

13 mars, 2018

## TP2 - Architecture à deux bus et introduction à l’assembleur IA-32

INF1600 : Architecture des micro-ordinateurs

Groupe 1 (B2)

Présenté à

M Saif-Eddine Sajid

Nicole Joyal (1794431)

Jennifer Girard (1875532)

Exercice 1 – Architecture avec microcodes

Questions

1. Recherche d’instruction :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **RTN** | **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 |

1. Exécution d’une instruction générique :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **RTN** | **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 + IR<11..0>; | 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 |

1. Simulation

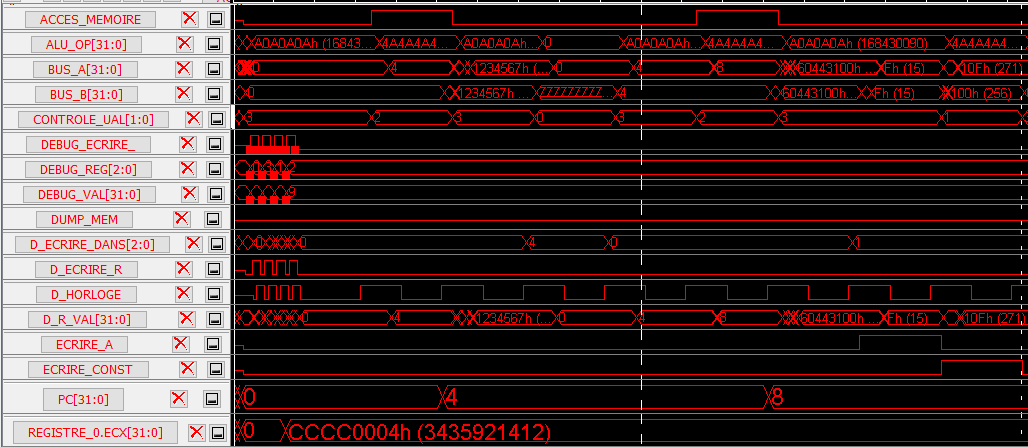


Figure 1: Simulation jusqu'à l'instruction 0x8



Figure 2: Suite jusqu'à l'instruction 0xC

1. L’opération NAND :

En observant le schéma de l’*ALU*, nous sommes capables de déduire un *opcode* de manière logique. Puisque nous ne nous servons pas de l’unité *and32* uniquement, op[6] aura la valeur 0. Ensuite, les unités *add32* et *shift32* ne nous sont pas utiles pour l’opération voulue, op[5] et op[0] seront donc à 0. Il nous reste les bits [3 :0], allant dans l’unité *oplogique32*, ainsi que op[4] du multiplexeur. En faisait la table de vérité de NAND (tableau I) et en alignant les bits résultants, nous avons 0111. Puisque la réponse de l’opération logique est situé à la position 0 du multiplexeur, la valeur de op[4] sera de 0.

Nous finissons avec le code suivant : 0b0000111, ce qui équivaut à 0x07.

Tableau I : Table de vérité NAND

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A !& B** |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

