Micro architecture de l'ARM v2A

Laniel Francis
francis.laniel@etu.upmc.fr

13 janvier 2016

Résumé

Rapport présentant la micro architure de l'ARM v2A

Table des matières

1 Introduction

1.1 L'UE VLSI

Dans le cadre du cours d'*initiation à la conception* ***Very Large Scale Integration (VLSI)*** il m'a été demandé de réaliser une architecture simplifiée d'un processeur basée sur celle de l'ARM v2A.
Pour la modélisation, j'ai utilisé le langage **Very High Speed Integrated Circuit Hardware Description Language (VHDL)** ainsi que différents outils dont voici la liste :

ghdl : un compilateur vhdl libre basé sur gnat

gtkwave : un outil libre de visualisation de simulation

Alliance CAD tools : une suite d'outil libre pour la conception assistée par ordinateur de design VLSI

1.2 L'ARM v2A

Le processeur étudié est un processeur **Reduced Instruction Set Computer** (RISC) 32 bits comportant un pipeline découpé en 5 étages (IFETCH, DECOD, EXE, MEM, WBK). C'est un processeur ARM par conséquent son jeu d'instructions s'appuie sur une gestion élégante des conditions qui sont symbolisées par 4 registres d'un bit appelés "flags" :

N : ce flag est positionné si une instruction a produit un résultat négatif

Z : ce flag sera positionné par une instruction ayant produit un résultat nul

C : ce flag sera levé lorsqu'une opération non signée produit un dépassement de capacité

V : le flag V agit identiquement au flag C mais dans le cas d'opérations signées

Grâce à ces flags il est possible de conditionner chaque opération, voici un petit aperçu de la puissance de ce langage d'assemblage face à celui de l'architecture MIPS :

```
#code C
int i;
for(i = 0; i < size; i++){
  if(tab[i] < val)
    tab[i] += val;
}
#R4 est l'adresse de notre itérateur
#R6 est l'adresse de fin du tableau
#R7 est la valeur à comparer et à potentiellement ajouter
```

```
#MIPS
_loop :
LW R5, 0(R6)
SLT R10, R5, R7
BEQ R10, R0, _endif
NOP
ADD R5, R5, R7
_endif :
ADDIU R4, R4, 4
BNE R4, R6, _loop
NOP
```

```
#ARM
_loop :
LDR R5, 0(R6)
CMP R7, R5
ADDLT R5, R5, R7
ADD R4, R4, #4
BNE R4, R6, _loop
```

Pour cet **exemple** il est clair que même en optimisant le code de l'assembleur MIPS le code ARM sera meilleur en terme de cycles par instruction. Bien entendu, il est impossible d'affirmer qu'en **général** un langage d'assemblage est meilleur qu'un autre. Surtout que le nombre de cycles n'est pas la seule variable à prendre en compte.

La modélisation de ce processeur aurait du m'amener à obtenir le dessin des masques en utilisant les outils Alliance sur le code VHDL écrit. Malheureusement la simulation ne s'est pas déroulée comme prévu et je n'ai pas pu obtenir ces dessins...

Dans ce rapport je présenterai d'abord les différents étages de notre processeur puis je concluerai sur mon travail.

2 Les étages du processeur

2.1 IFECTH

Cet étage a pour principale tâche d'aller lire dans le cache d'instructions la prochaine instruction à éxécuter. Une fois ceci fait il enverra à l'étage DECOD l'instruction lue.

