Homework #3

Author: ZHU He (Ryker Zhu) **Student ID**: 202283930036 **Major**: Software Engineering

Exercises (The answer to the questions are all in orange): .....

Exercise 4.1 Consider the following instruction:

Instruction: `and rd, rsl, rs2`

Interpretation: `Reg[rd] = Reg[rs1] AND Reg[rs2]`

1. What are the values of control signals generated by the control in Figure 4.10 for this instruction?
2. Which resources (blocks) perform a useful function for this instruction?
3. Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

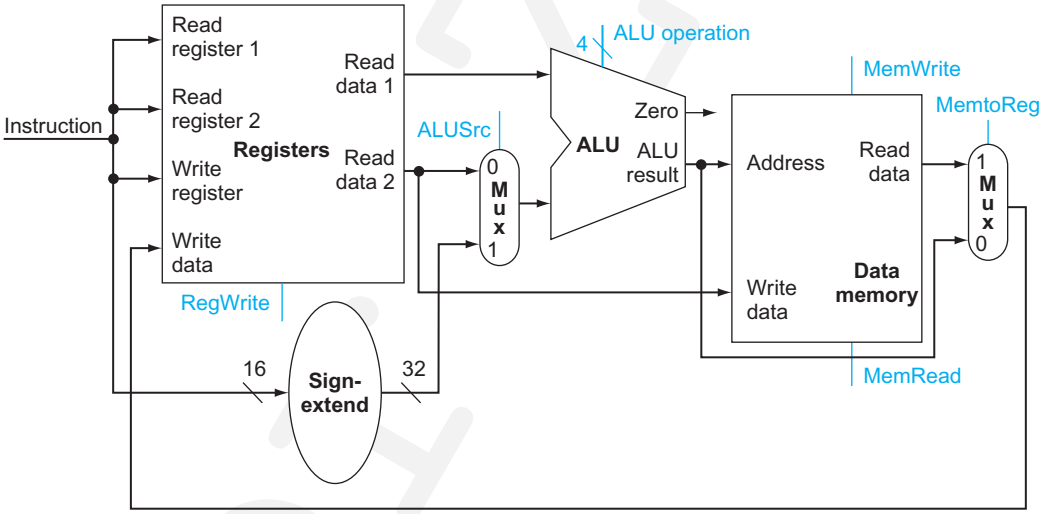

Figure 4.10 The datapath for the memory instructions and the R-type instructions.

Solution

1.

RegWrite	ALUSrc	ALU operation	MemWrite	MemRead	MemtoReg
`true`	`0000`	AND	`false`	`false`	`0`

2. Registers, the ALUSrc Mux, ALU, and the MemToReg Mux.
3. All the blocks in the figure above do produce output but the one of data memory is not used.

.....

Exercise 4.3 Consider the following instruction mix:

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

1. What fraction of all instructions use data memory?
2. What fraction of all instructions use instruction memory?
3. What fraction of all instructions use the sign extend?
4. What is the sign extend doing during cycles in which its output is not needed?

Solution

1. Since only the load and store instructions will use memory, fraction = 25%+10% = 35%.

2. All kinds of instructions must be fetched and then decoded from instruction memory before its execution, so it is 100%.
3. All kinds of instructions except for the R-type one does not require extending the sign, and thus $100\% - 24\% = 76\%$.
4. If the result is not needed, then it would be ignored.

Exercise 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
250ps	150ps	25ps	200ps	150ps	5ps	30ps	20ps	50ps	50ps

"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.

1. 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)?
2. What is the latency of `lw`? (Check your answer carefully. Many students place extra muxes on the critical path.)
3. What is the latency of `sw`? (Check your answer carefully. Many students place extra muxes on the critical path.)
4. What is the latency of `beq`?
5. What is the latency of an arithmetic, logical, or shift I-type (non-load) instruction?
6. What is the minimum clock period for this CPU?

