4.5 Overview of Pipelining (continued)

Forwarding a Result from the ALU (Resolving a Data Hazard):

1) Sequential Instructions

```
add   $t2, $t1, $t0    ; Produces $t2 in EX stage
add   $t4, $t2, $t3    ; Reads $t2 and $t3 in ID stage
```

When first add is in EX, second add is in ID

Incorrect value of $t2 is read in ID stage by second add instruction

Solution: swap out incorrect $t2 value (***ReadData1*** from register file) with correct $t2 value (***ALUresult*** from ALU)

```
add   $t2, $t1, $t0    ; Produces $t2 in EX stage
add   $t4, $t3, $t2    ; Reads $t2 and $t3 in ID stage
```

Solution: swap out incorrect $t2 value (***ReadData2*** from register file) with correct $t2 value (***ALUresult*** from ALU)

2) One Intervening Instruction

```
add   $t2, $t1, $t0    ; Produces $t2 in EX stage
lw    $t6, 16($t5)     ;
add   $t4, $t2, $t3    ; Reads $t2 and $t3 in ID stage
```

When first add is in MEM, second add is in ID

Incorrect value of $t2 is read in ID stage by second add instruction

Solution: swap out incorrect $t2 value (***ReadData1*** from register file) with correct $t2 value (***ALUresult*** from previous cycle that has not yet been written to register file)

3) Two Intervening Instructions

```
add   $t2, $t1, $t0    ; Produces $t2 in EX stage
lw    $t6, 16($t5)     ;
lw    $t7, 20($t5)     ;
add   $t4, $t2, $t3    ; Reads $t2 and $t3 in ID stage
```

When first add is in WB, second add is in ID

Correct value of $t2 is read in ID stage by second add instruction because registers are written in first half-cycle by WB stage and read in second half-cycle by ID stage

Forwarding a Result from the Data Memory Unit (Resolving a Data Hazard):

1) Sequential Instructions (Load-Use Hazard)

```
lw    $t2, 8($t0)      ; Produces $t2 in MEM stage
add   $t4, $t2, $t3    ; Reads $t2 and $t3 in ID stage
```

When lw is in EX, add is in ID

Incorrect value of $t2 is read in ID stage by add instruction

No forwarding solution possible since lw has not yet completed MEM when add starts EX. Value does not yet exist when it is needed by ALU

Pipeline *bubble*:

Insert a **nop** (no operation) into the pipeline and delay the following instructions

Since the register file ignores writes to register 0,
0x0000 0000 is a **nop** -> **sll $zero, $zero, 0**)

Compiler/assembler generated bubbles:

When generating code insert a **nop** instruction if no useful instruction available for that instruction slot

Hardware load-use hazard detection and bubble generation:

Compare sequential instructions looking for lw opcode in first and matching rd value in first with either rs or rt in second

2) One intervening Instruction

```
lw    $t2, 8($t0)      ; Produces $t2 in MEM stage
addi  $t5, $zero, 7    ;
add   $t4, $t2, $t3    ; Reads $t2 and $t3 in ID stage
```

When lw is in MEM, add is in ID

Incorrect value of $t2 is read in ID stage by add instruction

Solution: swap out incorrect $t2 value (***ReadData1*** from register file) with correct $t2 value (***ReadData*** from Data Memory)

3) Two Intervening Instructions

No problem for same reason as above

Methods for Handling Control Hazards:

```
Target: <inst1>
        <inst2>
beq   $t2, $t1, Target ; Zero flag produced in EX stage
add   $t4, $t5, $t3    ;
sub   $t6, $t5, $t3    ;
```

When beq is in EX, add is in ID and sub is in IF

Zero flag calculated at end of EX stage -- if ***Zero*** asserted, two incorrect instructions are in pipeline (add and sub) when <inst1> and <inst2> should have been

Partial solution: put fast compare circuitry at end of ID stage to test if ***ReadData1*** and ***ReadData2*** are the same (bank of XOR gates) and check if opcode is for beq (or bne)

Now, only instruction in IF stage (add) is wrong when beq reaches ID and branch decision resolved

Solutions to one remaining possibly incorrect instruction in IF:

1) Stall on Branch

a) hardware always converts instruction following a branch to a **`nop`** and uses old PC value again(instead of PC+4)

or

b) compiler/assembler required to place and actual nop in instruction stream following any branch instruction

