Computer Architecture Chapter 4 Class 5 Key Topics and Terms

4.7 Data Hazards: Forwarding versus Stalling

Forwarding (3-way) MUXs (Figure 4.54)*:*

2-bit MUX control signals

ForwardA	-	Select first operand input value to ALU
ForwardB	-	Select second operand input value to ALU

Control signal values (***ForwardA*** or ***Forward B***):

00	Use operand value read from register file
01	Use value being written to Register File this cycle Forwarding from WB stage to EX stage (when instruction 2 back generated needed value)
10	Use ALU result value produced last cycle Forwarding from MEM stage to EX stage (when previous instruction generated needed value)
11	Invalid value

Conditions Needed for Forwarding to Occur <u>from</u> a Stage (MEM or WB):

1) The instruction currently in that stage actually writes to the register file

RegWrite control signal is asserted

Example: **`sw $t2, 16($t1)`**
Should not forward a value since it does not generate a value

2) The destination register number (***rd***) matches one (or both) source operands (***rs*** and/or ***rt***) of instruction in EX

3) The destination register number is not 0

Example:

```
addi  $zero, $zero, 5
```

This should be a nop since register 0 is hardwired to 0
In WB this instruction has

a) ***RegWrite*** asserted
b) 00000 applied to ***WriteRegister*** input of Register File
c) 0x0000 0005 applied to ***WriteData*** input of Reg. File

Forwarding would incorrectly pass 5 to ALU instead of 0

4) If both MEM and WB have the same destination register (***rd***) and both actually write (***RegWrite*** asserted), then MEM should win because it holds the instruction later in program sequence

Example:

```
addi  $t2, $zero, 5    ; In WB
addi  $t2, $zero, 7    ; In MEM
add   $t5, $t2, $zero  ; In EX
```

Value 7 should be forwarded to add, not 5

Verilog-Like Equation Description of Forwarding Logic (Forwarding Unit in Figure 4.57)*:*

```
ForwardA = 00                                   Default to not forward Oper. A
ForwardB = 00                                   Default to not forward Oper. B
```

Forward MEM to Operand A:

```
if  (EX/MEM.RegWrite)                           RegWrite asserted
and  (EX/MEM.rd != 0)                           rd is not 0
and  (EX/MEM.rd = ID/EX.rs)                     Operand A register # match
then ForwardA = 10
```

Forward MEM to Operand B:

```
if  (EX/MEM.RegWrite)                           RegWrite asserted
and  (EX/MEM.rd != 0)                           rd is not 0
and  (EX/MEM.rd = ID/EX.rt)                     Operand A register # match
then ForwardB = 10
```

Forward WB to Operand A:

```
if  (MEM/WB.RegWrite)            RegWrite asserted
and  (MEM/WB.rd != 0)            rd is not 0
and  (MEM/WB.rd = ID/EX.rs)      Operand A register # match
and  (ForwardA != 10)            MEM forward to A wins
then ForwardA = 01
```

Forward WB to Operand B:

```
if  (MEM/WB.RegWrite)            RegWrite asserted
and  (MEM/WB.rd != 0)            rd is not 0
and  (MEM/WB.rd = ID/EX.rt)      Operand A register # match
and  (ForwardB != 10)            MEM forward to B wins
then ForwardB = 01
```

Forwarding for Load/Store Memory Copy Operations:

The following load/store pair copies data in memory

```
lw    $t2, 12($t1)     ; load memory value into $t2
sw    $t2, 16($t1)     ; store value to next word
                       ; higher in memory
```

sw needs $t2 value in MEM stage
when sw is in MEM, lw is in WB, so sw read wrong $t2 value in ID
A MUX and control logic can solve this problem in a similar manner

Load/Use Data Hazard Stalls:

The following data hazard cannot be resolved with forwarding:

```
lw    $t2, 12($t1)
add   $t5, $t2, $t0
```

Above forwarding logic would forward the value12+$t1 from MEM rather than the correct $t2 value (which has not yet been read from memory)

However, this data hazard is already resolved by forwarding from WB to EX:

```
lw    $t2, 12($t1)
nop
add   $t5, $t2, $t0
```

A bubble can be inserted into the pipeline by hardware (if it is not already in the machine code) by doing the following:

1) Hold the instructions in the IF and ID stages for an extra clock cycle

2) Allow the instructions in EX and MEM to proceed as usual to the MEM and WB stages respectively

3) Fill the whole created in the EX stage with a nop

A nop is any instruction which:

1) Does not cause a change in the register file values (***RegWrite*** deasserted)

2) Does not cause activity in data memory (***MemWrite*** and ***MemRead*** deasserted)

3) Does not cause a branch to occur (***Branch*** deasserted)

Notes:

Branches in real MIPS processors occur in the ID stage (so that there is only one branch delay slot), so #3 above is not relevant to EX, MEM or WB stages

Books says (in "Elaboration" on page 316) that only RegWrite and MemWrite are relevant, however the authors may not have thought about a nop causing a phantom cache miss (Chapter 6)

Verilog-like Logic for a Load/Use Stall (Hazard Detection Unit in Figure 4.60):

```
if (ID/EX.MemRead)                    ID instruction is a load
and ((ID/EX.rt = IF/ID.rs)
  or (ID/EX.rt = IF/ID.rt))           Either source operand of IF
                       instruction is the same as load's destination
then stall
```

Stall is implemented by:

1) Rereading the same instruction from instruction memory as the previous cycle

PCWrite deasserted such that new PC value is the same as old

2) Holding the existing value in the IF/ID register

IF/IDWrite deasserted

3) Placing 0s in all the control signals of ID/EX register

This deasserts ***RegWrite***, ***MemWrite***, ***MemRead*** and ***Branch***

Section 4.8 will not be covered