

Objective:

The objective of this project is to design and build a pipelined RISC processor in VHDL.

Upon completion, the student must be able to:

- Design, realize and test a pipelined processor;
- Demonstrate an understanding for pipelining concepts, including hazards, in the context of RISC processors and instruction set architectures.

Pre laboratory:

1. Explain why we need pipeline registers? And how many instructions will be executed simultaneously during any clock cycle? (1 point)
2. Explain why we only need four pipeline registers in this design (not five, one for each stage)? (1 point)
3. Explain why we need to flush the pipeline, and when and how it is done (1 point).
4. What are the advantages and disadvantages of using multiple-clock-cycle pipeline diagrams and single-clock-cycle pipeline diagrams? (1 point)
5. Explain why no write signals are needed for the PC and pipeline registers? (1 point)
6. Explain at what stage control information is created, and how they are passed to their appropriate datapath stages? (1 point)
7. What happens when a register is read and written in the same clock cycle? How can you avoid this problem? (1 point)
8. Why do we need to check RegWrite before forwarding any data? (1 point)
9. Explain why we should be able to take inputs to the ALU from any pipeline register rather than just ID/EX? (1 point)
10. Explain why “Assume branch not taken” optimization technique, can half the cost of control hazards? (1 point)

Because of data dependencies between instructions in the pipeline and branch operations, some additional complications arise that can be solved using hazard detection, data forwarding, and branch flushing techniques.

See Sections 6.2 to 6.6 of your course textbook for more background information on pipelining.

Laboratory:

In this lab, you will proceed to implement a pipelined processor, with the datapath shown in figure 1. The input/output specification of the processor is shown in table 1. We will be again working with 8-bit datapaths and 32-bit instruction widths, while the control lines obviously remains constant. Also note that the 8-bit datapaths will only allow for 8-bit number representations and operations. We will be implementing a register file of eight 8-bit registers. Instruction memory will remain at 32-bits, and will contain a maximum of 256 instructions. The data memory, however, will be 8-bits, and will contain a maximum of 256 words.

The MuxOut[7..0] and ValueSelect[2..0] output and input are intertwined in the manner shown in table 2. The InstructionOut[31..0] and InstrSelect[2..0] output and inputs are intertwined in the manner shown in table 3, where any of the instructions being executed on the pipeline (i.e. InstructionOut1 to InstructionOut5) can be viewed.

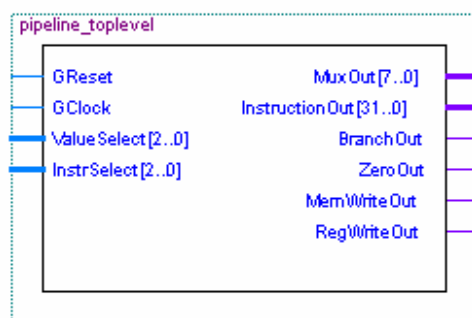


Figure3 – Pipeline top level entity

<i>Port Type</i>	<i>Name</i>	<i>Description</i>
Input	GClock	Global clock needed to synchronize the circuitry
Input	GReset	Global reset needed to bring the internals to known states
Input	ValueSelect[2..0]	Selector for MuxOut[7..0]
Input	InstrSelect[2..0]	Selector for InstructionOut[31..0]
Output	MuxOut[7..0]	Multiplexer output controlled by ValueSelect[2..0]
Output	InstructionOut[31..0]	The current instruction being executed
Output	BranchOut	The branch control signal of each processor
Output	ZeroOut	The zero status signal of each processor
Output	MemWriteOut	The memory write control signal of each processor
Output	RegWriteOut	The register write control signal of each processor

Table 1: Input/Output Specification

<i>ValueSelect[2..0]</i>	<i>MuxOut[7..0]</i>	<i>Description</i>
000	PC[7..0]	The program counter value
001	ALUResult[7..0]	The result of the current ALU operation
010	ReadData1[7..0]	The read data 1 port of the register file
011	ReadData2[7..0]	The read data 2 port of the register file
100	WriteData[7..0]	The write data port of the register file
Other	['0', RegDst, Jump, MemRead, MemtoReg, AluOp[1..0], AluSrc]	The remaining control information

Table 2: Output Multiplexer Selection

<i>InstrSelect[2..0]</i>	<i>InstructionOut[31..0]</i>
000	InstructionOut1[31..0]
001	InstructionOut2[31..0]
010	InstructionOut3[31..0]
011	InstructionOut4[31..0]
100	InstructionOut5[31..0]
Other	Zeros[31..0]

Table 3: Instruction Multiplexer Selection

Part 1) Pipelined Datapath and Control(6.2-6.3)

Pipeline the MIPS (Lab 2) design. Test your design by running a simulation of the example program shown below. (For initializing you might need to add several NOP instructions after their corresponding lw instruction to avoid data hazard at this stage of your design. Use sw instruction at the end to store your results in memory avoiding data hazards)

```

...
sub   $3, $1, $2    -> t3 = BB - 22 = 99
or    $4, $2, $1    -> t4 = 22 or BB = BB
add   $5, $1, $2    -> t5 = BB + 22 = DD
and   $6, $2, $1    -> t6 = 22 and BB = 22
sub   $7, $2, $2    -> t7 = 22 - 22 = 00
add   $2, $2, $3    -> t2 = 99 + 22 = BB
sub   $1, $3, $2    -> t1 = 99 - 22 = 77
...

