Instruction-Level Parallel Processing

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE
Remind: Performance w/ Pipelining

- We finish one instruction per cycle
  - After the initial warm-up period

- Instruction throughput
  - Latch at end of each stage adds latency
  - Longest stage determines clock cycle time
  - Example:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Cycle Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>1.0</td>
</tr>
<tr>
<td>ID</td>
<td>0.6</td>
</tr>
<tr>
<td>EX</td>
<td>0.9</td>
</tr>
<tr>
<td>MEM</td>
<td>1.2</td>
</tr>
<tr>
<td>WB</td>
<td>0.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Cycle Time</th>
<th>Inst/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single cycle</td>
<td>4.1 ns (sum)</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline</td>
<td>1.2 ns (max)</td>
<td>1</td>
</tr>
</tbody>
</table>

Speedup due to pipelining: 3.42
Note: ideally it would be 5
Then, How about using Deeper pipelines?

➢ Called **Super-pipelining**

- **Goal:** Speedup ~ number of stages
- **Idea:** let’s have a lot of stages
  - Vaguely defined by deep pipelining

- **What does this mean by?**
  - While splitting a stage into multiple sub-stages, the machine can issue a new instruction every minor cycle
  - If one splits the stage into “m” sub-stages, the clock cycle period “T” should be reduced by “T/m”
### Ideal Case of Super-pipelining

The super-pipeline produces the result every $\frac{T}{m}$ clock cycle.

<table>
<thead>
<tr>
<th>Instr. $i$</th>
<th>Fetch</th>
<th>Decode</th>
<th>Inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. $i+1$</td>
<td>Fetch</td>
<td>Decode</td>
<td>Inst</td>
</tr>
<tr>
<td>Instr. $i+2$</td>
<td>Fetch</td>
<td>Decode</td>
<td>Inst</td>
</tr>
<tr>
<td>Instr. $i+3$</td>
<td>Fetch</td>
<td>Decode</td>
<td>Inst</td>
</tr>
</tbody>
</table>

- **Pipelining**

  - Cycle 1: $F_1$, $F_2$, $D_1$, $D_2$, $E_1$, $E_2$
  - Cycle 2: $F_1$, $F_2$, $D_1$, $D_2$, $E_1$, $E_2$
  - Cycle 3: $F_1$, $F_2$, $D_1$, $D_2$, $E_1$, $E_2$
  - Cycle 4: $F_1$, $F_2$, $D_1$, $D_2$, $E_1$, $E_2$
  - Cycle 5: $F_1$, $F_2$, $D_1$, $D_2$, $E_1$, $E_2$

- **Super-Pipelining**

  - Cycle 1: $F_1$ $F_2$ $D_1$ $D_2$ $E_1$ $E_2$
  - Cycle 2: $F_1$ $F_2$ $D_1$ $D_2$ $E_1$ $E_2$
  - Cycle 3: $F_1$ $F_2$ $D_1$ $D_2$ $E_1$ $E_2$
  - Cycle 4: $F_1$ $F_2$ $D_1$ $D_2$ $E_1$ $E_2$
  - Cycle 5: $F_1$ $F_2$ $D_1$ $D_2$ $E_1$ $E_2$

**Gain**

$\frac{T}{m}$
**Problems of Super-pipelining**

1. Require a really high clock frequency!
2. Instruction dependences (similar to pipelining, but much **critical**)

**Control dependences**

```
beq 1 2 3 4 5 6 32
Instr1 1 2 3 4 5 6 7 ...
Instr2 1 2 3 4 5 6 ...
Instr3 1 2 3 4 5 ...
Flush 31 Instr.!
```

**Data dependences**

```
LW R1,0(R2) 1 2 3 4 5 ...
Instr31 1
```

**Comparison**

```
ADD R2,R3,R1
```

Flush 31 Instr.!

Value from memory arrives

```
Value of R1 needed
```

Stall 15 cycles!
There is an Optimal Stages

- For a given architecture and associated instruction set
- Also need to consider workload characteristics

**Diminishing Returns:**
Increasing the number of stages over this limit reduces the overall performance

Source: “Runtime Aware Architectures”, Mateo Valero, HiPEAC CSW 2014
Another Instruction-Level Parallelism (ILP)

Internal of operation principle of ILP-processors

- Pipelined operation
  - EX₁
  - EX₂
  - EX₃

- Parallel operation
  - EX₁
  - EX₂
  - EX₃

Instr → Orthogonal Design

Pipelined processors

Superscalar / VLIW
Super-scalar

**Dynamic** multiple-issue processors (decision making at run time by the **hardware**)

- IBM Power 3
- Pentium 4
- MIPS R10K
- HP PA 8500
Superscalar architectures allow several instructions to be issued and completed per clock cycle.

