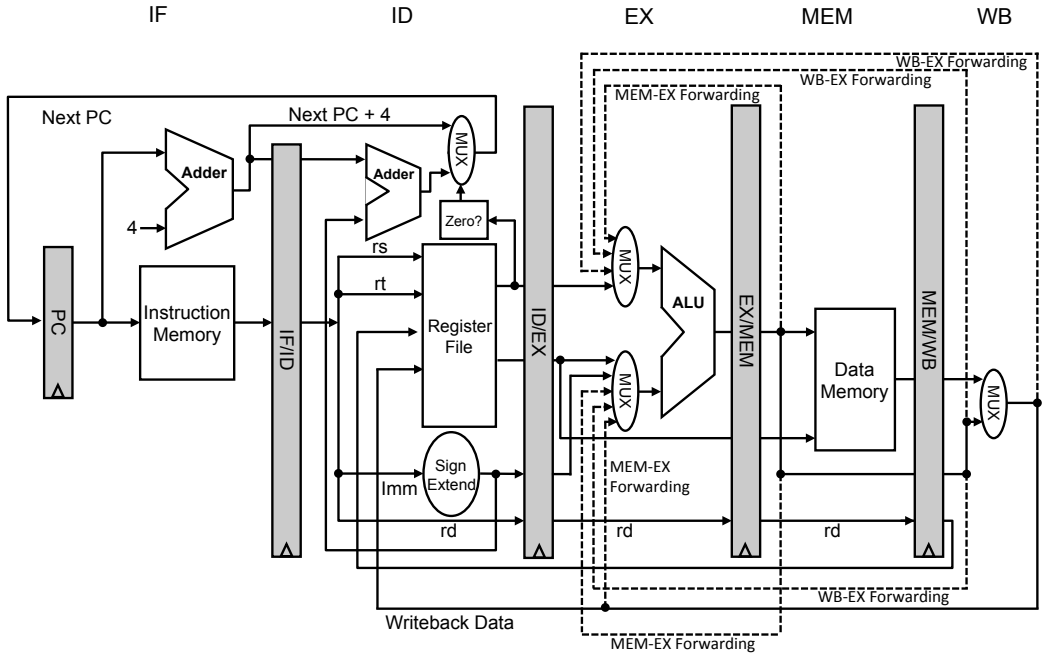
ECE 570
High-Performance Computer Architecture
Solutions Set #1
Winter 2015

[20 pts]

1- For this problem, consider the following two code sequences executing on the 5-stage datapath with normal forwarding and bypassing shown below:

Code 1		Code 2	
…		…	
LD	R4, 0(R2)	SUB	R4, R3, R2
ADD	R1, R4, R2	BNEZ	R4, Loop
…		…	



(a) The first code is an example of a "load followed by its immediate use". Show and explain the correct execution timing that would allow the datapath to properly execute Code 1.

(b) The second code is an example where *MEM-to-ID forwarding* (also know as branch forwarding) will benefit on the 5-stage pipeline.

(i) Show and explain the execution timing as to why the datapath would not properly execute the above code sequence without the MEM-ID forwarding.

(ii) Make the necessary modifications to the datapath to implement the MEM-ID forwarding and allow Code 2 to run with minimal stalling and show the execution timing for the modified design. Clearly explain your design.

Solutions

(a) The current datapath with normal forwarding and bypassing can only forward from either MEM or WB to EX. Thus, there is no forwarding pass from the output of Data Memory to the inputs of the ALU. Thus, the values read from the Data Memory has to be forwarded from the output of the MUX in the WB stage to one of the inputs of the ALU. However, this requires an artificial stall between the LD and the ADD instructions (i.e., in the MEM stage). In addition, this requirement has to be recognized in the ID stage. This leads to the following execution timing.

		1	2	3	4	5	6	7	8
LD	R4, 0(R2)	IF	ID	EX	MEM	WB			
ADD	R1, R4, R2		IF	ID	stall	EX	MEM	WB	

What this timing is showing is that during cycle 3, when LD is in EX and ADD is in ID, some logic (referred to as the Hazard Detection Unit) has to detect this and insert a stall between LD and ADD instructions. Then, during cycle 5, this stall will allow the value read from the Data Memory (in cycle 4) to be forwarded to one of the inputs to the ALU.

(b) As can be seen from the datapath above, currently there is no forwarding path from EX, MEM, or WB to the "Zero?" comparator in ID. Therefore, if the code were to be executed without any modifications, as shown below, R4 would be needed by BNEZ in the ID stage in cycle 3 and yet SUB would not write the value until the WB stage in cycle 5!

