La valeur présente sur l'entrée `Write data` sera copiée dans le registre de numéro `Write register` au front montant d'horloge si l'entrée `Write` est à `1` ;
- Le champ `EXL` du registre numéro 12 (*Status register*) est mis à `1` au top d'horloge si l'entrée `Set EXL` est à `1` ;
- Le champ `EXL` du registre numéro 12 est mis à `0` au top d'horloge si l'entrée `Reset EXL` est à `1` ;
- Si la broche `Init` est à `1`, de manière asynchrone, le *Status register* est mis à `0` (ce qui a pour effet de mettre les bits `UM`, `EXL` et `IE` à `0`).

Nous disposons aussi du composant représenté dans la figure 3.3 qui permet de faire la traduction d'adresse lorsque l'on est en mode utilisateur. Son fonctionnement est décrit ci-dessous :

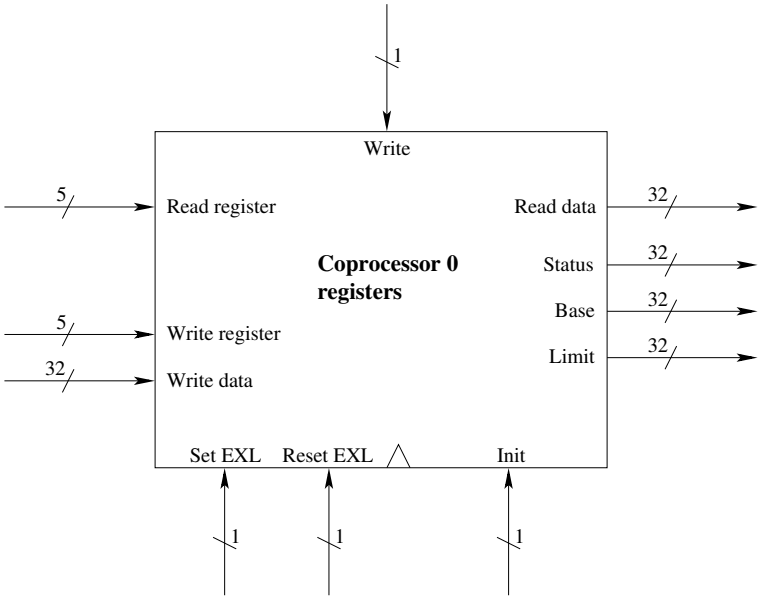


FIGURE 3.2 – Le banc de registres du coprocesseur 0

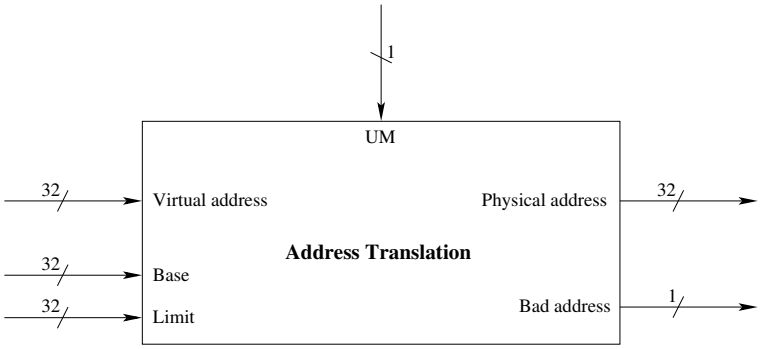


FIGURE 3.3 – L'unité de traduction d'adresses du coprocesseur 0

- Lorsque l'entrée `UM` est à `0`, cela veut dire que l'on est en mode superviseur et la sortie `Physical address` correspond simplement à la valeur présente sur l'entrée `Virtual address`. Lorsque `UM` est à `0`, la sortie `Bad address` est à `0` ;
- Lorsque l'entrée `UM` est à `1`, cela veut dire que l'on est en mode utilisateur. La sortie `Physical address` est alors la somme des entrées `Virtual address` et `Base`. Lorsque `UM` est à `1`, si la sortie `Physical address` est strictement supérieur à l'entrée `Limit` alors `Bad address` est à `1`.

Exercice 11 : compléter l'unité de traitement afin d'y intégrer le coprocesseur 0.

