Université du Québec à Montréal

Cours : EMB7020

Codesign

Devoir 02:

Microarchitecture du Processeur pour le Jeu Bulle (Parties 2 à 4)

Présenté par :

Luis Fabiano Batista : BATL24077305

Trimestre : Hiver 2013

1. Introduction

Le but de ce devoir est de proposer une trousse d'instructions utiles pour implémenter la fonctionnalité de gestion de la taille et de la position des disques (pas d'instructions d'entrées/sorties) pour le Jeu Bulle.

Le code du jeu « Bulle » conçu en langage d'haut niveau (Visual Basic .NET) a été analysé pour la conception d'un jeu d'instructions utiles à ce projet. Aussi, dans le cadre de ce travail, le schéma d'un processeur dédié sera présenté, qui supportera toutes les instructions nécessaires au bon fonctionnement du jeu. Conception du jeu d'instruction d'un processeur.

Finalement, le processeur dédié sera codé en VHDL, et un *testbench* sera développé pour valider son fonctionnement.

2. Conception du jeu d'instruction d'un processeur

La solution VHDL sera inspire dans la section 7.6 du livre « Digital Design and Computer Architecture » de D. M. and S. Harris. Dans ce livre juste les fonctions suivantes ont été implémentées dans la microarchitecture du processeur :

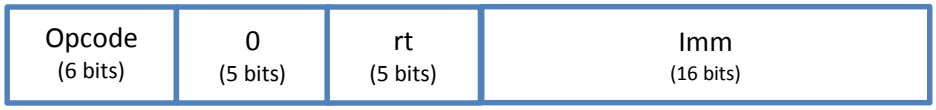
- Instructions du type R arithmétiques et logiques: *add, addi, sub, and, or, slt*
- Instructions de mémoire : *lw, sw*
- Branches: *beq*
- Jumps : *j*

L'architecture du processeur présenté dans le livre sera modifiée pour accommoder les instructions supplémentaires suivantes :

Name	Description	Syntaxe	Opération
li	Load immediate	li rd, imm	Charge *16-bit immediate* dans le registre spécifié par rd
mul	Multiplication without overflow	mul rd, rs, rt	Stocke dans le registre spécifié par rd la partie *low-order 32 bits* du produit des registres spécifiés par rs et rt
jal	Jump and link	jal JTA	$ra = PC+4, PC = JTA
jr	Jump register	Jal rs	Donne au PC la valeur stocké dans rs
bge	Branch on greater or equal	bge rs,rt, branch	PC=branch si rs>=rt

2.2 Instruction « li »

Cette instruction doit charger la valeur des bits immédiats dans le registre indiqué. Elle doit faire partie de ce projet parce qu'il faut charger des paramètres fixes (tel comme les limites de la taille d'écran) dans des registres spécifiques.

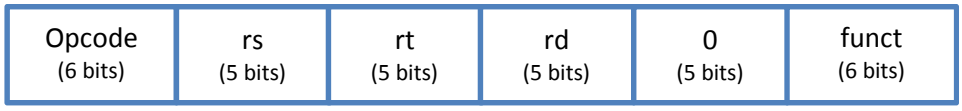


Champ	Valeur
Opcode	8
rs	<adresse du registre qui devra garder la valeur
Imm	<valeur de 16 bits à être garder dans le registre>

C'est une pseudo-instruction équivalente à l'instruction *addi rt, $0, imm*. Dans ce cas-ci, cette instruction ne demande pas des changements dans la microarchitecture. Elle a été ajoutée dans la syntaxe MIPS du jeu juste pour simplifier la tâche de représentation du code.

2.3 Instruction « mul »

Cette instruction effectue la multiplication de deux registres de 32 bits, et les premières 32 bits (moins significatifs) du résultat seront gardés sur un troisième registre. Cette instruction sera nécessaire aux calculs de la puissance de deux de la différence des coordonnés X et Y entre deux bulles.



