CEG 3556: Conception Avancée des Systèmes Informatiques (Hiver 2019)
Prof. Rami Abielmona
Laboratoire #2: Processeur à Simple Cycle

07 Février, 2019

1 Objectif

L'objectif de ce laboratoire est de concevoir et de construire un processeur de type RISC à cycle simple en VHDL.

À la fin de ce laboratoire, l'étudiant(e) doit être capable de:

- Concevoir, réaliser et de tester un processeur RISC à cycle simple;
- Démontrer une compréhension complète des processeurs RISC et les architectures des ensembles d'instructions.

2 Pré-Lab

Modifiez le chemin de données pour le processeur à cycle simple montré dans le livre et inclus dans ce laboratoire (figure 9), pour soutenir une nouvelle instruction: branche si pas égal (BNE). Montrer les modifications de votre chemin de données, ainsi que toute modifications de signaux de contrôle qui doivent être fait pour soutenir la nouvelle instruction. Présenter et expliquer votre travail au TA.

3 Introduction au Processeurs RISC à Cycle Simple

Dans le monde d'aujourd'hui, nous avons une petite sélection de styles de conception appliqué aux processeurs. Nous avons étudié, en classe, un style spécifique de conception, nommé le **Reduced Instruction Set Computer (RISC)**. Une exemple spécifique qui a été étudiée en détaile est le processeur

MIPS, qui a été développé dans les années 1980. L'ensemble d'instructions de MIPS est effectivement utilisé par NEC, Nintendo, Silicon Graphics, Sony et plusieurs autres fabricants d'ordinateurs.

Nous allons examiner une implementation du processeur MIPS, qui comprend les fonctionalités suivantes:

- Instructions de référence à la mémoire (lw and sw)
- Instructions de logique arithmétique (add, sub, and, or and slt)
- Instructions de contrôle de flow (beq and j)

Notre implementation ne comprendra pas toutes les opérations de nombres entièrs soutenous par MIPS ISA (ex. multiplication et division), ni ne soutenir les opérations de point mobile, qui est aussi soutenous par processeur MIPS. Cependant, la conception sera modulaire et extensible afin de soutenir des opérations supplémentaires si le besoin se pose.

Les instructions qui sont soutenu exécutent deux étapes génériques: d'utiliser le *program counter (PC)* pour fournir l'addresse d'instruction et de lire une (dans le cas d'une instruction lw) ou deux (dans tous les autres cas) registres. Ceux-ci garantir que l'instruction et les opérandes nécessaire sont récupérés dans le processeur et sont prêts à êtres exécutés. L'*Unité Logique d'Arithmétique (ALU)* est ensuite utilisé pour effectuer des calculs. Dans le cas d'une instruction de références à la mémoire, l'ALU est utilisée pour calculer l'addresse, dans la mémoire de données, du mot que nous chargerons à partir d'un registre ou que nous stockerons dans un registre. Dans le cas d'une instruction arithmétique ou logique, l'ALU est utilisée pour effectuer l'opération. Efin, dans le cas d'une instruction de contrôle de flow, l'ALU est utilisée pour effectuer une comparaison, ou le résultat de cette comparaison est utilisée dans le processs de décision de contrôle de flow.

Après l'accès du ALU, chaque instruction diffère en termes de chemin parcouru pour l'achèvement de l'exécution de l'instruction. Dans le cas d'une instruction de référence à la mémoire, un accés de mémoire est fait pour écrire des données (sw) ou pour lire des données (lw). Dans le cas d'une instruction arithméique ou logique, une réécriture est effectuée dans un registre pour sauvegarder le résultat de l'opération du ALU. Finalement, dans le cas d'une instruction de contrôle de flow, l'adresse d'instruction suivante peut avoir besoin d'être changé en fonction du résultat de la comparaison.

Prenons maintenant la vue de haut niveau de notre processeur (référez vous a la figure1). Il est intéressant de noter que le chemin de données est montré a l'horizontale, et aucune logique de contrôle est encore montré. Notez également que les différentes mémoires pour les instructions et les données sont utilisées. Le premier est utilisé pour stocker les instructions pour lecture seule du programme à exécuter, tandis que le deuxième est utilisé pour stocker des données accessible à lecture et écriture utilisés lors de l'exécution du programme.

