4.1.2 [5] <§4.3> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.3> Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

4.2 [10] <§4.4> Explain each of the "don't cares" in Figure 4.18.

4.3 Consider the following instruction mix:

R-type	I-type (non-ld)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

4.4 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get "broken" and always register a logical 0. This is often called a "stuck-at-0" fault.

4.4.1 [5] <§4.4> Which instructions fail to operate correctly if the `MemToReg` wire is stuck at 0?

4.4.2 [5] <§4.4> Which instructions fail to operate correctly if the `ALUSrc` wire is stuck at 0?

4.5 In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: `0x00c6ba23`.

4.5.1 [10] <§4.4> What are the values of the ALU control unit's inputs for this instruction?

4.5.2 [5] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.5.3 [10] <§4.4> For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at `Reg [xn]`.

4.5.4 [10] <§4.4> What are the input values for the ALU and the two add units?

4.5.5 [10] <§4.4> What are the values of all inputs for the registers unit?

4.6 Section 4.4 does not discuss I-type instructions like `addi` or `andi`.

4.6.1 [5] <§4.4> What additional logic blocks, if any, are needed to add I-type instructions to the CPU shown in Figure 4.21? Add any necessary logic blocks to Figure 4.21 and explain their purpose.

4.6.2 [10] <§4.4> List the values of the signals generated by the control unit for `addi`. Explain the reasoning for any "don't care" control signals.

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?

4.8 [10] <§4.4> Suppose you could build a CPU where the clock cycle time was different for each instruction. What would the speedup of this new CPU be over the CPU presented in Figure 4.21 given the instruction mix below?

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

4.9 Consider the addition of a multiplier to the CPU shown in Figure 4.21. This addition will add 300 ps to the latency of the ALU, but will reduce the number of instructions by 5% (because there will no longer be a need to emulate the multiply instruction).

4.9.1 [5] <§4.4> What is the clock cycle time with and without this improvement?

4.9.2 [10] <§4.4> What is the speedup achieved by adding this improvement?

4.9.3 [10] <§4.4> What is the slowest the new ALU can be and still result in improved performance?

4.10 When processor designers consider a possible improvement to the processor datapath, the decision usually depends on the cost/performance trade-off. In the following three problems, assume that we are beginning with the datapath from Figure 4.21, the latencies from Exercise 4.7, and the following costs:

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Single gate	Control
1000	200	10	100	30	2000	5	100	1	500

Suppose doubling the number of general purpose registers from 32 to 64 would reduce the number of `ld` and `sd` instruction by 12%, but increase the latency of the register file from 150 ps to 160 ps and double the cost from 200 to 400. (Use the instruction mix from Exercise 4.8 and ignore the other effects on the ISA discussed in Exercise 2.18.)

4.10.1 [5] <§4.4> What is the speedup achieved by adding this improvement?

4.10.2 [10] <§4.4> Compare the change in performance to the change in cost.

4.10.3 [10] <§4.4> Given the cost/performance ratios you just calculated, describe a situation where it makes sense to add more registers and describe a situation where it doesn't make sense to add more registers.

4.11 Examine the difficulty of adding a proposed `lwi.d rd, rs1, rs2` ("*L*oad *W*ith *I*ncrement") instruction to RISC-V.

Interpretation: `Reg[rd]=Mem[Reg[rs1]+Reg[rs2]]`

4.11.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.11.2 [5] <§4.4> Which existing functional blocks (if any) require modification?

4.11.3 [5] <§4.4> Which new data paths (if any) do we need for this instruction?

4.11.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12 Examine the difficulty of adding a proposed `swap rs1, rs2` instruction to RISC-V.

Interpretation: `Reg[rs2]=Reg[rs1]; Reg[rs1]=Reg[rs2]`

4.12.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.12.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.12.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.12.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.13 Examine the difficulty of adding a proposed `ss rs1, rs2, imm` (Store Sum) instruction to RISC-V.

Interpretation: `Mem[Reg[rs1]]=Reg[rs2]+immediate`

4.13.1 [10] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.13.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.13.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.13.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.13.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.14 [5] <§4.4> For which instructions (if any) is the Imm Gen block on the critical path?

4.15 `ld` is the instruction with the longest latency on the CPU from Section 4.4. If we modified `ld` and `sd` so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in `rs1` before calling `ld/sd`), then no instruction would use both the ALU and Data memory. This would allow us to reduce the clock cycle time. However, it would also increase the number of instructions, because many `ld` and `sd` instructions would need to be replaced with `ld/add` or `sd/add` combinations.

4.15.1 [5] <§4.4> What would the new clock cycle time be?

