Hardware Components of Any Computer

Five classic components of all computers:

1. Control Unit; 2. Datapath; 3. Memory; 4. Input; 5. Output

Processor or Central Processing Unit (CPU)
CPU Organization

• Datapath Design:
  – Capabilities & performance characteristics of principal Functional Units (FUs):
  – (e.g., Registers, ALU, Shifters, Logic Units, ...)
  – Ways in which these components are interconnected (buses connections, multiplexors, etc.).
  – How information flows between components.

• Control Unit Design:
  – Logic and means by which such information flow is controlled.
  – Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).

• Hardware description with a suitable language, possibly using Register Transfer Notation (RTN).
Hierarchy of Computer Architecture

High-Level Language Programs

Software

Machine Language Program

Software/Hardware Boundary

Hardware

Logic Diagrams

Circuit Diagrams

Application

Operating System

Compiler

Firmware

Instr. Set Proc.

I/O system

Datapath & Control

Digital Design

Circuit Design

Layout

Assembly Language Programs

Instruction Set Architecture

Microprogram

Register Transfer Notation (RTN)
# A Hierarchy of Computer Design

<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Modules</th>
<th>Primitives</th>
<th>Descriptive Notation</th>
<th>Media</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Electronics</td>
<td>Gates, FF’s</td>
<td>Transistors, Resistors, etc.</td>
<td>Circuit Diagrams</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Logic</td>
<td>Registers, ALU’s ...</td>
<td>Gates, FF’s …</td>
<td>Logic Diagrams</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Organization</td>
<td>Processors, Memories</td>
<td>Registers, ALU’s …</td>
<td>Register Transfer Notation (RTN)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Microprogramming</td>
<td>Assembly Language</td>
<td>Microinstructions</td>
<td>Microprogram</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Assembly language programming</td>
<td>OS Routines</td>
<td>Assembly language Instructions</td>
<td>Assembly Language Programs</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Procedural Programming</td>
<td>Applications</td>
<td>OS Routines</td>
<td>High-level Language Programs</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Application</td>
<td>Systems</td>
<td>Procedural Constructs</td>
<td>Problem-Oriented Programs</td>
<td></td>
</tr>
</tbody>
</table>

Low Level - Hardware

Firmware

High Level - Software

---

**EECC550 - Shaaban**

#4 Final Review Spring 2000 5-9-2000
Computer Performance Measures: Program Execution Time

• For a specific program compiled to run on a specific machine “A”, the following parameters are provided:
  – The total instruction count of the program.
  – The average number of cycles per instruction (average CPI).
  – Clock cycle of machine “A”

• How can one measure the performance of this machine running this program?
  – Intuitively the machine is said to be faster or has better performance running this program if the total execution time is shorter.
  – Thus the inverse of the total measured program execution time is a possible performance measure or metric:

\[
\text{Performance}_A = \frac{1}{\text{Execution Time}_A}
\]

How to compare performance of different machines?
What factors affect performance? How to improve performance?
CPU Execution Time: The CPU Equation

• A program is comprised of a number of instructions
  – Measured in: instructions/program

• The average instruction takes a number of cycles per instruction (CPI) to be completed.
  – Measured in: cycles/instruction

• CPU has a fixed clock cycle time = 1/clock rate
  – Measured in: seconds/cycle

• CPU execution time is the product of the above three parameters as follows:

\[
\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]
## Factors Affecting CPU Performance

CPU time = \(\frac{\text{Seconds}}{\text{Program}}\) = \(\text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}\)

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>(X)</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>(X)</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Instruction Set Architecture (ISA)</td>
<td>(X)</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>(X)</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td>(X)</td>
<td></td>
</tr>
</tbody>
</table>

EECC550 - Shaaban

#7 Final Review Spring2000 5-9-2000
Aspects of CPU Execution Time

CPU Time = Instruction count \( \times \) CPI \( \times \) Clock cycle

- Depends on:
  - Program Used
  - Compiler
  - ISA

Instruction Count

- Depends on:
  - Program Used
  - Compiler
  - ISA
  - CPU Organization

CPI

- Depends on:
  - Program Used
  - Compiler
  - ISA
  - CPU Organization

Clock Cycle

- Depends on:
  - CPU Organization
  - Technology
Instruction Types & CPI: An Example

- An instruction set has three instruction classes:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

- Two code sequences have the following instruction counts:

<table>
<thead>
<tr>
<th>Code Sequence</th>
<th>Instruction counts for instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A 2, B 1, C 2</td>
</tr>
<tr>
<td>2</td>
<td>A 4, B 1, C 1</td>
</tr>
</tbody>
</table>