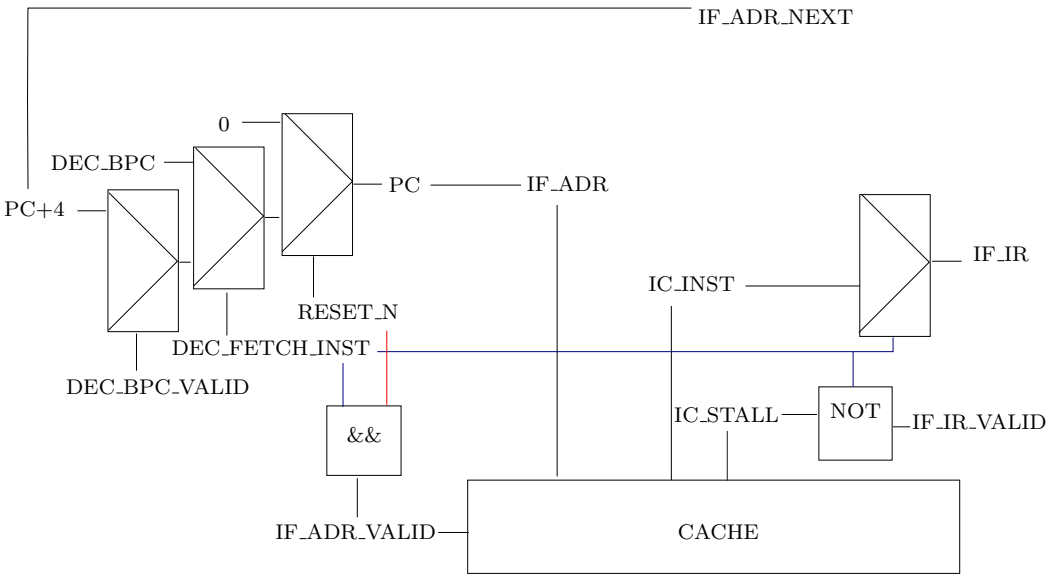

FIGURE 1 – Schéma de l'étage IFETCH (les couleurs servent uniquement à clarifier le dessin)

C'est aussi cet étage qui s'occupe de la gestion du registre **PC** (Program Counter) et qui répercute les cycles de gel sur la suite du pipeline notamment grâce à cette ligne :

```
if_ir_valid <= not ic_stall;
```

2.2 DECOD

C'est dans cet étage du pipeline que seront décodées les instructions venant de l'étage IFETCH.
Les instructions sont codées sur 32 bits, il existe plusieurs types d'instructions dont notamment :

REGOP : Les instructions de **Data Processing**, ce sont les instructions classiques comme add, or ou and.

MULT : Instruction de multiplication, il existe une version utilisant 4 registres où le dernier registre sera accumulé au résultat ($dest = op1 * op2 + op3$).

TRANS : Les instructions de transfert tels quels STRB ou LDR.

BRANCH : Les instructions de branchement.

La condition d'exécution des instructions est toujours codée sur les 4 premiers bits (31 .. 28).

Par exemple un **addnes R1, R2, R3** aura pour codage :
0001 0000 1001 0010 0001 0000 0000 0011
Et ce code **0000 0000 0000 1000 0000 0100 1001 0001** correspond à un **muleq R8, R1, R4**.

Dans le cas d'une REGOP le second opérande peut prendre plusieurs formes :

— Un simple immédiat.
— Un simple registre.
— Un registre décalé d'une valeur sur 5 bits (donc de 0 à 31).
— Un registre décalé de la valeur contenue dans un registre.

Pour mon architecture le dernier cas ne fonctionne pas...

Il existe plusieurs types de décalages, les voici :

LSL : Pour Logical Shit Left, c'est un simple décalage à gauche.

LSR : Logical Shit Right, il fonctionne comme le LSL mais à droite.

ASR : Arithmetic Shit Right, son comportement est identique à celui de LSR mais le registre sera étendu par la valeur de son bit de signe

ROR : Rotate Right, une rotation vers la droite

RRX : Rotate Right Extend, une rotation vers la droite étendu utilisant le carry flag. RRX ne peut pas prendre de valeur de décalage.

Dans mon architecture l'ASR ne fonctionne pas, en effet la valeur sign_op2 est calculé trop tôt et par conséquent elle est toujours égale à X"00000000".

DECOD implémente aussi une machine à état afin de gérer le "fetch" d'une instruction. Soit la suite d'instruction suivante :

```
ADD R5, R6, R7
SUB R10, R5, R11
```

Il y a clairement une dépendance de données entre ces instructions sur le registre R5. Par conséquent tant que l'instruction ADD n'aura pas produit son résultat il sera impossible de lancer l'instruction SUB. Donc pendant ce temps le prochain état (comprendre l'état de DECOD pour l'instruction SUB) sera OPWAIT.
J'ai complété le code du process gérant la machine à état mais je n'ai pas vérifié son fonctionnement. Par conséquent je ne pense pas qu'elle fonctionne.