Champ	Valeur
Opcode	0
rd	<adresse du registre qui devra garder le premières 32bit du résultat
rs	<adresse du première registre d'entré>
rt	<adresse du deuxième registre d'entré>
funct	0x30

Pour l'implémentation de cette fonction, il faudra juste modifier l'ALU et la logique de l'unité de contrôle pour exécuter des opérations de multiplication.

2.4 Instruction « jal »

Inconditionnellement donne au PC la valeur spécificité dans le champ *target* multiplié par 4, et garde la valeur de la prochaine instruction dans le registre $ra. Cette instruction est nécessaire à la implémentation des procédures de calculs, pour de cette façon faciliter la réutilisation du code.

Opcode (6 bits)	target (26 bits)

Champ	Valeur
Opcode	3
target	<adresse de la branche appelée divisée par 4>

L'implémentation de l'instruction *jal* est très similaire à celle de l'instruction j. La différence consiste à faire l'opération d'écriture de l'adresse de retour (PC+4) dans le registre $ra (registre $31).

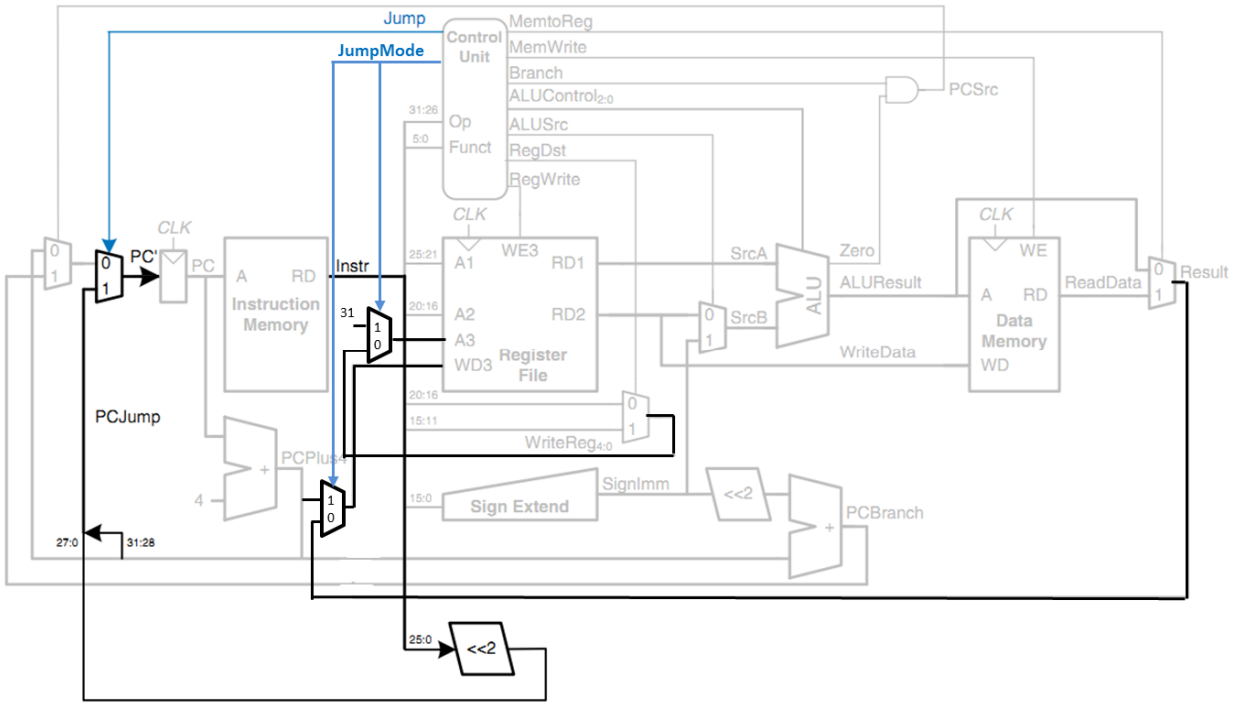


Figure 1 - Microarchitecture avec l'implémentation de la fonction jal

2.5 Instruction « jr »

Inconditionnellement donne au PC la valeur stockée dans l'adresse indiquée par le champ *rs*. Cette instruction sera appelée à la fin de chaque bloc de procédure exécutée.

