4.1

4.1.1 The value of the signals is as follows:

RegWrite	ALUSrc	ALUoperation	MemWrite	MemRead	MemToReg
true	0	"and"	false	false	0

Mathematically, the `MemRead` control wire is a "don't care": the instruction will run correctly regardless of the chosen value. Practically, however, `MemRead` should be set to false to prevent causing a segment fault or cache miss.

4.1.2 Registers, ALUsrc mux, ALU, and the MemToReg mux.

4.1.3 All blocks produce some output. The outputs of DataMemory and Imm Gen are not used.

4.2 `Reg2Loc` for `ld`: When executing `ld`, it doesn't matter which value is passed to "Read register 2", because the ALUSrc mux ignores the resulting "Read data 2" output and selects the sign extended immediate value instead.

`MemToReg` for `sd` and `beq`: Neither `sd` nor `beq` write a value to the register file. It doesn't matter which value the MemToReg mux passes to the register file because the register file ignores that value.

4.3

4.3.1 25 + 10 = 35%. Only `Load` and `Store` use Data memory.

4.3.2 100% Every instruction must be fetched from instruction memory before it can be executed.

4.3.3 28 + 25 + 10 + 11 + 2 = 76%. Only R-type instructions do not use the Sign extender.

4.3.4 The sign extend produces an output during every cycle. If its output is not needed, it is simply ignored.

4.4

4.4.1 Only `loads` are broken. `MemToReg` is either 1 or "don't care" for all other instructions.

4.4.2 I-type, `loads`, `stores` are all broken.

4.5 For context: The encoded instruction is `sd x12, 20(x13)`

4.5.1

ALUop	ALU Control Lines
00	0010

4.5.2 The new PC is the old PC + 4. This signal goes from the PC, through the "PC + 4" adder, through the "branch" mux, and back to the PC.

4.5.3 `ALUsrc`: Inputs: `Reg[x12]` and `0x0000000000000014`; Output: `0x0000000000000014`

`MemToReg`: Inputs: `Reg[x13] + 0x14` and `<undefined>`; output: `<undefined>`

`Branch`: Inputs: `PC+4` and `0x000000000000000A`

4.5.4 ALU inputs: `Reg[x13]` and `0x0000000000000014`

PC + 4 adder inputs: `PC` and `4`

Branch adder inputs: `PC` and `0x0000000000000028`

4.6

4.6.1 No additional logic blocks are needed.

4.6.2
`Branch: false`
`MemRead: false` (See footnote from solution to problem 4.1.1.)
`MemToReg: 0`
`ALUop: 10` (or simply saying "add" is sufficient for this problem)
`MemWrite: false`
`ALUsrc: 1`
`RegWrite: 1`

4.7

4.7.1 `R-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.2 `ld`: 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950 ps

4.7.3 `sd`: 30 + 250 + 150 + 200 + 25 + 250 = 905

4.7.4 `beq`: 30 + 250 + 150 + 25 + 200 + 5 + 25 + 20 = 705

4.7.5 `I-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.6 950ps

4.8 Using the results from Problem 4.7, we see that the average time per instruction is

.52*700 + .25*950 + .11*905 + .12 * 705 = 785.6ps

In contrast, a single-cycle CPU with a "normal" clock would require a clock cycle time of 950.

Thus, the speedup would be 925/787.6 = 1.174

4.9

4.9.1 Without improvement: 950; With improvement: 1250

4.9.2 The running time of a program on the original CPU is 950*n. The running time on the improved CPU is 1250*(0.95)*n = 1187.5. Thus, the "speedup" is 0.8. (Thus, this "improved" CPU is actually slower than the original).

4.9.3 Because adding a multiply instruction will remove 5% of the instructions, the cycle time can grow to as much as 950/(0.95) = 1000. Thus, the time for the ALU can increase by up to 50 (from 200 to 250).

4.10

4.10.1 The additional registers will allow us to remove 12% of the loads and stores, or (0.12)*(0.25 + 0.1) = 4.2% of all instructions. Thus, the time to run n instructions will decrease from 950*n to 960*.958*n = 919.68*n. That corresponds to a speedup of 950/895.73 = 1.03.