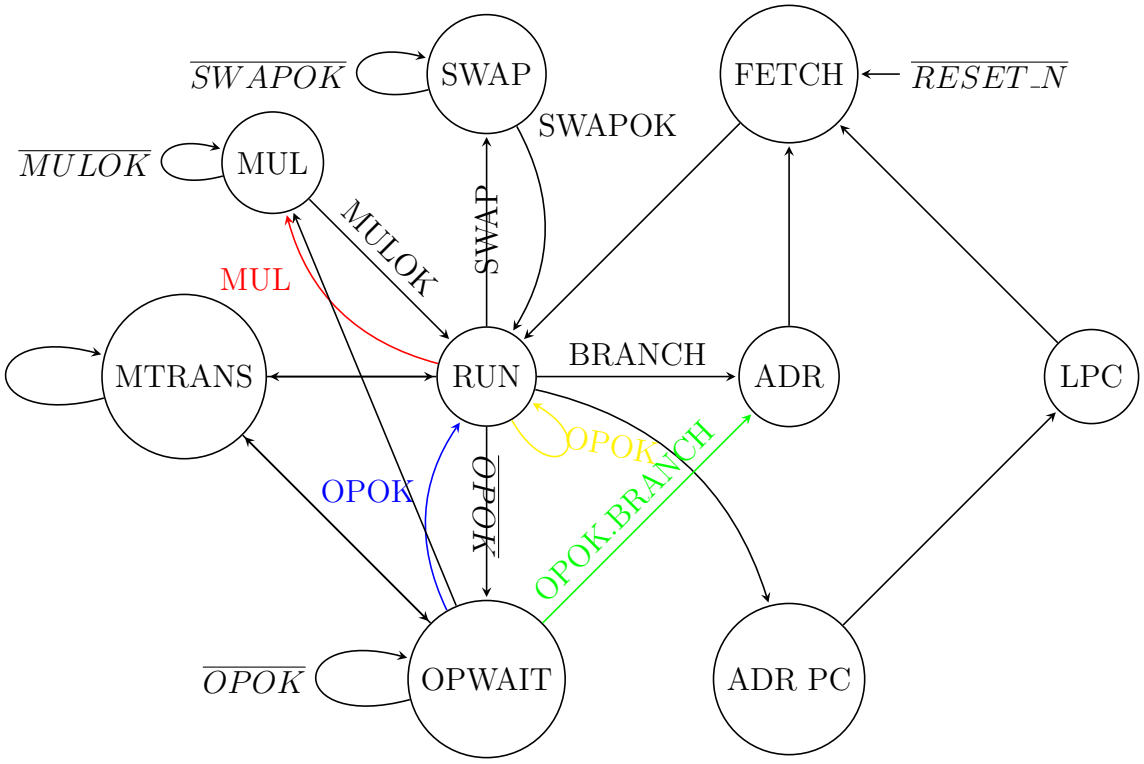

FIGURE 2 – Machine à état de DECOD (les couleurs servent uniquement à clarifier le dessin)

2.3 EXE

C'est ici, qu'aura lieu l'exécution de nos instructions ainsi que les décalages. Dans notre processeur, nous avons décidé d'implémenter réellement que 4 instructions : add, or, xor et and.

En effet, à partir de ces 4 instructions, il est possible de recréer toutes les REGOP. Par exemple, un sub ne sera en réalité qu'un add dont le second opérande aura été complémenté à 2 afin d'obtenir son négatif.

En VHDL, il existe les opérateurs and, or et xor, par conséquent les instructions correspondantes sont facilement réalisable.

Par contre, ce n'est pas le cas de l'addition puisque l'addition n'est pas définie entre des objets du types **std_logic_vector**. Il aurait donc fallu convertir ces objets, ce qui aurait sûrement posé problème lors de la synthèse.

Pour pallier à cela, il a fallu utiliser une autre stratégie. Optimalement il aurait fallu utiliser un additionneur de type **Carry-lookahead** mais je ne suis pas arrivé à développer cette solution. Je me suis donc rabattu sur un additionneur complet 1 bit itéré 32 fois...

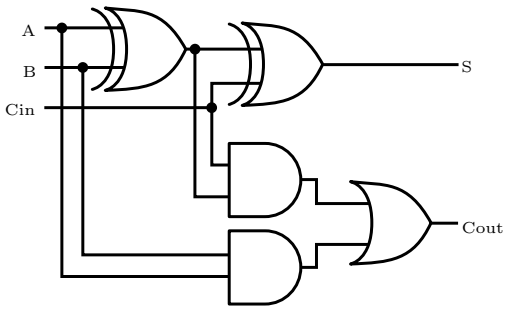

FIGURE 3 – Schéma d'un additionneur complet

```
--code du fulladder
cout(0) := dec_alu_cy;

for i in 0 to 31 loop
  sout1(i) := op1(i) xor op2(i);

  cout1(i) := op1(i) and op2(i);

  sout(i) := cout(i) xor sout1(i);
  cout2(i) := cout(i) and sout1(i);

  cout(i + 1) := cout1(i) or cout2(i);
end loop;
```

D'après mes tests, toutes les REGOP donnent un résultat correct.
Cet étage comporte aussi un bypass (**EXE_ALU_RES** sur la figure 4) afin de limiter les cycles de gel.

