université de BORDEAUX

Architecture des ordinateurs : Câblage des processeurs dans l'environnement « *y86* »

(INF155)

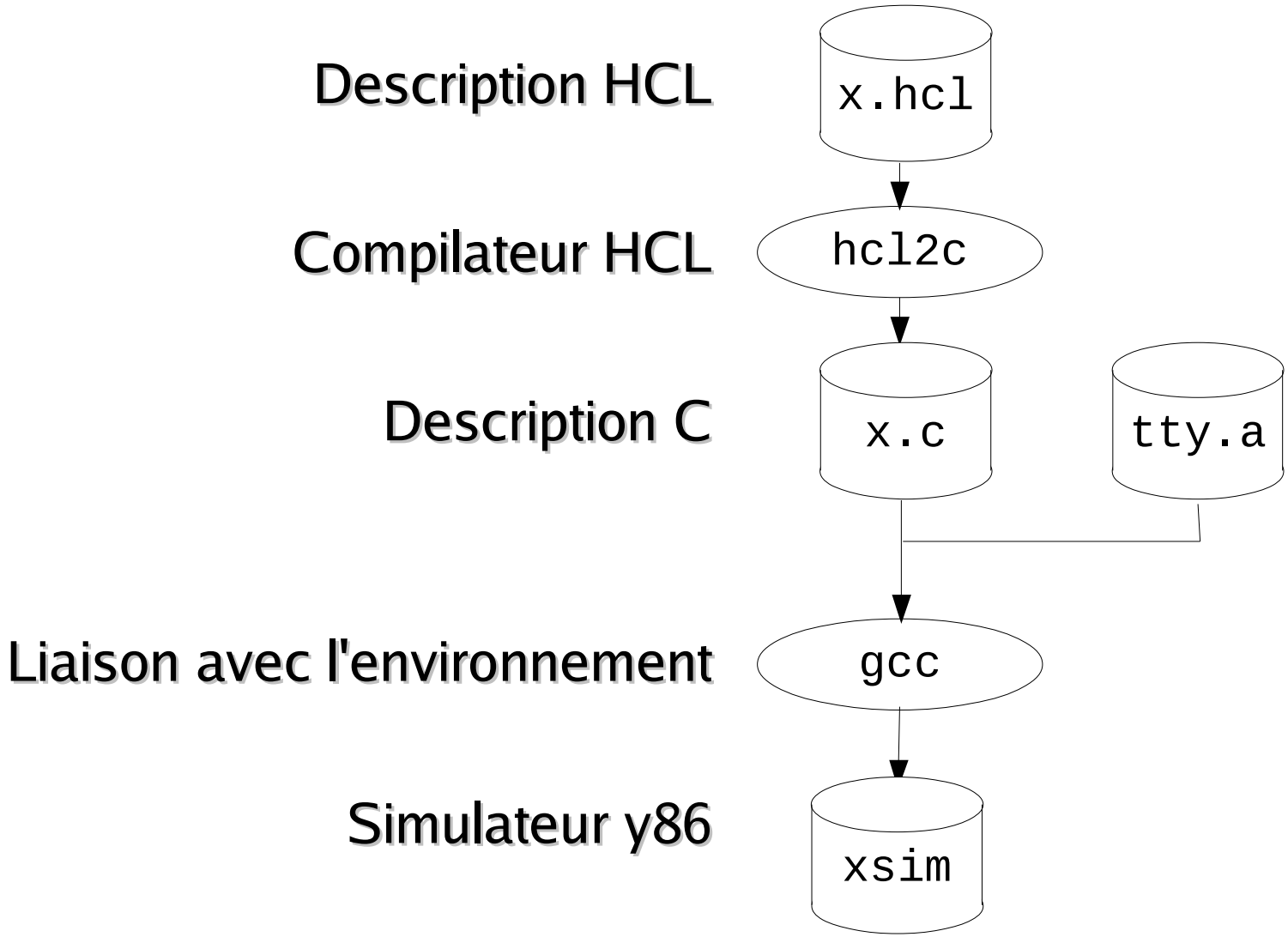
F. Pellegrini
Université de Bordeaux

Modélisation du câblage du y86

- L'environnement pédagogique y86 couvre aussi le câblage du processeur y86 lui-même
- Câblage modélisé par un langage dédié : HCL (« *Hardware Control Language* »)
  - Simplification de langages tels que Verilog et VHDL
- Compilation du HCL en C et intégration du code produit au sein de l'émulateur y86
  - Permet de mettre en œuvre plusieurs câblages du même jeu d'instructions

http://csapp.cs.cmu.edu/

Utilisation de HCL



Syntaxe HCL (1)

