Design of a Pipelined 3-Element Adder

Objective

To design and implement a simple pipelined system (other than CPU).

It is important to obtain a deep understanding of the basic concepts of pipelining such as data-stationary control, forwarding, stalling, and flushing. Since the textbook has presented a complete design of the pipelined CPU, it does not provide an opportunity for students to arrive at the basic design of a new pipelined system by themselves. It is hoped that this lab provides such an opportunity.

Introduction

The operation to be performed here (the instruction to be executed) is very simple. Using a pipelined system, a series of such simple instructions are to be executed very much like in a CPU. We need to take care of data dependencies by designing appropriate forwarding unit (FU) and hazard detection and stalling unit (HDU).

Part 1

In this part of the lab, we study a pipelined adder for summing up *three* 16-bit quantities. If an *overflow* is generated then the sum shall not be written back into the destination (result) register.

```
($R) <= ($Z) + ($Y) + ($X) if there is no overflow.
```

The pipeline has 5 stages: **Instruction Fetch (IF), Instruction Decode (= Register Fetch) (ID), Execution 1 (EX1), Execution 2 (EX2),** and **Write Back (WB)**. In EX1, `X_plus_Y` is produced. In EX2, `Z` is added to `X_plus_Y`.

Part 2

Part 2 is similar to Part 1 except that it has only **four** stages. The **EX2** and **WB** stages of part 1 are merged into one state called **EX2WB**.

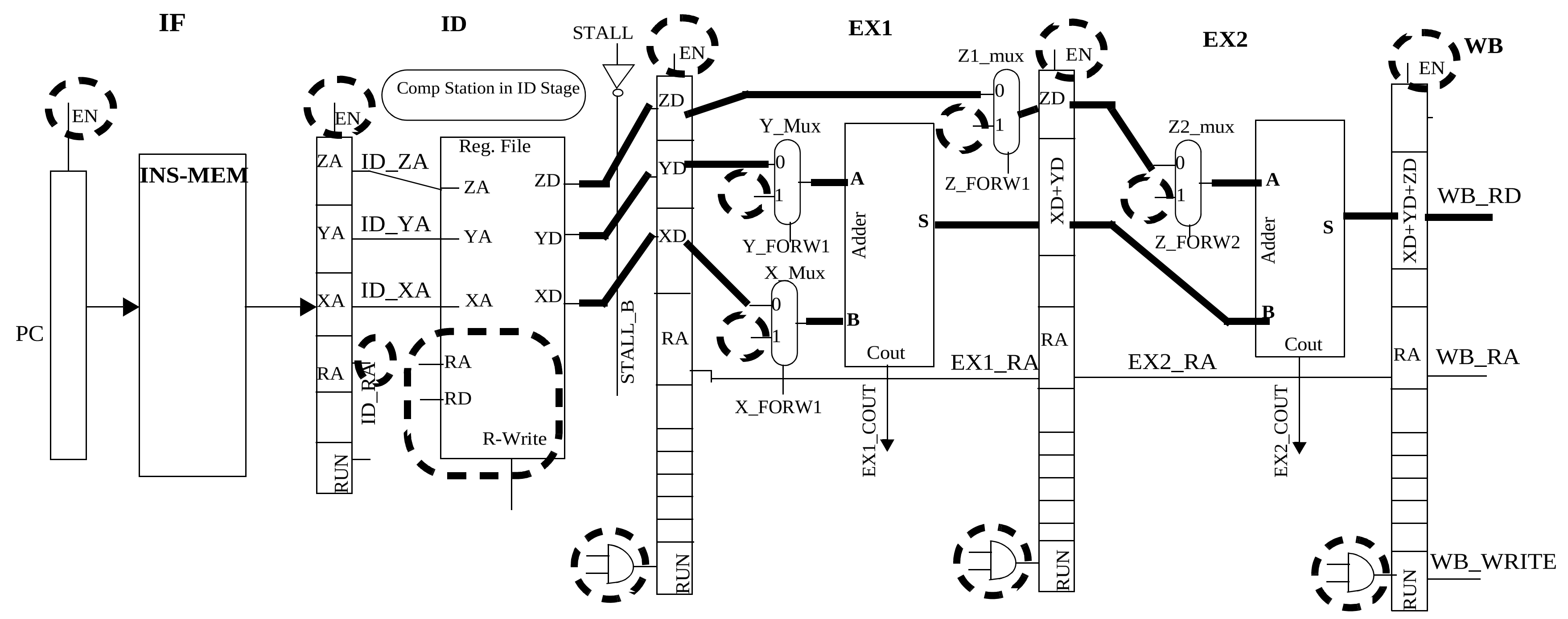
Part 1 Datapath

Please see figure 1 on next page.

