# CZ3001: Advanced Computer Architecture

## Lab 4 Report

Sharma Shantanu
U1622895F

Lab Group: SSP1

Write the MIPS assembly code for the computation of "e = (a-b)*(c^d)". Note that all variables are integers. The addresses and data of the variables are given in table below. You can load this data to data memory.

| Data | a=0xD | b=0x3 | c=0xE | d=0x2 | e=0xF |
|---|---|---|---|---|---|
| Addresses | 0x00000001 | 0x00000002 | 0x00000003 | 0x00000004 | 0x00000005 |

^ indicates XOR operation

* indicates multiply operation

 - indicates subtraction operation

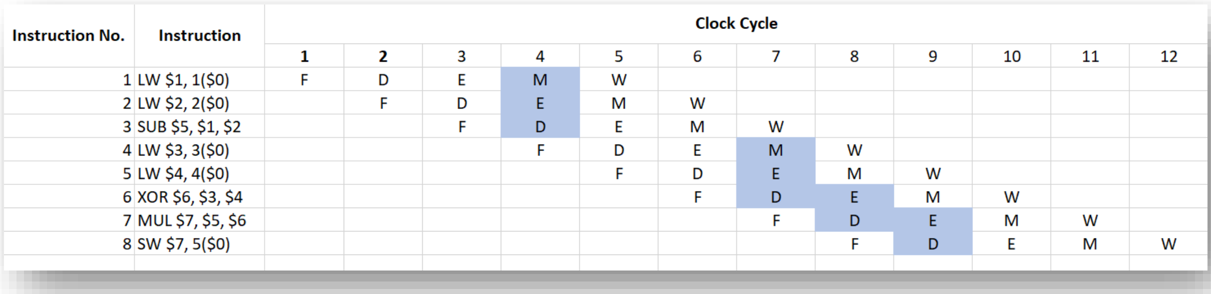
**a) Write the MIPS assembly code for the computation of "e = (a-b)*(c^d)" with minimum number of instructions.**

**Answer:**

```
LW $1, 1($0)   //Load content of address 1 in Register 1 ($1=0xD)
LW $2, 2($0)   //Load content of address 2 in Register 2 ($2=0x3)
SUB $5, $1,$2  //Subtract a-b, store in register 5
LW $3, 3($0)   //Load contents of address 3 in Register 3 ($3=0xE)
LW $4, 4($0)   //Load contents of address 4 in Register 4 ($3=0x2)
XOR $6, $3,$4  //XOR: c^d, store in register 6
MUL $7, $5, $6   //Multiply a-b and c^d and store in register 7
SW $7, 5($0)   //Store contents of register 7 in address 5
```

**b) Modify the MIPS assembly code for the computation of "e = (a-b)*(c^d)" for a five stage pipelined architecture given in lab 4, after including NOPs for removing data-dependencies.**

**Answer:**



| Instruction No. | Instruction | Clock Cycle | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 1 | LW $1, 1($0) | F | D | E | M | W | | | | | | | |
| 2 | LW $2, 2($0) | | F | D | E | M | W | | | | | | |
| 3 | SUB $5, $1, $2 | | | F | D | E | M | W | | | | | |
| 4 | LW $3, 3($0) | | | | F | D | E | M | W | | | | |
| 5 | LW $4, 4($0) | | | | | F | D | E | M | W | | | |
| 6 | XOR $6, $3, $4 | | | | | | F | D | E | M | W | | |
| 7 | MUL $7, $5, $6 | | | | | | | F | D | E | M | W | |
| 8 | SW $7, 5($0) | | | | | | | | F | D | E | M | W |

