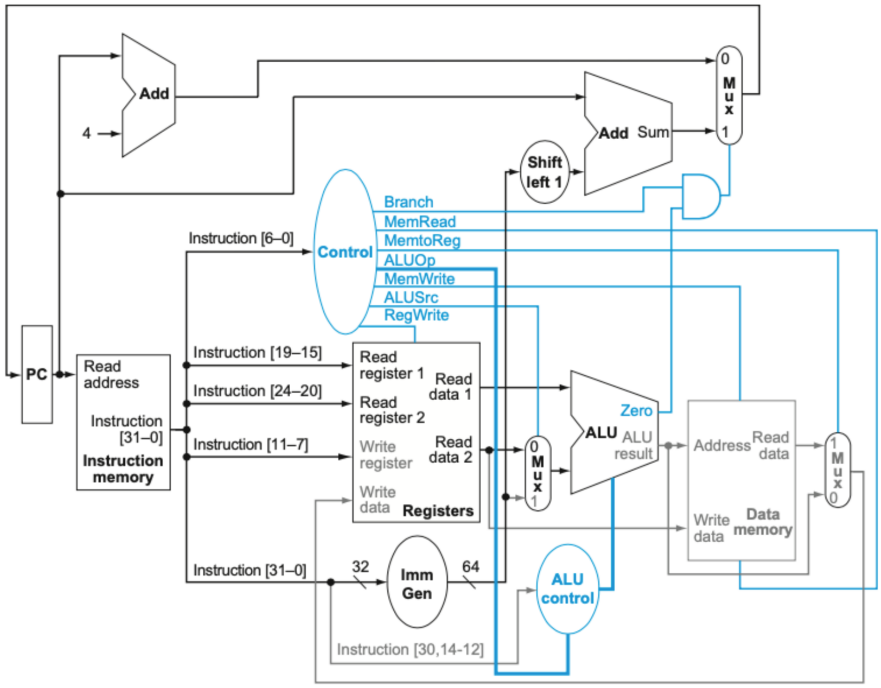
1 Handwritten (48%)

Each 2%.



4.7 Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

4.7.1 [5] <§4.4> What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?

4.7.2 [10] <§4.4> What is the latency of `ld`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.3 [10] <§4.4> What is the latency of `sd`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.4 [5] <§4.4> What is the latency of `beq`?

4.7.5 [5] <§4.4> What is the latency of an I-type instruction?

4.7.6 [5] <§4.4> What is the minimum clock period for this CPU?

