université de BORDEAUX

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

(INF155)

F. Pellegrini
Université de Bordeaux

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer, de même que le présent texte.

université de BORDEAUX

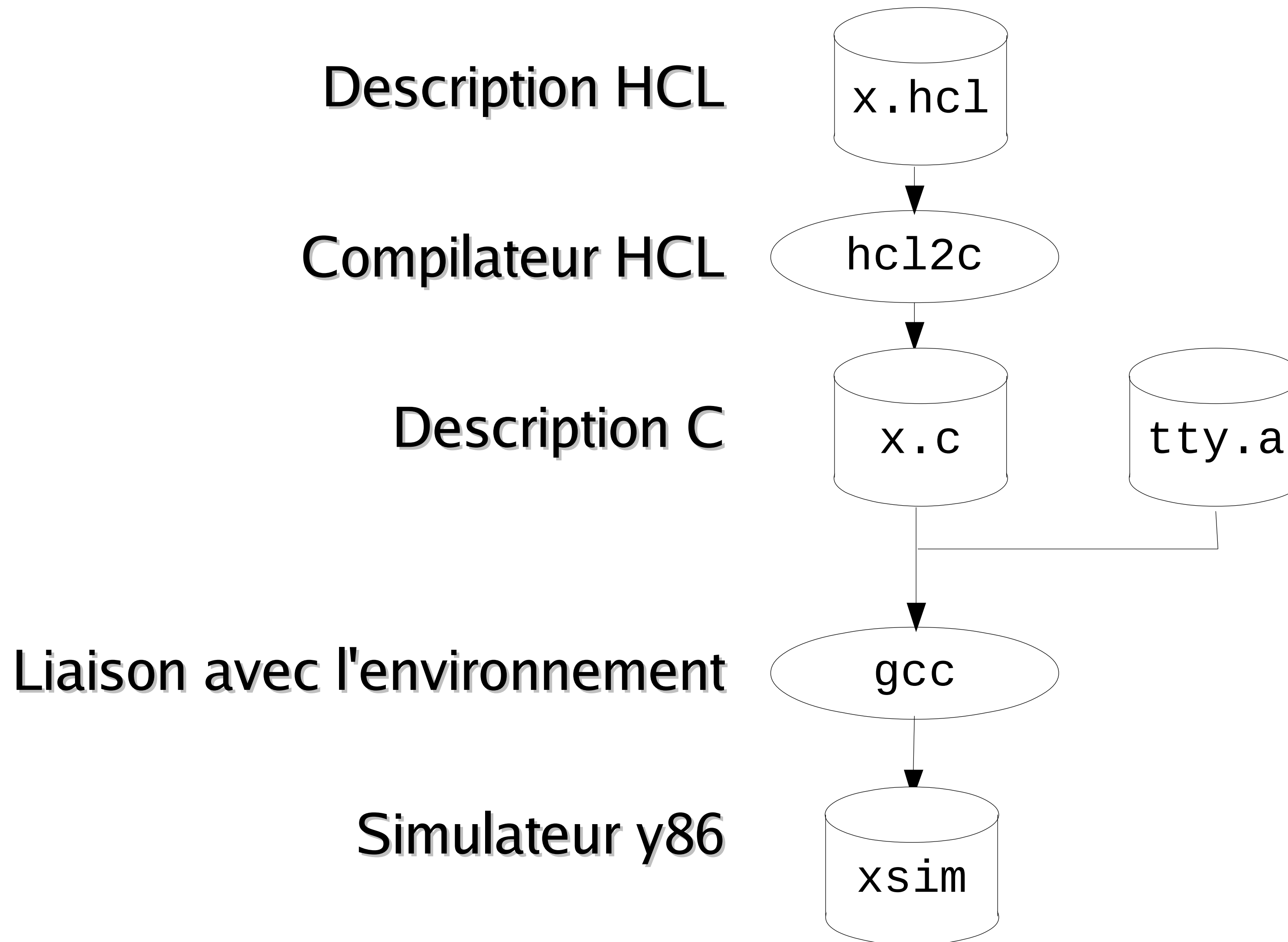
Modélisation du câblage du y86

- 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/

université de BORDEAUX

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, …

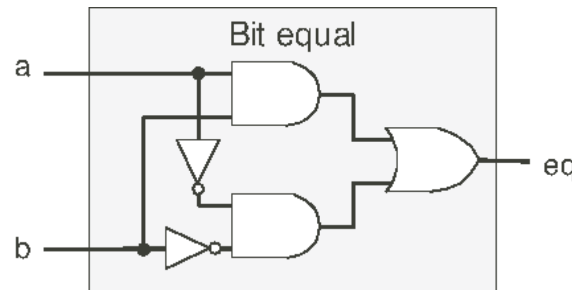
Syntaxe HCL (2)

