ECE 570
High-Performance Computer Architecture
Winter 2014
Solutions Set #2

[25 pts]

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

```
if (d==0)
    d=1;
if (d==1)
```

The code shown below is a typical assembly code sequence for this segment, assuming that d is assigned to R1:

```
     BNEZ      R1, L1       ; branch b1   (d!=0)
     DADDIU    R1, R0, #1   ; d==0, so d=1
L1:  DADDIU    R3, R1, #-1
     BNEZ      R3, L2       ; branch b2   (d!=1)
…    …         …
L2:
```

Assume that the following list of 8 values of d is to be processed:
1, 2, 1, 2, 1, 2, 1, 2

(a) Suppose 2-bit BPBs are used to predict the execution of the two branches in this loop. 2-bit BPBs consists of four states: strongly not taken (sNT), weakly not taken (wNT), strongly taken (sT), and weakly taken (wT), and behave according to Figure 2.4 in the text. Show the trace of predictions and the actual outcomes of branches b1 and b2. Assume initial values of the 2–bit predictors are sNT. What are the prediction accuracies for b1 and b2? What is the overall prediction accuracy?

	1	2	1	2	1	2	1	2
b1 predicted	sNT	wNT	sT	sT	sT	sT	sT	sT
b1 actual	T	T	T	T	T	T	T	T
b2 predicted	sNT	sNT	wNT	sNT	wNT	sNT	wNT	sNT
b2 actual	NT	T	NT	T	NT	T	NT	T

b1 prediction accuracy => 6/8=75%
b2 prediction accuracy => 4/8=50%
Overall prediction accuracy = 10/16= 62.5%

(b) Suppose two-level (1, 2) branch prediction is used predict the execution of the two branches in this loop. That is, in addition to the 2-bit predictor, a 1-bit global register (g) is used. Assume the 2-bit predictors are initialized to sNT and g is initialized to NT. Show the trace of predictions and the actual outcomes of branches b1 and b2. What are the prediction accuracies for b1 and b2? What is the overall prediction accuracy?

The table below shows the answer. The predictions used for b1 and b2 are indicated by the boldface font in the table. If the correlation bit, i.e. the previous branch outcome, has the value "Not Taken" then the left prediction is used. If the previous branch outcome is "taken" then the right prediction is used. Mispredictions are noted in the "b1 action" and "b2 action" columns.

d = ?	**b1 prediction**	**b1 action**	**New b1 prediction**	**b2 prediction**	**b2 action**	**Newb2 prediction**
1	**sNT**/sNT	T (miss)	wNT/sNT	sNT/**sNT**	NT	sNT/sNT
2	**wNT**/sNT	T (miss)	sT/sNT	sNT/**sNT**	T (miss)	sNT/wNT
1	sT/**sNT**	T (miss)	sT/wNT	sNT/**wNT**	NT	sNT/sNT
2	**sT**/wNT	T	sT/wNT	sNT/**sNT**	T (miss)	sNT/wNT
1	sT/**wNT**	T (miss)	sT/sT	sNT/**wNT**	NT	sNT/sNT
2	**sT**/sT	T	sT/sT	sNT/**sNT**	T (miss)	sNT/wNT
1	sT/**sT**	T	sT/sT	sNT/**wNT**	NT	sNT/sNT
2	**sT**/sT	T	sT/sT	sNT/**sNT**	T (miss)	sNT/wNT

b1 prediction accuracy => 4/8=50%
b2 prediction accuracy => 4/8=50%
Overall prediction accuracy = 8/16= 50%