4.27 Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add  x15, x12, x11
ld   x13, 4(x15)
ld   x12, 0(x2)
or   x13, x15, x13
sd   x13, 0(x15)
```

4.27.1 [5] <§4.7> If there is no forwarding or hazard detection, insert `NOP`s to ensure correct execution.

4.27.2 [10] <§4.7> Now, change and/or rearrange the code to minimize the number of `NOP`s needed. You can assume register `x17` can be used to hold temporary values in your modified code.

4.27.3 [10] <§4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

4.27.4 [20] <§4.7> If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.59.

4.27.5 [10] <§4.7> If there is no forwarding, what new input and output signals do we need for the hazard detection unit in Figure 4.59? Using this instruction sequence as an example, explain why each signal is needed.

4.27.6 [20] <§4.7> For the new hazard detection unit from 4.26.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

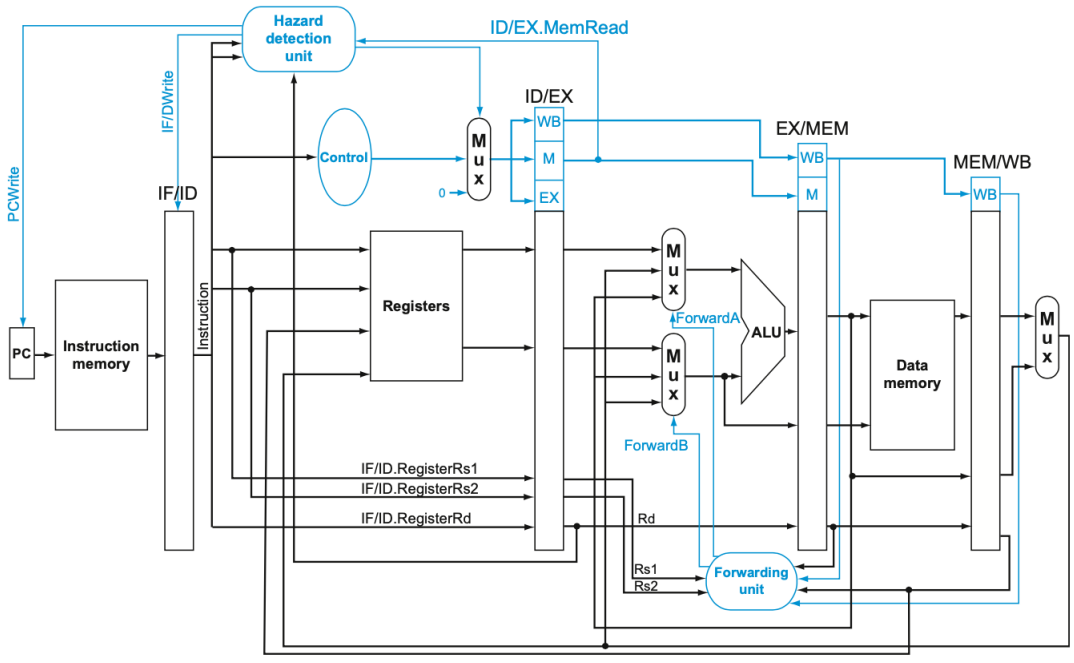


FIGURE 4.58 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

4.28 The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-type	beqz/bnez	jal	ld	sd
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

4.28.1 [10] <§4.8> Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the ID stage and applied in the EX stage that there are no data hazards, and that no delay slots are used.

4.28.2 [10] <§4.8> Repeat 4.28.1 for the "always-not-taken" predictor.

4.28.3 [10] <§4.8> Repeat 4.28.1 for the 2-bit predictor.

4.28.4 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions to some ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.28.5 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.28.6 [10] <§4.8> Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

4.29 This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: `T`, `NT`, `T`, `T`, `NT`.

4.29.1 [5] <§4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.29.2 [5] <§4.8> What is the accuracy of the 2-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.61 (predict not taken)?

4.29.3 [10] <§4.8> What is the accuracy of the 2-bit predictor if this pattern is repeated forever?

4.29.4 [30] <§4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.29.5 [10] <§4.8> What is the accuracy of your predictor from 4.29.4 if it is given a repeating pattern that is the exact opposite of this one?

