author:

- Matthieu BREUIL (e2102555)

Notes

Le projet suivant a été réalisé sur logisim 2.16.1.4 .exe sur Windows 11. Attention, lorsque vous suivrez la partie **Faire fonctionner le microprocesseur**, il faudra bien suivre les clics proposés : en effet, on pourrait se dire qu'après avoir fait tourner le programme, cliquer immédiatement sur le circuit noté **RAM** dans la barre des circuits nous montrera la ram active, mais non, il faut repasser par le circuit **microprocesseur** dans tous les cas pour observer l'état de n'importe quel circuit.

Fonctionnement microprocesseur

Unité logique 8 bits

Dans cette partie, on nous demande de créér des opérateurs logiques (and, or, xor) sur 8 bits. Pour ce faire et par soucis de gain de temps, j'ai d'abord créé les circuits sur deux bits, puis j'ai copié le ciruit de deux bits pour celui du quatre bits en changeant les portes logiques par celles créées préceddement, et j'en ai fait de même pour celui de 8 bits. J'ai aussi implémenté le not, le nand, le nor et le nxor : en effet, en regardant un peu la suite, on va avoir à faire un additionneur : mes opérations logiques plus l'additionneur font un total de 8 opérationsn soit 2^3 ce qui sera sympathique pour le choix du multiplexeur dans la question de l'UAL.

Additionneur 8 bits

Dans cette partie, on va commencer par réaliser un additionneur 1 bit, puis en chainer 8 en propageant la retenue afin de faire un additionneur 8 bits. Cet additionneur à propagation de retenue est une implémentation naïve : en se renseignant sur internet, on comprend que de nombreuses méthodes peuvent éviter d'attendre **n** cycles pour un additionneur **n** bits, en effet, l'additionneur n°2 nécessitant la carry de l'additionneur n°1, il doit attendre que celui ci ai fini. De manière générale, l'additionneur **n** devra attendre que l'additionneur **n-1** ai fini, pour tout n>1. On pourra implémenter un meilleur additionneur si le temps et la charge de travail dans les autres matières nous le permet. Pour implémenter cet additionneur 1 bit, voici notre table de vérité :

RIN	A	B	ROUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ce qui nous donne les équations suivantes :

```
ROUT = A.B.not(RIN) + not(A).B.RIN + A.not(B).RIN + A.B.RIN
     = A.B.(not(RIN) + RIN) + not(A).B.RIN + A.not(B).RIN    // Factorisation par A.B
     = A.B + not(A).B.RIN + A.not(B).RIN                     // Réduction de not(a) + a en 1
     = A.B + RIN.(not(A).B + A.not(B))                       // Factorisation par RIN
```

On va poser la table de `not(A).B + A.not(B)` :

A	B	`not(A).B + A.not(B)`
0	0	0
0	1	1
1	0	1
1	1	0

On se rend ainsi compte que `not(A).B + A.not(B) == A XOR B`

```
ROUT = A.B + RIN.(A XOR B) // Substitution par A XOR B
```

Pour **S** on va procéder différemment et se rendre compte que nous avons un 1 en sortie quand un nombre impair de variables en entrée sont à 1 : autrement dit, nous avons une addition modulo 2. L'addition modulo 2 se traduit en algèbre de bool par une porte xor : donc `S = A XOR B XOR RIN`

On retient ainsi les équations suivantes :

```
ROUT = A.B + RIN.(A XOR B)
S    = A XOR B XOR RIN
```

Ce qui fait un additionneur en 5 portes logiques (2x XOR, 2x AND, 1x OR):

- A.B (utilisé 1 fois)
- A XOR B (utilisé 2 fois)
- (A XOR B) XOR RIN (utilisé 1 fois)
- RIN.(A XOR B) (utilisé 1 fois)
- (A.B) + (RIN.(A XOR B)) (utilisé 1 fois)

Conception de l'UAL

Dans cette partie, on va créér une UAL prenant deux opérandes 8 bits et en ressortant une des opérations suivantes :

- add
- and
- or
- nand
- nor
- xor
- nxor
- not

Comme on a 8 opérations, on va utiliser un multiplexeur prenant 3 bits de contrôle (2^3 == 8) et prenant des opérandes de 8 bits en entrée et en sortie. Les signaux de contrôle correspondent à l'ordre de la liste précédente (000 pour add, 001 pour and, ...). La manière dont notre UAL est implémentée fait qu'à chaque fois que celle-ci est solicitée elle utilise toutes les opérations disponibles sur les opérandes, et choisie la bonne en sortie à l'aide du plexeur. Ce n'est pas très réaliste car à chaquer cycle, une seule opération peut être retournée par notre UAL, néanmoins, cette implémentation est suffisante pour la suite de notre microprocesseur.