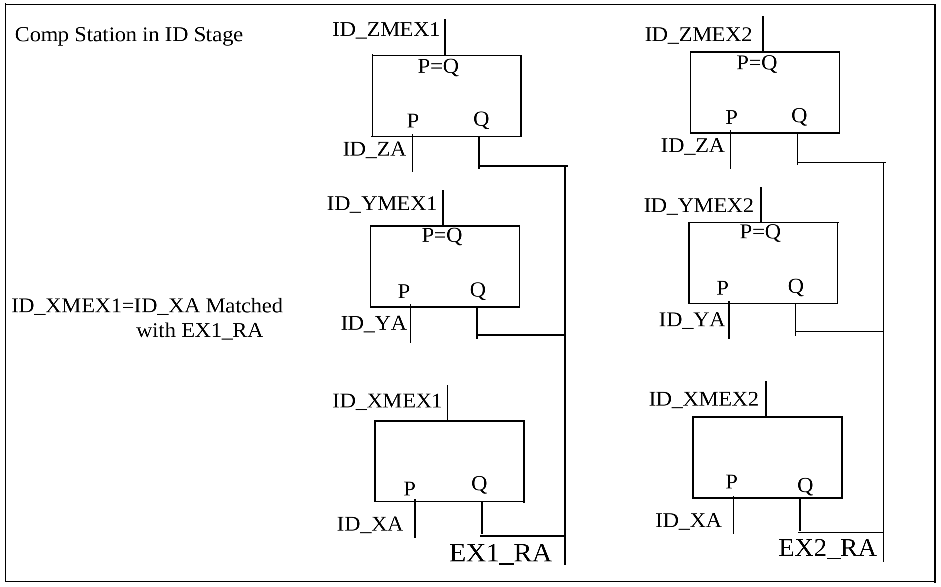
In the **IF** stage, we have a Program Counter (**PC**) and an Instruction Memory (**INS_MEM**). The instruction memory holds a sequence of the summation instructions. The instruction provides the destination register ID (ID = identification = address), `RA[3:0]` ("R" for result), and the three source register ID's, `ZA[3:0]`, `YA[3:0]`, `XA[3:0]`, and a `RUN` control signal. If the `RUN` signal is active then you `ADD`. If it is inactive, then the instruction is treated as a `NOP`. For simplicity, the instruction format has been kept at 32 bits, though we use only 17 bits (the most significant bit `INSTR[31]` is the `RUN` signal and the lower 16 bits `INSTR[15:12]`, `INSTR[11:8]`, `INSTR[7:4]`, and `INSTR[3:0]` are the four register IDs, `R`, `Z`, `Y`, and `X` respectively.

The second stage is called the ID stage (Instruction Decode stage) though there is nothing to decode here. Perhaps RF (for Register Fetch stage) would be a more appropriate name. In the **ID** stage, we have a multi-ported register file with three read ports `X`, `Y`, and `Z` and one write port `R` (R for Result). The register file is an *internal forwarding* register file.

Each of the two execution stages, **EX1** and **EX2**, consists of a 16-bit adder with a carry out. There are forwarding muxes in EX1 and EX2. If an *overflow* is generated then the sum shall not be written back into the register file. This means that the writing into the register file is *conditional* and so is forwarding data to the instructions behind. Overflow converts an instruction into a `NOP`.



1. Complete all missing connections marked in dotted lines ▬ ▬ ▬
   Also complete the RA(Result Address) connection in ID stage (ID_RA).
2. Complete all five enable (EN) controls on the pipeline registers (including PC).
3. Complete the forwarding paths into the four forwarding muxes.
4. Complete logic to inject bubble into the next stage if the current instruction is being stalled or being flushed.
5. Draw the logic on a separate page for generating **STALL, X_FORW1,Y_FORW1, Z_FORW1, Z_FORW2**.