3.2.2 Unité de contrôle microprogrammée

Nous allons implémenter le contrôle de cette architecture multi-cycles à l'aide d'un micro-programme. La table 3.2 résume les 5 étapes nécessaires pour exécuter une instruction. Nous ne décrivons pas dans cette table la gestion des interruptions et exceptions et la traduction d'adresses. Comme nous l'avons vu, toutes les instructions n'auront pas besoin de ces 5 étapes (une instruction s'exécutera entre 3 et 5 étapes). La quatrième opération de l'étape "Chargement des registres" permet de précalculer l'adresse du compteur ordinal au cas où un saut conditionnel aurait lieu à l'étape suivante. Cela permet de gagner un cycle.
La table 3.2 représente l'automate de contrôle (sans la gestion des interruptions et exceptions et de la traduction d'adresses). Ainsi, l'étape "Chargement de l'instruction" sera représenté par 1 état, l'étape "Chargement des registres" par 1 état, l'étape "Opération, calcul de l'adresse de saut, `eret`, `syscall`" par 7 états, l'étape "Accès mémoire ou sauvegarde de l'opération" par 2 états et l'étape "Fin de la lecture mémoire" par 1 état. Les transitions se feront entre les états de deux étapes successives inconditionnellement ou selon l'opcode de l'instruction.

Exercice 12 : représenter l'automate de contrôle sans tenir compte de la gestion des interruptions et des exceptions et de la traduction d'adresses.

La vision de haut niveau de l'unité de contrôle est donnée à la figure 3.4. La partie `Stockage des micro-instructions` est une *ROM* qui contient le micro-programme (en langage machine). Une micro-instruction [3] en langage machine va comprendre une partie qui permettra de définir les valeurs des différentes lignes de contrôle de l'unité de traitement (`Lignes de contrôle` sur la figure), une partie qui va nous permettre de savoir quelle prochaine micro-instruction doit être exécutée s'il y a des exceptions ou des interruptions (`Exception` sur la figure) et une dernière partie pour connaître quelle micro-instruction doit être exécutée ensuite s'il n'y a pas d'exceptions et d'interruptions (`Séquencement` sur la figure). La partie `Compteur` est l'équivalent du compteur ordinal (`PC`) de l'unité de traitement et indique la prochaine micro-instruction à exécuter. La partie `Sélection de l'adresse` va permettre, en fonction de la valeur des champs `Exception` et `Séquencement` de la micro-instruction et des valeurs des broches `Compteur + 1`, `Opcode de l'instruction`, `Int`, `UM`, `EXL`, `IE`, `Bad@` et `Overflow`, de calculer l'adresse de la prochaine micro-instruction.

Il n'est pas facile d'élaborer le micro-programme si nous travaillons directement avec des micro-instruction en langage machine. En s'inspirant de ce que l'on fait en programmation, nous allons définir un langage assembleur pour les micro-instructions. Ce langage nous permettra d'écrire

3. Nous mettons le préfixe "micro" pour bien faire la différence entre une instruction *MIPS* et une instruction de l'unité de contrôle.

Étape	Action pour les instructions arithmétiques et logiques	Action pour les instructions de référence à la mémoire	Action pour les transferts entre les bancs de registres	Action pour les sauts conditionnels	Action pour les sauts inconditionnels	Action pour `eret`	Action pour `syscall`
Chargement de l'instruction	`IR <- Memory[PC]` `PC <- PC + 4`						

Chargement des registres	`A <- Reg[IR[25-21]]`
	`B <- Reg[IR[20-16]]`
	`C <- RegC0[IR[20-16]]`
	`ALUOut <- PC + (sign-extend(IR[15-0]) ≪ 2)`						

Opération, calcul de l'adresse de saut, `eret`, `syscall`	`ALUOut <- A op B`	`ALUOut <- A + sign-extend(IR[15-0])`	`mfc0 : Reg[IR[15-11]] <- C`
			ou
			`mtc0 : RegC0[IR[15-11]] <- B`
				`if (A == B)`
				`PC <- ALUOut`
					`PC <- (PC[31-28], IR[25-0],00)`	`PC <- EPC`	`PC <- 0x00000004`

