UNIVERSITÉ BORDEAUX 1 Sciences Technologies

Année 2010-2011
Étape : Licence Sciences Technologies (semestre 4)
UE : INF155 (Architecture des ordinateurs)
Modalités : Épreuve sans document

Date : 5 mai 2011
Heure : 14h00
Durée : 3h00

1 Architecture en pipeline (questions de cours)

Q1 Expliquez comment, en comparaison avec l'implémentation naïve de la version pipelinée du processeur Y86, il est parfois possible d'éviter la génération de bulles à l'étage *Execute*.

Q2 À titre d'illustration, donnez une suite de deux instructions provoquant l'apparition de bulles entre les deux instructions dans la version naïve, mais n'engendrant aucune bulle dans la version optimisée. Expliquez, sur votre exemple, pourquoi la seconde instruction peut « suivre » la première sans délai à tous les étages.

Q3 Donnez une suite de deux instructions provoquant l'apparition d'une bulle même dans la version optimisée. Expliquez précisément pourquoi ce n'est pas évitable dans la version pipelinée du Y86.

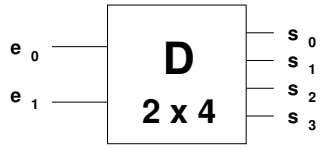
2 Circuits combinatoires

On considère un « décodeur » à deux entrées et quatre sortie (voir ci-contre).

Q1 Donnez la table de vérité de ce ciruit, et proposez-en une implémentation à l'aide de portes logiques simples.

Q2 Construisez un tel décodeur 2x4 uniquement en utilisant un démultiplexeur.

Q3 On souhaite utiliser un tel décodeur 2x4 pour construire un décodeur à trois entrées et huit sorties. En utilisant un ou plusieurs circuits supplémentaires (idée : un démultiplexeur devrait faire l'affaire), proposez une implémentation simple du décodeur 3x8.

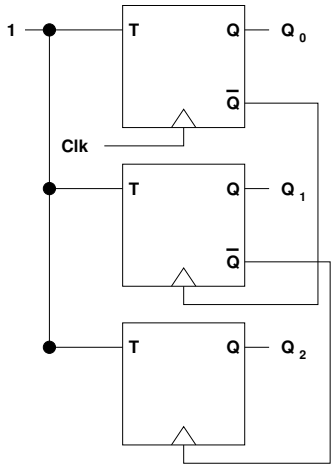


3 Circuits séquentiels

Le circuit ci-contre est constitué de trois bistables T. Dans un bistable T, **lors du front montant d'horloge**, la sortie Q change d'état si l'entrée T (*Toggle*) est à 1. Si l'entrée T est à 0, le bistable ne change pas d'état.

Q1 On suppose qu'initialement, les trois bistables ont pour état 0 ($Q_0 = Q_1 = Q_2 = 0$). Tracez un chronogramme, sur huit périodes d'horloge, sur lequel vous indiquerez soigneusement l'évolution des valeurs de Q_0, Q_1 et Q_2. Considérant que $Q_2Q_1Q_0$ est un nombre sur trois bits, expliquez ce que fait ce circuit.

Q2 Construisez un bistable T à partir d'un bistable D.



4 Version pipelinée du processeur Y86

Q1 Examinez le programme ci-contre. Que fait-il ? Que vaut la variable `res` une fois l'exécution terminée ? Prenez soin à bien détailler votre réponse !

Q2 Montrez, à l'aide d'un joli chronogramme, la progression des instructions dans les différents étages du pipeline (à partir de l'instruction suivant l'étiquette `loop` jusqu'à l'instruction `jne loop` incluse) pendant **les deux premiers tours de boucles seulement**. On suppose que le comportement du pipe-line est donné par le fichier HCL joint. Attention aux prédictions de branchement...

Q3 On se propose d'introduire une nouvelle instruction dans le processeur Y86 : `CMPL ra, rb` (*Compare*). Cette instruction (qui prend deux registres en arguments) effectue l'équivalent de `subl ra, rb`, excepté qu'elle ne modifie pas la valeur de rb. Le seul effet de cette instruction est donc de modifier les indicateurs des codes de conditions (notamment *Zero Flag* et *Sign Flag*).
Expliquez comment il est possible de simplifier légèrement le programme ci-contre en utilisant cette nouvelle instruction `CMPL`.