4.10.2 The cost of the original CPU is 4507; the cost of the improved CPU is 4707.

PC: 5
I-Mem: 1000
Register file: 200
ALU: 100
D-Mem: 2000
Sign Extend: 1002
Controls: 10002
adders: 30*24
muxes: 4*102
single gates: 2*1

Thus, for a 3% increase in performance, the cost of the CPU increases by about 4.4%.

4.10.3 From a strictly mathematical standpoint it does not make sense to add more registers because the new CPU costs more per unit of performance. However, that simple calculation does not account for the utility of the performance. For example, in a real-time system, a 3% performance may make the difference between meeting or missing deadlines. In which case, the improvement would be well worth the 4.4% additional cost.

4.11

4.11.1 No new functional blocks are needed.

4.11.2 Only the control unit needs modification.

4.11.3 No new data paths are needed.

4.11.4 No new signals are needed.

4.12

4.12.1 No new functional blocks are needed.

4.12.2 The register file needs to be modified so that it can write to two registers in the same cycle. The ALU would also need to be modified to allow read data 1 or 2 to be passed through to write data 1.

4.12.3 The answer depends on the answer given in 4.12.2: whichever input was not allowed to pass through the ALU above must now have a data path to write data 2.

4.12.4 There would need to be a second RegWrite control wire.

4.12.5 Many possible solutions.

4.13

4.13.1 We need some additional muxes to drive the data paths discussed in 4.13.3.

4.13.2 No functional blocks need to be modified.

4.13.3 There needs to be a path from the ALU output to data memory's write data port. There also needs to be a path from read data 2 directly to Data memory's Address input.

4.13.4 These new data paths will need to be driven by muxes. These muxes will require control wires for the selector.

4.13.5 Many possible solutions.

4.14 None: all instructions that use sign extend also use the register file, which is slower.

4.15

4.15.1 The new clock cycle time would be 750. ALU and Data Memory will now run in parallel, so we have effectively removed the faster of the two (the ALU with time 200) from the critical path.

4.15.2 Slower. The original CPU takes 950*n picoseconds to run n instructions. The same program will have approximately 1.35*n instructions when compiled for the new machine. Thus, the time on the new machine will be 750*1.35n = 1012.5*n. This represents a "speedup" of .93.

4.15.3 The number of loads and stores is the primary factor. How the loads and stores are used can also have an effect. For example, a program whose loads and stores tend to be to only a few different address may also run faster on the new machine.

4.15.4 This answer is a matter of opinion.

4.16

4.16.1 Pipelined: 350; non-pipelined: 1250

4.16.2 Pipelined: 1250; non-pipelined: 1250

4.16.3 Split the ID stage. This reduces the clock-cycle time to 300ps.

4.16.4 35%.

4.16.5 65%

4.17 $n + k - 1$. Let's look at when each instruction is in the WB stage. In a k-stage pipeline, the 1st instruction doesn't enter the WB stage until cycle k. From that point on, at most one of the remaining $n - 1$ instructions is in the WB stage during every cycle.

This gives us a minimum of $k + (n - 1) = n + k - 1$ cycles.

4.18 x13 = 33 and x14 = 36

4.19 x15 = 54 (The code will run correctly because the result of the first instruction is written back to the register file at the beginning of the 5th cycle, whereas the final instruction reads the updated value of x1 during the second half of this cycle.)

4.20
```
addi x11, x12, 5
NOP
NOP
add x13, x11, x12
addi x14, x11, 15
NOP
add x15, x13, x12
```

4.21

4.21.1 Pipeline without forwarding requires 1.4*n*250ps. Pipeline with forwarding requires 1.05*n*300ps. The speedup is therefore (1.4*250)/(1.05*300) = 1.11.

4.21.2 Our goal is for the pipeline with forwarding to be faster than the pipeline without forwarding. Let y be the number of stalls remaining as a percentage of "code" instructions. Our goal is for 300*(1+y)*n < 250*1.4*n. Thus, y must be less than 16.7%.

