POLYTECHNIQUE
MONTRÉAL

INF1600
Architecture des micro-ordinateurs

TP4

Groupe 02 (B2)

Soumis par :
Charles de Lafontaine - **2076524**
Geneviève Pelletier-Mc Duff - **2088742**

Le 7 avril 2021

Barème de correction

TP 4		**/4,00**
Q1	/0,50	
Q2	/0,25	
Q3	/0,25	
Q4	/0,25	
Q5	/0,25	
Q6	/0,25	
Q7	/0,25	
Q8	/0,25	
Q9	/0,25	
Q10	/1,50	

Réponses aux questions

Q1 : <u>Le processeur qui vous est fourni suit-il une architecture de von Neumann ou de Harvard ? Justifiez votre réponse.</u>

Harvard, puisque notre processeur est connecté à deux bus de mémoire. Le premier étant responsable de stocker les instructions dans le registre *inst_memory*, alors que le second est responsable des données du programme dans le registre *data_memory*. Nous ne sommes donc pas dans une architecture de von Neumann, où une seule mémoire se charge de stocker à la fois les instructions du programme et les données.

Q2 : <u>Assurez-vous que le banc d'essai (*simple_risc_tb*) exécute le programme *program_0* puis inspecter les signaux de la simulation. Selon vos observations, quel est le *CPI* d'une instruction *op_alu* ?</u>

L'instruction *op_alu* s'effectue en trois cycles d'horloges en regardant les lignes *clk* et *state*. Le *CPI* d'une instruction de type *op_alu* est donc 3 (*fetch*, *decode*, *op_alu* ; **Figure 1** ci-bas).

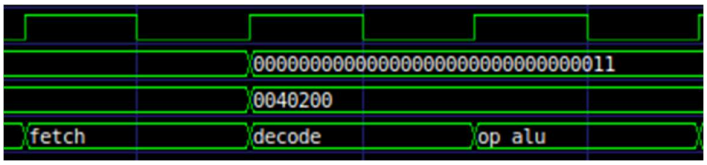

Figure 1. Simulation de l'exécution en trois cycles d'horloge de l'instruction *op_alu* avec *GTKWave*.

Q3 : <u>Donnez le *RTN* concret d'une instruction d'addition de deux registres, incluant la recherche d'instruction. Notez *Mi* la mémoire d'instructions.</u>

Nous avons un processeur avec l'architecture *RISC*. Ainsi, le compteur de programme doit être incrémenté de 1 et non de 4 d'une instruction à l'autre. Par ailleurs, trois cycles d'horloge sont nécessaires pour exécuter l'instruction d'addition, passant par un *fetch*, un *decode* et un *op_alu*. Veuillez noter que nous avons seulement deux étapes dans notre *RTN* concret, puisque l'étape *decode* correspond à un processus interne au processeur qui n'est pas représenté dans le *RTN* concret.

***RTN* abstrait :**

R [*rdst*] ← R [*rsrc1*] + R [*rsrc2*];

***RTN* concret:**

PC ← PC + 1 : IR ← Mi [PC];

R [*rdst*] ← R [*rsrc1*] + R [*rsrc2*]; si *wreg* = 1, *rdst* ≡ IR<20..16>, *rsrc1* ≡ IR<12..8> et *rsrc2* ≡ IR<4..0>.

Q4 : Assurez-vous que le banc d'essai (*simple_risc_tb*) exécute le programme *program_0* puis inspecte les signaux de la simulation. Selon vos observations, quel est le *CPI* d'une instruction *ldi* ?

Le *CPI* d'une instruction *ldi* est 3 (*fetch*, *decode*, *ldi*) puisqu'il faut trois cycles d'horloge pour l'effectuer (voir **Figure 2**).

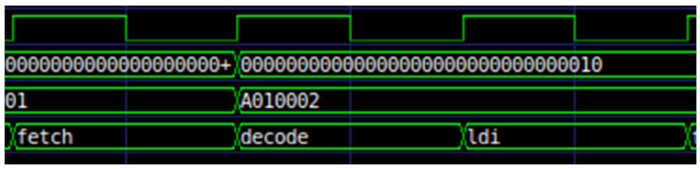

Figure 2. Simulation de l'exécution en trois cycles d'horloge de l'instruction *ldi* avec *GTKWave*.