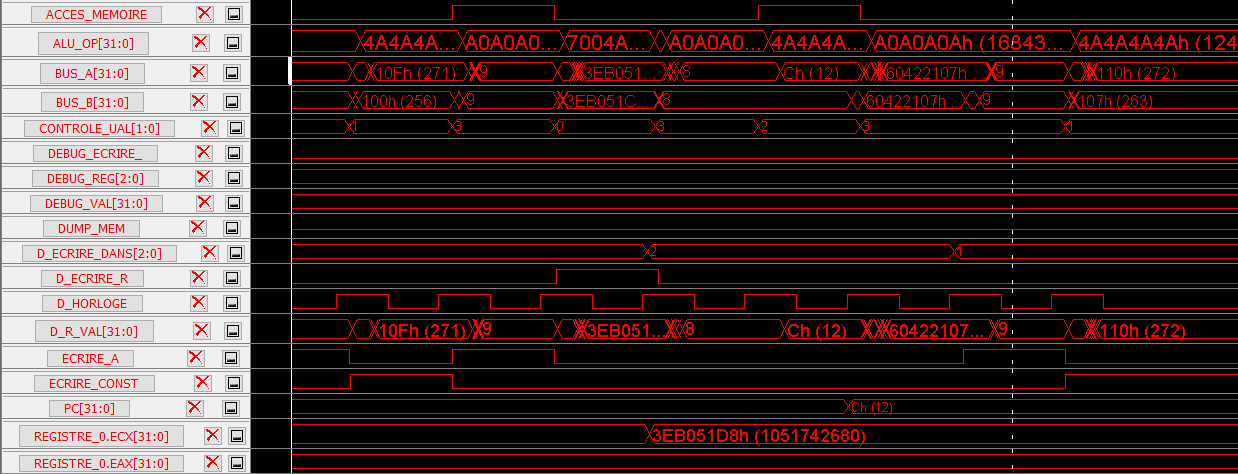


Figure 3: Simulation avec opcode 0x07

1. Compréhension :

Selon la décomposition des instructions, les bits 0x5555 correspondent à IR<16..0>, car les instructions sont stockées en *little-endian*. Ceci dit, les bits IR<11..0> sont inutilisés. Cependant, les bits IR<16..12> sont utilisés en *rc*, il faut donc que ces bits correspondants demeurent les mêmes afin de garder l’instruction inchangée. Une autre instruction pourrait être : 0x0005…



Un avantage d’une architecture à deux bus est que ceci nous permet d’envoyer une donnée vers la mémoire ou les registres et vice-versa, dans un seul cycle d’horloge. Nous nous sommes servies de cet avantage dans le deuxième exercice, notamment à la 4e instruction (R[ra] <- A oper MD;). Nous avions pu faire l’opération et directement stocker la réponse dans un registre. L’architecture à deux bus nous évité l’utilisation du registre temporaire de l’UAL.



L’architecture dans ce tp peux être plus flexible que celle du dernier tp. En effet, l’utilisation d’une architecture à deux bus permet d’envoyer deux instructions simultanément entre les composantes, ce qui causerait un problème dans le tp1. Ceci rends donc les opérations arithmétiques et logiques plus flexibles.

Exercice 2 – Assembleur avec processeur à pile

Code assembleur

.global func\_s

func\_s:

flds b

flds d #b = s[1], d = s[0]

fmulp #b \* d = s[0]

flds c #c = s[0], b\*d = s[1]

fsubrp #s[1] - s[0] = s[0]

fstps a #résultat mis dans a

flds f

flds g #f = s[1], g = s[0]

fsubrp #s[1] - s[0] = s[0]

flds a #résultat sous. = s[1], c = s[0]

fdivp # s[0] / s[1] = s[0]

flds e # résultat div. = s[1], e = s[0]

faddp # s[0] + s[1] = s[0]

fstps a # s[0] mis dans a

flds e

flds g # e = s[1], g = s[0]

fsubp # s[0] - s[1] = s[0]

flds f # résultat = st[1], f = st[0]

fdivrp # st[1] / st[0] = st[0]

