

CE/CZ3001

LAB: 4

In lab 4, you will modify the 4-stage pipelined processor of Lab-3 to include LW, SW, BEQ and J instructions and convert that to a 5-stage pipelined processor. In this lab you will understand the working of the 5-stage pipelined datapath and simulate the LW, SW, BEQ and J instructions along with R-type and I-type instructions like ADDI. You will be provided with Verilog code for the datapath. You need to understand the functionality of the 5-stage CPU using the test bench. Meanwhile, due to dependencies, there are two kinds of hazards which may lead to wrong results using pipelining: data hazard and control hazard. You need to understand these two hazards during this lab.

I. FIVE STAGE PIPELINED IMPLEMENTATION OF R TYPE ALONG WITH ADDI, LW AND SW FOR 32 BIT CPU—UNDERSTANDING DATA HAZARD

In order to further understand pipelining better you are asked to implement a five-stage pipelined CPU as according to the Fig. 1.

We are having separate instruction memory and data memory. The data memory has 32-bit address port, a 32-bit input data port, a 32-bit output data port, and an active-high write-enable 'MemWrite' signal. If the 'MemWrite' signal is high, the memory will write the input data bits to the specified address. The read operation is possible anytime for both instruction and data memories; once address is provided at its input port. The Verilog module for the data memory is given to you – it is identical to the instruction memory: you should instantiate one copy with 'fileid = 0' for the instruction memory and another copy with 'fileid = 1' for the data memory. The data is preloaded to the data memory. The data are loaded to data memory with the help of a text file by the name "dmem_test0.txt". Each data is written as per the 32 bit format explained in lab 2. Note that memory is clocked and hence we have a clock cycle delay for getting the output from memory.

In the first stage, program counter 'PC.v' provides address to instruction memory. In the second stage, the instruction is read and the operand addresses are connected directly to the address ports of RF and the control logic. The instruction is decoded to generate the control signals and to get the data from the RF. The third stage comprises of execution using 'ALU.v'. In the fourth stage we have the data memory read and write using necessary control signals and final stage being written back. The ALU generates both data and addresses. For R-type and ADDI instructions, the ALU calculates the data. For LW and SW instructions, the addresses calculated by ALU are used to read or write data to data memory. Finally in the writeback stage, the ALU output or the data memory output is chosen with the help of control signal and is written back to the register file.

Note that both IMEM and DMEM are word addressable and hence the next data or instruction is taken by adding 1 with the current address. Both IMEM and DMEM provide the output after one clock cycle. This provides an imaginative pipeline after instruction fetch. It is also the reason why the output of DMEM is not passed through the last pipeline stage (indicated in Fig.1).

The Verilog file 'pipelined_regfile_5stage.v' (provided) is the top module of the five-stage pipelined CPU. This top module consists of 'PC.v', 'regfile.v', 'control.v', 'memory.v', 'alu.v', 'ID_EXEstage.v',

'EXE_MEMstage.v', and 'MEM_WBstage.v'. The 'ID_EXstage.v', 'EXE_MEMstage.v', and 'MEM_WBstage.v' are the pipeline registers between decode /execute, execute/datamemory and datamemory/writeback section respectively.