A superscalar architecture consists of a number of pipelines that are working in parallel (N-way Superscalar)

Can issue up to N instructions per cycle.
Super-pipelining vs. Super-scalar

Note that super-pipelining is orthogonal with super-scalar
Superscalarity is Important!

- Ideal case of N-way Super-scalar
  - All instructions were independent
  - Speedup is “N” (Superscalarity)

- What if all instructions are dependent?
  - No speed up, super-scalar brings nothing
  - (Just similar to pipelining)
**Pipelining vs. Super-scaller**

- **Superscalarity example: sum of array elements**
  (assume that load-to-use dependence only take a cycle for the sake of brevity)

**Assembly Code**

```
sum += a[i--]

loop: ld $r2, 10($r1)
      add $r3, $r3, $r2
      sub $r1, $r1, 1
      bne $r1, $r0, loop
```

**Pipelining architecture**

```
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne</td>
</tr>
</tbody>
</table>
```

**Superscaler architecture**

```
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne</td>
</tr>
</tbody>
</table>
```

We should find out independent instructions!
From Sequential Instructions to Parallel Execution

Q1. How’s the ability of a superscalar processor to execute instructions in parallel determined?

A. The number and nature of parallel pipelines

B. The mechanism that the processor uses finds independent instructions (which can be executed in parallel)

The policies used for the instruction execution are characterized by the following factors

- The order in which instructions are issued for execution (In-Order/Out-Order)
- The order in which instructions are completed (In-Order/Out-Order)

Q. BTW, what’s the meaning of controlling instruction completion?

A. Determine when the instruction write results into registers and memory locations
Execute in Parallel
But Make Sure Sequential Order

• Execute and complete instructions in their sequential order (but a little chance to execute in parallel)
• To improve parallelism, superscalar has to look ahead and try to find independent instructions to execute in parallel

Instructions will be executed in an order different from the strictly sequential one, with the restriction that the result must be correct

⇒ Ideas for the execution policies; we will learn more details with specific techniques for this, partially today and next lectures
Example Scenario of Parallel Execution Policy

- We consider the following instruction sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1: ADDF R12,R13,R14</td>
<td>R12 ← R13 + R14 (float. pnt.)</td>
</tr>
<tr>
<td>I2: ADD R1,R8,R9</td>
<td>R1 ← R8 + R9</td>
</tr>
<tr>
<td>I3: MUL R4,R2,R3</td>
<td>R4 ← R2 * R3</td>
</tr>
<tr>
<td>I4: MUL R5,R6,R7</td>
<td>R5 ← R6 * R7</td>
</tr>
<tr>
<td>I5: ADD R10,R5,R7</td>
<td>R10 ← R5 + R7</td>
</tr>
<tr>
<td>I6: ADD R11,R2,R3</td>
<td>R11 ← R2 + R3</td>
</tr>
</tbody>
</table>

- Assumption:
  - I1 requires two cycles to execute
  - I3 and I4 are in conflict for the same functional unit
  - I5 depends on the value produced by I4 (we have a true data dependency between I4 and I5);
  - I2, I5 and I6 are in conflict for the same functional unit
Example Scenario of Parallel Execution Policy

- **Parallel execution policy 1**
  - Issue: In-Order & Completion: In-Order
  - Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order

- **Parallel execution policy 2**
  - Issue: Out-of-Order & Completion: Out-of-Order
  - Out-of-order issue takes the set of decoded instructions the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct
Example Scenario of Parallel Execution Policy

- **Parallel execution policy 1**
  - Issue: In-Order & Completion: In-Order
  - Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order

- **Parallel execution policy 2**
  - Issue: Out-of-Order & Completion: Out-of-Order
  - Out-of-order issue takes the set of decoded instructions the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct
### Parallel Execution Policy

**In-Order Issue with In-Order Completion**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decode/Issue</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1 I2</td>
<td></td>
<td>I1</td>
</tr>
<tr>
<td>2</td>
<td>I3 I4</td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>3</td>
<td>I5 I6</td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>I1 I2</td>
<td>I2</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>I1 I2</td>
<td>I2</td>
</tr>
</tbody>
</table>

#### Considerations

- Two cycle to execute
- I2, I5, I6 same functional unit
- I3, I4 same functional unit
- True data dependency

An instruction cannot be issued before the previous one has been issued.