		1	2	3	4	5	6	7	8
SUB	R4, R3, R2	IF	ID	EX	MEM	WB			
BNEZ	R4, Loop		IF	ID	EX	MEM	WB		

One way to solve this problem is to appropriately stall BNEZ so that SUB has sufficient time to write back R4. The execution timing of this is shown below:

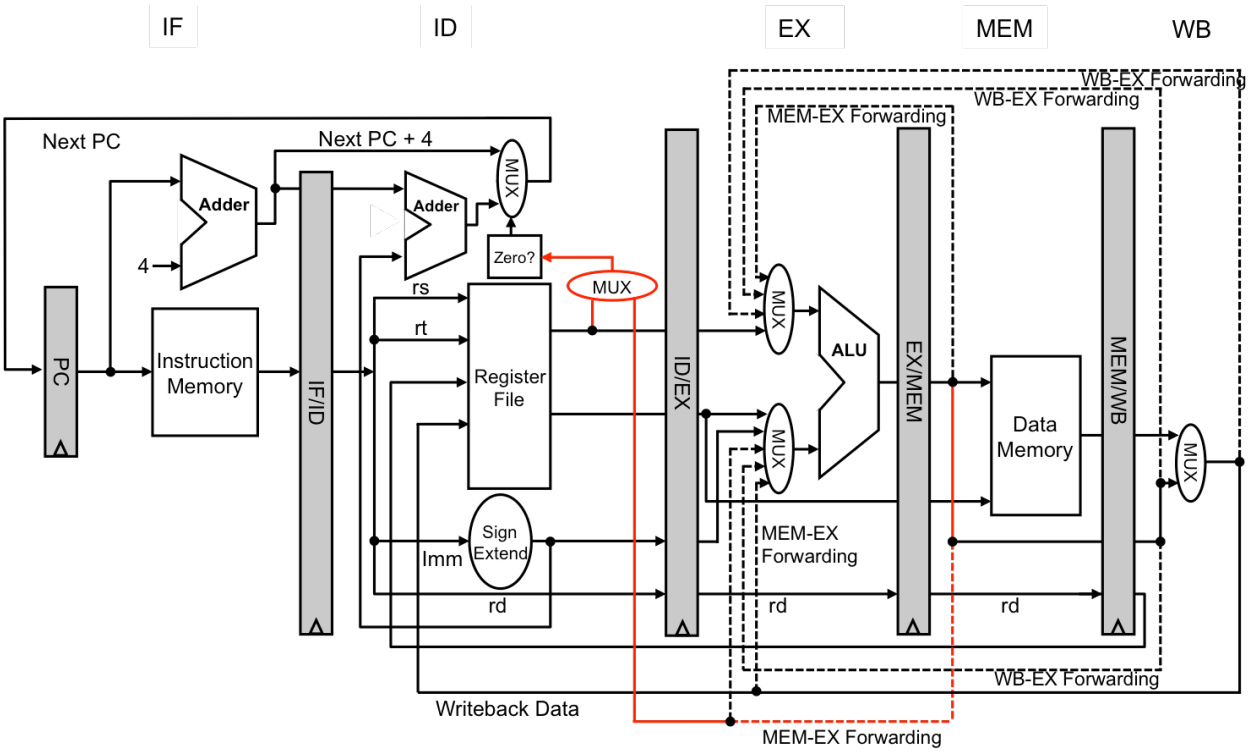
		1	2	3	4	5	6	7	8
SUB	R4, R3, R2	IF	ID	EX	MEM	WB			
BNEZ	R4, Loop		IF	ID	stall	stall	EX	MEM	WB

This would allow R4 to be written back to the register file in the first half of cycle 5 and then R4 is read in the second half of cycle 5, which is referred to as *forwarding through the register file*. However, this would require two stalls.

In order to reduce the number of stalls, the data that will be written back to R4 needs to be forwarded from the beginning of the MEM stage to the "Zero?" comparator in the ID stage. This can be done by having a multiplexer at the front of the Zero Unit to choose between the output of the register file for rs and the forwarded value as shown below. Then, the output of the ALU (i.e., the value for R4) can be forwarded at the beginning of the MEM stage back to the input of the MUX in the ID Stage. In addition, a stall needs to be generated between the SUB and the BNEZ instructions as shown below:

		1	2	3	4	5	6	7	8
SUB	R4,R3,R2	IF	ID	EX	MEM	WB			
BNEZ	R4, Loop		IF	ID	stall	EX	MEM	WB	

You may wonder why we can't just eliminate the stall by having the value forwarded from the output of the ALU in the EX stage, instead of in the MEM stage. The reason is that there is delay involved with forwarding paths. Thus, the clock cycle would have to be longer to accommodate this extra delay, which in turn slows the processor down for all the instructions.