Accès mémoire ou sauvegarde de l'opération	`Reg[IR[15-11]] <- ALUOut`	`lw : MDR <- Memory[ALUOut]`
		ou
		`sw : Memory[ALUOut] <- B`					

Fin de la lecture mémoire		`lw : Reg[IR[20-16]] <- MDR`					

TABLE 3.2 – Les cinq étapes permettant d'exécuter les instructions

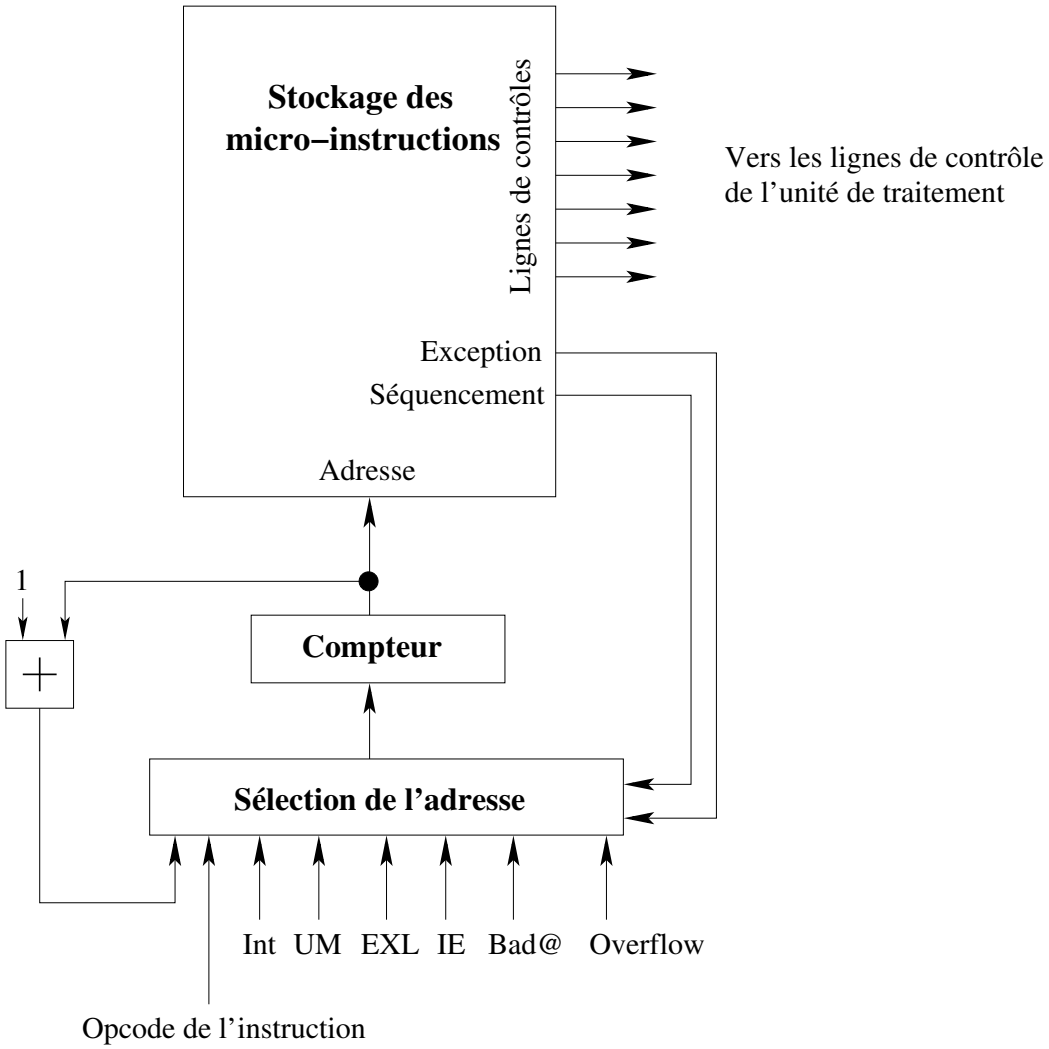


FIGURE 3.4 – Vision de haut niveau de l'unité de contrôle microprogrammée

beaucoup plus facilement le micro-programme qui sera ensuite traduit en langage machine. Il nous reste donc à définir les différents champs d'une micro-instruction en assembleur. Notre but est de définir un format qui soit facile à comprendre et donc à utiliser. On voudrait aussi que le format que l'on choisisse rende difficile ou impossible l'écriture de micro-instruction en assembleur qui une fois traduite serait inconsistantes (par exemple la même brôche de contrôle devrait prendre deux valeurs différentes). Pour ne pas avoir de traduction qui génère des micro-instructions inconsistantes, il suffit qu'un même signal de contrôle ne puisse être attribué à plus d'un champ de la micro-instruction en assembleur. La table 3.3 donne la représentation d'une micro-instruction en assembleur. On y voit les 9 différents champs ainsi que les valeurs qu'ils peuvent prendre. Notons qu'un champ peut être laissé vide, cela indique que ce champ n'intervient pas dans la micro-instruction présente et ceci se concrétisera par la mise à zéro de tous les signaux de contrôle représentés par ce champ. Notons que des valeurs de champ existent malgrés le fait que les broches qu'elles contrôlent doivent être mises à zéro (par exemple la valeur `Add` du champ `ALU control`). Ces valeurs de champ permettront de rende plus lisible le micro-programme en assembleur.
La partie "Signaux de contrôle" donne la valeur des différents signaux de contrôle de l'unité de traitement sauf dans le cas des champs `Exception` et `Sequencing` où les signaux de contrôle vont permettre d'indiquer quelle sera la prochaine ligne du micro-programme à exécuter.

Exercice 13 : compléter la partie "Signaux de contrôle" des 7 premiers champs de la table 3.3.

Il nous reste à détailler les deux derniers champs de la table 3.3 :

