EECS 645

Hao Luo

Dec 7th

Homework 11

4.11 Consider the following loop.

```
loop:   lw r1,0(r1)
        and r1,r1,r2
        lw r1,0(r1)
        lw r1,0(r1)
        beq r1,r0,loop
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, and that the pipeline has full forwarding support. Also assume that many iterations of this loop are executed before the loop exits.

4.11.1 [10] Show a pipeline execution diagram for the third iteration of this loop, from the cycle in which we fetch the first instruction of that iteration up to (but not including) the cycle in which we can fetch the first instruction of the next iteration. Show all instructions that are in the pipeline during these cycles (not just those from the third iteration).

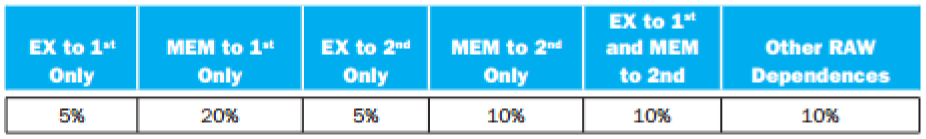
Solution:

	1	2	3	4	5	6	7	8	9	10	11	12	13
Lw3	WB												
Beq	ID	EX	MEM	WB									
Lw1	IF	ID	EX	MEM	WB								
And			IF	ID	EX	MEM	WB						
Lw2				IF	ID	EX	MEM	WB					
Lw3						IF	ID	EX	MEM	WB			
Beq									IF	ID	EX	MEM	WB
Lw1										IF	ID	EX	MEM
And												IF	ID
Lw2													IF

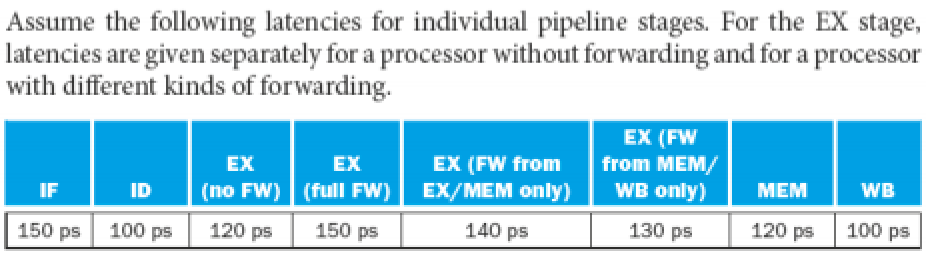
4.11.2 [10] How often (as a percentage of all cycles) do we have a cycle in which all five pipeline stages are doing useful work?

Solution: From the pipeline execution diagram in problem 1, the total number of instructions is 10, and there are only clock cycles for load instructions which use all 5 stages of the pipeline. Therefore, it means the percentage of cycle in which all five pipeline stages do useful work is 6 / 10 = 60%.

4.12 This exercise is intended to help you understand the cost/complexity/ performance trade-off s of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.45. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions have 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 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. Also, assume 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 MEM to 2nd	Other RAW Dependences
5%	20%	5%	10%	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
150 ps	100 ps	120 ps	150 ps	140 ps	130 ps	120 ps	100 ps

4.12.1 [10] If we use no forwarding, what fraction of cycles are we stalling due to data hazards?

Solution: Derived CPI = initial CPI + (Ex to 1st Only + Ex to 1st and Mem to 2nd + Mem to 1st Only) * 2 + (Mem to 2nd Only + Ex to 2nd Only + Ex to 1st and Mem to 2nd) * 1 = 1 + (0.05 + 0.2 + 0.1) * 2 + (0.1 + 0.5 + 0.1) * 1 = 1.95

Current CPI = Derived CPI – 1 = 1.95 – 1 = 0.95
Fraction of stalling cycles = Current CPI / Derived CPI = 0.95 / 1.95 = 48.7%

4.12.2 [5] If we use full forwarding (forward all results that can be forwarded), what fraction

of cycles are we stalling due to data hazards?

Solution: From the above table, when we use full forwarding, the percentage of instruction stalled in one cycle is 20%

Derived CPI = Based CPI + Current CPI = 1 + 0.2 = 1.2
Fraction of stall cycles = Current CPI / Derived CPI = 0.2 / 1.2 = 16.7%

4.12.3 [10] Let us assume that we cannot afford to have three-input muxes 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). Which of the two options results in fewer data stall cycles?

Solution: Stall cycles for EX/MEM = (Mem to 1st Only + Ex to 1st and Mem to 2nd + Ex to 2nd Only + Mem to 2nd Only) = (0.2 + 0.1 + 0.05 + 0.1) = 0.45

Stall cycles for MEM/WB = (Ex to 1st Only + Mem to 1st Only + Ex to 1st and Mem to 2nd) = (0.05 + 0.2 + 0.1) = 0.35

From these two results we can see, there is less data stall cycles in forwarding of MEM/WB pipeline register. Thus, we'd better to forward only from MEM/WB register.

4.12.4 [10] For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

Solution: The clock cycle time without forwarding = Derived CPI without forwarding * without forwarding latency = 1.95 * 120ps = 234ps

The clock cycle time with forwarding = Derived CPI with full forwarding * full forwarding latency = 1.2 * 150ps = 180ps

Speedup = 234 / 180 = 1.3

4.12.5 [10] What would be the additional speedup (relative to a processor with forwarding) if we added time-traveled forwarding that eliminates all data hazards? Assuming that the yet to-be-invented time-travel circuitry adds 100ps to the latency of the full-forwarding EX stage.

Solution: The clock cycle time with forwarding = Derived CPI with full forwarding * full forwarding latency = 1.2 * 150ps = 180ps

The clock cycle time with travel forwarding = Based CPI * time traveling latency = 1 * (150ps + 100ps) = 250ps

Speedup = 180 / 250 = 0.72

4.12.6 [10] Repeat 4.12.3 but this time determine which of the two options result in shorter time per instruction.

Solution: CPI of forwarding from EX/MEM register = based CPI + current CPI = 1 + 0.45 = 1.45, and the clock cycle time of forwarding = Derived CPI * full forwarding latency = 1.45 * 150ps = 217.5ps

CPI of forwarding from MEM/WB register = based CPI + current CPI = 1 + 0.35 = 1.35, and the clock cycle time of forwarding = Derived CPI * full forwarding latency = 1.35 * 150ps = 202.5ps

Comparing with the clock cycle time between EX/MEM register and MEM/WB register, we find that MEM/WB register has a shorter clock cycle time per instruction.

4.13 This exercise is intended to help you understand the relationship between forwarding, hazard detection, and ISA design. Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a 5-stage pipelined datapath:

```
add  r5,r2,r1
lw   r3,4(r5)
lw   r2,0(r2)
or   r3,r5,r3
sw   r3,0(r5)
```

4.13.1 [10] If there is no forwarding or hazard detection, insert nops to ensure correct exeution.

Solution:

Add	IF	ID	EX	MEM	WB								
Lw1		IF	ID	NOP	NOP	EX	MEM	WB					
Lw2			IF	NOP	NOP	ID	EX	MEM	WB				
Or						IF	ID	NOP	EX	MEM	WB		
sw							IF	NOP	ID	NOP	NOP	EX	MEM	WB

The modification code is:

```
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 [10] Repeat 4.13.1 but now use nops only when a hazard cannot be avoided by changing or rearranging these instructions. You can assume register R7 can be used to hold temporary values in your modified code.

Solution:

4.13.3 [10] If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when this code executes?

Solution: Without the hazard detection unit, when this code executes, instructions which depends on the result from preceding load instruction will get the value which store in the register before the load instruction.

4.13.4 [20] If there is forwarding, for the first five cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.60.

Solution:

Cycle					
Instruction	ADD r5,r2,r1	LW r3,4(r5)	LW r2,0(r2)	OR r3,r5,r3	SW r3,0(r5)
signal	PCWrite=1 ALUin1 = X ALUin2 = X	PCWrite=1 ALUin1 = X ALUin2 = X	PCWrite=1 ALUin1 = 0 ALUin2 = 0	PCWrite=1 ALUin1 = 1 ALUin2 = 0	PCWrite=1 ALUin1 = 0 ALUin2 = 0
Cycle 1	IF				
Cycle 2	ID	IF			
Cycle 3	EX	ID	IF		
Cycle 4	MEM	EX	ID	IF	
Cycle 5	WB	MEM	EX	ID	IF

4.13.5 [10] If there is no forwarding, what new inputs and output signals do we need for the hazard detection unit in Figure 4.60? Using this instruction sequence as an example, explain why each signal is needed.