4.21.3 This time, our goal is for 300(1 + y)*n < 250(1 + x)*n. This happens when y < (250x − 50)/300.

4.21.4 It cannot. In the best case, where forwarding eliminates the need for every NOP, the program will take time 300*n to run on the pipeline with forwarding. This is slower than the 250*1.075*n required on the pipeline with no forwarding.

4.21.5 Speedup is not possible when the solution to 4.21.3 is less than 0. Solving 0< (250x − 50)/300 for x gives that x must be at least 0.2.

4.22

4.22.1 Stalls are marked with **:

```
sd  x29, 12(x16)  IF ID EX ME WB
ld  x29, 8(x16)      IF ID EX ME WB
sub x17, x15, x14       IF ID EX ME WB
bez x17, label             ** ** IF ID EX ME WB
add x15, x11, x14                    IF ID EX ME WB
sub x15,x30,x14                         IF ID EX ME WB
```

4.22.2 Reordering code won't help. Every instruction must be fetched; thus, every data access causes a stall. Reordering code will just change the pair of instructions that are in conflict.

4.22.3 You can't solve this structural hazard with NOPs, because even the NOPs must be fetched from instruction memory.

4.22.4 35%. Every data access will cause a stall.

4.23

4.23.1 The clock period won't change because we aren't making any changes to the slowest stage.

4.23.2 Moving the MEM stage in parallel with the EX stage will eliminate the need for a cycle between loads and operations that use the result of the loads. This can potentially reduce the number of stalls in a program.

4.23.3 Removing the offset from `ld` and `sd` may increase the total number of instructions because some `ld` and `sd` instructions will need to be replaced with a `addi/ld` or `addi/sd` pair.

4.24 The second one. A careful examination of Figure 4.59 shows that the need for a stall is detected during the ID stage. It is this stage that prevents the fetch of a new instruction, effectively causing the add to repeat its ID stage.

4.25

4.25.1 … indicates a stall. ! indicates a stage that does not do useful work.

```
ld x10, 0(x13)     IF ID EX ME | WB
ld x11, 8(x13)        IF ID EX | ME WB
add x12, x10, x11        IF ID | .. EX ME! WB
addi x13, x13, -16          IF | .. ID EX  ME! WB
bnez x12, LOOP                 | .. IF ID  EX  ME! WB!
ld x10, 0(x13)                       IF  ID  EX  ME  WB
ld x11, 8(x13)                           IF  ID  EX  ME WB
add x12, x10, x11                            IF  ID  .. EX | ME! WB
addi x13, x13, -16                               IF  .. ID | EX  ME! WB
bnez x12, LOOP                                       IF | ID  EX  ME! WB!
Completely busy               | N  N  N   N   N   N   N  N  |
```

4.25.2 In a particular clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. As the diagram above shows, there are not any cycles during which every pipeline stage is doing useful work.

4.26

4.26.1
```
// EX to 1st only:
add x11, x12, x13
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only:
ld x11, 0(x12)
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only:
add x11, x12, x13
add x5, x6, x7
add x14, x11, x12

// MEM to 2nd only:
ld x11, 0(x12)
add x5, x6, x7
add x14, x11, x13

// EX to 1st and EX to 2nd:
add x11, x12, x13
add x5, x11, x15
add x16, x11, x12
```

4.26.2
```
// EX to 1st only: 2 nops
add x11, x12, x13
nop
nop
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only: 2 stalls
ld x11, 0(x12)
nop
nop
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only: 1 nop
add x11, x12, x13
add x5, x6, x7
nop
add x14, x11, x12
```

```
// MEM to 2nd only: 1 nop
ld x11, 0(x12)
add x5, x6, x7
nop
add x14, x11, x13

// EX to 1st and EX to 2nd: 2 nops
add x11, x12, x13
nop
nop
add x5, x11, x15
add x16, x11, x12
```

