POUILLART Julien Licence 2

BOISTEAU-DESDEVISES Erwan Groupe : 585K

UNIVERSITÉ DE NANTES

Projet : Réalisation du processeur Fifth

Architecture des ordinateurs

X31I050

Université de Nantes Année universitaire 2021 - 2022

1) La pile

La pile est composée de quatre entrées, dont une valeur SPsel (sur 2 bits) déterminant quelle modification doit être effectuée sur le registre SP concerné, et une valeur Wsel (sur 2 bits également) déterminant à quel registre sera attitré la nouvelle valeur.

Nous pouvons déterminer ces valeurs en fonction des instructions effectuées dans les opérations qui nous intéressent. Nous obtenons ainsi la table de vérité suivante :

operation	opcode	SPsel	Wsel	SPcsel	Wcsel
add	000000	01	10	10	11
sub	000001	01	10	10	11
neg	000010	10	00	10	11
mul	000011	01	10	10	11
div	000100	01	10	10	11
rem	000101	01	10	10	11
not	000110	10	00	10	11
and	000111	01	10	10	11
or	001000	01	10	10	11
shl	001001	01	10	10	11
sra	001010	01	10	10	11
lt	001011	01	10	10	11
eq	001100	01	10	10	11
gt	001101	01	10	10	11
jmp *adr*	001110	10	11	10	11
jmpt *adr*	001111	01	11	10	11
jmpf *adr*	010000	01	11	10	11
push *val*	010001	11	01	10	11
push [*adr*]	010010	11	01	10	11
pop	010011	01	11	10	11
pop [*adr*]	010100	01	11	10	11
call *adr*	010101	10	11	11	01
ret	010110	10	11	01	11
dup	010111	11	01	10	11
halt	011000	00	11	00	11

Remarque : SPsel et Wsel correspondent aux valeurs pour une pile de données tandis que SPcsel et Wcsel sont attribuées à une pile d'appels.

Petite difficulté rencontrée : lors de notre première version de la pile, nous n'avions pas pensé à insérer un démultiplexeur pour choisir le registre dans lequel l'écriture se fait, pourtant essentiel pour éviter d'écrire dans tous les registres à chaque fois (comme c'était le cas dans notre première version). Ce problème étant résolu, notre pile fonctionne.

2) L'Unité Arithmétique Logique

Avant d'implémenter notre UAL, la première étape était de déterminer les valeurs des signaux qui nous intéressent ici (Overflow Flag, Sign Flag et Zero Flag).

En effet, l'Overflow Flag sera levé en cas de changement anormal de signe (on observe le bit de poids fort). Observons les conséquences dans le cadre de l'addition et de la soustraction :

- 010 + 011 = 101
  Dans cet exemple, les bits de poids fort de nos deux termes valent 0, pourtant le bit de poids fort du résultat passe à 1 : l'OF est levé.

- 110 + 101 = 1011
  Ici, les bits de poids fort a2 des termes valent 1, tandis que le bit a2 du résultat vaut 0 : une nouvelle fois, l'OF est levé.

On en déduit la formule suivante pour déterminer l'Overflow Flag pour une addition :

- **OF = an bn ¬cn + ¬an ¬bn cn**

Qu’en est-il pour la soustraction ?

- 100 - 011 = 001
  Concentrons-nous sur le bit de poids fort : 1-0 = 0. Dans ce cas, naturellement, l’Overflow Flag est levé.

- 011 - 100 = 1111
  Regardons à nouveau le bit de poids fort a2 des termes : 0 - 1 = 1. Cela sous-entend qu’en soustrayant un nombre positif à un nombre négatif, nous obtenons un nombre négatif, ce qui est incohérent. L’Overflow Flag est donc levé.

On obtient alors la formule suivante pour déterminer l’Overflow Flag d’une soustraction :

- **OF = an ¬bn ¬cn + ¬an bn cn**

Les deux autres indicateurs qui nous intéressent sont en effet assez simples à déterminer : le Sign Flag est positionné si le résultat est négatif, et le Zero Flag est levé si le résultat obtenu est nul.