Solution: If there is no forwarding:

The instruction in ID stage which need value from previous instruction in EX stage or MEM stage should be stalled. Also, we need to check both EX stage and MEM stage's destination register. If the instruction is in EX stage, it is necessary to check on Rd for load and R-type instruction. For MEM stage, before selecting destination register, we need to check the number of register.

The additional inputs we need are, Rd from ID/EX register, ALU output from EX/MEM register, and Rt from ID/EX register. For output, there is no need for the additional outputs.

4.13.6 [20] For the new hazard detection unit from 4.13.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

Solution: PCWrite = 0 if there is a data hazard detection unit, otherwise, there is no data hazard detection unit.

Instruction	C1	C2	C3	C4	C5	signal
ADD	IF	ID	EX	MEM	WB	PCWrite=1
LW1	IF	ID	EX	MEM		PCWrite=1
LW2	IF	ID	EX			PCWrite=1
OR	IF	ID				PCWrite=0
SW	IF					PCWrite=0

4.14 This exercise is intended to help you understand the relationship between delay slots, control hazards, and branch execution in a pipelined processor. In this exercise, we assume that the following MIPS code is executed on a pipelined processor with a 5-stage pipeline, full forwarding, and a predict-taken branch predictor:

```
        lw r2,0(r1)
label1: beq r2, r0, label2 # not taken once, then taken
        lw r3,0(r2)
        beq r3,r0,label1 # taken
        add r1,r3,r1
label2: sw r1,0(r2)
```

4.14.1 [10] Draw the pipeline execution diagram for this code, assuming there are no delay slots and that branches execute in the EX stage.

Solution: There are 16 pipeline cycles when there are no delay slots and that branches execute in the EX stage.

Lw1: r2,0(r1)

Beq1: r2, r0, label2

Lw2: r3,0(r2)

Beq2: r3,r0,label1

Add: r1,r3,r1

Sw: r1,0(r2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Lw1	IF	ID	EX	MEM	WB											
Beq1			IF	ID	EX	MEM	WB									
Sw					IF											
Lw2						IF	ID	EX	MEM	WB						
Beq2								IF	ID	EX	MEM	WB				
Beq1										IF	ID	EX	MEM	WB		
Sw												IF	ID	EX	MEM	WB

4.14.2 [10] Repeat 4.14.1, but assume that delay slots are used. In the given code, the instruction that follows the branch is now the delay slot instruction for that branch.

Solution: When the delay slots are used, there are 14 pipeline cycles.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Lw1	IF	ID	EX	MEM	WB									
Beq1			IF	ID	EX	MEM	WB							
Sw					IF									
Lw2						IF	ID	EX	MEM	WB				
Beq2								IF	ID	EX	MEM	WB		
Beq1									IF	ID	EX	MEM	WB	
sw										IF	ID	EX	MEM	WB

4.14.3 [20] One way to move the branch resolution one stage earlier is to not need an ALU operation in conditional branches. The branch instructions would be "bez rd,label" and "bnez rd,label", and it would branch if the register has and does not have a zero value, respectively. Change this code to use these branch instructions instead of beq. You can assume that register R8 is available for you to use as a temporary register, and that an seq (set if equal) R-type instruction can be used.

Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in Figure 4.62. However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

Solution:

```
lw r2,0(r1)
label1:  bez r2, label2
label2:  sw r1,0(r2)
lw r3,0(r2)
bez r3,,label1
label1:  bez r2, label2
label2:  sw r1,0(r2)
```

Lw 1	IF	ID	EX	MEM	WB														
Label1			IF	-	-	ID	EX	MEM	WB										
Label2					IF	-	-	-	-	-									
Lw2						IF	ID	EX	MEM	WB									
Bez1								IF	-	-	ID	EX	MEM	WB					
Label1												IF	ID	-	-	EX	MEM	WB	
Label2													IF	ID	-	-	EX	MEM	WB

4.14.4 [10] Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in Figure 4.62. Which type of hazard is this new logic supposed to detect?

Solution: The detection logic of hazard is used for detecting data hazard. For example, there is a branch which depends on the result of preceding instructions. If this previous instruction is immediately preceded LW, then the branch should be stalled for 2 cycles because LW result is only generated by MEM stage. And if this previous instruction is R-type, the branch only need to be stalled for 1 cycle because R-type's result is generated by EX stage and forwarded to EX stage of branch. Also, when this branch depend on result of second previous instruction. When branch exists in the ID stage, the MEM stage of load instruction will be finished, so branch also need to be stalled for 1 cycles.

4.14.5 [10] For the given code, what is the speedup achieved by moving branch execution into the ID stage? Explain your answer. In your speedup calculation, assume that the additional comparison in the ID stage does not affect clock cycle time.

Solution: Pipeline execution diagram with forwarding, here instruction names which I use in this diagram. There is 15 clock cycles in this diagram.

Lw1: r2,0(r1)

Beq1: r2, r0, label2

Lw2: r3,0(r2)

Beq2: r3,r0,label1

Add: r1,r3,r1

Sw: r1,0(r2)

Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lw1	IF	ID	EX	MEM	WB										
Beq1		IF	Stall	Stall	ID	EX	MEM	WB							
Lw2						IF	ID	EX	MEM	WB					
Beq2							IF	Stall	Stall	ID	EX	MEM	WB		
Beq1										IF	ID	EX	MEM	WB	
Sw											IF	ID	EX	MEM	WB

Then, here is the pipeline execution diagram without forwarding, and there is 14 clock cycles in this diagram

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Lw1	IF	ID	EX	MEM	WB									
Beq1		IF	ID	Stall	EX	MEM	WB							
Lw2						IF	ID	EX	MEM	WB				
Beq2							IF	ID	Stall	EX	MEM	WB		
Beq1								IF	Stall	ID	EX	MEM	WB	
Sw										IF	ID	EX	MEM	WB

The speedup = clock cycles with no forwarding / clock cycles with forwarding = 14 / 15 = 0.933

4.14.6 [10] Using the first branch instruction in the given code as an example, describe the forwarding support that must be added to support branch execution in the ID stage. Compare the complexity of this new forwarding unit to the complexity of the existing forwarding unit in Figure 4.62.

Solution: Branch instructions are now executed in the ID stage instead of EX stage. Then, forwarding unit should be used like the following cases.

1. If the branch use value in register from the immediate pervious instruction, it is necessary to stall the branch because the previous instruction is in EX stage when branch already use the stale value from ID stage.

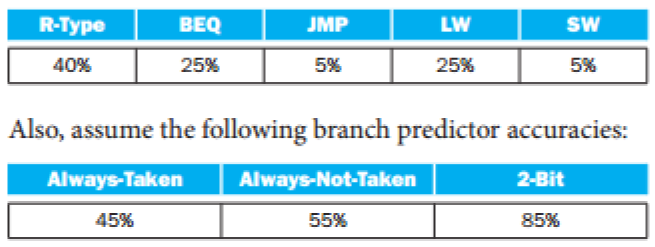
2. If branch in ID stage is dependent to the R-type instruction which is in MEM stage, we should forward to ensure the branch's execution.

3. If branch in ID stage is dependent to a load instruction in WB stage, we also need to forward to ensure its execution.

For the new forwarding unit, it should control two multiplexer which placed before the branch comparator. When it works, each multiplexer can select the value read from Registers, the EX/ MEM register, the MEM/WB register.

The complexity of the new forwarding unit is the same as the original one's.

4.15 The importance of having a good branch predictor depends on how oft en 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	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

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

4.15.1 [10] 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 EX stage, that there are no data hazards, and that no delay slots are used.

Solution: Control hazard of branches has 2 stage stalls, which are IF, ID. It means the stall cycles of branches are 3. For the mispredicted branches with always-taken predictor, we can use the breakdown of beq, 25%. Also, we can read the accuracy of always-not-taken from above table, which is 55%.

Extra CPI = stall cycles * breakdown * accuracy of mispredicted of always-not-taken = 2 * 0.25 * 0.55 = 0.275. Hence, the extra CPI is 0.275.

4.15.2 [10] Repeat 4.15.1 for the "always-not-taken" predictor

Solution: For always-not-taken predictor, the stall cycles are still 2, and we still pick the breakdown 25% of beq. The accuracy should be always-taken now, which is 45%.

Extra CPI = stall cycles * breakdown * accuracy of mispredicted of always-taken = 2 * 0.25 * 0.45 = 0.225

4.15.3 [10] Repeat 4.15.1 for for the 2-bit predictor

Solution: For the 2-bit predictor, the stall cycles are still 2, and we still pick the breakdown 25% of beq. The accuracy of 2-bit predictor = 1 – 0.85 = 0.15.

Extra CPI = stall cycles * breakdown * accuracy of mispredicted of not 2-bit predictor = 2 * 0.25 * 0.15 = 0.075

4.15.4 [10] With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

Solution: CPI without conversion = based CPI + extra CPI = 1 + 0.075 = 1.075

CPI with conversion = based CPI + extra CPI / 2 = 1 + 0.075/2 = 1.0375

Speedup = CPI without conversion / CPI with conversion = 1.075 / 1.0375 = 1.036

4.15.5 [10] 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.

Solution: For a beq instruction, the probability of making an incorrect prediction = 1 / 2 = 0.5. Also, we already get the CPI 1.075 for 2-bit predictor without conversion.

Extra CPI with conversion = (based CPI + stall cycles * accuracy of mispredicted of not 2-bit predictor) * breakdown of beq * probability of incorrect predictions = (1 + 2 * 0.15) * 0.25 * 0.5 = 0.1625

CPI with conversion = 1 + 0.1625 = 1.1625

Speedup = CPI without conversion / CPI with conversion = 1.075 / 1.1625 = 0.925

4.15.6 [10] 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: We get the correct predicted accuracy of 2-bit predictor = 1 – 0.85 = 0.15. On the remaining 20% of the branch instructions, the accuracy will be 15% / 20% = 75%.

4.16 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

4.16.1 [5] What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

Solution: For always-taken predictor

Actual will be: T, NT, T, T, NT

Predict will be: T, T, T, T, T

So, the accuracy = 3 / 5 = 60%

For always-not-taken predictor

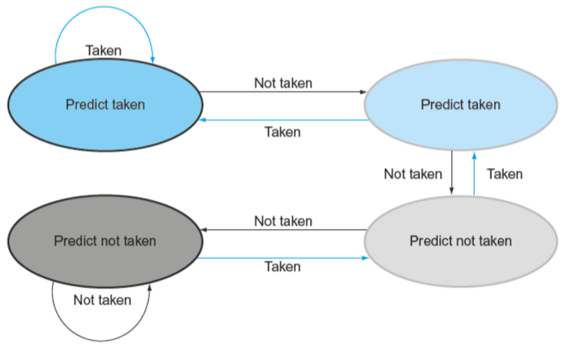
Actual will be: T, NT, T, T, NT

Predict will be: NT, NT, NT, NT, NT

The accuracy = 2 / 5 = 40%

4.16.2 [5] What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken)?

Solution: To the first 4 branches, the Actual will be: T, NT, T, T, and I name the four states in Figure 4.63 as 00(predict not taken left), 01 (predict not take right), 10 (predict taken right), 11 (predict taken left)



Prediction	00-NT	01-NT	00-NT	01-NT
Outcome	T	NT	T	T
Hit/miss	0	1	0	0
New prediction	01-NT)	00-NT	01-NT	10-T

The accuracy of predictor will be 1 / 4 =25%

4.16.3 [10] What is the accuracy of the two-bit predictor if this pattern is repeated forever?

