Name	**:**	**HTET NAING**
Lab Group	**:**	**SS1**
Module	**:**	**CZ3001**
Assignment	**:**	**LAB4 REPORT**
Submission Date	**:**	**April 3rd, 2017**

a. The MIPS assembly code for the computation of "e = (a+b) ^ (c*d)"

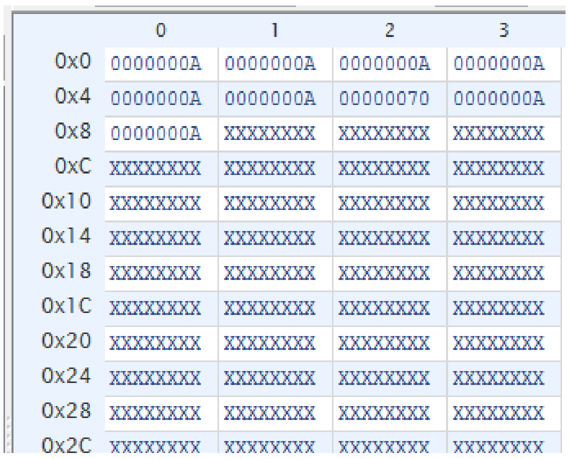
```
LW $4, 2($0)        # load a to R4
LW $5, 3($0)        # load b to R5
ADD $5, $4, $5      # R5 ← R4 + R5
LW $4, 4($0)        # load c to R4
LW $6, 5($0)        # load d to R6
MUL $4, $6, $4      # R4 ← R6 * R4
XOR $4, $5, $4      # R4 ← R5 ^ R4
SW $4, 6($0)        # store final result from R4 to e
```

b. The MIPS assembly code for the computation of "e = (a+b) ^ (c*d)" including NOPs for removing data-dependencies

```
LW $4, 2($0)        # load a to R4
LW $5, 3($0)        # load b to R5
NOP                 # 1 stall cycle
NOP                 # 1 stall cycle
ADD $5, $4, $5      # R5 ← R4 + R5
LW $4, 4($0)        # load c to R4
LW $6, 5($0)        # load d to R6
NOP                 # 1 stall cycle
NOP                 # 1 stall cycle
MUL $4, $6, $4      # R4 ← R6 * R4
NOP                 # 1 stall cycle
NOP                 # 1 stall cycle
XOR $4, $5, $4      # R4 ← R5 ^ R4
NOP                 # 1 stall cycle
NOP                 # 1 stall cycle
SW $4, 6($0)        # store final result from R4 to e
```

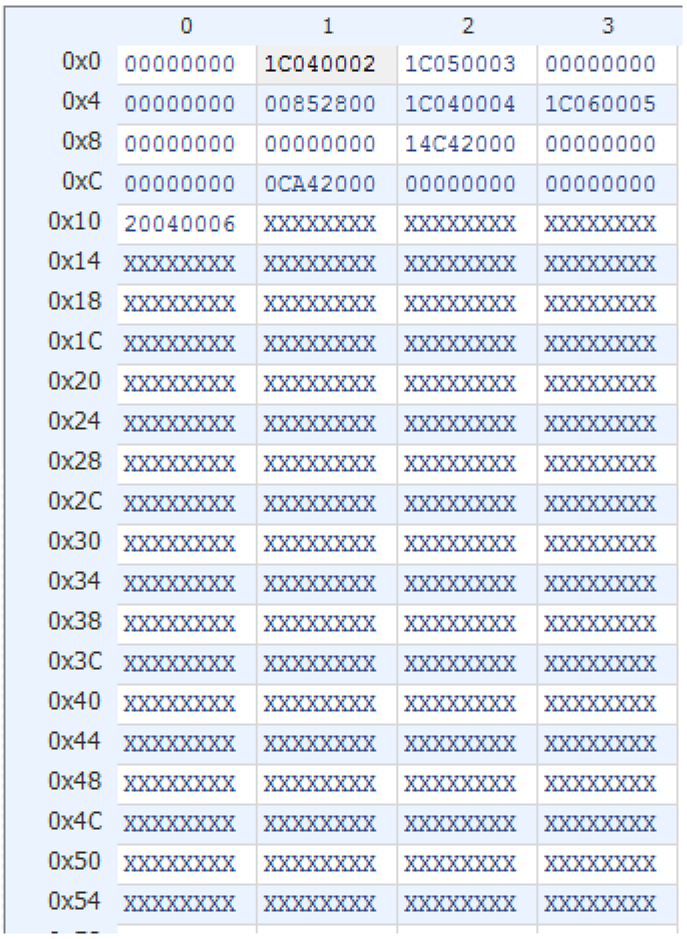
c. The instruction and data-memory (all values in hexadecimal)