- CPU cycles for sequence 1 = 2 x 1 + 1 x 2 + 2 x 3 = 10 cycles
  CPI for sequence 1 = clock cycles / instruction count
  = 10 / 5 = 2

- CPU cycles for sequence 2 = 4 x 1 + 1 x 2 + 1 x 3 = 9 cycles
  CPI for sequence 2 = 9 / 6 = 1.5
Instruction Frequency & CPI

- Given a program with $n$ types or classes of instructions with the following characteristics:

  $C_i = \text{Count of instructions of type } i$
  $CPI_i = \text{Average cycles per instruction of type } i$
  $F_i = \text{Frequency of instruction type } i$
  
  $= C_i/ \text{total instruction count}$

Then:

$$CPI = \sum_{i=1}^{n} (CPI_i \times F_i)$$
# Instruction Type Frequency & CPI: A RISC Example

Base Machine (Reg / Reg)

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

Typical Mix

\[
CPI = \sum_{i=1}^{n} (CPI_i \times F_i)
\]

\[
CPI = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2
\]
Metrics of Computer Performance

Each metric has a purpose, and each can be misused.
Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used.

- Amdahl’s Law:

  Performance improvement or speedup due to enhancement E:

  \[
  \text{Speedup(E)} = \frac{\text{Execution Time without E}}{\text{Execution Time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}
  \]

  - Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

    \[
    \text{Execution Time with E} = ((1-F) + \frac{F}{S}) \times \text{Execution Time without E}
    \]

    Hence speedup is given by:

    \[
    \text{Speedup(E)} = \frac{\text{Execution Time without E}}{((1-F) + \frac{F}{S}) \times \text{Execution Time without E}} = \frac{1}{(1 - F) + \frac{F}{S}}
    \]

  Note: All fractions here refer to original execution time.
Pictorial Depiction of Amdahl’s Law

Enhancement E accelerates fraction F of execution time by a factor of S

Before:
Execution Time without enhancement E:

<table>
<thead>
<tr>
<th>Unaffected, fraction: (1 - F)</th>
<th>Affected fraction: F</th>
</tr>
</thead>
</table>

Unchanged

<table>
<thead>
<tr>
<th>Unaffected, fraction: (1 - F)</th>
<th>F/S</th>
</tr>
</thead>
</table>

After:
Execution Time with enhancement E:

\[
\text{Speedup}(E) = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + \frac{F}{S}}
\]
Amdahl's Law With Multiple Enhancements: Example

- Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:
  - Speedup$_1$ = $S_1 = 10$ Percentage$_1$ = $F_1 = 20\%$
  - Speedup$_2$ = $S_2 = 15$ Percentage$_1$ = $F_2 = 15\%$
  - Speedup$_3$ = $S_3 = 30$ Percentage$_1$ = $F_3 = 10\%$

- While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.
- What is the resulting overall speedup?

\[
\text{Speedup} = \frac{1}{\left(1 - \sum_i F_i \right) + \sum_i \frac{F_i}{S_i}}
\]

- Speedup = $1 / \left([(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30]\right)$
  = $1 / [ .55 + .0333 ]$
  = $1 / .5833 = 1.71$
MIPS Instruction Formats

- **R-Type**: Operand, operation of the instruction.
- **rs, rt, rd**: The source and destination register specifiers.
- **shamt**: Shift amount.
- **funct**: Selects the variant of the operation in the “op” field.
- **address / immediate**: Address offset or immediate value.
- **target address**: Target address of the jump instruction.

---

# MIPS Instruction Formats

<table>
<thead>
<tr>
<th>R-Type</th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-Type: ALU</th>
<th>Load/Store, Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J-Type: Jumps</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>op</td>
<td>target address</td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
A Single Cycle MIPS Datapath

- Inst Memory
  - Adr
  - Rs, Rt, Rd, Imm16

- ALUctr
- MemWr
- MemtoReg

- ALUSrc
- ExtOp

- 32 32-bit Registers
  - Rw, Ra, Rb

- Mux
  - busA, busB

- Adder
- Extender

- Memory
  - Data

- Clk
- WrEn
- Addr

- Instruction<31:0>
# Control Signal Generation

<table>
<thead>
<tr>
<th></th>
<th>10 0000</th>
<th>10 0010</th>
<th>Don’t Care</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>sub</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>nPCsSel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
<td></td>
</tr>
</tbody>
</table>

