Département Sciences Informatiques
4° année :

Création et simulation de processeur RISC RP2 en VHDL

Réalisé par :
AUZIAS Maël (SAR1)
dans le cadre de la matière Architecture Machine durant 6 semaines.

Année scolaire 2011/2012

Introduction :

Présentation du travail :

Le travail réalisé dans ce projet est la création et simulation du processeur RP2 en VHDL. Ce rapport décrit le travail effectué et explique les choix que j'ai été amené à faire, leurs limites, et leurs améliorations possibles.

Ce projet a été réalisé seul (même si j'ai été amené à parler et avoir quelques réflexions avec d'autre personnes). Notre enseignant nous a donné une grande liberté d'implémentation en nous fournissant seulement un fil directeur et répondant aux questions auxquelles nous ne pouvions pas répondre seul.

Je n'ai pas eu le temps de finir ce projet, la partie de décodage et l'assemblage final n'ont pas été réalisé. Je le regrette car je sais que ce sont deux parties très importantes. Cependant j'ai pu acquérir une grande quantité de connaissance avec le travail que j'ai effectué.

Comment lire le rapport :

Mon rapport se décompose en différentes parties, cf le sommaire juste après. Les points importants (tel que les choix réalisés, les problèmes soulevés, et les améliorations possibles) sont colorés de façon à les mettre en valeur. Si l'architecture du processeur est connue par le lecteur alors ce sont ces points à lire en priorité ainsi que ceux mis en valeur dans le sommaire.

Sommaire :

I] Composants basiques :

Half adder :

Entité :

Nom	Type	Taille	Utilité
A	IN	bit	Ceux sont les deux bits à additionner
B			
S	OUT		Résultat de la somme
Cy			Retenue de la somme

Architecture :

S correspond au nombre de bit en entrée à 1 (soit 1 ou 0), cette fonction est réalisé avec un *xor*.
Cy est à 1 si les deux bit d'entrée sont à 1.

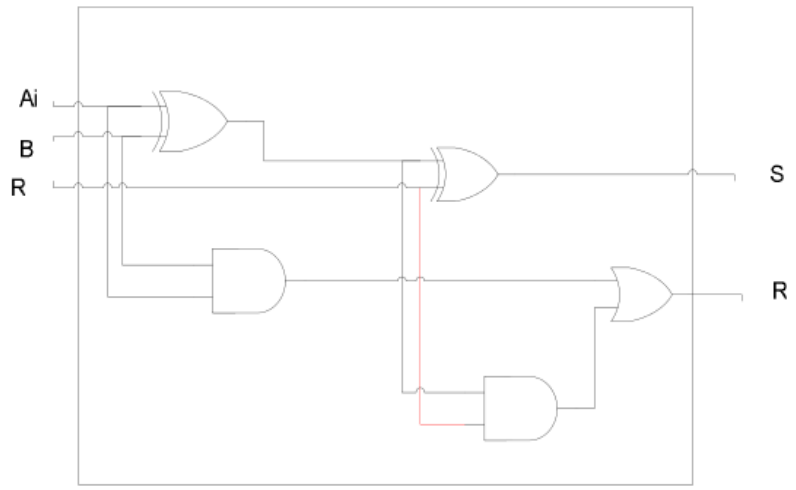
Full adder :

Entité :

Nom	Type	Taille	Utilité
A	IN	bit	Ceux sont les deux mots binaires à additionner
B			
S	OUT		Résultat de la somme
Cin	IN		Retenue entrante
Cout	OUT		Retenue sortante

Architecture :

L'additionneur se réalise avec deux demi-additionneurs :
Les deux bits à additionner sont mis en entrée du premier demi-additionneur. On utilise ensuite un second demi-additionneur auquel on met la retenue entrante et le résultat du premier demi-additionneur en entrée.
Pour la retenue sortante on utilise un *or* entre : (Cin *and* résultat du premier additionneur) et (A *and* B).
NB : **Cin** permet de cascader les additionneurs ainsi réalisés.
Schéma équivalent : (source : amrouche.esi.dz/doc/ch4_circuitscombinatoires.pdf)



