SC3050 Advanced Computer Architecture

LAB-4

In lab 4, you will modify the 5-stage pipelined processor of Lab-3 to include CBZ and B instructions. In this lab you will understand the working of the 5-stage pipelined datapath and simulate the CBZ and B instructions along with the previous R-type and D-type instructions. You will be provided with Verilog code for the datapath. CBZ and B instructions changes the flow of the program thus introducing control hazards. You need to understand the effect of control hazard and eliminate them for the correct result in pipeline.

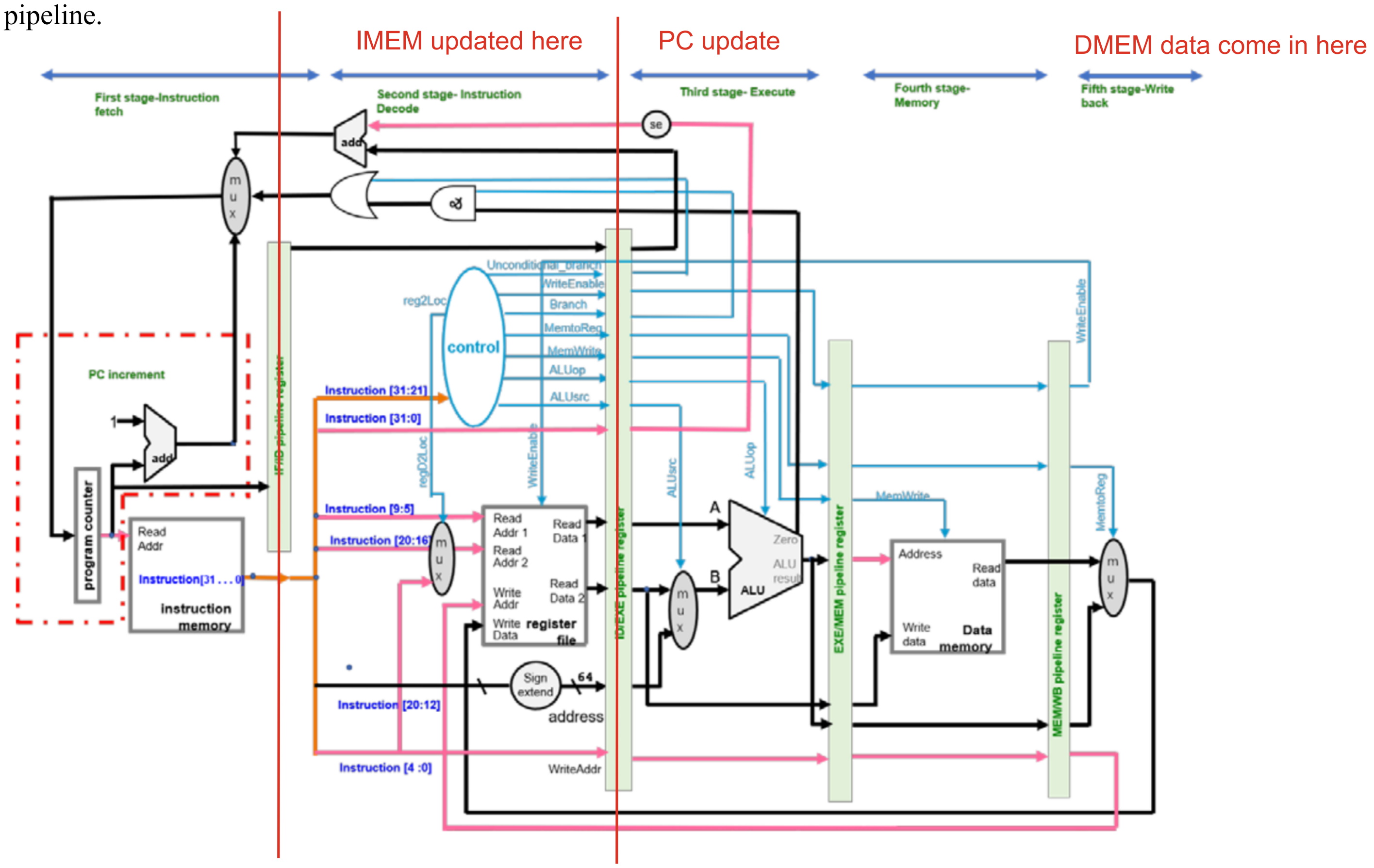


Fig. 1 Five-stage pipelined CPU implementation diagram for R, D, CB and B-type instructions.

Note PC is passed into pipeline register

FIVE STAGE PIPELINED IMPLEMENTATION OF R, D, CB AND B TYPE INSTRUCTION FORMAT FOR 64 BIT CPU—UNDERSTANDING CONTROL HAZARD

In this part, we are adding both "CBZ" and "B" instructions to the data path along with the previous set of instructions covered in previous labs 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.

CB format CBZ instruction format: (Note the opcode used as per define.v= 8'b00000111)
Meaning: [Rt] == 0, then branch to the target location with respect to program counter value.

Compare the content of register Rt with zero in the ALU stage and if they are same (ALU zero flag=1), PC is updated to a new location whose address is "PC+ sign-extended(Address)", called as branch target address. Note that PC is the instruction's PC. Also note that CBZ uses PC relative addressing. The bit assignments to different fields of CB-format are shown below.

opcode	address	Rt
31 24	23 5	4 0

Example:

1. CBZ X5, #3 ([X5] == 0, then branch to the location PC + (3)). Here "3" is an immediate value. Note that here we are not multiplying the immediate address by 4 as the instruction memory is not byte addressing. The machine format is 07000065H (given below is binary format).

