SIMULATION D'UNE MONTRE DIGITALE

C. LALANNE, H. YU-GUAN, L. SANCHEZ, F. STRANSKY

Introduction

Le but de ce travail est de construire un programme qui modélise un processeur sur lequel tourne un programme imitant une montre digitale. Ce rapport accompagne le simulateur.

Table des matières

1. Simulateur

Le simulateur fonctionne en OCaml. Le programme de simulation utilise le code netlist traité par le parseur et exécute un cycle de simulateur à chaque fois qu'il est appelé ; modifiant les données en RAM à chaque fois. Comme les registres dans MiniJazz ne fonctionnent pas pour des nappes de fils, nous avons utilisé des mises en RAM et des chargements pour modéliser les registres - les 128 premiers octets de la RAM servent à cela. Ces registres sont initialisés à 0.
Un programme au-dessus du simulateur fait tourner ce programme de simulation, et nous ressort les valeurs stockées en RAM : ces valeurs

2. Assembleur

Nous avons choisi de faire un assembleur extrêmement simple, codé en python : il substitue les instructions en pseudocode par leur code binaire. En particulier, il faut indiquer manuellement les registres utilisés pour charger une variable. Comme notre code n'utilise

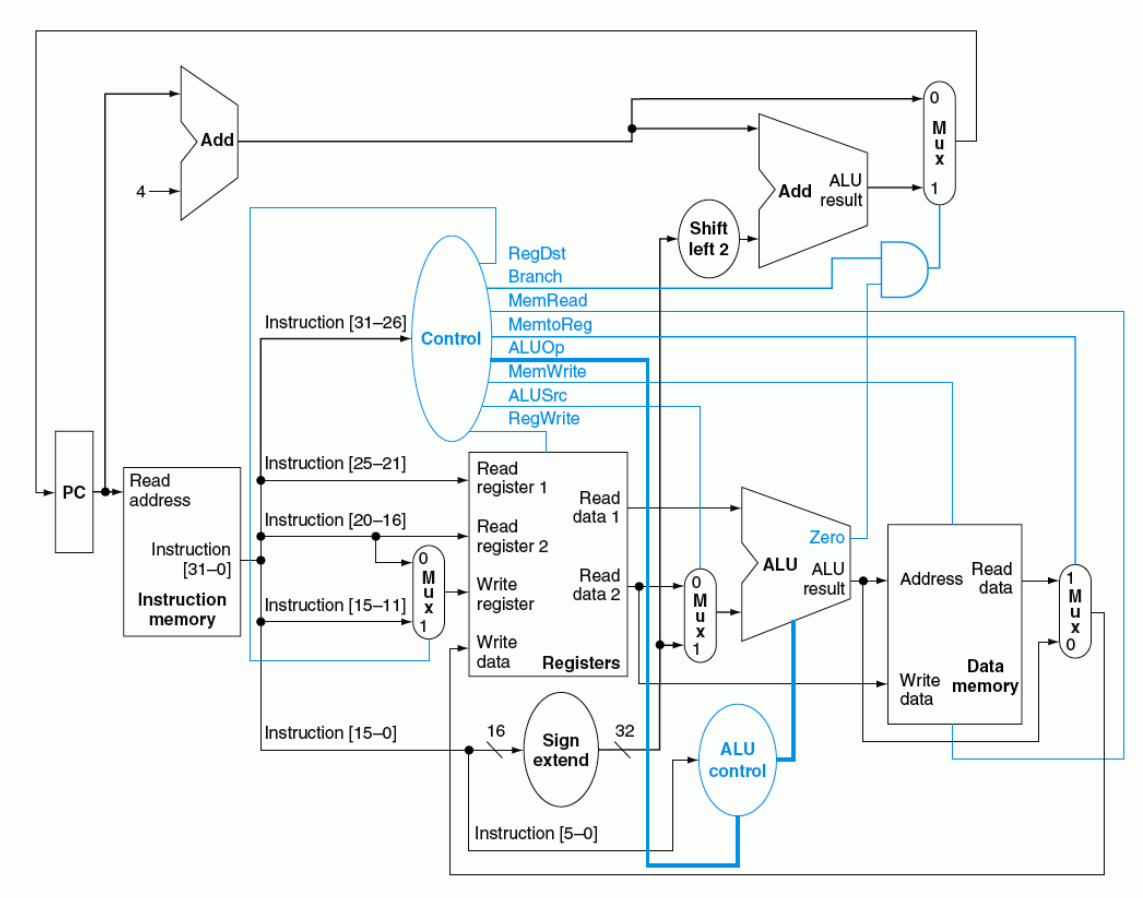
pas plus de variables que le processeur ne contient de registres, nous n'avons pas besoin de sauvegarder des valeurs dans la mémoire - nous n'utilisons que des registres.