Adder 16 bit :

Entité :

Nom	Type	Taille	Utilité
A	IN unsigned	2 octets	Ceux sont les deux mots binaires à additionner
B			
S	OUT unsigned		Résultat de la somme
Cin	IN	bit	Retenue entrante
Cout	OUT		Retenue sortante

Architecture :

Cin permet de cascader les additionneurs réalisés : si l'on veut réalisé un additionneur de 32 bits il suffira de mettre le Cin de l'additionneur des LSB au niveau bas, le Cout sur le Cin de l'additionneur des MSB, et concaténer les résultats obtenus.
L'additionneur de 16 bits a été réalisé en utilisant 16 additionneurs complets cascadés.
Un signal (tableau de 16 bit) de retenues intermédiaires est utilisé dans cette architecture.
Branchements :

Spécifité à l'additionneur 0 :

Le bit **Cin** est branché à cet additionneur.

Additionneur N :

N^{ième} bit de **A** et de **B** branchés en entrée **A** et **B** de l'additionneur.
N-1^{ième} bit du signal de retenues branché en entrée **Cin** de l'additionneur.
Résultat de l'additionneur branché au N^{ième} bit de **S**.

Cout correspond au MSB du tableau des retenues intermédiaires.

Note importante : les octets utilisés dans les additionneurs sont de type unsigned au moment où je rédige le rapport. Je les modifierais plus tard car ce sont des données et non des signaux de commandes qui transitent par l'additionneur !

Multiplexer :

Entité :

Nom	Type	Taille	Utilité
sel	IN unsigned	2 bits	Choix (adresse) de l'entrée à diriger en sortie
input0	IN (cf note)	2 octets	Entrées à redirigées en fonction de la valeur de sel
input1			
input2			
input3			
output	OUT (cf note)		Sortie égale à l'entrée choisie

Architecture :

Le comportement de composant est très simple : on redirige la sortie dont le numéro d'index correspond à la valeur de sel.
Si **sel** vaut `00' l'entrée **input0** sera dirigée sur la sortie **output**.
Ce composant est utilisé plusieurs fois dans le processeur RP2. Parfois seul deux entrées sont utilisées au lieu de quatre. Un composant avec un nombre d'entrées générique aurait eu plus de sens, mais je ne sais pas si cela est possible.

Note importante : pour des raisons d'incompatibilité de signaux signés et non signés le composant multiplexer existe en deux versions : une dirigeant des signaux de données (signés) et une des signaux de commandes (non signés).

Registre :

Entité :

Nom	Type	Taille	Utilité
input	IN (cf note)	générique	Mot d'entrée à mémoriser
output	OUT (cf note)		Sortie
clock	IN	bit	Synchronisation du composant
reset			Remet à zéro le registre (reset asynchrone)

Architecture :

Le comportement de composant est très simple : au front montant on répète l'entrée sur la sortie. Si un reset a lieu on met à 0 la sortie.

Note importante : pour des raisons d'incompatibilité de signaux signés et non signés le composant registre existe en deux versions. L'assemblage aurait pu fonctionner sans ces deux versions différentes, mais pour plus de cohérence j'ai décidé de faire des registres pour différencier les commandes des données.

Program counter :

Entité :

Nom	Type	Taille	Utilité
input	IN unsigned	1 octet	Choix de l'adresse à charger
output	OUT unsigned	1 octet	Adresse courante
load	IN	bit	Cf tableau suivant
stall			
clock			Synchronisation du composant
reset			Remet à zéro le composant (reset asynchrone)

Architecture :

stall	load	comportement
0	0	Aucune opération réalisée
0	1	
1	0	Chargement de l'adresse renseignée sur **input**
1	1	Incrémentation de l'adresse courante

La lecture de l'adresse d'entrée, l'incrémentation, et la paralysation se font sur front montant.
L'écriture de l'adresse se fait sur front descendant.
Le **reset** est asynchrone.

