Système digital – rapport intermédiaire

Architecture, jeu d'instructions

Théophile Bastian, Noémie Cartier, Nathanaël Courant

13 décembre 2015

Résumé

Nous avons choisi d'utiliser une architecture proche d'ARM, en réduisant toutefois le jeu d'instructions et avec de légères incompatibilités.

1 Architecture

L'architecture que nous avons choisie est fortement inspirée d'ARM, dans le but d'une part de se rapprocher d'une architecture réelle, et d'autre part de pouvoir compiler du code vers notre assembleur en utilisant un compilateur standard (comme gcc par exemple), quitte à retravailler l'assembleur fourni pour se ramener à un jeu d'instructions gérées.

1.1 Mémoire

Le processeur gère un nombre de registres à définir. L'accès à la ROM est impossible (§1.2) ; l'accès à la RAM est limité à un accès lecture et un accès écriture par cycle (en particulier, une seule RAM est gérée). Les nombres seront représentés sur 64 bits, concrètement gérés par des nappes de 64 fils. Notons que le simulateur netlist gère les opérations bitwise directement sur les nappes de fils.

Certains registres seront déclarés entrées ou sorties du circuit netlist, ce qui permettra de gérer les entrées/sorties.

1.2 Mémoire d'instructions

La ROM contiendra et représentera exclusivement les instructions du programme assemblé. En particulier, l'accès à la ROM est donc impossible du point de vue de l'utilisateur, car le processeur y accède déjà à chaque cycle pour lire l'instruction à effectuer.

1.3 Arithmétique

Les opérations arithmétiques gérées sont uniquement l'addition, la soustraction et les opérations *bitwise* standard. Cela sera suffisant : l'horloge doit seulement s'incrémenter, et retirer 60 ou 24. La sortie sur 7 segments se fait quant à elle par recherche dans une table.

2 Assembleur

Dans cette section, nous dressons une liste exhaustive des opérations que nous avons prévu de supporter (et brièvement leur fonctionnement).

2.1 Syntaxe d'une opération

Nous allons utiliser la syntaxe ARM pour décrire nos opérations : une opération est de la forme

```
<Opération>[<conditionnelle>][S] <argument(s) de l'opération>
```

où `[...]` désigne une partie optionnelle ; `<conditionnelle>` désigne une condition portant sur les flags d'exécution de l'instruction (*ie.*, l'instruction sera exécutée *ssi* la condition est vraie, ce qui permet d'éviter les `JMP` sur des conditions simples) ; et `S`, lorsque présent, met à jour les flags (pour certaines opérations, `S` n'existe pas et les flags sont toujours mis à jour). Les arguments, quant à eux, sont spécifiques au type de l'opération.

2.2 Opérations arithmétiques

Une opération arithmétique a trois arguments, avec la syntaxe suivante :

```
Rd, Rn, Opérande2
```

où `Rd` est le registre de destination (qui n'est pas nécessairement l'un des registres des opérandes), `Rn` est le registre de l'opérande 1 (op_1) et `Opérande2` est la description de l'opérande 2 (op_2) (§2.4).

— `ADD` : $op_1 + op_2$
— `ADC` : $op_1 + op_2 +$ carry bit
— `SUB` : $op_1 - op_2$
— `SBC` : $op_1 - op_2 +$ carry bit $- 1$
— `RSB` : $op_2 - op_1$
— `RSC` : $op_2 - op_1 +$ carry bit $- 1$
— `AND` : op_1 AND op_2
— `EOR` : op_1 XOR op_2
— `ORR` : op_1 OR op_2
— `BIC` : op_1 AND NOT op_2

2.3 Comparaisons

Une opération de comparaison a deux arguments, avec la syntaxe suivante :

```
Rn, Opérande2
```

où `Rn` est le registre de l'opérande 1 (op_1) et `Opérande2` est la description de l'opérande 2 (op_2) (§2.4).

De plus, *le paramètre `S` n'existe pas* pour les opérations de comparaison, car automatiquement vrai. Le *seul effet* de ces instructions est donc de mettre à jour les flags.