3. Le processeur

Nous utilisons comme base du processeur celui décrit dans "Computer Organization and Design", de David Patterson John Henessy, au chapitre 5. Nous avons choisi un processeur 32-bits avec 32 registres. Notre processeur est donc constitué des parties suivantes :

- Une ALU gérant les opérations suivantes : addition, soustraction, ET bit-par-bit, OU bit-par-bit, égalité numérique et supérieur ou égal numérique. Elle a deux sorties : une nappe de 32 fils pour le résultat des calculs et un fil d'erreur.
- Une mémoire RAM.
- Une banque de registres. Cette banque contient 32 registres (modélisés en RAM) de 32 bits.
- Un code, d'instructions 32 bit, écrites dans la ROM.
- Un circuit de contrôle (voir plus loin).
- Un circuit de contrôle de l'ALU.
- Un compteur d'instructions.

Le schéma suivant, tiré du livre susnommé, illustre notre structure :



4. Instructions

Nous utilisons les instructions suivantes, que nous avons choisies sur la fiche technique d'un processeur MIPS. Voici les instructions retenues, avec leurs explications (advance_pc(x) incrémente PC de 8*x) :

- ADD
  - Description : Adds two registers and stores the result in a register
  - Operation : $d = $s + $t ; advance_pc (4) ;
  - Syntax : add $d, $s, $t
  - Encoding : 0000 00ss ssst tttt dddd d000 0010 0000
- ADDI
  - Description : Adds a register and a sign-extended immediate value and stores the result in a register
  - Operation : $t = $s + imm ; advance_pc (4) ;
  - Syntax : addi $t, $s, imm
  - Encoding : 0010 00ss ssst tttt iiii iiii iiii iiii

- BEQ
  - Description : Branches if the two registers are equal
  - Operation : if $s == $t advance_pc (offset ¡¡ 2)) ; else advance_pc (4) ;
  - Syntax : beq $s, $t, offset
  - Encoding : 0001 00ss ssst tttt iiii iiii iiii iiii
- BGEZ
  - Description : Branches if the register is greater than or equal to zero
  - if $s ¿= 0 advance_pc (offset ¡¡ 2)) ; else advance_pc (4) ;
  - Syntax : bgez $s, offset
  - Encoding : 0000 01ss sss0 0001 iiii iiii iiii iiii
- J
  - Description : Jumps to the calculated address
  - Operation : PC = addr(target ¡¡ 2) ;
  - Syntax : j target
  - Encoding : 0000 10ii iiii iiii iiii iiii iiii iiii
- LW
  - Description : A word is loaded into a register from the specified address.
  - Operation : $t = MEM[$s + offset] ; advance_pc (4) ;
  - Syntax : lw $t, offset($s)
  - Encoding : 1000 11ss ssst tttt iiii iiii iiii iiii
- SUB
  - Description : Subtracts two registers and stores the result in a register
  - Operation : $d = $s - $t ; advance_pc (4) ;
  - Syntax : sub $d, $s, $t
  - Encoding : 0000 00ss ssst tttt dddd d000 0010 0010
- SW
  - Description : The contents of $t is stored at the specified address.
  - Operation : MEM[$s + offset] = $t ; advance_pc (4) ;
  - Syntax : sw $t, offset($s)
  - Encoding : 1010 11ss ssst tttt iiii iiii iiii iiii

La table qui suit indique les valeurs des paramètres de contrôle pour chaque instruction :

Opération	RegDst	Jump	Branch	MemToReg	ALUop	MemWrite	AluSrc	RegWrite
ADD	1	0	0	0	00	0	0	1
ADDI	0	0	0	0	01	0	1	1
BEQ	0	0	1	0	10	0	0	0
BGEZ	0	0	1	0	11	0	0	0
J	0	1	0	0	00	0	0	0
LW	0	0	0	1	01	0	1	1
SUB	1	0	0	0	00	0	0	1
SW	0	0	0	0	01	1	1	0
AND	1	0	0	0	00	0	0	1
OR	1	0	0	0	00	0	0	1

4.1. **Paramètres de contrôle.** Voici les paramètres de contrôle utilisés par le processeur :

RegDst: S'il vaut 1, l'adresse du second registre se trouve dans les bits 11-15, sinon il est dans les bits 16-20.