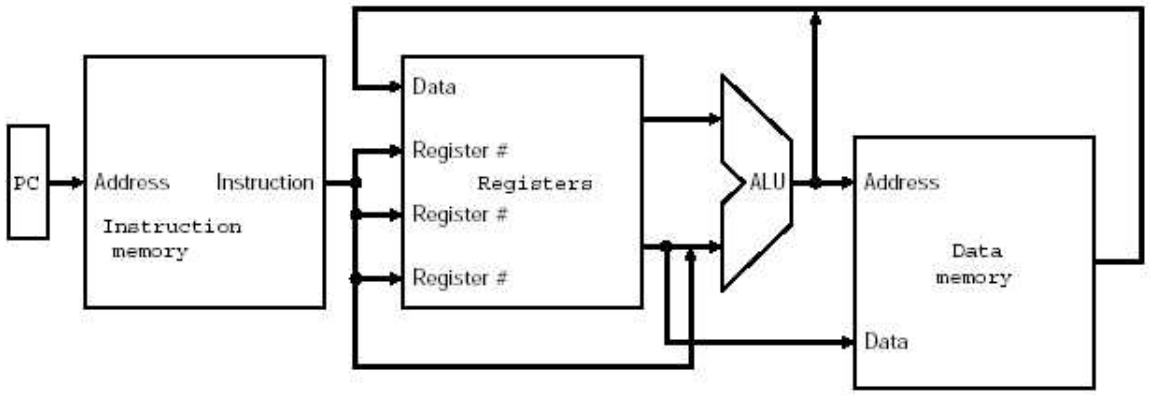


Figure 1: Chemin de données d'un processeur MIPS typique

3.1 Formats d'Instructions MIPS

L'ensemble d'instruction de MIPS définit trois types principaux d'instructions: Type-R, Type-I et Type-J. Les instructions de Type-R sont principalement des instructions arithmétiques et logiques (ex. add, sub, and), avec une instruction typique étant `add $t1, $t2, $t3`, qui dirige le processeur à ajouter les contenus des registres t1 et t2, et stocker le résultat dans le registre t3.

Les instructions de Type-I sont principalement des instructions de référence à la mémoire (ex. la charge et le stockage d'un mot), avec un instruction typique étant `lw $t1, offset_value($t2)`, qui dirige le processeur à ajouter le offset_value au contenu du registre t2. Le résultat va former l'adresse, dans la mémoire des données, du mot qui doit être accédé. Dans le cas d'un stockage (store), le contenu du registre t1 est transférée à ce mot dans la mémoire, alors que dans le cas d'une charge, le contenu du mot dans la mémoire est transférée au registre t1.

Finalement, les instructions de Type-J sont principalement des instruction de contrôle de flow (ex. branche et saut), avec un instruction typique étant `beq $t1, $t2, 25`, qui dirige le processeur à comparer le contenu des registres t1 et t2. Si ils sont égal, un branchement est effectué vers un position dans la mémoire qui est 25 instructions au-dessous du PC actuel. Notez qu'une instruction de branchement peut provoquer un saut au-dessus ou au-dessous le PC actuel, alors qu'une instruction de saut ne peut que provoquer une branche en dessous du PC actuel. Une instruction typique de saut est `j 2500`, qui dirige le processeur à sauter á l'adresse 2500 (dans l'adressage de mot).

Regardez aux figures 2 et 3, pour la représentation interne de tous nos instructions. Notez les différents opcodes pour chaque type d'instruction (0 pour opérations d'ALU, 35 pour opérations de chargement de mot, 43 pour opérations

de stockage de mot, 4 pour opérations de branchement si égal à et 2 pour opérations de saut).