ALU :

Entité :

Nom	Type	Taille	Utilité
input1	IN signed	2 octets	Mots binaires à traiter
input2			
codeOp	IN unsigned	3 bit	Code de l'opération à réaliser (cf tableau suivant)
output	OUT signed	2 octets	Résultat

Architecture :

Ce tableau représente les opérations traitées par mon code au moment de la rédaction du rapport :

codeOp	comportement
000	output = input1 **+** input2
001	output = input1 **-** input2
010	output = not(input1) --complément à 2 car signal signé
autre	output = input1 * input2

Note importante : il n'y a aucune présence de flag ! Les flags de comparaison n'ont pas lieu d'être pour la simple, bonne et unique raison que l'ALU ne réalise pas de comparaison. Cependant les flags overflow, zero, ou carry ne sont pas implémentés. Les développeur n'aurons donc aucun moyen de vérifier si les résultats sont corrects et si leur données ne sont pas erronées (car non traitement de la retenue, ou bien d'un overflow).

Banc de registres :

Entité :

Nom	Type	Taille	Utilité
N1	IN unsigned	4 bits	Adresse d'écriture
N2			Adresses de lecture
N3			
op1	IN signed	2 octets	Mot binaire à écrire à l'adresse N1
op2	OUT signed		Sorties des mots binaires des adresses N2 (op2) et N3 (op3)
op3			
clock	IN	bit	Synchronisation du composant
write			Autorise ou non l'écriture

Architecture :

Ce banc de registre permet deux lectures et une écriture simultanées et synchrones.
Si **write** est au niveau haut alors au front descendant suivant la donnée **op1** sera enregistrée dans le registre d'adresse renseignée par **N1**. A chaque front montant les données d'adresses renseignées par **N2** et **N3** seront écrites respectivement sur les sorties **op2** et **op3**.

Mémoire de données :

Entité :

Nom	Type	Taille	Utilité
op	IN signed	2 octets	Donnée à écrire
add	IN unsigned	1 octet	Adresse à laquelle lire ou écrire
write	IN	bit	Choix de lecture ou d'écriture
s	OUT signed	2 octets	Sortie de la donnée en mode lecture

Architecture :

Lorsque le signal **write** est au niveau bas :

s correspond à la donnée mémoirisée à l'addresse **add**.

Lorsque le signal **write** est au niveau haut :

la donnée **op** est écrite à l'adresse **add**.

Extension de signe :

Entité :

Nom	Type	Taille	Utilité
data_in	IN signed	1 octet	Entré d'un mot d'un octet
data_out	OUT signed	2 octets	Même donnée qu'en entrée avec extension de signe

Architecture :

L'architecture se résumé à un if... then... else... :
Si le dernier bit est à 1 alors les 8 MSB de la sortie seront à 1, sinon ils seront à 0.
On aurait pu réaliser le tout sans if avec la ligne suivante :
data_out <= data_in(7) & data_in(7) & data_in(7) & data_in(7) & data_in(7) & data_in(7) & data_in(7) & data_in(7) & data_in(7 downto 0)
Mais la lisibilité est perdue. La ligne ci-dessus est l'équivalent du pseudo-code :
data_out = 8 MSB égaux au septième bit concaténés au mot d'entrée.

II] Étages du pipeline :

Dans notre processeur les cinq étages du pipeline sont : Instruction Fetch (IF), Decode and operation fetch (DE), Execute (EX), Memory Access (MEM) et Write Back (WB).
L'étage Write Back n'est pas un étage proprement dit, on ne trouve pas un assemblage de composants réalisant l'étape write back.
Les traitements sont réalisés par des mots de 16 bit. Ces données sont codées en complément à deux, et une entité d'extension de signe est présente à l'entrée du banc de registre. Le processeur est cadencé par une seule horloge.

Instruction fetch :

Entité :

Nom	Type	Taille	Utilité
input	IN unsigned	1 octet	Entrée du program counter
output	OUT unsigned	2 octets	Sortie du registre
load	IN	bit	Entrées du program counter
stall			
clock			Synchronisation du composant
reset			Remet à zéro le composant (reset asynchrone)

Architecture :
L'étage d'instruction fetch est réalisé par l'assemblage de différents composants en accord avec la documentation RP2 qui nous a été fournie.
Ces composants sont :

- Program counter,
- Instruction memory,
- Registres.