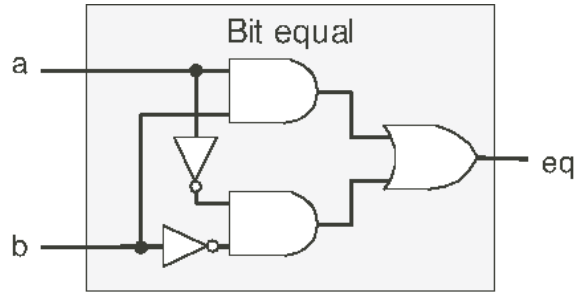
- HCL permet de représenter le câblage des blocs fonctionnels du processeur y86
  - Syntaxe proche du langage C
- Types de données supportés :
  - Booléens : type « bool »
    - Signaux pilotant le fonctionnement des unités fonctionnelles du processeur
  - Valeurs entières (au plus 32 bits) : type « int »
    - Servent à coder les opérandes, adresses, …

Syntaxe HCL (2)

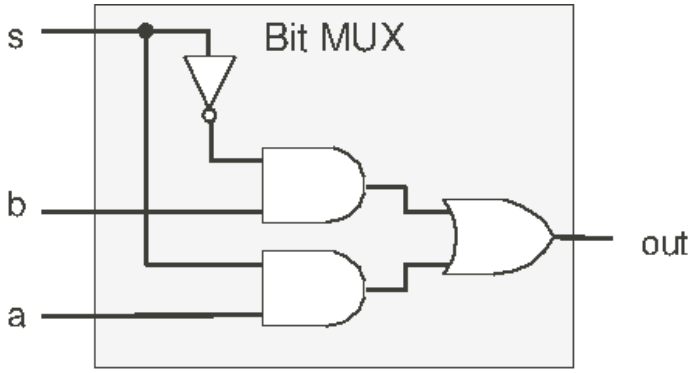
- Opérations supportées :
  - Fonctions logiques sur des valeurs booléennes
  - Fonctions logiques sur des valeurs entières
- Certains blocs fonctionnels ne font pas partie du périmètre du langage :
  - Banque multi-ports des registres (« *register file* »)
  - Mémoire centrale (instructions et données)
  - UAL

Syntaxe HCL (3)

- Fonctions logiques sur des valeurs booléennes retournant un booléen :
  - bool eq = (a && b) || (!a &&!b) ; # Test d'égalité