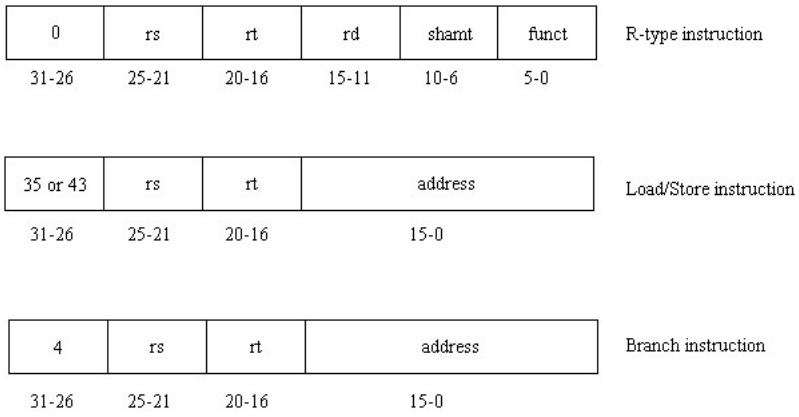


Figure 2: Format d'Instructions MIPS

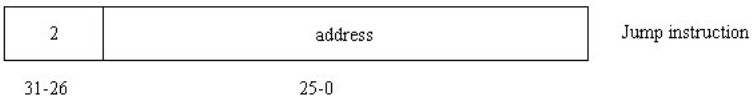


Figure 3: Format d'Instruction de Saut MIPS

Notez que *rs* et *rt* sont des registres de source dans tous les instructions soutenus, tandis que *rd* est un registre de destination, utilisé seulement quand une instruction d'ALU doit écrire le résultat dans un fichier de registre. Le champ *funct* dans les instructions de Type-R définit la fonction suivante du ALU comme suit:

- L'opération d'addition effectuée quand le champ de fonction est 32
- L'opération de soustraction effectuée quand le champ de fonction est 34
- L'opération de ET (AND) effectuée quand le champ de fonction est 36
- L'opération de OU (OR) effectuée quand le champ de fonction est 37
- L'opération de définir si moins de (SET IF LESS THAN) effectuée quand le champ de fonction est 42

Le champ *shamt* dans les instructions de Type-R est utilisé pour les opérations de déplacement qui ne sont pas actuellement soutenus par notre processeur, et donc le champ peut-être ignoré.

Les instructions de chargement et de stockage utilise *rs* comme registre de source, et *rt* comme registre de source ou destination selon que l'opération soit effectué est un stockage our chargement, respectivement. L'instruction de branchement utilise *rs* et *rt* comme registres de source, et le champ d'adresse à 16 bits comme valeur d'offset dans l'opération. Finalement, l'instruction de saut utilise le champ d'adresse à 26 bits dans le calcul de l'adresse d'instruction suivante.

3.2 Fetch and Incrément

La première chose a faire est de garantir que le PC est récupéré de façon appropriée, et incrémenté de quatre, puisque notre mémoire d'instructions est 32 bis de large. Ceci est effectué simultanément comme indiqué dans la figure 4. Notez l'utilisation d'une mémoire d'instructions pour stocker les instructions de notre programme, ainsi qu'un additionneur pour incrémenter le PC par 4. Le PC est un registre qui contient l'addresse de la prochaine instruction à exécuter, et est chargeable sur chaque cycle d'horloge, d'où le signale de la commande d'écriture est omis de la figure, et le PC est entendu d'être mis à jour chaque cycle d'horloge.

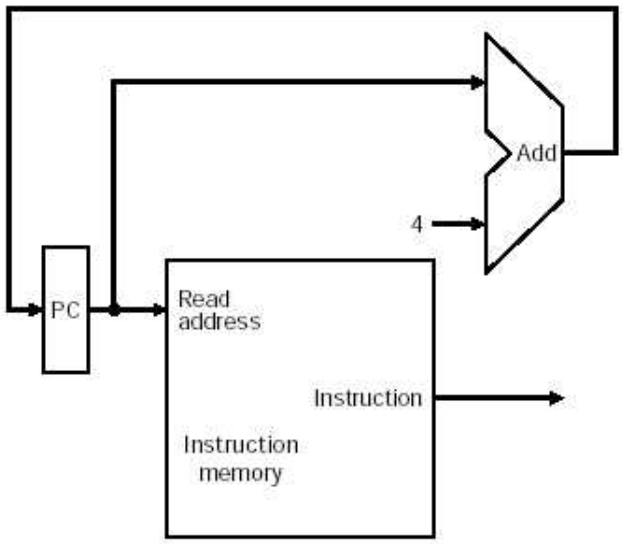


Figure 4: Chemin de données de fetch and incrément

3.3 Chemin de Données des Instructions de Type-R

Le chemin de données d'une instruction de Type-R est illustré dans la figure 5 qui utilise deux blocs de construction de base: un fichier de registre et un ALU. Tous les signaux de données sont 32 bits de large, et nous avont trente-deux registres à 32 bits, d'où un regisre d'adressage de 5 bits est nécessaire pour le fichier de registre. Le registre d'adressage contient deux ports de lecture et un port d'écriture, permettant trois accès simultanés (deux pour lecture et un pour écriture) au fichier de registre. L'ALU, d'autre part, est très similaire à celui conçu dans la classe, et se compose d'un ALU MIPS à 32 bits, capable de faire des opérations d'addition, de soustraction, ET/OU et SET LESS THAN (comparaison). Il fournit un signale d'état de *Zéro* qui indiquant si le résultat du ALU est égal a zéro (Zéro = 1) ou non (Zéro = 0). Notez que, comme susmontionné, le résultat du ALU doit être écrit de nouveau dans le fichier de registre pour utilisation ultérieure. Finalement, notez qu'un signal de contrôle (*RegWrite*) est nécessaire pour contrôler le mécanisme d'écriture du fichier de registre.

