ASR1 – TD11 : Un vrai RISC dans mon FPGA !

{ Andreea.Chis, Matthieu.Gallet, Bogdan.Pasca } @ens-lyon.fr

11-12, 18-19, décembre 2008

Nous allons donc réaliser ici un vrai processeur RISC 16 bits, plus précisément celui que nous sommes parvenus (non sans mal !) à spécifier lors des derniers TD. Pour ce faire, nous allons le décrire intégralement en VHDL, puis nous l'implémenterons sur un FPGA (*Field-Programmable Gate Array*), un circuit intégré complètement programmable capable d'implémenter n'importe quel circuit.

1 FPGA

Cette section n'a pas vocation à être un cours complet sur les FPGA, mais juste une présentation rapide pour que vous ayiez une idée de la manière dont votre processeur RISC va être implanté.

1.1 Vue générale

Le paradigme qui se cache derrière les FPGA est le parallélisme poussé à l'extrême, jusqu'au niveau le plus fin (on parle de *fine-grain parallelism*) : il s'agit d'avoir un grand nombre de toutes petites unités de calcul programmables, reliés entre eux par suffisamment de fils pour ne pas risquer de congestion.

Ces unités de calcul (ou PE, pour *processing elements*) sont toutes identiques et généralement regroupées en îlots de 4 ou 8 PE en général. Ces îlots sont ensuite disposés sur une grille bidimensionnelle : un FPGA classique en compte plusieurs milliers[1].

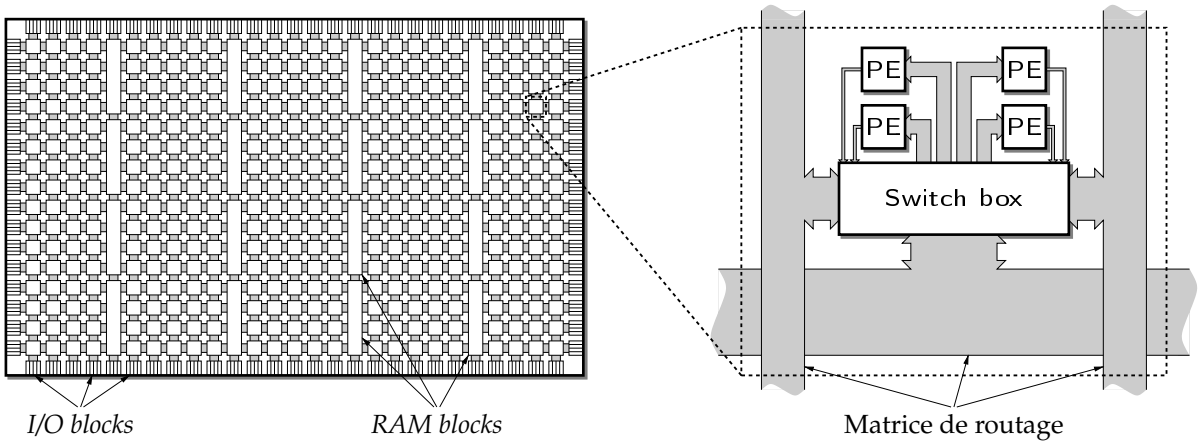
Les îlots sont ensuite reliés entre eux grâce à une matrice de routage programmable, permettant ainsi d'effectuer toutes les connections nécessaires entre les îlots d'un circuit, et ce, où qu'ils soient situés sur le FPGA.

On trouve aussi sur tout le pourtour du FPGA des ports d'entrée/sortie (*I/O blocks*) eux aussi programmables et accessibles par la matrice de routage, permettant ainsi de lire ou d'envoyer des signaux sur les plots du FPGA, et donc de l'interfacer avec le monde extérieur.

Généralement, un FPGA contient aussi de quelques dizaines à plusieurs centaines de petits blocs de RAM (*RAM blocks*) de 16 ou 18kbits chacun, disposés en colonnes régulièrement réparties sur la surface de la puce.

Sur les modèles les plus récents, on y trouve aussi des petits multiplieurs 18×18 bits, voire carrément des cœurs de processeurs classiques (PowerPC ou ARM).

Voici de manière très simplifiée l'architecture globale d'un FPGA, ainsi que le détail d'un îlot :

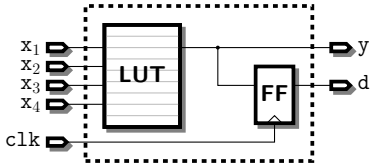


[1] Celui que nous allons utiliser en a $13\,696$, mais c'est un gros modèle.