Solution

1. R-type: $30\text{ps} + 250\text{ps} + 150\text{ps} + 25\text{ps} + 200\text{ps} + 25\text{ps} + 20\text{ps} = 700\text{ps}$;
2. `lw`: $30\text{ps} + 250\text{ps} + 150\text{ps} + 25\text{ps} + 200\text{ps} + 250\text{ps} + 25\text{ps} + 20\text{ps} = 950\text{ps}$;
3. `sw`: $30\text{ps} + 250\text{ps} + 150\text{ps} + 25\text{ps} + 200\text{ps} + 250\text{ps} = 905\text{ps}$;
4. `beq`: $30\text{ps} + 250\text{ps} + 150\text{ps} + 25\text{ps} + 200\text{ps} + 5\text{ps} + 25\text{ps} + 20\text{ps} = 705\text{ps}$;
5. I-type (non-load): $30\text{ps} + 250\text{ps} + 150\text{ps} + 25\text{ps} + 200\text{ps} + 25\text{ps} + 20\text{ps} = 700\text{ps}$;
6. The minimum clock cycle required is the largest time slot that a instruction takes, i.e. the amount of time of accessing memory via `lw` instruction, 950ps.

Exercise 4.8 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-`lw`)	`lw`	`sw`	`beq`
52%	25%	11%	12%

Solution As is computed in the previous exercise, the average time per instruction for the instruction mix is $52\% \times 700\text{ps} + 25\% \times 950\text{ps} + 11\% \times 905\text{ps} + 12\% \times 705\text{ps} = 785.65\text{ps}$ while a CPU executing different instructions for the same clock cycle time is 950ps. Thus we can obtain the speedup factor: $\dfrac{950\text{ps}}{785.65\text{ps}} = 1.209$.

Exercise 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).

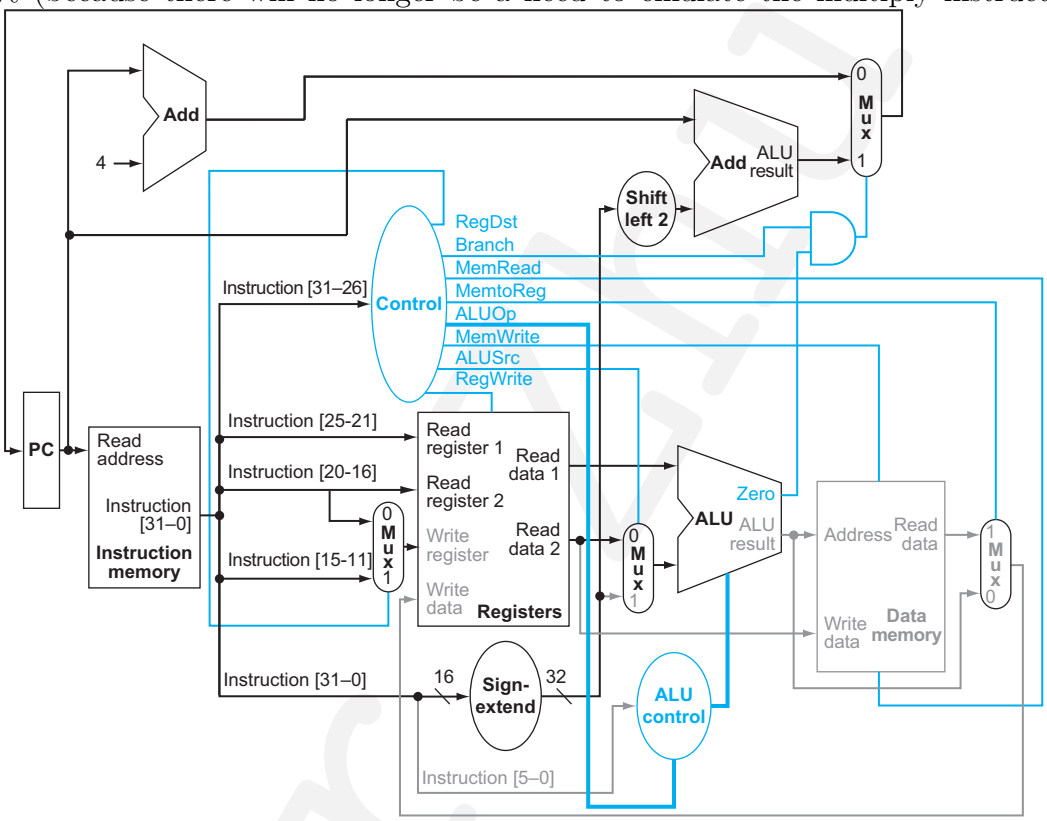

Figure 4.21 The datapath in operation for a branch-on-equal instruction.

1. What is the clock cycle time with and without this improvement?
2. What is the speedup achieved by adding this improvement?
3. What is the slowest the new ALU can be and still result in improved performance?

