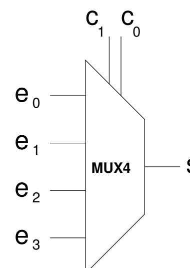
UNIVERSITÉ BORDEAUX 1
Sciences Technologies

Année 2008-2009
Étape : Licence Sciences Technologies (semestre 4)
UE : INF155 (Architecture des ordinateurs)
Modalités : Épreuve sans document

Date : 15 mai 2009
Heure : 14h00
Durée : 3h00

1 Circuits combinatoires

On souhaite construire un multiplexeur « MUX4 » à 4 entrées (utilisant donc 2 bits de commande) comme illustré sur la figure ci-contre.



Q1 (question d'échauffement) Donnez une implémentation de ce circuit à l'aide de portes logiques simples.

Q2 On dispose désormais de multiplexeurs « MUX2 » à 2 entrées (à un 1 bit de commande). Montrez que l'on peut construire MUX4 avec 3 exemplaires de MUX2 (idée : filtrer en deux étapes). Prenez soin de bien numéroter les fils sur votre schéma.

2 Branchement à adresse calculée

Le jeu d'instruction Y86 ne possède pas d'instruction permettant de faire un saut inconditionnel à une adresse contenue dans un registre. C'est pourtant fort utile. On se propose donc d'écrire une fonction `jmp_eax` qui effectue un saut à l'adresse contenue dans le registre `%eax` au moment de l'appel. Dans le code ci-contre, les deux premières lignes sont ainsi équivalentes à l'instruction `jmp label`.

```
   irmovl $label, %eax
   call jmp_eax
   ...
label:
   ...
```

Q1 Pour réaliser la fonction `jmp_eax`, l'idée est d'utiliser l'instruction `ret` après avoir modifié l'adresse de retour... Donnez le code de la fonction `jmp_eax`.

Q2 Expliquez calmement et soigneusement le fonctionnement du programme ci-contre. À quoi servent les deux lignes `addl %ebx, %ebx` ? Quel bloc d'instructions (0, 1 ou 2) sera exécuté dans cet exemple ?

```
  mrmovl x, %ebx
  addl %ebx, %ebx
  addl %ebx, %ebx
  mrmovl array(%ebx), %eax
  call jmp_eax
case0:
  ... ; bloc 0
  jmp fin
case1:
  ... ; bloc 1
  jmp fin
case2:
  ... ; bloc 2
  jmp fin
fin:
  halt
  .align 4
array:
  .long case0
  .long case1
  .long case2
x:
  .long 1
```

Q3 Imaginez le code C dont la compilation produirait un tel code assembleur. Quel est intérêt d'utiliser une telle structure de code assembleur, plutôt qu'un code correspondant à un enchaînement de `if ... else if ... else if ...` ?

Q4 On souhaite ajouter dans le jeu d'instructions Y86 une instruction «*Jump Register*» (`jr %reg`), effectuant un saut inconditionnel à l'adresse contenue dans le registre opérande. Cela permettra typiquement d'utiliser `jr %eax` au lieu de `call jmp_eax`. En considérant la version pipelinée du processeur, expliquez le problème posé par cette instruction concernant la prédiction de branchement. À quel étage du pipeline connaîtra-t-on avec certitude l'adresse de l'instruction suivante ? Déduisez-en le nombre de cycles «perdus» dans le pipeline.

Q5 Répondre aux mêmes questions que dans **Q4** dans le cas d'une instruction «*Jump Memory*» (`jm depl(%reg)`) permettant de faire un saut à une adresse chargée depuis la mémoire (e.g. `jm 8(%ebp)`).

Q6 Les deux séquences de code ci-après sont donc fonctionnellement équivalentes. Qu'en est-il de leur performance ? Appuyez-vous sur des chronogrammes montrant la progression des instructions dans le pipeline Y86 pour chacune de deux séquences. On notera simplement `OP` l'instruction se trouvant à l'adresse destinataire du branchement. Attention aux dépendances !