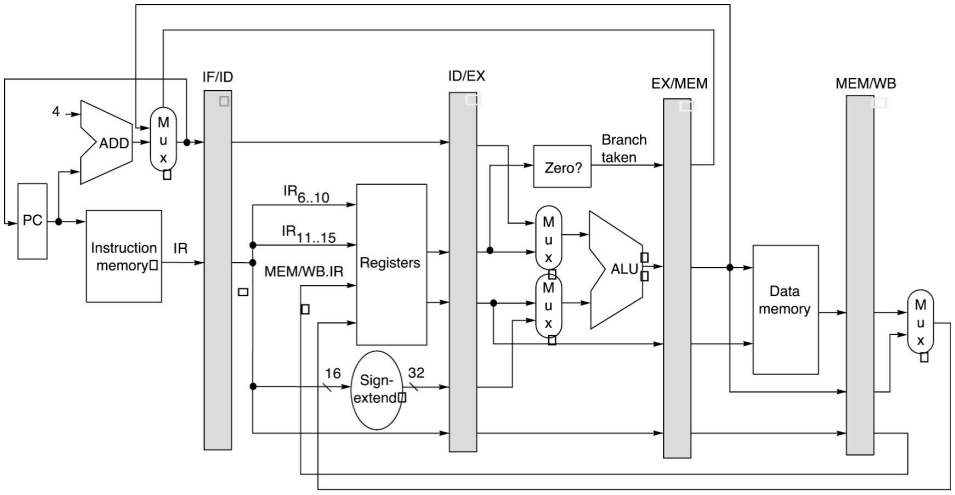
(c) Comment on the performance of 2-bit versus two-level predictors. That is, explain why one predictor performed better than the other.

The reason why 2-bit predictor performed similar to the two-level predictor is that there is no correlation between b1 and b2. That is, b1 will never be not taken because the values are never 0. Therefore, whether b2 is taken or not is totally independent of the outcomes of b1.

[25 pts]

2- Consider the 5-stage pipeline datapath shown below. Consider a branch target buffer (BTB) that buffers both unconditional branches and conditional branches that are taken. If necessary, the BTB is updated in the MEM stage. Assume the BTB has a miss penalty is 1 cycle (i.e., the pipeline is stalled for one cycle while the instruction in the MEM stage that cause the BTB miss updates the BTB in the IF stage), 90% BTB hit rate, and the following:

Conditional branch	35%
of which	55% are taken
Unconditional branch	5%



(a) What is the CPI for the 5-stage pipeline without BTB that assumes branch not taken.

Based on the figure, the branch penalty is 3 cycles. Since the branch penalty of unconditional branch is the same as conditional branch, we have

$$CPI = 1 + bp_b p_t + bp_{ub} = 1 + 3\times0.35\times0.55 + 3\times0.05 = 1.73$$

(b) What is the CPI for the 5-stage pipeline with BTB? How much faster is the design with BTB?

There are three cases that require additional cycles: (1) there is a BTB miss and the branch is taken, (2) there is a BTB hit but the branch is not taken, and (3) an unconditional branch misses on the BTB. Thus, we have

$$CPI = 1+ (b+c)p_m p_b p_t + b(1-p_m)p_b(1-p_t) + (b+c)p_m p_{ub}$$
$$CPI = 1+ (3+1)\times0.1\times0.35\times0.55 + 3\times0.9\times0.35\times0.45 + (3+1)\times0.1\times0.05$$
$$= 1+0.077+0.425+0.02 = 1.522$$

Thus, BTB design is 1.73/1.522=1.137 times faster.

(c) Consider enhancing the BTB with branch folding for both conditional and unconditional branches. Branch folding buffers the actual target instruction in the BTB to implement zero-cycle unconditional branches and sometimes zero-cycle conditional branches. What is the resulting CPI? How much faster is the BTB with branch folding compared to without branch folding?

We can find out which portions are saved by developing a more complete equation than the one used in part (b).

$$CPI = (1-p_b-p_{ub}) + [(1+b)p_t+(1-p_t)+cp_t]p_bp_m+[p_t+ (1+b)(1-p_t)]p_b(1-p_m) + [(1+b)p_m+(1-p_m)+cp_m]p_{ub}$$

Rearranging this equation gives you the same answer is in part (b).
The terms $p_tp_b(1-p_m)$ and $(1-p_m)p_{ub}$ indicates savings achieved, i.e., (1) conditional branch is taken and the target instruction is found in BTB, and (2) the target instruction for the unconditional branch is found in BTB. This results in

$$CPI = (1-p_b-p_{ub}) + [(1+b)p_t+ (1-p_t) + cp_t]p_bp_m+[(1+b)(1-p_t)]p_b(1-p_m) + [(1+b)p_m +cp_m]p_{ub}$$
$$CPI = 1-0.35-0.05+[4\times0.55+0.45+1\times0.55]\times0.35\times0.1+[4\times0.45]\times0.35\times0.9 + [4\times0.1+ 1\times0.1]\times0.05$$
$$CPI = 0.6 + 0.112 + 0.567 + 0.025 = 1.304$$

Thus, 1.522/1.304 = 1.167 times faster

We could have also simply figured out the two case where savings are achieved, i.e., $p_tp_b(1-p_m)$ and $(1-p_m)p_{ub}$

$$CPI_{saving} = 1.522 – p_tp_b(1-p_m) – (1-p_m)p_{ub}$$
$$CPI_{saving} = 1.522 – 0.55\times0.35\times0.9 – 0.9\times0.05 = 1.522 – 0.173 – 0.045 = 1.304,$$

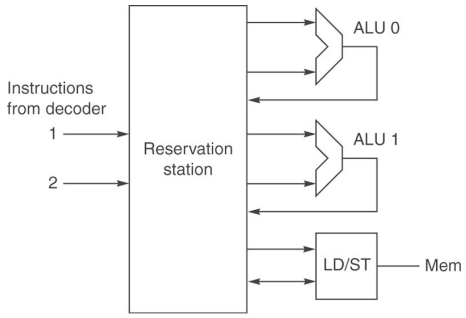
which is the same as before!

[25 pts]

3- 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 LAU 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)
        MUL.D  F2, F0, F2
        DIV.D  F8, F0, F2
        L.D    F4, 0(Ry)
        ADD.D  F4, F0, F4
        ADD.D  F10, F8, F2
        S.D    F4, 0(Ry)
        DADDI  Rx, Rx, #8
        DADDI  Ry, Ry, #8
        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)
        MUL.D  F2, F0, F2
        DIV.D  F8, F0, F2
        L.D    F4, 0(Ry)
        ADD.D  F4, F0, F4
        ADD.D  F10, F8, F2
        S.D    F4, 0(Ry)
        DADDI  Rx, Rx, #8
        DADDI  Ry, Ry, #8
        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	MUL.D F2,F0,F2	DADDI Rx,Rx,#8	
5	ADD.D F4,F0,F4	DSUB R20,R4,Rx	
6	BNEZ R20, Loop		
7			S.D F4,0(Ry)
8	DIV.D F8,F0,F2	DADDI Ry,Ry,#8	
9			
10			
11			
12			
13			
14			
15			
16			
17			
18	ADD.D F10,F8,F2		
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.

The difference between the code in part (a) and here 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
        MUL.D  T1, F0, T0    ; F1 renamed to T1
        DIV.D  T2, F0, T1    ; F8 renamed to T2
        L.D    T3, 0(Ry)     ; F4 renamed to T3
        ADD.D  T4, F0, T3    ; F4 renamed to T4
        ADD.D  T5, T2, T0    ; F10 renamed to T5
        S.D    T4, 0(Ry)
        DADDI  T6, Rx, #8    ; Rx renamed to T6
        DADDI  T7, Ry, #8    ; Ry renamed to T7
        DSUB   T8, R4, T6    ; T8 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	MUL.D T1,F0,T0		
5	ADD.D T4,F0,T3		
6			
7			S.D T4,0(Ry)
8	DIV.D T2,F0,T1		
9			
10			
11			
12			
13			
14			
15			
16			
17			
18	ADD.D T5,T2,T0		
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 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.

The timing below illustrates the content of the RS and which instruction(s) (in red) will be dispatched. 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)
MUL.D  T1,F0,T0
```

Cycle 1

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

Cycle 2

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

ADD.D  T4,F0,T3
ADD.D  T5,T2,T0
```

Cycle 3

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

ADD.D  T4,F0,T3
ADD.D  T5,T2,T0
S.D    T4,0(Ry)
DADDI  T6,Rx,#8
```

Cycle 4

```
DIV.D  T2,F0,T1

ADD.D  T4,F0,T3
ADD.D  T5,T2,T0
S.D    T4,0(Ry)

DADDI  T7,Ry,#8
DSUB   T8,R4,T6
```

Cycle 5

```
DIV.D  T2,F0,T1

ADD.D  T5,T2,T0
S.D    T4,0(Ry)

DSUB   T8,R4,T6
BNEZ   T8,Loop
```

Cycle 6

```
DIV.D  T2,F0,T1

ADD.D  T5,T2,T0
S.D    T4, 0(Ry)

BNEZ   T8,Loop
```

Cycle 7

```
DIV.D  T2,F0,T1

ADD.D  T5,T2,T0
```

Cycle 8

```
ADD.D  T5,T2,T0
```

…

Cycle 17

```
ADD.D  T5,T2,T0
```

Cycle 18

```
...
```

Cycle	ALU0	ALU1	LD/ST
1			L.D T0,0(Rx)
2			L.D T3,0(Ry)
3			
4	MUL.D T1,F0,T0	DADDI T6,Rx,#8	
5	ADD.D T4,F0,T3	DADDI T7,Ry,#8	
6	DSUB T8,R4,T6		
7	BNEZ T8,Loop		S.D F4,0(Ry)
8	DIV.D T2,F0,T1		
9			
10			
11			
12			
13			
14			
15			
16			
17			
18	ADD.D T5,T2,T0		

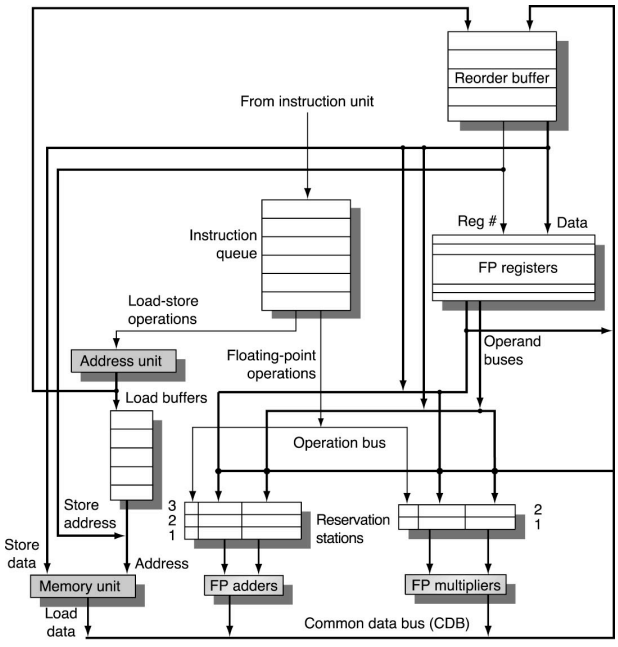
[25 pts]

4- 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 latency is 1 cycle and multiply latency is 2 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 7 cycles to complete!*

```
ADD.D  F4,F0,F8
MUL.D  F2,F0,F4
ADD.D  F4,F4,F8
MUL.D  F8,F4,F2
```



Adder

	Op	Dest	V_j	Q_j	V_k	Q_k
1						
2						
3						

Mult/Div

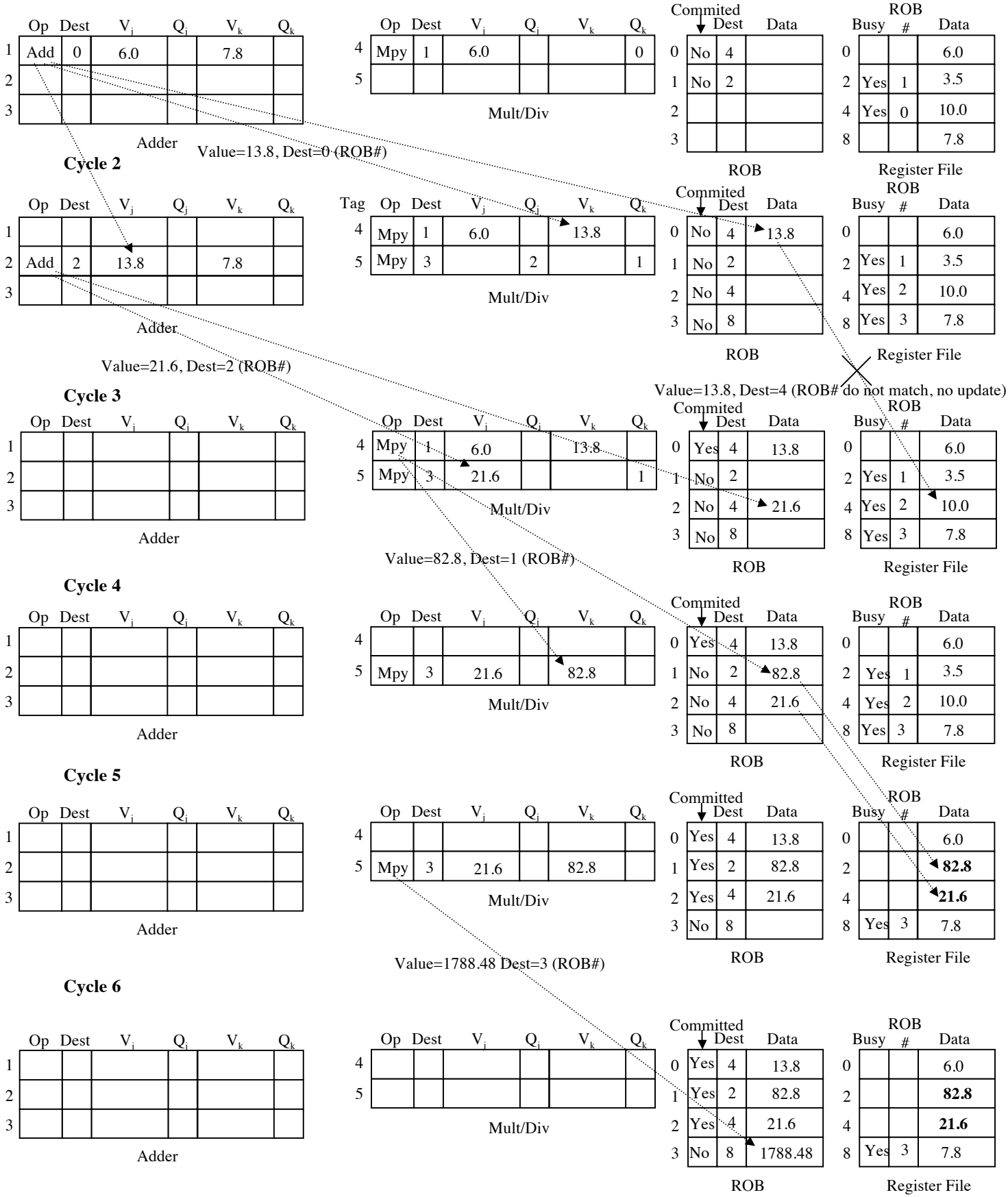
	Op	Dest	V_j	Q_j	V_k	Q_k
4						
5						

ROB

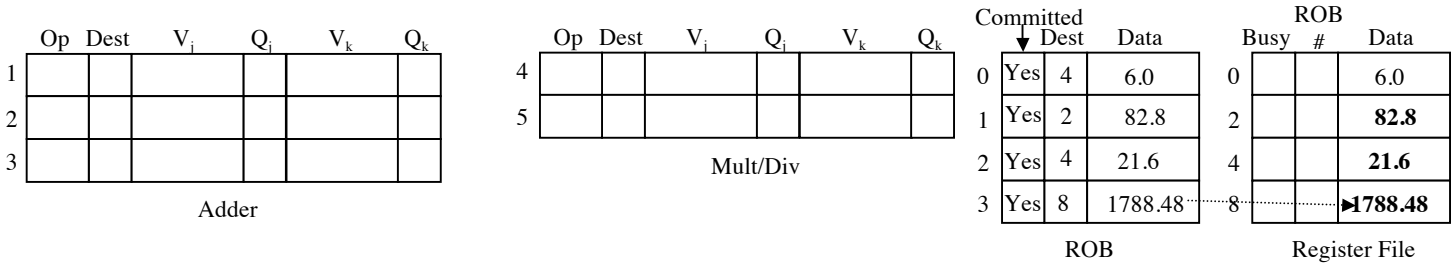
	Committed	Dest	Data
0			
1			
2			
3			

Register File

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


Op Dest Vj Qj Vk Qk
1 Add 0 6.0 7.8
2
3
Adder
Op Dest Vj Qj Vk Qk
4 Mpy 1 6.0 0
5
Mult/Div
Commited
Dest Data
0 No 4
1 No 2
2
3
ROB
Busy # Data
0 6.0
2 Yes 1 3.5
4 Yes 0 10.0
8 7.8
Register File
Value=13.8, Dest=0 (ROB#)
Cycle 2
Op Dest Vj Qj Vk Qk
1
2 Add 2 13.8 7.8
3
Adder
Tag Op Dest Vj Qj Vk Qk
4 Mpy 1 6.0 13.8
5 Mpy 3 2 1
Mult/Div
Commited
Dest Data
0 No 4 13.8
1 No 2
2 No 4
3 No 8
ROB
Busy # Data
0 6.0
2 Yes 1 3.5
4 Yes 2 10.0
8 Yes 3 7.8
Register File
Value=21.6, Dest=2 (ROB#)
Value=13.8, Dest=4 (ROB# do not match, no update)
Cycle 3
Op Dest Vj Qj Vk Qk
1
2
3
Adder
Op Dest Vj Qj Vk Qk
4 Mpy 1 6.0 13.8
5 Mpy 3 21.6 1
Mult/Div
Commited
Dest Data
0 Yes 4 13.8
1 No 2
2 No 4 21.6
3 No 8
ROB
Busy # Data
0 6.0
2 Yes 1 3.5
4 Yes 2 10.0
8 Yes 3 7.8
Register File
Value=82.8, Dest=1 (ROB#)
Cycle 4
Op Dest Vj Qj Vk Qk
1
2
3
Adder
Op Dest Vj Qj Vk Qk
4
5 Mpy 3 21.6 82.8
Mult/Div
Commited
Dest Data
0 Yes 4 13.8
1 No 2 82.8
2 No 4 21.6
3 No 8
ROB
Busy # Data
0 6.0
2 Yes 1 3.5
4 Yes 2 10.0
8 Yes 3 7.8
Register File
Cycle 5
Op Dest Vj Qj Vk Qk
1
2
3
Adder
Op Dest Vj Qj Vk Qk
4
5 Mpy 3 21.6 82.8
Mult/Div
Committed
Dest Data
0 Yes 4 13.8
1 Yes 2 82.8
2 Yes 4 21.6
3 No 8
ROB
Busy # Data
0 6.0
2 82.8
4 21.6
8 Yes 3 7.8
Register File
Value=1788.48 Dest=3 (ROB#)
Cycle 6
Op Dest Vj Qj Vk Qk
1
2
3
Adder
Op Dest Vj Qj Vk Qk
4
5
Mult/Div
Committed
Dest Data
0 Yes 4 13.8
1 Yes 2 82.8
2 Yes 4 21.6
3 No 8 1788.48
ROB
Busy # Data
0 6.0
2 82.8
4 21.6
8 Yes 3 7.8
Register File

Cycle 7


Op Dest V_j Q_j V_k Q_k
1
2
3
Adder
Op Dest V_j Q_j V_k Q_k
4
5
Mult/Div
Committed
Dest Data
0 Yes 4 6.0
1 Yes 2 82.8
2 Yes 4 21.6
3 Yes 8 1788.48
ROB
ROB
Busy # Data
0 6.0
2 82.8
4 21.6
8 1788.48
Register File