- 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

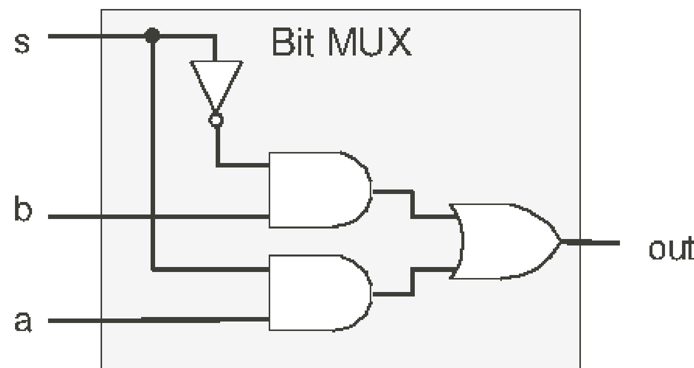
© 2014 F. Pellegrini

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

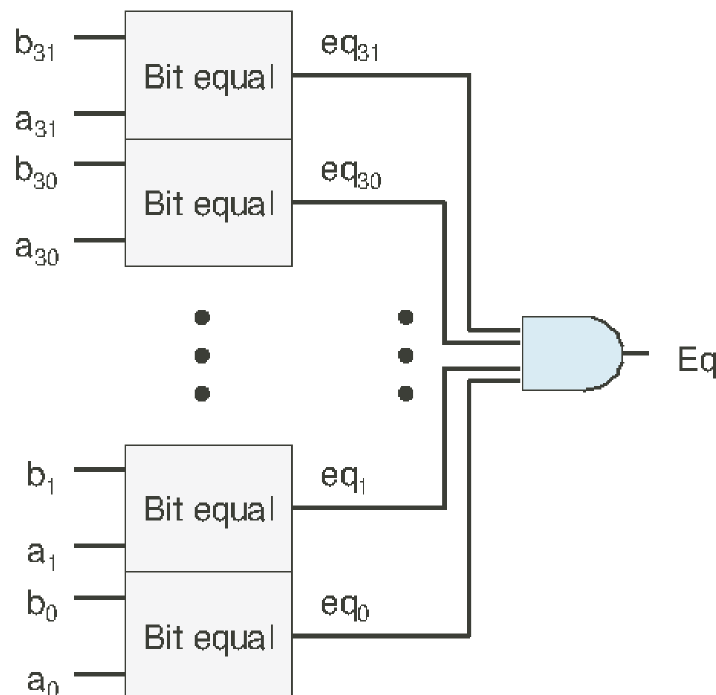


© 2014 F. Pellegrini

université de BORDEAUX

Syntaxe HCL (4)

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



© 2014 F. Pellegrini

Syntaxe HCL (5)

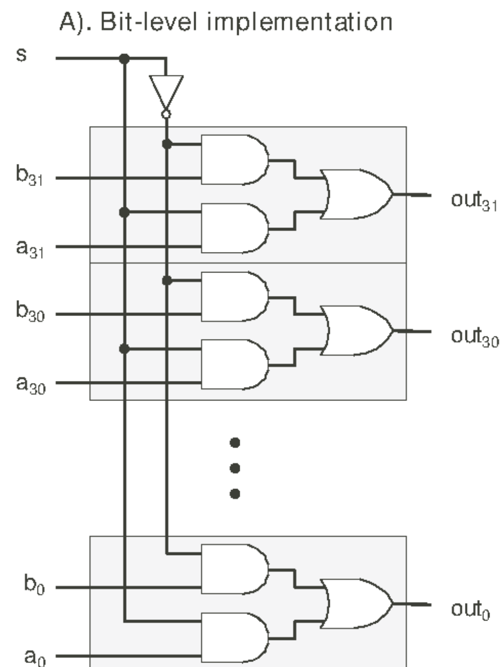
- bool s1 = (code == 2) || (code == 3) ; # Au cas par cas
  bool s0 = (code == 1) || (code == 3) ;
- bool s1 = code in { 2, 3 } ;   # Notation ensembliste
  bool s0 = code in { 1, 3 } ;

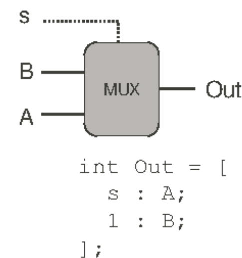
© 2014 F. Pellegrini

université de BORDEAUX

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
    ] ;