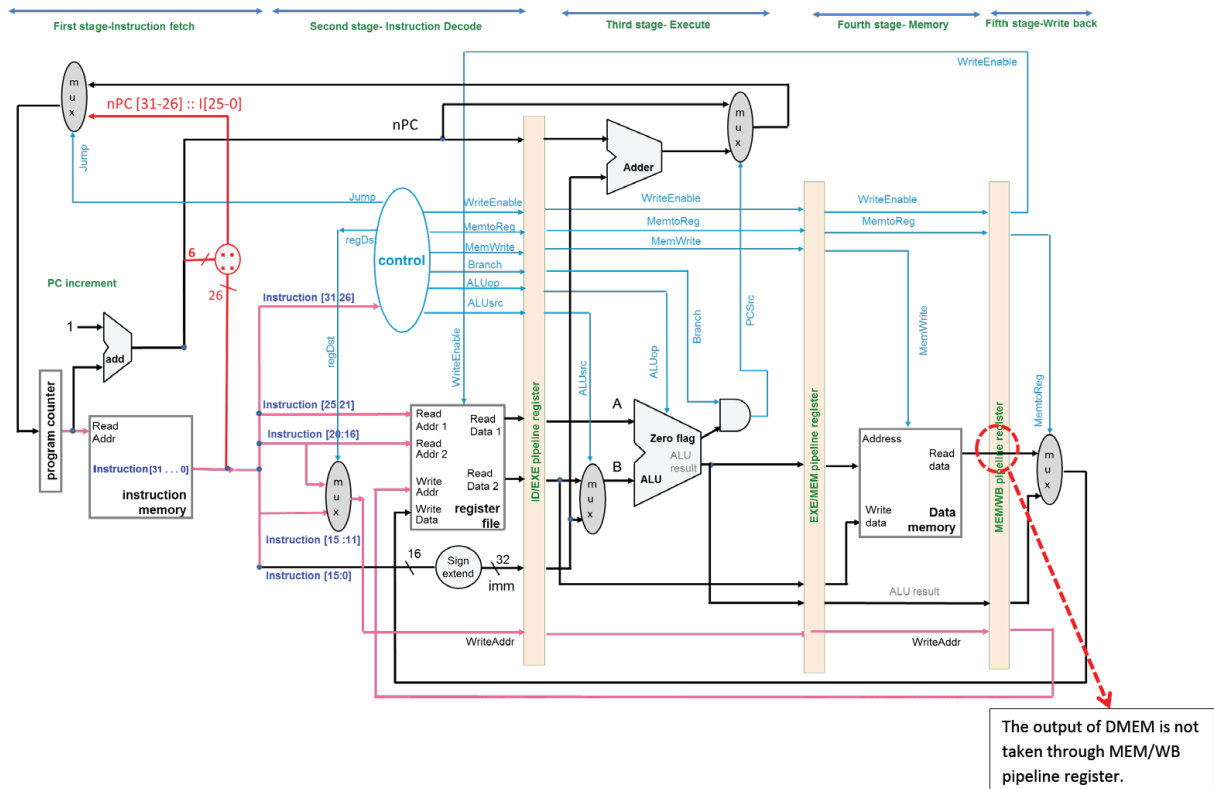


Fig. 1 Five-stage Pipelined CPU implementation diagram

You can note that we are able to operate register type of instructions (ADD, SUB, AND, XOR, COM and MUL) and ADDI operation along with LW and SW. In order to initialize the registers in the register file, data from the memory can be utilized.

1. We can note that 'write enable' of register file is now no longer a constant as SW do not write back to register. Hence 'write enable' signal is a variable and is now pipelined through all stages.
2. Double click RTL schematic to view the circuit diagram for the five-stage pipelined implementation. The RTL schematic should closely resemble Fig. 1.
3. Test the 5-stage pipelined datapath using the given test bench 'pip5_data_test.v'.

Note that there are two different instruction sets (two files: imem_testdw.txt and imem_testdr.txt). When you run the simulation, you can notice that if you use the 'imem_testdw.txt', you will get the wrong result shown in Fig. 2.

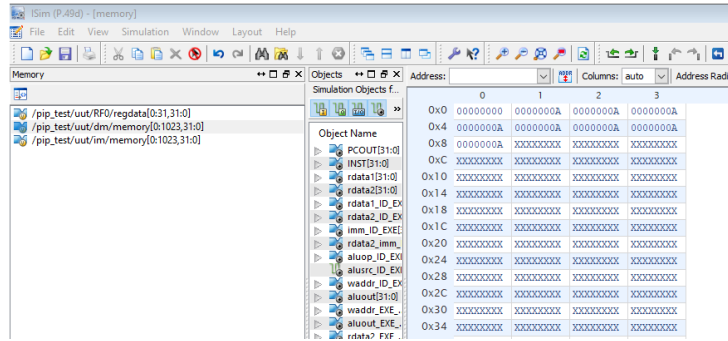


Fig. 2 DMEM register after executing the instructions in 'imem_testdw.txt'

However, if you use the 'imem_testdr.txt', you will get the correct result as shown in Fig. 3.

