# ASR1 – TD9 : Un microprocesseur RISC 16 bits

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

27 et 28 novembre 2008

## 1 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 Définition des signaux de commande

3. Tirez des fils entre ces différents blocs, et placez lorsque nécessaire des multiplexeurs commandés par des signaux binaires.
4. Donnez des numéros au hasard à chacun de ces signaux de commande.
5. 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'ALU, etc... Si l'on en oublie on pourra toujours les rajouter par la suite).
6. 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.

## 3 Synthèse de l'automate de commande

7. 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'ALU, et toute autre hypothèse simplificatrice raisonnable. Attention, ce qui est fait à chaque étape dépend de l'instruction.
8. 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.
9. Écrivez un tableau donnant, en fonction de l'instruction, ceux des signaux de commande numérotés qui sont actifs à chaque étape.

**10.** 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.

**11.** Placez ce bloc sur le schéma (il n'est pas utile de *dessiner* tous les fils des signaux de commande).

## 4 Remplissage des boîtes noires

**12.** Construisez l'intérieur de la boîte noire de l'ALU.

**13.** Construisez l'intérieur de la boîte noire de contrôle d'exécution.

**14.** Construisez l'intérieur des autres boîtes noires.

## 5 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** : $\geq 0$ **N** = **V**
  3. **EQ** : $= 0$ **Z** $= 1$
  4. **NE** : $\neq 0$ **Z** $= 0$
  5. **LE** : $\leq 0$ **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

## 6 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.

# 7 Schéma complet du processeur

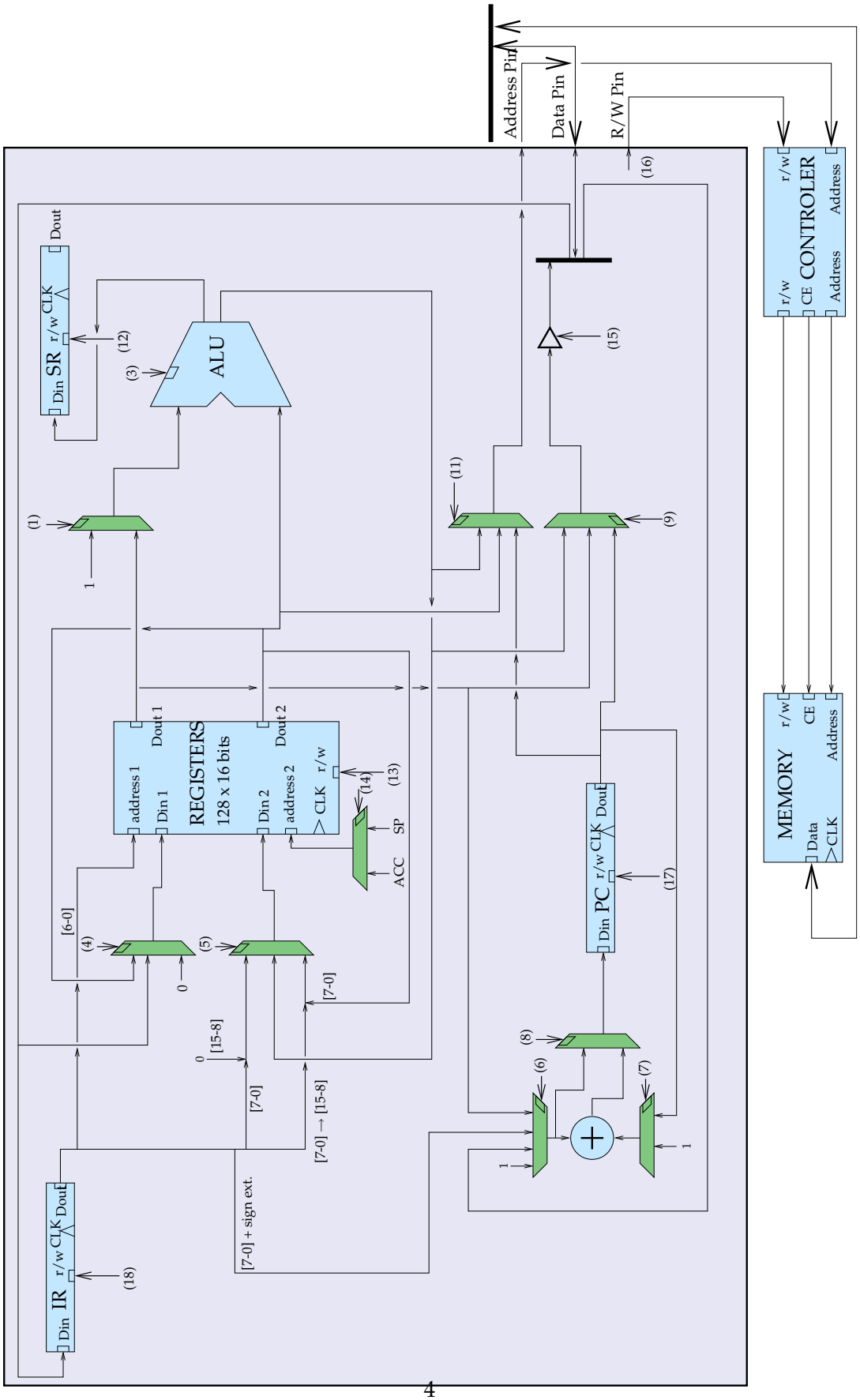


FIG. 1 – Schéma du processeur