# 4

# Solutions

## 4.1

**4.1.1** The values of the signals are as follows:

| RegWrite | MemRead | ALUMux | MemWrite | ALUop | RegMux | Branch |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 (Imm) | 1 | ADD | X | 0 |

ALUMux is the control signal that controls the Mux at the ALU input, 0 (Reg) selects the output of the register file, and 1 (Imm) selects the immediate from the instruction word as the second input to the ALU.

RegMux is the control signal that controls the Mux at the Data input to the register file, 0 (ALU) selects the output of the ALU, and 1 (Mem) selects the output of memory.

A value of X is a "don't care" (does not matter if signal is 0 or 1)

**4.1.2** All except branch Add unit and write port of the Registers

**4.1.3** Outputs that are not used: Branch Add, write port of Registers

No outputs: None (all units produce outputs)

## 4.2

**4.2.1** This instruction uses instruction memory, both register read ports, the ALU to add Rd and Rs together, data memory, and write port in Registers.

**4.2.2** None. This instruction can be implemented using existing blocks.

**4.2.3** None. This instruction can be implemented without adding new control signals. It only requires changes in the Control logic.

## 4.3

**4.3.1** Clock cycle time is determined by the critical path, which for the given latencies happens to be to get the data value for the load instruction: I-Mem (read instruction), Regs (takes longer than Control), Mux (select ALU input), ALU, Data Memory, and Mux (select value from memory to be written into Registers). The latency of this path is 400 ps + 200 ps + 30 ps + 120 ps + 350 ps + 30 ps = 1130 ps. 1430 ps (1130 ps + 300 ps, ALU is on the critical path).

**4.3.2** The speedup comes from changes in clock cycle time and changes to the number of clock cycles we need for the program: We need 5% fewer cycles for a program, but cycle time is 1430 instead of 1130, so we have a speedup of (1/0.95)*(1130/1430) = 0.83, which means we actually have a slowdown.

**4.3.3** The cost is always the total cost of all components (not just those on the critical path, so the original processor has a cost of I-Mem, Regs, Control, ALU, D-Mem, 2 Add units and 3 Mux units, for a total cost of $1000 + 200 + 500 + 100 + 2000 + 2*30 + 3*10 = 3890$.

We will compute cost relative to this baseline. The performance relative to this baseline is the speedup we previously computed, and our cost/performance relative to the baseline is as follows:

New Cost: $3890 + 600 = 4490$

Relative Cost: $4490/3890 = 1.15$

Cost/Performance: $1.15/0.83 = 1.39$. We are paying significantly more for significantly worse performance; the cost/performance is a lot worse than with the unmodified processor.

## 4.4

**4.4.1** I-Mem takes longer than the Add unit, so the clock cycle time is equal to the latency of the I-Mem:

200 ps

**4.4.2** The critical path for this instruction is through the instruction memory, Sign-extend and Shift-left-2 to get the offset, Add unit to compute the new PC, and Mux to select that value instead of PC+4. Note that the path through the other Add unit is shorter, because the latency of I-Mem is longer that the latency of the Add unit. We have:

200 ps + 15 ps + 10 ps + 70 ps + 20 ps = 315 ps

**4.4.3** Conditional branches have the same long-latency path that computes the branch address as unconditional branches do. Additionally, they have a long-latency path that goes through Registers, Mux, and ALU to compute the PCSrc condition. The critical path is the longer of the two, and the path through PCSrc is longer for these latencies:

200 ps + 90 ps + 20 ps + 90 ps + 20 ps = 420 ps

**4.4.4** PC-relative branches.

**4.4.5** PC-relative unconditional branch instructions. We saw in part c that this is not on the critical path of conditional branches, and it is only needed for PC-relative branches. Note that MIPS does not have actual unconditional branches (bne zero,zero,Label plays that role so there is no need for unconditional branch opcodes) so for MIPS the answer to this question is actually "None".

**4.4.6** Of the two instructions (BNE and ADD), BNE has a longer critical path so it determines the clock cycle time. Note that every path for ADD is shorter than or equal to the corresponding path for BNE, so changes in unit latency will not affect this. As a result, we focus on how the unit's latency affects the critical path of BNE.

This unit is not on the critical path, so the only way for this unit to become critical is to increase its latency until the path for address computation through sign extend, shift left, and branch add becomes longer than the path for PCSrc through registers, Mux, and ALU. The latency of Regs, Mux, and ALU is 200 ps and the latency of Sign-extend, Shift-left-2, and Add is 95 ps, so the latency of Shift-left-2 must be increased by 105 ps or more for it to affect clock cycle time.

