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.

1 Modèle de programmation

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.

1.2 Registres

Nous présentons dans la table 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.

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

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

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 1 – Les 32 registres de la machine *MIPS*

- `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.

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:
```

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 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.

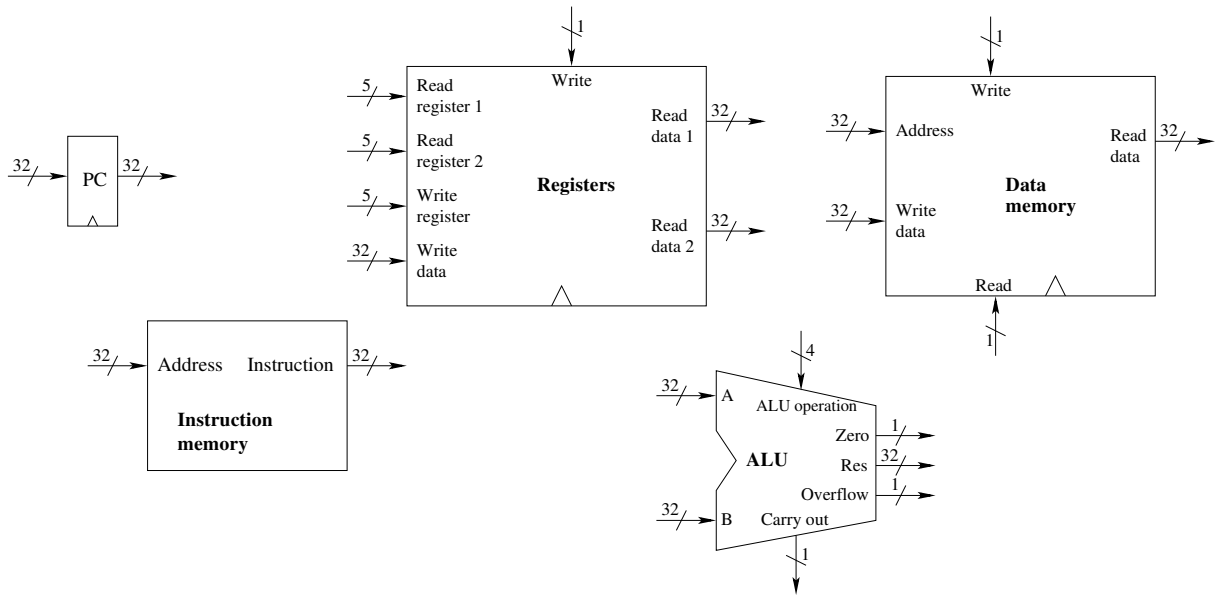

FIGURE 1 – Blocs principaux de l'unité de traitement

2.1 Unité de traitement

La figure 1 présente les blocs principaux de l'unité de traitement. Nous allons détailler chacun de ces éléments.

2.1.1 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.

2.1.2 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. On y lira ici toujours des valeurs sur 32 bits.

2.1.3 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`.

2.1.4 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.

2.1.5 Unité arithmétique et logique (`ALU`)

`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 – Fonctionnement de l'unité arithmétique et logique

La table 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 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 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 3 – Détection d'un débordement

Exercice 2 : réaliser l'unité arithmétique et logique.

2.1.6 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 Unité de contrôle

2.2.1 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, 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 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.

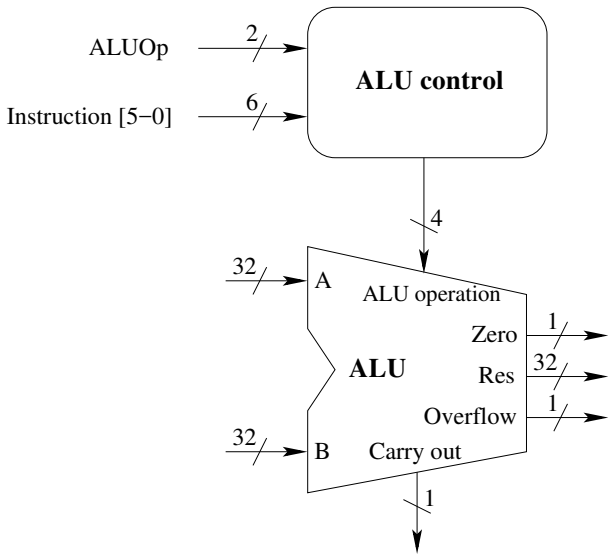

FIGURE 2 – Sous-unité de contrôle permettant de gérer l'unité arithmétique et logique

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

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 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*

2.2.2 Contrôle principal

La table 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.

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).

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