© 2014 F. Pellegrini

université de BORDEAUX

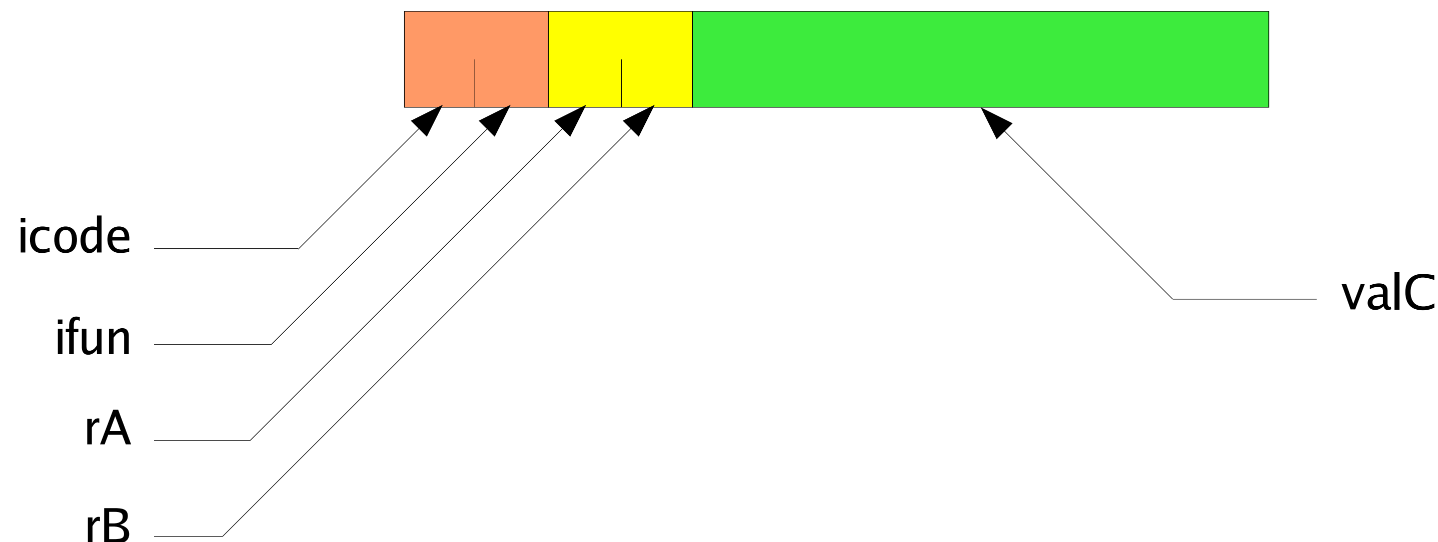
Syntaxe HCL (6)

```
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
];
```

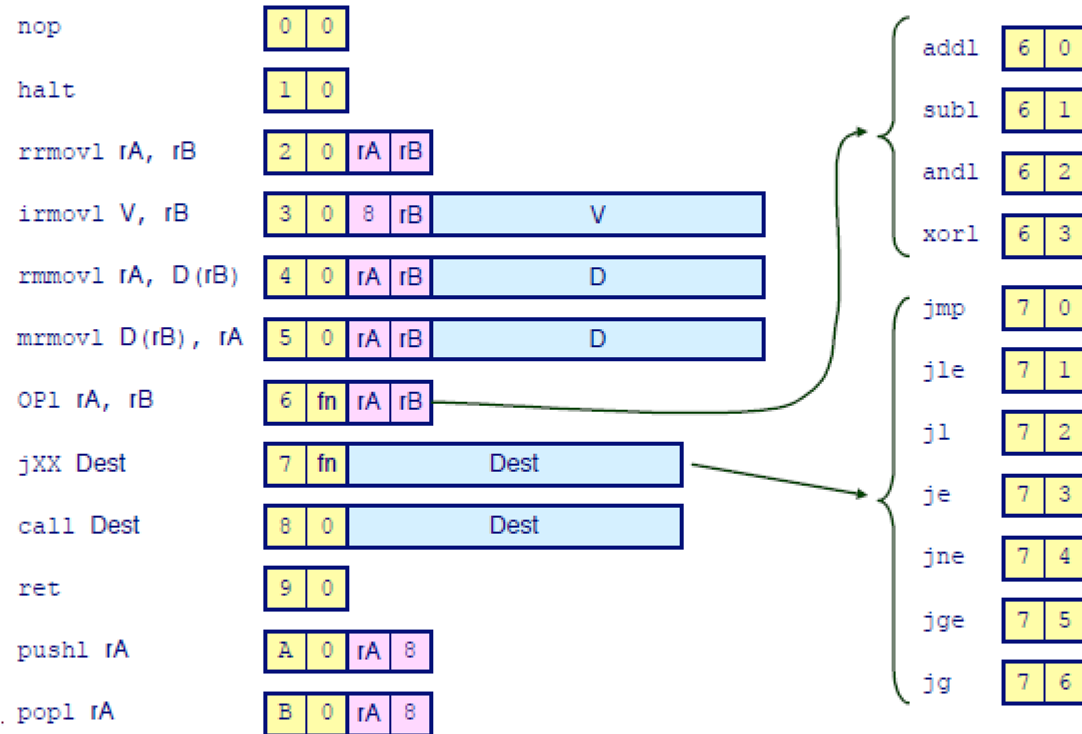
Rappels sur l'ISA du y86 (1)