- `Exception` : ce champ comporte 5 valeurs différentes car si aucune exception n'est à traiter, on laissera ce champ vide et on aura donc `Ex = 000`. Il devra donc être codé sur 3 bits en langage machine :
  - ⋆ `Int else bad fetch` : si la broche `Int` (indiquant qu'il y a une demande d'interruption) est à `1` et que l'on peut prendre en compte cette demande (`EXL` est à `0` et `IE` est à `1`), la prochaine adresse dans le micro-programme sera celle permettant de gérer les interruptions. Sinon, si la broche `Bad@` de l'unité de traitement est à `1`, la prochaine adresse dans le micro-programme sera celle permettant de gérer l'exception `AdEL`. Pour cette valeur, `Exception = 001` ;
  - ⋆ `Bad @` : si la broche `Bad@` de l'unité de traitement est à `1`, selon que l'on effectue un chargement ou une sauvegarde d'une donnée, la prochaine adresse dans le micro-programme sera celle permettant de gérer les exceptions `AdEL` et `AdES` respectivement. Pour cette valeur, `Exception = 010` ;
  - ⋆ `Bad privilege` : si l'instruction est une instruction nécessitant les droits superviseur et que l'on est en mode utilisateur, la prochaine adresse dans le micro-programme sera celle permettant de gérer l'exception `RI`. Pour cette valeur, `Exception = 011` ;
  - ⋆ `Overflow` : si l'instruction est une instruction arithmétique ou logique et qu'un débordement est détecté, la prochaine adresse dans le micro-programme sera celle permettant de gérer l'exception `Ov`. Pour cette valeur, `Exception = 100`.

- `Sequencing` : ce champ comporte 4 valeurs différentes et devra donc être codé sur 2 bits en langage machine. La partie `Sequencing` sera réalisée seulement si la partie `Exception` n'a pas déjà fournie l'adresse de la prochaine micro-instruction. La signification des différentes valeurs est la suivante :
  - ⋆ `Fetch` : la prochaine micro-instruction est la première du micro-programme (La valeur du `Compteur` est remise à `0`). Pour cette valeur, `Sequencing = 00` ;
  - ⋆ `Dispatch I` : cette valeur permet de gérer le passage, selon l'opcode de l'instruction, de l'étape "Chargement des registres" à l'étape "Opération, calcul de l'adresse de saut,

