Homework Assignment 2

CS/ECE 6810: Computer Architecture
September 26,2018
Name: Jake Pitkin
UID: u0891770

ILP and Branch Prediction

Due Date: October 03, 2018.
120 points

1. **Multi-cycle Instructions.** A pipelined architecture comprises instruction fetch (IF), instruction decode (ID), register read (RR), execute (EX), and write-back (WB) stages.

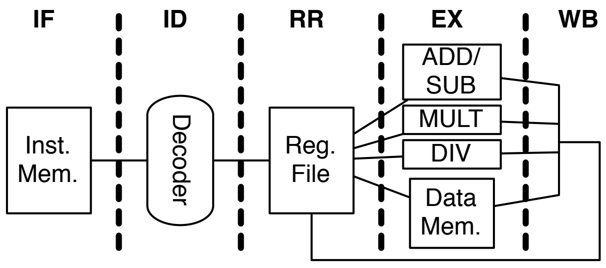

Figure 1: Pipelined core architecture.

Except EX, each stage requires one clock cycle to complete. The EX stage includes 4 functional units that perform memory and ALU operations—e.g., ADD, SUB, MULT, DIV, load, and store. The table below shows the latency of each operation in terms of clock cycles. The architecture implements forwarding paths from the EX/WB pipeline registers to the EX input. Moreover, the register file may be bypassed during the WB stage to send the register value to EX.

	ADD	SUB	MULT	DIV	Load	Store
Latency	1	1	3	7	2	1

i Identify all structural and data hazards in the following code. **(10 points)**

Load F6, 20(R5)
Load F2, 28(R5)
MUL F0, F2, F4
SUB F8, F6, F3
DIV F10, F0, F6
ADD F6, F8, F2
Store F8, 50(R5)

Below is a table of all data hazards present in the above list of instructions. *Instruction 1* is the instruction that comes first temporally which is followed by *Instruction 2*. *Hazard Type* indicates what type of hazard is detected and *Register* is the register causing the hazard.

Instruction 1	**Instruction 2**	**Hazard Type**	**Register**
Load F2, 28(R5)	MUL F0, F2, F4	RAW	F2
Load F6, 20(R5)	SUB F8, F6, F3	RAW	F6
MUL F0, F2, F4	DIV F10, F0, F6	RAW	F0
Load F6, 20(R5)	DIV F10, F0, F6	RAW	F6
SUB F8, F6, F3	ADD F6, F8, F2	RAW	F8
Load F2, 28(R5)	ADD F6, F8, F2	RAW	F2
SUB F8, F6, F3	Store F8, 50(R5)	RAW	F8
SUB F8, F6, F3	ADD F6, F8, F2	WAR	F6
DIV F10, F0, F6	ADD F6, F8, F2	WAR	F6
Load F6, 20(R5)	ADD F6, F8, F2	WAW	F6

My notes for the above data hazards: I listed all hazards including the hazards that didn't end up being an issue in part ii.

Structural hazards: The two successive load instructions will cause a structural hazard on the EX Data Mem as Load takes two cycles the second load will have to wait for the EX Data Mem resource.

ii Create a timing diagram for the code showing the execution of the code in time (clock cycles). **(20 points)**

Instruction	**Cycle 1**	**Cycle 2**	**Cycle 3**	**Cycle 4**	**Cycle 5**	**Cycle 6**
Load F6, 20(R5)	IF	ID	RR	EX Data	EX Data	WB
Load F2, 28(R5)	-	IF	ID	RR	STALL	EX Data
MUL F0, F2, F4	-	-	IF	ID	RR	STALL
SUB F8, F6, F3	-	-	-	IF	ID	RR
DIV F10, F0, F6	-	-	-	-	IF	ID
ADD F6, F8, F2	-	-	-	-	-	IF
Store F8, 50(R5)	-	-	-	-	-	-

Instruction	**Cycle 07**	**Cycle 08**	**Cycle 09**	**Cycle 10**	**Cycle 11**	**Cycle 12**
Load F6, 20(R5)						
Load F2, 28(R5)	EX Data	WB				
MUL F0, F2, F4	STALL	EX MULT	EX MULT	EX MULT	WB	
SUB F8, F6, F3	EX SUB	STALL	STALL	STALL	STALL	WB
DIV F10, F0, F6	RR	STALL	STALL	STALL	EX DIV	EX DIV
ADD F6, F8, F2	ID	RR	EX ADD	STALL	STALL	STALL
Store F8, 50(R5)	IF	ID	RR	EX Data	STALL	STALL