See Appendix A
# The Truth Table For The Main Control

<table>
<thead>
<tr>
<th>op</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>“R-type”</th>
<th>Or</th>
<th>Add</th>
<th>Add</th>
<th>Subtract</th>
<th>.xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop &lt;2&gt;</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;1&gt;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;0&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>
PLA Implementation of the Main Control

R-type ori lw sw beq jump

RegWrite ALUSrc RegDst MemtoReg MemWrite Branch Jump ExtOp ALUop<2> ALUop<1> ALUop<0>
**Worst Case Timing (Load)**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Old Value</th>
<th>New Value</th>
<th>Delay Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clk-to-Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
<td>Instruction Memory Access Time</td>
</tr>
<tr>
<td>Rs, Rt, Rd, Op, Func</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUctr</td>
<td></td>
<td></td>
<td>Delay through Control Logic</td>
</tr>
<tr>
<td>ExtOp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUSrc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegWr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>busA</td>
<td></td>
<td></td>
<td>Delay through Extender &amp; Mux</td>
</tr>
<tr>
<td>busB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>busW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Memory Access Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU Delay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register File Access Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Write Occurs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## MIPS Single Cycle Instruction Timing Comparison

### Arithmetic & Logical

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>mux</th>
<th>ALU</th>
<th>mux</th>
<th>setup</th>
</tr>
</thead>
</table>

### Load

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>mux</th>
<th>ALU</th>
<th>Data Mem</th>
<th>mux</th>
<th>setup</th>
</tr>
</thead>
</table>

**Critical Path**

### Store

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>mux</th>
<th>ALU</th>
<th>Data Mem</th>
</tr>
</thead>
</table>

### Branch

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>cmp</th>
<th>mux</th>
</tr>
</thead>
</table>
CPU Design Steps

1. Analyze instruction set operations using independent RTN => datapath requirements.

2. Select set of datapath components & establish clock methodology.

3. Assemble datapath meeting the requirements.

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic.
CPU Design & Implantation Process

- **Bottom-up Design:**
  - Assemble components in target technology to establish critical timing.

- **Top-down Design:**
  - Specify component behavior from high-level requirements.

- **Iterative refinement:**
  - Establish a partial solution, expand and improve.

Diagram:

Instruction Set ➞ Architecture

processor

datapath

- Reg. File
- Mux
- ALU
- Reg
- Mem

control

Decoder
Sequencer

Cells
Gates
Drawback of Single Cycle Processor

• Long cycle time.

• All instructions must take as much time as the slowest:
  – Cycle time for load is longer than needed for all other instructions.

• Real memory is not as well-behaved as idealized memory
  – Cannot always complete data access in one (short) cycle.
Reducing Cycle Time: Multi-Cycle Design

- Cut combinational dependency graph by inserting registers / latches.
- The same work is done in two or more fast cycles, rather than one slow cycle.
Instruction Processing Cycles

1. **Instruction Fetch**: Obtain instruction from program storage.
2. **Next Instruction**: Update program counter to address of next instruction.
3. **Instruction Decode**: Determine instruction type, obtain operands from registers.
4. **Execute**: Compute result value or status.
5. **Result Store**: Store result in register/memory if needed (usually called Write Back).

**Common steps for all instructions**
Partitioning The Single Cycle Datapath

Add registers between smallest steps

Diagram of the single cycle datapath with the following steps:
- Instruction Fetch
- Operand Fetch
- Execution
- Memory Access
- Result Store
Example Multi-cycle Datapath

Registers added:

IR: Instruction register
A, B: Two registers to hold operands read from register file.
R: or ALUOut, holds the output of the ALU
M: or Memory data register (MDR) to hold data read from data memory
# Operations In Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R ← A + B</td>
<td>R ← A OR ZeroExt[imm16]</td>
<td>R ← A + SignEx(Im16)</td>
<td>R ← A + SignEx(Im16)</td>
<td>If Equal = 1 PC ← PC + 4 + (SignExt(imm16) x4) else PC ← PC + 4</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M ← Mem[R]</td>
<td>Mem[R] ← B</td>
<td></td>
</tr>
<tr>
<td><strong>Write Back</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finite State Machine (FSM) Control Model

- State specifies control points for Register Transfer.
- Transfer occurs upon exiting state (same falling edge).

inputs (conditions)

Next State Logic

Control State

Output Logic

outputs (control points)

State X

Register Transfer Control Points

Depends on Input
Control Specification For Multi-cycle CPU
Finite State Machine (FSM)

- IR $\leftarrow$ MEM[PC]  
  "instruction fetch"

- A $\leftarrow$ R[rs]  
  B $\leftarrow$ R[rt]  
  "decode / operand fetch"

- R-type
- ORi
- LW
- SW
- BEQ & Equal
- PC $\leftarrow$ PC + 4
- MEM[R] $\leftarrow$ B
- PC $\leftarrow$ PC + 4
- R[rd] $\leftarrow$ R
- PC $\leftarrow$ PC + 4
- R[rt] $\leftarrow$ R
- PC $\leftarrow$ PC + 4
- R[rt] $\leftarrow$ M
- PC $\leftarrow$ M
- PC $\leftarrow$ PC + 4
- SX || 00

To instruction fetch

To instruction fetch

To instruction fetch
Traditional FSM Controller

datapath + state diagram => control

• Translate RTN statements into control points.