4.26.3 Consider this code:

```
ld  x11, 0(x5)       # MEM to 2nd --- one stall
add x12, x6, x7      # EX to 1st  --- two stalls
add x13, x11, x12
add x28, x29, x30
```

If we analyze each instruction separately, we would calculate that we need to add 3 stalls (one for a "MEM to 2nd" and two for an "EX to 1st only". However, as we can see below, we need only two stalls:

```
ld x11, 0(x5)
add  x12, x6, x7
nop
nop
add  x13, x11, x12
add  x28, x29, x30
```

4.26.4 Taking a weighted average of the answers from 4.26.2 gives 0.05*2 + 0.2*2 + 0.05*1 + 0.1*1 + 0.1*2 = 0.85 stalls per instruction (on average) for a CPI of 1.85. This means that 0.85/1.85 cycles, or 46%, are stalls.

4.26.5 The only dependency that cannot be handled by forwarding is from the MEM stage to the next instruction. Thus, 20% of instructions will generate one stall for a CPI of 1.2. This means that 0.2 out of 1.2 cycles, or 17%, are stalls.

4.26.6 If we forward from the EX/MEM register only, we have the following stalls/NOPs

EX to 1st:	0
MEM to 1st:	2
EX to 2nd:	1
MEM to 2nd:	1
EX to 1st and 2nd:	1

This represents an average of 0.05*0 + 0.2*2 + 0.05*1 + 0.10*1 + 0.10*1 = 0.65 stalls/instruction. Thus, the CPI is 1.65

IF we forward from MEM/WB only, we have the following stalls/NOPs

EX to 1st:	1
MEM to 1st:	1
EX to 2nd:	0
MEM to 2nd:	0
EX to 1st and 2nd:	1

This represents an average of 0.05*1 + 0.2*1 + 0.1*1 = 0.35 stalls/instruction. Thus, the CPI is 1.35.

4.26.7

	No forwarding	EX/MEM	MEM/WB	Full Forwarding
CPI	1.85	1.65	1.35	1.2
Period	120	120	1.20	130
Time	222n	198n	162n	156n
Speedup	–	1.12	1.37	1.42

4.26.8 CPI for full forwarding is 1.2
CPI for "time travel" forwarding is 1.0
clock period for full forwarding is 130
clock period for "time travel" forwarding is 230

Speedup = (1.2*130)/ (1*230) = 0.68 (That means that "time travel" forwarding actually slows the CPU.)

4.26.9 When considering the "EX/MEM" forwarding in 4.26.6, the "EX to 1st" generates no stalls, but "EX to 1st and EX to 2nd" generates one stall. However, "MEM to 1st" and "MEM to 1st and MEM to 2nd" will always generate the same number of stalls. (All "MEM to 1st" dependencies cause a stall, regardless of the type of forwarding. This stall causes the 2nd instruction's ID phase to overlap with the base instruction's WB phase, in which case no forwarding is needed.)

4.27

4.27.1
```
add   x15, x12, x11
nop
nop
ld    x13, 4(x15)
ld    x12, 0(x2)
nop
or    x13, x15, x13
nop
nop
sd    x13, 0(x15)
```

4.27.2 It is not possible to reduce the number of NOPs.

4.27.3 The code executes correctly. We need hazard detection only to insert a stall when the instruction following a load uses the result of the load. That does not happen in this case.

4.27.4

Cycle	1	2	3	4	5	6	7	8	
add	IF	ID	EX	ME	WB				
ld		IF	ID	EX	ME	WB			
ld			IF	ID	EX	ME	WB		
or				IF	ID	EX	ME	WB	
sd					IF	ID	EX	ME	WB

Because there are no stalls in this code, PCWrite and IF/IDWrite are always 1 and the mux before ID/EX is always set to pass the control values through.