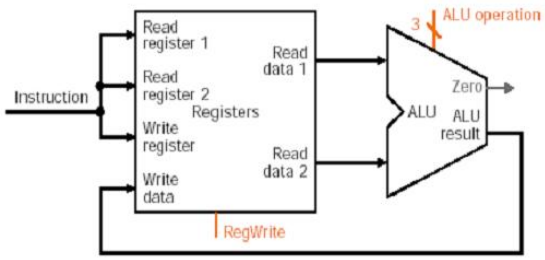


Figure 5: Chemin de données d'une instruction Type-R

3.4 Chemin de Données des Instructions de Type-I

Le chemin de données d'une instruction de Type-I est illustré dans la figure 6 qui utilise quatre blocs de construction de base: un fichier de registre, un ALU, une mémoire de données et une unité d'extension de signe. Tous les signaux de données sont 32 bits de large, à l'exception du champ d'adresse de 16 bits. Le fichier de registre est utilisé pour décoder les contenus du registre et pour les ajouter à la valeur d'offset avec signe étendu. Ceci est fait pour maintenir le signe de la valeur d'offset (cela s'appelle **adressage relatif à PC**). Une fois que l'adresse est calculée comme sortie du ALU, il est envoyé à l'unité de mémoire de données, qui contient un port d'écriture et un port de lecture, permettant une lecture et écriture simultanée à partir de et dans la mémoire. Dans le cas d'une instruction de stockage, le contenu du prèmiere opérande (dans l'instruction) sont verrouillés sur le bus de *Write Data*, tandis que le résultat du ALU est

verouillés sur le bus d'*Address* de la mémoire de données. L'action résultant est le stockage du contenu du prèmiere opérande dans la mémoire de données. Dans le cas d'une instruction de chargement, le contenu du mot de mémoire de données pointé par le bus d'*Address* est verouillés sur le bus de *Read Data* et écrit de nouveau (dans le cycle d'horloge suivant) dans le fichier de registre, dans le registre indiqué par la prèmiere opérande de l'instruction. L'action resultant est le chargement dans la prèmiere opérande, le contenus du mot de mémoire de mot. Notez que la mémoire de données est contrôlé par deux signaux différents, un qui permet de lire, et un autre qui permet d'écrire.

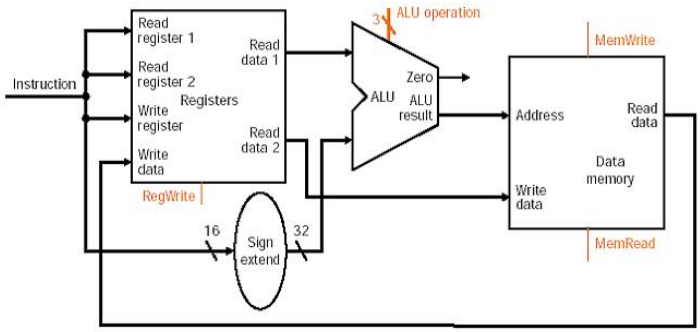


Figure 6: Chemin de données des instruction de Type-I

3.5 Chemin de Données des Instructions de Type-J

Le chemin de données d'une instruction de Type-J est illustré dans la figure 7 qui utilise trois blocs de construction de base: un fichier de registre, un ALU, et une unité d'extension de signe. Tous les signaux de données sont 32 bits de large, à l'exception du champ de valeur d'offset d'adresse. Le fichier de registre émets le contenu des deux opérandes, dans le cas d'une instruction de branchement, tandis que l'ALU procède à faire la comparaison des deux nombres. Si le dernier est reconnu comme étant égaux, le signale d'état de *Zéro* est mis à 1. L'unité de contrôle principale décide alors de prendre la branche ou non. Si la branche est à prendre, le trajectoire de branche est écrit dans le PC. Le trajectoire de branche comprend une adresse de base (PC + 4) et l'offset à signe étendu, décalé par 2 vers la gauche, afin de rendre le saut un mot d'offset. Dans le cas d'une instruction de saut, montré plus tard, nous calculons la nouvelle adresse simplement par un décalage vers la gauche par 2 du champ d'adresse dans l'instruction.