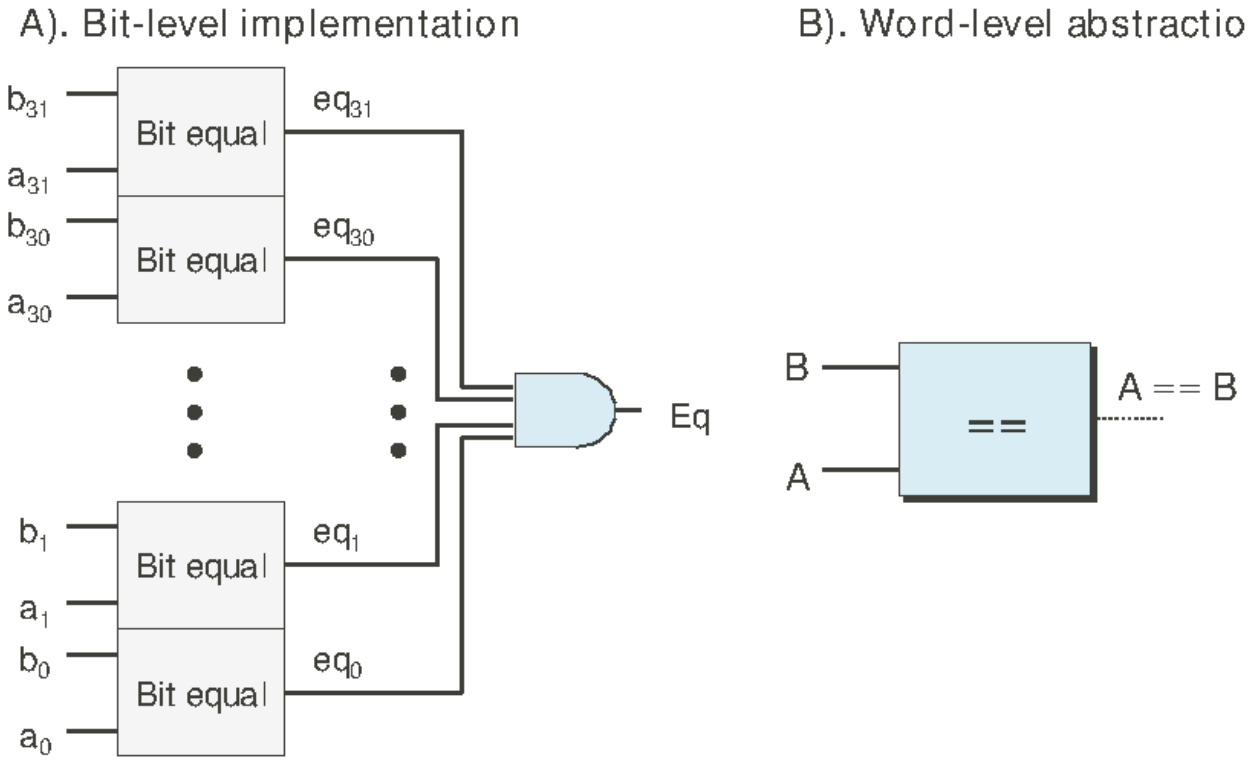


  - bool out = (s && a) || (!s && b) ; # Multiplexeur 1 bit



Syntaxe HCL (4)

- Fonctions logiques sur des valeurs entières retournant un booléen :
  - bool eq = (A == B) ; # Existent aussi : <, <=, >, >=,!=

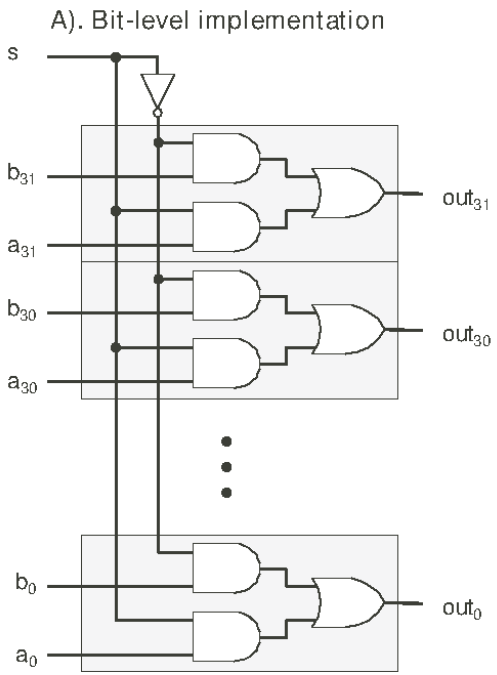


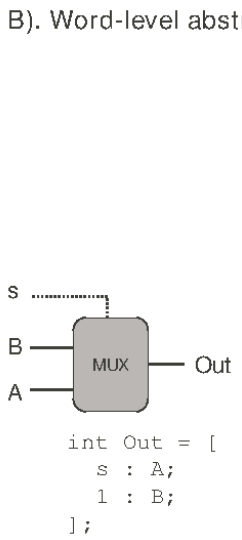
Syntaxe HCL (5)

- bool s1 = (code == 2) || (code == 3) ; # Au cas par cas
  bool s0 = (code == 1) || (code == 3) ;
- bool s1 = code in { 2, 3 } ; # Notation ensembliste
  bool s0 = code in { 1, 3 } ;

Syntaxe HCL (5)

- Fonctions logiques retournant des mots :
  - int out = [# Multiplexeur
    s : A ; # La valeur de sortie est la première
    1 : B ; # valeur dont l'étiquette soit vraie
    ] ;