```

To modify your design from lab #2 design, simply create a new “StageName_top” module for each stage and include your old modules and their corresponding pipeline registers in them as shown in Figure 2. This way you minimise changes to your old modules from lab #2.

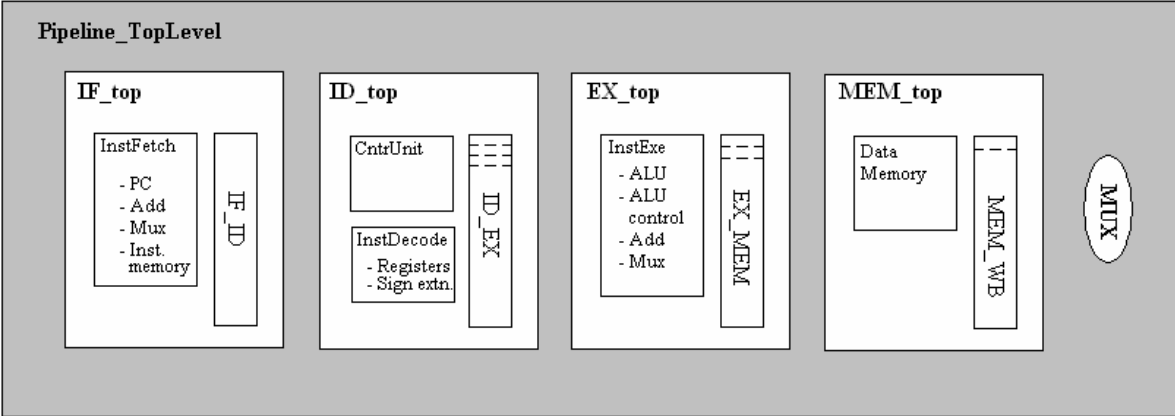


Figure 2 – Pipeline top level module

Add “d_” in front of datapath signal name to indicate it is the input to a pipeline register. Signals that go through two pipeline registers would be “dd_” and three would be “ddd_”. Add “e_”, “m_”, or “w_” in front of the control signal name to indicate it is the input to a pipeline register and the destination stage that it is heading. For example a WB stage control signals that goes through two pipeline registers would be “ww_” and three would be “www_”. Block diagram of IF/ID pipeline register is shown in figure 3. The MIPS instruction ADD \$0, \$0, \$0 is all zeros and does not modify any values in registers or memory. It is used to initialize the IF/ID pipeline at reset. For this lab you implement only the instructions listed in Figure 6.23 of the textbook.

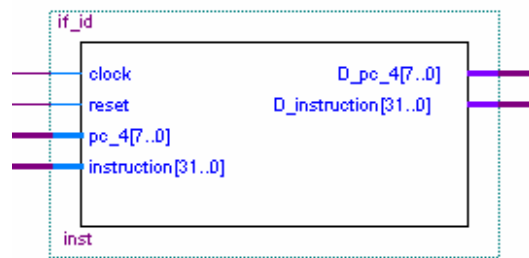


Figure 3 – IF/ID pipeline register module

Part 2) Data Forwarding (6.4)

Once the MIPS is pipelined, data hazards can occur between the instructions present in the pipeline. This problem can be fixed by adding two forwarding multiplexers to each ALU input in the execute stage. In addition to the existing values feeding in the two ALU inputs, the forwarding multiplexers can also select the last ALU result or the last value in the data memory stage.

These multiplexers are controlled by comparing the rd, rt, and rs register address fields of instructions in the decode, execute, or data memory stages. Instruction rd fields will need to be added to the pipelines in execute, data memory, and write-back stages for the forwarding compare operations. Since register 0 is always zero, do not forward register 0 values. The forwarding multiplexers in the EX stage handle dependences on instructions in the EX and MEM stages.

For Part 3 of the lab with Branches you will need to have the ALU's data forwarding multiplexers moved back into the decode stage, so you might save time by moving the forwarding multiplexers to the decode stage for Part 2 of the lab. Note that the multiplexer signals to forward are now needed one clock cycle earlier, so you will need to rework the textbook's forwarding equations to reflect this change before coding them in VHDL. It would be a good idea to work through several examples by hand to double check your new forwarding equations. If necessary, draw a picture like Fig 6.28 to obtain the new forwarding equations. You can assume

that there is enough time in a single clock cycle to take the current ALU output (before a pipeline register) and forward it to the decode stage for a branch decision.