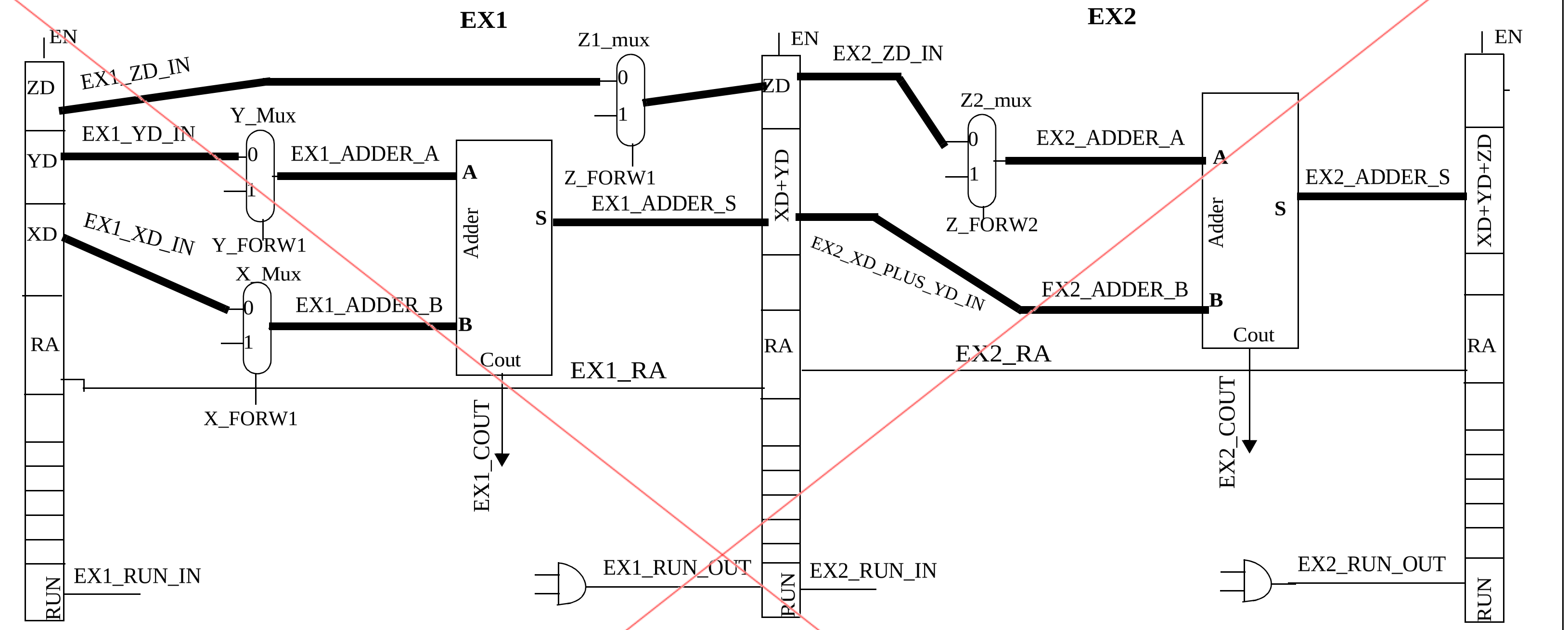


Pinelined 3-element Adder
Block Diagram
LAB 7 Part1

Fig. 1

We stopped completing the structural Verilog implementation of this lab. So, this page is crossed out.

Signal names used in our incomplete Verilog code ee457_lab7_P1.v



```
wire EX1_RUN_IN,EX1_RUN_OUT, X_FORW1, Y_FORW1, Z_FORW1, EX1_COUT;
wire EX1_XMEX2,EX1_YMEX2,EX1_ZMEX2,EX1_ZMEX1;
wire [3:0] EX1_RA;
wire [15:0] EX1_XD_IN,EX1_YD_IN,EX1_ZD_IN, EX1_ADDER_A, EX1_ADDER_B,
            EX1_ADDER_S, EX1_ZD_OUT;
wire [31:0] EX1_INSTR_IN, EX1_INSTR_OUT;
```

← for reverse assembly and creating TimeSpace.txt

```
wire EX2_RUN_IN,EX2_RUN_OUT, Z_FORW2, EX2_COUT;
wire [3:0] EX2_RA;
wire [15:0] EX2_XD_PLUS_YD_IN,EX2_ZD_IN, EX2_ADDER_A, EX2_ADDER_B,
            EX2_ADDER_S;
wire [31:0] EX2_INSTR_IN, EX2_INSTR_OUT;
```

← for reverse assembly and creating TimeSpace.txt

Fig. 1A

We stopped completing the structural Verilog implementation of this lab. So, this page is crossed out.

Structural coding vs. RTL coding:

Here, in this lab, structural coding style is used.

However, you are aware of the fact that the structural coding is less desirable for coding a module.

The other style of coding, RTL_coding, is shown in lab #7 part 3.

Generic Stage register component, `pipe_reg2`

```
module pipe_reg2(rstb,clk,en,
vec16_in1,vec16_in2,vec16_in3,vec16_out1,vec16_out2,vec16_out3,
vec4_in1,vec4_in2,vec4_out1,vec4_out2,
bit_in1, bit_in2, bit_in3, bit_in4, bit_in5, bit_in6,
bit_out1,bit_out2,bit_out3,bit_out4,bit_out5,bit_out6,
instr_in,instr_out);
```

The generic component, pipe_reg2, defined in the ee457_lab7_components.v, is used in the design, ee457_lab7_P1.v three times (instantiated three times) to serves as ID/EX1, EX1/EX2, and EX2/WB stage registers.

This register provides passage of three 16-bit items, two 4-bit items, six single-bit items.

The 16-bit items are for carrying data, the 4-bit items are for carrying register IDs, and the 1-bit items are for carrying control signals such register matches and the RUN signal.

The number of 16-bit, 4-bit, and 1-bit items carried across stages varies between ID/EX1, EX1/EX2, and EX2/WB stage registers.
The pipe_reg2 component is made adequately big (or bigger than needed, perhaps). Students should carefully consider what is needed and tie zeros for unused inputs and leave open unused outputs.

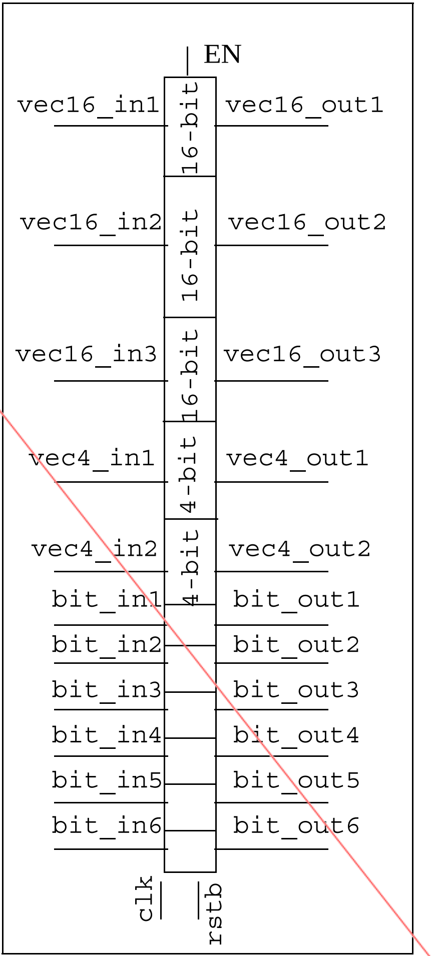


Fig. 1B

Instruction Format

The instruction format is as follows:

```
Add $R, $Z, $Y, $X
instr[31]     = Run      { 1 = Run = ADD },  { 0 = NOP }
instr[15:12] = RA[3:0]
instr[11:8]  = ZA[3:0]
instr[7:4]    = YA[3:0]
instr[3:0]    = XA[3:0]
```

`instr[30:16]` are not used and are always `000_0000_0000_0000`.