[20 pts]

2- Consider the following code, which represents Y = a×X+Y operation for a vector length of 100. Assume the pipeline latencies from Figure 3.2 in the text and a 1-cycle delay branch that is resolved in the ID stage. In addition, as discussed in Problem #1, the pipeline uses MEM-to-ID forwarding to forward the result of an ALU operation from the MEM stage to the ID stage.

```
        DADDIU    R4, R1, #800  ; R1 = X, R4 = upper bound for X
foo:    L.D       F2, 0(R1)     ; load X(i)
        MUL.D     F4, F2, F0    ; F0 = a, perform a*x(i)
        L.D       F6, 0(R2)     ; load Y(i)
        ADD.D     F6, F4, F6    ; perform a*X(i) + Y(i)
        S.D       F6, 0(R2)     ; store Y(i)
        DADDIU    R1, R1, #8    ; increment X index
        DADDIU    R2, R2, #8    ; increment Y index
        DSUBU     R3, R4, R1    ; test if done
        BNEZ      R3, foo       ; loop if not done
```

(a) Show how this loop would execute without any scheduling. Maximize the performance of this code by applying both instruction reordering (also know as pipeline scheduling) and delay branch techniques. Ignoring the startup delays and assuming the loop executes 100 times, determine the number of cycles required to execute the code *before* and *after* the optimizations. Do not be concerned about what happens after the loop.

(b) Unroll the loop as many times as necessary to schedule it without stalls and show the instruction schedule. Again, assuming the loop executes 100 times, determine the number of cycles required to execute the code *before* and *after* unrolling.

Solutions

(a)

Un-scheduled code

```
foo:    L.D       F2, 0(R1)
```

```
        stall
        MUL.D     F4, F2, F0    ; load double to FP ALU (1-cycle latency)
        L.D       F6, 0(R2)
        stall
        stall
        ADD.D     F6, F4, F6    ; FP ALU to FP ALU (3-cycle latency)
        stall
        stall
        S.D       F6, 0(R2)     ; FP ALU to Store double (2-cycle latency)
        DADDUI    R1, R1, #8
        DADDUI    R2, R2, #8
        DSUBU     R3, R4, R1
        stall                   ; Branch forwarding
        BNEZ      R3, foo
        stall                   ; 1-cycle delay slot
```

Scheduled code

```
foo:    L.D       F2, 0(R1)
        L.D       F6, 0(R2)
        MUL.D     F4, F2, F0
        DADDUI    R1, R1, #8
        DADDUI    R2, R2, #8
        DSUBU     R3, R4, R1
        ADD.D     F6, F4, F6
        stall
        BNEZ      R3, foo
        S.D       F6, 0(R2)
```

Before the optimization, the code requires 7 stalls. Thus, the loop takes (9+7)*100 = 1600 cycles.

After the optimization, the loop requires (9+1)*100 = 1000 cycles since all 9 instructions (including the delay slot) plus the stall are executed for each iteration.

(b) The code can be unrolled once to eliminate all the stalls after scheduling. When unrolling a loop with no loop-carried dependences, the following basic guidelines can be followed.

First, copy all the statements in the original loop and put them after the original loop statements. Second, rename all the registers in the copied instructions so that they are distinct from the original statements. This can be done by adding a fixed value to each register number, assuming there are enough registers available. In our case, 6 was added to the instruction belong to the second iteration because the first iteration already uses F4 and F6. Then, remove loop overhead instructions. Moreover, instructions that use or update index calculations will have to be modified based on reordering of instructions or elimination of intermediate index updates. The code below shows this process.

```
foo:    L.D       F2, 0(R1)
        stall
        MUL.D     F4, F2, F0
        L.D       F6, 0(R2)
        stall
        stall
        ADD.D     F6, F4, F6
        stall
        stall
        S.D       F6, 0(R2)     ; drop DADDUI and BNEZ
        L.D       F8, 8(R1)
        stall
        MUL.D     F4, F8, F0
        L.D       F12, 8(R2)
        stall
        stall
```

```
        ADD.D     F12, F4, F12
        stall
        stall
        S.D       F12, 8(R2)
        DADDIU    R1, R1, #16   ; increment X index
        DADDIU    R2, R2, #16   ; increment Y index
        DSUBU     R3, R4, R1    ; test if done
        stall
        BNEZ      R3, foo       ; loop if not done
        stall                   ; 1-cycle delay slot
```

