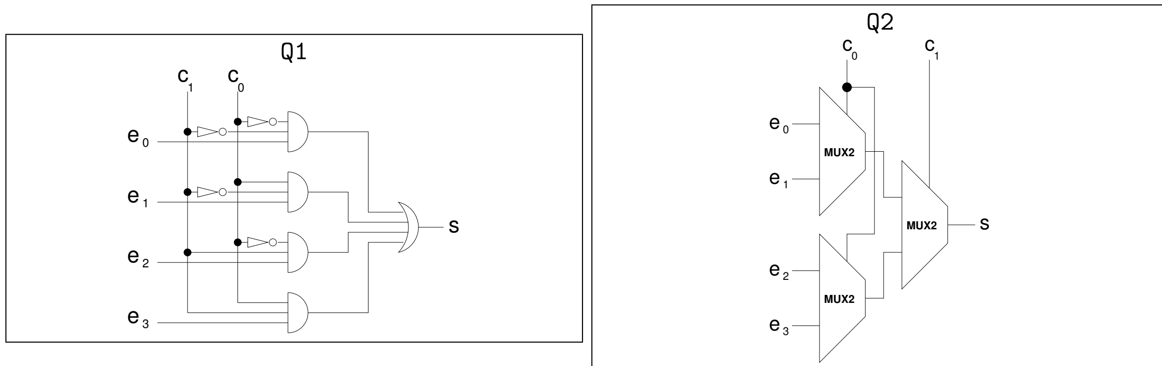
R. Namyst 6 mai 2009

Licence, mention informatique — Architecture des ordinateurs
Corrigé de l'examen du 15 mai 2009

1 Circuits combinatoires



2 Branchement à adresse calculée

Q1

```
jmp_eax:    rmmovl %eax, 0(%esp)
            ret
```

Q2 Le programme exécute l'un des trois blocs d'intruction *bloc0*, *bloc1* ou *bloc2*, en fonction de la valeur de la variable globale `x` qui est supposée appartenir à l'intervalle `[0..2]`. Voici comment il procède :

– L'adresse de chacun des blocs est rangée dans le tableau `array` (`array[0]` contient l'adresse `case0`, etc.)
– Pour récupérer l'adresse du bloc à exécuter, il faut donc multiplier la valeur de `x` par quatre (ce que font les deux `addl` successifs) de façon a pouvoir indicer correctement le tableau `array` (ce tableau contient des adresses codés sur 32 bits, donc sur quatre octets).
– Une fois cet indice placé dans le registre `%ebx`, l'adresse de destination est chargée dans `%eax` depuis l'emplacement mémoire `array + %ebx`.
– Enfin, la fonction `jmp_eax` est appelée pour effectuer un saut à l'adresse contenue dans `%eax`.

Dans le cas présent, la variable `x` est initialisée à 1, donc l'adresse destination est celle se trouvant en mémoire à l'adresse `array + 4`, c'est-à-dire `case1`. C'est donc le *bloc1* qui est exécuté.

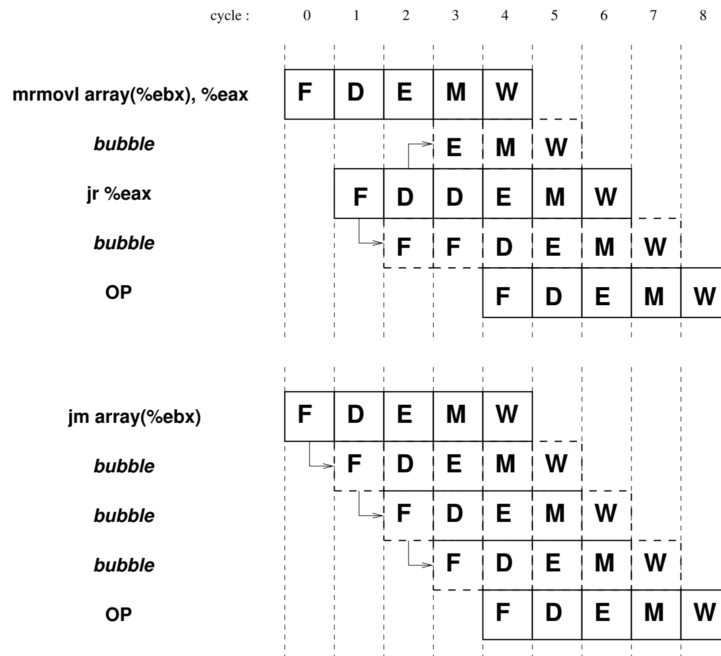
Q3 L'avantage pour un compilateur C d'utiliser un saut à adressage calculé dans le cas d'un *switch* est d'obtenir un code plus efficace en moyenne : même si la récupération de l'adresse est légèrement plus coûteuse qu'un simple test, elle est bien plus efficace que d'effectuer une succession de tests et de sauts.

```
switch(x)
{
case 0: ... // bloc0
        break;
case 1: ... // bloc1
        break;
case 2: ... // bloc2
        break;
}
```