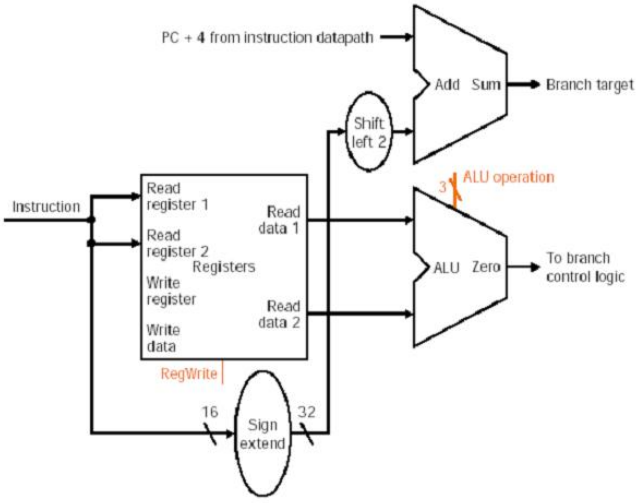


Figure 7: Chemin de Données des Instructions de Type-J

4 Le Chemin de Données Complet

Laissez-nous maintenant regrouper les chemins de données que nous avont construit dans la section précédente, et ajouter tous les multiplexeurs et signaux de contrôle associés, ainsi que l'unité de contrôle du ALU et l'unité de contrôle principale, pour construire notre processeur RISC à cycle simple. Le résultat est montré dans la figure 9.

4.1 Remarques Sur la Conception à Cycles Simple

Il ya quelques questions à noter sur notre processeur à cycle simple. Ils sont:

- Toutes les instructions doivent commencer et se terminer dans un seul cycle d'horloge;
- Toutes les valeurs doivent atteindre un état stable;
- Le temps de cycle est déterminé par le retard du plus long trajet;
- La logique du contrôle n'est pas défini systématiquement.

Cela dit, il ya quelques inefficacités impliqués dans la conception d'un processeur à cycle simple, qui comprennent la difficulté de soutenir des instructions plus complexe (ex. point mobile), ainsi qu'une fréquence d'horloge inférieure. Celles-ci pourraient être atténués par differents solutions, dont il est question ci-dessous.

5 Laboratoire

Dans ce laboratoire, vous allez implementer un processeur à cycle simple montré sur la figure9. Les spécifications d'entrées et de sorties du processeur sont montrés dans la Table 1. Nous allons travailler avec des chemins de données à 8 bits et des instructions de 32 bits de large, tandis que les lignes de contrôles restes évidemment constantes. A noter également que les chemins de données à 8 bit ne permettra les représentations et opérations en 8 bits. Nous allons implementer un fichier de registre de **huit registres à 8 bits**. La mémoire d'instructions restera à 32 bits, et contiendra un maximum de 256 instructions. La mémoire des données, cependant, sera de 8 bits, et va contenir un maximum de 256 mots.

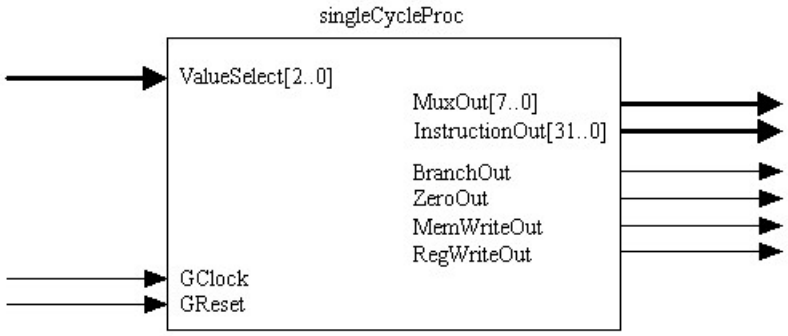


Figure 8: Entité d'un processeur à cycle simple

Port Type	*Name*	*Description*
Input	GClock	Global clock needed to synchronize the circuitry
Input	GReset	Global reset needed to bring the internals to known states
Input	ValueSelect[2..0]	Selector for MuxOut[7..0]
Output	MuxOut[7..0]	Multiplexer output controlled by ValueSelect[2..0]
Output	InstructionOut[31..0]	The current instruction being executed
Output	BranchOut	The branch control signal
Output	ZeroOut	The zero status signal
Output	MemWriteOut	The memory write control signal
Output	RegWriteOut	The register write control signal

