Rapport du projet d'ASR 1

PAULIN Loïs, STAUB Ruben

4 janvier 2016

1 Partie 1 : Processeur avec pipeline

1.1 Étage IF

Question 1 : Module IF :
Entrées : PC
Sorties : code de l'instruction correspondante + PC.

Question 2 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.2 Étage ID

Question 1 : Module ID :
Entrées : INS - PC
Sorties : OP - rT - SRC1 - SRC2 - ope0
Le module ID prend en entrée un code d'instruction et PC et retourne les différentes composantes du code (op code, numéro du registre de retour, valeurs des registres en paramètres, valeur de la constante).

Question 2 : Register file :
Entrées : SRC1_in - SRC2_ - WE - TGT_sel - TGT
Sorties : SRC1 - SRC2
Register file contient les 8 registres, SRC1 et SRC2 sont le contenu des registres désignés par les adresses SRC1_in et SRC2_in, de plus lorsque WE vaut 1, TGT est stocké dans le registre TGT_sel.

Question 3 : CTL7 :
Entrées : OP_in - rA
Sorties : OP_out - rT
CTL7 met rT à 000 si l'instruction ne modifie pas de registre sinon rT vaut rA. OP_out vaut toujours OP_in.

Question 4 : CTL6 :
Entrées : OP
Sorties : s2 - op0
CTL6 met s2 à 1 si OP désigne ADD ou NAND, et à 0 sinon. op0 est à 1 si OP désigne LUI, et à 0 sinon.

Question 5 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.3 Étage EX

Question 1 : Module EX :
Entrées : OP - rT - PC - OPERAND0 - OPERAND1 - OPERAND 2
Sorties : OP - rT - PC - STORE_DATA - ALU_OUTPUT
Le module EX effectue les calculs nécessaires pour l'instruction désigné par OP à partir de OPERAND0 - OPERAND1 - OPERAND 2 et met le résultat dans ALU_OUTPUT. Dans le cas d'un OPERAND2 est transmis dans STORE_DATA.

Question 2 : CTL3 :
Entrées : OP - EQ !
Sorties : MUXpc - FUNCalu
CTL3 détermine dans MUXpc la façon dont est calculée PC suite à l'exécution de l'instruction (PC+1 si l'instruction n'est pas un branchement, la nouvelle adresse sinon), FUNCalu choisi quelle opération doit effectuer l'ALU.

Question 3 : ALU :
Entrées : SRC1 - SRC2 - FUNCalu
Sorties : EQ ! - out
L'ALU met dans out, en fonction de FUNCalu, soit une addition, soit un NAND, soit SRC2, il met EQ ! à 1 si SRC1 vaut SRC2.

Question 4 : CTL4 :
Entrées : OP
Sorties : MUXimm
CTL4 choisi l'entrée SRC2 de l'ALU en fonction de l'instruction. Si l'instruction est un JALR alors SRC2 vaut PC+1, sinon si l'instruction possède une constante alors SRC2 est OPERAND0 sinon SRC2 vaut OPERAND2.

Question 5 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.4 Étage MEM

Question 1 : Module MEM :
Entrées : OP - rT - PC - MEM - ALU_OUTPUT
Sorties : rT - RF_WRITE_DATA
MEM gère la mémoire. Si OP désigne SW alors on stocke MEM à l'adresse ALU_OUTPUT. Sinon si OP désigne LW alors on met dans RF_WRITE_DATA la valeur à l'adresse ALU_OUTPUT. Sinon on transmet ALU_OUTPUT dans RF_WRITE_DATA.

Question 2 : CTL2 :
Entrées : OP
Sorties : WEdmem - MUXout
CTL2 choisi dans WEdmem selon OP si l'on lit ou écrit dans la mémoire, et choisi dans MUXout si la sortie de MEM est ALU_OUTPUT (OP différent de LW) ou une valeur de la mémoire (OP égal à LW).

Question 3 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.5 Étage WB

Question 1 : Module WB :
Entrées : rT - RF_WRITE_DATA
Sorties : rT_out - RF_WRITE_DATA_out
WB envoie à Register_File le numéro du registre dans lequel écrire ainsi que quoi y écrire.

Question 2 : CTL1 :
Entrées : rT Sorties : WErf CTL1 met WErf à 1 si le registre rT n'est pas 000.

Question 3 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.6 Pipeline

Question 1 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/ProcoDeal.circ` ou ici pour le fichier brut.