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



© 2014 F. Pellegrini

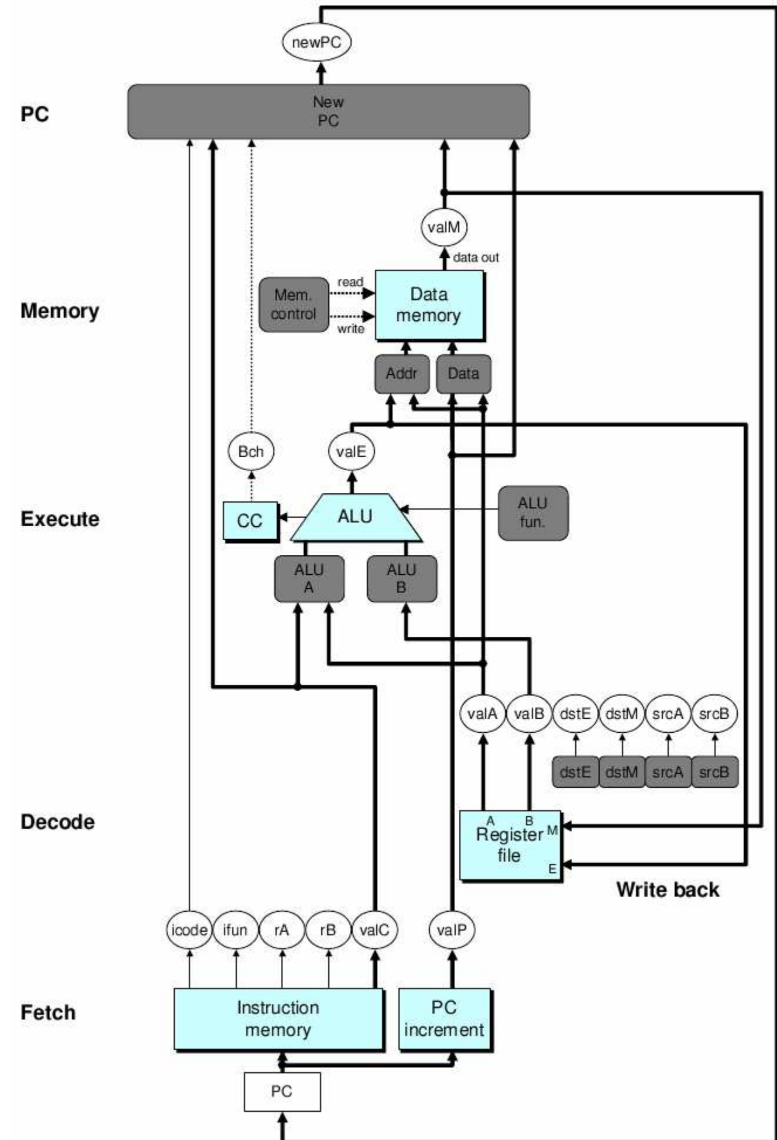
Rappels sur l'ISA du y86 (2)



université de BORDEAUX

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

© 2014 F. Pellegrini

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

© 2014 F. Pellegrini

université de BORDEAUX

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

© 2014 F. Pellegrini

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

© 2014 F. Pellegrini

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

© 2014 F. Pellegrini

Câblage du y86-seq (1)