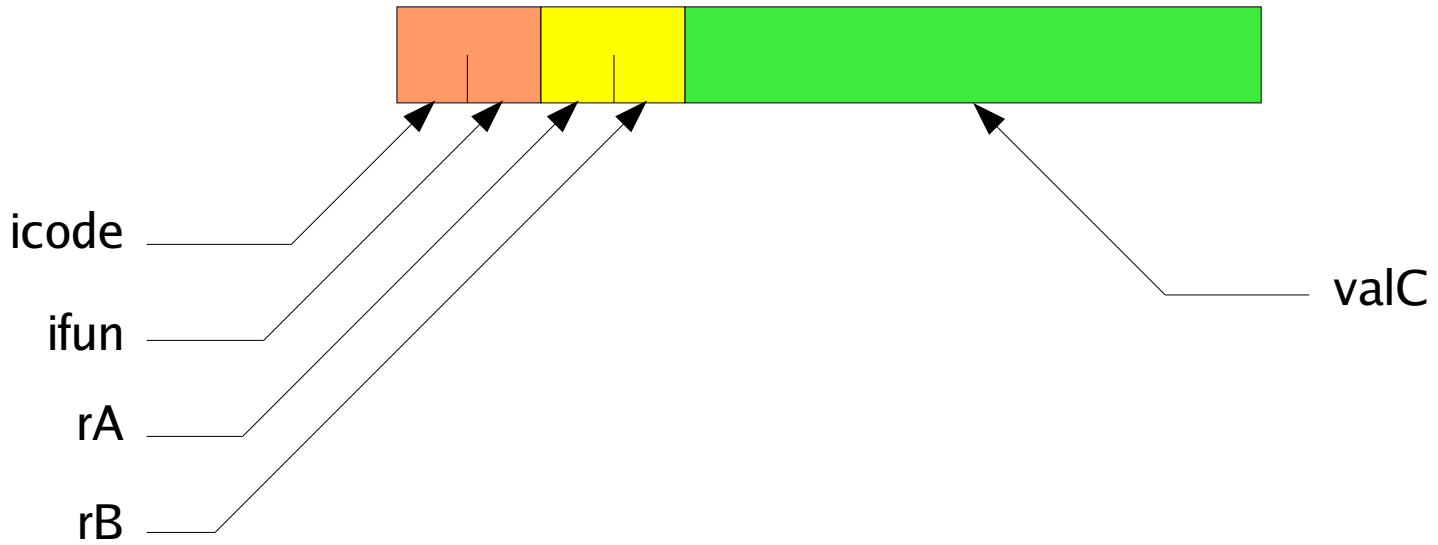
Syntaxe HCL (6)

```
int mult4 = [                  # Un multiplexeur multiplexe...
  !s1 && !s0 : A ;             # 00
  !s1 :        B ;             # 01
  !s0 :        C ;             # 10
  1 :          D ;             # 11
] ;
```

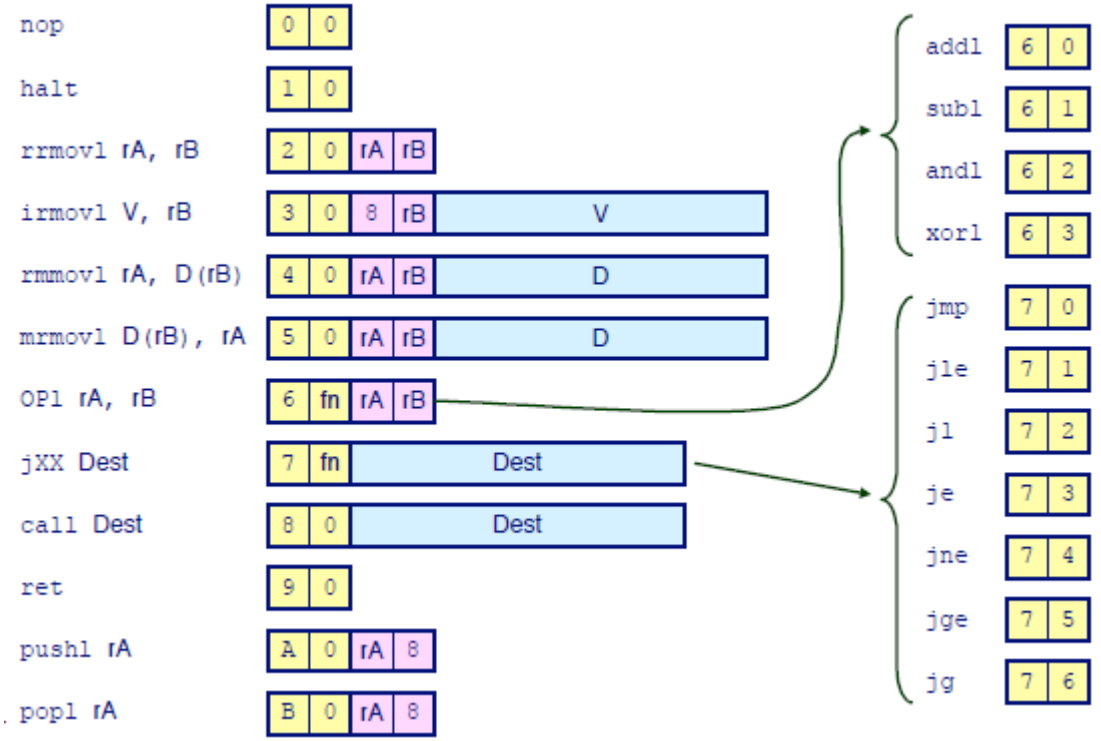
```
int min3 = [                   # Toute fonction peut s'exprimer
  A <= B && A <= C : A ;# sous la forme d'un multiplexeur
  B <= A && B <= C : B ;
  1 : C
] ;
```