4.29.6 [20] <§4.8> Repeat 4.29.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

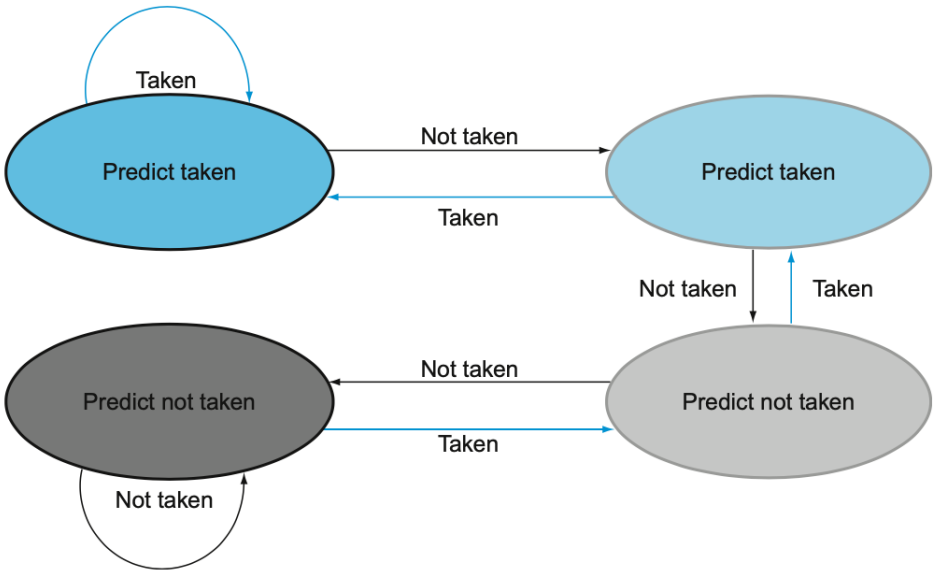


FIGURE 4.61 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

2 Programming (62%)

Pipelined CPU (47%)

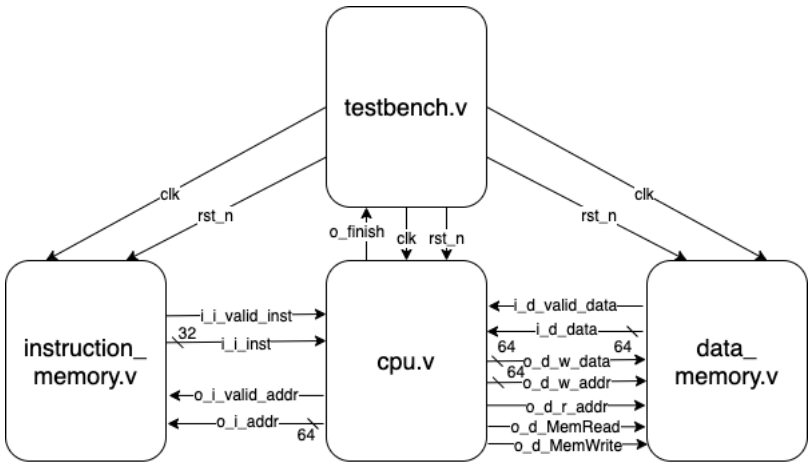
In this section, we are going to implement a pipeline cpu.
The provided instruction memory is as follows:

Signal	I/O	Width	Functionality
i_clk	Input	1	Clock signal
i_rst_n	Input	1	Active low asynchronous reset
i_valid	Input	1	Signal that tells pc-address from cpu is ready
i_addr	Input	64	64-bits address from cpu
o_valid	Output	1	Valid when instruction is ready
o_inst	Output	32	32-bits instruction to cpu

And the provided data memory is as follows:

Signal	I/O	Width	Functionality
i_clk	Input	1	Clock signal
i_rst_n	Input	1	Active low asynchronous reset
i_data	Input	64	64-bits data that will be stored
i_w_addr	Input	64	Write to target 64-bits address
i_r_addr	Input	64	Read from target 64-bits address
i_MemRead	Input	1	One cycle signal and set current mode to reading
i_MemWrite	Input	1	One cycle signal and set current mode to writing
o_valid	Output	1	One cycle signal telling data is ready (used when `ld` happens)
o_data	Output	64	64-bits data from data memory (used when `ld` happens)

The test environment is as follows:



The naming of the wire
is in the perspective of cpu

We will only test the instructions highlighted in the red box, as the figures below

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

And one more instruction to be implemented is

i_inst	Function	Description
32'b11111111111111111111111111111111	`Stop`	Stop and set o_finish to 1

All the environment settings are the same as HW3 except the rule of accessing `data_memory.v` and `instruction_memory.v`, and the interface of modules are changed this time. See the supplementary.pdf for more information.

You may want to reference the diagram of pipelined cpu from textbook.

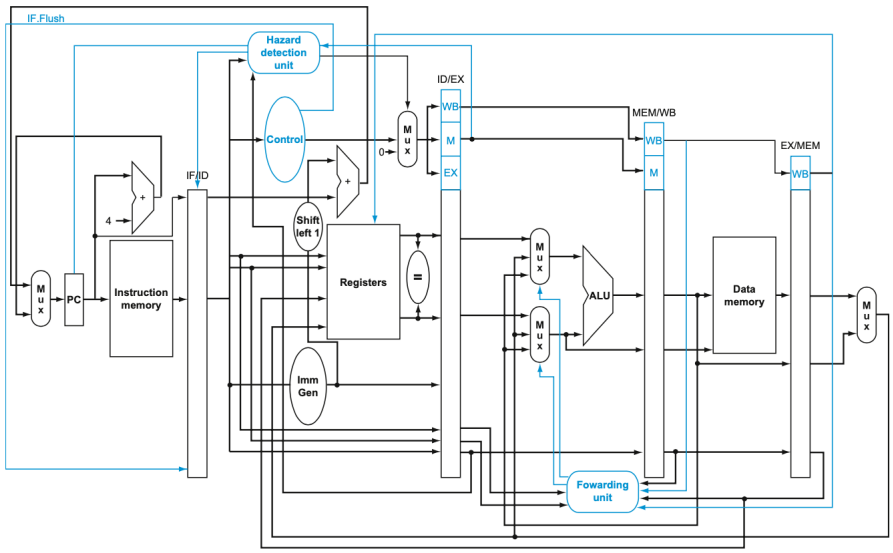


FIGURE 4.62 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.55 and the multiplexor controls from Figure 4.49.

To make sure that pipeline is actually implemented in your design, we are going to use an open source synthesis tool Yosys to check the timing of the critical path in your design. We'll also use the FreePDK 45 nm process standard cell library provided here.

You can either build Yosys yourself or use the image provided