1.2 Anatomie d'un PE

En simplifiant là aussi grandement, un PE peut être réduit à une petite mémoire programmable de $2^4 \times 1$ bit (ou LUT, pour *look-up table*), à laquelle est adjoint un registre de type flip-flop (FF).

La table permet donc d'implanter n'importe quelle fonction booléenne à 4 variables en stockant dans la LUT la table de vérité de cette fonction. Le bit du résultat de la table peut alors soit être stocké dans le registre, soit être routé directement vers un autre PE pour la suite du calcul.



Ainsi, grâce à la matrice de routage, on peut cascader les PE pour calculer n'importe quelle fonction booléenne, et les FF nous permettent de créer des registres pour nos signaux. A priori, on dispose là de tout ce qu'il nous faut pour y implanter notre processeur RISC !

2 Notre plateforme

2.1 Vue d'ensemble

Le FPGA sur lequel nous allons travailler se trouve sur une carte de développement munie de plein de ports d'entrée/sortie tels que USB, RS232 (port série), PS/2, audio, SATA, VGA,... ainsi que d'un emplacement pour une barrette de mémoire DDR. Tous ces ports et périphériques sont bien entendu directement interfacés avec le FPGA grâce aux I/O blocks de celui-ci. La carte dispose en plus d'une horloge cadencée à 100MHz, reliée elle aussi à un port d'entrée du FPGA.

2.2 Contrôleur vidéo

Le contrôleur vidéo affiche à l'écran une résolution de 640×480 pixels en 24 bits / pixel (16 millions de couleurs).

Il s'agit en effet du contrôleur étudié par le premier DM. Ce contrôleur est mappé sur l'espace d'adressage correspondant à : 001X XXXX XXXX XXXX

2.3 Contrôleur PS/2

Le protocole PS/2 est un protocole série sur deux fils, à la I^2C, le maître étant le périphérique. Ce bus est bidirectionnel : le clavier (ou la souris) peut écrire sur le bus, mais peut aussi recevoir des données, comme par exemple l'état allumée/éteinte des LED sur le clavier. Le contrôleur écrit pour ce TD étant très simpliste, il se contente d'écouter sur le bus, mais n'y écrit jamais rien.

Les messages envoyés par le périphérique font toujours 11 bits de long : 1 bit de début de transmission, 8 bits de données, 1 bit de parité impaire et enfin 1 bit de fin de transmission.

De manière à ne pas avoir à gérer les phénomènes d'interruptions dans notre processeur, le contrôleur PS/2 fonctionnel est equipé avec une file d'attente cyclique, longue de 254 éléments, cache pour le processeur. Dans cette file d'attente ne seront stockés que les 8 bits de données de chaque transmission du périphérique.

Ce contrôleur est mappé sur l'espace d'adressage correspondant à :
010X XXXX XXXX XXXX

2.4 Mémoire

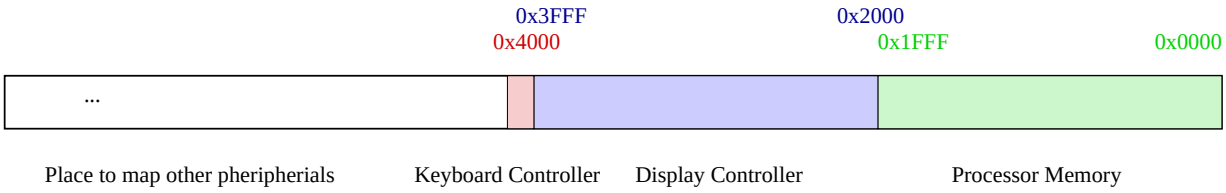
La mémoire DDR étant réellement une horreur à gérer, nous allons nous contenter d'utiliser la mémoire disponible sur le FPGA sous forme de RAM blocks de 16kbits. En effet, ces blocs ont le mérite d'être rapides d'accès, mais aussi et surtout de disposer de deux paires indépendantes de ports lecture/écriture. Ainsi, nous n'aurons pas à nous soucier des accès simultanés à la mémoire. De plus, les RAM blocks proposent plusieurs modes d'adressage, de $16\text{k} \times 1$ bit à 512×32 bits, ce qui va nous permettre de gérer différentes tailles de données de manière transparente.