4.15.2 [10] <§4.4> Would a program with the instruction mix presented in Exercise 4.7 run faster or slower on this new CPU? By how much? (For simplicity, assume every `ld` and `sd` instruction is replaced with a sequence of two instructions.)

4.15.3 [5] <§4.4> What is the primary factor that influences whether a program will run faster or slower on the new CPU?

4.15.4 [5] <§4.4> Do you consider the original CPU (as shown in Figure 4.21) a better overall design; or do you consider the new CPU a better overall design? Why?

4.16 In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5] <§4.5> What is the clock cycle time in a pipelined and non-pipelined processor?

4.16.2 [10] <§4.5> What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor?

4.16.3 [10] <§4.5> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.16.4 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.16.5 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.17 [10] <§4.5> What is the minimum number of cycles needed to completely execute `n` instructions on a CPU with a `k` stage pipeline? Justify your formula.

4.18 [5] <§4.5> Assume that `x11` is initialized to 11 and `x12` is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for

addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers x13 and x14 be?

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
```

4.19 [10] <§4.5> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register x15 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x11, x11
```

4.20 [5] <§4.5> Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x13, x12
```

4.21 Consider a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n-instruction program requires an additional 4*n NOP instructions to correctly handle data hazards.

4.21.1 [5] <§4.5> Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that adding forwarding hardware will reduce the number of NOPs from .4*n to .05*n, but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

4.21.2 [10] <§4.5> Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

4.21.3 [10] <§4.5> Repeat 4.21.2; however, this time let x represent the number of NOP instructions relative to n. (In 4.21.2, x was equal to .4.) Your answer will be with respect to x.

4.21.4 [10] <§4.5> Can a program with only `.075*n NOPs` possibly run faster on the pipeline with forwarding? Explain why or why not.

4.21.5 [10] <§4.5> At minimum, how many `NOP`s (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

4.22 [5] <§4.5> Consider the fragment of RISC-V assembly below:

```
sd   x29, 12(x16)
ld   x29, 8(x16)
sub  x17, x15, x14
beqz x17, label
add  x15, x11, x14
sub  x15, x30, x14
```

Suppose we modify the pipeline so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

4.22.1 [5] <§4.5> Draw a pipeline diagram to show were the code above will stall.

4.22.2 [5] <§4.5> In general, is it possible to reduce the number of stalls/`NOP`s resulting from this structural hazard by reordering code?

4.22.3 [5] <§4.5> Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding `NOP`s to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

4.22.4 [5] <§4.5> Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix from Exercise 4.8.)

4.23 If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. (See Exercise 4.15.) As a result, the MEM and EX stages can be overlapped and the pipeline has only four stages.

4.23.1 [10] <§4.5> How will the reduction in pipeline depth affect the cycle time?

4.23.2 [5] <§4.5> How might this change improve the performance of the pipeline?

4.23.3 [5] <§4.5> How might this change degrade the performance of the pipeline?

4.24 [10] <§4.7> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
ld x11, 0(x12):     IF ID EX ME WB
add x13, x11, x14:   IF ID EX..ME WB
or x15, x16, x17:     IF ID..EX ME WB
```

Choice 2:

```
ld x11, 0(x12):   IF ID EX ME WB
add x13, x11, x14:  IF ID..EX ME WB
or x15, x16, x17:    IF..ID EX ME WB
```