**4.5**

**4.5.1** The data memory is used by LW and SW instructions, so the answer is:

25% + 10% = 35%

**4.5.2** The sign-extend circuit is actually computing a result in every cycle, but its output is ignored for ADD and NOT instructions. The input of the sign-extend circuit is needed for ADDI (to provide the immediate ALU operand), BEQ (to provide the PC-relative offset), and LW and SW (to provide the offset used in addressing memory) so the answer is:

20% + 25% + 25% + 10% = 80%

**4.6**

**4.6.1** To test for a stuck-at-0 fault on a wire, we need an instruction that puts that wire to a value of 1 and has a different result if the value on the wire is stuck at zero:

If this signal is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. So if we place a value of zero in R30 and a value of 1 in R31, and then execute ADD R31,R30,R30 the value of R31 is supposed to be zero. If bit 0 of the Write Register input to the Registers unit is stuck at zero, the value is written to R30 instead and R31 will be 1.

**4.6.2** The test for stuck-at-zero requires an instruction that sets the signal to 1, and the test for stuck-at-1 requires an instruction that sets the signal to 0. Because the signal cannot be both 0 and 1 in the same cycle, we cannot test the same signal simultaneously for stuck-at-0 and stuck-at-1 using only one instruction. The test for stuck-at-1 is analogous to the stuck-at-0 test:

We can place a value of zero in R31 and a value of 1 in R30, then use ADD R30,R31,R31 which is supposed to place 0 in R30. If this signal is stuck-at-1, the write goes to R31 instead, so the value in R30 remains 1.

**4.6.3** We need to rewrite the program to use only odd-numbered registers.

**4.6.4** To test for this fault, we need an instruction whose MemRead is 1, so it has to be a load. The instruction also needs to have RegDst set to 0, which is the case for loads. Finally, the instruction needs to have a different result if

MemRead is set to 0. For a load, MemRead=0 result in not reading memory, so the value placed in the register is "random" (whatever happened to be at the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.

**4.6.5** To test for this fault, we need an instruction whose Jump is 1, so it has to be the jump instruction. However, for the jump instruction the RegDst signal is "don't care" because it does not write to any registers, so the implementation may or may not allow us to set RegDst to 0 so we can test for this fault. As a result, we cannot reliably test for this fault.

## 4.7

### 4.7.1

| Sign-extend | Jump's shift-left-2 |
|---|---|
| 00000000000000000000000000010100 | 0001100010000000000001010000 |

### 4.7.2

| ALUOp[1-0] | Instruction[5-0] |
|---|---|
| 00 | 010100 |

### 4.7.3

| New PC | Path |
|---|---|
| PC+4 | PC to Add (PC+4) to branch Mux to jump Mux to PC |

### 4.7.4

| WrReg Mux | ALU Mux | Mem/ALU Mux | Branch Mux | Jump Mux |
|---|---|---|---|---|
| 2 or 0 (RegDst is X) | 20 | X | PC+4 | PC+4 |

### 4.7.5

| ALU | Add (PC+4) | Add (Branch) |
|---|---|---|
| -3 and 20 | PC and 4 | PC+4 and 20*4 |

### 4.7.6

| | | | | |
|---|---|---|---|---|
| Read Register 1 | Read Register 2 | Write Register | Write Data | RegWrite |

## 4.8

### 4.8.1

| Pipelined | Single-cycle |
|---|---|
| 350 ps | 1250 ps |

### 4.8.2

| Pipelined | Single-cycle |
|---|---|
| 1750 ps | 1250 ps |

### 4.8.3

| Stage to split | New clock cycle time |
|---|---|
| ID | 300 ps |

**4.8.4**

| | |
|---|---|
| a. | 35% |

**4.8.5**

| | |
|---|---|
| a. | 65% |

**4.8.6** We already computed clock cycle times for pipelined and single cycle organizations, and the multi-cycle organization has the same clock cycle time as the pipelined organization. We will compute execution times relative to the pipelined organization. In single-cycle, every instruction takes one (long) clock cycle. In pipelined, a long-running program with no pipeline stalls completes one instruction in every cycle. Finally, a multi-cycle organization completes a LW in 5 cycles, a SW in 4 cycles (no WB), an ALU instruction in 4 cycles (no MEM), and a BEQ in 4 cycles (no WB). So we have the speedup of pipeline

| | Multi-cycle execution time is X times pipelined execution time, where X is: | Single-cycle execution time is X times pipelined execution time, where X is: |
|---|---|---|
| a. | 0.20*5+0.80*4=4.20 | 1250 ps/350 ps=3.57 |