Dependence from an instruction in the WB stage to one in the ID stage must also be considered. You can handle this dependence in one of 2 ways. The first method is to ensure that the register file write in one cycle occurs before the register file read in the same cycle. This can be accomplished by writing to the register file on the falling edge (instead of the rising) of each clock cycle. A register file read from the same register will then have its value clocked into the ID/EX pipeline register at the next rising edge, ensuring that the just-written value is passed to the EX stage. The second method to handle this type of dependence is to add two forwarding multiplexers to the InstDecode module so that the register file is bypassed in this situation. If the register file write address equals one of the two read addresses, the register file write data value should be forwarded to the ID/EX pipeline register instead of the normal register file data value. Since you may already have the data forwarding multiplexers in the decode stage for Part 3, you may find this option easier.

Add forwarding to your pipelined datapath from Lab 2 and test it with the following program:

```
...
sub  $1, $2, $3    -> t1 = BB - 66 = 55
or   $4, $1, $2    -> t4 = 55 or BB = FF
add  $6, $5, $1    -> t6 = 22 + 55 = 77
and  $7, $1, $4    -> t7 = 55 and FF = 55
sub  $2, $4, $4    -> t2 = FF - FF = 00
...
```

NOTE: Remember that data forwarding can not solve all the problems when you are dealing with the load instruction, because the data is still being read in clock cycle 4 while the next instructions might need that data. To avoid this problem you can either implement a Hazard detection unit (section 6.5 of textbook) to automatically insert stalls after load instructions if a hazard has been detected for 10% bonus mark. Or you can simply avoid the problem by inserting appropriate number of NOPs manually in your programs.

Part 3) Branch hazards (6.6)

In this part of the lab, you will implement a hazard unit that will flush the pipeline to handle branch hazards. Assume branch not taken, and move the branch decision hardware forward to the decode stage as shown in Figure 6.38 of textbook. Also, be aware that when you implement the textbook's Figure 6.38 solution for early branch condition evaluation, your forwarding multiplexers must be placed in the ID stage before the comparison unit rather than the EX stage, so that the new branch comparison unit can use forwarded values for branch instructions that are dependent on preceding instructions. In addition to the new branch compare circuit that must be added to decode, the branch address adder must be moved from execute to decode. With these changes, only 1 instruction following a branch taken must be flushed. When a taken branch is detected, ONLY the IF/ID register should be reset to all 0's (i.e. a NOP) to flush the instruction that was fetched incorrectly. (Figure 6.38)

Add this branch flush hardware to your pipelined datapath and test it with the following program:

```
    . . .
    Beq  $0,$4,label1
    Add  $3,$2,$4
    Add  $5,$5,$1
label1: Beq  $3,$5,label2
        Sub $5,$5,$4
label2: Beq  $5,$6,label2
label3: Beq  $0,$0,label3
    . . .
```

Part 4) Enhanced performance

Compare your pipelined design with your single cycle (lab #2) design (latency, throughput, maximum clock rate, CPU execution time, etc). Simulate both designs with a scenario that illustrates their differences and explain the advantages and drawbacks of each design using these simulations. In order to calculate CPU execution time, assume memory units have a 2 ns operation time, while the register file has a 1 ns operation time. However, calculate the worst path delay of your ALU and adders, assuming that a gate delay is 0.01 ns. Show all your work and reasoning. (See page 372 of your textbook)

Report Guideline:

The laboratory report is an important aspect of the experiment and should not be taken lightly. It should be understandable by an average person with an engineering background. Basically, there are three parts to a design implementation report. They are:

1. Background information

The report should provide a brief overview of the technical aspects and terms used in this experiment.

2. Problem to be solved

What is it that you are trying to accomplish in this lab? Again, only a brief explanation suffices.

3. Design path

How did you come up with your design? How does your design function? Please include relevant flow charts and diagrams if it aids the explanation of your logic. Always try to be concise and, at the same time, be complete. Remember, some things may be obvious to the designer but it would take another designer quite some time to digest it.

4. Design testing

How can you prove that your implementation has met the requirements and solved the problem? Simulation diagrams and output files should go into this section. Simply sticking in a simulation diagram without any explanation will not be acceptable!

5. Final design evaluation

Now that the design is proven to solve the problem, how can it be improved or modified later?

Report reminder:

- Include timing simulations with explanation for all VHDL source files
- Describe and comment all your VHDL source files
- Include a flowchart representation and/or block diagram of your solution to the problem
- Submit a soft copy of all VHDL and graphical design files with your report