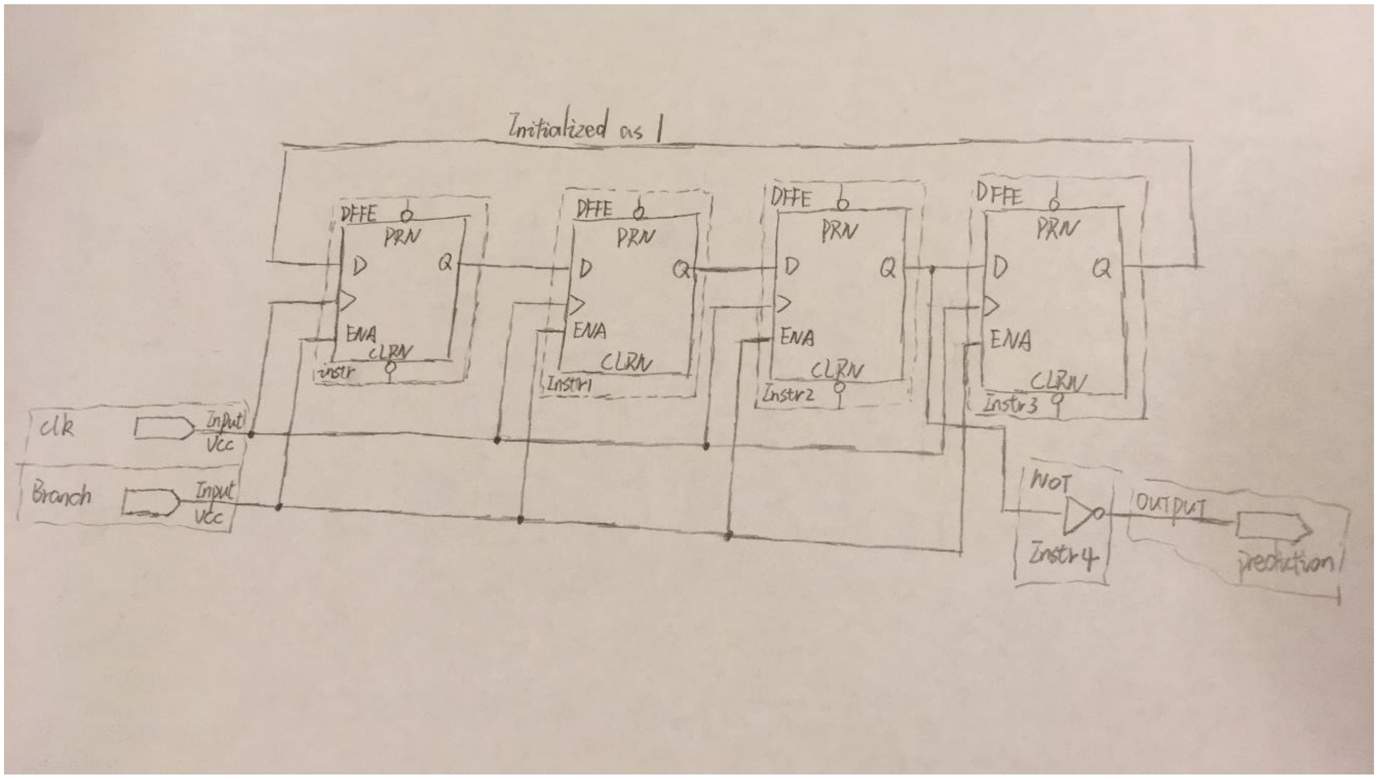
Solution: From the setting of last question, I make a table to observe the repeating patterns of branch predictor.

Prediction	10-T	11-T	10-T	11-T	11-T		01-NT	10-T	01-T	10-T	11-T		10-T	11-T	10-T	11-T	11-T
Outcome	T	NT	T	T	NT		T	NT	T	T	NT		T	NT	T	T	NT
Hit/miss	1	0	1	1	0		0	0	0	1	0		1	0	1	1	0
New prediction	11-T	10-T	11-T	11-T	10-T		10-T	01-NT	10-T	11-T	10-T		11-T	10-T	11-T	11-T	10-T

In this table, we can this pattern become steady in the third iteration, the prediction are corresponded to the accuracy. There are three correct predictions, thus the accuracy = 3 / 5 =60%.

4.16.4 [30] 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.