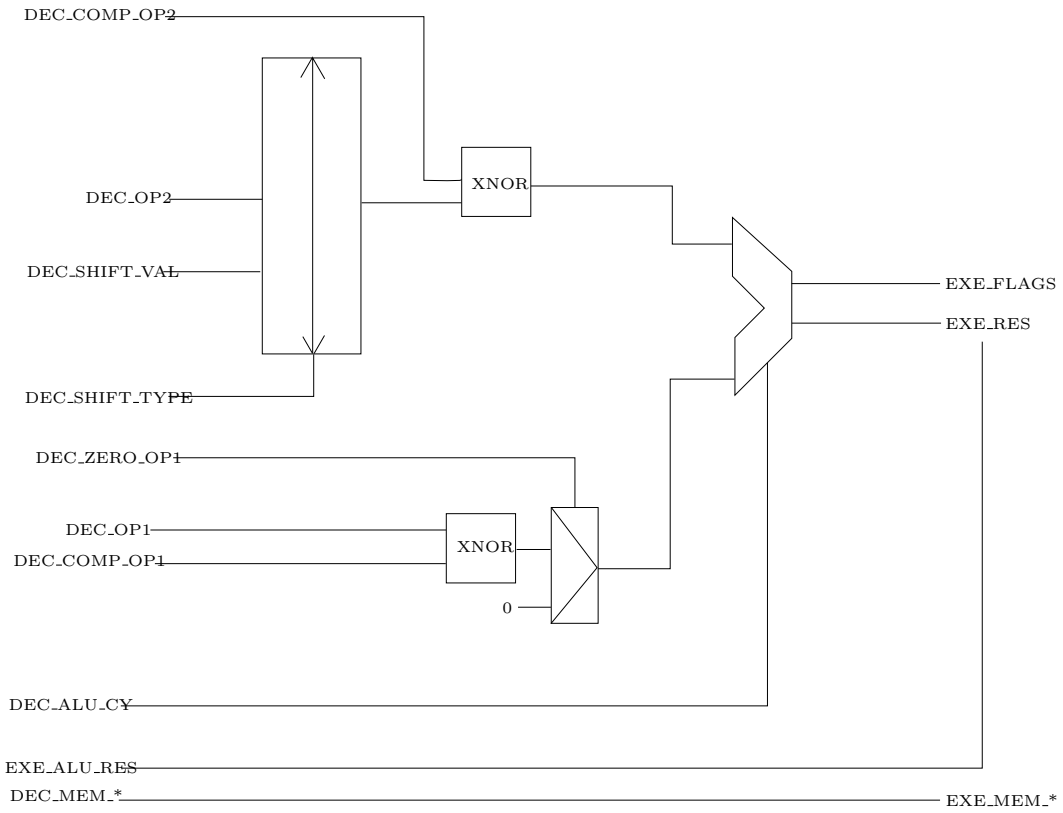

FIGURE 4 – Schéma de l'étage EXE

DEC_SHIT_TYPE correspond en fait à **DEC_SHIFT_LSL**, **DEC_SHIFT_LSR**, **DEC_SHIFT_ASR**, **DEC_SHIFT_ROR** et **DEC_SHIFT_RRX**. Tout comme **EXE_FLAGS** correspond en fait aux 4 "flags" présenté dans l'introduction

2.4 MEM

C'est dans cet étage que les instructions STR et LDR prennent tout leur sens. En effet l'adresse calculée à l'étage EXE ainsi que la potentielle donnée à stocker seront envoyées au cache de données via cet étage.

3 Conclusion

Il est clair que les objectifs de l'UE n'ont pas été atteints, la synthèse n'a pu être menée à terme et l'architecture présentée n'est clairement pas complète.
Les deux principales fonctions de cette architecture sont :

- Le décodage des instructions
- Les opérations et presque tous les décalages de l'étage EXEC

Pourtant, cette UE m'a permis d'apprendre le VHDL qui est un langage très particulier. En effet la modélisation est très différente de la programmation. De plus même s'il est possible d'utiliser des algorithmes, ceci sont très différents de ceux que j'ai l'habitude d'utiliser puisqu'ils sont plus bas niveau.

Le VHDL permet de se placer au niveau des bits là où les langages de programmation classiques sont limités à l'octet.

En apprenant le VHDL j'ai aussi pu me rendre compte qu'il était possible d'effectuer beaucoup de tâches en matériel.

Enfin cette UE m'a aussi permis d'utiliser de nouveaux outils tels que ghdl, gtkwave ou les outils Alliance.