Example: `Add $4, $3, $2, $1` ; Translation: `80004321` (Hex)

Part 1 Data-stationary Control

Please see Fig. 1. Data-stationary control is employed here. Since there is no multi-bit opcode to be decoded, there is no "control unit" to act as a *"translator of opcode"* to translate it into *"control signals"* here. The `RUN` control signal is a single-bit opcode and does not need any more decoding. Similar to the HDU (Hazard Detection Unit) and FU (Forwarding Unit) of the pipelined CPU *where register IDs are compared*, here in the pipelined 3-element adder, we have a comparator station, COMP_STATION, where we compare the source register ID's (of `X`, `Y`, and `Z`) of the instruction in ID stage with the destination register ID (of the result register `R`) of the instructions in the EX1 and EX2 stages, and generate appropriate inferences. The inference labels are interpreted as follows:

XMEX1 : Source register X (ID_XA) *Matches* with the destination register in EX1 (EX1_RA), and so on.

Some of these inferences are used in the ID stage itself to stall the instruction, if needed. Others are carried through the pipeline, and are used for forwarding.

Note that, *unlike in the pipelined CPU of Lab #6, where some comparisons are done in HDU, HDU_Br, FU_Br in ID stage and some comparisons are done in FU in EX stage*, here all comparisons are done at **one place** (in the ID stage). (This is like in Lab 6 Part 5.) Hence some of the inferences drawn in the comparator station, may have to be *carried through the pipe and used in later stages* of the pipeline (following the data-stationary method of control).

Overflow and Flushing

In adding up the three quantities, `X`, `Y`, and `Z`, if there is an overflow in any stage (EX1 or EX2), then the result must **not** be written back to the register file. This is achieved by converting the instruction into a NOP (a BUBBLE) by disabling its `RUN` control signal. Thus if the calculation of `X + Y` in the EX1 stage generates an overflow then the instruction must be converted to a `NOP` and a bubble is sent into the EX2 stage, effectively *flushing* the instruction out of the pipeline. Similarly, if the calculation of `(X_plus_Y + Z)` in the EX2 stage generates an overflow, then the instruction must be converted to a `NOP` and a bubble is sent into the WB stage.

Data Hazards/Dependencies, Stalling, and Forwarding

Data dependencies between instructions must be taken care of by your pipeline control, by forwarding and, if forwarding is not possible, by stalling. Wherever possible, data dependencies should be resolved by forwarding. The register file is an *internally forwarding* type (like in the pipelined CPU) and resolves the dependency of the instruction in ID stage on the instruction in WB stage. Other dependencies: Here, we are proposing to provide necessary arrangement for **forwarding data into the EX1 stage and into the EX2 stage from the WB stage** only. This is because we cannot ***generally*** (note, we said generally; there may be exceptions to it, and we use it in Part 2) forward data from the EX2 stage to the EX1 stage as the *final result* is not available at the *beginning* of the clock. For example, the following dependency cannot be resolved by forwarding.

```
($R) <= ($Z) + ($Y) + ($X)
($6) <= ($3) + ($4) + ($5)        ; -- instruction I
($9) <= ($8) + ($7) + ($6)        ; -- instruction II
```

Here the `instruction II` is dependent for **`$6`** `(register X)` upon $6 `(register R)` of the `instruction I`. This dependency can **not** be resolved by the forwarding circuitry. (Question 1.1 **Why ?**). **Please assume that we are not allowed to re-order the order of summation of X, Y, and Z.** In order to resolve the above dependency, the pipeline must stall the dependent instruction in the **ID** stage (and consequently the next instruction in **IF** stage) until a point where the dependency can be handled by forwarding. Question 1.2 Do you need to stall the dependent instruction for *one* clock or *two* clocks or *three* clocks in the above case? Question 1.3 Do you need to stall the dependent instruction in ID stage only or can you stall it in EX1 stage? In lab #6 (MIPs pipelined CPU), do you need to stall a dependent instruction *only in ID stage* or could you possibly let it progress to the EX stage and then stall it? Question 1.4 Can you possibly stall a (any) dependent instruction in EX2 or is it that there is never a meaningful need to stall an instruction in EX2 stage? **Q** **Q** **Q** **Q**

Unlike in the above sequence of instructions, note that in the following sequence of instructions, the dependency of ***$6*** `(register Z)` of the `instruction IV` upon *$6* `(register R)` of the `instruction III`, can be resolved by forwarding (Question 1.5 **How** ?). Hence we need not (and should not) stall the dependent instruction `IV` here. **Q**

```
($R) <= ($Z) + ($Y) + ($X)
($6) <= ($3) + ($4) + ($5)        ; -- instruction III
($9) <= ($6) + ($7) + ($8)        ; -- instruction IV
```

The key point here is that forwarding help can be delayed until the help is really necessary by the dependent instruction for some computation or storage. The `Z` register is only needed in the EX2 stage.
You notice that there are two forwarding muxes, one in EX1 (Z1_mux) and the other in EX2 (Z2_mux), to help the source register `Z` in our data path. Question 1.6 Keeping in mind that, in our design, data is forwarded from the WB stage only, should we use Z1_mux in EX1 or Z2_mux in EX2 to receive forwarding help for *$6* in the above sequence? Write a sequence of instructions which the other `Z` mux (Z1_mux in EX1 or Z2_mux in EX2) is used for forwarding. Question 1.7 Instead of *two 2-to-1 muxes*, Z1_mux and Z2_mux, can we go for *one 3-to-1 mux* in either EX1 or EX2 stage? Answer fully substantiating your reasons with any sketches. You are the designer. Do not jump to conclusions. **Q** **Q**

