INF1600 - TP 2

Samuel Rondeau
Pacôme Bondet

1er novembre 2014

Exercice 1 : architecture avec microcodes

a) recherche d'instruction

Donnez la séquence de microcodes qui correspond à la recherche d'une instruction. Donnez la réponse sous la forme d'un tableau. Le tableau suivant montre un exemple de ce qui est attendu (la première ligne étant déjà écrite pour vous) :

RTN concret	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**hexa**
MA ← PC ;	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0x3060
MD ← M[MA] : PC ← PC + 4 ;	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0x6CC0
IR ← MD ;	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0x8260

b) instruction générique

Sous la même forme qu'en a), écrivez la séquence de microcodes permettant d'exécuter l'instruction d'opérations arithmétiques/logiques typiques décrite par le RTN abstrait :

```
(IR<31..27> = opcode) ->
R[IR<26..22>] <- R[IR<21..17>] oper M[R[IR<16..12>] + IR<11..0>];
```

où opcode correspond au code d'opération requis pour exécuter l'opération arithmétique/logique oper .
N'oubliez pas d'inclure le RTN concret à chaque ligne du tableau pour justifier vos choix de signaux de contrôle.

RTN concret	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**hexa**
A ← R[rc] ;	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0x006E
MA ← A + const ;	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0x1021
MD ← M[MA] : A ← R[rb] ;	0	0	0	0	1	1	0	0	1	1	1	0	1	0	1	0	0x0CEA
R[ra] ← A oper MD ;	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0x8210

c) simulation

Prenez une capture d'écran et intégrez-la dans votre rapport. Cette capture doit bien montrer le résultat placé dans R[1] (ECX dans Electric) après l'exécution de l'instruction à l'adresse 8 (dans tp2mem.txt , donc après la 3ième instruction). Justifiez également le résultat obtenu dans votre rapport. Pour rappel, l'adresse présentement exécutée est contenue dans le registre PC. Votre capture d'écran doit donc montrer les cycles pendant lesquels PC vaut 8 (PC est affiché dans la cinquième ligne de la simulation).