La réalisation de ce composant a été rapide et aisée, aucun choix technologique n'a dû être fait. Il n'est qu'un ensemble « branchements » entre des composants déjà créés.

Note importante : mon implémentation diffère cependant de l'architecture RP2 donnée. Par exemple il n'y a pas d'incrémenteur, celui-ci est intégré directement dans le program counter.

Execute :

Entité :

Nom	Type	Taille	Utilité
ird	IN unsigned	4 bit	Ces signaux sont retardés d'un cycle d'horloge dans le but d'atteindre l'étage d'accès à la mémoire en même temps que le résultat de l'étage execute. En effet, il est totalement possible de souhaiter stocker le résultat d'une addition en désignant le numéro de registre, et dans ce cas il ne faut pas que le numéro de registre soit utilisé pour l'instruction précédente.
ord	OUT unsigned		
iwrRegFile	IN unsigned	1 bit	
owrRegFile	OUT unsigned		
iselMuxMem	IN unsigned		
oselMuxMem	OUT unsigned		
iwrMem	IN unsigned		
owrMem	OUT unsigned		
idataOutValid	IN unsigned		
odataOutValid	OUT unsigned		
selMuxAlu	IN unsigned	2 bit	Désigne si l'on désire utiliser pour la suite le résultat de l'ALU, la donnée immédiate, ou bien l'opérande 2
codeOp		3 bit	Code de l'opération à réaliser (cf partie ALU)
op1	IN signed	2 octets	Mots binaires à traiter
op2			
immediateData			Valeur des constantes passées dans l'instruction
memAddress	OUT unsigned	1 octet	Adresse de la donnée où stocker le résultat
memData	OUT signed	2 octets	Entrée du data memory de l'étage memory access
DataOut			Port de sortie (vers un ou plusieurs périphériques)
clock	IN	bit	Synchronisation du composant
reset			Remet à zéro le composant (reset asynchrone)

Architecture :

Tout comme instruction fetch, ce composant est un assemblage de composants :

- ALU,
- Multiplexer,
- Registres.

Note : à partir de l'entrée **op1** (16 bit) une adresse **memAddress** (8 bit) doit transiter à destination du data memory si l'instruction est un JMP, BEQ ou BGT. D'après le format de stockage des données, et surtout « à cause » de l'extension de signe, les 8 bit d'adressages transmis seront les 8 LSB. En effet l'adresse est stockée de cette façon :

8 MSB								8 LSB							
S	S	S	S	S	S	S	S	A	A	A	A	A	A	A	A

S : bit d'extension du signe. A : bit d'adresse. Si nous prenions les 8 MSB nous aurions alors que le registre 0x00 ou bien le 0xFF.

Particularité du code : dans le code source on trouve des tableaux d'un seul élément non signé. Cela est dû à un problème de compatibilité entre les std_*logic* et les *unsigned (natural range <>)*. En effet j'ai utilisé le registre, de taille générique, pour permettre la synchronisation des signaux, j'ai donc concaténé tous les signaux à retarder pour n'utiliser qu'un seul registre au lieux d'en utiliser le nombre de signaux à retarder. Cependant, j'ai tout de même utilisé deux registres de plus pour la sortie du multiplexer

pour plus de clarté, en effet les signaux traités par les ceux derniers registres sont les résultats de l'ALU et non juste des signaux à synchroniser.