Opcode (6 bits)	rs (5 bits)	0 (5 bits)	0 (5 bits)	0 (5 bits)	0 (6 bits)

Champ	Valeur
Opcode	5
rs	<adresse du registre $ra>

Comme l'instruction jr est aussi une variation de l'instruction *jal*, le signal *JumpMode* montré dans le dessin 2.4 devra être modifié (passer de 1 bit à 2 bits) pour accommoder cette instruction supplémentaire.

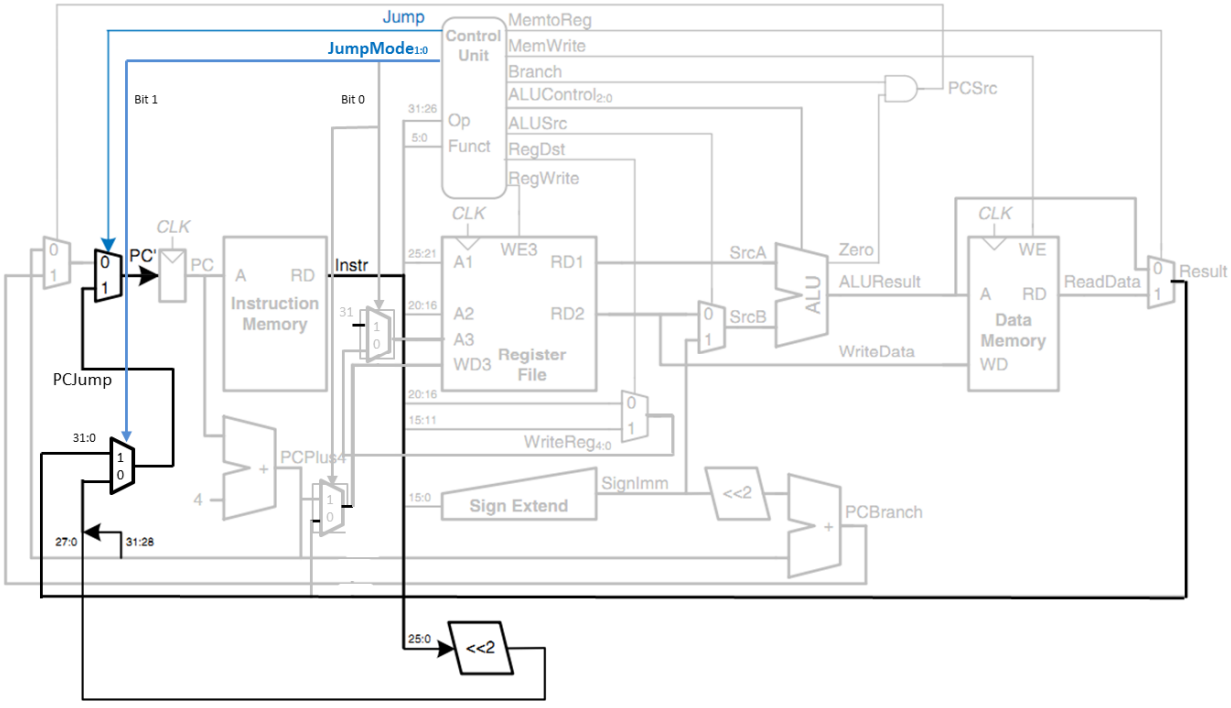


Figure 2 - Microarchitecture avec l'implémentation de la fonction jr

2.6 Instruction « bge»

Inconditionnellement branche le nombre d'instructions spécificité par l'offset si la différence entre les valeurs de deux registres d'entrée est plus grand ou égal à 0 ($rs-$rt < 0). Elle sert à comparer la distance entre le centre des bulles avec la somme de leur rayon pour la détection de superposition.