• Assign states.

• Implement the controller.
Mapping RTNs To Control Points Examples & State Assignments

IR $\leftarrow$ MEM[PC]  
0000

“A instruction fetch”

A $\leftarrow$ R[rs]  
B $\leftarrow$ R[rt]  
0001

“decode / operand fetch”

M $\leftarrow$ MEM[S]  
1001

MEM[S] $\leftarrow$ B  
PC $\leftarrow$ PC + 4  
1100

PC $\leftarrow$ PC + 4  
0011

BEQ & Equal

“imem_rd, IRen”

“Aen, Ben”

“ALUfun, Sen”

“RegDst, RegWr, PCen”

To instruction fetch state 0000

To instruction fetch state 0000

To instruction fetch state 0000

To instruction fetch state 0000
## Detailed Control Specification

<table>
<thead>
<tr>
<th>State</th>
<th>Op field</th>
<th>Eq</th>
<th>Next</th>
<th>PC en sel</th>
<th>Ops A B</th>
<th>Exec Ex Sr ALU S</th>
<th>Mem R W M</th>
<th>Write-Back M-R Wr Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>???????</td>
<td>?</td>
<td>0001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>0</td>
<td>0011</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>1</td>
<td>0010</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>R-type</td>
<td>x</td>
<td>0100</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>orI</td>
<td>x</td>
<td>0110</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>LW</td>
<td>x</td>
<td>1000</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SW</td>
<td>x</td>
<td>1011</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0101</td>
<td>0 1 fun</td>
<td>1</td>
<td></td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 0</td>
<td>0 0 or 1</td>
<td></td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>xxxxxx</td>
<td>x</td>
<td>0111</td>
<td>0 0 add 1</td>
<td>1 0</td>
<td></td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 0</td>
<td>1 0 add 1</td>
<td></td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>xxxxxx</td>
<td>x</td>
<td>1001</td>
<td>1 0 add 1</td>
<td></td>
<td></td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>xxxxxx</td>
<td>x</td>
<td>1010</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 0</td>
<td>1 0 add 1</td>
<td></td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>xxxxxx</td>
<td>x</td>
<td>1100</td>
<td>1 0</td>
<td>1 0 add 1</td>
<td></td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1 0</td>
<td></td>
<td></td>
<td>0 1</td>
<td></td>
</tr>
</tbody>
</table>
Alternative Multiple Cycle Datapath (In Textbook)

- Shared instruction/data memory unit
- A single ALU shared among instructions
- Shared units require additional or widened multiplexors
- Temporary registers to hold data between clock cycles of the instruction:
  - Additional registers: Instruction Register (IR), Memory Data Register (MDR), A, B, ALUOut
# Operations In Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Fetch</strong></td>
<td><strong>Decode</strong></td>
<td><strong>Memory</strong></td>
<td><strong>Write Back</strong></td>
<td><strong>Branch</strong></td>
</tr>
<tr>
<td><strong>Fetch</strong></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>ALUout ← A + B</td>
<td>ALUout ← A + ZeroExt[imm16]</td>
<td>ALUout ← A + SignEx[Im16]</td>
<td>ALUout ← A + SignEx[Im16]</td>
<td>If Equal = 1</td>
</tr>
<tr>
<td></td>
<td>A OR ZeroExt[imm16]</td>
<td></td>
<td></td>
<td></td>
<td>PC ← ALUout</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finite State Machine (FSM) Specification

IR ← MEM[PC]
PC ← PC + 4

A ← R[rs]
B ← R[rt]

ALUout ← PC + SX

R-type
ALUout ← A fun B
ALUout ← A op ZX
ALUout ← A + SX
ALUout ← A + SX

ORi
ALUout ← A + SX

LW
M ← MEM[ALUout]

SW
MEM[ALUout] ← B

BEQ
If A = B then
PC ← ALUout

Execute
R[rd] ← ALUout
R[rt] ← ALUout
R[rt] ← M

To instruction fetch
To instruction fetch
To instruction fetch
To instruction fetch
To instruction fetch
MIPS Multi-cycle Datapath
Performance Evaluation

• What is the average CPI?
  – State diagram gives CPI for each instruction type
  – Workload below gives frequency of each type

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI(_i) for type</th>
<th>Frequency</th>
<th>CPI(_i) x freq(_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Average CPI: 4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).
Control Implementation Alternatives

- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.