00000111	0000000000000000011	00101
31 24	23 5	4 0

Do note that the comparison of the value (X5 == 0) is 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 CBZ for the CBZ to take a decision. Hence to go for a conservative way to remove control hazard, we need to insert two NOPs after every CBZ instruction to remove the control hazard.

what other ways are there other than conservative way?
Branch prediction -> guess if branch taken. Static and dynamic prediction

B format instruction format: (Note the opcode used as per define.v = 6'b001000)
Meaning: B offset (Unconditional Branch to the target address which is calculated by "PC + sign-extended (Address[25:0])"). Please note that the PC is the instruction's PC. Unconditional branch also uses PC relative addressing mode. The bit assignments to different fields of B-format are shown below.

opcode	address
31 26	25 0

Example:

1. B #3 (branch to the immediate address 3 and the address to be inputted to the program counter is calculated as "PC+(3)"). Here "3" is an address. Note that here we are not multiplying the immediate address by 4 as the instruction memory is not byte addressing. The machine format is 20000003H (given below is binary format).

001000	00 0000 0000 0000 0000 0000 0011
31 26	25 0

Do note that the program counter (PC) is the instruction's PC of jump instruction, and the targeting address will be updated to program counter in the EXE stage as per Fig. 1. This indicates that there will be a penalty of two cycles after each B for the B to take a decision, which is the same as CBZ. Hence to go for a conservative way to remove the control hazard, we need to insert two NOPs after every B instruction to remove the control hazard.

