CS 450

Notes on Hazards and Forwarding from Lecture (with minimal changes to notation)

Motivation

There are several kinds of data hazards, broken into two types: those that involve the ALU inputs (ALU hazards and load hazards) and those involving the D input to the DM (load-store hazards). We will explain the situation for each type motivating the problem with several examples.

Note: The notation in this document refers to the registers in the diagram on the last page: Src.A means the A register in the Source pipeline registers, and WB.IR.Rw means the Rw field in the IR register in the Writeback pipeline registers, and so on.

ALU Hazards

Suppose that these instructions are in the indicated stages:

(Mem) add **$1**, $2, $3

(Ex) sub $2, **$1**, $4

(ID) mult $5, $1, $6

As discussed in class, there is a problem is getting the value from $1 in the **add** to the A input to the ALU (we will call this "ALU_A" in what follows) in time for the **sub** instruction. Similarly, in the next cycle, we have:

(WB) add **$1**, $2, $3

(Mem) sub $2, $1, $4

(Ex) mult $5, **$1**, $6

The solution in both cases is the same: we must short-circuit the data path to *forward* the value from the Res.A register (where is stored during the Mem phase) or from the WB.A register (in WB phase) to get it to the ALU_A input. This will happen only when

- The instruction in the Mem/WB phase writes to a register,
- The instruction in the Ex phase reads that same register; and
- That register is NOT $0 (since this is a constant 0 value).

Note that if both of these situations happen at the same time, e.g.,

(WB) add **$1**, $2, $3

(Mem) sub **$1**, $2, $4

(Ex) mult $5, **$1**, $6

we need to read the second value (produced by the sub), not the first. Finally, note that

the situation is essentially unchanged if the instruction reading the register is a sw or lw, e.g.,

(WB) add **$1**, $2, $3

(Mem) sw $2, 8($1)

(Ex) lw $5, 12(**$1**)

or if our hazards involves a jal and jr, e.g.,

(WB) jal 100

(Mem) sw $2, 8($1)

(Ex) jr **$14**

(recall that jal writes to the register $14—here the forwarding path is from WB.NPC to the 3 input to the PC Mux in the stage EX); the important point is that the first instruction produces a value, and the later one uses a value before that value would be written to the RF.

The problem with ALU hazards is that the value being consumed by the ALU is produced too early to read it from the RF, if we wait until the value is written during the WB stage; similar problems develop if the DM is producing a value. Thus, a final example of an ALU hazard is the following:

(WB) lw **$1**, 8($2)

(Mem) add $2, $5, $8

(Ex) sub $3, **$1**, $9

(in which case we forward the value from the WB.Q register to the ALU_A input).

Load-Store Hazards

The DM in fact can be both producer and consumer of the value:

(WB) lw **$1**, 8($2)

(Mem) sw **$1**, 8($3) % Note that $1 is in Rb slot (look at the binary format)

We can solve this, again, by forwarding, this time from the WB.Q register back to the DM_D input, but need only do it when:

- There is a lw instruction in the WB phase;
- There is a sw instruction in the Mem phase reading the register written by the lw; and
- That register is NOT $0 (since this is a constant 0 value).

Load Hazards

In the preceding cases, values produced were ready (somewhere in the circuit) to be consumed by an earlier stage, and we just had to provide the proper forwarding under the right conditions. The main point to see is that the value was produced by the end of one

cycle, and consumed (somewhere) no earlier than the beginning of the next cycle. However, in one case this is not possible:

```
(Mem)   lw $1, 8 ( $2 )
(Ex)    add $5, $1, $2
```

Notice that the value is produced no earlier than at the end of the Mem cycle, and needs to be consumed no later than the beginning of the Ex cycle. This is a distance of two cycles, and hence the add instruction (even with forwarding) can not get its value in time. In this case, we must introduce a nop (or other useful instruction) after the lw so that this reduces to the previous case of an ALU hazard:

```
(WB)   lw $1, 8 ( $2 )
(Mem) sll $0, $0, 0      // This is a nop; note that the binary encoding of this is all 0s
(Ex)    add $5, $1, $2
(ID)    sub $6, $1, $4
```