3) Le testeur

Pour implémenter le module de test, nous nous intéressons à 3 instructions : lt (lower than), eq (equal) et gt (greater than).

Nous avons étudié précédemment les signaux OF, SF et ZF. Nous avons donc déduit une table de vérité à partir de ces signaux pour déterminer le résultat de nos instructions de test :

OF	SF	ZF	lt	eq	gt
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	0	0	1
1	1	1	0	0	0

Pour illustrer notre table, voici un exemple simple : l'instruction eq.
En effet, pour que ce test soit vrai, les 2 valeurs que nous comparons doivent être égales. Autrement dit, étant donné que les tests sont obtenus à partir d'une soustraction des 2 valeurs, le résultat obtenu doit être nul. Autrement dit, eq retournerait 1 dès lors que le Zero Flag (ZF) est activé. Or, nous pouvons observer des contradictions : en effet, le Zero Flag et le Sign Flag ne peuvent être activés en même temps (0 n'étant pas une valeur négative). Il en va de même pour le cas du Zero Flag et de l'Overflow Flag activés en même temps : nous ne devrions pas observer de changement anormal de signe dans le cas d'une différence de 2 termes identiques. Nous en déduisons ainsi une formule très simple pour l'instruction eq :

- **eq = ¬OF ¬SF ZF**

En suivant le même type de raisonnement, nous obtenons les formules pour les instructions lt et gt :

- **lt = ¬ZF (¬OF SF + OF ¬SF) = ¬ZF (OF XOR SF)**
- **gt = ¬ZF (¬OF ¬SF + OF SF)**

Nous avons donc implémenté les circuits associés à ces formules et ajouté les multiplexeurs nécessaires pour sélectionner le test désiré.

4) Le module de mise à jour du PC

Ce module prend en entrée la valeur actuelle de PC, la valeur du dessus de Stackc, une adresse arbitraire, ainsi que le signal de modification sur 2 bits et un booléen représentant une condition de saut. A partir des ces informations, il permet à PC de s'actualiser et de pointer vers la prochaine ligne de code à exécuter. Ces informations sont traitées de la manière suivante :

- Le signal de modification choisit parmi les 4 modes de modification (jump, jump conditionnel, retour / fin de fonction, incrémentation)

- Dans le cas d'un saut conditionnel, le booléen détermine si le saut doit être effectué. Attention : il ne s'agit pas de la valeur du test effectué, mais bien de si le saut doit être effectué ou non.

- Modification de PC en fonction du signal de modification :

PCupdateSig	Nouvelle valeur de PC
00	adresse arbitraire (saut inconditionnel)
01	adresses arbitraire / PC + 1 (saut conditionnel)
10	adresse du dessus de Stackc (retour de fonction)
11	PC + 1 (incrémentation)

Indépendamment des ces modifications, ce module donne aussi toujours la valeur de PC + 1, utilisée par la pile d'adresses en cas d'appel de fonction.

5) Le contrôleur

Afin de pouvoir récupérer les signaux utilisés dans le circuit à partir d'un opcode, le contrôleur utilise un multiplexeur. Chaque entrée de ce multiplexeur correspond à un opcode, et est composée d'un mot constant de 20 bits. Le contrôleur fait ensuite correspondre chaque signal la sous-partie du mot de 20 bits.

Ainsi, chaque mots de 20 bits correspond à la concaténation des signaux tels que présentés dans le tableau ci-dessous, de gauche à droite.