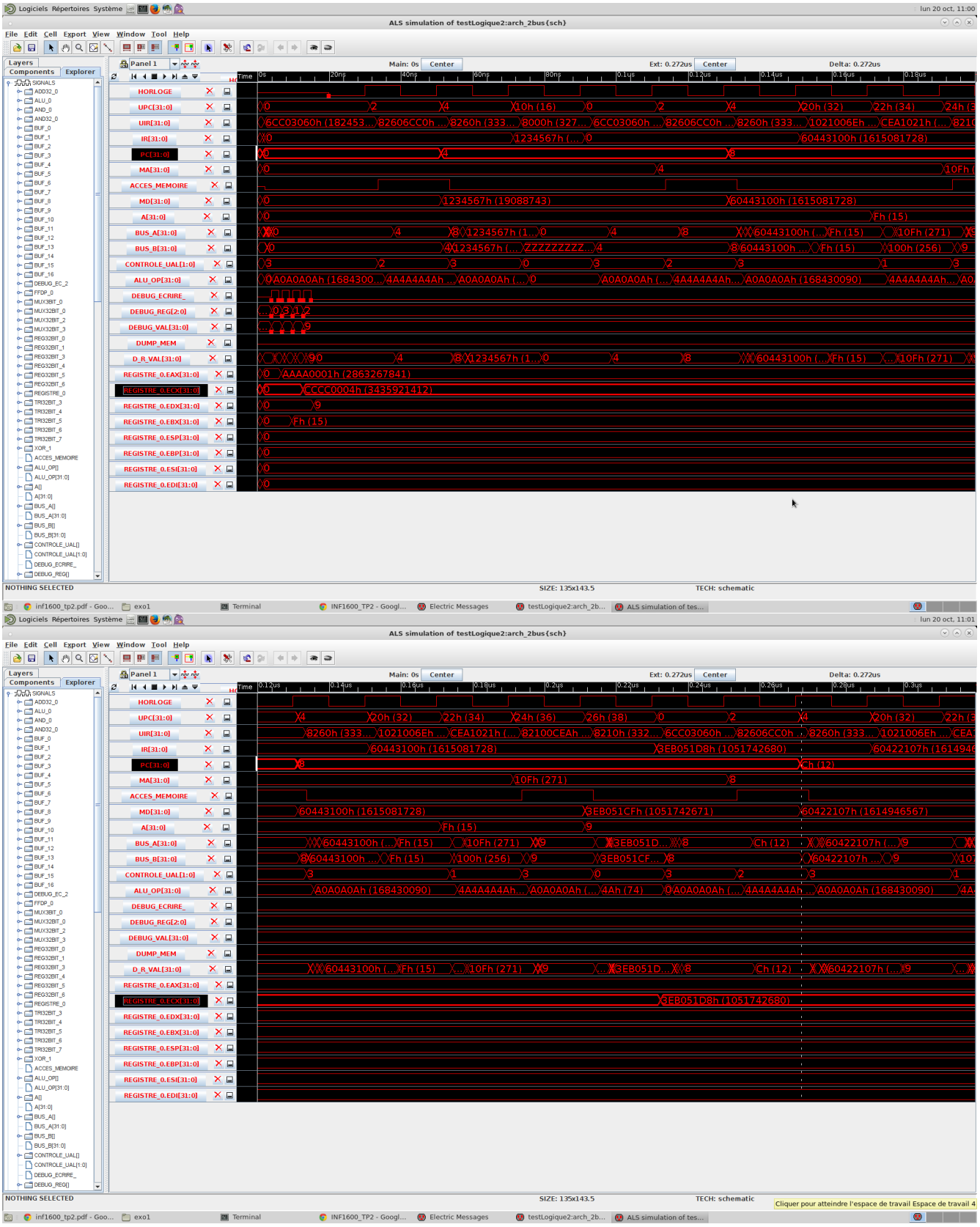

FIGURE 1 – Captures d'écran pour 1.c). La figure du haut représente la section de gauche (à partir de 0 ns) et la figure du bas représente la section de droite (jusqu'à 350 ns). La ligne pointillée montre le moment où PC cesse de valoir 8.

d) opération OR

Soit l'UAL définie dans la cellule ALU de la librairie du TP2 sur Electric. Quelle doit être la valeur de op[6 :0] pour que l'opération finale soit un OR (c'est-à-dire un OU logique bit à bit) ?
Vous devez naviguer dans Electric (CTRL+D et CTRL+U) pour répondre à cette question.
Écrivez cette valeur à l'endroit approprié du fichier tp2opalu.txt et relancez la simulation (redémarrez Electric pour être certain) afin de tester l'instruction OR de l'adresse 0xc dans tp2mem.txt .
Donnez aussi cette valeur dans votre rapport.

Étudions le code op de ADD dans tp2opalu.txt ainsi que l'ALU afin d'en comprendre le comportement et de trouver le bon code pour OR . Le bit op[6] doit être 0 puisque l'opération ne nécessite pas and32 . Le bit op[5] est aussi à 0 car on n'utilise pas add32 . Le bit op[4] est aussi à 0 car on n'utilise pas shift32 mais plutôt la branche 0 du multiplexeur. Ensuite, étudions la table de vérité de AND , pour vérifier comment obtenir op[3 :0] = 1010, soit A. On remarque que op[3 :0] est écrit selon la table de vérité suivant le Code Gray, le bit de poids fort étant celui du bas. Nous devrons en tenir compte pour déterminer op[3 :0] de OR , qui doit alors être op = 0000 1110, soit **0E**.
La simulation dans Electric est montrée à la figure 2.

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

(a) Usuelle

A	B	A ⊕ B
0	0	0
0	1	1
1	1	0
1	0	1

(b) Code Gray

TABLE 1 – Table de vérité de l'additionneur binaire (xor)

A	B	A | B
0	0	0
0	1	1
1	1	1
1	0	1

TABLE 2 – Table de vérité de l'opération or binaire

e) questions

1. Pour l'instruction 0x1234567 du processeur étudié dans ce TP, à quoi servent les données des deux derniers octets (0x4567) ? Donnez une autre instruction (sur 32 bits) qui aurait exactement le même effet d'exécution.