Spurious Stalls: Are *spurious stalls* possible here? In this design, since the opcode `RUN` (ADD/$\overline{\text{NOP}}$) is a single bit opcode (and hence does not take any time in decoding and recognizing whether it is a register reading instruction or not in the ID stage), spurious stalling of a `NOP` instruction is avoided.
Note that in the pipelined CPU of the textbook, and in our Lab 6, spurious stalls can occur because a *seemingly* dependent instruction may be stalled by the HDU. For example a jump instruction such as *j 3333* may be stalled in the ID stage if there is a spurious match between the 'source register fields' of *j 3333* and the destination register field of a load word instruction in EX stage. This can happen because we are NOT waiting to decode the instruction in the ID stage before we make a decision to stall. We are doing so because we assume that the decoding takes a long time and an attempt to avoid spurious stalls by waiting for decoding will cause elongating the critical path leading to a longer (slower) clock. *Here we are **avoiding** such spurious stalls.* **However,** in this design of the pipelined 3-element adder, we may still stall an instruction in ID stage sometimes because of its dependency upon an instruction ahead of it in EX1 stage and later that instruction in EX1 stage may turn itself into a NOP because of an overflow. In such cases, we lose a clock but our overall numerical results will be as the programmer expects. This is **an *unavoidable* stall**. Though this design is NOT intended to take into account timing aspects, please do not try to wait until the last minute (I should be saying 'last nanosecond'!) to make a decision to stall though it might cost you a clock. It means that you *can not wait* for an instruction in EX1 to finish addition and see if it has produced an overflow to decide whether to stall a dependent instruction in ID stage.

Produce all the 5 control signals needed in Fig 1 here and submit.

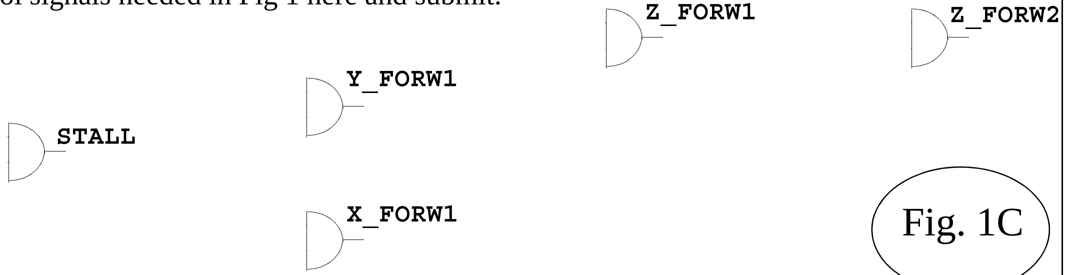


Fig. 1C

Initial contents of the Register File

Register id		Content	Register id		Content
Register 00	0	0001h	Register 08	8	0100h
Register 01	1	0002h	Register 09	9	0200h
Register 02	2	0004h	Register 10	A	0400h
Register 03	3	0008h	Register 11	B	0800h
Register 04	4	0010h	Register 12	C	1000h
Register 05	5	0020h	Register 13	D	2000h
Register 06	6	0040h	Register 14	E	FFF8h
Register 07	7	0080h	Register 15	F	FFFFh

Part 2 Datapath

Do not even read this part until you finished Part 1 design and answered all Part 1 questions, particularly the question 1.7. As we said before, this part is similar to Part 1, except that it has only **four** stages. The **EX2** and **WB** stages of part 1 are merged into one stage called **EX2WB**. Because of this merger, there may be changes to hazard detection/stalling operations and/or forwarding operations. We provided an incomplete block diagram for this part on the last page. We just removed the EX2/WB stage register (of the Part 1) but did not fix anything else. You please remove items which are not needed and complete the rest of this block diagram. We are not providing separate exercise verilog files for this part. Most likely you will not have time towards the end of the semester to implement this design.

Though we are not doing timing design, let us apply this simple rule regarding helping (forwarding) towards the end of the clock. The register file is internally forwarding and we assume that the clock is wide enough for the instruction in EX2WB stage to perform the original EX2 operation of adding Z, checking to see if there was an overflow, and writing into the register file and forwarding the result data (write data) at the end of the clock to the instruction in the ID stage. Question 2.1 If that is the case, the instruction in EX2WB stage should not have any difficulty to help the instruction in EX1 towards the end of the clock for the ____________ (X / Y / Z) register as the recipient of the help does not have to perform any addition operation on this data.

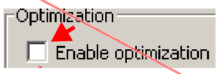
Question 2.2 State which mux(es) you *removed* and which mux(es) you *retained* and why. Question 2.3 Finally how many comparators in the COMP_STATION are really used (needed) in this design? Why the rest are not needed? Question 2.4 If the clock period is the same for Part 1 and Part 2, which of these (Part 1 or Part 2) performs better? Is the answer data dependent (meaning for some data Part 1 performs better than Part 2 and for some other data Part 2 performs better than Part 1)? Please explain. Note: There are no branches/jumps here and if we are executing millions of instructions, we should not care for a difference of just one clock. **QQ Q**

~~Instruction Streams~~

~~Please read the testbench file `ee457_lab7_P1_tb.v`. The testbench performs nearly exhaustive testing of all possible cases. It is important to read and understand the instruction streams in testbench files before you use them for debugging/proving your design.~~