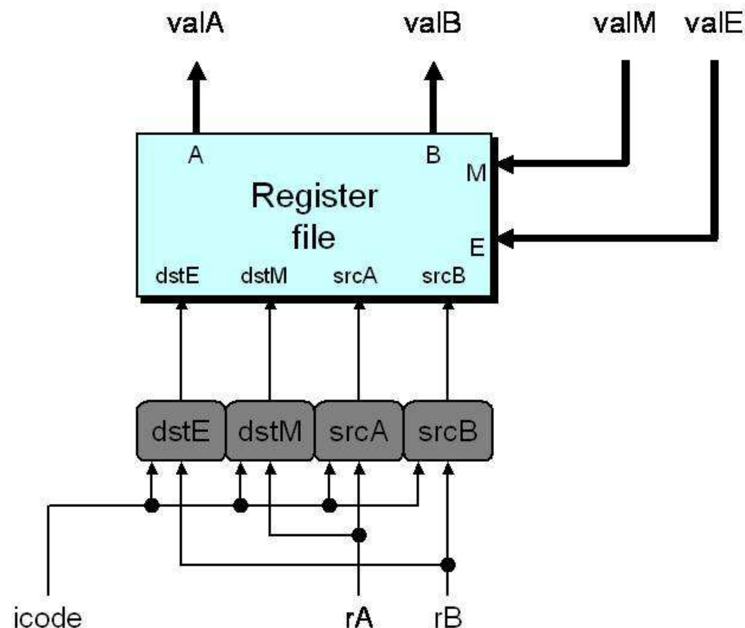
- « *Fetch* » : Récupération de l'instruction courante à exécuter
  - De 1 à 6 octets lus à partir de l'adresse contenue dans le compteur ordinal
  - Pré-calcul de l'adresse de l'instruction suivante

© 2014 F. Pellegrini

université de BORDEAUX

Câblage du y86-seq (2)

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

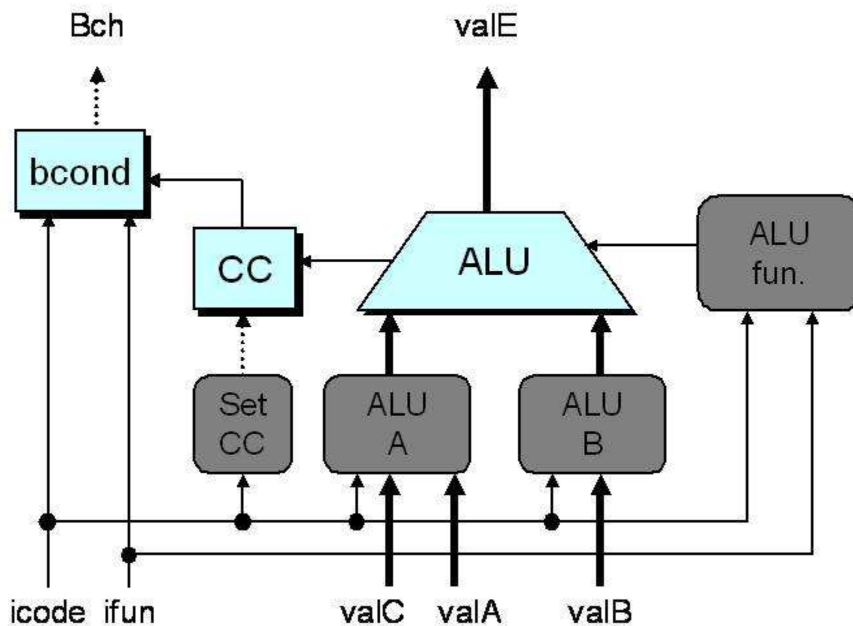



```
## What register should be used as the A source?
int srcA = [
  icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
  icode in { IPOPL, IRET } : RESP;
  1 : RNONE; # Don't need register ];
## What register should be used as the B source?
int srcB = [
  icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
  icode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

© 2014 F. Pellegrini

Câblage du y86-seq (3)

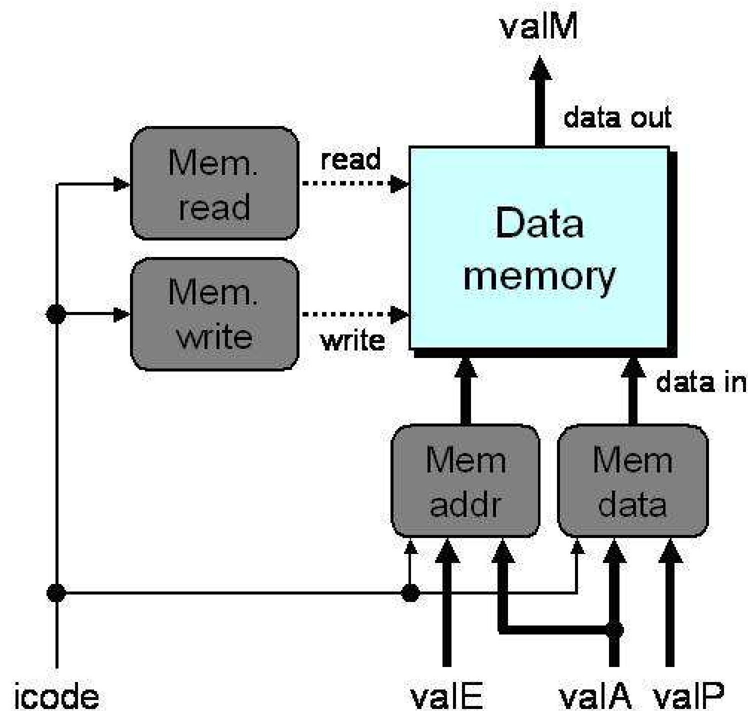
- « Execute » : Utilisation de l'UAL