Next, interleave the instructions by putting the i^{th} instruction in the group of copied instructions right after the i^{th} instruction in the original sequence. These steps yield a schedule without violating data dependences. This is shown below.

```
foo:    L.D       F2, 0(R1)
        L.D       F8, 8(R1)
        MUL.D     F4, F2, F0
        MUL.D     F4, F8, F0
        L.D       F6, 0(R2)
        L.D       F12, 8(R2)
        ADD.D     F6, F4, F6
        ADD.D     F12, F4, F12
        stall
        S.D       F6, 0(R2)     ; drop DADDUI and BNEZ
        S.D       F12, 8(R2)
        DADDIU    R1, R1, #16
        DADDIU    R2, R2, #16
        DSUBU     R3, R4, R1
        stall
        BNEZ      R3, foo
        stall                   ; 1-cycle delay slot
```

Finally, instructions are reordered to cover any remaining stalls, including scheduling the branch delay slot, which yields the code shown below. In our code, this was done by moving the first `DADDIU` to cover the first stall, and then moving the second `S.D` to cover the delay slot. Note that the displacement for the `S.D` instruction has to be modified to -8 because this instruction occurs after R2 is incremented by 16.

```
foo:    L.D       F2, 0(R1)
        L.D       F8, 8(R1)
        MUL.D     F4, F2, F0
        MUL.D     F4, F8, F0
        L.D       F6, 0(R2)
        L.D       F12, 8(R2)
        ADD.D     F6, F4, F6
        ADD.D     F12, F4, F12
        DADDIU    R1, R1, #16
        S.D       F6, 0(R2)     ; drop DADDUI and BNEZ
        DSUBU     R3, R4, R1
        DADDIU    R2, R2, #16
        BNEZ      R3, foo
        S.D       F12, -8(R2)
```

The unrolled and scheduled loop will take 14 cycles x 50 iterations = 700 cycles.

[25 pts]

3- Consider the following code segment within a loop body:

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```

Here is the equivalent MIPS code assuming `aa` and `bb` are assigned to registers `R1` and `R2`, respectively.

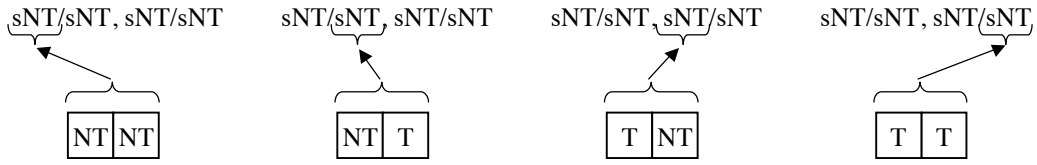
```
    DADDIU    R3, R1,#-2
    BNEZ      R3, L1        ; branch b1 (aa!=2)
    DADD      R1,R0,R0      ; aa=0
L1: DADDIU    R3, R2, #-2
    BNEZ      R3, L2        ; branch b2 (bb!=2)
    DADD      R2,R0,R0      ; bb=0
L2: DSUBU     R3,R1,R2      ; R3=aa-bb
    BEQZ      R3, L3        ; branch b3 (aa=bb)
```

Assume that the following list of 8 values of `aa` and `bb` are to be processed:

	iteration							
value	0	1	2	3	4	5	6	7
`aa`	0	2	0	2	0	2	0	2
`bb`	1	2	1	2	1	2	1	2

(a) Suppose 2-bit BPBs are used to predict the execution of the three branches in this loop. 2-bit BPBs flips when they are wrong two consecutive times and consist of four states: strongly not taken (sNT), weakly not taken (wNT), strongly taken (sT), and weakly taken (wT). Show the trace of predictions and the actual outcome of branches b1, b2, and b3 in the table shown below. The initial values of the 2–bit predictors are shown below. What are the prediction accuracies for b1, b2, and b3. What is the overall prediction accuracy?

(b) Suppose a two-level (2, 2) branch prediction is used predict the execution of the three branches in this loop. That is, in addition to the 2-bit predictor, a 2-bit global register (g) is used. Assume the 2-bit predictors are initialized to sNT and g is initialized to NT, NT, with the LSB representing the most recent branch outcome. Therefore, initial predictions, g, and their meaning are given below



Show the trace of predictions and updated g values for branches b1, b2, and b3 in the table shown below. What are the prediction accuracies for b1, b2, and b3? What is the overall prediction accuracy?

(c) Comment on the performance of 2-bit versus two-level predictors. That is, explain why one predictor performed better than the other.

Solutions

(a)

	0,1	2,2	0,1	2,2	0,1	2,2	0,1	2,2
b1 predicted	wNT	sT	wT	sT	wT	sT	sNT	wNT
b1 actual	**T**	**NT**	**T**	**NT**	**T**	**NT**	**T**	NT
b2 predicted	wNT	sT	wT	sT	wT	sT	sNT	wNT
b2 actual	**T**	**NT**	**T**	**NT**	**T**	**NT**	**T**	NT
b3 predicted	sNT	sNT	wNT	sNT	wNT	sNT	wNT	sNT
b3 actual	NT	T	NT	**T**	NT	**T**	NT	**T**

b1 prediction accuracy => 1/8=12.5%

b2 prediction accuracy => 1/8=12.5%
b3 prediction accuracy => 4/8=50%
Overall prediction accuracy = 6/24= 25%

(b) The table below shows the answer. The predictions (indicated in blue) used for b1, b2, and b3 are determined by g and g is updated by shifting in the branch outcome (indicated in red). Mispredictions are noted by √.

g=NT,T

aa,bb	**b1 prediction**	**g**	**b2 prediction**	**g**	**b3 prediction**	**g**
0,1	wNT/**sT**, sNT/sNT	T, **T**	sNT/wNT, sNT/**sT**	T, **T**	sT/sNT, sNT/**sNT**	T, **NT**
2,2	wNT/sT, **sNT**/sNT	NT, **NT**	**sNT**/wNT, sNT/sT	NT, **NT**	**sT**/sNT, sNT/sNT	NT, **T**
0,1	wNT/**sT**, sNT/sNT	T, **T**	sNT/wNT, sNT/**sT**	T, **T**	sT/sNT, sNT/**sNT**	T, **NT**
2,2	wNT/sT, **sNT**/sNT	NT, **NT**	**sNT**/wNT, sNT/sT	NT, **NT**	**sT**/sNT, sNT/sNT	NT, **T**
0,1	wNT/**sT**, sNT/sNT	T, **T**	sNT/wNT, sNT/**sT**	T, **T**	sT/sNT, sNT/**sNT**	T, **NT**
2,2	wNT/sT, **sNT**/sNT	NT, **NT**	**sNT**/wNT, sNT/sT	NT, **NT**	**sT**/sNT, sNT/sNT	NT, **T**
0,1	wNT/**sT**, sNT/sNT	T, **T**	sNT/wNT, sNT/**sT**	T, **T**	sT/sNT, sNT/**sNT**	T, **NT**
2,2	wNT/sT, **sNT**/sNT	NT, **NT**	**sNT**/wNT, sNT/sT	NT, **NT**	**sT**/sNT, sNT/sNT	NT, **T**

b1 prediction accuracy => 8/8=100%
b2 prediction accuracy => 8/8=100%
b3 prediction accuracy => 8/8=100%

Overall prediction accuracy = 24/24= 100%

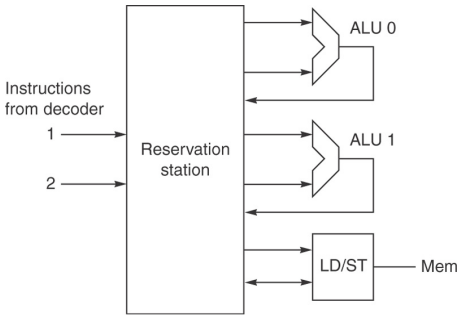
(c) The two-level predictor performed better than the 2-bit predictor because there is a strong correlation between (b1, b2) and b3. Actually, the reason why the prediction accuracy is 100% is because the initial predictions are based on the predictions when this loop ends (notice the same set of predictions for the last iterations). Thus, the predictors were already trained leading to very accurate results.