Jump: Contrôle le MUX de saut lors des instructions de JUMP et JAL.

Branch: Contrôle le MUX de branchement.

MemToReg: S'il vaut 1, les registres sont chargés depuis la RAM plutôt que depuis l'ALU.

ALUop: Ces deux bits contrôlent le type d'opération que fait l'ALU.

MemWrite: S'il faut écrire dans la mémoire.

AluSrc: Si vaut 1, le second opérande de l'ALU vient des bits d'immédiat, sinon il vient de la seconde sortie de la banque de registres.

RegWrite: Write-Enable sur les Registres.

4.2. **Instructions de l'ALU.** L'ALU exécute les instructions suivantes en fonction du code ALUop :

Code ALUop	Instruction
00	Selon derniers bits de l'instruction
01	Addition
11	Supérieur ou égal
10	Egalité

5. Le code

5.1. **Principe.** Après avoir initialisé les registres à une date de départ contenue en mémoire, le programme tourne en boucle en vérifiant la valeur contenue dans R30. Si cette valeur est négative, le programme boucle sur lui-même, ne changeant rien. Si la valeur est positive, le programme considère qu'une seconde a passé. Il incrémente le compteur de secondes de 1, et déclenche une cascade de comparaisons pour déterminer s'il faut incrémenter les minutes, les heures, etc.

5.2. **Décompte des jours par mois.** Le problème ici est que le nombre de jours par mois varie énormément. Pour résoudre ce problème, on charge en RAM deux tables contenant les durées des mois dans l'ordre (1 = janvier, 2 = février, etc...) Au moment du passage d'un jour, le programme recherche dans la RAM la durée du mois en cours (située à l'adresse I + 4*mois) et compare cette durée au numéro du jour.

5.3. **Nature bissextile des années.** Pour tenir compte de la durée de Février, on a quatre compteurs : un compteur $\$bis$ de période 4, un compteur $\$cen$ de période 100 et un compteur $\$fou$ de période 400. Ces compteurs sont mis à jour à chaque changement d'année.

```
LW R0 $sec $in0
LW R0 $min $in1
LW R0 $hrs $in2
LW R0 $wek $in3
LW R0 $day $in4
LW R0 $mon $in5
LW R0 $yea $in6

LW R0 $bis $in7
LW R0 $cen $in8
LW R0 $fou $in9

ADDI R0 $60 60
ADDI R0 $24 24
ADDI R0 $7 7
ADDI R0 $12 12
ADDI R0 $4 4
ADDI R0 $100 100
ADDI R0 $400 400

BEQ $clk R0 2
#17e instruction
J 17

LW R0 $clk $mmm
ADDI $sec $sec 1
BEQ $sec $60 2
J 17

LW R0 $sec $zer
ADDI $min $min 1
# 24e instruction
BEQ $min $60 2
J 17

LW R0 $min $zer
ADDI $hrs $hrs 1
BEQ $hrs $24 2
# 29e instruction
J 17
```

```
LW R0 $hrs $zer
ADDI $day $day 1
ADDI $wek $wek 1
BEQ $wek $7 2
J 37
# 35e instruction
LW R0 $zer $wek

# Longueur du mois
ADD $mon $mon $x
ADD $x $x $x
# x = 4*mon
BEQ $bis R0 2
J 46
# 40e instruction
BEQ $cen R0 2
J 48
BEQ $fou R0 2
J 46
J 48

LW $x $lmo $aaa
J 31
LW $x $lmo $bbb

# Mois - 48e instruction
BEQ $day $lmo 2
J 17

LW R0 $day $zer
ADDI $mon $mon 1
BEQ $mon $12 2
J 17

LW R0 $mon $zer
ADDI $yea $yea 1
ADDI $bis $bis 1
ADDI $cen $cen 1
ADDI $fou $fou 1
```

```
# 59e instruction
BEQ $bis $4 2
J 17
LW R0 $bis $zer

BEQ $cen $100 2
J 17
LW R0 $cen $zer

BEQ $fou $400 2
J 17
LW R0 $fou $zer

J 17
```