Les deux derniers octets correspondent à IR<15..0> , qui selon l'architecture, correspondent à la constante (inutilisée) et une partie de rc .
0x1234567 = 0x01 0x23 0x45 0x67. On remarque alors qu'il s'agit de l'instruction NOP (opcpde 0) de tp2mem.txt . En effet, l'opcode vaut IR<31..27> = 0 . Ainsi, la table de vérité op<3..0> vaut toujours 0, et les autres bits de l'opcode aussi. Ainsi, l' UAL n'effectue aucune opération. On pourrait donc écrire l'instruction 0x0 qui ferait la même chose, rien.

2. Nommez un avantage d'avoir une architecture à deux bus. Vous êtes -vous servi de cet avantage dans votre microprogramme développé en b) ?

Il est possible d'effectuer plus d'instructions en parallèlle, c'est-à-dire envoyer une information sur le bus A et une information sur le bus B pour faire deux opérations à la fois. C'est ce que nous avons fait à l'étape MD ← M[MA] : A ← R[rb] ; au b).

3. Diriez-vous que les instructions de cette architecture peuvent être aussi flexibles, en terme d'opérations arithmétiques/logiques, que celles du processeur étudié à l'exercice 5 du TP1 ? Pourquoi ?

Oui, voire même plus. Elle contient deux bus, ce qui permet plus d'opérations en parallèle. Elle contient également plus de registres temporaires au lieu de T, pour stocker les données à envoyer à l'UAL. De plus, l'accès à la mémoire est détachée de l'architecture principale, et ne requiert donc ni le bus A ni le bus B.