Conception du banc de registres

Dans un premier temps, nous allons concevoir des registres de 8 bits en les composants de 8 balances. En entrée de notre registre nous prenons une valeur de 8 bits à écrire et un bit d'enable activant justement l'écriture. En sortie nous retournons la valeur du registre 8 bits. Pour notre banc de registres, nous posons 4 registres derrière deux plexeurs de 2 bits de signal (2^2 == 4) retournant les valeurs des registres choisis. En entrée de ce banc, nous avons un signal de 7 bits : le premier activant ou non l'écriture dans les registres, les deux suivants choisissant le registre dans lequel écrire si le premier bit est 1, et les deux fois deux suivants décrivent les registres dont on souhaite les valeurs en sortie du banc de registres. La deuxième entrée de ce banc est une valeur de 8 bits à écrire (ou non selon le bit d'enable) dans un registre. En sortie, deux valeurs de 8 bits contenues dans les registres choisis. Il est possible de réaliser ce banc avec un signal de 6 bits en retirant le bit d'enable, auquel cas nous devons écrire dans un registre à chaque cycle, et ainsi lors d'opérations de lectures telles qu'un **STORE** nous devrons obligatoirement reboucler le contenu d'un registre dans lui même : nécessitant un peu de travail inutile car un bit d'enable dans notre signal est plus simple à gérer de mon point de vue.

Conception d'une unité de traitement complète

Rien à signaler, en remplissant les registres au préalable et en testant le rebouclage après passage dans l'ual tout fonctionne.

Intégration d'une mémoire programme et d'un pointeur programme (PC)

Dans cette partie, nous allons définir complètement notre jeu d'instructions afin de définir le nombre de bits à allouer par instructions (constant car nous implémentons un RISC).

Opérations logiques

- and r1, r2, r3
- nand r1, r2, r3
- or r1, r2, r3
- nor r1, r2, r3
- xor r1, r2, r3
- nxor r1, r2, r3
- not r1, r2

Opérations mathématiques

- add r1, r2, r3

Opérations autres

- load r1, const
- load r1, from addr

- mov r1, r2
- store r1, r2
- store addr, r1
- jmp addr
- jmp r1
- jz r1, r2
- jnz r1, r2

17 opérations : 5 bits car 2^5 = 32 (15 opérations de rab)

```
| 0-4   | 5-15    |
| instr | payload |
|       | 5-6     | 7-8     | 9-10    |
|       | R1      | R2      | R3      |

--> Largeur d'un mot d'instruction == 16 bits == 2 octets

00000 => add
00001 => and
00010 => or
00011 => nand
00100 => nor
00101 => xor
00110 => nxor
00111 => not

01000 => load r1, const
01001 => load r1, from addr
01010 => mov r1, r2
01011 => store r1, r2
01100 => store addr, r1

10000 => jmp addr
10001 => jmp r1
10010 => jz  r1, r2
10011 => jnz r1, r2
```

Conception du circuit du PC

Nous allons donc paramétrer une ROM avec comme largeur de mots 16 bits. Nous adresserons 2^8 mots dans cette ROM. Ainsi, notre PC est un registre de 8 bits. Etant donné que nous avons des opérations de saut, nous devons prévoir deux mécanismes pour modifier ce PC : le premier sera une incrémentation du PC par 1 si une opération autre qu'un jump est réalisée, le second sera l'attribution au PC d'une valeur précise si l'opération était un jump. Ainsi, notre circuit aura deux entrées, à savoir la nouvelle adresse du PC et un bit d'enable indiquant si cette nouvelle adresse doit être écrite dans le PC. Ce bit servira de signal à un plexeur de deux valeurs permettant de choisir entre l'adresse incrémenté par un à l'aide de notre additionneur 8 bits ou l'adresse passée par l'unité de décodage des jumps. En sortie, nous retournons les 16 bits de l'instruction pointée par le PC.

Conception de nos circuits de décodage de l'instruction

L'instruction de 16 bits est envoyé sur un premier décodeur qui consiste simplement à splitter nos 16 bits en 3 groupes :

- les 2 bits de poids forts décrivent si l'instruction est de type math/logique (00), load/store (01) ou jump (10).
- les 3 suivants décrivent l'opération à effectuer (voir ci-dessus) et les 11 suivants composent le payload propre à chaque instruction.