An instruction completes only after the previous one has completed.
Parallelism Depends on the Program
In-Order Issue with In-Order Completion

- The processor detects and handles (by stalling) true data dependencies and resource conflicts.
- As instructions are issued and completed in their strict order, the resulting parallelism is very much dependent on the way where the program is written/compiled.

I1: ADDF $R12, $R13, $R14
I2: ADD   $R1, $R8, $R9
I3: MUL   $R4, $R2, $R3
I4: MUL   $R5, $R6, $R7
I5: ADD   $R10, $R5, $R7
I6: ADD   $R11, $R2, $R3

Switch I3, I6 position

Parallel execution
### Rewrite Code for Better Parallelism

**In-Order Issue with In-Order Completion**

**Considerations**
- Two cycle to execute
- I2, I5, I6 same functional unit
- I3, I4 same functional unit
- True data dependency

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decode/Issue</th>
<th>Execute</th>
<th>Writeback/Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1, I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I6, I4</td>
<td>I1, I2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>I5, I3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>I1</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>I2</td>
<td>I1, I2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>I5</td>
<td>I5, I3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I3</td>
<td>I6, I4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>I4</td>
<td>I6, I4</td>
<td>I1, I2</td>
</tr>
</tbody>
</table>

**Instructions**
- I1: ADDF $R12, $R13, $R14
- I2: ADD $R1, $R8, $R9
- I6: ADD $R11, $R2, $R3
- I4: MUL $R5, $R6, $R7
- I5: ADD $R10, $R5, $R7
- I3: MUL $R4, $R2, $R3
Example Scenario of Parallel Execution Policy

- **Parallel execution policy 1**
  - Issue: In-Order & Completion: In-Order
  - Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order

- **Parallel execution policy 2**
  - Issue: Out-of-Order & Completion: Out-of-Order
  - Out-of-order issue takes the set of decoded instructions the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct
### Parallel Execution Policy

**Out-of-Order Issue with Out-of-Order Completion**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode/Issue</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I1</td>
<td>I2</td>
</tr>
<tr>
<td>Execute</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
</tr>
<tr>
<td>Writeback/Complete</td>
<td>I1</td>
<td>I5</td>
<td>I6</td>
<td>I1</td>
<td>I5</td>
<td>I4</td>
<td>I1</td>
<td>I3</td>
</tr>
</tbody>
</table>

### Out-of-Order Issue

- Out-of-Order Issue does not need to wait until I1 is executed.
- Need to remove true dependency issue!
- Similar to Issue, it does not need to wait until I1 is completed.

### Considerations

- Two cycle to execute
- I2, I5, I6 same functional unit
- I3, I4 same functional unit
- True data dependency

```
I1: ADDF $R12, $R13, $R14
I2: ADD $R1, $R8, $R9
I3: MUL $R4, $R2, $R3
I4: MUL $R5, $R6, $R7
I5: ADD $R10, $R5, $R7
I6: ADD $R11, $R2, $R3
```
Challenges and Considerations of Superscaler

- No free lunch! There are **dependencies**!
- Must check dependencies for all instructions, which are
  - Simultaneously decoded
  - In-progress in the pipeline (e.g., previously issued)

**Dependences (constraints)**

- **Data dependences**
- **Control dependences**
- **Resource dependences**

**Naïve solution**

**(Stalls** at decode by comparing)**

1) Rs vs. outstanding targets
2) Rt vs. outstanding targets + sources
3) No hardware resources

**Stall everything saves these as well, but do we have something better dependencies (this would introduce longer latency)?**
A Set of Better Solutions for Superscaler

➢ To address the dependency issues w/ in-flight instructions (will be covered in next lectures)

▪ Control dependency
  ➔ Loop unrolling of static scheduling (Lecture 7)
  ➔ Branch prediction

▪ Dependencies
  ➔ Scoreboard
  ➔ Static scheduling
  ➔ Hardware register renaming of Tomasulo

▪ Regulation of instruction ordering
  ➔ Reservation station
  ➔ Reorder buffer/ROB

▪ Exception handling
Superscalar Architecture

- Putting all together; More detailed Superscalar Architecture

Diagram showing the components of a superscalar architecture:
- Instr. buffer
- Fetch & Addr. Calc. & Branch pred.
- Decode & Rename & Dispatch
- Instr. Window (queues, reservation stations, etc.)
- Instruction issuing
- Floating Point unit
- Integer unit
- Integer unit
- Register Files
- Commit
- Memory
Ancient Superscalar Architecture

- **PowerPC 6XX**
  - **Six** independent execution units:
    - Branch execution unit
    - Load/store unit
    - Three integer units
    - Floating point unit
  - **Out-of-order** issue

- **Pentium I/II**
  - **P-I**: Three independent units
  - **P-II**: **out-of-order**, five instructions can be issued in a cycle
VLIW

Static multiple-issue processors (decision making at compile time by the compiler)

Intel Itanium series for the IA-64 ISA
- EPIC (Explicit Parallel Instruction Computer)
VLIW; Very Long Instruction Word

**Key Idea:** Replace a traditional sequential ISA with a new ISA that enables the compiler to encode instruction-level parallelism (ILP) directly in the hardware/software interface.