Table 1: Spécifications des entrées et sorties

Les sorties et entrées de MuxOut[7..0] et ValueSelect[2..0] sont étroitement liés dans la manière suivante:

ValueSelect[2..0]	*MuxOut[7..0]*	*Description*
000	PC[7..0]	The program counter value
001	ALUResult[7..0]	The result of the current ALU operation
010	ReadData1[7..0]	The read data 1 port of the register file
011	ReadData2[7..0]	The read data 2 port of the register file
100	WriteData[7..0]	The write data port of the register file
Other	['0', RegDst, Jump, MemRead, MemtoReg, AluOp[1..0], AluSrc]	The remaining control information

Table 2: Sélection des Sorties du Multiplexeur

5.1 Mémoire des Instructions et des Données

Depuis que nous travaillons avec des mémoires d'instructions et de données séparées, vous serez autorisés à utiliser deux noyaux LPM fourni par Altera. La mémoire d'instruction peut être une instanciation de la fonction LPM_ROM (256x32), tandis que la mémoire des données peut être une instanciation de la fonction LPM_RAM (256x32). Cela dit, si le groupe choisit d'implementer leur propres module de RAM et ROM, ils sont accueillir de le faire.

Un échantillon du fichier de l'initialisation de la mémoire d'instructions (MIF) est donnée ci-dessous:

```
DEPTH = 256;
WIDTH = 32;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT
BEGIN

-- Use no operations (nop) for default instructions
[00..FF]:   00000000; -- nop(add $t1, $t1, $t1)

-- Place MIPS instruction here
00: 8C020000;   --lw $2,0 memory(00)=55
01: 8C030001;   --lw $3,1 memory(01)=AA
02: 00430820;   --add $1,$2,$3
03: AC010003;   --sw $1,3 memory(03)=FF
04: 1022FFFF;   --beq $1,$2,-4
05: 1021FFFA;   --beq $1,$1,-24

END;
```

Un échantillon du fichier de l'initialisation de la mémoire des données (MIF)

est donnée ci-dessous:

```
DEPTH = 256;
WIDTH = 8;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN

-- Default Memory Values
[00..FF]: 00;

-- Initial Values
00:   55;
01:   AA;

END;
```

5.2 Vérification: Exécution d'un Programme de Référence

Une fois terminé, la conception doit être testée et vérifié. Le processus de test peut être accompli tout au long de la conception, avec les tests unitaires, ensuite avec les tests modulaire, et finalement avec les tests de système. Dès que le processeurs semble de se comporter selon les spécifications, une étape de vérification doit être effectuée en exécutant le programme suivant, stocker dans la mémoire d'instructions (n'oubliez pas d'initialiser la mémoire de donnés comme indiqué dans les commentaires):

```
lw $2, 0;          $t2 = memory(00) = 55
lw $3, 1;          $t3 = memory(01) = AA
sub $1, $2, $3;    $t1 = $t2 - $t3 = 55
or $4, $1, $3;     $t4 = $t1 or $t3 = FF
sw $4, 3;          memory(03) = $t4 = FF
add $1, $2, $3;    $t1 = $t2 + $t3 = FF
sw $1, 4;          memory(04) = $t1 = FF
lw $2, 3;          $t2 = memory(03) = FF
lw $3, 4;          $t3 = memory(04) = FF
j 11;              jump to address 44
beq $1, $1, -44;   loop back to beginning of program
beq $1, $2, -8;    test if $t1 = $t2 ?
```

L'exécution de ce programme devra être démontré à votre TA comme preuve de succès pour votre conception. De plus, le calcul de la fréquence d'horloge maximale doit être montré et expliqué dans votre rapport.

Finalement, afin de mesurer l'efficacité de notre processeur à cycle simple, nous devront calculer son temps d'exécution de CPU comme le montre l'exemple à la page 373 de votre livre de cours. En utilisant cet exemple, et le programme de référence ci-dessus, calculer le temps d'exécution de notre processeur. Assumer, comme dans l'exemple, que les unités de mémoires ont un temps de fonctionnement de 2 ns, tandis que le fichier de registre a un temps de fonctionnement de 1 ns. **Cependant**, calculer le retard le plus pire des chemins de votre ALU et additionneurs, supposant que le retard d'une porte est 0.01 ns. Montrez tout votre travail et le raisonnement de votre résultats.