1.7 Assembleur RiSC-16

Question 1 : Dans le programme suivant :
lui r2, 0x8000
lw r1, r2, 1

Le calcul de l'adresse (r2 + 1) à laquelle chercher le contenu pour le copier dans le registre r1 se fait à l'étage EXE. Or l'instuction précédente qui modifie r2 n'est arrivée qu'au début de l'étage MEM. Ainsi il faudrait faire un bypass de type EXE→EXE, ou ajouter entre les deux instructions : 3 instructions vides pour laisser le temps à r2 d'être mis à jour (WB→ID) au moment du calcul de r2+1. Par conséquent, une version correcte de ce programme est :

lui r2, 0x8000
.space 2
lw r1, r2, 1

ou de manière équivalente :

```
lui r2, 0x8000
nop
nop
lw r1, r2, 1
```

Question 2 : Voir q7.2.S sur le git associé : `https://github.com/Plopounet13/ProcoDeal/blob/628117f505d9bed9d1d7132adda3f4aa45548440/q7.2.S` ou ici pour le fichier brut.

Les instructions .space X permettent de laisser suffisament de temps pour les registres d'être écrits avant d'être lus par une autre commande (ce qui sera résolu dans la partie 2 avec des bypass) et après des sauts pour attendre que le PC soit bien mis à jour avant d'éxécuter une commande suivante (résolu dans la partie 4 avec la logique stall and stomp).

2 Partie 2 : Pipeline avec logique bypass

2.1 Étage WB

Question 1 : Les entrées du module WB restent identiques.

Par contre les sorties rT et RF_WRITE_DATA sont dédoublées après le passage au travers du registre et directement dirigées vers le module EXE. Ainsi, ces données sont données sont traitées comme des entrées conventionnelles de EXE (grâce à la synchronisation des registres après WB et avant EXE).

En effet, nous avons préféré implémenter le bypass principalement au niveau du pipeline, pour plus de simplicité au niveau des modules, et par soucis de modularité (il semble plus naturel que les bypass soient gérés par le pipeleine que par les modules en eux-mêmes).

Question 2 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/3016e78db55d66684f28d8adf5c98ce117739173/ProcoDeal.circ` ou ici pour le fichier brut.

2.2 Étage MEM

Question 1 : (Voir l'étage WB) Les entrées du module WB restent identiques.

Par contre les sorties rT et RF_WRITE_DATA sont dédoublées après le passage au travers du registre et directement dirigées vers le module EXE. Ainsi, ces données sont données sont traitées comme des entrées conventionnelles de EXE (grâce à la synchronisation des registres après WB et avant EXE).

En effet, nous avons préféré implémenter le bypass principalement au niveau du pipeline, pour plus de simplicité au niveau des modules, et par soucis de

modularité (il semble plus naturel que les bypass soient gérés par le pipeleine que par les modules en eux-mêmes).

Question 2 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/3016e78db55d66684f28d8adf5c98ce117739173/ProcoDeal.circ` ou ici pour le fichier brut.

2.3 Étage ID (Bonus)

Le module ID est également modifié puisque les entrées SCR1_in et SCR2_in du RegisterFile (dans l'étage ID) sont dupliquées et dirigées vers les sorties supplémentaires respectives s1 et s2 du module ID.

2.4 Étage EX

Question 1 : Module EXE :
Entrées supplémentaires : s1 - s2 - EXE_bypass - MEM_bypass - WB_bypass - EXE_rT - MEM_rT - WB_rT
Sorties : Identiques
Désormais, si le registre s1 ou s2 en entrée (via OPERAND2 ou OPERAND1) est écrit (dans rT) par une instruction en cours d'éxécution qui a déjà fait des calculs dessus (autrement dit, si un bypass est possible), alors la valeur de ce registre est directement récupérée de l'instruction en cours d'éxécution. Ainsi, si s1 correspond à (dans l'ordre de priorité décroissant) EXE_rT ou MEM_rT ou WB_rT, OPERAND1 est remplacée par EXE_bypass ou MEM_bypass ou WB_bypass correspondant, et de même pour s2.

Question 2 : CTL5 :
Entrées : s1 - s2 - EXE_rT - MEM_rT - WB_rT
Sorties : MUX_alu1 - MUX_alu2
Si s1 est non nul (r0 restera toujours égal à 0, et CTL1 met rT à 0 lorsque l'instruction n'écrit pas dans rA) et si s1 correspond à un registre en train d'être mis à jour dans les étages suivants (EXE_rT ou MEM_rT ou WB_rT), alors CTL5 indique à MUX_alu1 de prendre la valeur la plus à jour pour OPERAND1 (c'est-à-dire de faire le bypass EXE→EXE (MUX_alu1 = 1) en priorité sur MEM→EXE (MUX_alu1 = 2) et lui-même en priorité sur WB→EXE (MUX_alu1 = 3)). Si aucun bypass n'est mis en place alors MUX_alu1 = 0.