Access memory :

Entité :

Nom	Type	Taille	Utilité
ird	IN unsigned	4 bit	Ces signaux sont retardés d'un cycle d'horloge dans le but d'atteindre l'étage d'accès à la mémoire en même temps que le résultat de l'étage execute. En effet, il est totalement possible de souhaiter stocker le résultat d'une addition en désignant le numéro de registre, et dans ce cas il ne faut pas que le numéro de registre soit utilisé pour l'instruction précédente.
ord	OUT unsigned		
iwrRegFile	IN unsigned	1 bit	
owrRegFile	OUT unsigned		
iselMuxMem	IN unsigned		Dirige la donnée qui sera sur la sortie rebouclée à l'étage decode.
iwrMem	IN unsigned		Permet de signaler s'il faut lire ou écrire dans le data memory
address	IN unsigned	1 octet	Indique l'adresse de lecture ou d'écriture
data	IN signed	2 octets	Mot à écrire dans la mémoire
dataWriteBack	OUT signed		Données rebouclées à l'étage décode
clock	IN	bit	Synchronisation du composant
reset			Remet à zéro le composant (reset asynchrone)

Architecture :

Tout comme instruction fetch et execute, ce composant est un assemblage de composants :

- Mémoire de données,
- Multiplexer,
- Registres.

Dans cet étage aucun choix spécifique n'a été réalisé. Il a été très simple à assembler grâce à la documentation fournie.

III] Exhaustivité des testbench :

Composants de base :

Les testbench des composants de base ont été réalisé dans le but de tester le fonctionnement de chacun dans ses limites, i.e. : pour tester l'ALU la soustraction a été testée pour obtenir un nombre négatif. Cependant, ayant conscience que les erreurs ne sont pas traitées (cf la note sur les flags), je n'ai pas testé des cas d'overflow ou de carry.

- Pour le multiplexer une boucle a été utilisée avec des entrées différentes sur chacun des bus d'entrées. Le testbench teste donc toutes les possibilités de sorties (toutes les adresses possibles). J'ai testé avec des valeurs constantes en entrées (les entrées sont différentes pour chaque entrées, mais ne varie pas dans le temps). Cela aurait été inutile.
- Pour la mémoire de données je n'ai testé le fonctionnement sur qu'un seul registre, le premier. L'important pour moi était de vérifier si le comportement était celui attendu, plutôt que de vérifier de façon exhaustive chaque registre l'un après l'autre. Ceci aurait été plus complet et rapidement fait avec une boucle. Je le ferais si le temps me le permet (sachant qu'en plus j'ai modifié en générique cette entité, mais pas son testbench -qui prend tout de même la valeur par défaut si elle n'est pas renseignée-).
- Pour le program counter le comportement a été testé avec toutes les combinaisons des différentes valeurs de **stall** et de **load** ainsi que des valeurs en input différentes. L'entité a le comportement souhaité et la visualisation des chronogrammes avec gtkwave permet de bien le mettre en valeur.
- Pour le banc de registres....
- Pour l'extenseur de signe le test s'est réalisé avec 4 valeur différentes en entrées : 00000000, 10000000, 01111111, 10101010. A chaque nouvelle entrée la sortie était modifiée en fonction avec les bonnes valeurs. Encore une fois ici aussi nous aurions pu faire une boucle pour tester les 2^8 valeurs possibles, cependant tester les valeurs « limites » est pratique et plus rapide pour repérer d'éventuelles erreurs. De plus en réalisant des tests exhaustifs et complets se posent la question de savoir si la complexité du test ne le rend pas « faux ». On programme un composant. Une fois fait et qu'on le pense valide, on tente alors de le valider avec un autre programme qui le teste. Or rien ne certifie que le programme testant le composant est valide. Faire quelques cas limites et concret permet de trouver rapidement si le composant est valide ou non sans avoir à vérifier les 2^8 valeurs !

Étages du pipeline :

- Instruction memory : cet étage a été testé avec le programme fourni par l'enseignant qui se trouve dans le fichier : *instruction_memory-int.vhdl*. Pour tester on vérifie si le code de l'instruction sortante est bien celle attendue. Au démarrage ça doit être la première puis à chaque coup d'horloge l'instruction suivante doit apparaître. Les signaux **reset**, **stall** et **load** doivent évidement modifier la sortie en conséquence et ont été testés eux aussi.
- Execute : cet étage n'a pas été testé exhaustivement. Celui-ci a été testé avec un testbench le plus court possible en utilisant tous les composants. Sachant que l'ALU, les registres et le mux fonctionnent correctement il n'est pas utile de tester toutes les valeurs possibles pour toutes leurs entrées. J'ai donc testé que l'assemblage réalisé était valide en testant tous les registres sur un seul cycle

d'horloge, l'ALU pour une seule opération, et le mux pour deux valeurs différentes d'adresses.