2) Branch Prediction

Guess that instructions following a branch are likely to be the correct ones and and proceed as if no branch convert incorrect instructions to **`nop`** only if you are wrong (note that incorrect instructions in pipeline can do no damage until MEM or WB stage)

3) Delayed Branching

Change the definition of what is supposed to happen in a stream of machine code instructions to match the reality of pipelining

This is the MIPS solution - Always execute the instruction after the branch whether it is taken or not, then branch or not

Compiler/assembler must either move a useful instruction to the ***delay slot*** or place an actual **`nop`** instruction there

QtSpim: Simulator->Settings, on"MIPS" tab check "Enable Delayed Branches"

4.6 Pipelined Datapath and Control

Division of Single-Cycle Hardware into Stages (Figure 4.33)*:*

IF: PC with input MUX
PC+4 Adder
Instruction Memory Unit

ID: Register File (only used to read two register operands in this stage)
Sign-Extension Unit
Control Signal Generation (decode)

EX: ALU with input MUX on second operand
Branch Target Address Shifter/Adder

note: placing branch target hardware in ID such that there is only one branch delay slot is usual, but it is placed here in this chapter for simplicity

MEM: Data Memory Unit

WB: MUX for ***DataWrite*** input to Register File

The movement of information from later stages to earlier is the source of hazards:

Branch target address from EX to IF is source of control hazards

Register value from WB to ID is source of data hazards

Pipeline Registers (Figure 4.35)*:*

Need to hold information generated by a stage while the stage moves on to next task

Four pipeline registers (incomplete list of contents):

IF/ID: Machine code instruction (32 bits)
PC+4 value (32 bits)

ID/EX: ***ReadData1*** from Register File (32 bits)
ReadData2 from Register File (32 bits)
Sign extended immediate value (32 bits)
PC+4 value relative to original machine code address (32 bits)

EX/MEM: Branch target address (32 bits)
ALUresult (32 bits)
ReadData2 from Register File (32 bits)
Zero flag (1 bit)

MEM/WB: ***ReadData*** from Data Memory Unit (32 bits)
ALUresult (32 bits)

Writing the Correct Register (Figure 4.41):

Figures 4.33 and 4.35 showed the 5-bit ***WriteRegister*** input to the Register File coming directly from the machine code in the IF/ID pipeline register

Problem: the data applied to the ***WriteData*** input to the register file is associated with the instruction currently in the MEM/WB pipeline register (we say that the instruction is in the WB pipeline stage when its associated information is in MEM/WB), but the register number applied to ***WriteRegister*** is associated with the instruction in the ID stage

Solution: pass the 5-bit destination register number along the pipeline stages

Diagram Showing Pipeline State During a Specific Clock Cycle (Figure 4.45):

Instruction Sequence (note that $10 and $r10 are synonyms):

Note that this sequence has no data or control hazards

```
lw    $10, 20($1)      ; when this instruction is in WB
sub   $11, $2, $3      ; this instruction is in MEM and
add   $12, $3, $4      ; this instruction is in EX and
lw    $13, 24($1)      ; this instruction is in ID and
add   $14, $5, $6      ; this instruction is in IF
```

1) info for **`lw $10, 20($1)`** is in MEM/WB pipeline register
This instruction is said to be "in" the WB stage

2) info for **`sub $11, $2, $3`** is in EX/MEM pipeline register
This instruction is said to be "in" the MEM stage

3) info for **`add $12, $3, $4`** is in ID/EX pipeline register
This instruction is said to be "in" the EX stage

4) info for **`lw $13, 24($1)`** is in IF/ID pipeline register
This instruction is said to be “in” the ID stage

5) PC contains address of **`add $14, $5, $6`** instruction
This instruction is said to be “in” the IF stage

In a sense, PC can be thought of as the first pipeline register

Control Signals Need to be Passed Down Pipeline (Figures 4.46, 4.50 and 4.51):

Control signals are generated in the ID stage from the machine code but all used in later stages

Actually, the 4-bit ***ALUoperation*** control signal is finalized in the EX stage (using the 2-bit ***ALUop*** control signal and the ***funct*** field which is the same as the 6 LSb of the ***SignExtImm***), but ignore that in what follows

As the signals are used in a stage they may be dropped from subsequent pipeline registers (Figure 4.50)

Note that even though the 1-bit ***RegWrite*** control signal is shown the ID stage in Figure 4.46, it is driven by the MEM/WB pipeline register so that whether anything gets written to the Register File depends on the instruction currently in the WB stage