```
## 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



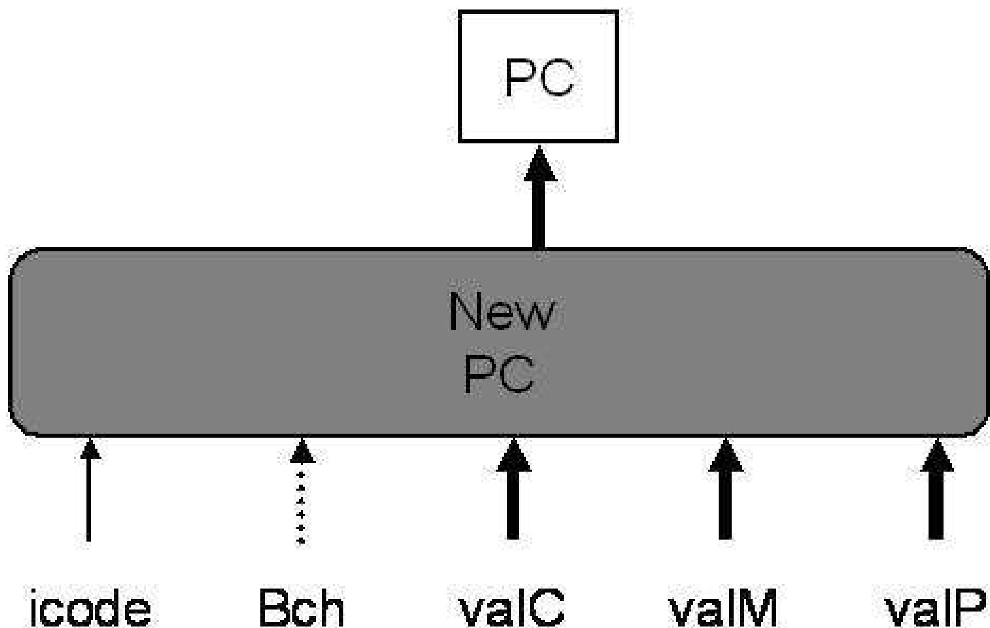

```
## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL,
                             ICALL };
## Select memory address
int mem_addr = [
  icode in { IRMMOVL, IPUSHL, ICALL,
             IMRMOVL } : valE;
  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
];
```