— `CMP` : $op_1 - op_2$ (sans conserver le résultat)
— `CMN` : $op_1 + op_2$ (sans conserver le résultat)
— `TST` : op_1 AND op_2 (sans conserver le résultat)
— `TEQ` : op_1 XOR op_2 (sans conserver le résultat)

2.4 Déplacement de données

Une opération de déplacement de données a deux arguments, avec la syntaxe suivante :

```
Rd, Opérande2
```

où `Rd` est le registre de destination et `Opérande2` est la description de l'opérande 2 (op_2) (§2.4).

— `MOV` : op_2
— `MVN` : NOT op_2

Description de l'opérande 2

Dans tous les paragraphes précédents, l'*opérande 2* (op_2) est :

— soit une constante numérique 8 bits précédée d'un `#`, prenant l'une des formes `#`⟨*constante_décimale*⟩, `#0x`⟨*constante_hexadécimale*⟩ ou `#0b`⟨*constante_binaire*⟩ ;
— soit un nom de registre.

De plus, si l'opérande 2 est suivie d'une instruction de décalage prenant la forme :

`, <instruction> <opérande>`

où `<opérande>` (*op*) est une constante 6 bits[1] ou un registre (seuls les 6 bits de poids faible seront pris en compte), et où `<instruction>` est l'une des instructions suivantes :

— `LSL` : décalage logique à gauche de *op* bits
— `LSR` : décalage logique à droite de *op* bits
— `ASR` : décalage arithmétique à droite de *op* bits

Par exemple, `ADD r0, r1, r1, LSL #2`, où l'opérande 2 est `r1, LSL #2`, effectue $r0 \leftarrow r1 + (r1 \ll 2)$.

2.5 Accès mémoire

Une opération d'accès mémoire a deux arguments, avec la syntaxe suivante :

`Rd, adresse`

où `Rd` est le registre de destination (R_d) et `adresse` est la description de l'adresse mémoire (*addr*). Ces opérations travaillent toutes sur des mots mémoire entiers (§1.1).

De plus, *le paramètre `S` n'existe pas* pour les opérations d'accès mémoire (car ces opérations ne sont pas branchées sur le circuit arithmétique).

Pour respecter la compatibilité avec ARM, la description d'une adresse mémoire a la syntaxe suivante :

`[`*`descr`*`]`

où *`descr`* est soit un registre, soit une constante explicite.

Les opérations d'accès mémoire sont :

— `LDR` : $R_d \leftarrow$ Mémoire[*addr*]
— `STR` : $R_d \rightarrow$ Mémoire[*addr*]

2.6 Sauts

Une ligne comportant exactement

$\langle nom \rangle$ `:`

crée un label nommé *nom* pointant sur l'instruction suivant cette ligne.

On définit (d'après la syntaxe précédente) une commande ayant pour argument un nom de label *lbl* et n'acceptant *pas* l'option `S` :

— `JMP` : saute au label *lbl*

Notons que ces deux éléments sont purement du sucre syntaxique, car le pointeur d'instructions est un registre comme les autres. Il suffit donc en pratique de transformer une instruction `JMP` en une instruction `MOV` ou `ADD` (selon si on veut se déplacer relativement ou de manière absolue).

2.7 Conditionnelles et flags

Les flags suivants existent :

— `N` : le dernier résultat est strictement négatif
— `Z` : le dernier résultat est nul
— `C` : le dernier calcul a produit une retenue sortante
— `V` : le dernier calcul a produit un overflow

Une conditionnelle (§2.1) est alors l'un des motifs décrits ci-dessous :

1. 5 dans la référence, mais nos nombres sont en 64 bits

Id	Motif	Condition
0000	EQ	Z
0001	NE	$\bar{Z}$
0010	HS / CS	C
0011	LO / CC	$\bar{C}$
0100	MI	N
0101	PL	$\bar{N}$
0110	VS	V
0111	VC	$\bar{V}$
1000	HI	$C \wedge \bar{Z}$
1001	LS	$\bar{C} \vee Z$
1010	GE	$(N \wedge V) \vee (\bar{N} \wedge \bar{V})$
1011	LT	$(N \wedge \bar{V}) \vee (\bar{N} \wedge V)$
1100	GT	$GE \wedge \bar{Z}$
1101	LE	$LT \vee Z$
1110	AL	1
1111	NV	réservé

Référence(s)

— La référence ARM utilisée : http://simplemachines.it/doc/arm_inst.pdf