Les 11 bits du payload seront traité comme suit selon les opérandes nécessaires à l'instruction :

```
| r1    | r2    | r3    | inutile |
| 2bits | 2bits | 2bits | 5bits   |
OU
| r1    | r2    | inutile |
| 2bits | 2bits | 7bits   |
OU
| r1    | const ou addr | inutile | // Ce cas comprend les opérations de type: instr r1, const ET instr const, r1
| 2bits | 8bits         | 1bit    |
OU
| r1    | inutile |
| 2bits | 9bits   |
OU
| const ou addr | inutile |
| 8bits         | 3bits   |
```

Nous allons ensuite rediriger tout ces bits sur deux décodeurs spécifiques : celui des opérations maths/logiques et celui des opérations load/store. Pour le décodeur des maths, c'est une fois de plus uniquement un splitter car toutes les opérations sont de la forme r1, r2, r3 donc le signal de contrôle du banc de registre peut être généré de la même manière pour toutes les opérations. Pour le décodeur des loads/store, c'est aussi un mélange de splitters et de différents multiplexeurs permettant nottamment de controler quel est la source du signal à load/store (RAM/raw/registre...). Les signaux générés par les unités de décodage sont ensuite plexés par les circuits nommés **which_XXX** et redirigés vers l'UAL et le banc de registres.

Intégration d'une mémoire de données

Rien à signaler, nous branchons la RAM aux signaux générés par l'unité de gestion des instructions load/store.

Instructions de branchement

Nous créeons un circuit de décodage qui va aller modifier le PC quand :

is_instr_jmp	is_simple_jmp	is_zero_jmp	is_nzero_jmp	modify
1	0	0	0	0
1	0	0	1	IF !0
1	0	1	0	IF 0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Nous ajoutons quelques plexeurs afin de déterminer la source du registre à vérifier et la source de l'adresse à écrire, et notre décodeur est complet : prêt à être câblé sur notre pointeur de programme.

Ecriture d'un programme de test

Nous allons coder la somme des 4 premiers entiers, voici son code en python sachant que nous ne possédons que l'opération '+' en arithmétique et ne pouvons faire que des comparaisons à 0:

```
def sum():
    ret  = 0
    i    = 1
    while i+251 != 0: # L'astuce c'est de faire un overflow de notre registre de comptage
        ret += i
        i   += 1
    return ret
```

Et le code assembleur se trouve dans le fichier **asm/sum.asm**. Pour l'assembler, il vous suffit d'avoir python et de lancer la commande suivante `python3 compilateur.py -i sum.asm > rom/sum.rom` qui enregistrera la rom au format logisim dans le fichier**rom/sum.rom** (la rom est déjà assemblée si vous préferez). Ensuite, chargez la rom dans le bloc PC, composante rom du microprocesseur et faite dérouler le programme jusqu'à arriver sur l'instruction **0000** (attention, ne pas l'executer). Si vous allez en RAM, vous verrez qu'en 0x00, nous avons bien 1+2+3+4 = 10 = 0x0a. N'hésitez pas à coder vos programmes et vous amuser !

Améliorations en vue pour la suite

- Ajouter un troisième mode de modification du PC par incrémentation selon un offset relatif
- Ajouter un registre de flags, et une opération de comparaison permettant d'aller set des bits de ce registre. Ainsi, nos opérations de jmp conditionnels pourront découler non pas de la comparaison d'un registre avec zero mais de bits set ou non dans notre registre des drapeaux.
- Optimiser tous les circuits en taille/perfs (utiliser des portes nand, transformer notre additionneur afin qu'il ne soit plus à propagation de retenue)
- Rendre requetable notre registre de pointeur programme
- Créer une instruction nop pour remplir la fin de notre rom et eviter les effets de bords, ou alors une interruption pause

Faire fonctionner le microprocesseur

Pour ce faire :

- Lancer logisim : Fichier > Ouvrir > **logisim_microproc/microproc.circ**
- Double cliquer sur le circuit **microprocesseur**
- Double cliquer sur le circuit **PC_AND_ROM** situé tout à gauche, en dessous de l'horloge
- Clic droit sur la rom > Charger l'image >**roms/sum.rom** pour charger le programme qui va faire la somme des 4 premiers entiers en @0x00 de la RAM
- Vous pouvez faire ctrl+t pour faire ticker l'horloge : lorsque la rom pointe sur 0000 (attention, pointe mais n'a pas encore executé !) vous pourrez remarquer en allant dans la ram (double clic sur le circuit microprocesseur dans la barre des circuits, puis double clic sur la ram qui est **en dessous du circuit PC_AND_ROM**, noté **RAM**) qu'en @0x00, il y a bien 1+2+3+4 = 10 = 0x0a.

Conclusion

Ce microprocesseur était un projet vraiment intéressant, j'ai pris un grand plaisir à le réaliser : j'aurais préféré avoir plus d'heures afin de rendre ce microprocesseur plus propre, nottament au niveau de l'optimisation des circuits. Je suis aussi conscient de l'abstraction proposée par logisim (tout se fait en 1 cycle...).