Comme le processeur RISC que nous allons implanter est un processeur 16 bits, les mots mémoire font 16 bits de même que les adresses. L'espace mémoire maximal est donc $2^{16} \times 16$ bits.

Comme nous ne disposons pas de suffisement de RAM blocks pour réaliser cette mémoire, nous allons nous limiter à l'utilisation de 8 RAM blocks de $1K \times 16$ bits. Nous ne pourrons donc addresser que 2^{13} mots de 16 bits.

L'interconnexion des blocs se fait d'une manière transparente en VHDL, en instantiant simplement une mémoire de $2^13 \times 16bits$

On a donc le mapping suivant, avec la zone de mémoire libre de `0x4001` à `0xFFFF` :



2.5 Processeur RISC 16 bits. Nom de code TRAJAN

Voici une spécification complète du processeur tel qu'il a été défini lors des TD précédents, avec ce que nous avons modifié.

2.5.1 Découpage d'une instruction

Plusieurs registres jouent un rôle particulier :
- **PC** : Program Counter
- **ACC** : ACCumulateur, alias $\mathbf{R}_0$
- **SP** : Stack Pointer, alias $\mathbf{R}_{127}$
- **SR** : Status Register à 3 bits, contenant les flags **Z**, **V**, **N**

Le découpage d'une instruction se fait de la façon suivante :
- 7 bits sont utilisés pour numéroter les registres,
- 1 bit supplémentaire signale si **SR** est mis à jour par l'instruction,
- 5 bits sont consacrés à la définition de l'instruction, ce qui laisse 32 instructions possibles,
- 3 bits sont réservés à la condition d'exécution de l'instruction :
  1. **GT** : > 0 **N** = **V** et **Z** $= 0$
  2. **GE** : ≥ 0 **N** = **V**
  3. **EQ** : $= 0$ **Z** $= 1$
  4. **NE** : $\neq 0$ **Z** $= 0$
  5. **LE** : ≤ 0 **Z** $= 1$ ou **N** $\neq$ **V**
  6. **LT** : < 0 **N** $\neq$ **V**
  7. **VS** : dépassement de capacité **V** $= 1$
  8. **NC** : le branchement se fait toujours

3 Liste des instructions

L'instruction est conditionné par les 3 bits `ccc`, et **SR** ne sera changé que si `s` est à 1.

1. `00000cccvvvvvvvv` (**LoadLo**) : charge les 8 bits `v` dans la partie basse de **ACC** et efface sa partie haute,
2. `00001cccvvvvvvvv` (**LoadHi**) : charge les 8 bits `v` dans la partie haute de **ACC**,
3. `00010ccc0iiiiiii` (**MoveAccToReg**) : recopie le contenu de **ACC** dans $\mathbf{R}_i$,
4. `00011ccc0iiiiiii` (**MoveRegToAcc**) : recopie le contenu $\mathbf{R}_i$ dans **ACC**,
5. `00100ccc0iiiiiii` (**Read**) : recopie le contenu de la mémoire à l'adresse **ACC** dans $\mathbf{R}_i$,
6. `00101ccc0iiiiiii` (**Write**) : recopie le contenu de $\mathbf{R}_i$ dans la mémoire à l'adresse **ACC**,
7. `00110ccc0iiiiiii` (**ReadInc**) : recopie le contenu de la mémoire à l'adresse **ACC** dans $\mathbf{R}_i$ et incrémente **ACC**,
8. `00111ccc0iiiiiii` (**WriteInc**) : recopie le contenu de $\mathbf{R}_i$ dans la mémoire à l'adresse **ACC** et incrémente **ACC**,
9. `01000ccc0iiiiiii` (**ReadDec**) : recopie le contenu de la mémoire à l'adresse **ACC** dans $\mathbf{R}_i$ et décrémente **ACC**,
10. `01001ccc0iiiiiii` (**WriteDec**) : recopie le contenu de $\mathbf{R}_i$ dans la mémoire à l'adresse **ACC** et décrémente **ACC**,
11. `01010ccc0iiiiiii` (**Jmp** – JuMP) : recopie le contenu de $\mathbf{R}_i$ dans **PC**,
12. `01011ccc0iiiiiii` (**Jmr** – JuMp Relative) : ajoute le contenu (signé) de $\mathbf{R}_i$ à **PC**,
13. `01100cccvvvvvvvv` (**Jmi** – JuMp Immediate) : charge les 8 bits `v` dans **PC**,
14. `01101cccvvvvvvvv` (**Jmri** – JuMp Relative Immediate) : ajoute les 8 bits `v` (signés) à **PC**,
15. `01110ccc0iiiiiii` (**Jsr** – Jump to SubRoutine) : recopie **PC** à l'adresse **SP**, incrémente **SP**, recopie le contenu de $\mathbf{R}_i$ dans **PC**,
16. `01111cccvvvvvvvv` (**Jsi** – Jump to Subroutine Immediate) : recopie **PC** à l'adresse **SP**, incrémente **SP**, ajoute les 8 bits `v` (signés) à **PC**,
17. `10000ccc00000000` (**Rts** – ReTurn from Subroutine) : décrémente **SP**, recopie le contenu incrémenté de l'adresse **SP** dans **PC**
18. `1000100000000000` (**Nop** – No OPeration) : no operation
19. `10010cccsiiiiiii` (**Add**) : additionne le contenu de $\mathbf{R}_i$ à **ACC**,
20. `10011cccsiiiiiii` (**Sub**) : soustrait le contenu de $\mathbf{R}_i$ à **ACC**,
21. `10100cccsiiiiiii` (**Mul**) : multiplie **ACC** par $\mathbf{R}_i$,
22. `10101cccsiiiiiii` (**Cmp** – CoMPare) : compare $\mathbf{R}_i$ à **ACC**,
23. `10110cccsiiiiiii` (**Swap**) : échange les parties haute et basse de $\mathbf{R}_i$,
24. `10111cccsiiiiiii` (**Clr** – CLeaR) : efface le contenu de $\mathbf{R}_i$,
25. `11000cccsiiiiiii` (**And**) : copie le ET logique de **ACC** et $\mathbf{R}_i$ dans **ACC**,
26. `11001cccsiiiiiii` (**Or**) : copie le OU logique de **ACC** et $\mathbf{R}_i$ dans **ACC**,
27. `11010cccsiiiiiii` (**And**) : copie le XOU logique de **ACC** et $\mathbf{R}_i$ dans **ACC**,
28. `11011cccsiiiiiii` (**Not**) : calcule le NON logique de $\mathbf{R}_i$ dans **ACC**,
29. `11100cccsiiiiiii` (**Lsr** – Logical Shift Right) : décale **ACC** de $\mathbf{R}_i$ vers les bits de poids faible,

30. `11101cccsiiiiiii` (**Lsl** – Logical Shift Left) : décale **ACC** de $\mathbf{R}_i$ vers les bits de poids fort,
31. `11110cccsiiiiiii` (**Ror** – ROtate Right) : rotation de **ACC** de $\mathbf{R}_i$ vers les bits de poids faible,
32. `11111cccsiiiiiii` (**RoL** – ROtate Left) : rotation de **ACC** de $\mathbf{R}_i$ vers les bits de poids fort.

3.0.2 L'architecture TRAJAN

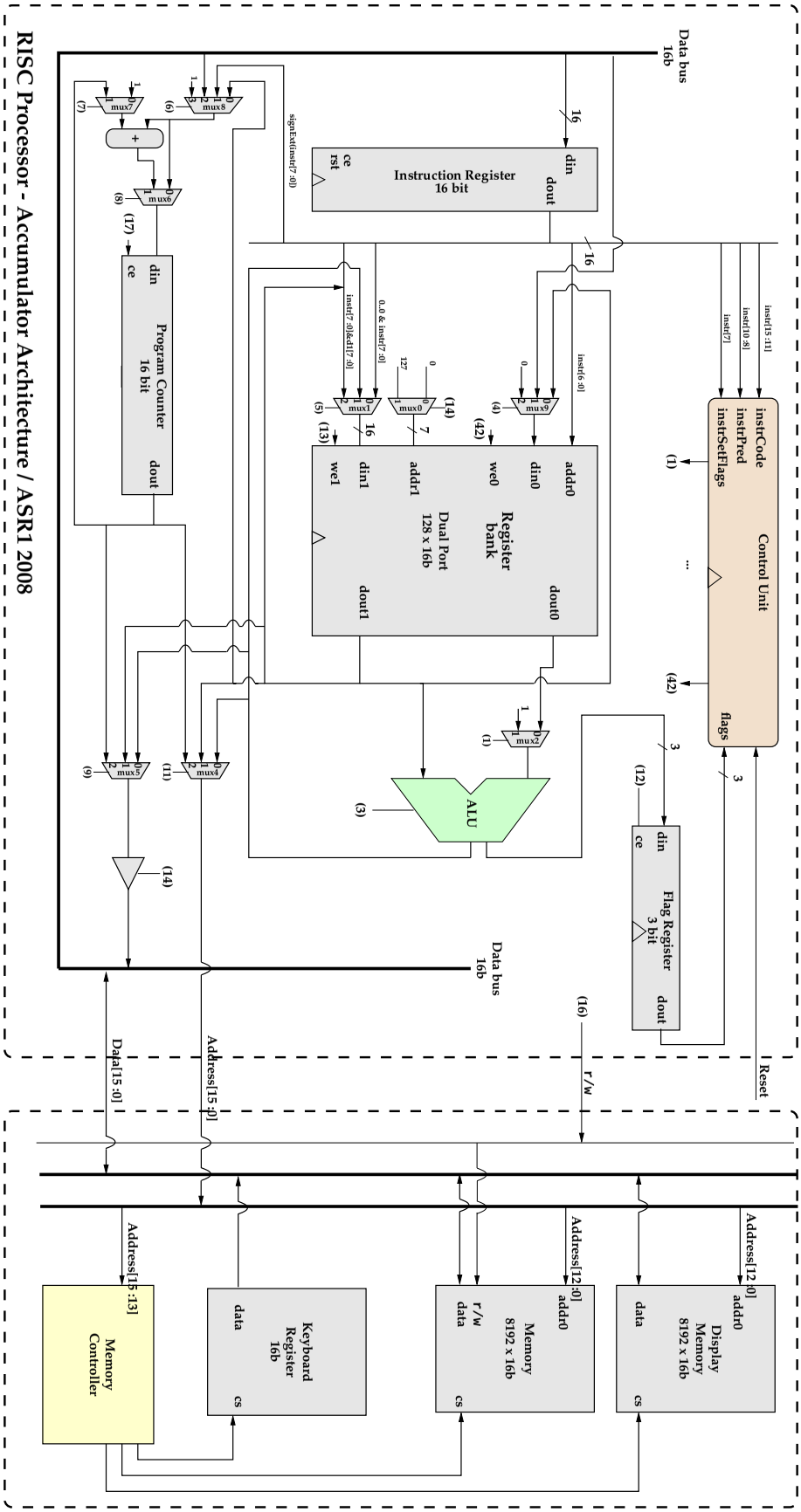
4 À vous de jouer

Pour réaliser ce processeur, vous êtes divisés en quatre groupes :
- le groupe A (le jeudi) est chargé de réaliser les registres (instruction, prédicats et banc de registres), mémoires, multiplexeurs, etc.
- le groupe B (le vendredi) s'occupe de l'ALU,
- le groupe C (le jeudi) réalise la partie contrôle,
- et le groupe D (le vendredi) s'occupe du contrôleur video.
- Pour finir, le 8-9 Janvier, on va assembler le processeur complet.

4.1 Architecture globale du processeur

Il s'agit ici de relier tous les composants (registres, ALU, contrôle) entre eux, en insérant multiplexeurs et registres correctement contrôlés aux endroits nécessaires. Il suffit en gros de suivre le schéma global du processeur donné précédemment.

Attention : lors d'un *reset*, le `PC` doit être initialisé à l'adresse du début du programme, soit `0x0800`, sinon on ne tombera pas dans la section de code !


RISC Processor - Accumulator Architecture / ASR1 2008
Data bus 16b
Instruction Register 16 bit
Program Counter 16 bit
Control Unit
instrCode
instrPred
instrSetFlags
flags
Register bank
Dual Port 128 x 16b
we0
din0
addr0
dout0
we1
din1
addr1
dout1
ALU
Flag Register 3 bit
mux0
mux1
mux2
mux3
mux4
mux5
mux6
mux7
mux8
signExt(instr[7:0])
instr[7:0]&cf[7:0]
0.0 & instr[7:0]
instr[6:0]
instr[15:11]
instr[10:8]
instr[7]
Data[15:0]
Address[15:0]
r/w
Reset
Memory Controller
Address[15:13]
Keyboard Register 16b
data
cs
Memory 8192 x 16b
r/w
Address[12:0]
addr0
Display Memory 8192 x 16b