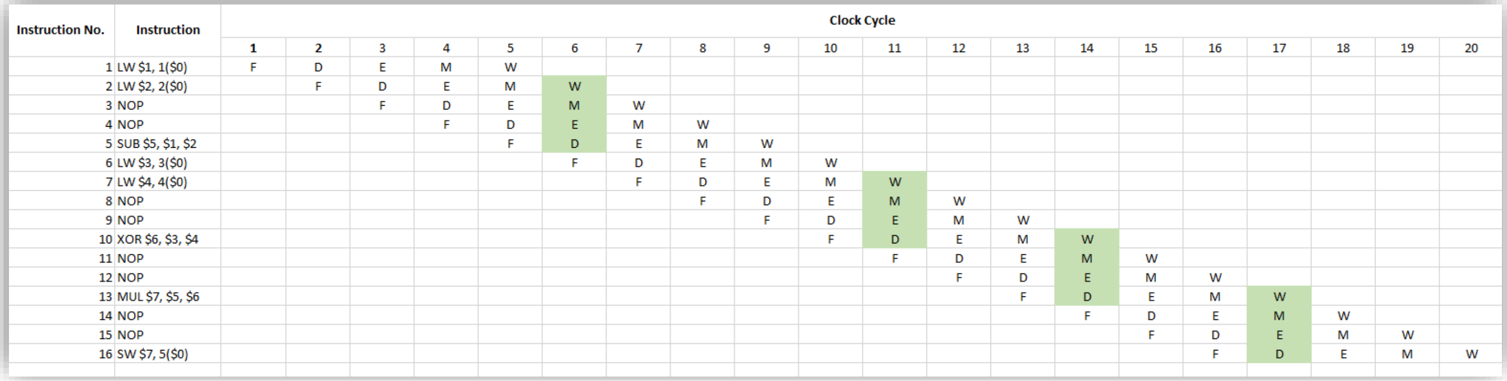
There are 3 types of data dependencies affecting a MIPS program:

- True/RAW (Read-After-Write)
- Output/WAW(Write-After-Write)
- Anti/WAR(Write-After-Read)

Remainder dependencies are based on name and not value. RAW dependencies take place when value in a register is changed by an instruction and then this new value is used in a subsequent instruction. Improper execution can lead to erratic behaviour and values. These dependencies can be handled by h/w or s/w stalls. S/w stalls are called No Operations or NOPS. These are software instructions that will not affect the values of any registers or data memory but would end up consuming one clock cycle per NOP.

```
LW $1, 1($0)
LW $2, 2($0)
NOP
NOP
SUB $5, $1, $2
LW $3, 3($0)
LW $4, 4($0)
NOP
NOP
XOR $6, $3, $4
NOP
NOP
MUL $7, $5, $6
NOP
NOP
SW $7, 5($0)
```

The new minimum number of instructions for the computation: 13



| Instruction No. | Instruction | Clock Cycle | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 | LW $1, 1($0) | F | D | E | M | W | | | | | | | | | | | | | | | |
| 2 | LW $2, 2($0) | | F | D | E | M | W | | | | | | | | | | | | | | |
| 3 | NOP | | | F | D | E | M | W | | | | | | | | | | | | | |
| 4 | NOP | | | | F | D | E | M | W | | | | | | | | | | | | |
| 5 | SUB $5, $1, $2 | | | | | F | D | E | M | W | | | | | | | | | | | |
| 6 | LW $3, 3($0) | | | | | | F | D | E | M | W | | | | | | | | | | |
| 7 | LW $4, 4($0) | | | | | | | F | D | E | M | W | | | | | | | | | |
| 8 | NOP | | | | | | | | F | D | E | M | W | | | | | | | | |
| 9 | NOP | | | | | | | | | F | D | E | M | W | | | | | | | |
| 10 | XOR $6, $3, $4 | | | | | | | | | | F | D | E | M | W | | | | | | |
| 11 | NOP | | | | | | | | | | | F | D | E | M | W | | | | | |
| 12 | NOP | | | | | | | | | | | | F | D | E | M | W | | | | |
| 13 | MUL $7, $5, $6 | | | | | | | | | | | | | F | D | E | M | W | | | |
| 14 | NOP | | | | | | | | | | | | | | F | D | E | M | W | | |
| 15 | NOP | | | | | | | | | | | | | | | F | D | E | M | W | |
| 16 | SW $7, 5($0) | | | | | | | | | | | | | | | | F | D | E | M | W |

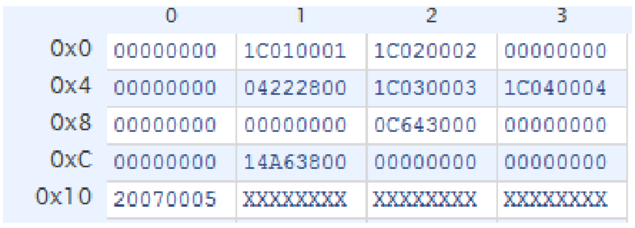
The data dependencies are removed by inserting NOP instructions,. Ex. cycle 6: value from memory add. 1 and 2 would have written to reg. $1 and $2, in time for inst. 5 to use them in the decode stage.

**c) Show the snapshot of instruction and data-memory (all values in hexadecimal) from the ISIM simulation window.**

**Answer:**

**imem_test0.txt**

```
00000000 00000000 // NOP
00000001 1C010001 // LW $1, 1($0)
00000002 1C020002 // LW $2, 2($0)
00000003 00000000 // NOP
00000004 00000000 // NOP
00000005 04222800 // SUB $5, $1, $2
00000006 1C030003 // LW $3, 3($0)
00000007 1C040004 // LW $4, 4($0)
00000008 00000000 // NOP
00000009 00000000 // NOP
0000000A 0C643000 // XOR $6, $3, $4 0000000B 00000000 // NOP
0000000C 00000000 // NOP
0000000D 14A63800 // MUL $7, $5, $6 0000000E 00000000 // NOP
0000000F 00000000 // NOP
00000010 20070005 // SW $7, 5($0)
```

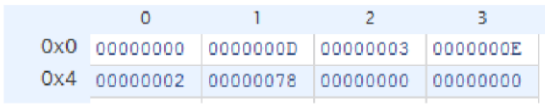


| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0x0 | 00000000 | 1C010001 | 1C020002 | 00000000 |
| 0x4 | 00000000 | 04222800 | 1C030003 | 1C040004 |
| 0x8 | 00000000 | 00000000 | 0C643000 | 00000000 |
| 0xC | 00000000 | 14A63800 | 00000000 | 00000000 |
| 0x10 | 20070005 | XXXXXXXX | XXXXXXXX | XXXXXXXX |

MIPS code is translated to 32-bit hex. instructions and stored in a text file to simulate the ***instruction memory.*** First column: address of instruction, Second column: Instruction

**Dmem_test0.txt**

```
00000000 00000000
00000001 0000000D
00000002 00000003
00000003 0000000E
00000004 00000002
00000005 0000000F
00000006 00000000
00000007 00000000
```



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0x0 | 00000000 | 0000000D | 00000003 | 0000000E |
| 0x4 | 00000002 | 00000078 | 00000000 | 00000000 |

The data is stored in a text file to simulate the data memory.

First column: Add. Of data

Second Column: Data value

**d) Explain the working of the five stage pipeline both for LW and SW instruction (used in this code) using ISIM window as reference.**