**4.9**

**4.9.1**

| Instruction sequence | Dependences |
|---|---|
| `I1: OR R1,R2,R3` | RAW on R1 from I1 to I2 and I3 |
| `I2: OR R2,R1,R4` | RAW on R2 from I2 to I3 |
| `I3: OR R1,R1,R2` | WAR on R2 from I1 to I2 |
| | WAR on R1 from I2 to I3 |
| | WAW on R1 from I1 to I3 |

**4.9.2** In the basic five-stage pipeline WAR and WAW dependences do not cause any hazards. Without forwarding, any RAW dependence between an instruction and the next two instructions (if register read happens in the second half of the clock cycle and the register write happens in the first half). The code that eliminates these hazards by inserting NOP instructions is:

| Instruction sequence | |
|---|---|
| `OR R1,R2,R3` | |
| `NOP` | Delay I2 to avoid RAW hazard on R1 from I1 |
| `NOP` | |
| `OR R2,R1,R4` | |
| `NOP` | Delay I3 to avoid RAW hazard on R2 from I2 |
| `NOP` | |
| `OR R1,R1,R2` | |

**4.9.3** With full forwarding, an ALU instruction can forward a value to EX stage of the next instruction without a hazard. However, a load cannot forward to the EX stage of the next instruction (by can to the instruction after that). The code that eliminates these hazards by inserting NOP instructions is:

| Instruction sequence | |
|---|---|
| `OR R1,R2,R3` | |
| `OR R2,R1,R4` | No RAW hazard on R1 from I1 (forwarded) |
| `OR R1,R1,R2` | No RAW hazard on R2 from I2 (forwarded) |

**4.9.4** The total execution time is the clock cycle time times the number of cycles. Without any stalls, a three-instruction sequence executes in 7 cycles (5 to complete the first instruction, then one per instruction). The execution without forwarding must add a stall for every NOP we had in 4.9.2, and execution forwarding must add a stall cycle for every NOP we had in 4.9.3. Overall, we get:

| No forwarding | With forwarding | Speedup due to forwarding |
|---|---|---|
| (7 + 4)*180 ps = 1980 ps | 7*240 ps = 1680 ps | 1.18 |

**4.9.5** With ALU-ALU-only forwarding, an ALU instruction can forward to the next instruction, but not to the second-next instruction (because that would be forwarding from MEM to EX). A load cannot forward at all, because it determines the data value in MEM stage, when it is too late for ALU-ALU forwarding. We have:

| Instruction sequence | |
|---|---|
| `OR R1,R2,R3` | |
| `OR R2,R1,R4` | ALU-ALU forwarding of R1 from I1 |
| `OR R1,R1,R2` | ALU-ALU forwarding of R2 from I2 |

**4.9.6**

| No forwarding | With ALU-ALU forwarding only | Speedup with ALU-ALU forwarding |
|---|---|---|
| (7 + 4)*180 ps = 1980 ps | 7*210 ps = 1470 ps | 1.35 |

## 4.10

**4.10.1** In the pipelined execution shown below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

| Instruction | Pipeline Stage | Cycles |
|---|---|---|
| `SW R16,12(R6)` | `IF ID EX MEM WB` | 11 |
| `LW R16,8(R6)` | `   IF ED EX MEM WB` | |
| `BEQ R5,R4,Lbl` | `      IF ID EX MEM WB` | |
| `ADD R5,R1,R4` | `         *** *** IF ID EX MEM WB` | |
| `SLT R5,R15,R4` | `                    IF ID EX MEM WB` | |

We can not add NOPs to the code to eliminate this hazard – NOPs need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

**4.10.2** This change only saves one cycle in an entire execution without data hazards (such as the one given). This cycle is saved because the last instruction finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

| Instructions Executed | Cycles with 5 stages | Cycles with 4 stages | Speedup |
|---|---|---|---|
| 5 | 4 + 5 = 9 | 3 + 5 = 8 | 9/8 = 1.13 |

**4.10.3** Stall-on-branch delays the fetch of the next instruction until the branch is executed. When branches execute in the EXE stage, each branch causes two stall cycles. When branches execute in the ID stage, each branch only causes one stall cycle. Without branch stalls (e.g., with perfect branch prediction) there are no stalls, and the execution time is 4 plus the number of executed instructions. We have:

| Instructions Executed | Branches Executed | Cycles with branch in EXE | Cycles with branch in ID | Speedup |
|---|---|---|---|---|
| 5 | 1 | 4 + 5 + 1*2 = 11 | 4 + 5 + 1*1 = 10 | 11/10 = 1.10 |

**4.10.4** The number of cycles for the (normal) 5-stage and the (combined EX/MEM) 4-stage pipeline is already computed in 4.10.2. The clock cycle time is equal to the latency of the longest-latency stage. Combining EX and MEM stages affects clock time only if the combined EX/MEM stage becomes the longest-latency stage:

| Cycle time with 5 stages | Cycle time with 4 stages | Speedup |
|---|---|---|
| 200 ps (IF) | 210 ps (MEM + 20 ps) | (9*200)/(8*210) = 1.07 |