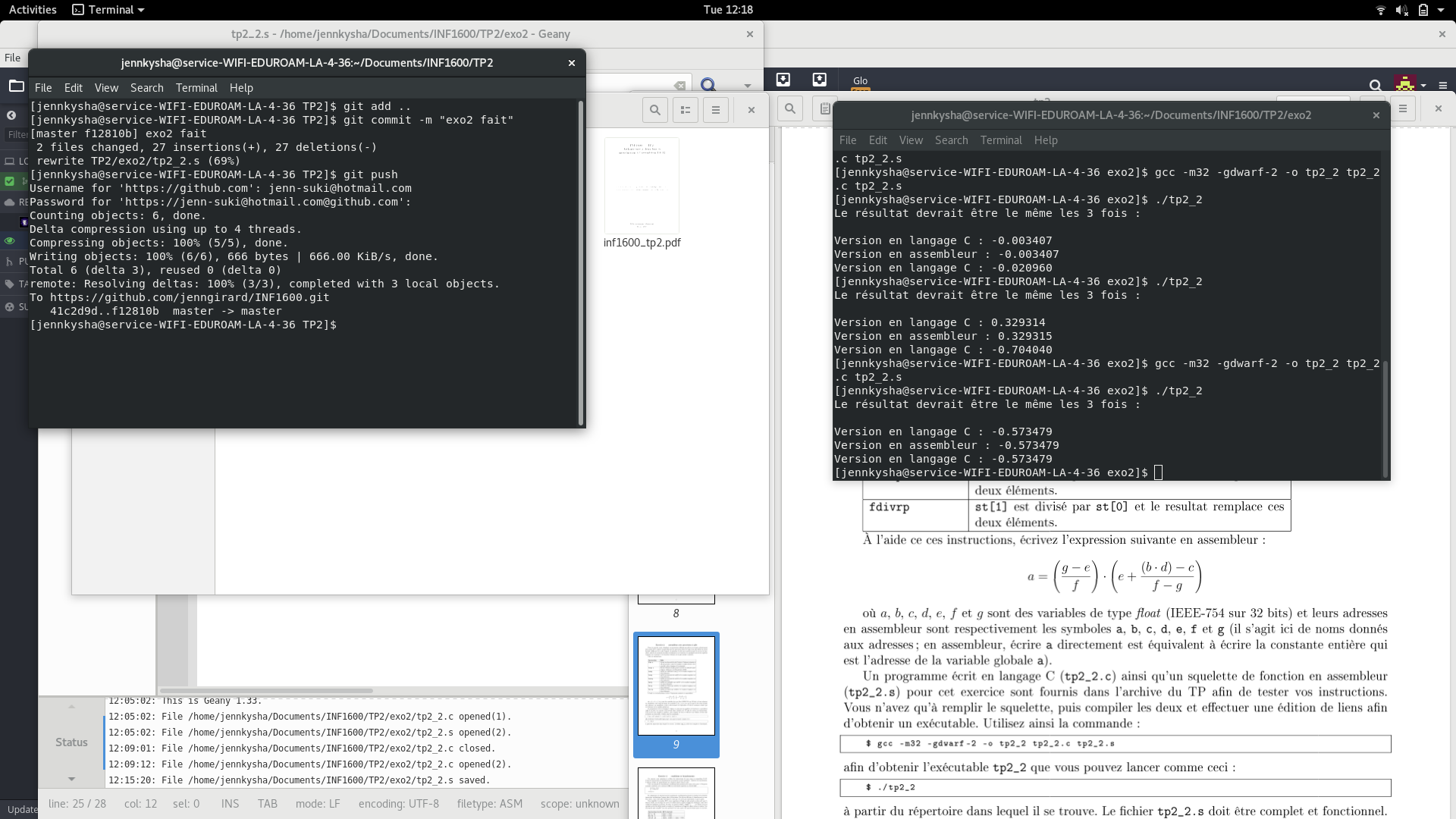
flds a # résultat = st[1], c = st[0]

fmulp # st[0] \* st[1] = st[0]

fstps a # met st[0] dans a

ret

Compilation et exécution



Exercice 3 – Conditions et branchements

Code assembleur

.data

.global func\_s

func\_s:

mov $10, %esi

mov $0, %edi

boucle:

mov d, %eax

mov e, %ebx

add %ebx, %eax # d+e

mov b, %ebx

sub %ebx, %eax #(d+e)-b

mov %eax, a # a = (d+e)-b

mov b, %ebx

sub $1000, %ebx # b-1000

mov c, %edx

add $500, %edx # c+500

cmp %ebx, %edx # (c+500) - (b-1000)

ja conditionIf # si (c+500) - (b-1000) > 0, jmp if

jnae conditionElse

conditionIf:

mov c, %edx

sub $500, %edx # c-500

mov %edx, c

mov b, %eax

cmp %edx, %eax # b - c

ja conditionIf2 # si b - c > 0

jmp break

conditionIf2:

mov b, %ebx

sub $500, %ebx # b-500

mov %ebx, b

jmp break

conditionElse:

mov b, %ebx

mov e, %eax

sub %eax, %ebx # b-e

mov %ebx, b

mov d, %edx

add $500, %edx # d+500

mov %edx, d

jmp break

break:

add $1, %edi #incremente compteur

cmp %edi, %esi # 10-compteur

jae boucle # si 10-compteur >= 0, boucle

ret