Rappels sur l'ISA du y86 (1)

- Opcode : 1 octet
  - Sa valeur détermine la taille de l'instruction
- Registres : 1 octet optionnel
- Valeur immédiate : 4 octets optionnels

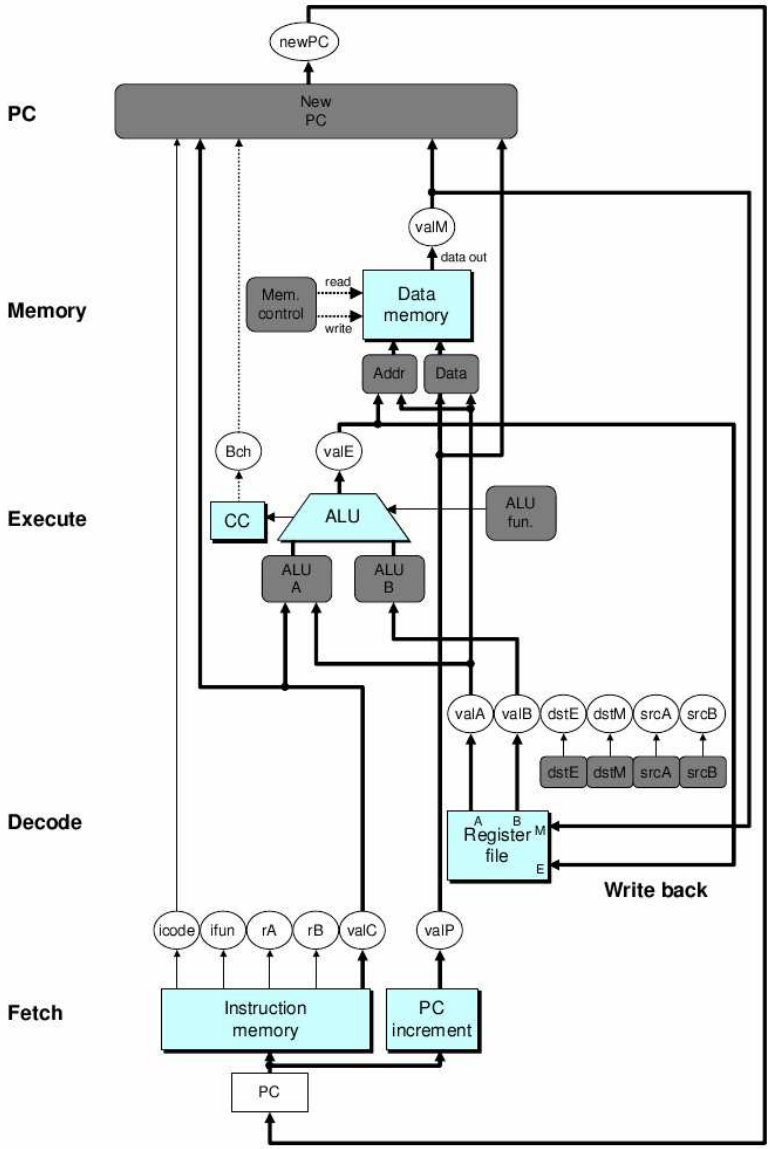


Rappels sur l'ISA du y86 (2)



Structure du y86-seq (1)

- L'exécution d'une instruction est organisée en six étapes principales :
  - « Fetch »
  - « Decode / Read »
  - « Execute »
  - « Memory »
  - « Write back »
  - « PC update »



Structure du y86-seq (2)