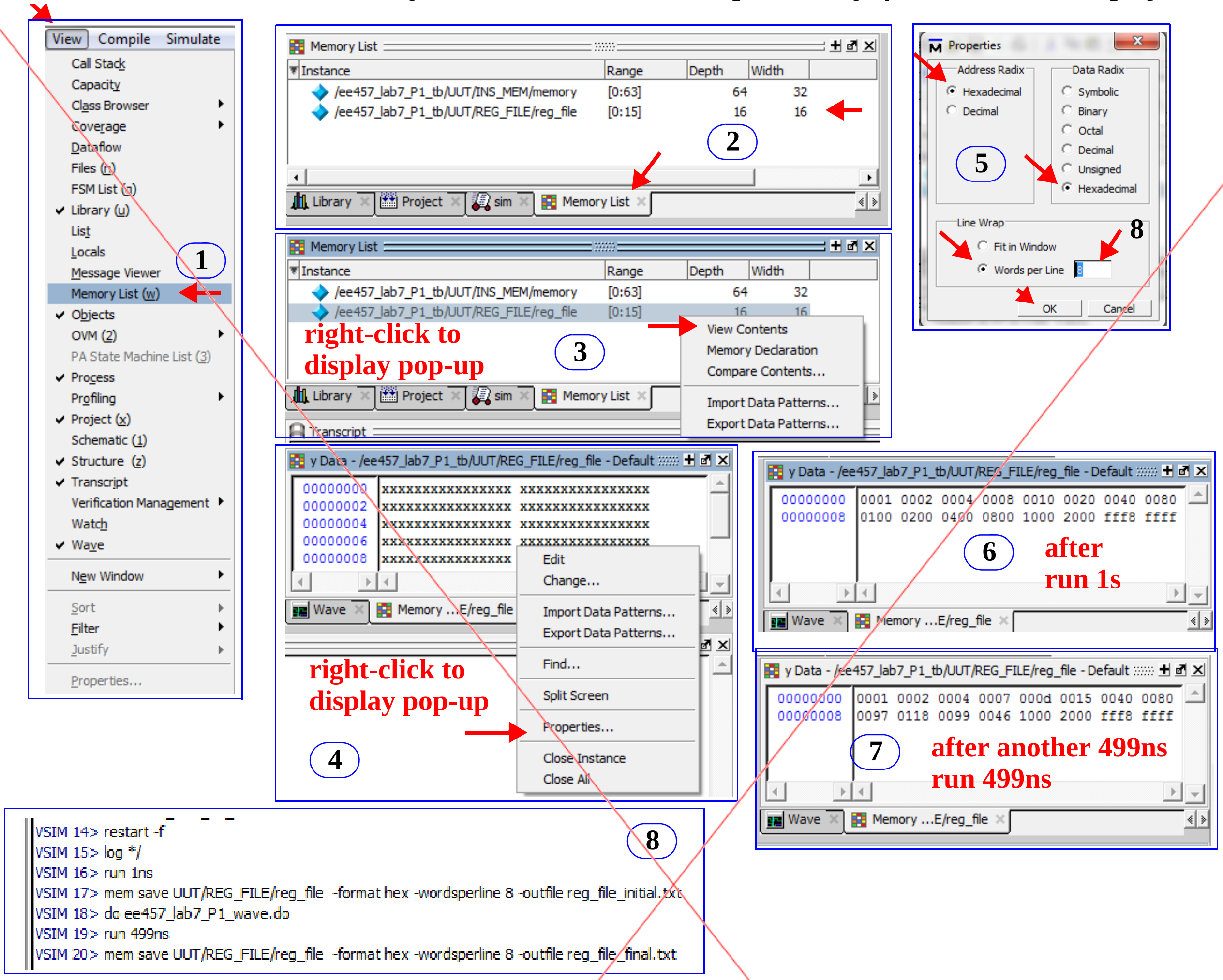
~~What you have to do~~

1. ~~Complete Figures 1 and 1C. Go though the Figure 1A and 1B.~~
2. ~~Create a folder `C:\ModelSim_projects\ee457_lab7_P1` (under your `C:\ModelSim_projects`). Download the `.zip` file (`ee457_lab7_P1.zip`) and extract the .v and `.do` files and place them in the above directory.~~
3. ~~Create a modelsim project with the project name `ee457_lab7_P1`. Choose `ee457_lab7_P1` for the project directory.~~
4. ~~Add all verilog files to the project.~~
5. ~~Go through **`ee457_lab7_components.v`**. Edit (in Notepad++) and complete the **`ee457_lab7_P1.v`** .~~
6. ~~Go through **`ee457_lab7_P1_tb.v`** and understand the instruction stream used for testing. Compile all 3 Verilog files.~~
7. ~~Start simulation by selecting `ee457_lab7_P1_tb`. Unselect "Enable optimization".~~
8. ~~Use the given .do file to set up the waveform (command: `do ee457_lab7_P1_wave.do`).~~


Optimization
☐ Enable optimization

We stopped completing the structural Verilog implementation of this lab. So, this page is crossed out.

9. Select the Memories tab in the workspace and double click on the reg_file to display its contents in the right pane.



Instance	Range	Depth	Width
/ee457_lab7_P1_tb/UUT/INS_MEM/memory	[0:63]	64	32
/ee457_lab7_P1_tb/UUT/REG_FILE/reg_file	[0:15]	16	16

```
00000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
00000002 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
00000004 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
00000008 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
```

```
00000000 0001 0002 0004 0008 0010 0020 0040 0080
00000008 0100 0200 0400 0800 1000 2000 fff8 ffff
```

```
00000000 0001 0002 0004 0007 000d 0015 0040 0080
00000008 0097 0118 0099 0046 1000 2000 fff8 ffff
```

```
VSIM 14> restart -f
VSIM 15> log */
VSIM 16> run 1ns
VSIM 17> mem save UUT/REG_FILE/reg_file  -format hex -wordsperline 8 -outfile reg_file_initial.txt
VSIM 18> do ee457_lab7_P1_wave.do
VSIM 19> run 499ns
VSIM 20> mem save UUT/REG_FILE/reg_file  -format hex -wordsperline 8 -outfile reg_file_final.txt
```

10. Initially the data content of the memory is displayed as xxxx.
You can simulate for a very short time (say 1ns) (`run 1ns`) to display the actual initial contents.