Nom du champ	Valeurs possibles	Signaux de contrôle
ALU control	Add	L'`ALU` effectue une addition : `ALUOp = 00`
	Sub	L'`ALU` effectue une soustraction : `ALUOp = 01`
	Func code	Utilise le champ `funct` pour déterminer ce que calcul l'`ALU` : `ALUOp = 10`
SRC1	PC	Le `PC` est la première opérande de l'`ALU`
	A	Le registre `A` est la première opérande de l'`ALU`
SRC2	B	Le registre `B` est la deuxième opérande de l'`ALU`
	4	La valeur 4 est la deuxième opérande de l'`ALU`
	Extend	La sortie du composant d'extension de signe est la deuxième opérande de l'`ALU`
	Extshft	La sortie du composant de décalage à gauche est la deuxième opérande de l'`ALU`
Register control	Read	Lit deux registres en utilisant les champs `rs` et `rt` du registre d'instruction en mettant les valeurs dans les registres `A` et `B`
	Write ALU	Met dans le registre `rd` du banc de registres la valeur de `ALUOut`
	Write MDR	Met dans le registre `rt` du banc de registres la valeur de `MDR`
	Write C0	Met dans le registre `rd` du banc de registres la valeur de `C`
Memory	Translate PC	Traduit l'adresse provenant du `PC`
	Translate ALU	Traduit l'adresse provenant de `ALUOut`
	Read PC	Lit une instruction de la mémoire (l'adresse provient de la traduction du `PC`
	Read ALU	Lit une donnée de la mémoire (l'adresse provient de la traduction de `ALUOut`)
	Write ALU	Écrit la mémoire avec la donnée provenant du registre `B` (l'adresse provient de la traduction de `ALUOut`)
PCWrite control	ALU	Écrit la sortie de l'`ALU` dans le `PC`
	ALUOut-cond	Si la broche `Zero` de l'`ALU` est active, charger le `PC` avec le contenu de `ALUOut`
	Jump @	Charger le `PC` avec l'adresse de saut provenant de l'instruction
	Handler	Charger le `PC` avec l'adresse du traitant d'exceptions et d'interruptions
	EPC	Charger le `PC` avec la valeur de celui-ci avant la prise en compte de l'exception ou de l'interruption
Coprocessor 0	Read reg	Met la valeur du registre `rt` du coprocessor 0 dans le registre `C`
	Read EPC	Met en sortie du banc de registres la valeur du registre 14 et met à `0` le champ `EXL` du registre 12
	Write reg	Met dans le registre `rd` du coprocessor 0 la valeur contenue dans le registre `B`
	Write PC	Met dans le registre 14 du banc de registres la valeur du `PC` et met `EXL` à `1`
	Write cause Int	Écrit dans le registre 13 le code d'exception `Int`
	Write cause AdEL	Écrit dans le registre 13 le code d'exception `AdEL`
	Write cause AdES	Écrit dans le registre 13 le code d'exception `AdES`
	Write cause Sys	Écrit dans le registre 13 le code d'exception `Sys`
	Write cause RI	Écrit dans le registre 13 le code d'exception `RI`
	Write cause Ov	Écrit dans le registre 13 le code d'exception `Ov`
Exception	Int else bad fetch	Va à la ligne du micro-programme gérant les interruptions ou va à la ligne gérant les erreurs de chargement d'une instruction
	Bad @	Va à la ligne du micro-programme gérant les erreurs d'adressage
	Bad privilege	Va à la ligne du micro-programme gérant les erreurs de droit
	Overflow	Va à la ligne du micro-programme gérant les erreurs de débordement
Sequencing	Fetch	Va à la première micro-instruction du micro-programme
	Dispatch I	Va à la ligne du micro-programme indiquée par le composant `Dispatch I`
	Dispatch II	Va à la ligne du micro-programme indiquée par le composant `Dispatch II`
	Next	Passe à la prochaine micro-instruction du micro-programme

TABLE 3.3 – Définition des différents champs d'une micro-instruction en assembleur et de leurs valeurs. La partie contrôle (qui permettra d'écrire la micro-instruction en langage machine) est à compléter

`eret`, `syscall`" de la table 3.2. Pour cette valeur, `Sequencing = 01` ;
- ⋆ `Dispatch II` : pour une instruction de référence à la mémoire (`lw` ou `sw`), cette valeur permet de gérer le passage de l'étape "Opération, calcul de l'adresse de saut, `eret`, `syscall`" à l'étape "Accès mémoire ou sauvegarde de l'opération" de la table 3.2. Pour cette valeur, `Sequencing = 10` ;
- ⋆ `Next` : on passe à la micro-instruction suivante. La valeur du `Compteur` (figure 3.4) est donc incrémentée. Pour cette valeur, `Sequencing = 11`.