- Blocs bleu clair : unités matérielles non incluses dans le périmètre d'HCL
- Blocs gris : logique de contrôle
- Liaisons (pas toutes montrées sur le schéma) :
  - Épaisses : 1 mot
  - Fines : 1 octet ou quelques bits
  - Pointillées : 1 bit
- Ovales blancs : noms des liaisons

Fonctionnement du y86-seq (1)

- Exécution de « OPl » / « rrmovl » / « irmovl »

Étage	OPl rA,rB	rrmovl rA, rB	irmovl valC,rB
Fetch	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valP = PC + 2	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valP = PC + 2	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC + 6
Decode	valA = R[rA] valB = R[rB]	valA = R[rA]	
Execute	valE = valB OP valA CC = SetCC ()	ValE = 0 + valA	ValE = 0 + valC
Memory			
Write back	R[rB] = valE	R[rB] = valE	R[rB] = valE
PC update	PC = valP	PC = valP	PC = valP

Fonctionnement du y86-seq (2)

- Exécution de « rmmovl » / « mrmovl »

Étage	rmmovl rA,D(rB)	mrmovl D(rB),rA
Fetch	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC + 6	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC + 6
Decode	valA = R[rA] valB = R[rB]	valB = R[rB]
Execute	valE = valB + valC	valE = valB + valC
Memory	M_4[valE] = valA	ValM = M_4[valE]
Write back		R[rA] = valM
PC update	PC = valP	PC = valP

Fonctionnement du y86-seq (3)

- Exécution de « pushl » / « popl »

Étage	pushl rA	popl rA
Fetch	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valP = PC + 2	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valP = PC + 2
Decode	valA = R[rA] valB = R[%esp]	valA = R[%esp] valB = R[%esp]
Execute	valE = valB + (-4)	valE = valB + 4
Memory	M_4[valE] = valA	ValM = M_4[valA]
Write back	R[%esp] = valE	R[%esp] = valE R[rA] = valM
PC update	PC = valP	PC = valP

Fonctionnement du y86-seq (4)

- Exécution de « jXX » / « call » / « ret »

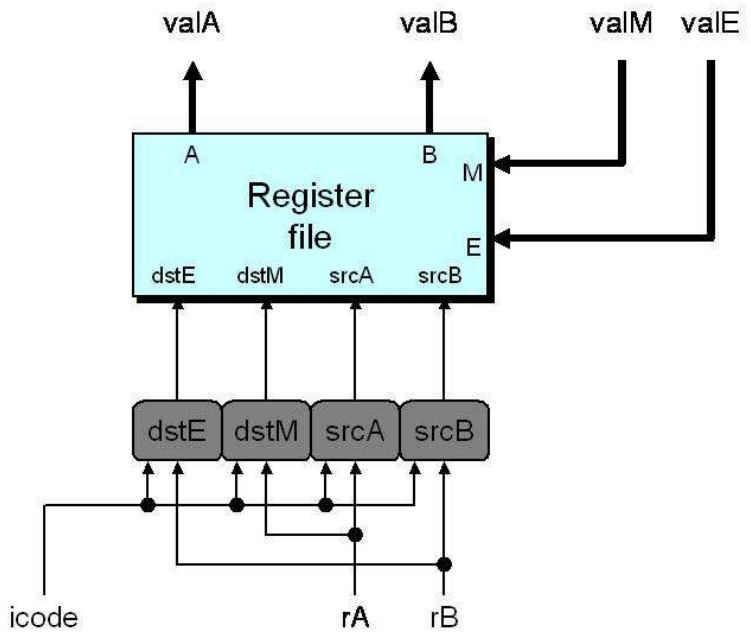
Étage	jXX valC	call valC	ret
Fetch	icode:ifun = M_1[PC] valC = M_4[PC+1] valP = PC + 5	icode:ifun = M_1[PC] valC = M_4[PC+1] valP = PC + 5	icode:ifun = M_1[PC] valP = PC + 1
Decode		valB = R[%esp]	valA = R[%esp] valB = R[%esp]
Execute	bch = Cond (CC, ifun)	valE = valB + (-4)	valE = valB + 4
Memory		M_4[valE] = valP	ValM = M_4[valA]
Write back		R[%esp] = valE	R[%esp] = valE
PC update	PC = bch ? ValC : valP	PC = valC	PC = valM

Câblage du y86-seq (1)

- « *Fetch* » : Récupération de l'instruction courante à exécuter
  - De 1 à 6 octets lus à partir de l'adresse contenue dans le compteur ordinal
  - Pré-calcul de l'adresse de l'instruction suivante