Data Memory:



	0	1	2	3
0x0	0000000A	0000000A	0000000A	0000000A
0x4	0000000A	0000000A	00000070	0000000A
0x8	0000000A	XXXXXXXX	XXXXXXXX	XXXXXXXX
0xC	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x10	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x14	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x18	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x1C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x20	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x24	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x28	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x2C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

Instruction Memory:



	0	1	2	3
0x0	00000000	1C040002	1C050003	00000000
0x4	00000000	00852800	1C040004	1C060005
0x8	00000000	00000000	14C42000	00000000
0xC	00000000	0CA42000	00000000	00000000
0x10	20040006	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x14	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x18	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x1C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x20	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x24	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x28	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x2C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x30	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x34	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x38	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x3C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x40	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x44	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x48	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x4C	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x50	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x54	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

d. The working of both the LW and SW instruction with reference to ISIM window

(i) LW instruction

LW $4, 2($0) from the above MIPS assembly code will be taken as an example to explain the working of LW instruction. **It is assumed that the fetch stage and decode stage takes one cycle each.**

In the first cycle : PC provides the address 0x00000001 to the instruction memory so that the LW instruction can be fetched.

In the second cycle : the LW instruction is decoded to generate the control signals and to get the data from the register file. Hence, the ID_EXE pipeline register now has the following values:

rdata1_ID_EXE	=	the address 0x00000000 (the first source for the ALU)
imm_ID_EXE	=	0x00000002 (sign-extended immediate value)
waddr_ID_EXE	=	0x04 (the address to write back to register $4)
alusrc_ID_EXE	=	"High" (the signal is asserted, i.e. the second source for the ALU will be taken from imm_ID_EXE instead of rdata2_ID_EXE)

In the third cycle : ALU executes the add operation for the two sources (0x00000000 + 0x00000002) and produces the result 0x00000002. The EXE_MEM pipeline register has the following values:

aluout_EXE_MEM	=	0x00000002
waddr_EXE_MEM	=	0x04 (It has been brought forward to this stage)

In the fourth cycle : The address (0x00000002) that is output from ALU will be used to read the content from the data memory. The value 0x0000000a is now retrieved from the data memory. However, the output of DMEM is not taken through MEM/WB pipeline register. Hence, now the MEM_WB pipeline has the following value:

waddr_MEM_WB	=	0x04 (it has been brought forward again)

In the fifth cycle : The ALU output or the data memory output is chosen with the help of the control signal. Hence, wdata_WB now holds the value 0x0000000a. It is written back to the register file using the write address 0x04 that have been carried forward through all stages. Finally, the register $4 is successfully loaded with the value 0x0000000a.

(ii) SW instruction

SW $4, 6($0) from the above MIPS assembly code (used for lab4) will be taken as an example to explain the working of SW instruction. **It is assumed that the fetch stage and decode stage takes one cycle each.**

In the first cycle : PC provides the address 0x00000010 to the instruction memory so that the SW instruction can be fetched.

In the second cycle : the SW instruction is decoded to generate the control signals and to get the data from the register file. Hence, the ID_EXE pipeline register now has the following values:

rdata1_ID_EXE	=	the address 0x00000000 (the first source for the ALU)
rdata2_ID_EXE	=	0x00000070 (the content from $4) (It will serve as the write data for Data Memory)
imm_ID_EXE	=	0x0000006 (sign-extended immediate value)
alusrc_ID_EXE	=	"High" (the signal is asserted, i.e. the second source for the ALU will be taken from imm_ID_EXE instead of rdata2_ID_EXE)

In the third cycle : ALU executes the add operation for the two sources (0x00000000 + 0x00000006) and produces the result 0x00000006. The EXE_MEM pipeline register has the following values:

aluout_EXE_MEM	=	0x00000006
rdata2_EXE_MEM	=	0x00000070 (It will serve as the write data for Data Memory)

