CS704 – Advanced Computer Architecture-II

Solution to Assignment 2

Instructions to Solve Assignments

The purpose of assignments is to give you hands on practice. It is expected that students will solve the assignments themselves. Following rules will apply during the evaluation of assignment.

- Cheating from any source will result in zero marks in the assignment.
- Any student found cheating in any two of the assignments submitted will be awarded "F" grade in the course.
- No assignment after due date will be accepted.

Question 1: Total Points (10+5+5=20)

Following code lines are written in a high level language:
a = c + d;
b = c + e;

The corresponding instructions for MIPS are:
LW R1, 0(R0)
LW R2, 4(R0)
ADD R3, R1, R2
SW R3, 12(R0)
LW R4, 8(R0)
ADD R5, R1, R4
SW R5, 16(R0)
These instructions are to be executed on a pipelined processor with forwarding.

(a) Identify hazards by showing the execution of these instructions per cycle bases.

Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
LW R1, 0(R0)	IF	ID	EX	MEM	WB								
LW R2, 4(R0)		IF	ID	EX	MEM	WB							
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB					
SW R3, 12(R0)				IF	stall	ID	EX	MEM	WB				
LW R4, 8(R0)					stall	IF	ID	EX	MEM	WB			
ADD R5, R1, R4							IF	ID	stall	EX	MEM	WB	
SW R5, 16(R0)								IF	stall	ID	EX	MEM	WB

(b) Reorder these instructions to avoid any pipeline stalls.

LW R1, 0(R0)
LW R2, 4(R0)
LW R4, 8(R0)
ADD R3, R1, R2
SW R3, 12(R0)
ADD R5, R1, R4
SW R5, 16(R0)

(c) How many cycles are saved after executing the reordered instructions?

Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
LW R1, 0(R0)	IF	ID	EX	MEM	WB						
LW R2, 4(R0)		IF	ID	EX	MEM	WB					
LW R4, 8(R0)			IF	ID	EX	MEM	WB				
ADD R3, R1, R2				IF	ID	EX	MEM	WB			
SW R3, 12(R0)					IF	ID	EX	MEM	WB		
ADD R5, R1, R4						IF	ID	EX	MEM	WB	
SW R5, 16(R0)							IF	ID	EX	MEM	WB

2 cycles are saved.

Question 2: Total Points (10+10+10=30)

Scheduling branch delay slots (see the three ways in Figure 2.1) can improve performance. Assume a single branch delay slot and an instruction pipeline that determines branch outcome in the second stage.

(a) For a delayed branch instruction, what is the penalty for each branch delay slot scheduling scheme if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?

(b) A cancel-if-not-taken branch instruction (also called branch likely and implemented in MIPS) does not execute the instruction in the delay slot if the branch is not taken. Thus, a compiler need not be as conservative when filling the delay slot. For each branch delay slot scheduling scheme, what is the penalty if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?

(c) Assume that an instruction set has a delayed branch and a cancel-if-not-taken branch. When should a compiler use each branch instruction and from where should the slot be filled?