Solution

1. Before the improvement the cycle time is 950psand after its introduction, the cycle time would then 300pslonger, i.e. 1250ps.
2. Running time for the original CPU is $n \times 950\text{ps}$ where n denotes the number of instructions. While after introduction, the running time would be $95\% \times n \times 1250\text{ps} = n \times 1187.5\text{ps}$. Hence, the speedup factor is $\dfrac{n \times 950\text{ps}}{n \times 1187.5\text{ps}} = 0.8$.
3. The worst case is the new one takes as much cycle time as the previous one, i.e. 950psand given that the improvement would reduce the number of instructions by 5% so the maximum cycle time is $\dfrac{950\text{ps}}{95\%} = 1000\text{ps}$. In other words, the time the new ALU takes should only increase at most 50ps, which means a slowest improved ALU should take 250ps to finish its computation.

Exercise 4.15 `lw` is the instruction with the longest latency on the CPU from Section 4.4. If we modified `lw` and `sw` so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in `rs` before calling `lw`/`sw`), 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 `lw`/`add` or `sw`/`add` combinations.

1. What would the new clock cycle time be?
2. 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 `lw` and `sw` instruction is replaced with a sequence of two instructions.)
3. What is the primary factor that influences whether a program will run faster or slower on the new CPU?
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?

Solution

1. Given that the instructions for the new version of CPU would have only 4 stages since the datapath of an instruction would choose either EX or ME but not in a synchronous way, thus a instruction takes at most $30\text{ps} + 250\text{ps} + 150\text{ps} + \max\{25\text{ps} + 200\text{ps},\ 25\text{ps} + 250\text{ps} + 25\text{ps} + 20\text{ps}\} = 750\text{ps}$.
2. Note that the original version of the CPU takes 950ps to run a cycle while the revised one takes 750ps. Besides, since `ld` (load double words) and `sd` (save double words) instructions are equivalent to two `lw`s and `sw`s, there will be an extra $25\% + 11\% = 36\%$ load/store instructions, resulting in $1.36 \times n \times 750\text{ps} = n \times 1012.5\text{ps}$ and the speedup factor $\dfrac{n \times 950\text{ps}}{n \times 1012.5\text{ps}} \approx 0.94 < 1$. Hence, the program run slower on this new CPU.
3. The number and effect of load/store instructions are the primary factors.
4. The new one is better. Despite increasing instruction count, this design streamlines memory access, enhances scalability, and facilitates future optimizations, making it a superior choice for systems prioritizing memory latency reduction and overall performance.

Exercise 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
250ps	350ps	150ps	300ps	200ps

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

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

1. What is the clock cycle time in a pipelined and non-pipelined processor?
2. What is the total latency of an `lw` instruction in a pipelined and non-pipelined processor?
3. 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. Assuming there are no stalls or hazards, what is the utilization of the data memory?
5. Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

Solution

1. For non-pipelined processors, the clock cycle time is the sum of stages, which equals $250\text{ps} + 350\text{ps} + 150\text{ps} + 300\text{ps} + 200\text{ps} = 1250\text{ps}$; for pipelined ones, it is equal to the running time of the longest stage, i.e. $t_{\text{ID}} = 350\text{ps}$.
2. The time cost for a non-pipelined processor is same as the cycle time, i.e. 1250ps; for the pipelined one, it is equal to the number of stages times the cycle time, i.e. $5 \times 350\text{ps} = 1750\text{ps}$.
3. Split the ID (Instruction Decode) stage since it is the stage with the longest running time and then the cycle time becomes the former second longest running time, i.e. 300ps.
4. 35%. 20% from load instructions, and 15% from store instructions.
5. 65%. 45% from ALU/logic instructions, and 20% from load instructions.

Exercise 4.18 Assume that `$s0` is initialized to 11 and `$s1` is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.6 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 `$s2` and `$s3` be?

```
addi    $s0, $s1, 5
add     $s2, $s0, $s1
addi    $s3, $s0, 15
```

Solution `$s2` $= 22 + 5 + 22 = 49$ and `$s3` $= 22 + 5 + 15 = 42$.