Exercice 14 : on peut désormais écrire le micro-programme en assembleur. Compléter le micro-programme de la table 3.4.

Exercice 15 : réaliser le composant `Sélection de l'adresse` de la figure 3.4.

Il ne reste plus qu'à traduire le micro-programme assembleur de la table 3.4 complétée, en micro-programme en langage machine. La traduction est immédiate en utilisant la table 3.3 et les commentaires décrivant celle-ci.

3.3 Perspectives

Afin d'avoir une architecture un peu moins sommaire, nous allons ajouter quelques instructions et la gestion du *timer*.

3.3.1 Instructions supplémentaires

Jeu d'instructions

- L'instruction `lw` permet de charger un mot de 32 bits à partir de la mémoire. Nous allons ajouter les instructions `lh` et `lb` qui permettent respectivement de charger un mot de 16 bits et un octet de la mémoire.
  L'instruction `lh rt, v(rs)` est codée en langage machine :

100001	rs	rt	v
6 bits	5 bits	5 bits	16 bits

  Les 16 bits chargées à partir de la mémoire sont étendues selon le signe sur les 32 bits du registre destination.

  L'instruction `lb rt, v(rs)` est codée en langage machine :

100000	rs	rt	v
6 bits	5 bits	5 bits	16 bits

  Là encore, les 8 bits chargées à partir de la mémoire sont étendues selon le signe sur les 32 bits du registre destination.

- L'instruction `sw` permet de sauvegarder un mot de 32 bits dans la mémoire. Nous allons ajouter les instructions `sh` et `sb` qui permettent respectivement de sauvegarder un mot de 16 bits et un octet dans la mémoire.
  L'instruction `sh rt, v(rs)` est codée en langage machine :

On suppose que les lignes de contrôle associées à un champ dans la valeur n'est pas spécifiée prennent la valeur zéro.

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Coprocessor 0	Exception	Sequencing
0 : FETCH					Translate PC			Int else bad fetch	Next
1 :	Add	PC	4		Read PC	ALU			Next
2 :									
3 :									
4 :									
5 :									
6 :									
7 :									
8 :									
9 :									
10 :									
11 :									
12 :									
13 :									
14 :									
15 :									
16 :									
17 :									
18 :									
19 :									
20 :									
21 :									
22 :									
23 :									
24 :									
25 :									
26 :									

TABLE 3.4 – Micro-programme à compléter

101001	rs	rt	v
6 bits	5 bits	5 bits	16 bits

L'instruction `sb rt, v(rs)` est codée en langage machine :

101000	rs	rt	v
6 bits	5 bits	5 bits	16 bits

- `addiu $s1, $s2, v`
  Cette instruction place dans le registre `$s1` la somme du registre `$s2` et de la valeur v, qui est sur 16 bits, en complétant celle-ci par 16 bits de poids forts à `0` (afin d'avoir une valeur sur 32 bits).
  L'instruction `addiu rt, rs, v` est codée en langage machine :

001001	rs	rt	v
6 bits	5 bits	5 bits	16 bits

- `sll $s1, $s2, v`
  Cette instruction permet de décaler la valeur contenue dans le registre `$s2` de `v` bits vers la gauche. Des `0` sont insérés à droite pour combler les bits manquants. Le résultat est placé dans le registre `$s1`.
  L'instruction `sll rd, rt, v` est codée en langage machine :

000000	00000	rt	rd	v	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `srl $s1, $s2, v`
  Cette instruction permet de décaler la valeur contenue dans le registre `$s2` de `v` bits vers la droite. Des `0` sont insérés à gauche pour combler les bits manquants. Le résultat est placé dans le registre `$s1`.
  L'instruction `srl rd, rt, v` est codée en langage machine :