Such a nop can be in introduced by software (assembler or compiler) or by the hardware (by stalling the pipeline); it is also possible to try to reorder the instructions to do something more useful with this instruction slot, but this is a subject for a later lecture…

Forwarding Circuits for ALU Hazards

We may now show the pseudo-style code for the case of the ALU hazards. We will give details for the case of the ALU_A input only, the B input being very similar. (Note that the ALU_A input also goes to the 3 input in the PC Mux.) The ALU_A input can come from the Mem phase (the Res.NPC or the Res.A registers) or from the WB phase; in this second case, note that we can just take the value from the WB_Out line, the WB Mux having done the work of choosing the appropriate WB register for us! Another feature of our language that will save us some work is that any instruction that writes to the register file will have a Rw field that is non-zero, and any instruction that reads from the register file will have an Ra (respectively, Rb) field that is non-zero. Thus, we do not have to check for opcodes except to check for the value coming from the Res.NPC register in case of the **jal** instruction. This will make our job much easier.

Thus, we have the following pseudo-code specification of the conditions and sources for forwarding to the ALU_A line, where the sequence of the cases indicates which conditions should take precedence:

```
if  (Res.IR.Op == 15 && Src.IR.Ra == 14) // Call this condition JalH
     ALU_A = Res.NPC;
else if ( Res.IR.Rw == Src.IR.Ra && Src.IR.Ra != 0 ) // MemH
      ALU_A = Res.A;
else if ( WB.IR.Rw == Src.IR.Ra && Src.IR.Ra != 0 )  //WBH
      ALU_A = WB_Out
else
      ALU_A = Src.A;    // Default
```

How we do implement these as circuits instead of C-style code? First, we must test for the various conditions, and then we must use these conditions to select the inputs to a mux that chooses among the inputs to the ALU_A line. This mux will select among four possible inputs for the ALU_A wire:

Sel[1]	Sel[0]	Input	Condition for selecting this input
0	0	Src.A	! JalH && ! MemH && ! WBH
0	1	Res.A	! JalH && MemH
1	0	Res.NPC	JalH
1	1	WB_Out	! JalH && ! MemH && WBH

From these we can easily figure out the circuits for Sel[0] and Sel[1].

The case for the B input is very similar (just substitute Rb for Ra), except you must make sure that you put the mux before the ALU.Src mux and before the line that branches off to the Mem phase. Both ALU_A and ALU_B are values that the Execute phase thinks come from the Src.A and Src.B registers, respectively.

Forwarding for Load-Store Hazards

A separate forwarding path is necessary for the load-store hazard. In this case, we must determine the value for the D input to the DM:

```
if ( WB.IR.Op == 12 and Res.IR.Op == 11   // Opcodes in decimal
     && WB.IR.Rw == Res.IR.Rb && Res.IR.Rb != 0 )
   DM_D = WB_Out;     // Or can get it from WB.Q directly
else
   DM _D = Res.B;     // Default
```

We do not have a symmetric case (i.e., both A and B), as there is only one input to the DM! It should be clear from our previous translation of the C-style code into circuits what to do in this case.

Inserting a Stall

In the case of a load hazard, we must insert a nop. It is important to do this as early as possible, which means when the lw is in Src.IR and the next instruction is in ID.IR.