**4.10.5**

| New ID latency | New EX latency | New cycle time | Old cycle time | Speedup |
|---|---|---|---|---|
| 180 ps | 140 ps | 200 ps (IF) | 200 ps (IF) | (11*200)/(10*200) = 1.10 |

**4.10.6** The cycle time remains unchanged: a 20 ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change does affect execution time because it adds one additional stall cycle to each branch. Because the clock cycle time does not improve but the number of cycles increases, the speedup from this change will be below 1 (a slowdown). In 4.10.3 we already computed the number of cycles when branch is in EX stage. We have:

| | Cycles with branch in EX | Execution time (branch in EX) | Cycles with branch in MEM | Execution time (branch in MEM) | Speedup |
|---|---|---|---|---|---|
| a. | 4 + 5 + 1*2 = 11 | 11*200 ps = 2200 ps | 4 + 5 + 1*3 = 12 | 12*200 ps = 2400 ps | 0.92 |

## 4.11

### 4.11.1

```
LW  R1,0(R1)     WB
LW  R1,0(R1)     EX  MEM WB
BEQ R1,R0,Loop   ID  *** EX  MEM WB
LW  R1,0(R1)     IF  *** ID  EX  MEM WB
AND R1,R1,R2             IF  ID  *** EX  MEM WB
LW  R1,0(R1)                 IF  *** ID  EX  MEM
LW  R1,0(R1)                         IF  ID  ***
BEQ R1,R0,Loop                           IF  ***
```

**4.11.2** 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. In the pipeline execution diagram from 4.11.1, a stage is stalled if its name is not shown for a particular cycles, and stages in which the particular instruction is not doing useful work are marked in blue. Note that a BEQ instruction is doing useful work in the MEM stage, because it is determining the correct value of the next instruction's PC in that stage. We have:

| Cycles per loop iteration | Cycles in which all stages do useful work | % of cycles in which all stages do useful work |
|---|---|---|
| 8 | 0 | 0% |

## 4.12

**4.12.1** Dependences to the 1st next instruction result in 2 stall cycles, and the stall is also 2 cycles if the dependence is to both 1st and 2nd next instruction. Dependences to only the 2nd next instruction result in one stall cycle. We have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.35*2 + 0.15*1 = 1.85 | 46% (0.85/1.85) |

**4.12.2** With full forwarding, the only RAW data dependences that cause stalls are those from the MEM stage of one instruction to the 1st next instruction. Even this dependences causes only one stall cycle, so we have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.20 = 1.20 | 17% (0.20/1.20) |

**4.12.3** With forwarding only from the EX/MEM register, EX to 1st dependences can be satisfied without stalls but any other dependences (even when together with EX to 1st) incur a one-cycle stall. With forwarding only from the MEM/WB register, EX to 2nd dependences incur no stalls. MEM to 1st dependences still incur a one-cycle stall, and EX to 1st dependences now incur one stall cycle because we must wait for the instruction to complete the MEM stage to be able to forward to the next instruction. We compute stall cycles per instructions for each case as follows:

| EX/MEM | MEM/WB | Fewer stall cycles with |
|---|---|---|
| 0.2 + 0.05 + 0.1 + 0.1 = 0.45 | 0.05 + 0.2 + 0.1 = 0.35 | MEM/WB |

**4.12.4** In 4.12.1 and 4.12.2 we have already computed the CPI without forwarding and with full forwarding. Now we compute time per instruction by taking into account the clock cycle time:

| Without forwarding | With forwarding | Speedup |
|---|---|---|
| 1.85*150 ps = 277.5 ps | 1.20*150 ps = 180 ps | 1.54 |

**4.12.5** We already computed the time per instruction for full forwarding in 4.12.4. Now we compute time-per instruction with time-travel forwarding and the speedup over full forwarding:

| With full forwarding | Time-travel forwarding | Speedup |
|---|---|---|
| 1.20*150 ps = 180 ps | 1*250 ps = 250 ps | 0.72 |

**4.12.6**

| EX/MEM | MEM/WB | Shorter time per instruction with |
|---|---|---|
| 1.45*150 ps = 217.5 | 1.35*150 ps = 202.5 ps | MEM/WB |

**4.13**

**4.13.1**

```
ADD R5,R2,R1
NOP
NOP
LW R3,4(R5)
LW R2,0(R2)
NOP
OR R3,R5,R3
NOP
NOP
SW R3,0(R5)
```

**4.13.2** We can move up an instruction by swapping its place with another instruction that has no dependences with it, so we can try to fill some NOP slots with such instructions. We can also use R7 to eliminate WAW or WAR dependences so we can have more instructions to move up.