[20 pts]

4- Consider the following code sequence executing on the microarchitecture shown below. Assume that the ALUs can perform all arithmetic and branch operations, and that the centralized Reservation Station (RS) can dispatch at most one operation to each functional unit per cycle (i.e., one instruction to each ALU plus one instruction to LD/ST unit). If more than two ALU instructions can be dispatched, then instructions are dispatched in program order. Assume that dispatching instructions from RS to functional units requires one cycle and once instructions are in the function units they have the latencies shown on right. Also, assume functional unit results can be fully bypassed to subsequent instructions, i.e., when a result is available at cycle *t*, any instructions dependent on the result (and have all their operands available) can be dispatched at cycle *t*+1. All functional units are fully pipelined.

```
Loop:     L.D     F2, 0(Rx)
          DIV.D   F8, F2, F0
          MUL.D   F2, F6, F2
          L.D     F4, 0(Ry)
          ADD.D   F4, F0, F4
          ADD.D   F10, F8, F2
          DADDI   Rx, Rx, #8
          DADDI   Ry, Ry, #8
          S.D     F4, 0(Ry)
          DSUB    R20, R4, Rx
          BNEZ    R20, Loop
```

Latencies	Cycles
DIV.D	10
MUL.D	4
L.D	3
ADD.D	2
DADDI, S.D, BNEZ, DSUB	1



(a) Suppose all of the instructions from the code sequence above are present in the RS without register renaming at cycle 0. Indicate all the RAW, WAR, and WAW hazards and show how the RS should dispatch these instructions using a timing table similar to the one shown below. The first `L.D` instruction dispatched at cycle 1 is shown.

The red arrows indicate RAW hazards, blue arrows indicate WAR hazards, and green arrows indicate WAW hazards. These dependencies force most of the instructions to be dispatched serially. The dispatch timing is shown below.

```
Loop:   L.D    F2, 0(Rx)
        DIV.D  F8, F2, F0
        MUL.D  F2, F6, F2
        L.D    F4, 0(Ry)
        ADD.D  F4, F0, F4
        ADD.D  F10, F8, F2
        DADDI  Rx, Rx, #8
        DADDI  Ry, Ry, #8
        S.D    F4, 0(Ry)
        DSUB   R20, R4, Rx
        BNEZ   R20, Loop
```

Cycle	ALU0	ALU1	LD/ST
1			L.D F2,0(Rx)
2			L.D F4,0(Ry)
3			
4	DIV.D F8,F2,F0	MUL.D F2,F6,F2	
5	ADD.D F4,F0,F4	DADDI Rx, Rx, #8	
6	DADDI Ry, Ry, #8	DSUB R20, R4, Rx	
7	BNEZ R20, Loop		S.D F4,0(Ry)
8			
9			
10			
11			
12			
13			
14	ADD.D F10,F8,F2		
15			
16			
17			
18			
19			

(b) Now rewrite the code using register renaming. Assume the free list contains rename registers T0, T1, T2, T2, etc. Also, assume these registers can be used to rename both FP and integer registers. Suppose the code with registers renamed is resident in the RS at cycle 0. Show how the RS should dispatch these instructions out-of-order to obtain the optimal performance.

Solutions

The difference between the code in part (a) and the code below is that all the WAR and WAW dependencies are eliminated. This allows dispatching of the two DADDIs, DSUB, and BNEZ earlier, which results in the following execution timing. It may appear that there is no improvement in performance, despite the fact that these instructions can be dispatched earlier. This is true when you only look at one iteration of this loop. However, when multiple iterations are executed, dispatching DADDIs, DSUB, and BNEZ earlier allows instructions from the next iteration to be dispatched earlier resulting in performance improvement.

```
Loop:   L.D    T0, 0(Rx)     ; F2 renamed to T0
        DIV.D  T1, T0, F0    ; F8 renamed to T1
        MUL.D  T2, F6, T0    ; F2 renamed to T2
        L.D    T3, 0(Ry)     ; F4 renamed to T3
        ADD.D  T4, F0, T3    ; F4 renamed to T4
        ADD.D  T5, T1, T2    ; F10 renamed to T5
        DADDI  T6, Rx, #8    ; Rx renamed to T6
        DADDI  T7, Ry, #8    ; Ry renamed to T7
        S.D    T4, 0(T7)
        DSUB   T8, R4, T6    ; R20 renamed to T8
        BNEZ   T8, Loop
```

Cycle	ALU0	ALU1	LD/ST
1	DADDI T6,Rx,#8	DADDI T7,Ry,#8	L.D T0,0(Rx)
2	DSUB T8,R4,T6		L.D T3,0(Ry)
3	BNEZ T8, Loop		
4	DIV.D T1,T0,F0	MUL.D T2, F6, T0	
5	ADD.D T4,F0,T3		
6			
7			S.D T4,0(Ry)
8			
9			
10			
11			
12			
13			
14	ADD.D T5, T1, T2		
15			
16			
17			
18			
19			