© 2014 F. Pellegrini

université de BORDEAUX

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;
];
```

© 2014 F. Pellegrini

université de BORDEAUX

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

© 2014 F. Pellegrini

université de BORDEAUX

Analyse de l'architecture y86-seq (2)

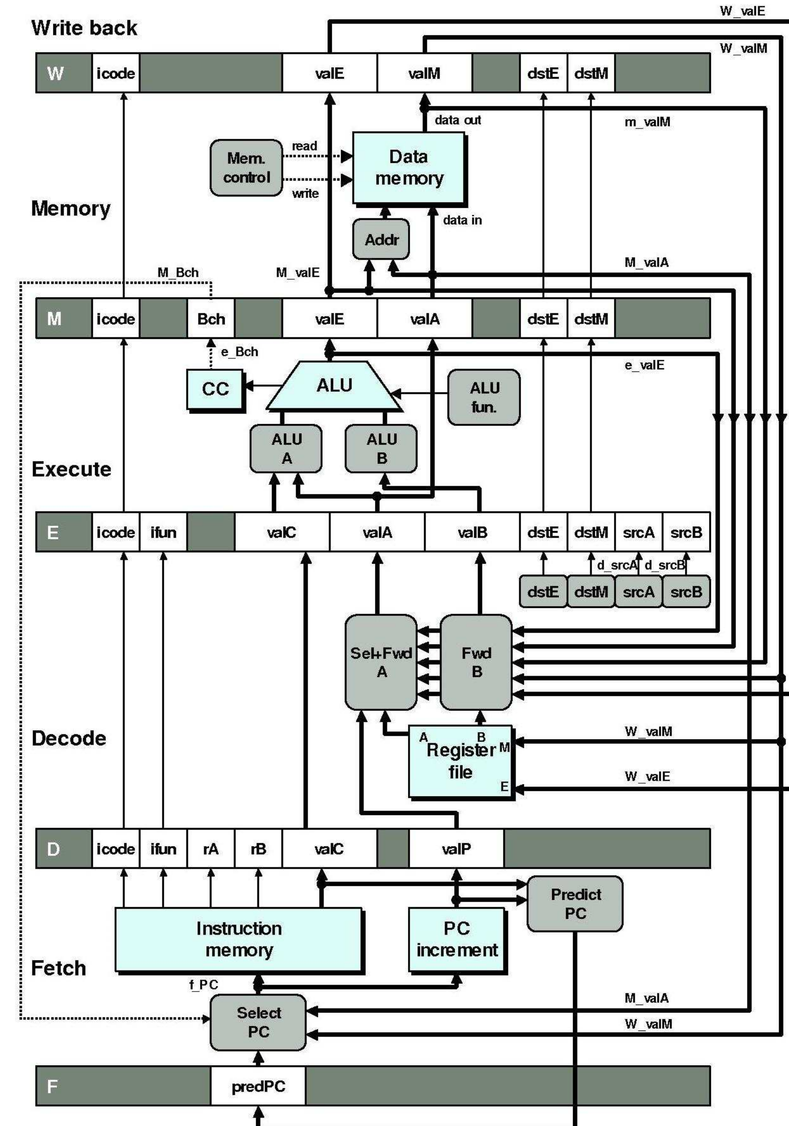
- Limitations :
  - Au cours du cycle, le flot de contrôle doit se propager au sein de tous les circuits
    - Chemin critique très long
    - Fréquence d'horloge trop basse pour être compétitive (vintage '80s : 1 MHz)
  - Les unités fonctionnelles ne sont réellement utiles que pendant une fraction du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

© 2014 F. Pellegrini

université de BORDEAUX

Structure de l'architecture y86-pipe

- Mise en œuvre d'un pipe-line à cinq étages
  - Fonctionnement dérivé de de l'architecture séquentielle
  - Réorganisation de certains blocs fonctionnels



© 2014 F. Pellegrini

Latches

- Chaque étage est isolé du précédent par un latch qui mémorise ce que l'étage doit effectuer lors du cycle courant
  - Nécessaire à l'exécution pipe-linée
- Informations disponibles à chaque étage :
  - Type d'instruction
  - Numéro des registres en cours de traitement
  - Indicateur de présence d'une bulle

© 2014 F. Pellegrini

Prédiction du compteur ordinal (1)

- Déplacement de la sélection du compteur ordinal en amont de l'étage *fetch*
  - Permet de prédire l'adresse de l'instruction suivante
    - Nécessaire pour pouvoir injecter une nouvelle instruction à chaque cycle élémentaire (« *instruction issue* »)
  - Rééquilibre le coût des étages du processeur

© 2014 F. Pellegrini

Prédiction du compteur ordinal (2)

- Prédiction de la prochaine adresse :
  - Pour la majorité des instructions, ce sera l'adresse incrémentée calculée à l'étage *fetch*
  - JMP : adresse de la valeur immédiate
- Instructions pour lesquelles on ne peut connaître l'adresse de la prochaine instruction :
  - Jxx : adresse connue après l'étage E
    - On a le choix entre deux adresses
  - RET : adresse connue après l'étage M
    - On ne peut absolument rien prédire

© 2014 F. Pellegrini

Prédiction du compteur ordinal (3)

- Lorsque l'adresse de la prochaine instruction ne peut être prédite, et tant que le résultat ne sera pas connu :
  - On bloque l'exécution de l'instruction suivante au niveau de l'étage *fetch* (« *stall* »)
    - D'où la présence d'un *latch* avant cet étage, contenant le compteur ordinal
  - On injecte une bulle dans l'étage suivant du pipe-line

© 2014 F. Pellegrini

Prédiction du compteur ordinal (4)

- Dans le cas des branchements conditionnels, on peut décider d'exécuter de façon prospective l'une des deux branches
  - On considérera toujours que les branchements conditionnels sont pris
  - S'il s'avère à l'étage E que ce n'est pas le cas, il faudra remettre le pipe-line sur la bonne voie

© 2014 F. Pellegrini

Réduction du nombre de signaux

- On peut réduire le nombre de latches en multiplexant au plus tôt les signaux mutuellement exclusifs
- Le bloc SelA sert à multiplexer les signaux valA et valP dès le moment où ils sont exclusifs :
  - Seul Jxx a besoin de valP à l'étage E
  - Seul CALL a besoin de valP à l'étage M
  - Aucune des deux instructions n'a besoin de valA
  - On peut multiplexer valA et valP dès l'étage D

© 2014 F. Pellegrini

université de BORDEAUX

Data forwarding

- Les entrées de l'étage d'exécution sont multiplexées avec les valeurs résultant des étages suivants
- Le muliplexage doit donner la priorité aux instructions les plus récentes, dont les valeurs peuvent écraser les plus anciennes