Q4 La feuille jointe au sujet reproduit l'essentiel du fichier pipe-std.hcl, qui décrit la version pipelinée du processeur Y86. Ecrivez directement sur cette feuille les ajouts que vous proposez pour traiter cette nouvelle instruction (de code `CMPL`).
NB : n'oubliez pas d'inscrire votre numéro d'anonymat sur la feuille avant de l'insérer dans votre copie.

Q5 Écrivez une fonction `cmp` à deux arguments réalisant l'équivalent de l'instruction `CMP` (en supposant que l'instruction `CMP` n'est pas disponible). La fonction ne retourne pas de valeur particulière dans `%eax`, elle doit juste positionner les codes de condition. On rappelle que les registres *callee-save* du Y86 sont `%ebx, %esi et %edi`.

```
      .pos 0
      irmovl 500, %esp
      mrmovl size, %ecx
      irmovl 0, %edi
      irmovl 0, %ebx
loop:
      mrmovl t(%edi), %eax
      pushl %ebx
      subl %eax, %ebx
      popl %ebx
      jge skip
      rrmovl %eax, %ebx
skip:
      iaddl 4, %edi
      isubl 1, %ecx
      jne loop
      rmmovl %ebx, res
      halt

      .align 4
      .pos 100
t:
      .long 12
      .long 5
      .long 20
      .long 8
size: .long 4
res:  .long 0
```

On s'intéresse maintenant au fichier HCL fournit sur la feuille suivante.

Q6 Concernant l'étage *Fetch*, on remarque le test : `M_icode == JXX`. Pourquoi teste-t-on le contenu du registe M, alors que sur la ligne suivante on teste le registre W ?

Q7 Concernant l'étage *Decode*, dans le calcul de `new_E_valA`, on remarque qu'il n'y a pas de test implicant la valeur `E_dstM`. Pourquoi ? Cela signifie-t-il que le fait que `srcA` soit égal `E_dstM` n'est pas gênant ? Ou, si problème il y a, il est traité ailleurs ?

— Feuille à détacher et à rendre avec la copie —

Numéro d'anonymat :

```
################ Fetch Stage     ###################################

## What address should instruction be fetched at
int f_pc = [
# Mispredicted branch.  Fetch at incremented PC
M_icode == JXX && !M_Bch : M_valA;
# Completion of RET instruction.
W_icode == RET : W_valM;
# Default: Use predicted value of PC
1 : F_predPC;
];

# Predict next value of PC
int new_F_predPC = [
f_icode in { JXX, CALL } : f_valC;
1 : f_valP;
];

################ Decode Stage ######################################

## What register should be used as the A source?
int new_E_srcA = [
D_icode in { RRMOVL, RMMOVL, OPL, PUSHL  } : D_rA;
D_icode in { POPL, RET } : RESP;
1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
D_icode in { OPL, RMMOVL, MRMOVL  } : D_rB;
D_icode in { PUSHL, POPL, CALL, RET } : RESP;
1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
D_icode in { RRMOVL, IRMOVL, OPL} : D_rB;
D_icode in { PUSHL, POPL, CALL, RET } : RESP;
1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int new_E_dstM = [
D_icode in { MRMOVL, POPL } : D_rA;
1 : RNONE;  # Don't need register
];

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
D_icode in { CALL, JXX } : 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
];

int new_E_valB = [
d_srcB == E_dstE : e_valE;    # Forward valE from execute
d_srcB == M_dstM : m_valM;    # Forward valM from memory
d_srcB == M_dstE : M_valE;    # Forward valE from memory
d_srcB == W_dstM : W_valM;    # Forward valM from write back
d_srcB == W_dstE : W_valE;    # Forward valE from write back
1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
int aluA = [
E_icode in { RRMOVL, OPL } : E_valA;
E_icode in { IRMOVL, RMMOVL, MRMOVL } : E_valC;
E_icode in { CALL, PUSHL } : -4;
E_icode in { RET, POPL } : 4;
# Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
E_icode in { RMMOVL, MRMOVL, OPL, CALL,
      PUSHL, RET, POPL } : E_valB;
E_icode in { RRMOVL, IRMOVL } : 0;
# Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
E_icode == OPL : E_ifun;
1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == OPL;

################ Memory Stage ######################################

## Select memory address
int mem_addr = [
M_icode in { RMMOVL, PUSHL, CALL, MRMOVL } : M_valE;
M_icode in { POPL, RET } : M_valA;
# Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { MRMOVL, POPL, RET };

## Set write control signal
bool mem_write = M_icode in { RMMOVL, PUSHL, CALL };

################ 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
        !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };

# 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};
```