(c) Part (b) assumes that the centralized RS contains all the instructions in the code sequence. But in reality, the entire code sequence of interest is not present in the RS, and thus the RS must choose to dispatch what it has. Suppose the RS is initially empty. In cycle 0, the first two register-renamed instructions of the code sequence appear in the RS. Further assume that the front-end (decoder and register renaming logic) will

continually supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. The contents of the RS for the Cycle 0 and Cycle 1 are shown below.

Solutions

The timing below illustrates the content of the RS and which instruction(s) (in red) will be dispatched in the following cycle. As these instructions are dispatched they disappear from the RS and new instructions are brought into the RS.

Cycle 0

```
L.D    T0,0(Rx)
DIV.D  T1,T0,F0
```

Cycle 1

```
DIV.D  T1,T0,F0
MUL.D  T2,F6,T0
L.D    T3,0(Ry)
```

Cycle 2

```
DIV.D  T1,T0,F0
MUL.D  T2,F6,T0

ADD.D  T4,F0,T3
ADD.D  T5,T1,T2
```

Cycle 3

```
DIV.D  T1,T0,F0
MUL.D  T2,F6,T0

ADD.D  T4,F0,T3
ADD.D  T5,T1,T2
DADDI  T6,Rx,#8
DADDI  T7,Ry,#8
```

Cycle 4

```
ADD.D  T4,F0,T3
ADD.D  T5,T1,T2
DADDI  T6,Rx,#8
DADDI  T7,Ry,#8
S.D    T4,0(T7)
DSUB   T8,R4,T6
```

Cycle 5

```
ADD.D  T5,T1,T2

DADDI  T7,Ry,#8
S.D    T4,0(T7)
DSUB   T8,R4,T6
BNEZ   T8,Loop
```

Cycle 6

```
ADD.D  T5,T1,T2

S.D    T4,0(T7)

BNEZ   T8,Loop
```

Cycle 7

```
ADD.D  T5,T1,T2
```

Cycle 8

```
ADD.D  T5,T1,T2
```

…

Cycle 13

```
ADD.D  T5,T1,T2
```

Cycle 14

...

Cycle	ALU0	ALU1	LD/ST
1			L.D T0,0(Rx)
2			L.D T3,0(Ry)
3			
4	DIV.D T1,T0,F0	MUL.D T2,F6,T0	
5	ADD.D T4,F0,T3	DADDI T6,Rx,#8	
6	DADDI T7,Ry,#8	DSUB T8,R4,T6	
7	BNEZ T8,Loop		S.D T4,0(Ry)
8			
9			
10			
11			
12			
13			

14	`ADD.D  T5,T1,T2`		
15			
16			
17			
…			

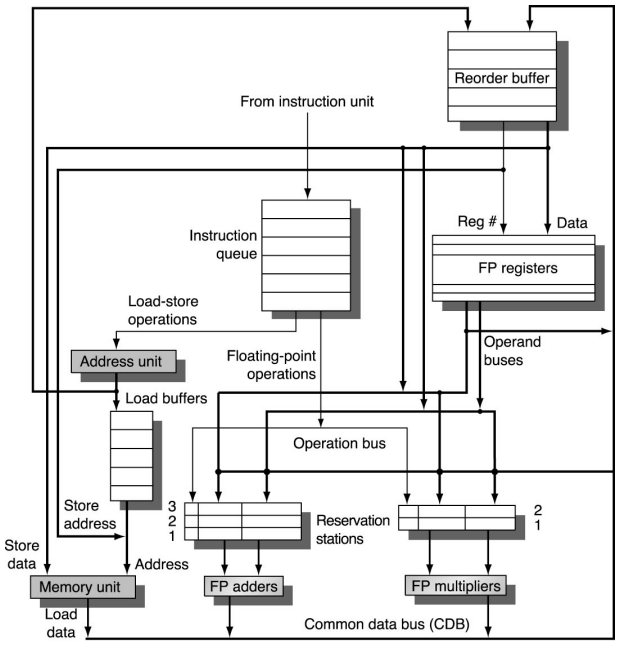
[20 pts]

5- Consider the implementation of the Tomasulo's algorithm with Reorder Buffer (ROB) shown below. It consists of four stages: Issue, Execute, Write-back, and Commit. Simulate the execution of the following piece of code using Tomasulo's algorithm and show the contents of the RS, ROB, and register file entries for each cycle (shown in the following page).