Opcode (6 bits)	rs (5 bits)	rt (5 bits)	offset (16 bits)

Champ	Valeur
Opcode	1
rs	<adresse du premier registre>
rt	<adresse du deuxième registre>
offset	<division par 4 de la différence entre l'adresse de l'étiquette de la branche et PC+4>

Dans ce cas-ci, l'ALU doit effectuer une soustraction entre les valeurs stockées dans *rs* et *rt*. Si le résultat n'est pas négatif, le PC doit recevoir la valeur d'offset plus (PC+4).

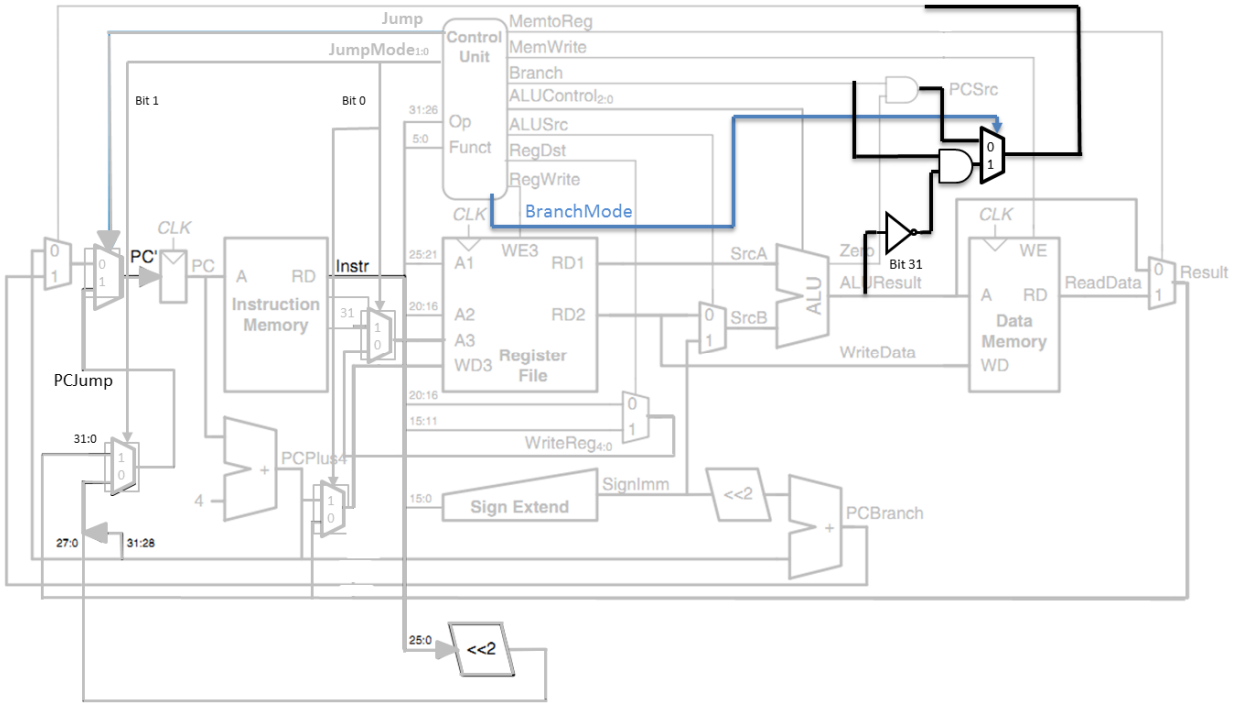


Figure 3 - Microarchitecture avec l'implémentation de la fonction bge

3. Table de vérité pour les nouvelles instructions

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	BranchMode	MemWrite	Jump	JumpMode	MemtoReg	ALUOp
li	001000	1	0	1	0	X	0	0	XX	0	00
mul	000000	1	1	0	0	X	0	0	XX	0	1X
jal	000011	1	X	X	0	X	0	1	01	0	XX
jr	000000	0	X	0	0	X	0	1	1X	0	00
bge	000001	0	X	0	1	1	0	0	XX	0	01

ALUOp	Funct	ALUControl
1X	11000 (mul)	011 (multiplication)[1]

4. Testbench

Un code de *testbench* a été développé pour vérifier si les nouvelles instructions fonctionnent comme prévu et pour vérifier si les changements apportés à la microarchitecture n'ont pas brisé les anciennes fonctionnalités déjà implémentés.