De même pour s2.

Question 3 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/3016e78db55d66684f28d8adf5c98ce117739173/ProcoDeal.circ` ou ici pour le fichier brut.

2.5 Pipeline

Question 1 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/3016e78db55d66684f28d8adf5c98ce117739173/ProcoDeal.circ` ou ici pour le fichier brut.

2.6 Assembleur RiSC-16

Question 1 : Dans le programme donné :
lui r2, 10
beq r1, r2, label
addi r2, r2, 1

Premièrement, l'instruction lui chargera les 10 bits de poids fort de 0000000000000010 et le reste à 0, donc 0000000000000000 dans r2...

Mais le vrai problème est que lors de l'instruction beq, le changement d'adresse de la prochaine instruction (adresse de l'instruction pointée par label) est calculée à partir de l'étage EXE. Ainsi, les 2 instructions suivantes (ici addi r2, r2, 1) seront éxécutées avant que le changement de PC soit effectif.

Par conséquent, une version correcte de ce programme est :

```
addi r2, r0, 10
beq r1, r2, label
.space 2
addi r2, r2, 1
```

ou de manière équivalente :

```
addi r2, r0, 10
beq r1, r2, label
nop
nop
addi r2, r2, 1
```

Question 2 : Voir q7.3.S sur le git associé : `https://github.com/Plopounet13/ProcoDeal/blob/6962e645c41df63370ce54c08b1a13df091c0340/q7.3.S` ou ici pour le fichier brut.

3 Partie 3 : Mapping memory

3.1 Étage MEM

Question 1 : Module MEM :
Entrées supplémentaires : READ_DATA
Sorties supplémentaires : WE
Le module MEM envoie WE aux blocs de mémoire qui à partir de l'adresse et de l'ALU_OUTPUT vont soit enregistrer MEM à l'adresse de ALU_OUTPUT, soit retourner la valeur présente en mémoire.

3.2 Memory mapping

Question 1 : On utilisera la segmentation mémoire suivante :
0000 - ffe5 : RAM
ffe6 - fff5 : LED MATRIX

fff6 - fff9 : KEYBOARD
fffa - fffd : TTY
fffe - ffff : JOYSTICK
On remarquera que la RAM n'a pas une taille en puissance de 2, ce qui décale les adresse et rend plus compliqué l'extraction des adresses locales des périphériques depuis les adresses sur 16 bits. Cependant, on a jugé le gain de RAM suffisemment important pour valoir ce coût.
Les périphériques sont mappés localement comme suivant :
LED MATRIX :
0000 → colonne gauche
... 1111 → colonne droite
KEYBOARD :
00 → registre d'activation de lecture
01 → registre de remise à zéro
10 → registre caractère présent dans le buffer
11 → registre de lecture
TTY :
00 → registre d'activation d'écriture
01 → registre de remise à zéro
10 → registre d'écriture
JOY :
0 et 1 → registre abcisses+ordonnées joystick
L'adresse globale est obtenue en ajoutant l'adresse locale à l'adresse globale de début du périphérique.

Question 2 : Module memory mapping :
Entrées : ADR - MEM - WE Sorties : READ_DATA Le module de memory mapping prend en entrée l'adresse ADR sur 16 bit et en extrait un périphérique à activer ainsi que l'adresse locale sur ce périphérique. Si WE vaut 1 MEM est écrit à l'adresse locale si possible (Ex : on ne peut pas écrire dans le joystick). Dans un souci de simplicité d'utilisation le module à été câblé directement dans le pipeline afin de pouvoir interagir avec et observer les périphérique depuis la vue du pipeline.

3.3 Pipeline

Question 1 : Voir version correspondante : `https://github.com/Plopounet13/ProcoDeal/blob/3016e78db55d66684f28d8adf5c98ce117739173/ProcoDeal.circ` ou ici pour le fichier brut.

3.4 Assembleur RiSC-16

Question 1 : Voir q4.1.S sur le git associé : `https://github.com/Plopounet13/ProcoDeal/blob/6962e645c41df63370ce54c08b1a13df091c0340/q4.1.S` ou ici pour le fichier brut.