5.3 Boni: Conception d'un Processeur à Cycle Simple de 32 bits

Pour **5 points supplémentaires**, modifier la conception de votre processeur pour le faire complètement comme un processeur à cycle simple de données de 32 bits et d'instructions de 32 bits, et d'inclure l'instruction de *branchement si pas égal (bne)* dans votre conception (opcode = 5). Ne montrer pas tous les componsants de nouveau, mais à la place, discuter les modifications effectuées sur les composants afin de lancer le processeur de 32 bits. Si la logique du chemin de données et du contrôle ont changé, présenter et discuter les modifications. Vous devez démontrer votre nouvelle conception à votre TA.

5.4 Boni: Conception d'un Processeur à Cycle Multiple

Pour **15 points supplémentaires**, implementer le processeur à cycle multiple montré en classe et dans votre manuel. Encore un fois, ne montrer pas tous les componsants de nouveau, mais à la place, discuter les modifications effectuées sur les composants afin de réaliser l'implementation de cycle multiple. Si des nouveaux éléments ont été ajoutés, leur conception et leur réalisation doit être discuté. Après avoir complété la conception, lancer le programme de référence et comparer le temps d'exécution de CPU avec l'implementation du processeur à cycle simple. Vous devez démontrer votre nouvelle conception à votre TA.

6 Restrictions de Conception

- Implémentations utilisant Verilog ne seront pas acceptées. Effectuer toutes les implémentations utilisant VHDL seulement.
- La modélisation au niveau comportemental ne sera pas acceptée. La conception doit être faite au niveau de la modélisation structurelle.
- La conception utilisant Register Transfer Logic (RTL) et la volonté de codage sont obligatoire.
- Utiliser la conception graphique pour l'entité de haut niveau, et utiliser votre jugement pour tous les autres sous-blocs. Cependant, tous les mod-

ules atomiques doivent être implémentés en VHDL (bascule de type D, additionneur de 1 bit, comparateur de 1 bit et ainsi de suite).

- Aucunes instanciations de base ne seront autorisées (LPMs de Altera ou noyaux d'IP gratuits par exemple), à l'exception des noyaux pour LPM_ROM et LPM_RAM inclus dans ce manuel de laboratoire. Tous autres blocs de construction doivent être conçus et réalisés par le groupe.
- L'entité de haut niveau est donné dans la spécification de format des entrées/sorties, mais les internes sont laissées au groupe à faire. Un échantillon de la solution schématique pour le processeur à cycle simple a été donne, mais le groupe est libre de concevoir le système en utilisant d'autres méthodes.
- La conception doit être synchrone et globalement réinitialisable. Cela signifie que les signales de l'horloge globale et de la réinitialisation sont nécessaires dans les deux blocs fonctionnels (les processeurs à cycle simple et multiple)
- Simuler tous les conceptions et vérifier vos résultats de simulation avec votre calculs théoriques (ex. lancer le programme de référence)
- Télécharger la conception à la puce Cyclone de votre planches DE-2, et utiliser les commutateurs DIP pour entrer ValueSelect[2..0], et les seize LEDs pour démontré la fonctionnalité correcte des sorties (MuxOutput[7..0] et des signaux de contrôle montré sur la figure 8)
- Montrer seulement l'instruction actuelle (InstructionOut[31..0]) dans la démonstration de la simulation. Vous n'avez pas besoin de la montrer dans la démonstration en direct.
- Chaque groupe doit démontrer une version fonctionnelle du laboratoire au TA avant la date d'échéance du rapport.