Details of the code given

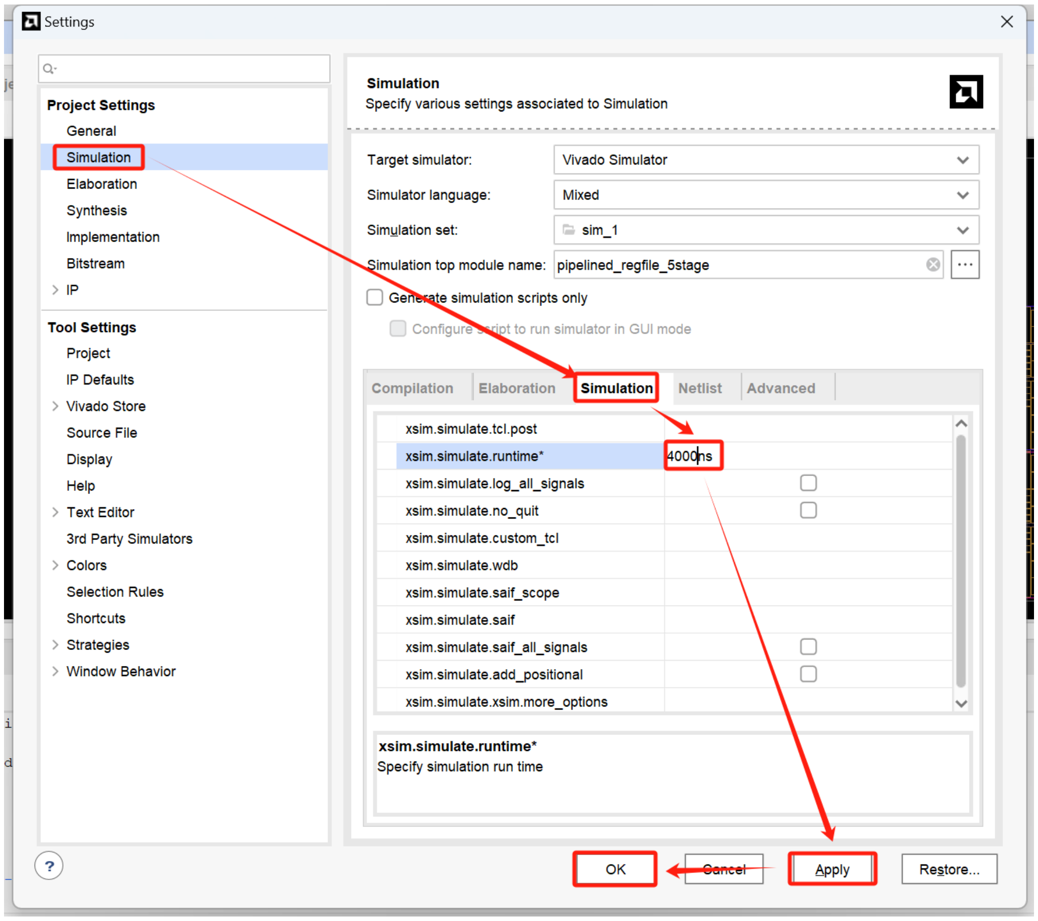
In the first stage, program counter "PC.v" provides address to instruction memory. Note that to calculate the targeting address, CBZ and B both require its own PC. Thus, we use a pipeline "IF_IDstage.v" to store current PC and pass it to the following stages. In the second stage, the instruction is read, control signals are generated. For both CBZ and B instructions, the instruction is decoded to generate the control signals and to get the data from the RF (B instruction don't need 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 (PC+1) to update the program counter. The program counter is updated in execute stage and we need to add two NOPs to reduce the branch penalty cycles. The rest of the instructions works in the similar manner as explained in previous labs.

Analysis:

1. Double click RTL schematic to view the circuit diagram for the five-stage pipelined implementation for CB and B type instructions. The RTL schematic should closely resemble Fig 1.

2. Test the 5-stage pipelined datapath implementation by using the test bench given in 'test_bench_5_stage_pipeline.v'. Add the clock and the test inputs as shown below.