(1) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(2) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(3) ForwardA = 0; ForwardB = 0 (no forwarding; values taken from registers)

(4) ForwardA = 2; ForwardB = 0 (base register taken from result of previous instruction)

(5) ForwardA = 1; ForwardB = 1 (base reguster taken from result of two instructions previous)

(6) ForwardA = 0; ForwardB = 2 (rs1 = x15 taken from register; rs2 = x13 taken from result of 1st ld—two instructions ago)

(7) ForwardA = 0; ForwardB = 2 (base register taken from register file. Data to be written taken from previous instruction)

4.27.5 The hazard detection unit additionally needs the values of rd that comes out of the MEM/WB register. The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by (or forwarded from) the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. The Hazard unit already has the value of `rd` from the EX/MEM register as inputs, so we need only add the value from the MEM/WB register.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

The value of `rd` from EX/MEM is needed to detect the data hazard between the `add` and the following `ld`. The value of `rd` form MEM/WB is needed to detect the data hazard between the first `ld` instruction and the `or` instruction.

4.27.6

Cycle	1	2	3	4	5	6
add	IF	ID	EX	ME	WB	
ld		IF	ID	–	–	EX
ld			IF	–	–	ID

(1) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(2) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(3) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(4) PCWrite = 0; IF/IDWrite = 0; control mux = 1

(5) PCWrite = 0; IF/IDWrite = 0; control mux = 1

4.28

4.28.1 The CPI increases from 1 to 1.4125.

An incorrectly predicted branch will cause three instructions to be flushed: the instructions currently in the IF, ID, and EX stages. (At this point, the branch instruction reaches the MEM stage and updates the PC with the correct next instruction.) In other words, 55% of the branches will result in the flushing of three instructions, giving us a CPI of $1 + (1 - 0.45)(0.25)3 = 1.4125$. (Just to be clear: the always-taken predictor is correct 45% of the time, which means, of course, that it is incorrect $1 - 0.45 = 55\%$ of the time.)

4.28.2 The CPI increases from 1 to 1.3375. $(1 + (.25)(1 - .55) = 1.1125)$

4.28.3 The CPI increases from 1 to 1.1125. $(1 + (.25)(1 - .85) = 1.0375)$

4.28.4 The speedup is approximately 1.019.

Changing half of the branch instructions to an ALU instruction reduces the percentage of instructions that are branches from 25% to 12.5%. Because predicted and mispredicted branches are replaced equally, the misprediction rate remains 15%. Thus, the new CPU is $1 + (.125)(1 - .85) = 1.01875$. This represents a speedup of 1.0375 / 1.01875 = 1.0184

4.28.5 The "speedup" is .91.

There are two ways to look at this problem. One way is to look at the two ADD instruction as a branch with an "extra" cycle. Thus, half of the branches have 1 extra cycle; 15% of the other half have 1 extra cycles (the pipeline flush); and the remaining branches (those correctly predicted) have no extra cycles. This gives us a CPI of $1 + (.5)(.25)*1 + (.5)(.25)(.15)*1 = 1.14375$ and a speedup of 1.0375 / 1.14375 = .91.

We can also treat the ADD instructions as separate instructions. The modified program now has 1.125n instructions (half of 25% produce

an extra instruction). .125n of these 1.125n instruction (or 11.1%) are branches. The CPI for this new program is 1 + (.111)(.15)*1 = 1.01665. When we factor in the 12.5% increase in instructions, we get a speedup of 1.0375 / (1.125 * 1.01665) = .91.

4.28.6 The predictor is 25% accurate on the remaining branches. We know that 80% of branches are always predicted correctly and the overall accuracy is 0.85. Thus, 0.8*1 + 0.2*x = 0.85. Solving for x shows that x = 0.25.

4.29

4.29.1

Always Taken	Always not-taken
3/5 = 60%	2/5 = 40%

4.29.2

Outcomes	Predictor value at time of prediction	Correct of Incorrect	Accuracy
T, NT, T, T	0,1,0,1	I,C,I,I	25%

4.29.3 The first few recurrences of this pattern do not have the same accuracy as the later ones because the predictor is still warming up. To determine the accuracy in the "steady state", we must work through the branch predictions until the predictor values start repeating (i.e. until the predictor has the same value at the start of the current and the next recurrence of the pattern).