| | |
|---|---|
| `I1: ADD R5,R2,R1` | |
| `I3: LW R2,0(R2)` | Moved up to fill NOP slot |
| `NOP` | |
| `I2: LW R3,4(R5)` | |
| `NOP` | Had to add another NOP here, |
| `NOP` | so there is no performance gain |
| `I4: OR R3,R5,R3` | |
| `NOP` | |
| `NOP` | |
| `I5: SW R3,0(R5)` | |

**4.13.3** With forwarding, the hazard detection unit is still needed because it must insert a one-cycle stall whenever the load supplies a value to the instruction that immediately follows that load. Without the hazard detection unit, the instruction that depends on the immediately preceding load gets the stale value the register had before the load instruction.

| |
|---|
| Code executes correctly (for both loads, there is no RAW dependence between the load and the next instruction). |

**4.13.4** The outputs of the hazard detection unit are PCWrite, IF/IDWrite, and ID/EXZero (which controls the Mux after the output of the Control unit). Note that IF/IDWrite is always equal to PCWrite, and ED/ExZero is always the opposite of PCWrite. As a result, we will only show the value of PCWrite for each cycle. The outputs of the forwarding unit is ALUin1 and ALUin2, which control Muxes that select the first and second input of the ALU. The three possible values for ALUin1 or ALUin2 are 0 (no forwarding), 1 (forward ALU output from previous instruction), or 2 (forward data value for second-previous instruction). We have:

| Instruction sequence | First five cycles<br>1 2 3 4 5 | Signals |
|---|---|---|
| `ADD R5,R2,R1` | IF ID EX MEM WB | 1: PCWrite=1, ALUin1=X, ALUin2=X |
| `LW R3,4(R5)` | IF ID EX MEM | 2: PCWrite=1, ALUin1=X, ALUin2=X |
| `LW R2,0(R2)` | IF ID EX | 3: PCWrite=1, ALUin1=0, ALUin2=0 |
| `OR R3,R5,R3` | IF ID | 4: PCWrite=1, ALUin1=1, ALUin2=0 |
| `SW R3,0(R5)` | IF | 5: PCWrite=1, ALUin1=0, ALUin2=0 |

**4.13.5** The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. For the instruction in the EX stage, we need to check Rd for R-type instructions and Rd for loads. For the instruction in the MEM stage, the destination register is already selected (by the Mux in the EX stage) so we need to check that register number (this is the bottommost output of the EX/MEM pipeline register). The additional inputs to the hazard detection unit are register Rd from the ID/EX pipeline register and the output number of the output register from the EX/MEM pipeline register. The Rt field from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

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

**4.13.6** As explained for part e, we only need to specify the value of the PCWrite signal, because IF/IDWrite is equal to PCWrite and the ID/EXzero signal is its opposite. We have:

| Instruction sequence | First five cycles 1 | 2 | 3 | 4 | 5 | Signals |
|---|---|---|---|---|---|---|
| ADD R5,R2,R1 | IF | ID | EX | MEM | WB | 1: PCWrite=1 |
| LW R3,4(R5) | | IF | ID | *** | *** | 2: PCWrite=1 |
| LW R2,0(R2) | | | IF | *** | *** | 3: PCWrite=1 |
| OR R3,R5,R3 | | | | | *** | 4: PCWrite=0 |
| SW R3,0(R5) | | | | | | 5: PCWrite=0 |

## 4.14

### 4.14.1

| Executed Instructions | Pipeline Cycles 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW R2,0(R1) | IF | ID | EX | MEM | WB | | | | | | | | | |
| BEQ R2,R0,Label2 (NT) | | IF | ID | *** | EX | MEM | WB | | | | | | | |
| LW R3,0(R2) | | | | | | IF | ID | EX | MEM | WB | | | | |
| BEQ R3,R0,Label1 (T) | | | | | | | IF | ID | *** | EX | MEM | WB | | |
| BEQ R2,R0,Label2 (T) | | | | | | | | IF | *** | ID | EX | MEM | WB | |
| SW R1,0(R2) | | | | | | | | | | IF | ID | EX | MEM | WB |

### 4.14.2

| Executed Instructions | Pipeline Cycles 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW R2,0(R1) | IF | ID | EX | MEM | WB | | | | | | | | | |
| BEQ R2,R0,Label2 (NT) | | IF | ID | *** | EX | MEM | WB | | | | | | | |
| LW R3,0(R2) | | | IF | *** | ID | EX | MEM | WB | | | | | | |
| BEQ R3,R0,Label1 (T) | | | | | | IF | ID | EX | MEM | WB | | | | |
| ADD R1,R3,R1 | | | | | | | IF | ID | EX | MEM | WB | | | |
| BEQ R2,R0,Label2 (T) | | | | | | | | IF | ID | EX | MEM | WB | | |
| LW R3,0(R2) | | | | | | | | | IF | ID | EX | MEM | WB | |
| SW R1,0(R2) | | | | | | | | | | IF | ID | EX | MEM | WB |