Exercise 4.20 Add `NOP` instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi    $s0, $s1, 5
add     $s2, $s0, $s1
addi    $s3, $s0, 15
add     $s4, $s2, $s1
```

Solution

```
addi    $s0, $s1, 5
nop
nop
add     $s2, $s0, $s1
addi    $s3, $s0, 15
nop
add     $s4, $s2, $s1
```

Exercise 4.22 Consider the fragment of MIPS assembly below:

```
sd      $s5, 12($s3)
ld      $s5, 8($s3)
sub     $s4, $s2, $s1
beqz    $s4, label
add     $s2, $s0, $s1
```

```
sub    $s2, $s6, $s1
```

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.

1. Draw a pipeline diagram to show were the code above will stall.
2. In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?
3. Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding NOPs to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.
4. Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix from Exercise 4.8.)

Solution

1.
```
sd     $s5, 12($s3)        IF ID EX ME WB
ld     $s5, 8($s3)            IF ID EX ME WB
sub    $s4, $s2, $s1             IF ID EX ME WB
beqz   $s4, label                   ** ** IF ID EX ME WB
add    $s2, $s0, $s1                        IF ID EX ME WB
sub    $s2, $s6, $s1                           IF ID EX ME WB
```

2. There is no possibility in reordering the code to reduce the number of stalls since every instruction must be fetched before any stages later from the same memory where data comes from.
3. It cannot be solved by inserting NOPs since NOP itself shall be fetched from instruction memory as well.
4. $25\% + 11\% = 36\%$ since every data access will lead to a stall.

Exercise 4.24 Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
ld     $11, 0($12)      IF ID EX ME WB
add    $13, $11, $14       IF ID EX .. ME WB
or     $15, $16, $17          IF ID .. EX ME WB
```

Choice 2:

```
ld     $11, 0($12)      IF ID EX ME WB
add    $13, $11, $14       IF ID .. EX ME WB
or     $15, $16, $17          IF .. ID EX ME WB
```

Solution Choice 2 since stall demands are detected in the stage of instruction fetching.

Exercise 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    $s3, $s1, $s0
lw     $s2, 4($s3)
```