000000	00000	rt	rd	v	000010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `nor $s1, $s2, $s3`
  Cette instruction place dans le registre `$s1` le *non ou logique bit à bit* des registres `$s2` et `$s3`. L'instruction `nor rd, rs, rt` est codée en langage machine :

000000	rs	rt	rd	0	100111
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `lui $s1, v`
  Où `v` est une valeur sur 16 bits. Cette instruction place dans le registre `$s1` la valeur `v` décalée de 16 bits vers la gauche (le registre `$s1` contient donc la valeur `v` dans ses 16 bits de poids forts et des `0` dans ses 16 bits de poids faibles). Cette instruction va nous permettre de charger des constantes dans des registres. En effet, pour charger la constante `0x12345678` dans le registre `$s1` par exemple, il suffira d'exécuter :

```
lui   $s1, 0x1234
addiu $s1, $s1, 0x5678
```

  L'instruction `lui rt, v` est codée en langage machine :

001111	00000	rt	v
6 bits	5 bits	5 bits	16 bits

- `jr $s1`
  Cette instruction permet de mettre la valeur du compteur ordinal égale à celle contenue dans le registre `$s1` (`PC <- $s1`).
  L'instruction `jr rs` est codée en langage machine :

000000	rs	0000000000000	00000	001000
6 bits	5 bits	10 bits	5 bits	6 bits

Exemple de programme

La multiplication n'est pas prise en compte dans la version réduite de l'architecture *MIPS* que nous avons présentée. Néanmoins, de manière logicielle, nous allons pouvoir mettre en œuvre cette opération. Nous donnons ci-dessous le code en *C* de la multiplication de deux entiers positifs (dite multiplication russe). On peut adapter ce qui suit pour prendre en compte les entiers négatifs.

```
int mul(int a, int b) {
  int res = 0;
  while(b != 0) {
    if ((b % 2) != 0) {
      res = res + a;
      b = b - 1;
    } else {
      a = a * 2;
      b = b / 2;
    }
  }
  return res;
}
```

Nous allons maintenant traduire ce code en supposant que les entiers sont codés sur 16 bits. L'opérande `a` se trouve dans le registre `$a0`, l'opérande `b` dans le registre `$a1`, l'adresse de retour de la fonction dans le registre `$ra` et la valeur retournée par la fonction dans le registre `$v0`. Le registre `$t0` représente la variable `res`.

```
mul:
       add   $t0, $0, $0       //res = 0;
       add   $t1, $0, $0       //$t1 == 0
       addiu $t1, $t1, 1       //$t1 == 1
while:
       beq   $a1, $0, endWhile //si b == 0 on sort du while
       and   $t2, $a1, $t1     //$t2 est égal à 1 si b est impair et à 0 sinon
       beq   $t2, $0, else
       add   $t0, $t0, $a1     //res = res + b
       sub   $a1, $a1, $t1     //b = b - 1
       jump  while
else:
       sll   $a0, $a0, 1       //a = a * 2
```

```
       srl   $a1, $a1, 1       //b = b / 2
       jump  while
endWhile:
       add   $v0, $t0, $0
       jr    $ra
```

3.3.2 Gestion du *timer*

Le coprocesseur 0 contient un *timer* intégré. Trois registres du coprocesseur 0 sont liés à la gestion du *timer*.

- Le registre numéro 9, appelé *Count*, est incrémenté de 1 à une fréquence donnée. Il est mis à `0` lors de l'initialisation de la machine (`reset` mis à `1`).

- Le registre numéro 11, appelé *Compare* permet de déclencher une demande d'interruption lorsque la valeur contenue dans ce registre est égale à la valeur du registre *Count*. Écrire une valeur dans ce registre à comme effet secondaire d'effacer l'éventuelle demande d'interruption.

- Le bit 15 (bit `IP7`) du registre numéro 13 (*Cause register*) est à `1` si une demande d'interruption liée au *timer* a été émise. Ce bit est accessible uniquement en lecture.

Bibliographie

[1] D.A. Patterson and J.L. Hennessy. Computer organization and design. the hardware/software interface (third edition revised). *Morgan Kaufmann*, 2007.