```
int E_valA = [
  D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
  d_srcA == E_dstE : e_valE;           # Forward valE from execute
  d_srcA == M_dstM : m_valM;           # Forward valM from memory
  d_srcA == M_dstE : M_valE;           # Forward valE from memory
  d_srcA == W_dstM : W_valM;           # Forward valM from write back
  d_srcA == W_dstE : W_valE;           # Forward valE from write back
  1 : d_rvalA;                         # Use value read from register file
];
```

```
irmovl 3,%eax
irmovl 4,%eax
addl   %eax,%ebx
```

© 2014 F. Pellegrini

Contrôle du pipe-line (1)

- Mise en œuvre de la gestion des dépendances en calculant pour chaque étage :
  - Si les données doivent y être bloquées (« *stall* »)
    - Désactivation du *latch* amont pour que les données de cet étage restent les mêmes à l'étage suivant
  - Si une bulle doit être générée à cet étage
    - Apparaîtra comme transmise à l'étage suivant
    - Positionnement des signaux du latch suivant aux valeurs correspondant à un nop (icode = INOP, srcA = srcB = … = RNONE)

© 2014 F. Pellegrini

Contrôle du pipe-line (2)

- Trois configurations nécessitent l'insertion de bulles et / ou le *stalling* de certains étages :
  - Instruction ret dans le pipe-line :
    - L'étage fetch doit rester bloqué tant que l'instruction ret n'a pas atteint l'étage M, afin que l'on connaisse l'adresse de la prochaine instruction à exécuter
  - « *Load / use hazard* » :
    - L'instruction utilisant le registre modifié par la lecture mémoire doit rester dans l'étage D tant que l'instruction mrmovl n'a pas atteint l'étage M
      - Utilisation du *data forwarding* pour la placer à l'étage E
      - Une seule bulle insérée à l'étage E, F et D en *stall*

© 2014 F. Pellegrini

université de BORDEAUX

Contrôle du pipe-line (3)

- Branchement conditionnel mal prédit
  - Une fois que l'instruction Jxx a atteint l'étage E, on sait si la prédiction effectuée a été la bonne, c'est-à-dire si les instructions des étages F et D sont les bonnes
  - Si la prédiction est mauvaise :
    - Les latches des étages D et E verront leurs instructions transformées à la volée en nop lors du cycle suivant
    - L'instruction chargée à l'étage F sera la bonne par *forwarding*
- Il faut faire attention aux cas particuliers mettant en jeu plusieurs de ces conditions
  - Cas d'un branchement conditionnel mal prédit ayant un ret comme instruction suivante

© 2014 F. Pellegrini

université de BORDEAUX

Contrôle du pipe-line (4)

```
################ Pipeline Register Control ##########################
# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
	# Conditions for a load/use hazard
	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } ||
	# Stalling at fetch while ret passes through pipeline
	IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
	# Conditions for a load/use hazard
	E_icode in { IMRMOVL, IPOPL } &&
	 E_dstM in { d_srcA, d_srcB };
bool D_bubble =
	# Mispredicted branch
	(E_icode == IJXX && !e_Bch) ||
	# Stalling at fetch while ret passes through pipeline
	# but not condition for a load/use hazard
	!D_stall &&  IRET in { D_icode, E_icode, M_icode };
```

© 2014 F. Pellegrini

Contrôle du pipe-line (5)

```
# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
          E_dstM in { d_srcA, d_srcB};

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
bool M_bubble = 0;
#/* $end pipe-all-hcl */
```

© 2014 F. Pellegrini

université de BORDEAUX

Contrôle du pipe-line (6)