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)
• process-level or task-level or thread-level parallelism (coarse-grained)
Instruction-Level Parallelism (ILP)

Can be exploited when instruction operands are independent of each other, for example,

- two instructions are independent if their operands are different
- an example of independent instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld R1, 0(R2)</code></td>
<td>Load R1 from memory at address 0(R2)</td>
</tr>
<tr>
<td><code>or R7, R3, R8</code></td>
<td>Logical OR of registers R3 and R8, store result in R7</td>
</tr>
</tbody>
</table>
**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
- **antidependence**
- **output dependence**

```
ld R1, 32(R3)
add R3, R1, R8
ld R1, 16(R3)
```
Dependences

**control** dependence

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

<table>
<thead>
<tr>
<th></th>
<th>beqz R2, target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw r1, 0(r3)</td>
</tr>
<tr>
<td>target:</td>
<td>add r1, ...</td>
</tr>
</tbody>
</table>

Dependences inhibit ILP
Instruction-Level Parallelism (ILP)

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

Implementation technique (but it is visible to 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

```
fetch d&u  ex  mem  wb
```

```
fetch d&u  ex  mem  wb
```

```
 t  t+1  t+2  t+3  t+4  t+5  t+6  t+7  t+8  t+9  ...
```

```plaintext
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
  - 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 the instruction that caused the hazard stall (out-of-order 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 (so still execute correct programs)
  - less hardware, lower cost
  - only for big hardware components
STRUCTURAL HAZARDS: EXAMPLES

FETCH DATA & INSTRUCTIONS

READ REGISTERS

INCREMENT PC
ARITH OPERATIONS

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

\[ \text{sub } 2, 1, 3 \]
\[ \text{and } 12, 2, 5 \]
\[ \text{or } 13, 6, 2 \]
\[ \text{add } 14, 2, 2 \]
\[ \text{sw } 15, 100 (2) \]
**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
  - 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

add $7, $12, $15

$7 is computed here

$7 is read here

sub $8, $7, $12

$7 is needed here

$7 is needed here

and $9, $13, $7

$7 is needed here

$7 is read here
Forwarding Implementation

Forwarding unit checks to see if values must 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 and register numbers – the **BIG** cost
- larger ALU MUXes for 2 bypass values
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**

\[ \text{lw} \, \$2, \, 20(\$1) \]

\[ \text{and} \, \$4, \, \$2, \, \$5 \]

\[ \text{or} \, \$8, \, \$2, \, \$6 \]

\[ \text{add} \, \$9, \, \$4, \, \$2 \]
Implementing Pipelined Interlocks

Detecting a stall situation

Hazard detection unit stalls the use after 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
- is the instruction in EX a load?

⇒ if 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
- **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
Loads

`lw $2, 20($1)`

hazard detection

data dependence
no hazard

and $4, $2, $5

decode again

fetch again

or $8, $2, $6

the bubble

add $9, $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
- 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