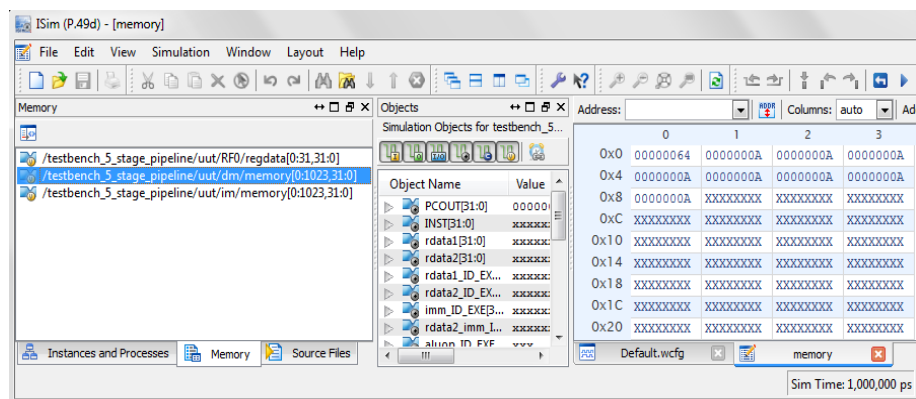


Fig. 3 DMEM register after executing the instructions in 'imem_testdr.txt'

This situation is called data hazard. The set of instructions in 'imem_testdw.txt' shown in Fig. 4 has data dependencies. The set of instructions loads data from memory location [0] and [1] and multiplies them together. And all the DMEM data we initialize to 0x0000000A. The result is stored back into address location "[0]" of DMEM. In order to get the correct result while running in a five stage pipelined architecture, we need to add NOPs where ever necessary. The 'imem_testdr.txt' is the file after adding NOPs. Using SW instruction, we can write back to DMEM register. Note that we are not writing back to the text file 'dmem_txt0.txt'. Hence the update is only visible at the DMEM register as shown in the simulation window represented in Fig. 2 and Fig. 3.

32 bit PC address (in hex)	32 bit instruction from IMEM	meaning
00000000	00000000	NOP
00000001	1C040000	LW \$4, 0(\$0)
00000002	1C050001	LW \$5, 1(\$0)
00000003	14855000	MUL \$10, \$4,\$5
00000004	200A0000	SW \$10, 0(\$0)

Fig. 4 Instructions in imem_testdw.txt

\$10 has the result of $[A * A] = [64]$ (in hexadecimal). It is written to location 0 as shown in Fig. 3.

As the instructions are already in IMEM (imem_testdw.txt or imem_testdr.txt) and data in DMEM (dmem_test0.txt), we can simulate and see the functionality of the datapath. The IMEM uses file operation to get the instruction and it reads “imem_testdw.txt” or “imem_testdr.txt” as per the address given by PC. The DMEM reads from “dmem_txt0.txt” as per the address given by ALU for LW. For SW, the data from the register file (RF) is written to the DMEM using the address generated by ALU. You can plot the flow of the pipelined instructions to have a better understanding of data hazard.

4. You can add new instructions and data by modifying the text files for both IMEM and DMEM.

II. FIVE STAGE PIPELINED IMPLEMENTATION OF R, I AND J TYPE INSTRUCTION FORMAT FOR 32 BIT CPU—UNDERSTANDING CONTROL HAZARD

In this part, we are adding both “branch when equal (BEQ)” and “Jump (J)” instructions to the data path along with the previous set of instructions covered in Part I based on the five-stage pipelined CPU shown in Fig. 1. As both the instructions change the ordering of instructions by changing the content of Program Counter (PC), we need to be careful about the control hazards introduced by these instructions.

I format BEQ instruction format: (Note the opcode used as per define.v= 6'b001001)

Meaning: meaning: [\$rt] == [\$rs], then branch to the target location with respect to program counter value. Compare the content of register \$rs with the content of register \$rt in the ALU stage and if they are same (ALU zero flag=1), PC is updated to a new location whose address is “PC+2+ sign-extended(immediate)”, called as branch target address. Note that PC is updated in “execute” cycle. Thus, we add 2 with the original PC. Also note that BEQ uses PC relative addressing. The BEQ instruction has a two address (Rs and Rt acts as both sources) and one immediate format. The bit assignments to different fields of I-format are shown below.

opcode	Rs	Rt	imm	
31	26 25	21 20	16 15	0

Example:

1. BEQ \$5, \$4, 3 (meaning: [\$5] == [\$4], then branch to the location PC + 2 + 3. Here ‘3’ is an immediate value. The machine format is 24A40003_H(given below is binary format).