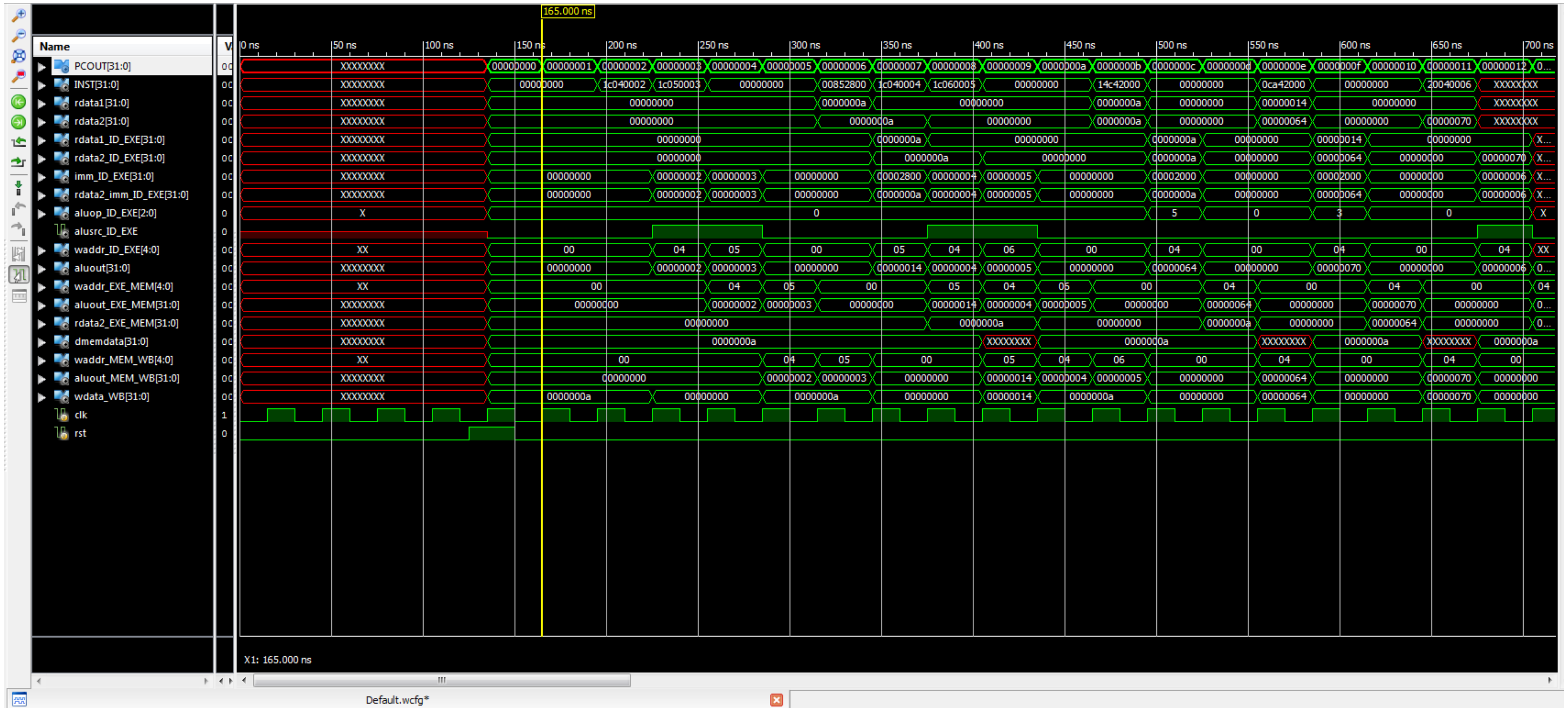
In the fourth cycle : The data 0x00000070 will be written to memory location 0x00000006 (i.e. output from ALU). Thus the required result is successfully stored there eventually.

