Instruction-Level Parallelism (ILP)

Fine-grained parallelism
Obtained by:
- instruction overlap in a pipeline
- executing instructions in parallel (later, with multiple instruction issue)
In contrast to:
- loop-level parallelism (medium-grained)
- thread-level or task-level or process-level parallelism (coarse-grained)

Can be exploited when instructions are independent of one another
- two instructions are independent if their operands are different
  - an example of independent instructions
    \[\text{ld R1, 0(R2)}\]
    \[\text{or R7, R3, R8}\]
Dependences

data dependence: arises from the flow of values through programs
  • consumer instruction gets a value from a producer instruction
  • determines the order in which instructions can be executed

| ld R1, 32(R3) |
| add R3, R1, R8 |

name dependence: instructions use the same register but no flow of data between them
  • anti-dependence
  • output dependence

| ld R1, 32(R3) |
| add R3, R1, R8 |
| ld R1, 16(R3) |

control dependence
  • arises from the flow of control
  • instructions after a branch depend on the value of the branch’s condition variable

beqz R2, target
  ld r1, 0(r3)
target: add r1, ...
**Instruction-Level Parallelism (ILP)**

ILP is important for executing instructions in parallel and hiding latencies
- each thread (program) has very little ILP
- dependences inhibit ILP
- tons of techniques to increase it

**Pipelining**

Implementation technique (but it is considered part of 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

\[
\text{optimal speedup} = \frac{T_{\text{without pipe}}}{T_{\text{with pipe}}} = \frac{i \times n}{i + n - 1} = \# \text{ of pipe stages}
\]
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
  - all stages are the same length which is determined by the longest stage
    - stage length determines clock cycle time
    - time for clock & control lines to reach all stages

IBM Stretch (1961): the first general-purpose pipelined computer
**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 (so still execute correct programs)
  - less hardware, lower cost
  - only for big hardware components
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), WAR (name: anti-dependence), WAW (name: output)
  Note: dependences are a function of the code; whether they result in a hazard is a function of the pipeline design.

HW solutions
- forwarding hardware (eliminate the hazard)
- stall via pipelined interlocks if can't forward

Compiler solution
- code scheduling (for loads)

Dependences vs. Hazards
Forwarding Example

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
  - 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 Implementation

**Forwarding unit** checks to see if values can be forwarded:

- 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, 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
  (probably only one, e.g., rs on MIPS R2/3000) is extra
- a comparator for each source-destination register pair
- **buses to ship data** — the **BIG** cost
- buses to ship register numbers
- 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?
- 2 solutions used together
  - stall via pipelined interlocks
  - compiler schedules independent instructions into the **load delay slot**
  (This is a reason that pipelines are part of the architecture: the pipeline structure is exposed to the compiler.)
**Loads**

```
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
```

- $s2 is available here
- $s2 is needed here
- $s2 is written here

- data dependence hazard
- no hazard

**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 yes, stall the pipe 1 cycle
Implementing Pipelined Interlocks

How stalling is implemented:

• **nullify the instruction in the ID stage**, the one that consumes 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 IF/ID pipeline register – the load consumer instruction will be decoded & its registers read again
  • disable writing the PC – the same instruction will be fetched again

Load hazard detection:

```
lw $2, 20($1)
and $4, $2, $5
or $3, $2, $5
the bubble
```

Data dependence:

```
add $3, $4, $2
```
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 next fetch

Early HW solutions
- stall
- fetch the next instruction & flush the pipeline if wrong
- move branch resolution hardware forward in the pipeline (detect a problem earlier)

Compiler solutions
- code scheduling
- static branch prediction

Today's HW solutions
- dynamic branch prediction