```
docker pull ntuca2020/hw4 # size ~ 1.28G
docker run --name=test -it ntuca2020/hw4
cd /root
ls
```

Folder structure for this homework:

```
HW4/
  |-- testcases/
  |    |-- generate.s
  |    `-- generate.cpp
  |-- codes/
  |    |-- cpu.v
  |    |-- data_memory.v         // provided data memory
  |    `-- instruction_memory.v  // provided instruction memory
  |-- testbench.v
  |-- Makefile
  |-- cpu.ys         // synthesis command
  `-- stdcells.lib   // FreePDK 45 nm standard cell library
```

Specify all the used modules in the **cpu.ys** file, then run

```
make       // Compile
make test  // Test all test cases
make time  // Show the timing and area used in your design
```

Information about your design is shown when running `make time`:

```
ABC: WireLoad = "none"  Gates =  13123 ( 14.8 %)   Cap =  3.2 ff (  1.9 %)
Area =    17519.56 ( 87.9 %)   Delay =  1091.13 ps  (  5.1 %)
```

You can optimize the cpu for the 3 workloads (code address range, data address range, etc), but it should not affect other test cases.

Grading:

- Correctness check (20%)
  - 10 testcases, each **2%** for correctness check
- Required area and frequency (inverse of delay) (13%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 10MHz (2%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 100MHz (2%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 200MHz (2%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 500MHz (2%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 800MHz (2%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 1000MHz (1%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 1200MHz (1%)
  - Area $<$ 25,000 μm^2, **and** frequency $>$ 1500MHz (1%)
- Required time (clock cycle * operating frequency) to finish workloads from last 3 testcases. (14%)
  - Workload1 $<$ 100,000 ns (2%)
  - Workload2 $<$ 150,000 ns (2%)
  - Workload3 $<$ 200,000 ns (2%)
  - Workload1 $<$ 10,000 ns (2%)
  - Workload2 $<$ 15,000 ns (2%)
  - Workload3 $<$ 20,000 ns (2%)
  - Workload1 $<$ 5,000 ns, **and** Workload2 $<$ 20,000 ns, **and** Workload3 $<$ 15,000 ns (1%)
  - Workload1 $<$ 3,500 ns, **and** Workload2 $<$ 9,000 ns, **and** Workload3 $<$ 10,000 ns (1%)

Report (15%)

You can describe your pipeline design and how you did it and answer the following questions.

- What is the latency of each module in your design? (e.g. ALU, register_file)
- Which path is the critical path of your cpu? And how can you decrease the latency of it?
- How to solve data hazard?
- How to solve control hazard?
- Describe 3 different workloads attributes, and which one can be improved tremendously by branch predictor?
- Is it always beneficial to insert multiple stage of pipeline in designs? How does it affect the latency?

Submission

- Zip and upload your file to ceiba in the following format:

```
r09922028/              <-- zip this folder
    |-- cpu.ys          // specify the used *.v file, not including testbench and memory
    |-- cpu.f           // specify the used *.v file, including testbench and memory
    |-- codes/          // put all your *.v file here, including cpu_syn.v
    |-- handwritten.pdf // handwritten part
    `-- report.pdf      // report on programming part
```

- Late submission within one-week: (Total score)*0.8
- Late submission within two-week: (Total score)*0.6
- Late submission over two-week: (Total score)*0
- If there's any question, please send email to ntuca2020@gmail.com.
- TA hour for this homework: Wed 15:00 17:00 and Thur 15:00 17:00