Module: Superscalar Processors and Hardware Scheduling - Part 1

© Krishna V. Palem, Weng-Fai Wong, and Sudhekar Yalamanchili, Georgia Institute of Technology (slides contributed by Prof. Weng-Fai Wong were prepared while visiting, and employed by Georgia Tech)

Reading for this Module

- Pipeline Review and Multi-Function Pipelines
  - Sections A.1, A.2, A.5,
- The MIPS 64 ISA
  - Section 2.12
- Dynamic Scheduling in Hardware
  - The scoreboard
    - Section A.8
  - Tomasulo’s Algorithm
    - Sections 3.1, 3.2, 3.3
Topical Map for this Module

- MIPS ISA

- Review of Pipelining Basics

- Basics of Multiple Instruction Issue
  - From dependencies to hazards
  - Taxonomy of multiple issue designs

- Hardware Out-of-Order Issue Techniques
  - Scoreboard
  - Tomasulo's Algorithm

The MIPS64 Instruction Set Architecture

- Load/store architecture
  - All operations are on data stored in registers
  - Memory operations only move data to/from registers

- Instruction formats: Simple, fixed field, small number
  - Lead to simplifications in the handling of pipelined processors
  - Uniformity of instruction decoding and control

<table>
<thead>
<tr>
<th>opcode</th>
<th>rd</th>
<th>rs</th>
<th>rt</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Register to register
| Conditional branches
| Immediate operations
| Memory operations

<table>
<thead>
<tr>
<th>opcode</th>
<th>rd</th>
<th>rs</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Unconditional branches

<table>
<thead>
<tr>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Register Files

- **General Purpose Registers (GPR)**
  - 64 bit registers
  - All integer operations are 64 bit
    - Smaller data types loaded and sign extended

- **Floating Point Registers (FPR)**
  - Instructions to move data between GPR and FPR
  - Separate instructions to convert between integer and floating point
  - Two concurrent single precision operations on the contents of a FP register

The MIPS64 Registers and Data Types

- 32, 64 bit general purpose registers
  - R0 hardwired to 0

- 32 double precision floating point registers
  - One half of the register is unused for single precision

- 8, 16, 32, and 64 bit data types
  - Use the 64-bit registers with sign-extension
  - Operate using 64 bit instructions

- Only immediate and displacement addressing modes
  - Encoded in the instruction types
The Memory System

- Memory is byte addressable with 64 bit addresses
- Programmable Little vs. Big Endian mode
- All memory accesses must be aligned
- Branches and jumps are relative to the PC
  - 28 bit PC-relative jump addresses
  - 18 bit PC-relative branch addresses

Sample Instructions

- LD R1, 64(R2)
  - displacement addressing mode
  - Load/store instructions for multiple operand sizes
- DADD R4, R1, R2       DMUL R4, R5, R6
  - Register-to-register operations
- SW R6, 22(R2)
  - Indexed addressed mode
- BNE R1, R2, LOOP       SLT R1, R2, R3
  - Use general purpose registers to store results of relational operations
  - Comparison can be used to synthesize all manner of conditions
A Review of Pipelining

- Stages of an instruction processing pipeline
  - IF: Instruction Fetch
  - ID: Instruction Decode and Operand Read
  - EX: Instruction Execute
  - MEM: Memory Read/Write
  - WB: Result writeback

Without Pipelining

Program Execution
Order (in instructions)

DADD R1, R2, R3
With Pipelining

Program Execution
Order (in instructions)

(Synchronous) Pipelining

- High clock frequencies are achieved with deeper pipelines
- At higher frequencies, clock skew and latch delay become a bigger percentage of the clock cycle → percentage of cycle used by logic is reduced
The Problem with Dependencies

Program Execution
Order (in instructions)

Static Scheduling Solution
Hardware Solution

- If ID.src1 or ID.src2
  == EXE.dest or
  == MEM.dest then

  disable latch over at IF and ID latches

The pipeline is *stalled.*
An Important “Trick”

- **Internal forwarding** move the result directly from the ALU into one of the two source ports of the ALU