Solution:



4.16.5 [10] What is the accuracy of your predictor from 4.16.4 if it is given a repeating pattern that is the exact opposite of this one?

Solution: The exact opposite of T, NT, T, T, NT is NT, T, NT, NT, T. Thus, all prediction will be the opposite of the new outcomes.

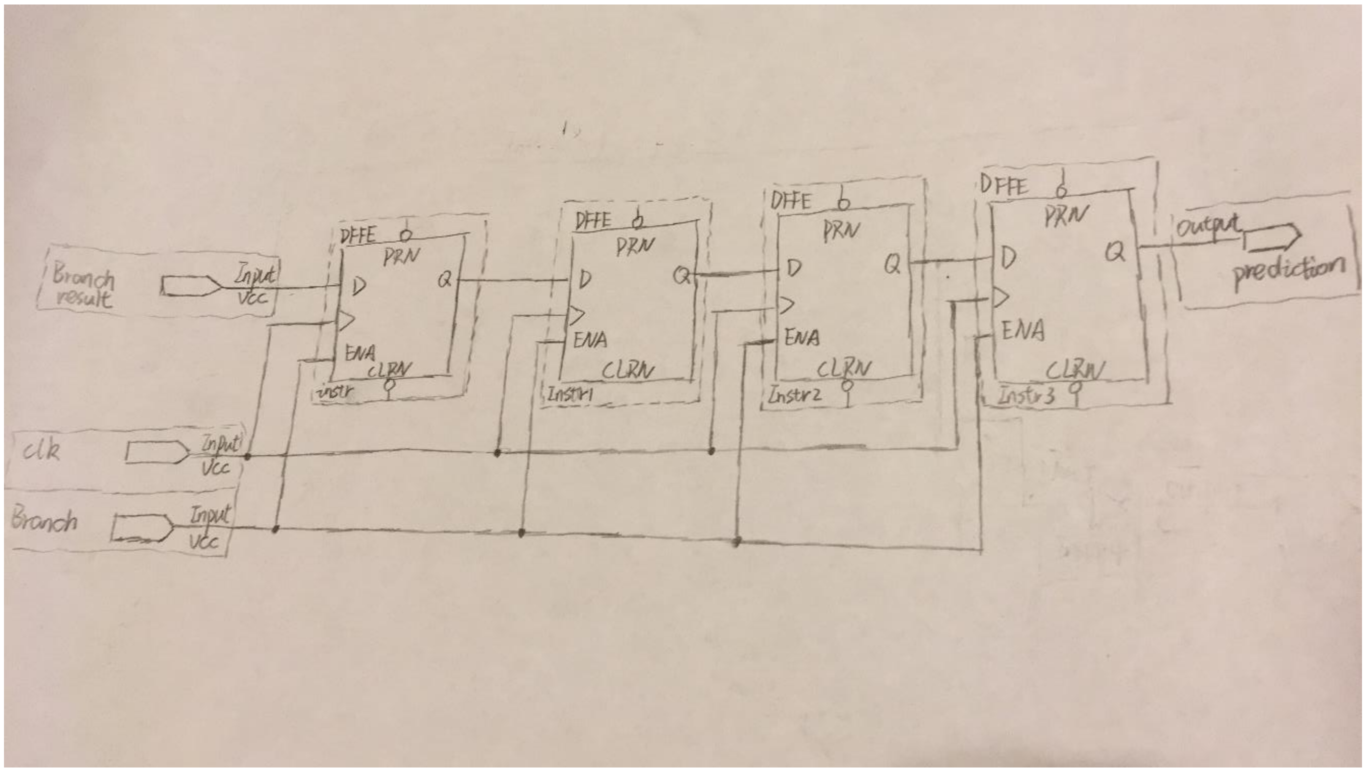
Actual: NT, T, NT, NT, T

Predict: T, NT, T, T, NT

The accuracy will be number of hits / total branches number = 0 / 5 =0%

4.16.6 [20] Repeat 4.16.4, 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.

Solution:



Branch result
Input
VCC
DFFE
PRN
D
Q
ENA
CLRN
instr
Instr1
Instr2
Instr3
clk
Branch
output
prediction