   Note: You need to change the simulation run time.
   Flow Navigator > Project Manager > Settings > Change 'xsim.simulate.runtime' to 4000ns > Apply > Ok



Settings

Simulation
Specify various settings associated to Simulation

Target simulator: Vivado Simulator
Simulator language: Mixed
Simulation set: sim_1
Simulation top module name: pipelined_regfile_5stage
Generate simulation scripts only
Configure script to run simulator in GUI mode

xsim.simulate.tcl.post	
xsim.simulate.runtime*	4000ns
xsim.simulate.log_all_signals	☐
xsim.simulate.no_quit	☐
xsim.simulate.custom_tcl	
xsim.simulate.wdb	
xsim.simulate.saif_scope	
xsim.simulate.saif	
xsim.simulate.saif_all_signals	☐
xsim.simulate.add_positional	☐
xsim.simulate.xsim.more_options	

xsim.simulate.runtime*
Specify simulation run time

3. Inserting CLK signal: Insert "always #15 clk = ~clk;" before the "initial" statement.

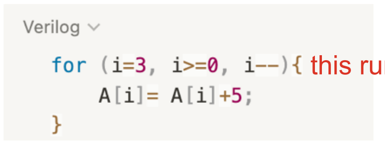
4. Add the following test vectors on the portion "//Add the stimulus"
   #25 rst = 1;
   #25 rst = 0;
   As there are some random instructions are already in IMEM (imem_test0.mem) and random data in DMEM (dmem_test0.mem), we can simulate and see the functionality of the datapath.

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

6. Analyze how PC counter is updated during the execution of a CBZ and B instruction? How many NOPs are inserted to remove hazards for both instructions?

Example for verification of CBZ and B

1. Convert the following pseudo code to LEGv8 instructions, to be executed in the given 5 stage pipelined datapath in Fig. 1.



Verilog

```
for (i=3, i>=0, i--){ this runs 4 times
    A[i]= A[i]+5;
}
```

You can use http://www.ntu.edu.sg/home/smitha/OPCoder/OPCoder/converter.html to convert the ARM instructions to machine code in binary or hexadecimal number system. Please do note to make corresponding changes to your opcode part of the web link (if needed) according to the opcode that you have used in "define.v" file.

For your easiness to convert to LEGv8 code, we have provided the instructions and its hexadecimal conversions in Fig.2 and also given the same as "imem_test_ex.mem".

64 bit PC address (in hex)	32 bit instruction from IMEM	instructions	Meaning
0000000000000000	001F03FF	NOP	
0000000000000001	00A06021	LDUR X1, [X1, #6]	Load terminat. index "4"
0000000000000002	00A07042	LDUR X2, [X2, #7]	Load a const value "1"
0000000000000003	00A08063	LDUR X3, [X3, #8]	Load a const value "5"
0000000000000004	00A00024	LDUR X4, [X1, #0]	Load A[i]
0000000000000005	00030084	ADD X4, X4, X3	Add A[i] and "5"
0000000000000006	00C00024	STUR X4, [X1, #0]	Store A[i]
0000000000000007	00220021	SUB X1, X1, X2	Subtract index by 1
0000000000000008	07000081(this instruction may be different due to where you want to branch. It should be outside the loop to exit out from the loop)	CBZ X1, #4	When termination index=0, exit
0000000000000009	23FFFFFB(this instruction may be different after you add NOPs)	B #-5	Branch to PC address 4, to load the next value.

Question: If i do not add NOP here, does the loop loop forever?

Fig. 2 Instructions in the IMEM

2. In order to verify the operation, the LEGv8 instructions for the above pseudocode are given in Fig. 2 and data values in Fig. 3 (given in "dmem_test_ex.mem"). Note that Array [A] starts from

memory address location [4] of DMEM. Please note that all registers are initialized to zero during rest. Hence there is no need to reinitialize the loop pointer.

3. Choose the input files as “imem_test_ex.mem” in imemory.v and “dmem_test_ex.mem” in dmemory.v for the simulation. Do note that the set of instructions given in Fig. 2 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 and control dependencies, we will get the updated result in DMEM as can be seen in Fig. 4.