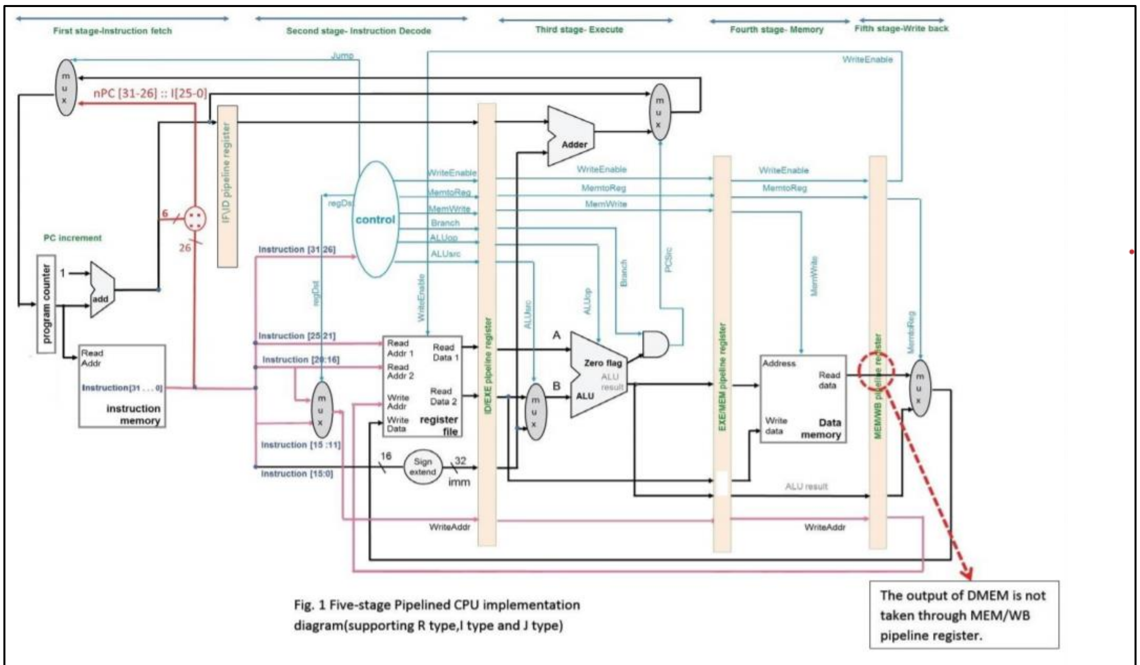
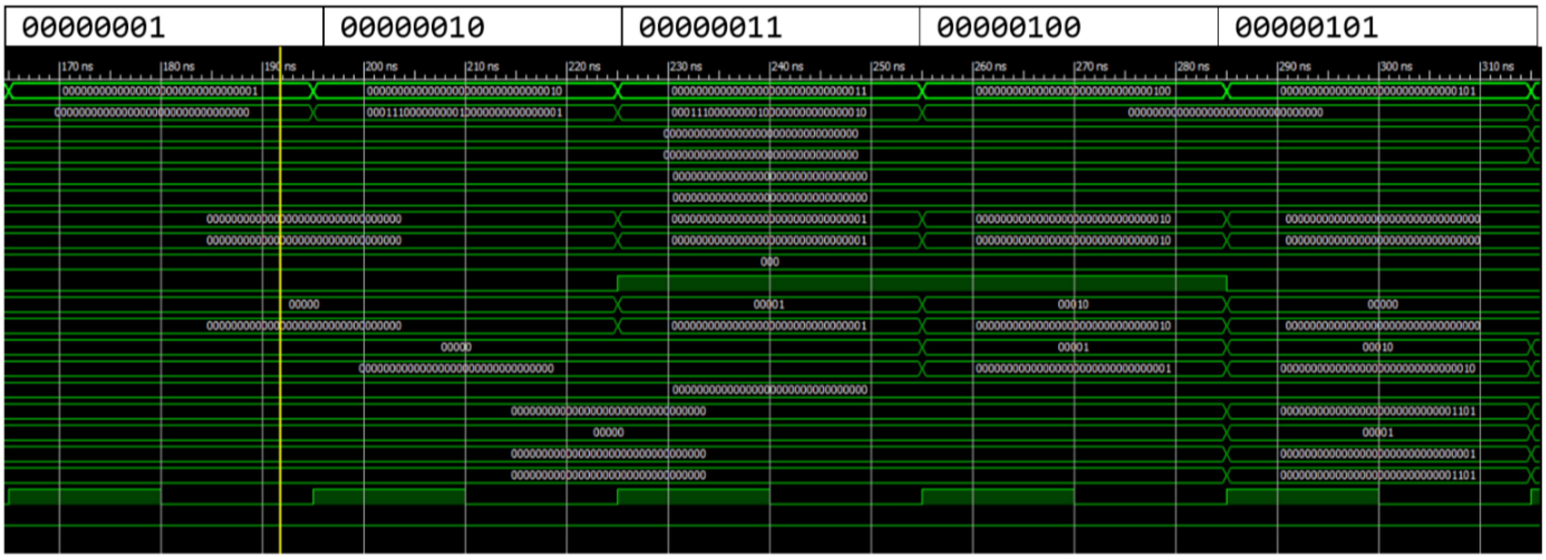