Q4 L'instruction «*Jump Register*» (`jr %reg`) saute à une adresse contenue dans un registre. Il n'est donc pas possible (en considérant l'architecture Y86 vue en cours) de prédire quoi que ce soit lors du chargement de cette instruction. L'adresse effective sera connue juste avant la fin de l'étage *Decode*, lorsque la lecture du registre sera effectuée. À ce moment, il sera possible de charger sans se tromper l'instruction suivante. On n'aura donc perdu que 1 cycle.

Q5 Dans le cas de l'instruction «*Jump Memory*» (`jm depl(%reg)`), on rencontre le même problème de non-prédictibilité de l'adresse de branchement. Toutefois, le problème est plus grave ici car l'adresse effective ne sera connue qu'à la fin de l'étage *Memory*, c'est-à-dire que l'on va devoir gaspiller 3 cycles (correspondant aux cycles pendant lesquels l'instruction `jm` traverse les étages *Decode*, *Execute* et *Memory*) avant de pouvoir charger l'instruction suivante.

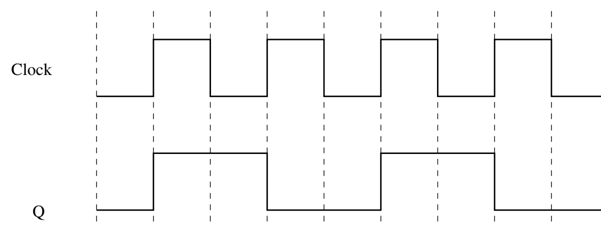
Q6 Voici une comparaison de la progression dans le pipeline Y86 des deux séquences de code. Comme on peut le constater, les performances sont équivalentes : dans les deux cas, l'instruction `OP` débute au cinquième cycle (i.e. cycle 4).



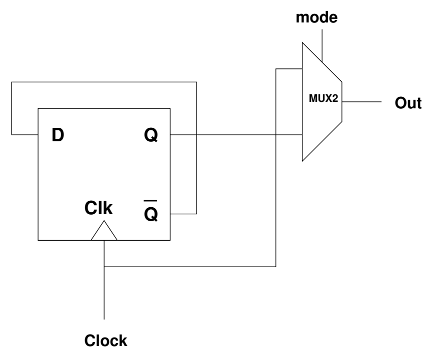
Q7 Voir correction en annexe de ce document.