- Acces memory : tout comme l'étage execute cet étage contient peut de composants différents et tous déjà testé. De ce fait je n'ai pas testé toutes les entrées possibles pour tous les composants mais seulement le comportement général. Une donnée est stockée dans le data memory, on vérifie qu'elle soit bien contenue en la lisant au front suivant, ensuite on modifie l'adresse du mux en testant son autre entrée.

IV] Divers :

Pendant un nop on fait quoi ?
Le décodage n'a pas été réalisé dans mon travail, cependant j'ai appris que la réalisation d'un 'nop' pose problèm (en parlant avec d'autres élèves, et me servant des cours de mon DUT GEII pour leur expliquer comment résoudre le problème). Lors d'un 'nop' le processeur est tout de même obligé de réaliser des opérations. La première, et vitale, est bien sûr l'incrémentation du program counter (sinon on risque de rien faire pendant longtemps!). Pour occuper le processeur on peut par exemple écrire la valeur d'un registre dans ce même registre.

Mes registres ont des signaux « reset » mais l'étage du pipeline qui en a besoin n'a pas de reset. Je fais comment ?
Tu ne fais pas :

- Soit un signal peut-être créé dans l'entité et initialisé de façon à ce que le registre ne reset jamais. Cela n'est absolument pas un problème ! Un exemple de ce « problème », plus haut niveau, est lorsque l'on déclare une variable en C et que celle-ci n'est pas initialisée alors sa valeur est une valeur aléatoire (valeur des données qui était à cette adresse mémoire). Ce n'est pas pour autant que nos PC explosent. Le programme ne lit pas au démarrage le contenu des registres.
- Soit une création d'un signal d'entrée reset qui sera relié au registre.

Aucune solution n'est mieux ou bien plus propre. Dans un cas on sait que les contenus des registres ne seront pas lus, dans l'autre on les initialise à zéro.

J'ai des warning meta-value, c'est grave ? Ca peut exploser ?
Les metavalues peuvent poser problème si elles se trouvent sur un signal « important ». Si une metavalue est présente sur le port de sortie du processeur alors on peut imaginer que le périphérique branché à cet endroit prennent ces données comme une instruction par exemple, et cela peut alors poser problème ! Cependant si une metavalue est détectée à l'entrée d'un registre alors, pour les mêmes raisons expliquées ci-dessus, celle-ci peut être ignorée en toute sécurité.

Conclusion :

Ce projet nous a apporté un compréhension très bas niveau d'un processeur. Il est certes simplifié mais assez complet pour nous avoir confronté à des problèmes concrets. Cette compréhension est tout de même importante à avoir en tant qu'informaticien. En effet, ne pas connaître le bas niveau pour nous est comme si un pilote de rallye ne connaissait pas la mécanique de la voiture qu'il conduit !

La liberté laissé par notre enseignant nous a parfois ralenti, néanmoins elle nous a permis de nous poser les bonnes questions, et de les résoudre par nous même.

Je n'ai hélas pas fini le projet (la partie de décodage et l'assemblage final n'ont pas été réalisé). Je regrette un peu de ne pas en avoir eu le temps.

La réalisation de ce projet en binôme, comme elle l'était les années passées, aurait aussi permis à plus de personne de finaliser le projet -je ne pense pas être le seul à ne pas l'avoir fini- et l'équité aurait été plus grande car certains ont tout de même travaillé en étroite collaboration, ont donc pu aller plus loin et donc assimiler plus.

En tant qu'ancien GEII ce projet m'a beaucoup plu, cependant je trouve dommage que l'enseignement qu'on ait eu en parallèle, dans la matière Informatique Industrielle, n'ait pas collaboré avec l'architecture machine. Cela aurait pu enrichir encore plus le projet et les connaissances acquises.