Fig. 1 Five-stage Pipelined CPU implementation diagram



## Load Word (LW) Instruction

Example: **LW $1, 1($0)**

**(1) Fetch**

In the fetch state, the PC value is 0X00000000. This is fed into the instruction memory which fetches the instruction. In the meanwhile the PC is incremented by 1 and is stored in the IF/ID pipeline registers. Instruction fetched is given to **INST[31:0],** value: “0x1C010001”

**(2) Decode** :

The 32-bit address is then decoded. INST[31:26] is fed in control logic, producing control signals for variables: ALU opcode, branch, write-back. The destination register (waddr) $1 is decoded and passed to dec/exec pipeline register. The immediate value 1 is also passed into the pipeline register. The read addr. 1 is set to $0 and is also passed to the pipeline register.

**(3) Execute** :

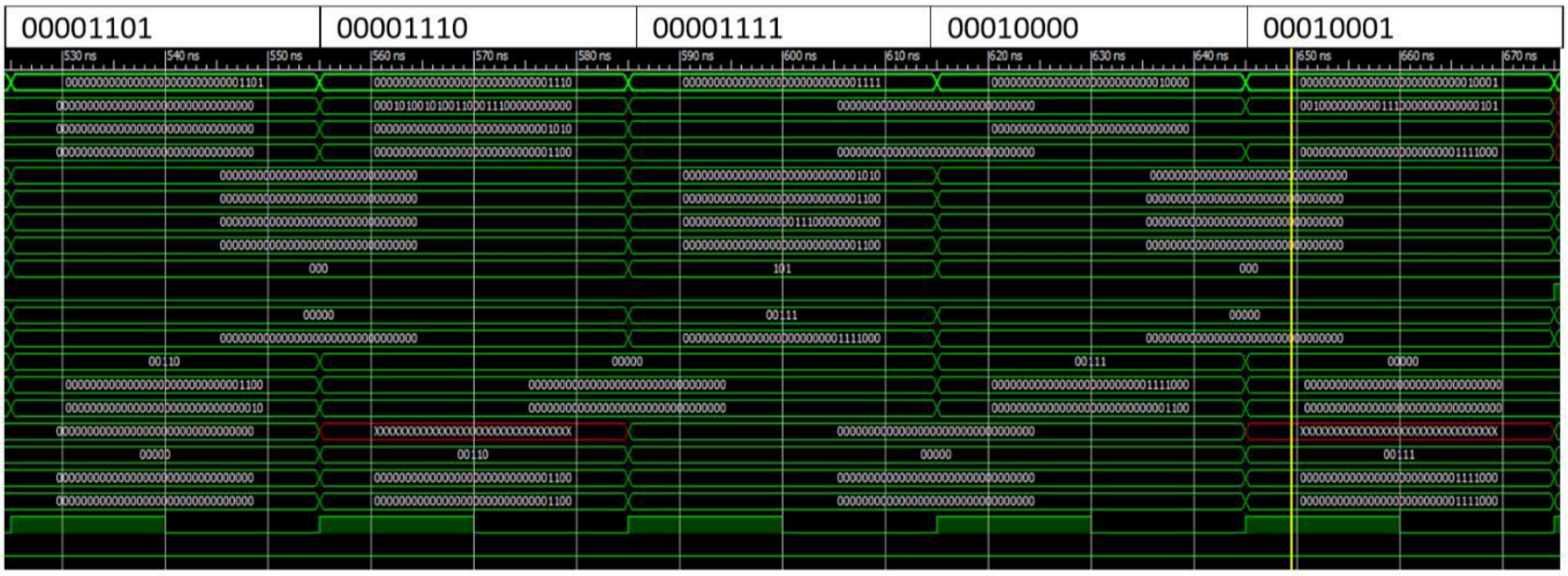
The alusrc is set to 1 indicating that the immediate value must be taken instead of the value from reg. The ALU adds the value of the content of register 0 and imm. 1. This contains the add. is in the data memory. The previous values from the ID/EXE register is continued to pass into this pipeline registers. The value is passed to the Execute/Memory state. **aluout**, will have an output of 1.

**(4) Memory**:

From the EXE/MEM pipeline register, the address that needs to look up in the Data memory is passes into the data memory block and the value is read.