001001	00101	00100	0000 0000 0000 0011	
31	26 25	21 20	16 15	0

Do note that the comparisons of these registers (\$5 and \$4) are done in the EXE cycle and the PC counter is also updated in the same EXE cycle as per Fig. 1 This indicates that there will be a penalty of two cycles after each BEQ for the BEQ to take a decision. (In class notes the PC was updated in MEM stage and hence penalty was indicated to be three clock cycles, whereas here in lab, PC is updated in the EXE cycle to reduce the penalty to two cycles). Hence to go for a conservative way to remove control hazard, we need to insert two NOPs after every BEQ instruction.

J format instruction format: (Note the opcode used as per define.v= 6'b001010)

Meaning: meaning: J offset (Jump to the offset address and address is “nPC[31:26]::offset[25:0]”: The offset is 26 bits and it is appended with the first 6 bits of PC. Please note that the data and instruction memory used in lab follows word addressing. Hence there is no need to convert to byte addressing mode by doing left shift by 2. Jump uses pseudo direct addressing mode. The bit assignments to different fields of I-format are shown below.

opcode	Offset
31	26 25 0

Example:

1. J 3 (meaning: jump to the offset 3 and the address to be inputted to the program counter is calculated as “ nPC[31:26]:: 3”. nPC is the next PC value, which is PC+1. Here ‘3’ is an offset value. The machine format is 28000003_H (given below is binary format).

001010	00 0000 0000 0000 0000 0000 0011
31	26 25 0

Do note that the program counter (PC) is updated in the decode cycle of jump instruction as per Fig. 5. The control signal is generated in the decode cycle for the multiplexer to select the output to program counter. This indicates that there will be a penalty of one cycle after each J for the J to take a decision. Hence to go for a conservative way to remove the control hazard, we need to insert one NOP after every J instruction

Details of the code given

In the first stage, program counter ‘PC.v’ provides address to instruction memory. In the second stage, the instruction is read, control signals are generated. For the jump instruction, decision takes place in this decode cycle. For Jump instruction, PC counter will be updated in the decode cycle according to the offset value. For BEQ, the instruction is decoded to generate the control signals and to get the data from the RF. The third stage comprises of comparison of the content of registers using ‘ALU.v’. Please note that a zero flag has been inserted to ALU such that whenever the output of ALU=0, then the ‘zero_flag=1’. The zero flag result and the branch control signal are ANDed to generate the derived control signal ‘PCSrc’. ‘PCSrc’ selects between the branch target address and nPC to update the program counter. The program counter is updated in execute cycle to reduce the branch penalty cycles. The rest of the instructions works in the similar manner as explained in Part I.

Analysis:

1. Double click RTL schematic to view the circuit diagram for the five-stage pipelined implementation for R, I and J type instructions. The RTL schematic should closely resemble Fig. 1.
2. Test the 5-stage pipelined datapath implementation by generating a test bench. Test bench can be generated in the ‘simulation mode’ by right clicking the top module and adding ‘New source’ to be ‘Verilog test fixture’. Name the Verilog test bench and choose the corresponding top module to generate the test bench. Once the test bench is generated, you need to check the bit-width of the inputs and outputs, add the clock and the test inputs as shown below.

Note: You need to change the simulation run time. In simulation mode, click Process → Process Properties and change “Simulation Run Time” to 4000 ns.

3. Inserting CLK signal: Insert ‘always #15 clk = ~clk;’ before the ‘initial’ statement.
4. Add the following test vectors on the portion “//Add the stimulus”

```
#25 rst=1;
#25 rst=0;
```

As the instructions are already in IMEM (imem_testcw.txt) and data in DMEM (dmem_txt1.txt), we can simulate and see the functionality of the datapath.

5. You can add new instructions and data by modifying the text files for both IMEM and DMEM.
6. Analyse how PC counter is updated during the execution of a BEQ and J instruction? You can add BEQ and J instructions in ‘imem_txt0.txt’ to see the number of branch penalty cycles inserted to remove hazards for both instructions.