La figure suivante contient la séquence d'instructions MIPS qui ont été utilisés pendent les vérifications de la architecture. Le fichier de mémoire d'instructions (*imem.vhdl*) contient les valeurs chargées par les registres d'instructions (voir colonne « Machine »).

[1] Fonction à être ajouté dans l'ALU

```
# mipstest.asm
# David_Harris@hmc.edu 9 November 2005
# Modified by Fabiano Batista, 12 April 2013
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j, jal, jr, mul, bge
# If successful, it should write the value 132 to address 84

#          Assembly                Description                  Address  Machine
main:      li   $2, 5              # initialize $2 = 5            0        20020005
           li   $3, 12             # initialize $3 = 12           4        2003000c
           addi $7, $3, -9         # initialize $7 = 3            8        2067fff7
           or   $4, $7, $2         # $4 <= 3 or 5 = 7             c        00e22025
           and  $5, $3, $4         # $5 <= 12 and 7 = 4           10       00642824
           add  $5, $5, $4         # $5 = 4 + 7 = 11              14       00a42820
           beq  $5, $7, end        # shouldn't be taken           18       10a7000a
           slt  $4, $3, $4         # $4 = 12 < 7 = 0              1c       0064202a
           beq  $4, $0, around     # should be taken              20       10800001
           addi $5, $0, 0          # shouldn't happen             24       20050000
around:    slt  $4, $7, $2         # $4 = 3 < 5 = 1               28       00e2202a
           add  $7, $4, $5         # $7 = 1 + 11 = 12             2c       00853820
           sub  $7, $7, $2         # $7 = 12 - 5 = 7              30       00e23822
           sw   $7, 68($3)         # [80] = 7                     34       ac670044
           lw   $2, 80($0)         # $2 = [80] = 7                38       8c020050
           bge  $4, $2, end        # shouldn't be taken ($4<$2)   3c       0482000a
           jal  sub                # jump to branch "sub" and link 40      0c000013
           bge  $2, $5, sub2       # should be taken ($2>$5)      44       04450004
           addi $2, $0, 1          # shouldn't happen             48       20020001
sub:       mul  $2, $5, $3         # $2 = 11 * 12 = 132           4c       00a31030
           jr   $31                # return to the address 44     50       17E00000
           addi $2, $0, 1          # shouldn't happen             54       20020001
sub2:      bge  $2, $2, sub3       # should be taken ($2=$2)      58       04420001
           addi $2, $0, 1          # shouldn't happen             5c       20020001
sub3:      j    end                # should be taken              60       0800001a
           addi $2, $0, 1          # shouldn't happen             64       20020001
end:       sw   $2, 84($0)         # write adr 84 = 132           68       ac020054
```

Figure 4 - Code MIPS pour vérification de la microarchitecture

Le code VHDL du *testbench* (testbench.vhdl) vérifie si l'exécution du code MIPS génère le résultat attendu. Dans ce cas-ci, la valeur 132 doit être écrite dans l'adresse de mémoire 84.

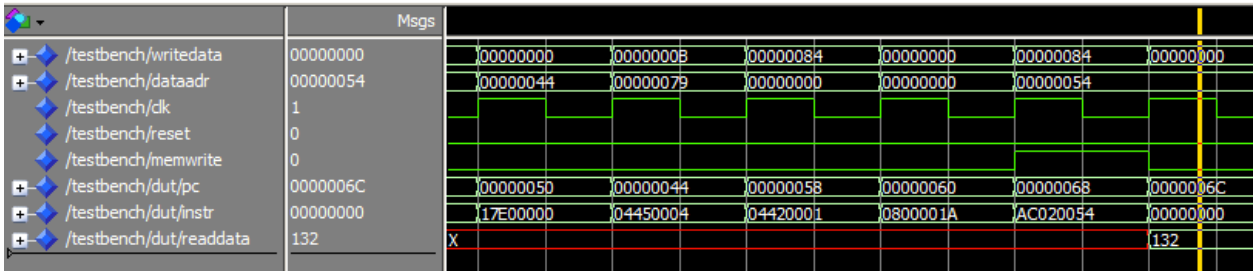


Figure 5 - Analyse de signaux du testbench

Le *testbench* a été exécuté avec ModelSim, et c'était constaté que l'implémentation a réussi le test.

