Instruction-Level Parallelism (ILP)

Fine-grained parallelism

Obtained by:

- instruction overlap in a pipeline
- executing instructions in parallel (later, with multiple instruction issue)

ILP hindered by:

- **data dependence**: arises from the flow of values through programs
- **name dependence**: instructions use the same register but no flow of data between them
- **control** dependence: arises from the flow of control
Pipelining

Implementation technique (but it is visible in the architecture)
  • overlaps execution of different instructions
  • execute all steps in the execution cycle simultaneously, but on different instructions
Exploits ILP by executing several instructions “in parallel”
Goal is to increase instruction throughput

\[
\text{optimal speedup} = \frac{T_{\text{without pipe}}}{T_{\text{with pipe}}} = \frac{i \times n}{i + n - 1} \approx \# \text{ of pipe stages}
\]
Pipelining

![Diagram](image)

- **fetch**
- **d&i**
- **ex**
- **mem**
- **wb**

**fill the pipeline**

**new instruction completes each cycle**
Pipelining

Not that simple!

- pipeline **hazards** (structural, data, control)
  - place a soft “limit” on the number of stages
- increase instruction latency (a little)
  - write & read pipeline registers for data that is computed in a stage
    - information produced in a stage travels down the pipeline with the instruction
  - time for clock & control lines to reach all stages
- all stages are the same length which is determined by the longest stage
  - stage length determines clock cycle time

IBM Stretch (1961): the first general-purpose pipelined computer
Hazards

Structural hazards
Data hazards
Control hazards

What happens on a hazard

• instruction that caused the hazard & previous instructions complete
• all subsequent instructions stall until the hazard is removed (in-order execution)
• only instructions that depend on that instruction stall (out-of-order execution)
• hazard removed
• instructions continue execution
Structural Hazards

Cause: instructions in different stages want to use the same resource in the same cycle
  e.g., 4 FP instructions ready to execute & only 2 FP units

Solutions:

• more hardware (eliminate the hazard)
• stall (tolerate the hazard)
  • less hardware, lower performance
  • only for big hardware components
STRUCTURAL HAZARDS: EXAMPLES

- READ REGISTERS
- FETCH DATA & INSTRUCTIONS
- INCREMENT PC
- ARITH. OPERATION
- COMPARISON TARGET CALCULATION
Data Hazards

Cause:

• an instruction early in the pipeline needs the result produced by an instruction farther down the pipeline before it is written to a register
• would not have occurred if the implementation was not pipelined

Types

RAW (data: flow), WAR (name: antidependence), WAW (name: output)

HW solutions

• forwarding hardware (eliminate the hazard)
• stall via pipelined interlocks

Compiler solution

• code scheduling (for loads)
**Dependences vs. Hazards**

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100 ($2)

*data dependence  
data hazard  
no hazard*
**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 useful
  - 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
  - results are used 1 or more stages after registers are read
- if you forward a result to an ALU input as soon as it has been computed, you can eliminate the hazard or reduce stalling
Forwarding Example
Forwarding Implementation

**Forwarding unit** checks whether forwarded values should be used:

- between instructions in ID and EX
  - compare the R-type **destination register number in EX/MEM**
    pipeline register to each **source register number in ID/EX**
  - between instructions in ID and MEM
  - compare the R-type **destination register number in MEM/WB**
    to each **source register number in ID/EX**

If a match, set MUX to choose bussed values from **EX/MEM or MEM/WB**
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
  (probably only one, e.g., \texttt{rs} on MIPS R2/3000) is extra)

• a comparator for each source-destination register pair

• buses to ship data and register numbers – the \textbf{BIG} cost

• larger ALU MUXes for 2 bypass values
Loads

Loads
• data hazard caused by a load instruction & an immediate use of the loaded value
• forwarding won’t eliminate the hazard why? data not back from memory until the end of the MEM stage
• 2 solutions used together
  • **stall** via pipelined interlocks
  • **schedule** independent instructions into the **load delay slot** (a pipeline hazard that is exposed to the compiler) so that there will be no stall
Loads

`lw $2, 20($1)`

$2$ is available here

$2$ is needed here

and $4$, $2$, $5$

$2$ is written here

or $8$, $2$, $6$

Data dependence hazard

no hazard

add $9$, $4$, $2$
Implementing Pipelined Interlocks

Detecting a stall situation

Hazard detection unit stalls the use after a load

- is the instruction in EX a load?
- 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 both yes, stall the pipe 1 cycle
Implementing 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
- **restart the instructions that were stalled 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
Loads

\textbf{lw} \$2, 20(\$1)

\textbf{hazard detection}

\textbf{data dependence}
\textbf{no hazard}

\textbf{and} \$4, \$2, \$5

\textbf{decode again}

\textbf{fetch again}

\textbf{the bubble}

\textbf{add} \$9, \$4, \$2

Winter 2006  CSE 548 - Basics of Pipelining
Implementing Pipelined Interlocks

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)

Trivial amount of hardware & needed for cache misses anyway
Control Hazards

**Cause:** condition & target determined after the next fetch has already been done

**Early HW solutions**
- stall
- assume an outcome & flush pipeline if wrong
- move branch resolution hardware forward in the pipeline

**Compiler solutions**
- code scheduling
- static branch prediction

**Today’s HW solutions**
- dynamic branch prediction