- Significantly reduce the number of stall cycles

- Will not work for load instructions - still need hardware stalling for these

---

Internal Forwarding

```
<table>
<thead>
<tr>
<th>Clk 1</th>
<th>Clk 2</th>
<th>Clk 3</th>
<th>Clk 4</th>
<th>Clk 5</th>
<th>Clk 6</th>
<th>Clk 7</th>
<th>Clk 8</th>
<th>Clk 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr Fetch</td>
<td>Decoding &amp; Read Reg</td>
<td>ALU Operation</td>
<td>Memory Operation</td>
<td>Reg Writeback</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr Fetch</td>
<td>Decoding &amp; Read Reg</td>
<td>ALU Operation</td>
<td>Memory Operation</td>
<td>Reg Writeback</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1 = ...</td>
<td>... = R1...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Implementation of Forwarding

- Provide data paths from all possible sources of data and pipeline control to select the correct operands
- Note that instructions immediately dependent on a LD instruction still introduces a stall cycle.

Performance

- **Performance Goal**: Exploit instruction-level parallelism to execute multiple instructions per cycle

- Architectural and compiler techniques to systematically increase the number of instructions executed per clock cycle
- Both hardware and compiler techniques are dependent on the instruction set architecture (ISA)
  - Set of instructions and resources they manipulate
  - The ISA is interface between the hardware and the compiler
Multiple Issue Machines

- Pipelining allows for multiple instructions at different stages of execution
- Advances in VLSI made it possible to put more functional units onto a single processor
- But programs are written for a von Neumann model - single PC, single instruction

Machine Model

- **Latency**: # cycles between an instruction that produces a result and one that consumes the result
- **Initiation interval**: rate at which instructions can be issued to a functional unit
Multiple Issue Machines

Compiler

Front-End & Optimizer

Determine Dependences

Determine Independences

Bind Resources

Sequential (superscalar)

Dependence Architecture (dataflow)

Independence Architecture (VLIW)

Hardware

This Module

Terminology

- **In-order**: events happen in the order dictated by the program’s sequential semantics

- **Instruction issue**: movement of an instruction from the decode stage into the execute stage

- **Instruction completion**: committing of results to the physical destination register (non-standard term)
Flynn’s Bottleneck

- Attributed to Michael Flynn of Stanford

- The throughput of a processor (the number of instructions executed per unit time) cannot exceed the rate of instruction issue

Instruction Level Parallelism

- Parallelism at the level of machine instructions
  - the number of parallel machine instructions that can be issued in a single cycle in a processor

- ILP is limited

- ILP is not evenly distributed
Limitations to ILP

- Data dependences
- Control dependences
- Resource constraints
- Capability of the compiler
  - to expose ILP
  - to effectively utilize CPU resources

Hazards

- Dependencies and resource constraints manifest themselves as potential hazards during pipeline execution
  - Data hazards
    - Ordering constraint between a pair of instructions in the pipeline
    - Three types of hazards: Read-after-Write (RAW), Write-after-Read (WAR) and Write-after-Write (WAW)
  - Structural hazards
    - Two instructions attempting to concurrently access the same hardware component
      - Function units
      - Register file ports
      - Buses
Read-After-Write (RAW) Hazard

- Follows from data dependencies in the program

\[
\begin{align*}
LD & \ R4, \ 32(R3) \\
\ldots & \\
DADD & \ R5, \ R4, \ R2
\end{align*}
\]

- Data dependencies may flow through registers or through memory

\[
\begin{align*}
SD & \ R4, \ 32(R3) \quad A[2^k - p + C] = X \\
\ldots & \\
LD & \ R6, \ 44(R5) \quad Y = A[2^m - p + B]
\end{align*}
\]

Write-After-Read (WAR) Hazard

- Follows from name dependencies
  - Occurs when two instructions share the same physical resource (i.e., name) but do not have a data dependence, for example a register
  - Two types of named dependencies

- Anti-dependence
  - \( \text{instr}_j \) tries to write operand before \( \text{instr}_i \) reads it

\[
\begin{align*}
I: & \ DADD \ R1, \ R2, \ R3 \\
J: & \ DSUB \ R2, \ R4, \ R3
\end{align*}
\]
Write-After-Write (WAW) Hazard

- Output Dependence
  - instr\textsubscript{j} tries to write an operand before instr\textsubscript{i} writes it
  
  Program order

  \[ \begin{align*}
  \text{I: DADD R4, R2, R3} \\
  \text{J: DSUB R4, R1, R3}
  \end{align*} \]

Control Dependencies

- Control dependencies determine execution ordering of program instructions

- Control dependencies
  - Determine which data dependencies are to be honored at run-time

\[ \begin{align*}
\text{DADD R5, R6, R7} \\
\text{BNE R4, R2, CONTINUE} \\
\text{DMUL R4, R2, R5} \\
\text{DSUB R4, R9, R5}
\end{align*} \]
Resource Constraints

- Number of registers
- Number of ports to registers and cache
- Memory bandwidth
- Number of parallel instruction decoders
- Number of functional units of various types
- Number of data paths between various CPU components

all these limit realizable ILP

A Taxonomy of ILP Processors

Canonical Model
Superpipelined Processors

VLIW

Independence Architecture
Superscalar Processors

Sequential Architecture

In-order Issue In-order Completion
• Should there be a hazard (dependences or resource constraints), entire decoding process stalls

• Otherwise, instructions would execute out-of-order - no means of tracking output and anti-dependences

• Straightforward design
In-order Issue Out-of-order Completion

- As soon as execution finishes, result may be completed - even if out-of-order

- Besides ensuring flow dependences, need extra hardware to take care of output dependences

- Two instructions writing to the same register may complete out-of-order - potentially violating output dependences if nothing is done about it

---

In-order Issue Out-of-order Completion: Example

1. DMUL R6, R8, R10
2. DSUB R10, R6, R1
3. DSUB R6, R12, R14
.. ..
n. DADD R16, R6, R20

- If instruction 1. completes after instruction 3., it will lead to incorrect results
- Output dependencies (WAW hazard) must be enforced to ensure correct execution
- WAR hazards cannot occur
A Big Problem - Precise Exceptions

- An exception is precise if the following conditions are met
  - All the instructions preceding the instruction causing the exception have been executed and have modified the process state correctly.
  - All instructions following the instruction causing the exception have not yet been executed and have done no modification to the process state.
  - The instruction that caused the exception may or may not have been executed. This will depend on the definition of the architecture and the cause of the exception.

Machine Model

- Different instructions may be in various stages of execution
Precise Exceptions

- An exception is imprecise if it is not precise

- Precise exception requires in-order execution semantics

- Certain important exceptions have to be precise

- We will return to support for precise exception in out-of-order completion machines later
Out-of-order Issue

Out-of-order Completion

- In-order issue stops as soon as conflict arises
  - no look-ahead

- Out-of-order issue: as soon as inputs are ready, instruction may be issued
  - Finite amount of look-ahead
  - Combined with out-of-order completion gives most ILP

The Instruction Window

- A pool of up to $n$ decoded instructions

- Any ready instruction in this pool may be issued

- Must deal with anti-dependences in addition to flow and output dependences
Out-of-order Issue, Out-of-order Completion: Example

1. DDIV R6, R8, R10
2. DSUB R10, R6, (R1)
   ..
   n. DADD (R1) R6, R20

- If instruction n. completes before instruction 2., reads its operands, it will lead to incorrect results
- Anti-dependencies (WAR hazard) must be enforced to ensure correct execution

Algorithms for Out-of-order Issue

- Scoreboarding
- Tomasulo’s Algorithm
- Others
Scoreboarding

• This idea was first introduced in the CDC 6600 (1964) which had 16 independent functional units -
  - 4 for floating point operations,
  - 5 for memory access and
  - 7 for integer operations.

• The scoreboard maintains information about the instructions, the functional units and the results
Key ideas

- Decompose the decode stage into issue and read operand (RO) steps
  - Stall on WAW or structural hazards in issue stage

- Allow bypassing in RO of independent (in terms of dataflow) instructions
  - Localize stalls → stall data dependent instructions

- Enforce WAR during write back
  - Detect and enforce hazards as late as possible

The Scoreboard

- Consists of:
  - instruction status table
  - functional unit status table
  - result table
Instruction Status Table

- Keeps the information about which activities of the execution process an instruction is currently in.
  - issue? - is the instruction issued?
  - rdopd? - has it completed reading its operands?
  - exec? - has it completed its execution?
  - wrback? - has it completed its writeback?

Functional Unit Status Table

- Has an entry for each functional unit and there are 9 fields for each entry:
  - busy? - indicates if the functional unit is busy;
  - op - the kind of operation being performed;
  - dest - the destination register;
  - src1, src2 - the two source registers;
  - Func1 (Q_i), func2 (Q_j) - the functional units producing the results in the two source registers;
  - ready1?, ready2? - indicates if src1 and src2 is ready;

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>Fk</th>
<th>Q1</th>
<th>Qk</th>
<th>R1</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Yes</td>
<td>Load</td>
<td>F2</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Yes</td>
<td>Sub</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td>Integer</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Result Status Table

<table>
<thead>
<tr>
<th>Unit</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>Mul 1</td>
<td>Integer</td>
<td>Add</td>
<td>Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Maintain an entry per register indicating the functional unit that will write the pending result into the register

Operation Using a Scoreboard

Instruction re-ordering occurs here!
Check for WAW and Structural Hazards
Check for RAW Hazards
Check for WAR Hazards
Scoreboarding - 1

1. When an instruction is fetched, an entry is made in the instruction status table.

2. After the instruction is decoded, the corresponding issue? entry in the instruction status table is marked.

Scoreboarding - 2

3. Select a functional unit. This is obtained by checking the busy? flag of all the functional units which can execute the current instruction.

4. Enter the relevant information in the corresponding functional unit status table. \texttt{func1} and \texttt{func2} are obtained from the corresponding entries in the result table. For example, if one of the source register is \texttt{R1}, then under the entry of \texttt{R1} in the result table, locate the functional unit responsible for writing to this register. If there is no entry, then mark \texttt{ready1}? as ‘ready’. The same goes for \texttt{src2}. This takes care of flow dependency.
5. An instruction is ready for issue if both `ready1?` and `ready2?` entries are marked ‘ready’, and the corresponding entry in the result table for the destination register is empty.

   - If the latter condition is not fulfilled, instruction issue is stalled. This avoid output dependency. This entry is overwritten by the number of the functional unit that will produce the result.

6. The instruction then proceeds to read its operands and executes with the corresponding entry in the instruction status table updated accordingly.

---

### Scoreboarding - 4

7. At the completion of the execution stage, the `busy?` corresponding entry in the functional unit status table is turned to ‘no’.

8. Before write-back, it is necessary to check for anti-dependency. If there exists anti-dependency, the current write-back must be stalled until the hazard is cleared.

9. During write-back, the result is written back to the register, the entry in the result table is turned to ‘empty’ and the instruction’s entry in the instruction status table is deleted. At the same time, the functional unit status table is scanned such that the `ready?` entries can be updated so as to reflect the fact that the result in this register is now ready.
### Example 1: Instruction Flow

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Issue</th>
<th>RO</th>
<th>Execute</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6, 34(R2)</td>
<td>0</td>
<td>1</td>
<td>2-3</td>
<td>4</td>
</tr>
<tr>
<td>L.D F2, 45(R3)</td>
<td>1</td>
<td>2</td>
<td>3-4</td>
<td>5</td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td>2</td>
<td>6</td>
<td>7-11</td>
<td>12</td>
</tr>
<tr>
<td>SUB.D F8,F6,F2</td>
<td>3</td>
<td>6</td>
<td>7-8</td>
<td>9</td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td>4</td>
<td>13</td>
<td>14-28</td>
<td>29</td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td>5</td>
<td>10</td>
<td>11-12</td>
<td>14</td>
</tr>
</tbody>
</table>

- Function unit latencies: FPADD = 2 cycles, FPMULT = 5 cycles, FPDIV = 15 cycles, FPLOAD = 2 cycles, Integer = 1 cycle
- Cannot read and write a register in the same cycle
- All units except FPDIV are pipelined

### Example 2: Instruction Flow

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Issue</th>
<th>RO</th>
<th>Execute</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td>0</td>
<td>1</td>
<td>2-3</td>
<td>4</td>
</tr>
<tr>
<td>MUL.D F2, F0, F3</td>
<td>1</td>
<td>5</td>
<td>6-10</td>
<td>11</td>
</tr>
<tr>
<td>L.D F3, 4(R1)</td>
<td>2</td>
<td>3</td>
<td>4-5</td>
<td>6</td>
</tr>
<tr>
<td>ADD.D F2, F2, F3</td>
<td>12</td>
<td>13</td>
<td>14-15</td>
<td>16</td>
</tr>
<tr>
<td>DSUB R1, R1, 8</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>BNEZ R1, F00</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>S.D F2, 8(R1)</td>
<td>15</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

- Function unit latencies: FPADD = 2 cycles, FPMULT = 5 cycles, FPDIV = 15 cycles, FPLOAD = 2 cycles, Integer = 1 cycle
- Cannot read and write a register in the same cycle
- All units are pipelined
Recap

Determine Dependences
Determine Independences
Bind Resources
Execute

Front-End & Optimizer

Sequential
(superscalar)

Dependence Architecture
(dataflow)

Independence Architecture
(VLIW)

Compiler

Hardware

Summary

- Out-of-order issue and out-of-order completion

- Performance limited by
  - Amount of ILP in the code segments
  - Number of entries in the scoreboard, i.e., amount of look-ahead
  - Number of functional units

- Complexity of scoreboard is on the order of a functional unit
Study Guide

- Given a pipeline structure, being able to write the conditions for existence of RAW, WAW, WAR hazards
- Give a code segment and machine description generate a cycle accurate execution schedule
- Compare and contrast type of ILP processors from the perspective of
  - Types of code segments for which they are best suited
  - HW/SW functionality required
  - Limitations
- Create code segments examples where
  - Large amounts of ILP can be exploited
  - Very little ILP can be exploited