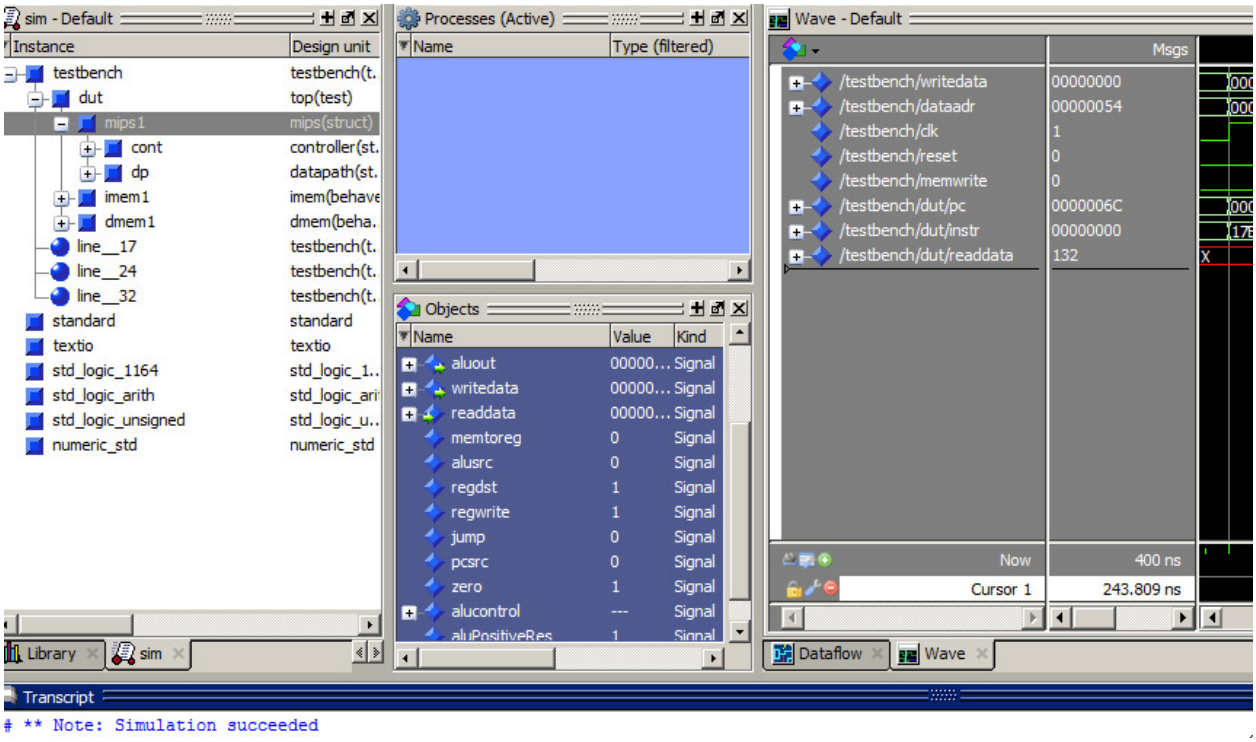


Figure 6 - Résultat de test ("Simulation succeeded")

Deux autres fichiers de testbench (alu_tb.vhdl et imem_tb.vhdl) ont été implémentés pour la vérification les fonctionnalités de l'alu et de la mémoire d'instructions respectivement.

5. Code VHDL

Le code originel de la microarchitecture proposé dans la section 7.6 du livre « Digital Design and Computer Architecture » de D. M. and S. Harris a été modifié pour implémenter les nouvelles instructions, en incluant les fichiers de *testbench*. Les fichiers VHDL correspondants sont inclus dans le fichier MIPSVHDL-BATL24077305.zip fourni avec ce document.