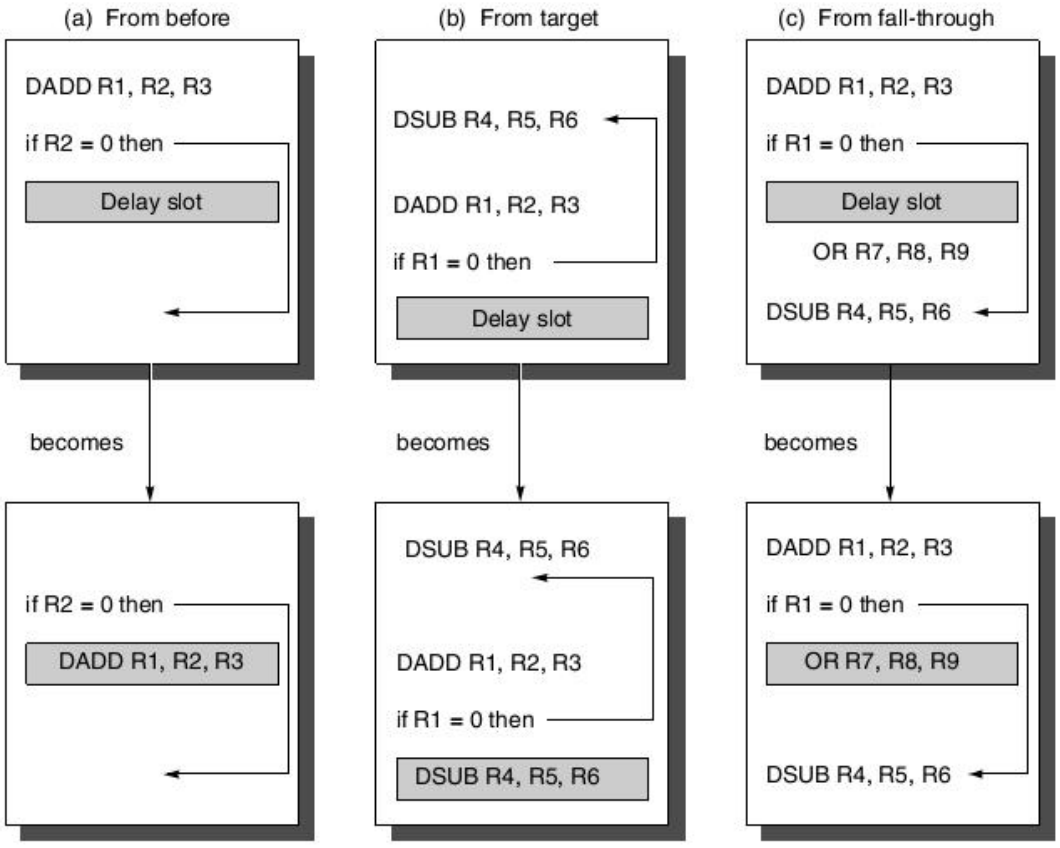

Figure 2.1 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a) the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b) the branch delay slot is scheduled from the target of the branch; usually the target

instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.

Solution:

(a) Delayed Branch—Penalties

	From Before	From Fall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	0	0	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. If the compiler takes an instruction from fall-through, it must make sure that there are no dependencies between that instruction and instructions from the target of the branch. If it takes an instruction from target, the compiler must make sure that there are no dependencies between that instruction and the instruction from fall-through. Please note that these rules only apply to true dependencies. If an instruction from the correct path overwrites the value produced from the instruction in the branch delay slot, then essentially we are correcting the value stored in that particular register. However, we would still have to pay the penalty.

(b) Cancel-If-Not-Taken—Penalties

	From Before	From Fall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	1	1	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. However we have to be concerned that the instruction in the delay slot might be cancelled. Therefore, the compiler has to account for this possibility. Since there would be a penalty regardless of the branch being taken or not, the compiler shouldn't take an instruction from fall-through. Since the branch delay slot instruction is canceled if not taken, the compiler does not have to worry about dependencies when taking an instruction from target to fill the slot.

(c)

If the compiler seeks to fill the branch delay slot from before the branch, then the delayed branch scheme is the best. This scheme will ensure the execution of the branch delay slot instruction regardless of the branch's outcome. If the compiler seeks to fill the branch delay slot from fall-through, then the branch delayed scheme is the best. As pointed out in part (b), there is a penalty using this scheme regardless of the branch's outcome.

If the compiler seeks to fill the branch delay slot from target, cancel-if-not-taken is better since the compiler would not have to worry about ensuring correctness when the branch is not taken.

Question 3: Total Points (10+10=20)

Tomasulo's algorithm has a disadvantage: Only one result can complete per clock, per CDB.

(a) Use the hardware configuration from Figure 3.1 and the FP latencies from Figure 3.2. Find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

(b) Generalize you result from part (a) by describing the characteristic of any code sequence that will eventually experience structural hazard stall given n CDBs.