### 4.14.3

```
        LW R2,0(R1)
Label1: BEZ R2,Label2 ; Not taken once, then taken
        LW R3,0(R2)
        BEZ R3,Label1 ; Taken
        ADD R1,R3,R1
Label2: SW R1,0(R2)
```

**4.14.4** The hazard detection logic must detect situations when the branch depends on the result of the previous R-type instruction, or on the result of two previous loads. When the branch uses the values of its register operands in its ID stage, the R-type instruction's result is still being generated in the EX stage. Thus we must stall the processor and repeat the ID stage of the branch in the next cycle. Similarly, if the branch depends on a load that immediately precedes it, the result of the load is only generated two cycles after the branch enters the ID stage, so we must stall the branch for two cycles. Finally, if the branch depends on a load that is the second-previous instruction, the load is completing its MEM stage when the branch is in its ID stage, so we must stall the branch for one cycle. In all three cases, the hazard is a data hazard.

Note that in all three cases we assume that the values of preceding instructions are forwarded to the ID stage of the branch if possible.

**4.14.5** For part a we have already shows the pipeline execution diagram for the case when branches are executed in the EX stage. The following is the pipeline diagram when branches are executed in the ID stage, including new stalls due to data dependences described for part d:

| Executed Instructions | Pipeline Cycles 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW R2,0(R1) | IF | ID | EX | MEM | WB | | | | | | | | | | |
| BEQ R2,R0,Label2 (NT) | | IF | *** | *** | ID | EX | MEM | WB | | | | | | | |
| LW R3,0(R2) | | | | | | IF | ID | EX | MEM | WB | | | | | |
| BEQ R3,R0,Label1 (T) | | | | | | | IF | *** | *** | ID | EX | MEM | WB | | |
| BEQ R2,R0,Label2 (T) | | | | | | | | | | IF | ID | EX | MEM | WB | |
| SW R1,0(R2) | | | | | | | | | | | IF | ID | EX | MEM | WB |

Now the speedup can be computed as:

$\boxed{14/15 = 0.93}$

**4.14.6** Branch instructions are now executed in the ID stage. If the branch instruction is using a register value produced by the immediately preceding instruction, as we described for part d the branch must be stalled because the preceding instruction is in the EX stage when the branch is already using the stale register values in the ID stage. If the branch in the ID stage depends on an R-type instruction that is in the MEM stage, we need forwarding to ensure correct execution of the branch. Similarly, if the branch in the ID stage depends on an R-type of load instruction in the WB stage, we need forwarding to ensure correct execution of the branch. Overall, we need another forwarding unit that takes the same inputs as the one that forwards to the EX stage. The new forwarding unit should control two Muxes placed right before the branch comparator. Each Mux selects between the value read from Registers, the ALU output from the EX/MEM pipeline register, and the data value from the MEM/WB pipeline register. The complexity of the new forwarding unit is the same as the complexity of the existing one.

## 4.15

**4.15.1** Each branch that is not correctly predicted by the always-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.45)*0.25 = 0.41 |

**4.15.2** Each branch that is not correctly predicted by the always-not-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.55)*0.25 = 0.34 |

**4.15.3** Each branch that is not correctly predicted by the 2-bit predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.85)*0.25 = 0.113 |

**4.15.4** Correctly predicted branches had CPI of 1 and now they become ALU instructions whose CPI is also 1. Incorrectly predicted instructions that are converted also become ALU instructions with a CPI of 1, so we have:

| CPI without conversion | CPI with conversion | Speedup from conversion |
|---|---|---|
| 1 + 3*(1-0.85)*0.25 = 1.113 | 1 + 3*(1-0.85)*0.25*0.5 = 1.056 | 1.113/1.056 = 1.054 |

**4.15.5** Every converted branch instruction now takes an extra cycle to execute, so we have:

| CPI without conversion | Cycles per original instruction with conversion | Speedup from conversion |
|---|---|---|
| 1.113 | 1 + (1 + 3*(1 – 0.85))*0.25*0.5 = 1.181 | 1.113/1.181 = 0.94 |

**4.15.6** Let the total number of branch instructions executed in the program be B. Then we have:

| Correctly predicted | Correctly predicted non-loop-back | Accuracy on non-loop-back branches |
|---|---|---|
| B*0.85 | B*0.05 | (B*0.05)/(B*0.20) = 0.25 (25%) |