11. Run the simulation for 499ns more (run 499ns) (total 500ns).

12. Verify the final register contents. Look at the waveform to see if any signals are misbehaving. Look at the **`TimeSpace.txt`** file, produced by the testbench and placed in the project directory. Use Notepad or Notepad++ to look at this file as WordPad (on Windows 7) refuses to open this file as the file is still being controlled/updated by ModelSim. (On Windows-XP, I could open the **`TimeSpace.txt`** file in Notepad++ while simulation is not yet done. If your O.S. does not allow you to open the file while simulation is going on, please end the simulation, inspect the file and restart the simulation again.
Debugging: Perform incremental simulation to find errors. Use **`restart -f`** to start the simulation again from 0ns. Now run for short lengths of time examining the register file contents, the waveforms, and the **`TimeSpace.txt`**.

13. After finishing all debugging, compile the revised .v file, again restart (**`restart -f`**), run for 1ns, examine register contents using the command-line command at VSIM> prompt: *"examine -radix hex UUT/REG_FILE/reg_file"* . Further run simulation for 499ns more (run 499ns) and again examine register contents using command *"examine -radix hex UUT/REG_FILE/reg_file"* .

```
VSIM> examine -radix hex UUT/REG_FILE/reg_file
# {0001 0002 0004 0007 000d 0015 0040 0080 0097 0118 0099 0046 1000 2000 fff8 ffff}
```

We stopped completing the structural Verilog implementation of this lab. So, this page is crossed out.

14. A better choice is to get the output files created by Modelsim for submission as shown below. Do the following to get the right files (`reg_file_initial.txt and reg_file_final.txt`) for submission.

```
restart -f
log */run 1ns
mem save UUT/REG_FILE/reg_file  -format hex -wordsperline 8 -outfile reg_file_initial.txt
do ee457_lab7_P1_wave.do
run 499ns
mem save UUT/REG_FILE/reg_file  -format hex -wordsperline 8 -outfile reg_file_final.txt
```

15. General Guidelines

15.1. Start early and seek help early if needed.

15.2. Finally submit online (through your unix account) (submission commands specified separately) one set of files for a team of two students.

15.3. You need to use the file names exactly as stated and follow the submission procedure exactly as specified. We use unix script files to automate grading.

15.4. Non-working lab submission. In simulation, it will be evident if your lab is not working. We discourage you from submitting a non-working lab. If you want to submit a non-working lab, each member of your team needs to send an email to all lab graders (with a copy to all TAs) stating in the subject line, "EE457 Non-working lab submission request" and obtain an approval from one of them. Submitting a non-working lab or partial lab without such approval is interpreted as an intention to cheat. Sorry to say all this, but this makes sure that the system works well.

What you have to turn-in

~~On-line (one submission for a team of 2 students)~~

~~Please turn in the following :~~

```
~~submit -user ee457lab -tag puvvada_lab7_p1 ee457_lab7_P1.v RF_Content_Lab7_P1.txt
TimeSpace.txt names.txt~~
```

Paper submission (individual effort, each student separately)

Part 1: Please complete, staple together, and submit page 2 (Fig. 1 Block diagram), page 6 (Fig. 1C Logic for the 5 signals, and this page (Q & A).

Selected questions for Part 1 paper submission

Q 1.6 Keeping in mind that, in our design, data is forwarded from the WB stage only, should we use Z1_mux in EX1 or Z2_mux in EX2 to receive forwarding help for *$6* in the sequence below?

Answer (circle): Z1MUX in EX1 / Z2MUX in EX2

```
($R) <= ($Z) + ($Y) + ($X)
($6) <= ($3) + ($4) + ($5)        ; -- instruction III
($9) <= ($6) + ($7) + ($8)        ; -- instruction IV
```

Write a sequence of instructions which the other Z mux (Z1_mux in EX1 or Z2_mux in EX2) is used for forwarding.

.

Q 1.7 Instead of *two 2-to-1 muxes*, Z1_mux and Z2_mux, can we go for *one 3-to-1 mux* in either EX1 or EX2 stage? Answer fully substantiating your reasons with any sketches. Do not jump to conclusions.

.