Example for verification of BEQ and J

1. Given the MIPS code (in imem_testcw.txt) of the following code to be executed in the given 5 stage pipelined datapath in Fig. 1.

```
for (i=0, i<=3, i++){
    A[i]= A[i]+5;
}
```

You can use <http://www.ntu.edu.sg/home/smitha/OPCoder/OPCoder/converter.html> to convert the MIPS instructions to machine code in binary or hexadecimal number system. Please do note to make corresponding changes to your opcode part according to the opcode that you have used in ‘define.v’ file.

32 bit PC address (in hex)	32 bit instruction from IMEM	instructions	Meaning
00000000	00000000	NOP	
00000001	18010004	ADDI \$1, \$0, 4	Termination index=4
00000002	1C640000	LW \$4, 0(\$3)	Load first value from DMEM
00000003	18840005	ADDI \$4, \$4, 5	Add 5 to that
00000004	20640000	SW \$4, 0(\$3)	Store to same location
00000005	18630001	ADDI \$3, \$3, 1	Increment address to get next data
00000006	18A50001	ADDI \$5, \$5, 1	Increment pointer
00000007	2425000X (the value for x is where you want to branch. It should be outside the loop to exit out from the loop)	BEQ \$1, \$5, exit	When termination index=pointer, exit
00000006	28000002	J 2	Jump to PC address 2, to load the next value.

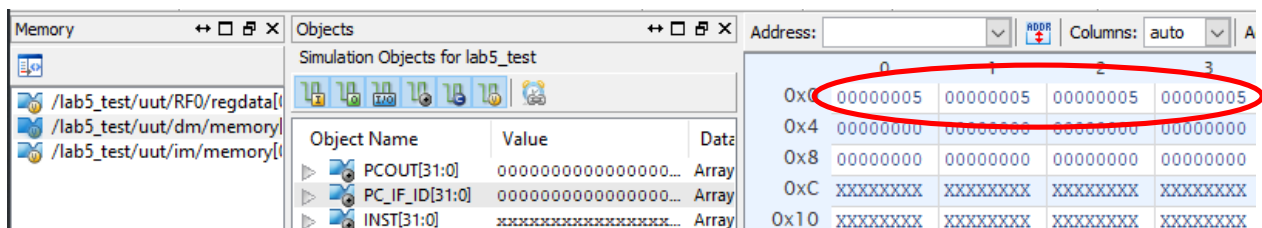
Fig. 5 Instructions in the IMEM

- In order to verify the operation, the MIPS instructions for the above pseudocode are given in Fig. 5 and data values in Fig. 6. Note that Array A starts from memory address location [0] of DMEM. Please note that all registers are initialized to zero during rest. Hence there is no need to reinitialize the loop pointer.
- The set of instructions given in Fig. 5 has data and control dependencies. In order to get the correct result while running in a five stage pipelined architecture; we need to add NOPs where ever necessary to remove hazards. Once the NOPS are correctly inserted to remove the data dependencies, we will get the updated result in DMEM as can be seen in Fig. 7

32 bit address (in hex)	32 bit data from DMEM
00000000	00000000
00000001	00000000
00000002	00000000
00000003	00000000
00000004	00000000
00000005	00000000
00000006	00000000
00000007	00000000
00000008	00000000
00000009	00000000
0000000A	00000000
0000000B	00000000

Fig. 6 Data in the DMEM

- You can note that writing back to DMEM register is done using SW. Note that we are not writing back to the text file. Hence the update is only visible at the DMEM register as shown below. If all dependencies are removed then you can get the result as shown in Fig. 7.



Address	0	1	2	3
0x0	00000005	00000005	00000005	00000005
0x4	00000005	00000005	00000005	00000005
0x8	00000000	00000000	00000000	00000000
0xC	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x10	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

Fig. 7 DMEM register after executing the modified instructions in Fig. 5 for data in Fig. 6

- If NOPs are not correctly inserted the result written back to DMEM will be wrong.

EVALUATION

- Find the execution time of the instructions given in Fig. 5 by adding NOPs to remove both data and control dependencies.
- Do loop unrolling by 4 (maximal loop unrolling) and reordering and find the total execution time. The results of both the programs should be the same as indicated in Fig. 7.