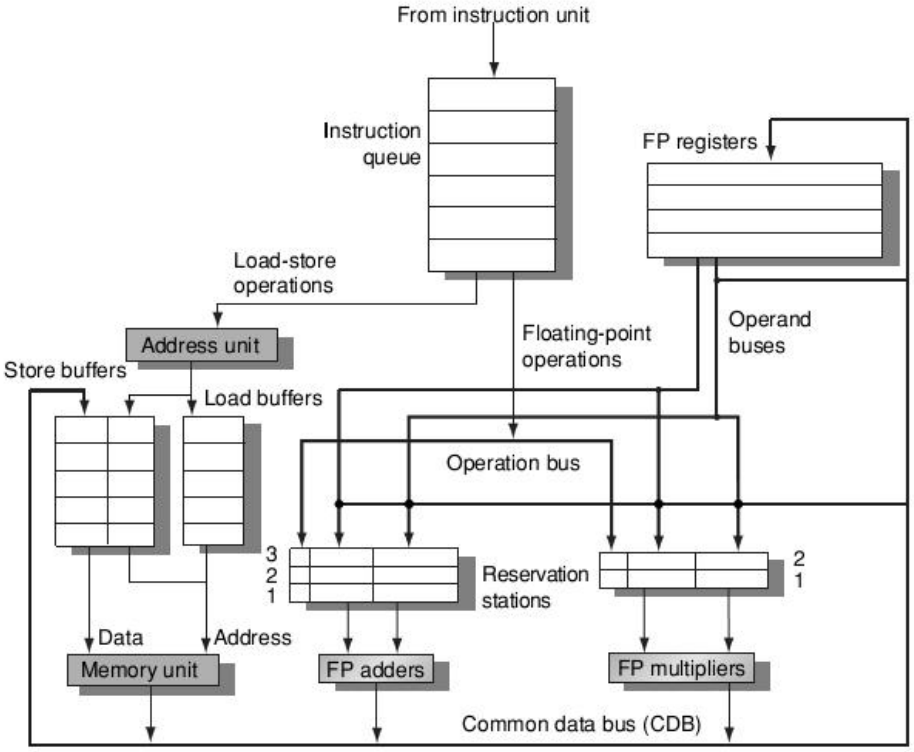

Figure 3.1 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP

adders implement addition and subtraction, and the FP multipliers do multiplication and division.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	3
Integer operation (including load)	Any	0

Figure 3.2 Pipeline latencies where latency is number of cycles between producing and consuming instruction.

Solution:

(a)

There are many code sequences that stall a Tomasulo-based CPU. Such code sequences contain two or more instructions that attempt to write their results in the same cycle. One such sequence is the following:

```
frob:  DMUL
       AND
       OR
       LD
```

(b)

The code characteristic is to have *n + 1* instructions using distinct function units that all complete in the same clock cycle. Because a machine with *n* CDBs cannot write more than *n* results per clock cycle, one of the instructions must stall. To have *n + 1* instructions complete simultaneously requires an appropriate set of execution initiation times in the function units relative to function unit latency and probably some operand sharing.

Question 4: Total Points (15)

Consider the following code

```
for (i=1; i<=100; i=i+1)
{
        A[i+1] = A[i] + C[i];          /* S1 */
        B[i+1] = B[i] + A[i+1];        /* S2 */
}
```

Write an optimal assembly code for the loop. Is it safe to use loop un-rolling here? Why or why not?

Solution:

Assuming that R1, R2 and R3 contain the starting addresses of arrays A, B and C respectively, here is the code:

```
Loop:  L.D      F0, 0(R1)
       L.D      F1, 0(R3)
       ADD.D    F2, F0, F1
       S.D      F2, 8(R1)
       L.D      F3, 0(R2)
       ADD.D    F4, F3, F2
       S.D      F4, 8(R2)
       DADDUI   R1, R1, #8
       DADDUI   R2, R2, #8
       DADDUI   R3, R3, #8
       BNE      R1, R4, Loop
```

There are two dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
2. S2 uses the value, A[i+1], computed by S1 in the same iteration.

The dependence of statement S1 on an earlier iteration of S1 is called loop-carried dependence. This dependence forces successive iterations of this loop to execute in series.

The second dependence above (S2 depending on S1) is within an iteration and is not loop-carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order.

Question 5: Total Points (15)

Based on the paper "**Instruction Issue Logic for High-Performance Interruptable Pipelined Processors**", explain how the Register Update Unit (RUU) is used to resolve data dependencies and maintain precise interrupts.

Solution:

Research-paper based question.