e. Execution Time = Ending Time – Starting Time
= 765ns – 165ns
= 600ns

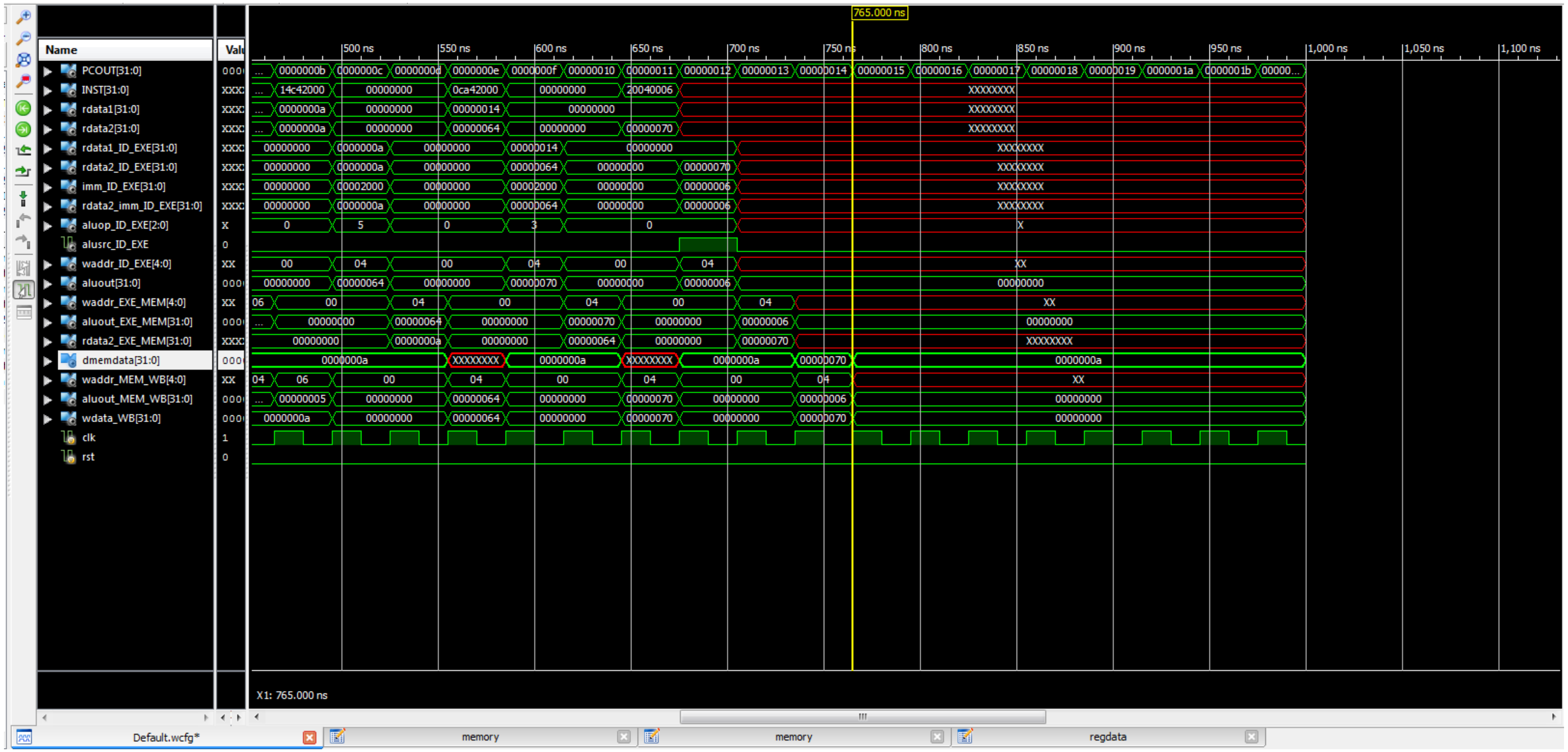
f. The steady state CPI

$$\boldsymbol{CPI} = \frac{(\textbf{no. of instructions})+(\textbf{no. of stalls})}{\textbf{no. of instructions}}$$
$$= \frac{(\mathbf{8})+(\mathbf{8})}{\mathbf{8}}$$
$$= \mathbf{2}$$

Starting time: 165 ns



Ending time: 765 ns


765.000 ns
Name
Value
PCOUT[31:0]
INST[31:0]
rdata1[31:0]
rdata2[31:0]
rdata1_ID_EXE[31:0]
rdata2_ID_EXE[31:0]
imm_ID_EXE[31:0]
rdata2_imm_ID_EXE[31:0]
aluop_ID_EXE[2:0]
alusrc_ID_EXE
waddr_ID_EXE[4:0]
aluout[31:0]
waddr_EXE_MEM[4:0]
aluout_EXE_MEM[31:0]
rdata2_EXE_MEM[31:0]
dmemdata[31:0]
waddr_MEM_WB[4:0]
aluout_MEM_WB[31:0]
wdata_WB[31:0]
clk
rst
500 ns
550 ns
600 ns
650 ns
700 ns
750 ns
800 ns
850 ns
900 ns
950 ns
1,000 ns
1,050 ns
1,100 ns
X1: 765.000 ns
Default.wcfg*
memory
memory
regdata