64 bit address (in hex)	64 bit data from DMEM
0000000000000000	0000000000000000
0000000000000001	0000000000000000
0000000000000002	0000000000000000
0000000000000003	0000000000000000
0000000000000004	0000000000000000
0000000000000005	0000000000000000
0000000000000006	0000000000000004
0000000000000007	0000000000000001
0000000000000008	0000000000000005
0000000000000009	0000000000000000
000000000000000A	0000000000000000
000000000000000B	0000000000000000

Fig. 3 Data in the DMEM

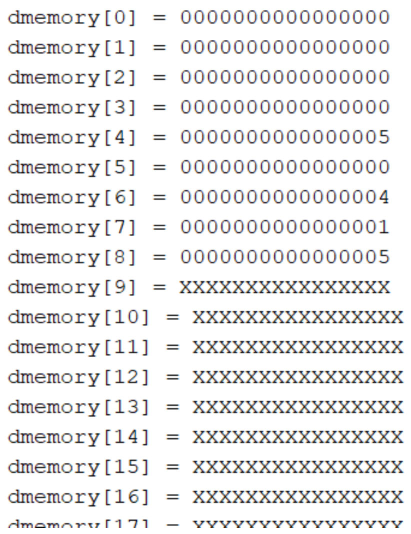
You can note that writing back to DMEM register is done using STUR. Note that we are not writing back to the memory file. Hence the update is only visible at the DMEM register as shown below. To see the content of dmemory data, you can use the Tcl Console to get access to its values by running the following commands **one-by-one**:

```
set NREG 64

set scope_path "/test_bench_5_stage_pipeline/uut/dm"

for {set i 0} {$i < $NREG} {incr i} {
    set dm_value [get_value ${scope_path}/dmemory[$i]]
    puts "dmemory[$i] = $dm_value"
}
```

If all dependencies are removed, you can get the result as shown in Fig. 4.



```
dmemory[0] = 0000000000000000
dmemory[1] = 0000000000000000
dmemory[2] = 0000000000000000
dmemory[3] = 0000000000000000
dmemory[4] = 0000000000000005
dmemory[5] = 0000000000000000
dmemory[6] = 0000000000000004
dmemory[7] = 0000000000000001
dmemory[8] = 0000000000000005
dmemory[9] = XXXXXXXXXXXXXXXX
dmemory[10] = XXXXXXXXXXXXXXXX
dmemory[11] = XXXXXXXXXXXXXXXX
dmemory[12] = XXXXXXXXXXXXXXXX
dmemory[13] = XXXXXXXXXXXXXXXX
dmemory[14] = XXXXXXXXXXXXXXXX
dmemory[15] = XXXXXXXXXXXXXXXX
dmemory[16] = XXXXXXXXXXXXXXXX
dmemory[17] = XXXXXXXXXXXXXXXX
```

Fig. 4 DMEM register after executing the modified instructions in Fig. 2 for data in Fig. 3

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

Evaluation

1) Find the execution time of the instructions given in Fig. 2 by adding NOPs to remove both data and control dependencies.

2) Do loop unrolling by 2 and 4 (maximal loop unrolling) and reordering, respectively, and find the total execution time. The results of both the programs should be the same as indicated in Fig. 4.

Lab Report for Lab 4 (Name:________________, Matric:_________, Group:_____)

EVALUATION

1) Find the execution time of the instructions given in Fig. 2 by adding NOPs to remove both data and control dependencies.

   Execution time: ________________

2) Do loop unrolling by 2 and 4 (maximal loop unrolling) and reordering, respectively, and find the total execution time. The results of both the programs should be the same as indicated in Fig. 4.

   Execution time (by a factor of 2): ________________

   Execution time (by a factor of 4): ________________

```
`define ADD 11'b00000000000
`define SUB 11'b00000000001
`define AND 11'b00000000010
`define XOR 11'b00000000011
`define ORR 11'b00000000100
`define LDUR 11'b00000000101
`define STUR 11'b00000000110
`define CBZ 8'b00000111
`define B 6'b001000
```

data hazard: RAW dependency -> 2NOP
Control: CBZ/B -> 2NOP