## 4.16

### 4.16.1

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

**4.16.2**

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

**4.16.3** 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 or Incorrect (in steady state) | Accuracy in steady state |
|---|---|---|---|
| T, NT, T, T, NT | 1st occurrence: 0,1,0,1,2<br>2nd occurrence: 1,2,1,2,3<br>3rd occurrence: 2,3,2,3,3<br>4th occurrence: 2,3,2,3,3 | C,I,C,C,I | 60% |

**4.16.4** 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.

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

**4.16.6** 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.

**4.17**

**4.17.1**

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

**4.17.2** 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.

**4.17.3** 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.

**4.17.4** 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 computer the address in EX, loads the handler's address in MEM, and sets the PC in WB.

**4.17.5** 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.

**4.18**

**4.18.1**

```
       ADD R5,R0,R0
Again: BEQ R5,R6,End
       ADD R10,R5,R1
       LW R11,0(R10)
       LW R10,1(R10)
       SUB R10,R11,R10
       ADD R11,R5,R2
       SW R10,0(R11)
       ADDI R5,R5,2
       BEW R0,R0,Again
End:
```

### 4.18.2

| Instructions | Pipeline | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R5,R0,R0 | IF | ID | EX | ME | WB | | | | | | | | | | | | | | | | | | | |
| BEQ R5,R6,End | IF | ID | ** | EX | ME | WB | | | | | | | | | | | | | | | | | | |
| ADD R10,R5,R1 | | IF | ** | ID | EX | ME | WB | | | | | | | | | | | | | | | | | |
| LW R11,0(R10) | | IF | ** | ID | ** | EX | ME | WB | | | | | | | | | | | | | | | | |
| LW R10,1(R10) | | | | IF | ** | ID | EX | ME | WB | | | | | | | | | | | | | | | |
| SUB R10,R11,R10 | | | | IF | ** | ID | ** | ** | EX | ME | WB | | | | | | | | | | | | | |
| ADD R11,R5,R2 | | | | | | IF | ** | ** | ID | EX | ME | WB | | | | | | | | | | | | |
| SW R10,0(R11) | | | | | | IF | ** | ** | ID | ** | EX | ME | WB | | | | | | | | | | | |
| ADDI R5,R5,2 | | | | | | | | | IF | ** | ID | EX | ME | WB | | | | | | | | | | |
| BEW R0,R0,Again | | | | | | | | | IF | ** | ID | ** | EX | ME | WB | | | | | | | | | |
| BEQ R5,R6,End | | | | | | | | | | | IF | ** | ID | EX | ME | WB | | | | | | | | |
| ADD R10,R5,R1 | | | | | | | | | | | IF | ** | ID | ** | EX | ME | WB | | | | | | | |
| LW R11,0(R10) | | | | | | | | | | | | | IF | ** | ID | EX | ME | WB | | | | | | |
| LW R10,1(R10) | | | | | | | | | | | | | IF | ** | ID | ** | EX | ME | WB | | | | | |
| SUB R10,R11,R10 | | | | | | | | | | | | | | | IF | ** | ID | ** | EX | ME | WB | | | |
| ADD R11,R5,R2 | | | | | | | | | | | | | | | IF | ** | ID | ** | ** | EX | ME | WB | | |
| SW R10,0(R11) | | | | | | | | | | | | | | | | | IF | ** | ** | ID | EX | ME | WB | |
| ADDI R5,R5,2 | | | | | | | | | | | | | | | | | IF | ** | ** | ID | EX | ME | WB | |
| BEW R0,R0,Again | | | | | | | | | | | | | | | | | | | | IF | ID | EX | ME | WB |
| BEQ R5,R6,End | | | | | | | | | | | | | | | | | | | | IF | ID | ** | EX | ME | WB |

**4.18.3** The only way to execute 2 instructions fully in parallel is for a load/store to execute together with another instruction. To achieve this, around each load/store instruction we will try to put non-load/store instructions that have no dependences with the load/store.

```
       ADD R5,R0,R0
Again: ADD R10,R5,R1
       BEQ R5,R6,End
       LW R11,0(R10)
       ADD R12,R5,R2
       LW R10,1(R10)
       ADDI R5,R5,2
       SUB R10,R11,R10
       SW R10,0(R12)
       BEQ R0,R0,Again
End:
```

Note that we are now computing a+i before we check whether we should continue the loop. This is OK because we are allowed to “trash” R10. If we exit the loop one extra instruction is executed, but if we stay in the loop we allow both of the memory instructions to execute in parallel with other instructions