Q5 : Donnez le *RTN* concret de l'instruction ldi, incluant la recherche d'instruction. Notez *Mi* la mémoire d'instructions.

Veuillez noter que nous avons seulement deux étapes dans notre *RTN* concret, alors que le *CPI* est de trois puisque l'étape de *decode* prend un cycle d'horloge, mais est un processus interne au processeur qui n'est pas représenté dans le *RTN* concret.

RTN abstrait :

R [*rdst*] ← IR<15..0>;

RTN concret:

PC ← PC + 1 : IR ← Mi [PC];

R [*rdst*] ← R [*imm*]; si *wreg* = 1 et *imm* ≡ IR<15..0>.

Q6 : <u>Assurez-vous que le banc d'essai (*simple_risc_tb*) exécute le programme program_0 puis inspecte les signaux de la simulation. Selon vos observations, quel est le *CPI* d'une instruction *read_mem* ?</u>

Le *CPI* d'une instruction *read_mem* est 3 (*fetch*, *decode*, *read_mem*) puisqu'il faut trois cycles d'horloge pour l'effectuer (voir **Figure 3**).

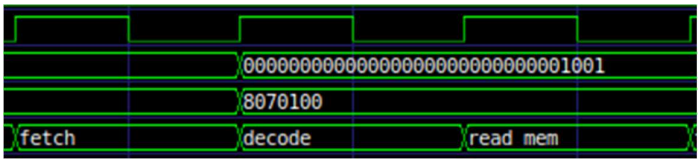

Figure 3. Simulation de l'exécution en trois cycles d'horloge de l'instruction *read_mem* avec *GTKWave*.

Q7 : <u>Donnez le *RTN* concret de l'instruction *read mem*, incluant la recherche d'instruction. Notez *Mi* la mémoire d'instructions.</u>

Comme il a été mentionné précédemment, l'étape de *decode* prend un cycle d'horloge de plus, mais n'est pas représentée dans le *RTN* concret ci-bas.

RTN abstrait :

R [*rdst*] ← Md [R [*rsrc1*]];

RTN concret:

PC ← PC + 1 : IR ← Mi [PC];
R [*rdst*] ← Md [R [*rsrc1*]]; si *wreg* = 1, *rdst* ≡ IR<20..16> et *rsrc1* ≡ IR<12..8>.

Q8 : <u>Assurez-vous que le banc d'essai (*simple_risc_tb*) exécute le programme program_0 puis inspecte les signaux de la simulation. Selon vos observations, quel est le *CPI* d'une instruction *write_mem* ?</u>

Le *CPI* d'une instruction *write_mem* est 3 (*fetch*, *decode*, *write_mem*) puisqu'il faut trois cycles d'horloge pour l'effectuer (voir **Figure 4**).

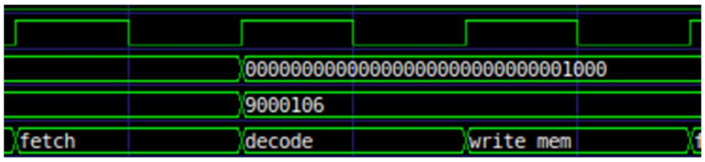

Figure 4. Simulation de l'exécution en trois cycles d'horloge de l'instruction *write_mem* avec *GTKWave*.

Q9 : <u>Donnez le *RTN* concret de l'instruction *write_mem*, incluant la recherche d'instruction. Notez *Mi* la mémoire d'instructions</u>.

Malgré le *RTN* concret de deux lignes, notre *CPI* est bel et bien de 3 puisque le *decode* est une étape interne au processeur non représentée demandant un cycle d'horloge.

RTN abstrait :

Md [R [*rsrc1*]] ← R [*rsrc2*];

RTN concret:

PC ← PC + 1 : IR ← Mi [PC];

Md [R [*rsrc1*]] ← R [*rsrc2*]; si *wmem* = 1, *rsrc1* ≡ IR<12..8> et *rsrc2* ≡ IR<4..0>.