Instruction	**Cycle 13**	**Cycle 14**	**Cycle 15**	**Cycle 16**	**Cycle 17**	**Cycle 18**
Load F6, 20(R5)						
Load F2, 28(R5)						
MUL F0, F2, F4						
SUB F8, F6, F3						
DIV F10, F0, F6	EX DIV	EX DIV	EX DIV	EX DIV	EX DIV	WB
ADD F6, F8, F2	STALL	STALL	STALL	STALL	STALL	STALL
Store F8, 50(R5)	STALL	STALL	STALL	STALL	STALL	STALL

Instruction	**Cycle 19**	**Cycle 20**	**Cycle 21**	**Cycle 22**	**Cycle 23**	**Cycle 24**
Load F6, 20(R5)						
Load F2, 28(R5)						
MUL F0, F2, F4						
SUB F8, F6, F3						
DIV F10, F0, F6						
ADD F6, F8, F2	WB					
Store F8, 50(R5)	STALL	WB				

My Notes for the above timelne:

- Pipeline registers contain a FIFO queue allowing them to contain a queue of multiple waiting values.
- Stalls are used to maintain the WB order of the instructions even if there is no hazard.
- Instruction MUL F0, F2, F4 will have its register read of F2 overwritten in cycle 8 by a forward from the EX/WB pipeline.
- Instruction ADD F6, F8, F2 will have it's register read of F8 overwritten in cycle 9 by a forward from the EX/WB pipeline.
- Instruction Store F8, 50(R5) will have its register read of F8 overwritten in cycle 10 by a forward from the EX/WB pipeline.
- Instruction DIV F10, F0, F6 will have its register read of F0 overwritten in cycle 11 by a forward from the EX/WB pipeline.

2. **Points of Production and Consumption.** Consider an unpipelined processor where it takes 36 ns to go through the circuits and 0.5 ns for the latch overhead. Assume that the point of production and point of consumption in the unpipelined processor are separated by 12ns. Assume that half the instructions do not introduce a data hazard and half the instructions depend on their preceding instruction.

   i What is the maximum throughput of the unpipelined architecture in instructions per second (IPS). **(10 points)**

   The IPS of an unpipelined architecture is given by IPS = IPC * 1/CT. The architecture is unpipelined so IPC = 1 and the CT is given by 36 ns + 0.5 ns.

$$IPS = IPC * \frac{1}{CT}$$
$$IPS = 1 * \frac{1}{36.5 * 10^{-9}}$$
$$IPS = 2.7397 * 10^{7}$$

ii We build a 12-stage pipelined architecture for the processor. Please compute the percentages of increase/decrease in throughput for the pipelined architecture compared to unpipelined process. **(10 points)**

To compute the change in throughput we must first compute the new IPS. To calculate this we need the new IPC and the new CT.
We know the circuits take 36ns so with a 12-stage pipelined architecture each stage will take 3ns without the latch. This means the point of production and point of consumption is separated by 12ns/3ns or 4 pipeline stages.
If this hazard occurs half the time, we know that half our instructions will take one cycle while the other half take four. This means we will finish two instructions per five cycles so IPC = 2/5 = 0.4.
CT will simply be the length of a stage plus the latch time so CT = 3.5 ns.

$$IPS_{new} = IPC_{new} * \frac{1}{CT_{new}}$$
$$IPS_{new} = 0.4 * \frac{1}{3.5 * 10^{-9}}$$
$$IPS_{new} = 1.1429 * 10^{8}$$

Using this we can compute the increase in throughput:

$$Throughput\ Increase = IPC_{new}/IPC$$
$$Throughput\ Increase = (1.1429 * 10^{8})/(2.7397 * 10^{7})$$
$$Throughput\ Increase = 4.1716$$
$$Throughput\ Increase = 417.16\%$$

3. **Software Optimization.** Below are examples of a C and assembly codes for a **for**-loop. Notice that `i` is an 8-byte long integer. Register names indicate if floating point (F) or integer (R) operations are necessary for instructions.

Source code
```
for (i = 125; i > 0; i--) {
    x[i] = s * y[i] + z;
}
```

Assembly code
```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       MUL F3, F1, F2
       ADD F5, F3, F4
       Store F5, 20000(R1)
       ADDI R1, R1, #-8
Chck:  BNEQ R1, R0, Loop
```

Consider a five-stage scalar pipeline with multi-cycle functional units for floating-point and integer operations at the EX stage. Assume the following delays between dependent (producer-consumer) instructions:
(a) Load feeding any instruction: 1 stall cycle
(b) FP MULT/ADD feeding store: 4 stall cycles
(c) Integer ADD feeding a branch: 1 stall cycle
(d) A conditional branch has 1 delay slot (the next instruction after a conditional branch is fetched and executed to completion without knowing the outcome of the branch)

i First show all of the stall cycles necessary in the original assembly code. Then, find an optimized schedule for this loop through reordering instructions but without resorting to loop unrolling. **(10 points)**

Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       stall
       MUL F3, F1, F2
       stall
       ADD F5, F3, F4
       stall
       stall
       stall
       stall
       Store F5, 20000(R1)
       ADDI R1, R1, #-8
       stall
Chck:  BNEQ R1, R0, Loop
       stall
```

Explanation of inserted stalls:

- Stall 1: Load F1, 0(R1) feeds MUL F3, F1, F2 the register F1.
- Stall 2: MUL F3, F1, F2 feeds ADD F5, F3, F4 the register F3. (from Canvas)
- Stall 3-6: ADD F5, F3, F4 feeds Store F5, 20000(R1) the register F5.
- Stall 7: ADDI R1, R1, #-8 feeds BNEQ R1, R0, Loop the register R1.
- Stall 8: BNEQ R1, R0, Loop is a conditional branch.

Optimized Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       ADDI R1, R1, #-8
       MUL F3, F1, F2
       stall
       ADD F5, F3, F4
       stall
       stall
       stall
       stall
       Store F5, 20008(R1)
Chck:  BNEQ R1, R0, Loop
       stall
```

Explanation of loop optimization: The instruction ADDI R1, R1, #-8 was moved up to be the second instruction in the loop. To keep the Store F5, 20000(R1) instruction correct, 8 is added to the offset to make up for subtracting 8 from R1 before executing the Store. This allowed for removal of two of the stall operations from the loop body.

ii Optimize the schedule by loop unrolling 1x, 2x, and 3x. Notice that a 1x unroll includes the original loop body plus the 1 time unrolled instructions. **(10 points)**

Loop unrolling 1x without reordering:

Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       stall
       MUL F3, F1, F2
       stall
       ADD F5, F3, F4
       stall
       stall
       stall
       stall
       Store F5, 20000(R1)
       Load F6, -8(R1)
       stall
       MUL F7, F6, F2
       stall
       ADD F8, F7, F4
       stall
       stall
       stall
       stall
       Store F8, 19992(R1)
       ADDI R1, R1, #-16
       stall
Chck:  BNEQ R1, R0, Loop
       stall
```

Loop unrolling 1x reordered:

Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       Load F6, -8(R1)
       MUL F3, F1, F2
       MUL F7, F6, F2
       ADD F5, F3, F4
       ADD F8, F7, F4
       stall
       stall
       ADDI R1, R1, #-16
       Store F5, 20016(R1)
       Store F8, 20008(R1)
Chck:  BNEQ R1, R0, Loop
       stall
```

Explanation of loop optimization: Similar to part i, the ADDI instruction is moved up and the Store instruction offset is adjusted to take in account the ADDI instruction is now being executed prior to the Store. We can interweave the unrolled Load, MUL, and ADD operations to remove additional stalls. An identical approach will be used for x2 and x3 unrolling so I will skip the versions without reordering.

Loop unrolling 2x reordered:

Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       Load F6, -8(R1)
       Load F9, -16(R1)
       MUL F3, F1, F2
       MUL F7, F6, F2
       MUL F10, F9, F2
       ADD F5, F3, F4
       ADD F8, F7, F4
       ADD F11, F10, F4
       stall
       ADDI R1, R1, #-24
       Store F5, 20024(R1)
       Store F8, 20016(R1)
       Store F11, 20008(R1)
Chck:  BNEQ R1, R0, Loop
       stall
```

Loop unrolling 3x reordered:

Assembly code

```
       ADDI R1, R0, #1000
       JMP Chck
Loop:  Load F1, 0(R1)
       Load F6, -8(R1)
       Load F9, -16(R1)
       Load F12, -24(R1)
       MUL F3, F1, F2
       MUL F7, F6, F2
       MUL F10, F9, F2
       MUL F13, F12, F2
       ADD F5, F3, F4
       ADD F8, F7, F4
       ADD F11, F10, F4
       ADD F14, F13, F4
       ADDI R1, R1, #-32
       Store F5, 20032(R1)
       Store F8, 20024(R1)
       Store F11, 20016(R1)
       Store F14, 20008(R1)
Chck:  BNEQ R1, R0, Loop
       stall
```

With 3x unrolling we've optimized the code to have less overhead and removed all stall cycles.

4. **Branch Prediction.** Assume a 32-bit five-stage scalar pipeline with the fetch, decode, execute, memory, and write back stages. All pipeline stages require 1 cycle except the load and store operations that need 3 cycles to access the data memory; branch instructions need 2 clock cycles to determine the outcome. Example C and assembly codes are given for a user application.

Source code

