Rapport Circuits et Architecture

Circuit

Le projet couvre le rendu minimal indiqué dans le sujet.

`DecodeIR` a été modifié afin de décoder le contenu du registre IR courant, permettant d'alimenter les étiquettes associées aux opérations étant `Arith`, `Load`, `Store` et `Jump`. `WriteReg` est quant à elle toujours alimentée si l'un des fils des étiquettes précédentes l'est aussi, à l'exception de `Store` ou seulement l'instruction `TRAP` modifie un registre.
Ce décodage a été effectué en splittant les bits 12, 13, 14 et 15 de IR et en les regroupant par formule booléenne grâce au tableau de la répartition des op-codes.

Pour mettre à jour la valeur actuelle du registre d'état de NZP ainsi que l'étiquette de `TestNZP` il est nécessaire de traiter le bit de poids fort du résultat courant. Le registre de NPZ sera changé avec sa nouvelle valeur si l'instruction courante écrit dans un registre. Cette information est récupérée par la valeur de l'étiquette `WriteReg` que l'on met à jour dans `DecodeIR` comme indiqué précédemment.
Pour savoir si un test de saut est requis, il faut comparer les bits 9, 10 et 11 de IR avec la nouvelle valeur du registre NZP. Si au moins un de ces bits est vrai, il faut donc faire un test de saut.

Le sous-circuit `WriteVal` a été implémenté de manière à sélectionner la bonne valeur à écrire dans `RegIN`, dépendamment de si l'instruction courante est un `LEA`, un `JSR` ou plus généralement des instructions `Load`. Pour sélectionner cette valeur, un multiplexeur à 3 bits de contrôle a été utilisé, dont voici la table de vérité:
Les trois bits de contrôle étant composés des bits de `LEA`, `JSR` puis `Load`.

LOAD	**JSR**	**LEA**	7	6	5	4	3	2	1	0
0	0	0	x	x	x	x	x	x	x	1
0	0	1	x	x	x	x	x	x	1	x
0	1	0	x	x	x	x	x	1	x	x
0	1	1	x	x	x	x	1	x	x	x
1	0	0	x	x	x	1	x	x	x	x
1	0	1	x	x	1	x	x	x	x	x
1	1	0	x	1	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x

Le cas de `JSR` est couvert par l'entrée 2. Pour `LEA`, lorsque les bits de `LOAD` et `LEA` sont à 1, l'entrée 5 met `RegIN` à `PC+SEXT(Offset9)`. Pour les bits de` LOAD` restants, il s'agit des entrées 4, 6, 7 qui vont donner `MemOut` au registre. Les entrées restantes sont pour le résultat de l'ALU.

Tests des instructions

LEA:

```
        v2.0 raw
        e001 0000 0054 0065 0073 0074 0000
# LEA R0, s
# NOP
# s:    .STRINGZ "Test"
```

Ce code stocke un string "Test" avec le label s après un NOP. Dans R0 sera stocké l'adresse de début du string.

LDR:

```
        v2.0 raw
        6001 1020
# LDR R0, 0, 1 <=> R0 <- mem[1]
```

Ce code écrit dans R0 l'élément à l'adresse 0x0001, ici s'agissant du code hexadécimal 0x1020.

LD:

```
        v2.0 raw
        5020 2000 1020
# LD R0, 0 <=> R0 <- mem[PC + 1]
```

Ce code écrit dans R0 l'élément à l'adresse 0x0002, similaire à LDR mais se basant sur PC pour calculer l'élément de mémoire à récupérer.

ST:

```
        v2.0 raw
        5020 1025 3000
# AND R0, R0, 0  <=> R0 <- 0
# ADD R0, R0, 5  <=> R0 <- 5
# ST R0, 1       <=> mem[PC] <- R0
```

Ce code initialise dans R0 une valeur que ST va écrire dans mem[PC]. En l'occurrence ce test va écrire 0x0005 à l'adresse 0x0003.

STR:

```
        v2.0 raw
        5020 1025 7000
# AND R0, R0, 0  <=> R0 <- 0
# ADD R0, R0, 5  <=> R0 <- 5
# STR R0, R0, 0   <=> mem[R0 + 0] <- R0
```

Ce code initialise dans R0 une valeur que STR va écrire à l'adresse mémoire que contient ce registre. En l'occurrence ce test va écrire 0x0005 à l'adresse 0x0005.

Lucas RODRIGUEZ 22002335
Florian GAIE 22015287