18-547 Lecture 18

Instruction Level Parallelism

Instructor: Prof. Falsafi

Electrical & Computer Engineering
Carnegie Mellon University

Slides developed by Profs. Hill, Wood, Sohi, and Smith of
University of Wisconsin — Madison

Basic Structure of DLX-T FP Unit
Tomasulo’s Algorithm

3 major steps

1. Dispatch
   - get instruction from queue
   - ALU op: check for available reservation station
   - load: check for available load buffer
   - if available: issue and copy ready regs to RS
   - if not: stall due to structural hazard

2. Issue
   - if not all operands are available, begin execution
   - if not, monitor CDB for operand

3. Complete
   - if CDB available, write result on CDB
   - if not, stall

Reservation stations
   - handle distributed hazard detection and instruction control

Everything receiving data gets the tag of the data
   - 4-bit tag specifies reservation station or load buffer
   - specifies which FU will produce result

Register specifier is used to assign tags
   - then it is discarded
   - register specifiers are only used in dispatch
Tomasulo vs. Scoreboard

Similar to scoreboard but

- no checking for WAW or WAR
- CDB broadcasts results to functional units, not just registers

Implementation

Reservation stations

- \( \text{op} \) -- opcode
- \( Q_j, Q_k \) -- tag fields (sources)
- \( V_j, V_k \) -- operand values (sources)
- \( \text{busy} \) -- currently in use

Register file and store buffer

- \( Q_i \) -- tag field
- \( \text{busy} \) -- currently in use
Tomasulo Code Example

\[
\begin{align*}
\text{If} & \quad f6, 34(r2) \\
\text{If} & \quad f2, 45(r3) \\
\text{mul} & \quad f0, f2, f4 \\
\text{sub} & \quad f8, f6, f2 \\
\text{div} & \quad f10, f0, f6 \\
\text{add} & \quad f6, f8, f2
\end{align*}
\]

Instruction Status (Illustration Only)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>If</td>
<td>f6, 34(r2)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>If</td>
<td>f2, 45(r3)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mul</td>
<td>f0, f2, f4</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>f8, f6, f2</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>f10, f0, f6</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>f6, f8, f2</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
### Tomasulo Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>V</th>
<th>V_k</th>
<th>Q_j</th>
<th>Q_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add 1</td>
<td>Yes</td>
<td>SUB</td>
<td>Mem[+Reg(r3)]</td>
<td>Load 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 2</td>
<td>Yes</td>
<td>ADD</td>
<td>Add 1</td>
<td>Load 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult 1</td>
<td>Yes</td>
<td>MULT</td>
<td>Regs[f4]</td>
<td>Load 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult 2</td>
<td>Yes</td>
<td>DIV</td>
<td>Mem[+Reg(r2)]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Q**<sub>1</sub> Mult1 Load2
- **Q**<sub>4</sub> Mult2
- **Q**<sub>6</sub> Load2
- **Q**<sub>8</sub> Add2 Add1 Mult2
- **Q**<sub>10</sub> ...
- **Q**<sub>30</sub>

#### Register Status

- Busy: Yes, Yes, No
- Load buffers: Add1, Add2, Add3
- Store buffers: Mult1, Mult2

#### Field

<table>
<thead>
<tr>
<th>Address</th>
<th>Load 1</th>
<th>Load 2</th>
<th>Load 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Redo Tomasulo

```plaintext
loop:

-if f0, 0(r1)
-multf f4, f0, t2
-st 0(r1), f4
-sub r1, r1, 8
-bnez r1, loop
```

---

**Copyright © 2001 Faisal, from Hill, Smith, Sohi, Vijaykumar, and Wood**

© Chapter 4: Instruction Level Parallelism
Tomasulo Details

Out-of-order loads and stores?
- what about WAW, RAW, WAR?
- compare all load addresses against
  - the addresses in store buffers
  - stall if they match

CDB is bottleneck
- duplicate
- increases required hardware

Complex implementation

Tomasulo

Advantages
- distribution of hazard detection, eliminates WAR and WAW

CDB
- broadcasts results to multiple instructions
- central bottleneck

Register renaming
- eliminates WAR and WAW
- allows dynamic loop unrolling (important with only 4 regs!)
- requires many associative compares
Scoreboard vs. Tomasulo

<table>
<thead>
<tr>
<th>Feature</th>
<th>Scoreboard</th>
<th>Tomasulo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>stall in Issue for FU</td>
<td>stall in Dispatch for RS, stall in RS for FU</td>
</tr>
<tr>
<td>RAW</td>
<td>via Registers</td>
<td>via CDB</td>
</tr>
<tr>
<td>WAR</td>
<td>stall in Write</td>
<td>copy operand to RS</td>
</tr>
<tr>
<td>WAW</td>
<td>stall in Issue</td>
<td>Register renaming</td>
</tr>
<tr>
<td>Logic</td>
<td>centralized</td>
<td>distributed</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>no register bypass</td>
<td>one result bus</td>
</tr>
<tr>
<td></td>
<td>(stall in Issue, block)</td>
<td></td>
</tr>
</tbody>
</table>

Superscalar

Limits on pipeline performance

- latch overheads, signal skew
- “atomic” instruction issue logic
- Flynn bottleneck: CPI >= 1 (why?)

How can we make the CPI = 0.5?

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
i & IF & ID & E & MEM & WB & \\
I+1 & IF & ID & E & MEM & WB & \\
I+2 & IF & ID & E & MEM & WB & \\
I+3 & IF & ID & E & MEM & WB & \\
I+4 & IF & ID & E & MEM & WB & \\
I+5 & IF & ID & E & MEM & WB & \\
\end{array}
\]
Pipeline Stages

IF: parallel access to I-cache, require alignment?
ID: replicate logic, fixed length instrs? hazard checks? dynamic?
EX: parallel/pipelined
MEM: >1 per cycle? If so, hazards, multi-ported register D-cache?
WB: different register files? multi-ported register files?
More things replicated
More possibilities for hazards
More loss due to hazards (why?)

Superscalar

Integer + floating-point
Any two instructions
Any four instructions
Any n instructions?
Superscalar Processors

Single instruction issue limits performance (IPC <= 1)

More pipeline stages (faster clock) has limits

- latch overheads, signal skew
- "atomic" instruction issue

Often combined with dynamic issue (or scheduling)

- although static issue - e.g., Alpha 21064

The Big Picture

Program Form

- Static program
- Dynamic instruction stream
- Execution window
- Completed instructions

Processing Phase

- Instruction fetch & branch prediction
- Dependence checking & dispatch
- Instruction issue
- Instruction execution
- Instruction reorder & commit
Elements of Advanced Superscalar

High performance instruction fetching
  - multiple instructions per cycle
  - branch and jump prediction

dispatch and resolving dependences
  - eliminate false (WAR and WAW) dependences
  - set up linkages for true dependences (RAW)
  - e.g., register renaming

Elements of Advanced Superscalar

Parallel out-of-order instruction issue
Speculative execution
Parallel resources
  - functional units, paths/buses, register ports
high performance memory systems
methods for committing processor state correctly
  - precise interrupts
  - speculative execution
A Generic Superscalar Processor

Diagram of a superscalar processor including components such as floating point register file, instruction buffer, functional units, and memory interface.