```
n = 250;
i = 0;
do {
   x[i] = x[i] + 1;
   i = i+1;
} while(i < n);
```

Assembly code

```
       ADDI R3, R0, #1000
       ADDI R2, R0, #0
Loop:  Load R1, 0(R2)
       ADDI R1, R1, #1
       Store R1, 0(R2)
       ADDI R2, R2, #4
       SUB R4, R3, R2
       BNEQ R4, R0, Loop
```

i Without any branch prediction, how many stall cycles are necessary due to the branch instructions in the original code? **(10 points)**

Assembly code

```
       ADDI R3, R0, #1000
       ADDI R2, R0, #0
       stall
Loop:  Load R1, 0(R2)
       stall
       stall
       stall
       ADDI R1, R1, #1
       stall
       Store R1, 0(R2)
       ADDI R2, R2, #4
       stall
       SUB R4, R3, R2
       stall
       BNEQ R4, R0, Loop
       stall
       stall
```

As per the statement by the professor on the Canvas discussion board: "A branch needs n cycles to produce outcome means in the absence of a branch predictor, n stall cycles are necessary after a branch instruction.". The problem description states it takes branch instructions two clock cycles to determine the outcome, so we require two stalls.

Additionally, we will require three stalls between Load R1, 0(R2) and ADDI R1, R1, #1 so the register R1 can be loaded. As well as a single stall after the instruction ADDI R2, R0, #0, the instruction ADDI R1, R1, #1, the instruction ADDI R2, R2, #4, and the instruction SUB R4, R3, R2 as the following instructions use their result and the problem statement states that ADDI and SUB require 1 cycle.

The question only asks for the stall cycles due to the branch instruction which is two.

ii Assume a static branch predictor capable of predicting always-taken or always-not-taken. Compute the percentages of improvements in IPC compared to the previous case with no branch predictor? **(10 points)**

We can calculate speedup by considering the formula from the Branch Prediction lecture slides:

$$Speedup = \frac{1 + bc}{1 + (1 - a)bc}$$

Where a is the prediction accuracy, b is the frequency of branches , and c is the misprediction cost.

First we can consider the prediction accuracy of a static branch predictor that predicts always-taken. The BNEQ R4, R0, Loop instruction will be checked 250 times. The predictor will be correct 249 times and incorrect 1 time. So a = 249/250 = 0.996.

Next we consider the frequency of branches. As written above with the stalls, there are three instructions as well as the Loop body containing 12 instructions (not including the two stalls after the BNEQ instruction that are no longer needed). Out of the 12 * 250 + 3 instructions 250 of them will be branch instructions. So b = 250 / 3,003 = 0.08325.

Finally c is the misprediction cost which is two stalls so c = 2.

$$Speedup = \frac{1 + (0.08325 * 2)}{1 + (1 - 0.996)(0.008325 * 2)}$$

$$Speedup = 1.16642$$

$$Percent\ Performance\ Gain = \ 16.64\%$$

iii Assume a single 3-bit saturating counter for dynamic branch prediction. The initial state of the counter is 000. States 000–011 predict not taken; while, 100–111 indicate taken. Compute the total number of mis-predictions. Compute the percentages of improvements in IPC compared to the case with no branch predictor? **(10 points)**

Here again we will consider the speedup formula as in part ii. The frequency of branches, b, will stay the same at b = 0.08325. As well as the cost of a misprediction, c, which is c = 2. With the new dynamic branch predictions the prediction accuracy, a, will change.

First we will compute the mispredictions. Starting in state 000 we will mispredict the first iteration of the loop. Moving to state 001 we will mispredict again on the second iteration. Moving to state 010 we will mispredict on the third iteration. Moving to state 0110 we will mispredict yet again on the four iteration. Finally on iteration 250 we will be in state 111 and mispredict. **That is a total of 5 mispredictions**. This gives a prediction accuracy a = 245/250. We can use this to calculate the speedup over no branch predictor:

$$Speedup = \frac{1 + (0.08325 * 2)}{1 + (1 - 0.98)(0.008325 * 2)}$$

$$Speedup = 1.16611$$

$$Percent\ Performance\ Gain = \ 16.61\%$$