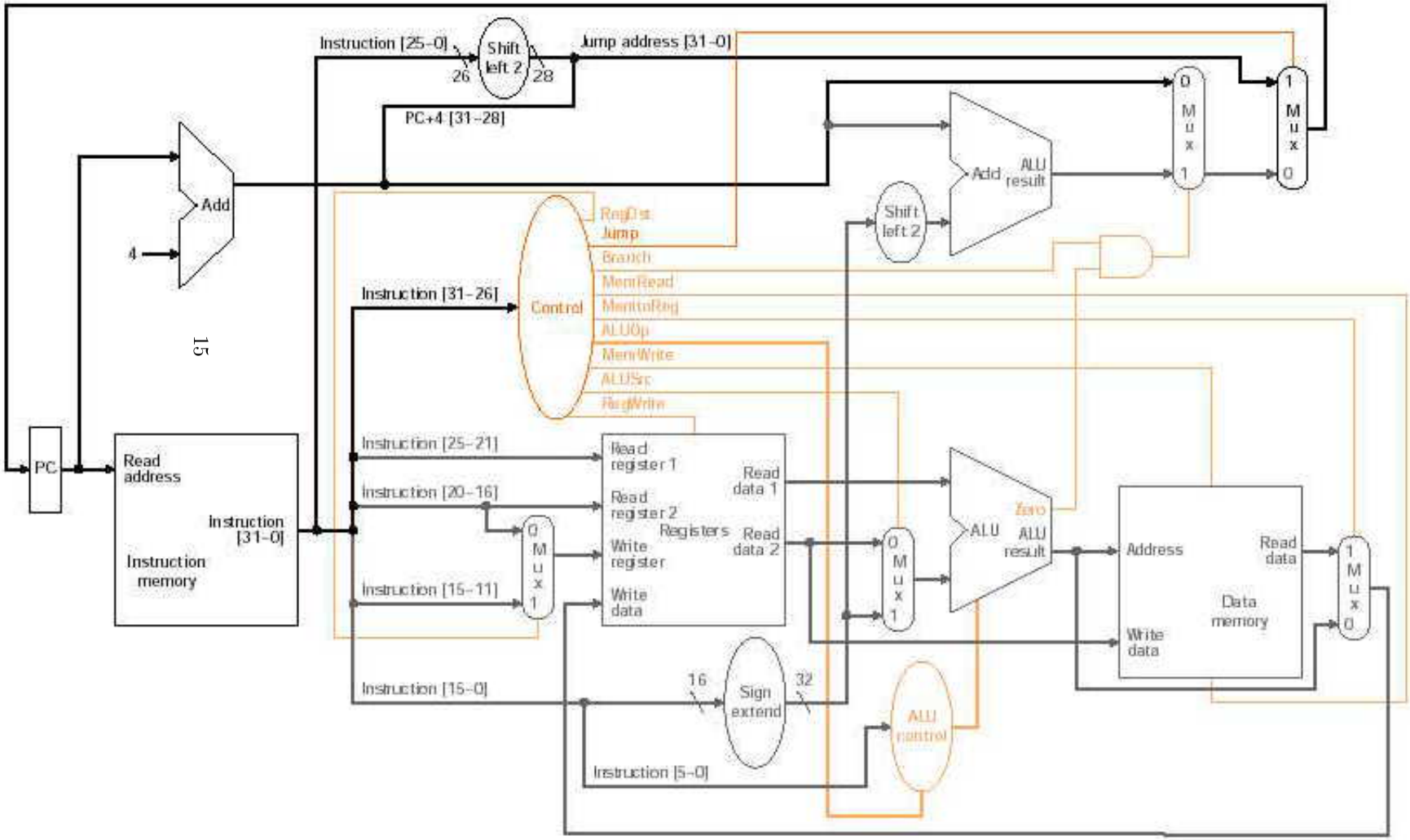
7 Rappels pour le Rapport

- Inclure simulations de synchronisation temporelle avec des explications pour tous fichiers de source VHDL.
- Décrire et commenter tous vos fichiers de source VHDL.
- Inclure une représentation organigramme de votre solution au problème.
- Inclure un schéma de blocs de votre solution au problème.
- Si vous utilisez la conception d'ASM, inclure toutes les diagrammes et chemins appropriées (contrôle et données).
- Si vous utilisez la conception de FSM, inclure le diagramme d'états avec toutes les entrées et sorties appropriées.

- Décrivez, dans vos propres mots, votre solution au problème.
- Décrivez vos obstacles de conception et comment ils ont été résolus.
- Annexer tout code VHDL et la conception graphique dans votre rapport.
- Soumettre une copie électronique de tous les fichiers VHDL et de conception graphique dans votre rapport.

8 Acknowledgements

All figures (except 2, 3 and 8) are taken out of Chapter 4 of our course textbook. For more detailed explanations and examples, please refer to *Computer Organization and Design, The Hardware/Software Interface* by David A. Patterson and John L. Hennessy.


Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
M u x
0
1
ALU result
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data