Outcomes	Predictor value at time of prediction	Correct of Incorrect (in steady state)	Accuracy
T, NT, T, T, NT	1st occurrence: 0,1,0,1,2 2nd occurrence: 1,2,1,2,3 3rd occurrence: 2,3,2,3,3 4th occurrence: 2,3,2,3,3	C,I,C,C,I	60%

4.29.4 The predictor should be an N-bit shift register, where N is the number of branch outcomes in the target pattern. The shift register should be initialized with the pattern itself (0 for NT, 1 for T), and the prediction is always the value in the leftmost bit of the shift register. The register should be shifted after each predicted branch.

4.29.5 Since the predictor's output is always the opposite of the actual outcome of the branch instruction, the accuracy is zero.

4.29.6 The predictor is the same as in part d, except that it should compare its prediction to the actual outcome and invert (logical NOT) all the bits in the shift register if the prediction is incorrect. This predictor still always perfectly predicts the given pattern. For the opposite pattern, the first prediction will be incorrect, so the predictor's state is inverted and after

that the predictions are always correct. Overall, there is no warm-up period for the given pattern, and the warm-up period for the opposite pattern is only one branch.

4.30

4.30.1

Instruction 1	Instruction 2
Invalid target address (EX)	Invalid data address (MEM)

4.30.2 The Mux that selects the next PC must have inputs added to it. Each input is a constant address of an exception handler. The exception detectors must be added to the appropriate pipeline stage and the outputs of these detectors must be used to control the pre-PC Mux, and also to convert to NOPs instructions that are already in the pipeline behind the exception-triggering instruction.

4.30.3 Instructions are fetched normally until the exception is detected. When the exception is detected, all instructions that are in the pipeline after the first instruction must be converted to NOPs. As a result, the second instruction never completes and does not affect pipeline state. In the cycle that immediately follows the cycle in which the exception is detected, the processor will fetch the first instruction of the exception handler.

4.30.4 This approach requires us to fetch the address of the handler from memory. We must add the code of the exception to the address of the exception vector table, read the handler's address from memory, and jump to that address. One way of doing this is to handle it like a special instruction that puts the address in EX, loads the handler's address in MEM, and sets the PC in WB.

4.30.5 We need a special instruction that allows us to move a value from the (exception) Cause register to a general-purpose register. We must first save the general-purpose register (so we can restore it later), load the Cause register into it, add the address of the vector table to it, use the result as an address for a load that gets the address of the right exception handler from memory, and finally jump to that handler.

4.31.1

```
                    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
li x12, 0           IF ID EX ME WB
jal ENT             IF ID .. EX ME WB

bne x12, x13, TOP      IF .. ID EX ME WB
slli x5, x12, 3        IF .. ID .. EX ME WB

add x6, x10, x5              IF .. ID EX ME WB
ld x7, 0(x6)                 IF .. ID .. EX ME WB

ld x29, 8(x6)                      IF .. ID EX ME WB
sub x30, x7, x29                   IF .. ID .. .. EX ME WB

add x31, x11, x5                         IF .. .. ID EX ME WB
sd x30, 0(x31)                           IF .. .. ID .. EX ME WB

addi x12, x12, 2                                 IF .. ID EX ME WB
bne x12, x13, TOP                                IF .. ID .. EX ME WB

slli x5, x12, 3                                        IF .. ID EX ME WB
add x6, x10, x5                                        IF .. ID .. EX ME WB

ld x7, 0(x6)                                                 IF .. ID EX ME WB
ld x29, 8(x6)                                                IF .. ID .. EX ME WB

sub x30, x7, x29                                                   IF .. ID .. EX ME WB
add x31, x11, x5                                                   IF .. ID .. .. EX ME WB

sd x30, 0(x31)                                                           IF .. .. ID EX ME WB
addi x12, x12, 2                                                         IF .. .. ID .. EX ME WB

bne x12, x13, TOP                                                                  IF .. ID EX ME WB
slli x5, x12, 3                                                                    IF .. ID .. EX ME WB
```