Q10 : <u>Modifiez le contenu de *program_1* dans *simple_risc_programs.vhd* afin d'implémenter un programme qui calcule les six (6) premiers termes d'une suite numérique S(*n*), soit S(0) à S(5), et les stocke séquentiellement en mémoire.</u>

Tout d'abord, nous avons utilisé le jeu d'instructions du processeur donné dans ce laboratoire, soit le *RISC* simple, afin d'écrire le *program_1* permettant de calculer les 6 termes de la suite suivante : *S(n+2) = 1 + S(n+1) + S(n)* où S(0) = 1 et S(1) = 2 (voir **Figure 5**). Veuillez noter que nous avons écrit notre programme en utilisant une boucle afin qu'il puisse calculer les *n* termes de la suite. Nous n'avons qu'à modifier la ligne 27 (contenu du registre 1) pour indiquer le nombre de termes voulu (nombre d'itérations = nombre de termes (*n*) - 2). De plus, nous avons annoté chacune des lignes afin d'expliquer le fonctionnement plus en détail du programme.

Il est possible de voir le résultat de la simulation de notre programme *program_1* sur *GTKWave* à la **Figure 6**. Nous pouvons constater que les 6 premiers termes de la suite *S(n+2) = 1 + S(n+1) + S(n)* où S(0) = 1 et S(1) = 2 sont exacts et stockés séquentiellement en mémoire de données à leurs indices respectifs. Ainsi, notre programme est valide.

```
26  constant program_1 : inst_mem_type := (
27      x"A010004", -- r1    <- +4              - addr x00 - À changer pour le nombre d'itérations voulues suite pour compléter la suite de { 1, 2, ... }
28      x"A020001", -- r2    <- +1              - addr x01 - Constante à 1
29      x"A030001", -- r3    <- +1              - addr x02 - Correspond à notre variable temporaire 1 pour les calculs qui suivent
30      x"A040002", -- r4    <- +2              - addr x03 - Correspond à notre variable temporaire 2 pour les calculs qui suivent
31      x"9000003", -- M[r0] <- r3              - addr x04 - Stocker dans la mémoire le premier résultat de la suite, soit S0 (1)
32      x"9000204", -- M[r2] <- r4              - addr x05 - Stocker dans la mémoire le second résultat de la suite, soit S1 (2)
33      x"A050001", -- r5    <- +1              - addr x06 - Mettre à 1 le registre correspondant à l'index de la mémoire pour laquelle nous voulons insérer les résultats de la suite
34      x"0030203", -- r3    <- r3 + r2         - addr x07 - Nous additionnons la constante (1) à la variable temporaire 1; la réponse est stockée dans la variable temporaire 1
35      x"0030403", -- r3    <- r4 + r3         - addr x08 - Nous additionnons les deux variables temporaires ensemble; la réponse est stockée dans la variable temporaire 1
36      x"0050502", -- r5    <- r5 + r2         - addr x09 - Nous incrémentons de 1 le registre correspondant à l'index de la mémoire
37      x"9000503", -- M[r5] <- r3              - addr x0A - Stocker dans la mémoire les résultats de la suite où le nombre d'itérations est impair
38      x"1010102", -- r1    <- r1 - r2         - addr x0B - Nous décrémentons le nombre d'itérations de 1
39      x"C010013", -- jz 0x13                  - addr x0C - Le programme se termine si le nombre d'itérations est nul
40      x"0040403", -- r4    <- r4 + r3         - addr x0D - Nous additionnons les deux variables temporaires ensemble; la réponse est stockée dans la variable temporaire 2
41      x"0040402", -- r4    <- r4 + r2         - addr x0E - Nous additionnons la constante (1) à la variable temporaire 2; la réponse est stockée dans la variable temporaire 2
42      x"0050502", -- r5    <- r5 + r2         - addr x0F - Nous incrémentons de 1 le registre correspondant à l'index de la mémoire
43      x"9000504", -- M[r5] <- r4              - addr x10 - Stocker dans la mémoire les résultats de la suite où le nombre d'itérations est pair
44      x"1010102", -- r1    <- r1 - r2         - addr x11 - Nous décrémentons le nombre d'itérations de 1
45      x"C020007", -- jnz 0x07                 - addr x12 - Nous bouclons si le nombre d'itérations n'est pas nul pour compléter la suite
46      others => (others => '1')
47  );
```

Figure 5. Capture d'écran présentant le programme *program_1* permettant de calculer les 6 premiers termes de la suite suivante : *S(n+2) = 1 + S(n+1) + S(n)* où S(0) = 1 et S(1) = 2.

Figure 6. Résultat de l'affichage de la simulation du *program_1* sur *GTKWave*. Veuillez noter que les 6 premiers termes de la suite se retrouvent bel et bien dans la mémoire des données à leurs indices respectifs.