Notes-04

Data Dependences

Informal: If an instruction j uses a result produced by a (previous) instruction i, then j is data depedent on i. Of course, this dependence is transitive: if k is dependent on i and j is dependent on k, then j is dependent on i. A dependence is a property of the program.

When a program is executed in a pipeline, overlapped execution occurs. A dependence is likely to cause a pipeline hazard. No matter how we execute the program, whether we reorder the program instructions, whether we run it in an overlapped fashion (pipelining), whether we run two instructions simultaneously in two separate pipelines (superscalar architectures), **the end result MUST BE the same as that for strictly sequential execution of the instructions**. A hazard indicates the potential **VIOLATION of this sequential semantics**.

The solu

Formal: Data hazards are classified as follows:

(i) RAW (read after write)—j tries to read a source before I writes to it, so j incorrectly reads the old value.

(ii) WAW (write after write) – j tries to write an operand before it is written by i. The writes end up being performed in the wrong order.

(iii) WAR (write after read) – j tries to write a destination before it is read by I, so I incorrectly gets the new value.

Branch Hazards

Informal: Consider the following program:

i1 (if condition1 then jmp label10)
i2
i3
i4

------
label10: i10

Here, if condition1 is true, instruction i10 should execute immediately after i2. But with pipelining, instruction i2 enters the pipeline BEFORE i1 has finished execution. Thus i2 starts execution after i1. Clearly, this is undesirable.

In the following diagram, we show how this problem can be tackled. Essentially, i2 has to be **stalled** by one cycle.

Figure C.11 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Figure C.11 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Pipeline Scheduling

Informal: Hazards can be tackled by inserting stalls However, this results in inefficiency. A solution is to **do useful work, i.e. execute some instruction, instead of stalling**. Thus, if the instruction sequence is i1, i2, i3, i4, i5, etc and a stall is required between i1 and i2, then some instruction ik (k >2) can be executed after i1, provided **sequential semantics is preserved**. The net effect is the same as far as the hazard is concerned: the time delay between i1 and i2 is ensured, although not by stalling, but, instead, by out-of-order execution of instruction ik.

Example-4.1

Figure 3.2 shows the latencies of floating-point operations.These latencies, or delay, are required to avoid pipeline hazards.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

Now consider the following program fragment.

The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:   L.D      F0,0(R1)      ;F0=array element
        ADD.D    F4,F0,F2      ;add scalar in F2
        S.D      F4,0(R1)      ;store result
        DADDUI   R1,R1,#-8     ;decrement pointer
                               ;8 bytes (per DW)
        BNE      R1,R2,Loop    ;branch R1!=R2
```

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies from Figure 3.2.

L.D is a "Load double" instruction. It is followed by ADD.D which is an "FP ALU op". So a latenc BNEy of 1-cycle is required (See Figure 3.2 above) between the two instructions. Similarly, a latency of 2 cycles is required between ADD.D, an "FP ALU op" and S.D, a "Store double". Finally, the DADDUI changes register R1 and this is read by the following BNE instruction. In order to ensure that DADDUI completes execution BEFORE BNE reads registers, a one cycle latency is required. This is explained in the following diagram where the latencies are ensured by inserting *stall*s.

Example Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations, but remember that we are ignoring delayed branches.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                          Clock cycle issued
Loop:  L.D      F0,0(R1)        1
       stall                    2
       ADD.D    F4,F0,F2        3
       stall                    4
       stall                    5
       S.D      F4,0(R1)        6
       DADDUI   R1,R1,#-8       7
       stall                    8
       BNE      R1,R2,Loop      9
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

```
Loop:  L.D      F0,0(R1)
       DADDUI   R1,R1,#-8
       ADD.D    F4,F0,F2
       stall
       stall
       S.D      F4,8(R1)
       BNE      R1,R2,Loop
```

The stalls after ADD.D are for use by the S.D.

In the top-half, we observe that the code is executed in 9 cycles. However, the execution time can be improved by scheduling, i.e. by **reordering** the instructions, without violating sequential execution semantics. This is shown in the **bottom half** of the Figure above. Instead of inserting a stall between L.D and ADD.D, the DADDUI instruction is executed in between. DADDUI does not depend on any other instruction; so it is unaffected by moving up. Also, ADD.D doesn't depend on DADDUI; so it remains unaffected by the upward movement of DADDUI. What have we gained ? Instead of wasting a cycle in between L.D and ADD.D, we have executed an instruction, i.e. DADDUI. So the total exexcution time is reduced ! We have ensured the required 1-cycle latency between L.D and ADD.D; ADD.D is executed 2 cycles after L.D, as required.

The latency between DADDUI and BNE is now (i.e. in the scheduled code) 4 cycles, which is more than the required amount of 1-cycle. So the stall before the BNE is no longer required. Thus, we have saved **TWO stalls !**

However, a small change is required in the code. In the unscheduled code, S.D uses the memory address 0(R1). Suppose, at he beginning of the current iteration of the loop, R1 contains the value *x*. Then, in the unscheduled code (top-half), S.D uses the memory address

0(x). This has to be maintained in the scheduled code also. But in the scheduled code, DADDUI is executed before S.D. As a result, R1 contains (x-8) when S.D begins execution. So, the S.D instruction must be changed to read memory location 8(R1) because 8(x-8) is equivalent to 0(x).[offset(base) format for memory reference; address = base + offset]. So, in the scheduled code, the S.D instruction becomes

S.D F4, 8(R1)

Thus, by scheduling, i.e. reordering the code, we have been able to reduce the execution time of each iteration of the loop from 9 cycles to 7 cycles .