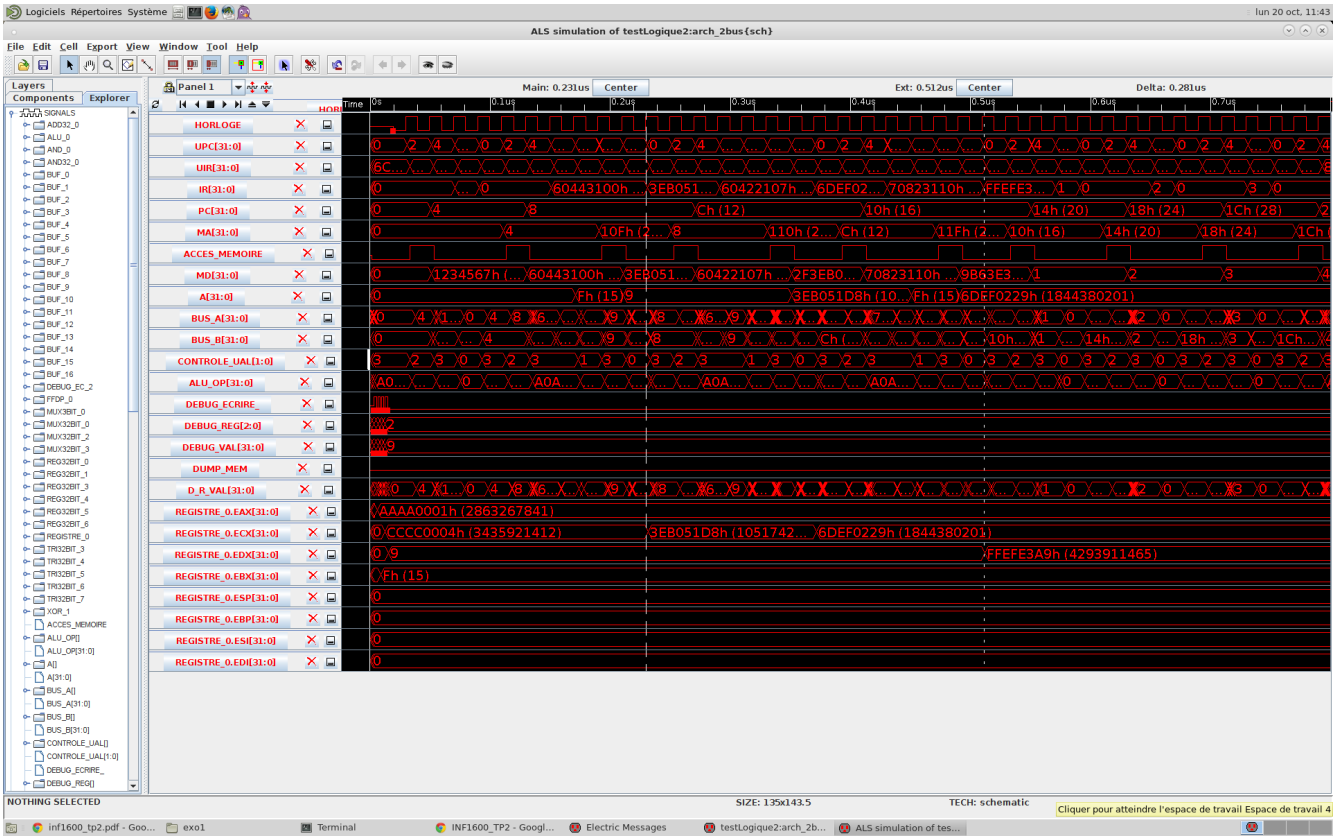

FIGURE 2 – Capture d'écran pour 1.d). La première ligne pointillée montre le moment où les 2 opérations AND sont effectuées. La seconde ligne pointillée montre le moment où l'opération OR est effectuée. Chaque opération correspond au bon PC décalé, puisque PC <- PC + 4 .

Exercice 2 : assembleur avec processeur à pile

Écrivez l'expression suivante en assembleur :

$$a = \left(\frac{b \times c}{f + c}\right)\left(\frac{g - d}{e}\right) + e \times (g - d)$$

Dans le code en assembleur, nul besoin de déclarer des étiquettes, car `a`, `b`, `c`, `d`, `e`, `f` et `g` sont déclarées commes variables globales dans la fonction principale. Ainsi, on peut tout simplement écrire le code assembleur suivant qui donne le même résultat que l'instruction en C.

```
.global func_s

func_s:
    flds b
    flds c
    fmulp
    fstps a    #a = b * c
    flds f
    flds c
    faddp
    flds a
    fdivp
    fstps a    #a = (b * c) / (f + c)
    flds g
    flds d
    fsubrp
    flds e
    fdivrp
    flds a
    fmulp
    fstps a    #a = ((b * c) / (f + c)) * ((g - d) / e)
    flds g
    flds d
    fsubrp
    flds e
    fmulp
    flds a
    faddp
    fstps a    #a = (((b * c) / (f + c)) * ((g - d) / e)) + (e * (g - d))


    ret
```

Exercice 3 : conditions et branchements

Écrivez la séquence suivante décrite en langage C :

```
a = b;
if (c + 1600 > e + 2013) {
    a = c;
    if ((b <= c) || (d == e)) {
        a = e;
    }
} else {
    a = a + b;
}
```

où a , b , c , d et e sont des entiers signés (type *int* en langage C) sur 32 bits. Vous pouvez utiliser directement ces symboles pour représenter leurs adresses en assembleur, comme à l'exercice 2.

```
.global func_s

func_s:
    mov b, %ebx
    mov %ebx, a        #a = b
    mov c, %ecx
    add $1600, %ecx    #(c + 1600)
    mov e, %edx
    add $2013, %edx    #(e + 2013)
    cmp %ecx, %edx
    jae else           #si (c + 1600 <= e + 2013) sauter au else
    mov c, %ecx
    mov %ecx, a        # a = c
    cmp %ebx, %ecx
    jae if2            #si (b <= c) entrer dans le if
    mov d, %edx
    mov e, %ecx
    cmp %edx, %ecx
    je if2             #sinon, si (d == e) entrer dans le if
    jmp fin            #sinon, quitter

if2:
    mov e, %ecx
    mov %ecx,a         #a = e
    jmp fin

else:
    mov a, %eax
    add b, %eax        #a = a + b
    mov %eax, a

fin:
    ret
```