**(5) Write Back**:

Value retrieved from accessing the data memory is 0xC=12. The value is then written back to the destination register $1.



**Store Word (SW) Instruction**

Example: **SW $7, 5($0)**

***Assume that the fetch stage and decode stage takes one cycle each.**

**(1) Fetch** :

Address 14 in the instruction memory is fetched. In the fetch state, the PC value is "0x20070005". This is fed into the instruction memory which fetches the instruction.

**(2) Decode** :

The 32-bit address is then decoded. Instruction [31:26] is fed into the CL which produces the control signals for variables like ALU opcode, branch, write back. The source register $7 is decoded and the value is then stored into the ID/EXE pipeline register. **Aluout**, will have an output of 5. The value stored into register $0 is also passed into the pipeline register along with the control signals that were generated.

**(3) Execute**:

The alusrc is set to 1 indicating that the immediate value must be taken instead of the value from the register. The ALU adds the value of the content of register 0 and the immediate value 1. This contains the address in the data memory where the content of the register $7 needs to be stored. The value is passed to the Execute/Memory state. The previous values from the ID/EXE register is continued to pass into this pipeline registers.

**(4) Memory**:

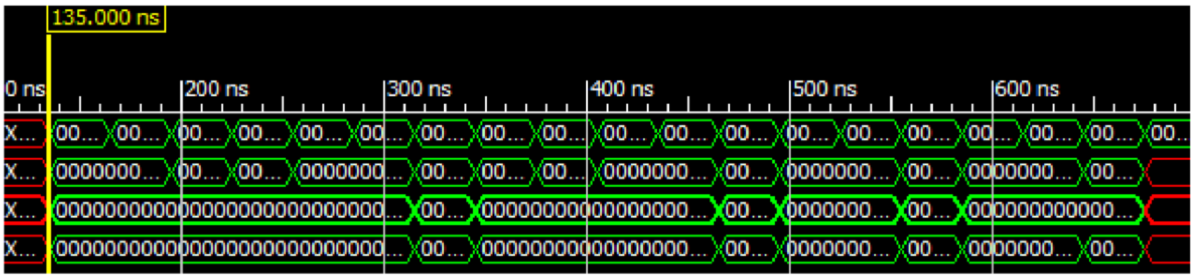
The value from **rdata2_EXE_MEM** (136) will be written to memory location 0x00000005 (**aluout**). Thus, the required result is successfully stored there eventually.

**(5) Write Back**:

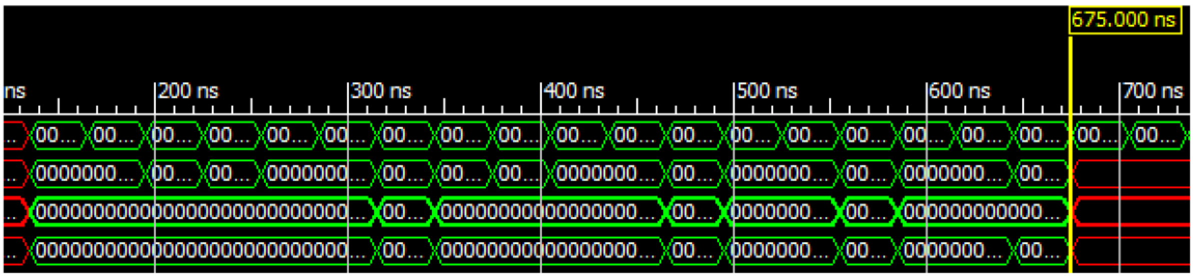
As the instruction is a SW instruction, there will be no Writeback stage but due to the specifications of the 5-stage pipelined architecture used

**e) Indicate the execution time for running this program along with a snapshot of starting and ending time of the code in the ISIM simulator.**

**START TIME**



**END TIME**



$Execution\ Time = Clock\ period \times CPI \times Number\ of\ instructions = 30ns \times 1 \times 17 = 510ns$

**f) Calculate the steady state CPI of the code while running in a five stage pipelined architecture.**

$Steady\ State\ CPI =$ (Number of instructions + Number of Stalls) / Number of Instructions

$= 8+(9-1)8 = 2$