3 Circuits séquentiels et horloge

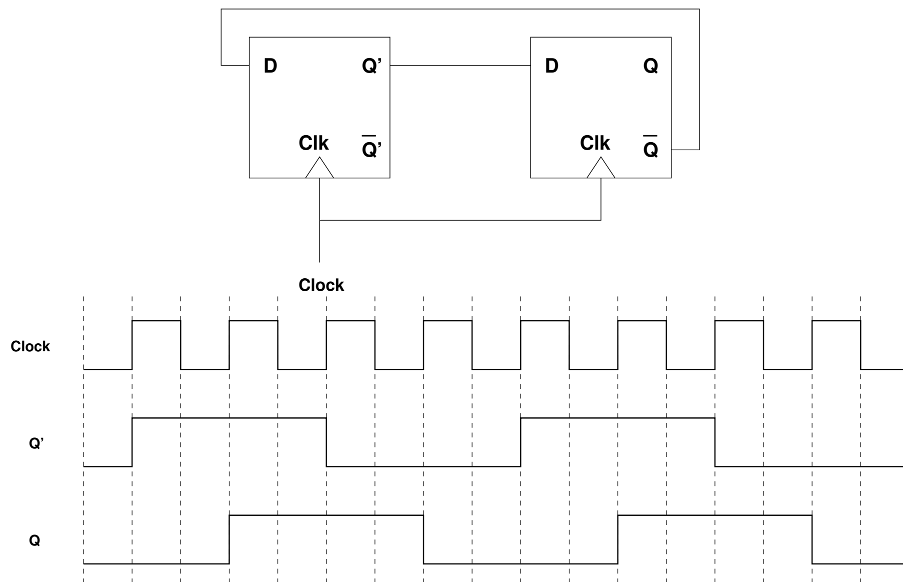
Q1



Q2

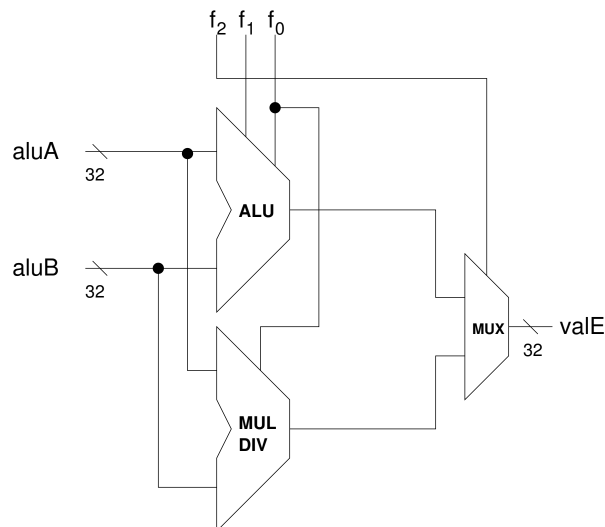


Q3 Voici une solution possible s'inpirant du compteur de *Johnson* :



4 Super ALU

Q1 Voici le codage (sur 3 bits) que l'on se propose d'utiliser pour les opérations : 000 = ADD, 001 = SUB, 010 = AND, 011 = XOR, 100 = MUL, 101 = DIV.



Q2 Si l'on suppose qu'une opération MUL ou DIV a besoin de 2 cycles pour se stabiliser, alors il faut injecter une *bulle* dans le pipeline au moment où l'instruction s'apprête à consommer son second cycle à l'étage *Execute*. Plus précisément, il s'agit d'injecter une opération `NOP` à l'étage *Memory* et de reconduire pour un cycle supplémentaire (i.e. *stall*) les instructions se trouvant aux étages *Fetch* et *Decode*.

Q3 Si l'on juxtapose ALU et MULDIV à l'étage *Execute*, alors il est possible d'effectuer une opération parmi ADD, SUB, AND, XOR pendant le second cycle d'une multiplication/division (à condition que les deux instructions soient indépendantes bien sûr). Dans l'état actuel de l'architecture Y86, cela pose quelques problèmes.

- En premier lieu, il faut pouvoir alimenter les deux circuits avec des valeurs potentiellement différentes, d'où l'introduction de nouvelles valeurs `muldivA` et `muldivB`.
- Ensuite, les deux instructions vont se retrouver simultanément aux étages *Memory* et *Write Back*. Cela ne pose pas de problème à l'étage *Memory* puisque ce type d'instruction ne sollicite pas la mémoire à cet étage. Par contre, il existe un problème à l'étape *Write Back* à cause de la collision sur `valE`. Pour le solutionner, il faudrait introduire deux valeurs distinctes `valE1` et `valE2` et étendre le fichier de registres de manière à autoriser trois écritures simultanées (`dstM`, `dstE1` et `dstE2`).
- Enfin, il faut gérer correctement la modification des codes de condition (`CC`). En l'occurrence, lorsque les deux unités de calcul terminent en même temps (ce qui signifie que MULDIV a débuté un cycle avant l'ALU), il faut que seuls les codes de conditions produits par l'ALU soient pris en compte.

Exercice 2 — Question 7

Conseil : utilisez la fonction *Rechercher* de votre lecteur PDF pour retrouver tous les « IJM » cachés dans ce qui suit...

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

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

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

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

################ 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, IJM } : 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, IJM } : 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, IJM } : 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, IJM };

## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };

## Select memory address
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL, IJM } : 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 in { IRET, IJM } : valM;
        # Default: Use incremented PC
        1 : valP;
];
```