operation	opcode	SPsel	Wsel	SPcsel	Wcsel	RamL	RamW	jumpIsTrue	PCupdateSig	StackInputSel	Op
add	000000	01	10	10	11	0	0	0	11	010	0000
sub	000001	01	10	10	11	0	0	0	11	010	0001
neg	000010	10	00	10	11	0	0	0	11	010	0010
mul	000011	01	10	10	11	0	0	0	11	010	0011
div	000100	01	10	10	11	0	0	0	11	010	0100
rem	000101	01	10	10	11	0	0	0	11	010	0101
not	000110	10	00	10	11	0	0	0	11	010	0110
and	000111	01	10	10	11	0	0	0	11	010	0111
or	001000	01	10	10	11	0	0	0	11	010	1000
shl	001001	10	00	10	11	0	0	0	11	010	1001
sra	001010	10	00	10	11	0	0	0	11	010	1010
lt	001011	01	10	10	11	0	0	0	11	011	0001
eq	001100	01	10	10	11	0	0	0	11	011	0001
gt	001101	01	10	10	11	0	0	0	11	011	0001
jmp *adr*	001110	10	11	10	11	0	0	0	00	000	0000
jmpt *adr*	001111	01	11	10	11	0	0	1	01	000	0000
jmpf *adr*	010000	01	11	10	11	0	0	0	01	000	0000
push *val*	010001	11	01	10	11	0	0	0	11	100	0000
push [*adr*]	010010	11	01	10	11	1	0	0	11	001	0000
pop	010011	01	11	10	11	0	0	0	11	000	0000
pop [*adr*]	010100	01	11	10	11	0	1	0	11	000	0000
call *adr*	010101	10	11	11	01	0	0	0	00	000	0000
ret	010110	10	11	01	11	0	0	0	10	000	0000

dup	010111	11	01	10	11	0	0	0	11	000	0000
halt	011000	00	11	00	11	0	0	0	00	000	0000

Pour rappel,

SPsel		Wsel	
00	Reset	00	Stack[SP]
01	++ (pop)	01	Stack[SP-1]
10	no modif	10	Stack[SP+1]
11	-- (push)	11	no write

Regardons maintenant un exemple : add

operation	opcode	SPsel	Wsel	SPcsel	Wcsel	RamL	RamW	jumpIsTrue	PCupdateSig	StackInputSel	Op
add	000000	01	10	10	11	0	0	0	11	010	0000

Pour l'opération add :

- On écrit et se déplace sur la case SP+1 (SPsel = 01 et Wsel = 10)
- On ne modifie pas la pile d'appels (SPcsel == 10 et Wcsel = 11)
- On ne souhaite pas écrire, ni lire la mémoire de données (RamL = 0 & RamW = 0)
- Il n'est pas question de saut conditionnel (jumpIsTrue = 0)
- Suite à cette opération PC sera incrémenté pour exécuter l'instruction suivant ce add (PCupdateSig = 11)
- La valeur à empiler est le résultat d'une opération de l'UAL (StackInputSel = 010)
- On effectue une addition (Op = 0000)

6) Le Fifth

Pour tester notre processeur Fifth, nous avons décidé d'implémenter une fonction permettant de calculer le nième élément de la suite de fibonacci.

Version C++ :

```
int main() {

   int c;

   int n = 5;

   int a = 1;

   int b = 1;

   for(i = 1; i < n-1; i++) {

      c = a + b;

      a = b;

      b = c;

   }



   return 0;

}
```

Version assembleur Fifth :

//[0000] = n

//[0001] = i

//[0002] = a

//[0003] = b

//[0004] = c

//main

```
push 9
pop [0000]

push 1
pop [0001]

push 1
pop[0002]

push 1
pop [0003]

push 0
pop [0004]

//for
push [0001]
push [0000]
push 1
sub
lt
jmpf endfor

//do
```

```
push [0002]
push [0003]
add
pop [0004]

push [0003]
pop [0002]

push [0004]
pop [0003]

push [0001]
push 1
add
pop [0001]

call for
//endfor

halt
```

Version machine Fifth :

0x4409

0x5000

0x4401

0x5001

0x4401

0x5002

0x4401

0x5003

0x4400

0x5004

//for

0x4801

0x4800

0x4401

0x0400

0x2c00

0x401d

//do

0x4802

0x4803

0x0000

0x5004

0x4803

0x5002

0x4804

0x5003

0x4801

0x4401

0x0000

0x5001

0x540a

//endfor

0x6000

//A la fin du programme, le résultat est stocké dans MEM[0004]