- An ROB entry contains three fields:
  - Committed – Yes (committed) and No (not committed)
  - Dest – destination register identifier
  - Data – value
- In addition to the Busy and Value fields, a register contains ROB # that indicates the ROB entry that will generate the result.
- In addition to Op, Busy, V_j, V_k, Q_j, and Q_k fields, a RS contains Dest field that indicates the ROB entry where the result will be written to. (I left out Busy field because of lack of space).

Assume the following: (1) Dual issue, write-back, and commit, i.e., two instructions can be issued, forwarded to the CDB, and committed per cycle; (2) add/subtract latency is 1 cycle and multiply/divide latency is 3 cycles; (3) an instruction can begin execution in the same cycle that it is issued, assuming all dependencies are satisfied. Also, forwarded results are immediately available for use in the next cycle. *Note that this code takes exactly 8 cycles to complete!*

```
MUL.D  F0,F2,F4
SUB.D  F8,F2,F6
DIV.D  F0,F0,F6
ADD.D  F6,F8,F2
```



	Op	Dst	V_j	Q_j	V_k	Q_k
1	SUB.D	1	3.5		7.8	
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4	MUL.D	0	3.5		10.0	
5						

Mult/Div

	Comm.	Dst	Data
0	No	0	
1	No	8	
2			
3			

ROB

	Busy	ROB#	Data
0	Yes	0	
2	No		3.5
4	No		10.0
6	No		7.8
8	Yes	1	6.0
10			

Register File

Value=4.3, Dest=1 (ROB#)

	Op	Dst	V_j	Q_j	V_k	Q_k
1	ADD.D	3	-4.3	1	3.5	
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4	MUL.D	0	3.5		10.0	
5	DIV.D	2		0	7.8	

Mult/Div

	Comm.	Dst	Data
0		0	
1		8	4.3
2		0	
3		6	

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	Yes	1	6.0
10			

Register File

Value=-0.8, Dest=3 (ROB#)

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4	MUL.D	0	3.5		10.0	
5	DIV.D	2		0	7.8	

Mult/Div

	Comm.	Dst	Data
0	No	0	
1	No	8	4.3
2	No	0	
3	No	6	-0.8

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	Yes	1	6.0
10			

Register File

Value=35.0, Dest=0 (ROB#)

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5	DIV.D	2	35	0	7.8	

Mult/Div

	Comm.	Dst	Data
0	No	0	35
1	No	8	4.3
2	No	0	
3	No	6	-0.8

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	Yes	1	6.0
10			

Register File

Dest=0 does not match ROB#

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5	DIV.D	2	35	0	7.8	

Mult/Div

	Comm.	Dst	Data
0	Yes	0	35
1	Yes	8	4.3
2	No	0	
3	No	6	-0.8

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	**No**	1	**4.3**
10			

Register File

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5	DIV.D	2	35	0	7.8	

Mult/Div

	Comm.	Dst	Data
0	Yes	0	35
1	Yes	8	4.3
2	No	0	
3	No	6	-0.8

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	**No**	1	**4.3**
10			

Register File

Value=4.49 Dest=2 (ROB#)

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5					7	

Mult/Div

	Comm.	Dst	Data
0	Yes	0	35
1	Yes	8	4.3
2	No	0	4.49
3	No	6	-0.8

ROB

	Busy	ROB#	Data
0	Yes	2	
2	No		3.5
4	No		10.0
6	Yes	3	7.8
8	**No**	1	**4.3**
10			

Register File

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5					7	

Mult/Div

	Comm.	Dst	Data
0	Yes	0	35
1	Yes	8	4.3
2	Yes	0	4.49
3	Yes	6	-0.8

ROB

	Busy	ROB#	Data
0	**No**	2	**4.49**
2	No		3.5
4	No		10.0
6	**No**	3	**-0.8**
8	**No**	1	**4.3**
10			

Register File

	Op	Dst	V_j	Q_j	V_k	Q_k
1						
2						
3						

Add/Sub

	Op	Dst	V_j	Q_j	V_k	Q_k
4						
5						

Mult/Div

	Comm.	Dst	Data
0			
1			
2			
3			

ROB

	Busy	ROB#	Data
0			
2			3.5
4			10.0
6			7.8
8			6.0
10			

Register File