Hazards

Why pipelines don’t complete an instruction each cycle

- **structural hazards**
  - instructions in different stages want to use the same resource in the same cycle
  - if not solved, one instruction has to stall
  - example: one memory (early SPARCs)

- **data hazards**
  - an instruction needs the result produced by a previous instruction before it has been written to the register file
    - the second instruction is *data dependent* on the first
  - if not solved, the second instruction has to stall
  - example: \texttt{lw} $8, 12($9)
    \texttt{add} $4, $8, $10
    \texttt{sub} $5, $4, $10

- **control hazards**
  - the instruction after a branch needs to be fetched before the result of the branch condition is determined
    - the second instruction is *control dependent* on the first
  - if not solved, the instruction after the branch has to stall
  - example: \texttt{beq}

all occur because pipelines overlap instruction execution

---

Structural Hazards

**Cause** of the hazard:

- hardware resource conflicts
  - instructions in different stages want to use the same resource in the same cycle
  - rarely, one instruction wants to use the same resource twice in the same cycle

**Solutions:**

- **more hardware**
  - *separate ALUs* for arithmetic, target address calculation, PC incrementing
  - *dual-ported register file*
  - write/read registers on different phases in the same cycle
  - *separate memory (caches)* for instructions & data
  - better performance with a little more hardware --- it’s worth it!

- **just stall**
  - 1 port to memory: service the data access (the older instruction) before the instruction access
  - logic in the MEM stage:
    is this a load? if yes, stall the instruction fetch if the next instruction uses the loaded value
  - less hardware, but much lower performance
Data Hazards

Cause of the hazard:
- producer-consumer conflict over data values
  - an instruction needs data that is produced by a previous instruction
  - the data has not yet been written into a register
- code causes a data dependence
- the implementation causes the data hazard

Solutions:
- compiler code scheduling
  - nops (on a processor without pipeline interlocks, aka R2000)
  - reordering instructions
    - this is an example of where the pipeline structure & latencies are exposed to the architecture
- hardware
  - forwarding
    - output of one stage (result in that stage’s pipeline register) sent to the input of a previous stage
  - pipelined interlock (stall the pipeline)
    - hardware stalls the following instructions

Dependences vs. Hazards

```
sub $2, $1, $3  data dependence
hazard
no hazard

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100 ($2)
```
Code Scheduling

```
add $4, $5, $6
sub $7, $8, $4
add $15, $16, $17
addi $18, $19, 256
inserting nops
   add $4, $5, $6
   nop
   nop
   sub $7, $8, $4
   add $15, $16, $17
   addi $18, $19, 256
reordering instructions
   add $4, $5, $6
   add $15, $16, $17
   addi $18, $19, 256
   sub $7, $8, $4
```

Data dependence
no hazard
Code Scheduling

add $4, $5, $6  
data dependence  
no hazard

add $15, $16, $17

addi $18, $19, $16

sub $7, $8, $4

Forwarding

Forwarding (also called bypassing):

• output of one stage (the result in that stage’s pipeline register) is bused (bypassed) to the input of a previous stage

• why forwarding is possible: results are computed 1 or more stages before they are written to a register
  • at the end of the EX stage for computational instructions
  • at the end of MEM for a load

• if you forward a result to an ALU input as soon as it has been computed, you can eliminate or reduce stalling
Forwarding Example

```
add $7, $12, $15
sub $8, $7, $12
and $9, $13, $7
```

$7 is computed here
$7 is read here
$7 is needed here
$7 is written here
$7 is needed here
$7 is read here

Forwarding Logic

Forwarding unit checks to see if values must be forwarded:
1. between instructions in ID and EX
   - compare the R-type write register number in EX/MEM pipeline register to each read register number in ID/EX
2. between instructions in ID and MEM
   - compare the R-type write register number in MEM/WB to each read register number in ID/EX
   - it's a little more complicated than this

If a match, then forward the appropriate result values to an ALU source
- bus a value from EX/MEM or MEM/WB to an ALU source
Forwarding Hardware

Hardware to implement forwarding:

- destination register number in pipeline registers
  (but need it anyway because we need to know which register to write when storing an ALU or load result)
- source register numbers
  (only rs is extra)
- a comparator for each source-destination register pair
- buses to ship data and register numbers — the BIG cost
- larger ALU MUXes (+ 2 ALU bypass values)

Delayed Loads

Delayed load:

- data hazard caused by a load instruction & an immediate use of the loaded value
- forwarding won’t eliminate the hazard
  data not back from memory until the end of the MEM stage
- solutions
  - if processor has no hardware interlocks
    must schedule independent instructions or nops into the delay slot: R2000
    (the hazard is exposed to the compiler)
  - pipelined interlocks: everyone else
Delayed Loads

lw $2, 20($1)  $2 is available here
and $4, $2, $5  $2 is needed here
or $8, $2, $6  $2 is written here
add $9, $4, $2

Stalling

lw $2, 20($1)  data dependence
and $4, $2, $5  no hazard
or $8, $2, $6  the bubble
add $9, $4, $2
Pipelined Interlocks

How pipelined stalls are detected: hazard detection unit:

- stalls the use after a load
- is the instruction in EX a load?
  - i.e., is the MemRead signal in ID/EX set?
- does the destination register number of the load = either source register number in the next instruction?
  - compare the load write register number in ID/EX to each read register number in IF/ID

⇒ if yes, stall the pipe 1 cycle

Pipelined Interlocks

How stalling is implemented:

- **nullify the instruction in the ID stage**, the one that uses the loaded value
  - change EX, MEM, WB control signals in ID/EX pipeline register to 0
  - the instruction in the ID stage will have no side effects as it passes down the pipeline
- **repeat the instructions in ID & IF stages**
  - disable writing the PC --- the same instruction will be fetched again
  - disable writing the IF/ID pipeline register --- the load use instruction will be decoded & its registers read again
Stall Hardware

Hardware to implement stalling:

- rt register number in ID/EX pipeline register
  (but need it anyway because we need to know what register
to write when storing load data)
- both source register numbers in IF/ID pipeline register
  (already there)
- a comparator for each source-destination register pair
- buses to ship register numbers
- write enable/disable for PC
- write enable/disable for the IF/ID pipeline register
- a MUX to the ID/EX pipeline register (+ 0s)

Code Scheduling vs. Pipelined Interlocks

Pipeline interlocks preferred

- trivial amount of hardware
  stall logic needed for cache misses anyway
- smaller code size
  no nops when no independent instructions
- higher performance processors can have shorter clock cycles &
  more stages
  therefore there could have more than one delay slot

MIPS now uses pipeline interlocks!