- Sub-instructions within a long instruction *must* be independent.
- Multiple “sub-instructions” packed into one long instruction.
- Each “slot” in a VLIW instruction for a specific functional unit.
VLIW Hardware (TinyRV1 VLIW Processor)

- TinyRV1 VLIW ISA
  - 4-cycle Y-pipe, 1-cycle X-pipe, 2-cycle L-pipe, 2-cycle S-pipe
  - No hazard checking, assume ISA enables setting bypass muxing
VLIW Software (Compilation Techniques)

- **Key Questions:**
  - How do we find independent instructions to fetch/execute?
  - How to enable more compiler optimizations?

- **Key Ideas:**
  - Get rid of control flow
    - Predicated execution, loop unrolling
  - Optimize frequently executed code-paths
    - Trace scheduling
  - Others: Software pipelining
Compile Technique 1: Loop Unrolling

**Key idea:** Unroll loop to perform M iterations at once

**Limitations:** Code growth, does not handle inter-iteration

For (i=0; i<N; i++)

```
```

```
```

VLIW Compiler

Optimized C code

```
For (i=0; i<N; i+=4)
{
}
```

Expose more Instruction-level Parallelism!
Key idea: Eliminate hard-to-predict branches by converting control dependence to data dependence

Original C code

```c
if (cond) {
    b = 0;
} else {
    b = 1;
}
```

(normal branch code)

```c
p1 = (cond)
branch p1, TARGET
mov b, 1
jmp JOIN
```

(predicated code)

```c
p1 = (cond)
(!p1) mov b,1
(p1) mov b,0
```
Compile Technique 2: Predicated Execution

- **Key idea:** Eliminate hard-to-predict branches by converting control dependence to data dependence
- **Limitations:** Reduces perf. if misprediction cost < benefit

### Predicated Execution

```
Predicated Execution
Fetch  Decode  Rename  Schedule  RegisterRead  Execute
```

```
A
B
C
D
E
F
```

### Branch Prediction

```
Branch Prediction
Fetch  Decode  Rename  Schedule  RegisterRead  Execute
```

```
F  E  D  C  B  A
```

Pipeline flush!!
Advantages & Disadvantages of VLIW

**Advantages**

- Simpler hardware (potentially less power hungry)
- Potentially more scalable
  - Allow more instr’s per VLIW bundle and add more FUs

**Disadvantages**

- Programmer/compiler complexity and longer compilation times
  - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
- Object (binary) code incompatibility
- Needs lots of program memory bandwidth
- Code bloat
  - Noops are a waste of program memory space
  - Loop unrolling to expose more ILP uses more program memory space
## CISC vs RISC vs SuperScalar vs VLIW

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instr size</strong></td>
<td>variable size</td>
<td>fixed size</td>
<td>fixed size</td>
<td>fixed size (but large)</td>
</tr>
<tr>
<td><strong>Instr format</strong></td>
<td>variable format</td>
<td>fixed format</td>
<td>fixed format</td>
<td>fixed format</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>few, some special</td>
<td>many GP</td>
<td>GP and rename (RUU)</td>
<td>many, many GP</td>
</tr>
<tr>
<td><strong>Memory reference</strong></td>
<td>embedded in many instr’s</td>
<td>load/store</td>
<td>load/store</td>
<td>load/store</td>
</tr>
<tr>
<td><strong>Key Issues</strong></td>
<td>decode complexity</td>
<td>data forwarding, hazards</td>
<td>hardware dependency resolution</td>
<td>(compiler) code scheduling</td>
</tr>
<tr>
<td><strong>Instruction flow</strong></td>
<td>IF ID EX M WB</td>
<td>IF ID EX M WB</td>
<td>IF ID EX M WB</td>
<td>IF ID EX M WB</td>
</tr>
</tbody>
</table>
Instruction-Level Parallel Processing

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory
A MIPS VLIW (2-issue) Datapath

- No hazard hardware (so no load use allowed)
Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Done by code re-ordering (trace scheduling)
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW’s primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts