Lecture 16a: Dataflow & Superscalar Execution

Prof. Onur Mutlu

ETH Zürich
Spring 2023
27 April 2023
Roadmap for Today (and Past 2-3 Weeks)

- Microarchitecture Fundamentals
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures

- Pipelining & Precise Exceptions
  - Pipelining
  - Pipelined Processor Design
    - Control & Data Dependence Handling
    - Precise Exceptions: State Maintenance & Recovery

- Last+this week: Out-of-Order & Superscalar Execution
  - Out-of-Order Execution
  - Dataflow & Superscalar Execution
  - Branch Prediction
Readings

This week


- H&H Chapters 7.8 and 7.9


Out-of-Order Execution
(Restricted Dataflow)
Review & Wrap Up
1. **Link** the consumer of a value to the producer
   - **Register renaming**: Associate a “tag” with each data value

2. **Buffer** instructions until they are ready
   - Insert instruction into *reservation stations* after renaming

3. Keep **track of readiness** of source values of an instruction
   - Broadcast the “tag” when the value is produced
   - Instructions *compare their “source tags”* to the broadcast tag
     → if match, source value becomes ready

4. When all source values of an instruction are ready, **dispatch** the instruction to functional unit (FU)
   - *Wakeup and select/schedule* the instruction
Recall: Summary of OOO Execution Concepts

- **Register renaming** eliminates false dependences, enables linking of producer to consumers

- **Buffering in reservation stations** enables the pipeline to move for independent (ready) instructions

- **Tag & value broadcast** enables communication (of readiness of produced value) between instructions

- **Wakeup and select** enables out-of-order instruction dispatch into functional units
Recall: OOO Execution: Restricted Dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program

- The dataflow graph is limited to the instruction window
  - Instruction window: all decoded but not yet retired instructions
Recall: State of RAT and RS in Cycle 7

Slightly harder tasks for you:
1. Draw the dataflow graph for the executing code
2. Provide the executing code in sequential order

<table>
<thead>
<tr>
<th>Register</th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

Register Alias Table

RS for ADD Unit

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0 x</td>
</tr>
<tr>
<td>b</td>
<td>1 ~ 2</td>
</tr>
<tr>
<td>c</td>
<td>1 ~ 8</td>
</tr>
<tr>
<td>d</td>
<td>0 a</td>
</tr>
</tbody>
</table>

RS for MUL Unit

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1 ~ 1</td>
</tr>
<tr>
<td>y</td>
<td>0 b</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
</tr>
<tr>
<td>t</td>
<td>0</td>
</tr>
</tbody>
</table>
Recall: Reverse Engineered Dataflow Graph

We can “easily” reverse-engineer the dataflow graph of the executing code!
Recall: OoO Execution w/ Precise Exceptions

- Most modern processors use the following
  - Reorder buffer to support in-order retirement of instructions
  - A single register file (physical RF) to store all registers
    - Both speculative and architectural registers
    - INT and FP are still separate
  - Two register maps store pointers to the physical RF
    - Future/frontend register map → used for renaming
    - Architectural register map → used for maintaining precise state
- This design avoids value replication in RSs, ROB, etc.
Recall: OoO Execution w/ Precise Exceptions (II)

Frontend Register Map

<table>
<thead>
<tr>
<th>Register</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>18</td>
</tr>
<tr>
<td>R2</td>
<td>13</td>
</tr>
<tr>
<td>R3</td>
<td>10</td>
</tr>
<tr>
<td>R4</td>
<td>22</td>
</tr>
<tr>
<td>R5</td>
<td>14</td>
</tr>
<tr>
<td>R6</td>
<td>19</td>
</tr>
<tr>
<td>R7</td>
<td>17</td>
</tr>
<tr>
<td>R8</td>
<td>20</td>
</tr>
<tr>
<td>R9</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td>4</td>
</tr>
<tr>
<td>R11</td>
<td>1</td>
</tr>
</tbody>
</table>

Physical Register File (PRF)

<table>
<thead>
<tr>
<th>PR</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR1</td>
<td>1</td>
</tr>
<tr>
<td>PR2</td>
<td>2</td>
</tr>
<tr>
<td>PR3</td>
<td>3</td>
</tr>
<tr>
<td>PR4</td>
<td>4</td>
</tr>
<tr>
<td>PR5</td>
<td>5</td>
</tr>
<tr>
<td>PR6</td>
<td>6</td>
</tr>
<tr>
<td>PR7</td>
<td>7</td>
</tr>
<tr>
<td>PR8</td>
<td>8</td>
</tr>
<tr>
<td>PR9</td>
<td>9</td>
</tr>
<tr>
<td>PR10</td>
<td>10</td>
</tr>
<tr>
<td>PR11</td>
<td>11</td>
</tr>
<tr>
<td>PR12</td>
<td>12</td>
</tr>
<tr>
<td>PR13</td>
<td>13</td>
</tr>
<tr>
<td>PR14</td>
<td>14</td>
</tr>
<tr>
<td>PR15</td>
<td>15</td>
</tr>
<tr>
<td>PR16</td>
<td>16</td>
</tr>
<tr>
<td>PR17</td>
<td>17</td>
</tr>
<tr>
<td>PR18</td>
<td>18</td>
</tr>
<tr>
<td>PR19</td>
<td>19</td>
</tr>
<tr>
<td>PR20</td>
<td>20</td>
</tr>
<tr>
<td>PR21</td>
<td>21</td>
</tr>
<tr>
<td>PR22</td>
<td>22</td>
</tr>
</tbody>
</table>

Centralized Value Storage

Reorder Buffer (ROB)

<table>
<thead>
<tr>
<th>Entry 0</th>
<th>Entry 1</th>
<th>Entry 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Architectural Register Map

<table>
<thead>
<tr>
<th>Register</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>12</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>10</td>
</tr>
<tr>
<td>R4</td>
<td>22</td>
</tr>
<tr>
<td>R5</td>
<td>5</td>
</tr>
<tr>
<td>R6</td>
<td>9</td>
</tr>
<tr>
<td>R7</td>
<td>11</td>
</tr>
<tr>
<td>R8</td>
<td>20</td>
</tr>
<tr>
<td>R9</td>
<td>7</td>
</tr>
<tr>
<td>R10</td>
<td>6</td>
</tr>
<tr>
<td>R11</td>
<td>1</td>
</tr>
</tbody>
</table>

Pointers to PRF

Pointers to PRF
Recall: OoO Execution w/ Precise Exceptions (III)

At Decode/Rename: Allocate DestPR to Architectural DestReg (RS, ROB)
At Decode/Rename: Read and Update Frontend Register Map

RS for ADD Unit

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>PR</td>
</tr>
<tr>
<td>V</td>
<td>PR</td>
</tr>
</tbody>
</table>

RS for MUL Unit

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>PR</td>
</tr>
<tr>
<td>V</td>
<td>PR</td>
</tr>
</tbody>
</table>

Before Execution: Access Physical Register File to Get Source Values

After Execution: Access Physical Register File to Write Result Values

At Retirement: Update Architectural Register Map with DestPR
Recall: Examples from Modern Processors

Boggs et al., “The Microarchitecture of the Pentium 4 Processor,”
Intel Pentium Pro (1995)

Processor chip

Level 2 cache chip

Multi-chip module package

On-chip Level 2 Cache

https://www.anandtech.com/show/1621/3
General Organization of an OOO Processor

A Modern OoO Design: Intel Pentium 4

Figure 2. Stages of the Alpha 21264 instruction pipeline.

MIPS R10000

IBM POWER4

IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching
IBM POWER5


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).
AMD Zen/Zen 2 (2019)

https://en.wikichip.org/wiki/amd/microarchitectures/zen_2
https://courses.engr.illinois.edu/cs433/fa2020/slides/mini-project-amd-zen.pdf
Apple M1 Firestorm? (2020)

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2
Questions to Ponder

- Why is OoO execution beneficial?
  - **Latency tolerance:** OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently
  - What if all operations take a single cycle?

- What if an instruction takes 1000 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - **What limits the latency tolerance scalability of Tomasulo’s algorithm?**
    - **Instruction window size:** how many decoded but not yet retired instructions you can keep in the machine
Out-of-Order Execution Tradeoffs

**Advantages**

- **Latency tolerance**: Allows independent instructions to execute and complete in the presence of long-latency operations → Higher performance than in-order execution
- **Irregular parallelism**: Dynamically finds and exploits parallel operations in a program → Difficult to find/exploit such parallelism statically

**Disadvantages**

- **Higher complexity**
  - Potentially lengthens critical path delay → clock cycle time
- **More hardware** resources needed

**Recall: Execution time of an entire program**

- \( \text{# of instructions} \times \text{Average CPI} \times \text{clock cycle time} \)
Store-Load Forwarding Complexity