4.25 Consider the following loop.

```
LOOP: ld    x10, 0(x13)
      ld    x11, 8(x13)
      add   x12, x10, x11
      subi  x13, x13, 16
      bnez  x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

4.25.1 [10] <§4.7> Show a pipeline execution diagram for the first two iterations of this loop.

4.25.2 [10] <§4.7> Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the `subi` is in the IF stage. End with the cycle during which the `bnez` is in the IF stage.)

4.26 This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.53. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions has a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the next instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "EX to 3rd" and "MEM to 3rd" dependences

are not counted because they cannot result in data hazards. We also assume that branches are resolved in the EX stage (as opposed to the ID stage), and that the CPI of the processor is 1 if there are no data hazards.

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and EX to 2nd
5%	20%	5%	10%	10%

Assume the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
120 ps	100 ps	110 ps	130 ps	120 ps	120 ps	120 ps	100 ps

4.26.1 [5] <§4.7> For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

4.26.2 [5] <§4.7> For each RAW dependency above, how many `NOP`s would need to be inserted to allow your code from 4.26.1 to run correctly on a pipeline with no forwarding or hazard detection? Show where the `NOP`s could be inserted.

4.26.3 [10] <§4.7> Analyzing each instruction independently will over-count the number of `NOP`s needed to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, the sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

4.26.4 [5] <§4.7> Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

4.26.5 [5] <§4.7> What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

4.26.6 [10] <§4.7> Let us assume that we cannot afford to have three-input multiplexors that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). What is the CPI for each option?

4.26.7 [5] <§4.7> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

4.26.8 [5] <§4.7> What would be the additional speedup (relative to the fastest processor from 4.26.7) be if we added "time-travel" forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

4.26.9 [5] <§4.7> The table of hazard types has separate entries for "EX to 1st" and "EX to 1st and EX to 2nd". Why is there no entry for "MEM to 1st and MEM to 2nd"?

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.

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.

4.30 This exercise explores how exception handling affects pipeline design. The first three problems in this exercise refer to the following two instructions:

Instruction 1	Instruction 2
beqz x11, LABEL	ld x11, 0(x12)

4.30.1 [5] <§4.9> Which exceptions can each of these instructions trigger? For each of these exceptions, specify the pipeline stage in which it is detected.

4.30.2 [10] <§4.9> If there is a separate handler address for each exception, show how the pipeline organization must be changed to be able to handle this exception. You can assume that the addresses of these handlers are known when the processor is designed.

4.30.3 [10] <§4.9> If the second instruction is fetched immediately after the first instruction, describe what happens in the pipeline when the first instruction causes the first exception you listed in Exercise 4.30.1. Show the pipeline execution diagram from the time the first instruction is fetched until the time the first instruction of the exception handler is completed.

4.30.4 [20] <§4.9> In vectored exception handling, the table of exception handler addresses is in data memory at a known (fixed) address. Change the pipeline to implement this exception handling mechanism. Repeat Exercise 4.30.3 using this modified pipeline and vectored exception handling.

4.30.5 [15] <§4.9> We want to emulate vectored exception handling (described in Exercise 4.30.4) on a machine that has only one fixed handler address. Write the code that should be at that fixed address. Hint: this code should identify the exception, get the right address from the exception vector table, and transfer execution to that handler.

4.31 In this exercise we compare the performance of 1-issue and 2-issue processors, taking into account program transformations that can be made to optimize for 2-issue execution. Problems in this exercise refer to the following loop (written in C):

```
for(i=0;i!=j;i+=2)
    b[i]=a[i]-a[i+1];
```

A compiler doing little or no optimization might produce the following RISC-V assembly code:

```
        li    x12, 0
        jal   ENT
TOP:    slli  x5, x12, 3
        add   x6, x10, x5
        ld    x7, 0(x6)
        ld    x29, 8(x6)
        sub   x30, x7, x29
        add   x31, x11, x5
        sd    x30, 0(x31)
        addi  x12, x12, 2
ENT:    bne   x12, x13, TOP
```

The code above uses the following registers:

i	j	a	b	Temporary values
x12	x13	x10	x11	x5–x7, x29–x31

Assume the two-issue, statically scheduled processor for this exercise has the following properties:

1. One instruction must be a memory operation; the other must be an arithmetic/logic instruction or a branch.
2. The processor has all possible forwarding paths between stages (including paths to the ID stage for branch resolution).
3. The processor has perfect branch prediction.
4. Two instruction may not issue together in a packet if one depends on the other. (See page 324.)
5. If a stall is necessary, both instructions in the issue packet must stall. (See page 324.)

As you complete these exercises, notice how much effort goes into generating code that will produce a near-optimal speedup.

4.31.1 [30] <§4.10> Draw a pipeline diagram showing how RISC-V code given above executes on the two-issue processor. Assume that the loop exits after two iterations.

4.31.2 [10] <§4.10> What is the speedup of going from a one-issue to a two-issue processor? (Assume the loop runs thousands of iterations.)

4.31.3 [10] <§4.10> Rearrange/rewrite the RISC-V code given above to achieve better performance on the one-issue processor. Hint: Use the instruction "`beqz x13,DONE`" to skip the loop entirely if `j = 0`.

4.31.4 [20] <§4.10> Rearrange/rewrite the RISC-V code given above to achieve better performance on the two-issue processor. (Do not unroll the loop, however.)

4.31.5 [30] <§4.10> Repeat Exercise 4.31.1, but this time use your optimized code from Exercise 4.31.4.

4.31.6 [10] <§4.10> What is the speedup of going from a one-issue processor to a two-issue processor when running the optimized code from Exercises 4.31.3 and 4.31.4.

4.31.7 [10] <§4.10> Unroll the RISC-V code from Exercise 4.31.3 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the one-issue processor. You may assume that `j` is a multiple of 4.

4.31.8 [20] <§4.10> Unroll the RISC-V code from Exercise 4.31.4 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the two-issue processor. You may assume that `j` is a multiple of 4. (Hint: Re-organize the loop so that some calculations appear both outside the loop and at the end of the loop. You may assume that the values in temporary registers are not needed after the loop.)

4.31.9 [10] <§4.10> What is the speedup of going from a one-issue processor to a two-issue processor when running the unrolled, optimized code from Exercises 4.31.7 and 4.31.8?

4.31.10 [30] <§4.10> Repeat Exercises 4.31.8 and 4.31.9, but this time assume the two-issue processor can run two arithmetic/logic instructions together. (In other words, the first instruction in a packet can be any type of instruction, but the second must be an arithmetic or logic instruction. Two memory operations cannot be scheduled at the same time.)

4.32 This exercise explores energy efficiency and its relationship with performance. Problems in this exercise assume the following energy consumption for activity in Instruction memory, Registers, and Data memory. You can assume that the other components of the datapath consume a negligible amount of energy. ("Register Read" and "Register Write" refer to the register file only.)

I-Mem	1 Register Read	Register Write	D-Mem Read	D-Mem Write
140pJ	70pJ	60pJ	140pJ	120pJ

Assume that components in the datapath have the following latencies. You can assume that the other components of the datapath have negligible latencies.

I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
200 ps	150 ps	90 ps	90 ps	250 ps

4.32.1 [5] <§§4.3, 4.6, 4.14> How much energy is spent to execute an `add` instruction in a single-cycle design and in the five-stage pipelined design?

4.32.2 [10] <§§4.6, 4.14> What is the worst-case RISC-V instruction in terms of energy consumption? What is the energy spent to execute it?

4.32.3 [10] <§§4.6, 4.14> If energy reduction is paramount, how would you change the pipelined design? What is the percentage reduction in the energy spent by an `ld` instruction after this change?

4.32.4 [10] <§§4.6, 4.14> What other instructions can potentially benefit from the change discussed in Exercise 4.32.3?

4.32.5 [10] <§§4.6, 4.14> How do your changes from Exercise 4.32.3 affect the performance of a pipelined CPU?

4.32.6 [10] <§§4.6, 4.14> We can eliminate the `MemRead` control signal and have the data memory be read in every cycle, i.e., we can permanently have `MemRead=1`. Explain why the processor still functions correctly after this change. If 25% of instructions are loads, what is the effect of this change on clock frequency and energy consumption?

4.33 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a "cross-talk fault". A special class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). These faults, where the affected signal always has a logical value of either 0 or 1 are called "stuck-at-0" or "stuck-at-1" faults. The following problems refer to bit 0 of the Write Register input on the register file in Figure 4.21.

4.33.1 [10] <§§4.3, 4.4> Let us assume that processor testing is done by (1) filling the PC, registers, and data and instruction memories with some values (you can choose which values), (2) letting a single instruction execute, then (3) reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-0 fault on this signal?

4.33.2 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1 for a stuck-at-1 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

4.33.3 [10] <§§4.3, 4.4> If we know that the processor has a stuck-at-1 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal RISC-V processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data.

4.33.4 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 0 if the `branch` control signal is 0, no fault otherwise.

4.33.5 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 1 if `RegRd` control signal is 1, no fault otherwise. Hint: This problem requires knowledge of operating systems. Consider what causes segmentation faults.

Answers to Check Yourself

§4.1, page 240: 3 of 5: Control, Datapath, Memory. Input and Output are missing.
§4.2, page 243: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.
§4.3, page 250: I. a. II. c.
§4.4, page 262: Yes, Branch and ALUOp0 are identical. In addition, you can use the flexibility of the don't care bits to combine other signals together. ALUSrc and MemtoReg can be made the same by setting the two don't care bits of MemtoReg to 1 and 0. ALUOp1 and MemtoReg can be made to be inverses of one another by setting the don't care bit of MemtoReg to 1. You don't need an inverter; simply use the other signal and flip the order of the inputs to the MemtoReg multiplexor!
§4.5, page 275: 1. Stall due to a load-use data hazard of the `ld` result. 2. Avoid stalling in the third instruction for the read-after-write data hazard on `x11` by forwarding the `add` result. 3. It need not stall, even without forwarding.
§4.6, page 288: Statements 2 and 4 are correct; the rest are incorrect.
§4.8, page 314: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.
§4.9, page 321: The first instruction, since it is logically executed before the others.
§4.10, page 334: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.
§4.12, page 344: First two are false and the last two are true.