Câblage du y86-seq (2)

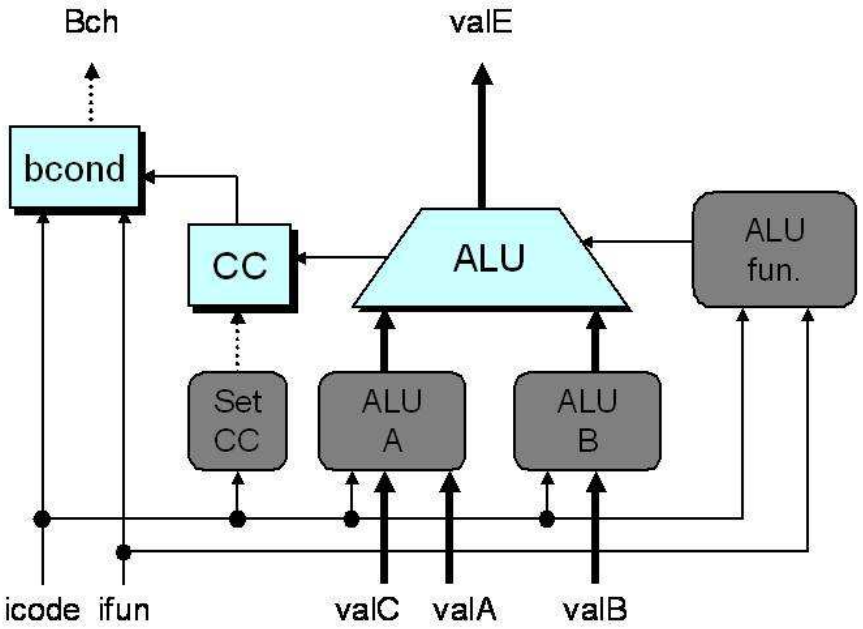
- « Decode »
  - Accès éventuel à la banque des registres




```
## What register should be used as the A source?
int srcA = [
  icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
  icode in { IPOPL, IRET } : RESP;
  1 : RNONE; # Don't need register ];
## What register should be used as the B source?
int srcB = [
  icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
];
## What register should be used as the E destination?
int dstE = [
  icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
];
## What register should be used as the M destination?
int dstM = [
  icode in { IMRMOVL, IPOPL } : rA;
  1 : RNONE; # Don't need register
];
```

Câblage du y86-seq (3)

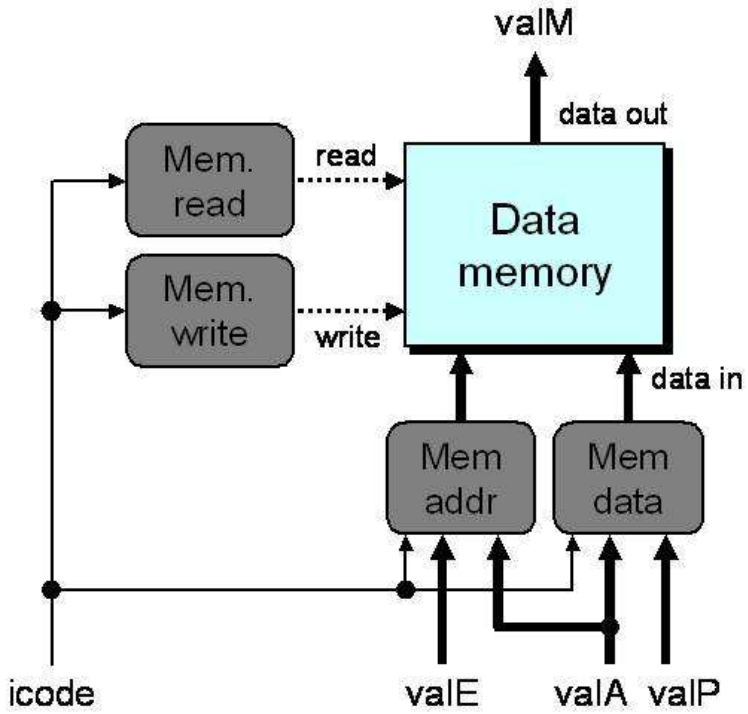
- « Execute » : Utilisation de l'UAL




```
## Select input A to ALU
int aluA = [
  icode in { IRRMOVL, IOPL } : valA;
  icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
  icode in { ICALL, IPUSHL } : -4;
  icode in { IRET, IPOPL } : 4;
# Other instructions don't need ALU
];
## Select input B to ALU
  int aluB = [
  icode in { IRMMOVL, IMRMOVL, IOPL,
                ICALL, IPUSHL, IRET, IPOPL } : valB;
  icode in { IRRMOVL, IIRMOVL } : 0;
# Other instructions don't need ALU
];
## Set the ALU function
int alufun = [
  icode == IOPL : ifun;
  1 : ALUADD;
];
## Should the condition codes be updated?
bool set_cc = icode in { IOPL };
```