A stall is inserted by inserting all zeros (nop, equal to sll $0, $0, 0) into the Src.IR, disallowing writes to the PC, ID.IR and ID.NPC and letting the rest of the phases proceed (in particular, the lw proceeds to the Mem phase). We may copy the ID.NPC to the Src.NPC with no problem, as the nop will not use it. For example, we might have:

```
lw  $1, 8 ( $2 )          (in Src.IR)
add $5, $1, $2            (in ID.IR)
```

At this point, we should detect a future load hazard using the following condition:

```
(Src.IR.Op == 12 && Src.IR.Rw !=0
 && (Src.IR.Rw == ID.IR.Ra
     || (Src.IR.Rw == ID.IR.Rb && ID.IR.Op != 11)
```

```
        )
    )
```

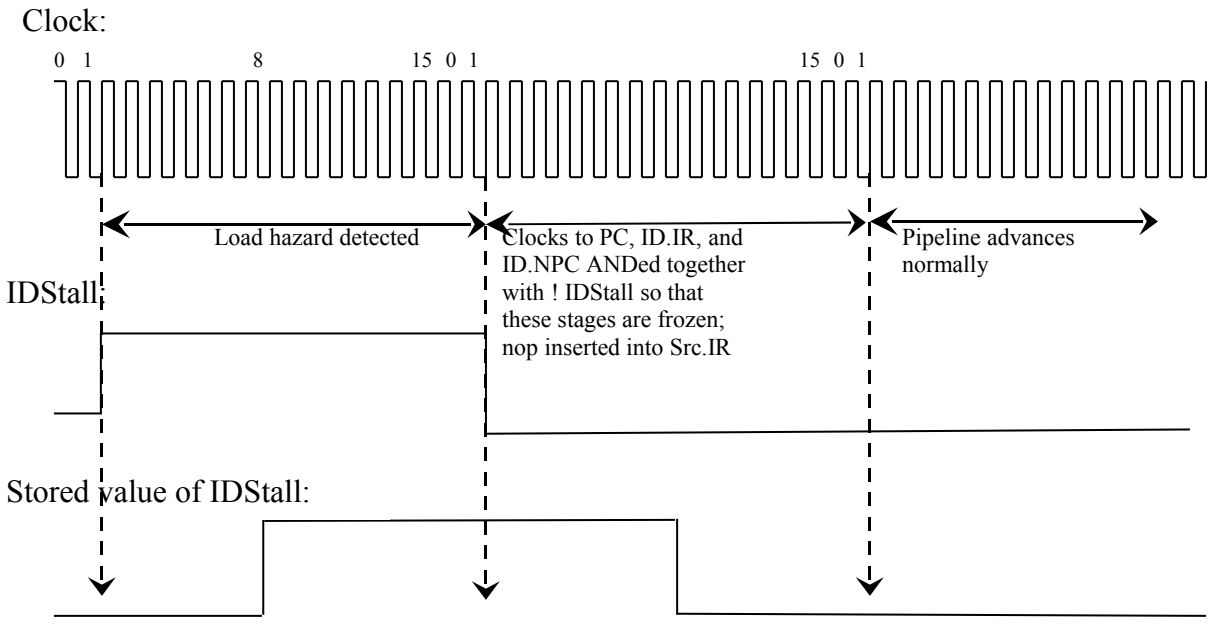
This condition can be tested by a circuit that takes all the relevant register fields as input and produces a single wire, IDStall, as output. Note how we have excluded the case of load-store hazards.

If this condition tests true, we insert a stall, which has the effect of a nop; in the next cycle we should have:

```
lw $1, 8 ( $2 )          (in Res.IR)
nop                      (in Src.IR)
add $5, $1, $2           (in ID.IR)
```

The easiest thing way to stall the IF and ID stages is to turn off the clocks to these stages by ANDing them together with the negation of IDStall (i.e., the rising edge for these clock will only occur when there is no stall required); to insert the nop we can put a mux before the Src.IR register, which chooses between the ID.IR (when IDStall is 0) and nop = 0 (when IDStall is 1). The rest of the registers in the Src pipe will be written with (duplicate) values from the (frozen) ID stage, but these will never be used by the nop instruction.

One subtle detail of this simple technique is that the IDStall value keep its value constant through the rising edge which writes the pipeline registers (clock1). The easiest way to do this is to delay it by storing it in a one-bit register, written sometime during the cycle (e.g., clock8). Here is the timing template for this:



MicroMIPS CPU with Basic Pipeline

Note: Clock and control units are not shown. All pipeline registers are controlled by a single clock signal, as described in the text above.