```
lw      $s1, 0($s4)
or      $s2, $s3, $s2
sw      $s2, 0($s3)
```

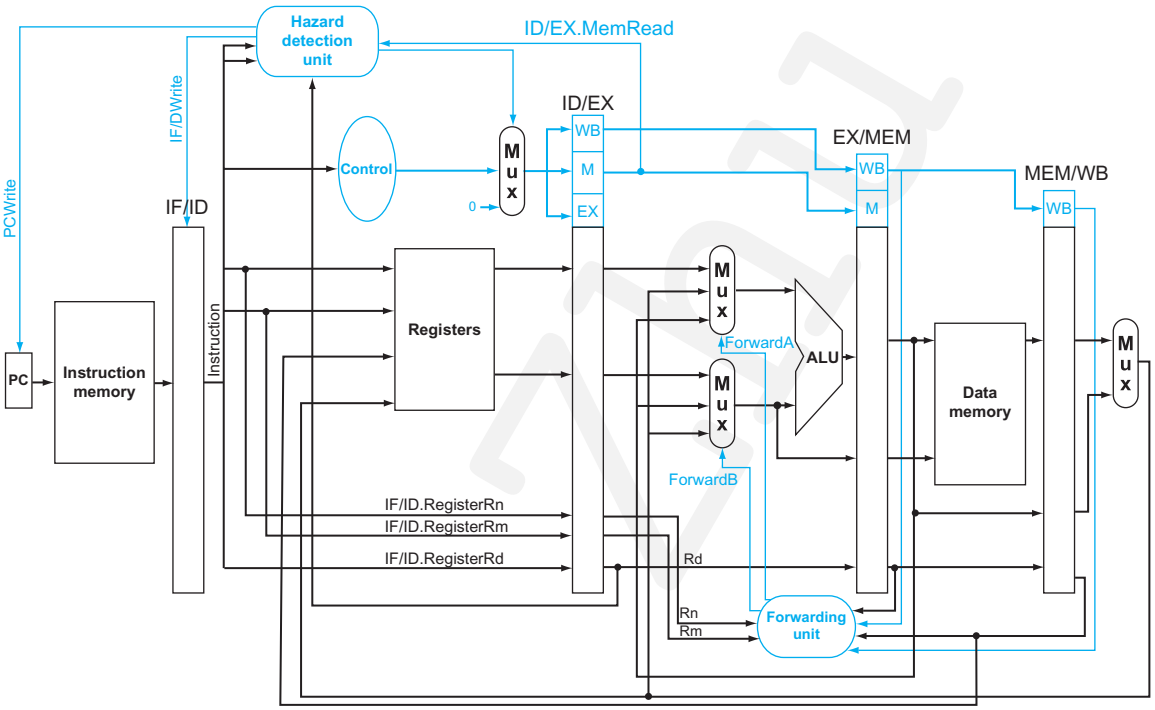

Figure 4.59 Pipelined control overview, showing the two multiplexers for forwarding, the hazard detection unit, and the forwarding unit.

1. If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.
2. Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register $t0 can be used to hold temporary values in your modified code.
3. If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?
4. 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.
5. 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.
6. 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.

Solution

1.
```
add     $s3, $s1, $s0
nop
nop
lw      $s2, 4($s3)
lw      $s1, 0($s4)
nop
or      $s2, $s3, $s2
nop
nop
sw      $s2, 0($s3)
```

2. There is no way to reduce the number of NOPs by rearranging the code.
3. It executes correctly. Only when a instruction uses the result from the preceding load instruction would hazard detection be needed, which is not the case.
4. The first seven cycles:

Cycle	1	2	3	4	5	6	7
add $s3, $s1, $s0	IF	ID	EX	ME	WB		
lw $s2, 4($s3)		IF	ID	EX	ME	WB	
lw $s1, 0($s4)			IF	ID	EX	ME	WB
or $s2, $s3, $s2				IF	ID	EX	ME
sw $s2, 0($s3)					IF	ID	EX

Hazard detection unit:

Since there are no existing stalls after the introduction of forwarding, PCWrite and IF/IDWrite are always true and the multiplexer before ID/EX is always set to select the control values.

Forwarding unit:

Cycle	ForwardA	ForwardB	Comment
1	/	/	There is no instruction in EX stage
2	/	/	There is no instruction in EX stage
3	00	00	Values are directly taken from register file
4	10	00	Value stored in base register is taken from the result of previous add instruction
5	00	00	Value comes directly from register files
6	00	10	The value stored in $s3 is obtained from register file since the result of addition is written back in this cycle and the value stored in $s2 is taken from the first lw instruction
7	00	10	Value in $s3 is taken from register file and data to be stored is fetched from the previous or instruction

5. The hazard detection unit requires the values of $rd from the MEM/WB register and there is no extra need for output signals. To detect potential data hazard between add and ld, the value $rd from EX/MEM should be employed. Besides, the one between the first ld and the or should be detected by using the value of $rd from MEM/WB.
6. The first five cycles:

Cycle	1	2	3	4	5
add $s3, $s1, $s0	IF	ID	EX	ME	WB
lw $s2, 4($s3)		IF	ID	**	**
lw $s1, 0($s4)			IF	**	**

Output signals for each cycle:

Cycle	PC Write	IF/ID Write	Multiplexer before Control
1	true	true	0
2	true	true	0
3	true	true	0
4	false	false	1
5	false	false	1

Exercise 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	lw	sw
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

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

1. 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.
2. Repeat 4.28.1 for the "always-not-taken" predictor.
3. Repeat 4.28.1 for the 2-bit predictor.
4. 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.
5. 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.
6. 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?

Solution

1. Since a mispredicted branch, which makes up 25% of the instruction mix, leads to a three-instruction penalty, i.e. the instructions in the IF, ID and EX stage and given that there are $100\% - 45\% = 55\%$ of instructions triggers the flush for the always-taken predictor, the CPI would reach $1 + 25\% \times 3 \times (100\% - 55\%) = 1.4125$.
2. Similarly, $1 + 25\% \times 3 \times (100\% - 45\%) = 1.3375$.
3. Similarly, $1 + 25\% \times 3 \times (100\% - 85\%) = 1.1125$.
4. Given that half of the branch instructions are converted into ALU instructions, the proportion of branchs becomes 12.5%, which leads to a lower CPI: $1+12.5\%\times3\times(1-85\%) = 1.05625$ and subsequently the speedup factor: $1.1125 \div 1.05625 \approx 1.0533$.

5. Given that half of the branch instructions are converted into two ALU instructions, 12.5% of the branch instruction will have an extra cycle since it would be replaced with ALU instructions and another 12.5% that is not replaced would then have a $100\% - 85\% = 15\%$ chance of penalty so the CPI becomes $1 + 12.5\% \times 1 + 12.5\% \times 15\% \times 1 = 1.14375$ and thus we can obtain the speedup factor: $1.1125 \div 1.14375 \approx 0.973$
6. Since 80% of the branch instructions are always predicted correctly, only the remaining 20% have to predict depending on the 2-bit predictor accuracy. The overall accuracy, however, does not change, i.e. 85%. Thus, the accuracy of the remaining ones is obtained by solving the equation $80\% + 20\% \times x = 85\%$, which is 25%.