Câblage du y86-seq (4)

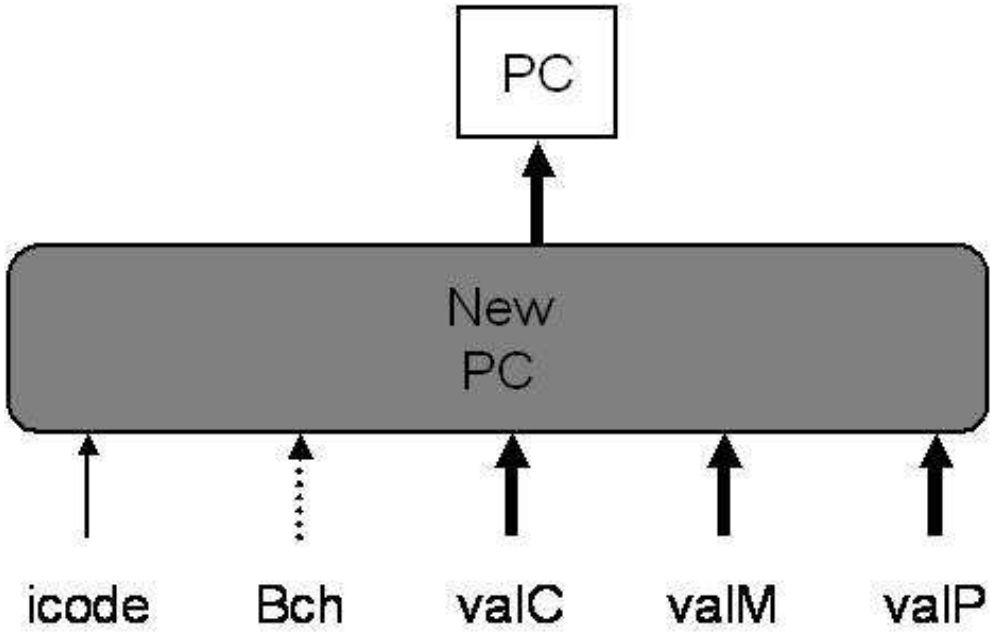
- «Memory» : Accès à la mémoire des données




```
## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL,
                                     ICALL };
## Select memory address
int mem_addr = [
  icode in { IRMMOVL, IPUSHL, ICALL,
                IMRMOVL } : valE;
  icode in { IPOPL, IRET } : valA;
# Other instructions don't need address
];
## Select memory input data
int mem_data = [ # Value from register
  icode in { IRMMOVL, IPUSHL } : valA;
# Return PC
  icode == ICALL : valP;
# Default: Don't write anything
];
```

Câblage du y86-seq (5)

- « PC update » : Mise à jour du compteur ordinal




```
int new_pc = [
# Call. Use instruction constant
  icode == ICALL : valC;
# Taken branch. Use instruction constant
  icode == IJXX && Bch : valC;
# Completion of RET instruction. Use value from stack
  icode == IRET : valM;
# Default: Use incremented PC
  1 : valP;
];
```

Analyse de l'architecture y86-seq (1)

- Caractéristiques :
  - Connecte des blocs fonctionnels prédéfinis au moyen de fonctions logiques combinatoires
  - Exprime chaque instruction sous la forme d'étapes simples
  - Flot de contrôle et de données unique pour chaque instruction

Analyse de l'architecture y86-seq (2)

- Limitations :
  - Au cours du cycle, le flot de contrôle doit se propager au sein de tous les circuits
    - Chemin critique très long
    - Fréquence d'horloge trop basse pour être compétitive (vintage '80s : 1 MHz)
  - Les unités fonctionnelles ne sont réellement utiles que pendant une fraction du cycle
- Ceci conduit naturellement à en dériver une version pipe-linée