## 4.18.4

| Instructions | Pipeline |
|---|---|
| ADD R5,R0,R0 | IF ID EX ME WB |
| ADD R10,R5,R1 | IF ID ** EX ME WB |
| BEQ R5,R6,End | IF ** ID EX ME WB |
| LW R11,0(R10) | IF ** ID EX ME WB |
| ADD R12,R5,R2 | IF ID EX ME WB |
| LW R10,1(R10) | IF ID EX ME WB |
| ADDI R5,R5,2 | IF ID EX ME WB |
| SUB R10,R11,R10 | IF ID ** EX ME WB |
| SW R10,0(R12) | IF ** ID EX ME WB |
| BEQ R0,R0,Again | IF ** ID EX ME WB |
| ADD R10,R5,R1 | IF ** ID EX ME WB |
| BEQ R5,R6,End | IF ** ID ** EX ME WB |
| LW R11,0(R10) | IF ** ID EX ME WB |
| ADD R12,R5,R2 | IF ** ID EX ME WB |
| LW R10,1(R10) | IF ID EX ME WB |
| ADDI R5,R5,2 | IF ID EX ME WB |
| SUB R10,R11,R10 | IF ID ** EX ME WB |
| SW R10,0(R12) | IF ID ** EX ME WB |
| BEQ R0,R0,Again | IF ** ID EX ME WB |
| ADD R10,R5,R1 | IF ** ID ** EX ME WB |
| BEQ R5,R6,End | IF ** ID EX ME WB |

## 4.18.5

| CPI for 1-issue | CPI for 2-issue | Speedup |
|---|---|---|
| 1.11 (10 cycles per 9 instructions). There is 1 stall cycle in each iteration due to a data hazard between the second LW and the next instruction (SUB). | 1.06 (19 cycles per 18 instructions). Neither of the two LW instructions can execute in parallel with another instruction, and SUB stalls because it depends on the second LW. The SW instruction executes in parallel with ADDI in even-numbered iterations. | 1.05 |

## 4.18.6

| CPI for1-issue | CPI for 2-issue | Speedup |
|---|---|---|
| 1.11 | 0.83 (15 cycles per 18 instructions). In all iterations, SUB is stalled because it depends on the second LW. The only instructions that execute in odd-numbered iterations as a pair are ADDI and BEQ. In even-numbered iterations, only the two LW instruction cannot execute as a pair. | 1.34 |

## 4.19

**4.19.1** The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have:

140 pJ + 2*70 ps + 60 pJ = 340 pJ

**4.19.2** The instruction memory is read for all instructions. Every instruction also results in two register reads (even if only one of those values is actually used). A load instruction results in a memory read and a register write, a store instruction results in a memory write, and all other instructions result in either no register write (e.g., BEQ) or a register write. Because the sum of memory read and register write energy is larger than memory write energy, the worst-case instruction is a load instruction. For the energy spent by a load, we have:

| |
|---|
| 140 pJ + 2*70 pJ + 60 pJ + 140 pJ = 480 pJ |

**4.19.3** Instruction memory must be read for every instruction. However, we can avoid reading registers whose values are not going to be used. To do this, we must add RegRead1 and RegRead2 control inputs to the Registers unit to enable or disable each register read. We must generate these control signals quickly to avoid lengthening the clock cycle time. With these new control signals, a LW instruction results in only one register read (we still must read the register used to generate the address), so we have:

| Energy before change | Energy saved by change | % Savings |
|---|---|---|
| 140 pJ + 2*70 pJ + 60 pJ + 140 pJ = 480 pJ | 70 pJ | 14.6% |

**4.19.4** Before the change, the Control unit decodes the instruction while register reads are happening. After the change, the latencies of Control and Register Read cannot be overlapped. This increases the latency of the ID stage and could affect the processor's clock cycle time if the ID stage becomes the longest-latency stage. We have:

| Clock cycle time before change | Clock cycle time after change |
|---|---|
| 250 ps (D-Mem in MEM stage) | No change (150 ps + 90 ps < 250 ps) |

**4.19.5** If memory is read in every cycle, the value is either needed (for a load instruction), or it does not get past the WB Mux (or a non-load instruction that writes to a register), or it does not get written to any register (all other instructions, including stalls). This change does not affect clock cycle time because the clock cycle time must already allow enough time for memory to be read in the MEM stage. It does affect energy: a memory read occurs in every cycle instead of only in cycles when a load instruction is in the MEM stage.

| I-Mem active energy | I-Mem latency | Clock cycle time | Total I-Mem energy | Idle energy % |
|---|---|---|---|---|
| 140 pJ | 200 ps | 250 ps | 140 pJ + 50 ps*0.1*140 pJ/200 ps = 143.5 pJ | 3.5 pJ/143.5 pJ = 2.44% |