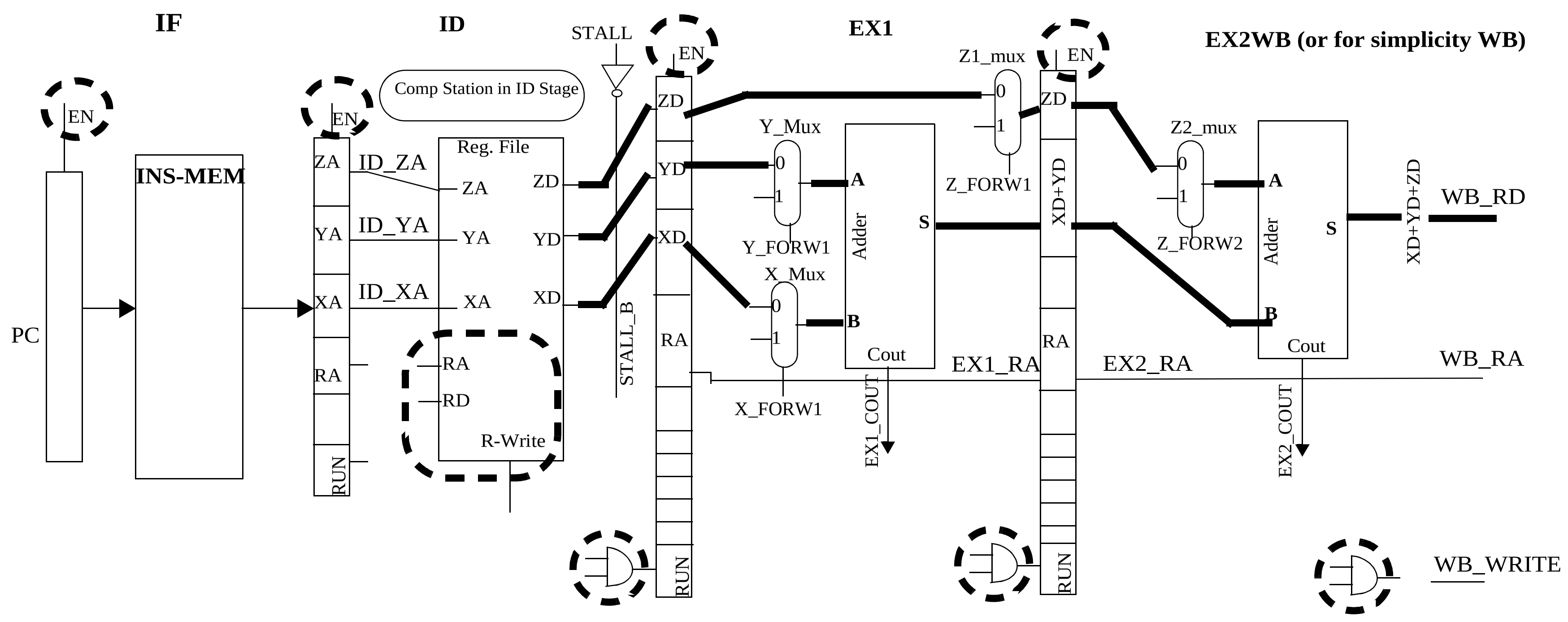
Part 2 paper submission: Please complete, staple together, and submit the last page (block diagram) and this page.

Selected questions for Part 2 paper submission:

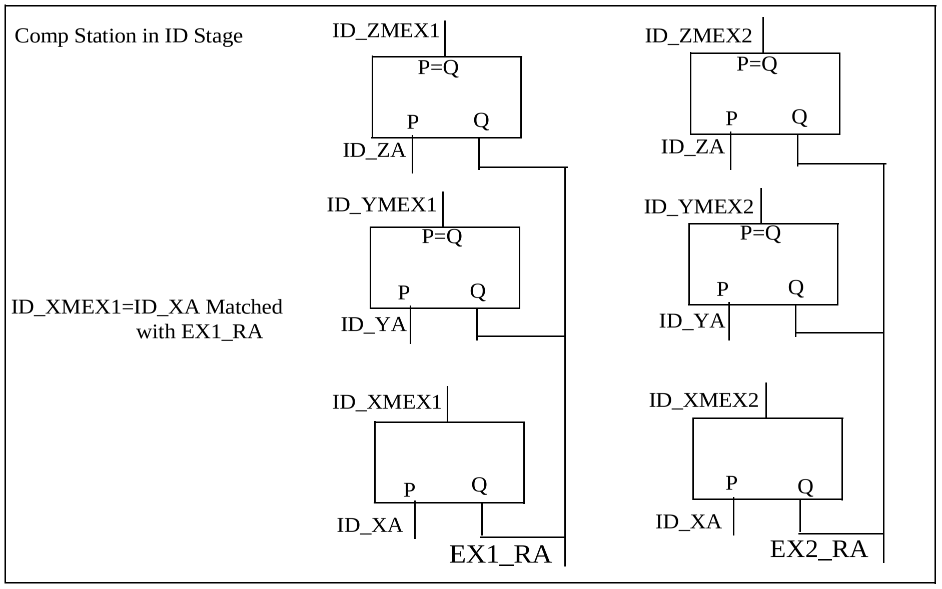
Q 2.1 You do not need to submit answer to the simple question 2.1 on page 7.
Q 2.2 State which mux(es) you *removed* which mux(es) you *retained* and why.

Q 2.3 Finally how many comparators in the COMP_STATION are really used in this design? Why the rest are not needed?

Q 2.4 If the clock period is the same for Part 1 and Part 2, which of these (Part 1 or Part 2) performs better? Is the answer data dependent (meaning for some data Part 1 performs better than Part 2 and for some other data Part 2 performs better than Part 1)? Please explain. Note: There are no branches/jumps here and if we are executing millions of instructions, we should not care for a difference of just one clock
.



This figure is basically the same as the Fig. 1 for part 1 on page 2 except that the EX2/WB stage register was removed as we are merging the EX2 and WB stages into one EX2WB stage.
You need to remove items which are not necessary and comple the rest..
You need to generate any STALL and/or FORW (forwadng) signals on this page itself.



Pinelined 3-element Adder
Block Diagram
LAB 7 Part2 (with EX2 and WB merged)

Fig. 2