- **Content Addressable Search** (based on Load Address)
- **Range Search** (based on Address and Size of both the Load and earlier Stores)
- **Age-Based Search** (for last written values)
- Load data can come from a combination of multiple places
  - One or more stores in the Store Buffer (SQ)
  - Memory/cache
Other Approaches to Concurrency (or Instruction Level Parallelism)
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
Review: Data Flow:
Exploiting Irregular Parallelism
Recall: OOO Execution: **Restricted Dataflow**

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program

- The dataflow graph is limited to the **instruction window**
  - Instruction window: all decoded but not yet retired instructions
Recall: State of RAT and RS in Cycle 7

Slightly harder tasks for you:
1. Draw the dataflow graph for the executing code
2. Provide the executing code in sequential order

<table>
<thead>
<tr>
<th>Register</th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Register Alias Table**

**RS for ADD Unit**

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
</tbody>
</table>

**RS for MUL Unit**

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>
Recall: Reverse Engineered Dataflow Graph

We can “easily” reverse-engineer the dataflow graph of the executing code!

Dataflow graph

Nodes: operations performed by the instruction

Arrows: tags in Tomasulo’s algorithm

MUL R1, R2 → R3 (X)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)
Data Flow Summary

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)

- Data Flow at the ISA level has not been (as) successful

- **Data Flow** implementations at the **microarchitecture** level (while preserving von Neumann model semantics) have been **very successful**
  - Out of order execution is the prime example

- **Data Flow** mapping of programs to reconfigurable hardware substrates (FPGAs) has also been **successful**
Recall: ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → sequential
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → dataflow

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?
ISA-Level Data Flow Tradeoffs

- Advantages
  - Very good at exploiting irregular parallelism
    - Only real dependences constrain processing
    - More parallelism can be exposed than Von Neumann model

- Disadvantages
  - No precise state semantics
    - Debugging very difficult
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Too much parallelism? (Parallelism control needed)
  - Large hardware overhead (tag matching, data/tag storage)
  - How to enable mutable data structures
  - ...

Recall: ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- **ISA:** Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

- **Microarchitecture:** How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
  - Programmer should see the order specified by the ISA
Readings & Lectures on Data Flow Model


More detailed Lecture Video & Slides on DataFlow:
- http://www.youtube.com/watch?v=D2uue7izU2c
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
Superscalar Execution
Superscalar Execution

- Idea: Fetch, decode, execute, retire multiple instructions per cycle
  - N-wide superscalar $\rightarrow$ N instructions per cycle

- Need to add the hardware resources for doing so

- Hardware performs the dependence checking between concurrently-fetched instructions

- Superscalar execution and out-of-order execution are orthogonal concepts

  - Can have all four combinations of processors:
    
    \[
    \text{[in-order, out-of-order]} \times \text{[scalar, superscalar]} \]


In-Order Superscalar Processor Example

- Multiple copies of datapath: Can fetch/decode/execute multiple instructions per cycle

- Dependences make it tricky to dispatch multiple instructions in the same cycle
  - Need dependence detection between concurrently-fetched instructions

Here: Ideal IPC = 2
In-Order Superscalar Performance Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

Ideal IPC = 2

Actual IPC = 2 (6 instructions issued in 3 cycles)
Superscalar Performance with Dependences

Ideal IPC = 2

Can you reorder the instructions to get IPC = 2?

Actual IPC = 1.2 (6 instructions issued in 5 cycles)

1w $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
Review: How to Handle Data Dependences

- **Six fundamental ways of handling flow dependences**
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Detect and move it out of the way for independent instructions
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

- **Can employ all of these in superscalar processors**
Superscalar Execution Tradeoffs

- **Advantages**
  - Higher instruction throughput
    - Higher IPC: instructions per cycle (i.e., lower CPI)

- **Disadvantages**
  - Higher complexity for dependence checking
    - Requires dependence checking between concurrent instructions
    - Register renaming becomes more complex in an OoO processor
    - Potentially lengthens critical path delay → clock cycle time
  - More hardware resources needed

- **Recall: Execution time of an entire program**
  - \( \text{# of instructions} \times \text{Average CPI} \times \text{clock cycle time} \)
Intel Pentium: In-Order Superscalar (1993)
Alpha 21164: In-Order, Superscalar (1995)


![Processor chip, Level 2 cache chip, Multi-chip module package]
Alpha 21264: OoO, Superscalar (1998)


On-chip Level 2 Cache

https://en.wikichip.org/wiki/amd/microarchitectures/zen_2
https://courses.engr.illinois.edu/cs433/fa2020/slides/mini-project-amd-zen.pdf
Apple M1 Firestorm? (2020)

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2