# ASR1 – TD7 : Un microprocesseur RISC 16 bits

{ Jeremie.Detrey, Patrick.Loiseau, Nicolas.Veyrat-Charvillon } @ens-lyon.fr

http://perso.ens-lyon.fr/jeremie.detrey/06_asr1/

13, 20 et 27 novembre 2006

## Présentation générale

On choisit un format de 16 bits pour la simplicité du dessin final. Pour respecter la philosophie RISC, adresses et données sont donc codées sur 16 bits. Toutes les instructions machines sont également codées sur 16 bits, *opérandes compris*.

La première partie consiste à définir le jeu d'instructions, le format du mot d'instructions, et à l'essayer sur quelques exemples. La seconde partie construit le processeur lui-même.

1. Quel est l'espace d'adressage de ce processeur ?
2. Quelle est la taille mémoire qu'il peut adresser ?
3. Combien d'instructions aura-t-il au maximum ?
4. Dessinez la boîte noire de ce processeur, comportant tous les signaux d'interface avec la mémoire (on ignore les questions d'*interruptions* des processeurs réels).

## 1 Le jeu d'instructions

On va découper le mot d'instruction en différents *champs* codant (entre autres) l'instruction à effectuer, ses différents opérandes, etc... On s'attachera à respecter le principe d'*orthogonalité* de la philosophie RISC, qui dit que ce découpage doit être constant, même pour des instructions très différentes.

1. Justifiez le principe d'orthogonalité.

### 1.1 Choix du nombre de registres

Il y a trois grandes architectures possibles pour la partie calcul d'un processeur :

- *Machine à trois adresses* : implémente des instructions de type `Rd` ← `Rx op Ry`, où `Rd` est le registre destination, `Rx` et `Ry` sont les registres opérandes, et `op` est une opération arithmétique ou logique.
- *Machine à deux adresses* : le registre destination est obligatoirement un des registres opérandes, donc les instructions sont du type `Rx` ← `Rx op Ry`.
- *Machine à une adresse*, ou *machine à accumulateur* : toute opération met en jeu un registre spécial, l'accumulateur (noté `Acc`), qui est à la fois destination et l'un des opérandes, soit `Acc` ← `Acc op Ry`.

1. Donnez des exemples de processeurs (réels) qui sont des machines à trois, deux ou une adresse.
2. Peut-on imaginer une machine à zéro adresse ?
3. Donnez le nombre de bits que va nécessiter le codage des opérandes et de la destination dans notre mot d'instruction, en fonction du choix d'architecture et du nombre de registres du processeur.
4. Discutez et choisissez.

### 1.2 Opérations mémoire

La philosophie RISC distingue bien les opérations de calcul, dont opérandes et destination ne sont que des registres, et les opérations d'accès à la mémoire, qui n'effectuent aucun calcul.

1. Discutez les avantages et inconvénients de cette approche.

On définit les deux opérations mémoire `LDR` (*LoaD to Register*) et `STR` (*STore Register*) qui chargent ou déchargent le contenu d'un registre en mémoire à une adresse particulière.

2. Définissez le fonctionnement de ces opérations.
3. Considérez les divers moyens d'implémenter une instruction `JMP` (*JuMP*). Choisissez.

### 1.3 Choix des instructions arithmétiques et logiques

Nous allons maintenant définir les quelques bits de notre mot d'instruction codant l'instruction (le champ instruction).

En plus des opérations sur la mémoire et les sauts précédents, on aura des opérations de calcul entier, de calcul binaire, une instruction de copie de registres, des décalages, etc...

1. Définissez un jeu d'instructions minimal (ou en tout cas très *Reduced*), mais permettant par exemple de programmer en assembleur multiplication et division (même si cet exercice n'est pas complètement possible à ce stade).
2. Si le nombre d'instructions obtenu est différent de 8, 16 ou 32, ajoutez ou on retirez des instructions jusqu'à bien remplir le champ instruction.
3. Discutez les différentes possibilités pour faire des opérations par des constantes.
4. Récapitulez précisément le jeu d'instructions et discutez son codage le plus simple possible dans le champ instruction.

### 1.4 Contrôle d'exécution

Il doit normalement nous rester quelques bits libres parmi les 16 de notre mot d'instruction. Ils vont servir à contrôler l'exécution.

Autrefois seul le branchement était conditionnel. Soyons modernes : par souci d'orthogonalité, toutes les instructions seront identiquement conditionnelles.

#### 1.4.1 Par des drapeaux (comme l'ARM)

Un champ condition, dans notre mot d'instruction, contiendra un codage de la condition sous laquelle l'instruction sera effectuée.

Les quatre drapeaux de base de toute UAL sont :
- $Z$ (*Zero*), qui vaut 1 si le résultat d'une opération est nul, et 0 sinon ;
- $N$ (*Negative*), qui vaut 1 si le résultat est strictement négatif, et 0 sinon ;
- $C$ (*Carry*), qui vaut 1 s'il y a dépassement de capacité sur 16 bits, et 0 sinon.
- $V$ (*oVerflow*), qui vaut 1 s'il y a dépassement de capacité sur 15 bits (dépassement signé), et 0 sinon.

**1.** Donnez une définition précise (en fonction du résultat du calcul) pour chacun de ces drapeaux.

**2.** Exprimez les conditions suivantes en fonction des drapeaux : résultat positif ou nul, négatif ou nul, strictement positif, strictement négatif, dépassement de capacité, pas de dépassement de capacité, etc...

**3.** Combien de bits faut-il pour coder un ensemble minimal de conditions (n'oubliez pas le "sans condition") ?

**4.** Définissez précisément le champ condition, et les mnémoniques correspondants.

#### 1.4.2 Par des prédicats (comme l'IA64)

Dans le jeu d'instruction IA64, il y a 64 *registres de prédicats* d'un bit qui peuvent être mis à jour par certaines instructions de comparaisons. Toutes les instructions peuvent être *prédiquées* par l'un de ces registres. Si vous n'avez pas compris demandez des explications.

**1.** Reprenez les questions ci-dessus.

**2.** Faites un choix.

### 1.5 Récapitulation et bouche-trous

**1.** Récapitulez le mot d'instruction jusqu'ici. Reste-t-il des bits inutilisés ? Si oui, voici des suggestions d'utilisation :
- On peut définir un bit qui dit si une instruction met à jour les drapeaux, ou pas. Ceci sera utile pour des séquences de code machine du type *si alors sinon*.
- On peut définir un bit qui dit si le résultat d'une opération est écrit dans le registre résultat, ou pas. Ceci permet de transformer toute instruction arithmétique ou logique en une instruction de test : par exemple il transforme `SUB` en `CMP` (*CoMPare*).
- On peut faire jouer son imagination.

**2.** Complétez le jeu d'instruction, vérifiez qu'il n'y a pas (trop) de redondance dans les instructions, récapitulez.

### 1.6 Test : programmation en assembleur

**1.** Pour se convaincre de la qualité de notre jeu d'instruction, écrire un programme réalisant la multiplication, un réalisant la division, puis un Pac-Man (ou Quake III si vous préférez).

**2.** Corrigez les questions précédentes en fonction des oublis.

# 2 Construction du processeur

Les paragraphes 2.1 à 2.3 doivent être étudiés plus ou moins en parallèle et itérés jusqu'à obtention d'un résultat satisfaisant.

## 2.1 Architecture générale

1. Placez sur une grande feuille les différents composants de notre processeur :
   - les bus (adresse et données) d'interface à la mémoire externe ;
   - le banc de registres du processeur (combien de ports pour cette mémoire ?) ;
   - la boîte noire de l'UAL, avec ses signaux de commande ;
   - le registre des drapeaux, et la boîte noire du contrôle d'exécution ;
   - le registre qui contiendra le mot d'instruction.

Attention, n'oubliez pas les drapeaux et autres bits pour la boîte noire de l'UAL. N'essayez pas de connecter les différents blocs à ce stade.

## 2.2 Le cycle d'instruction

1. Décrivez le cycle de von Neumann de ce processeur, et vérifiez que tous les blocs nécessaires sont présents.
2. Discutez, en fonction de l'occupation des différents blocs et des différents fils qui vont les relier, des questions du type : quand le PC est-il incrémenté ?

Attention, les choses à faire dans un cycle diffèrent selon que l'opération est une opération arithmétique ou logique ou bien un accès mémoire.

## 2.3 Définition des signaux de commande

1. Tirez des fils entre ces différents blocs, et placez lorsque nécessaire des multiplexeurs commandés par des signaux binaires.
2. Donnez des numéros au hasard à chacun de ces signaux de commande.
3. Donnez également des numéros aux différents signaux de commande des différents blocs (signaux *read* et *write* à destination de la mémoire externe, bits de l'UAL, etc... Si l'on en oublie on pourra toujours les rajouter par la suite).
4. Extasiez-vous devant les bienfaits du principe d'orthogonalité.

Le reste de cette partie consiste en la définition de l'automate qui va activer ces différents signaux dans le bon ordre pour assurer le fonctionnement du processeur.

## 2.4 Synthèse de l'automate de commande

1. Définissez très précisément un cycle d'instruction (en moins de 10 étapes, entre deux et six paraît un bon nombre). On pourra considérer que le délai de la mémoire est équivalent à celui de l'UAL, et toute autre hypothèse simplificatrice raisonnable. Attention, ce qui est fait à chaque étape dépend de l'instruction.
2. Construisez l'automate très simple qui, à partir d'une horloge unique, produit $n$ signaux binaires $\phi_i$, chacun actif au cours d'une étape.

3. Écrivez un tableau donnant, en fonction de l'instruction, ceux des signaux de commande numérotés qui sont actifs à chaque étape.
4. Implémentez cette table comme un bloc combinatoire prenant en entrée les $\phi_i$ et les autres signaux utiles, et produisant les signaux de commande numérotés.
5. Placez ce bloc sur le schéma (il n'est pas utile de *dessiner* tous les fils des signaux de commande).

### 2.5 Remplissage des boîtes noires

1. Construisez l'intérieur de la boîte noire de l'UAL.
2. Construisez l'intérieur de la boîte noire de contrôle d'exécution.
3. Construisez l'intérieur des autres boîtes noires.

## 3 C'est fini (ou presque)

1. Estimez très grossièrement le nombre de portes.
2. Réfléchissez à comment décrire tout ça en VHDL.