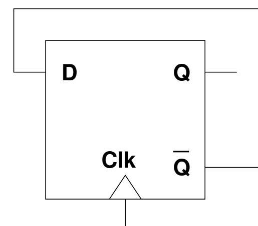
```
mrmovl array(%ebx), %eax
jr     %eax
```

```
jm     array(%ebx)
```

Q7 (indépendante de Q5 et Q6, sauf pour la définition de *Jump Memory*) La feuille jointe au sujet reproduit l'essentiel du fichier seq-std.hcl, qui décrit la version séquentielle simple du processeur Y86. On suppose qu'on dispose du symbole supplémentaire IJM pour désigner l'instruction "Jump Memory" : écrivez directement sur cette feuille les ajouts que vous proposez pour traiter cette instruction.
NB : n'oubliez pas d'inscrire votre numéro d'anonymat sur la feuille avant de l'insérer dans votre copie.

3 Circuits séquentiels et horloge

Le circuit ci-contre est un bistable D sur lequel on a relié la sortie «inversée» $\bar{Q}$ à l'entrée D. On suppose qu'initialement, la sortie Q vaut 0.



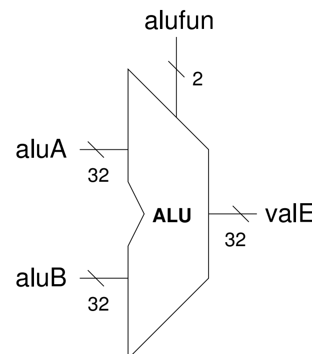
Q1 En utilisant un chronogramme s'étalant sur au moins 4 cycles d'horloge, tracez l'évolution de la sortie Q en regard de la valeur du signal d'horloge. Comparez les 2 signaux. Que constatez-vous ?

Q2 Déduisez-en une façon de réaliser un circuit «diviseur de fréquence» acceptant en entrée un signal d'horloge ainsi qu'un bit de commande (0 = normal, 1 = slow) et fournissant en sortie soit le signal d'horloge inchangé (mode *normal*) soit un signal d'horloge de période deux fois plus longue (mode *slow*).

Q3 Proposez un circuit composé de deux bistables D qui permette de diviser la fréquence d'horloge par quatre (idée : pensez au compteur de Johnson). Tracez le chronogramme s'étalant sur au moins 8 cycles d'horloge.

4 Super ALU

L'ALU du processeur Y86, représentée ci-contre, est capable d'effectuer une opération sélectionnée parmi quatre disponibles en fonction de la valeur `alufun` codée sur 2 bits : 00 = ADD, 01 = SUB, 10 = AND et 11 = XOR.
On suppose qu'on dispose également d'un circuit similaire, nommé MULDIV, capable d'effectuer une multiplication ou une division, suivant la valeur de son signal de commande binaire : 0 = MUL, 1 = DIV. Pour simplifier, on considère que le résultat de ce circuit sort également sur 32 bits.



Q1 Assemblez ces deux circuits en une seule «*Super ALU*» (en utilisant des portes et/ou des circuits supplémentaires) où la sélection de l'opération désirée s'effectuera au moyen d'un signal de commande à 3 bits. Choisissez un codage des opérations (ADD, SUB, AND, XOR, MUL, DIV) simplifiant la réalisation.

Q2 On décide donc de remplacer l'ancienne ALU par notre nouvelle *Super ALU* au sein de la version pipelinée de Y86. Si on suppose qu'une opération MUL ou DIV a besoin de 2 cycles pour se stabiliser, expliquer ce qu'il faut faire pour que l'exécution se déroule normalement lorsqu'une telle opération se trouve à l'étage *Execute*.

Q3 Finalement, ce n'était peut-être pas une si bonne idée d'assembler ALU et MULDIV. Si l'on avait simplement juxtaposé ces deux circuits dans le processeur, on aurait pu espérer effectuer certaines opérations en parallèle. Par exemple, une instruction `addl` chargée juste après une instruction `divl` pourrait entrer à l'étage *Execute* pendant le second cycle d'activité du circuit MULDIV... C'est précisément ce qui se passe dans un processeur superscalaire. Dans le cas de Y86, les choses ne sont malheureusement pas aussi simples. Par exemple, que se passerait-t-il pour les deux instructions `addl` et `divl` par la suite, lors de leur passage *simultané* aux étages suivants (on suppose que `divl` se comporte de manière similaire à `addl`) ? Comment pourrait-on résoudre ces problèmes ?

— Feuille à détacher et à rendre avec la copie —

Numéro d'anonymat :

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

# Does fetched instruction require a regid byte?
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

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

## 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
];

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

## 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 };

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

## 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
];

################ Program Counter Update ############################

## What address should instruction be fetched at

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;
];
```