<table>
<thead>
<tr>
<th>Initial Representation</th>
<th>Finite State Diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Explicit Next State Function</td>
<td>Microprogram counter + Dispatch ROMs</td>
</tr>
<tr>
<td></td>
<td>Logic Equations</td>
<td>Truth Tables</td>
</tr>
<tr>
<td></td>
<td>PLA</td>
<td>ROM</td>
</tr>
</tbody>
</table>

- "hardwired control"
- "microprogrammed control"
Microprogrammed Control

- Finite state machine control for a full set of instructions is very complex, and may involve a very large number of states:
  - Slight microoperation changes require new FSM controller.

- Microprogramming: Designing the control as a program that implements the machine instructions.

- A microprogram for a given machine instruction is a symbolic representation of the control involved in executing the instruction and is comprised of a sequence of microinstructions.

- Each microinstruction defines the set of datapath control signals that must asserted (active) in a given state or cycle.

- The format of the microinstructions is defined by a number of fields each responsible for asserting a set of control signals.

- Microarchitecture:
  - Logical structure and functional capabilities of the hardware as seen by the microprogrammer.
Types of "branching"
- Set state to 0 (fetch)
- Dispatch i (state 1)
- Use incremented address (seq) state number 2

Microprogrammed Control Unit

Microprogram Storage
ROM/PLA

Outputs
Microinstruction Address
Control Signal Fields

Inputs
Sequencing Control Field

Multicycle Datapath

Types of "branching"
- Set state to 0 (fetch)
- Dispatch i (state 1)
- Use incremented address (seq) state number 2

Microprogram Counter, MicroPC

Opcode

Address Select Logic

State Reg

Adder

1
### List of control Signals Grouped Into Fields

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSelA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg. is written</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU</td>
<td>Reg. write data input = memory</td>
</tr>
<tr>
<td>RegDst</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read, MDR ← Mem[addr]</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
</tr>
<tr>
<td>IorD</td>
<td>Memory address = PC</td>
<td>Memory address = S</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR ← Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC ← PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC ← PCSource</td>
</tr>
<tr>
<td>PCSrc</td>
<td>PCSrc = ALU</td>
<td>PCSrc = ALUout</td>
</tr>
</tbody>
</table>

### Single Bit Control

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td>00</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>ALUSelB</td>
<td>000</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>2nd ALU input = sign extended IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>2nd ALU input = sign extended, shift left 2 IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2nd ALU input = zero extended IR[15-0]</td>
</tr>
</tbody>
</table>
## Microinstruction Field Values

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field with Specific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>Subt.</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>Func code</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>Or</td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input = PC</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>4</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>Extend</td>
<td>2nd ALU input = sign ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extend0</td>
<td>2nd ALU input = zero ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extshft</td>
<td>2nd ALU input = sign ex., sl IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td>destination</td>
<td>rd ALU</td>
<td>Reg[rd] ← ALUout</td>
</tr>
<tr>
<td></td>
<td>rt ALU</td>
<td>Reg[rt] ← ALUout</td>
</tr>
<tr>
<td></td>
<td>rt Mem</td>
<td>Reg[rt] ← Mem</td>
</tr>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td></td>
<td>Read ALU</td>
<td>Read memory using ALU output</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write memory using ALU output, value B</td>
</tr>
<tr>
<td>Memory register</td>
<td>IR</td>
<td>IR ← Mem</td>
</tr>
<tr>
<td>PC write</td>
<td>ALU</td>
<td>PC ← ALU</td>
</tr>
<tr>
<td></td>
<td>ALUoutCond</td>
<td>IF ALU Zero then PC ← ALUout</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>Go to the first microinstruction</td>
</tr>
<tr>
<td></td>
<td>Dispatch i</td>
<td>Dispatch using ROM.</td>
</tr>
</tbody>
</table>
# Microprogram for The Control Unit

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td></td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dispatch</td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td></td>
<td>Read ALU</td>
<td></td>
<td>rt MEM</td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Rtype:</td>
<td>Func</td>
<td>rs</td>
<td>rt</td>
<td>rd ALU</td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALUoutCond.</td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
</tbody>
</table>
MIPS Integer ALU Requirements

(1) Functional Specification:
inputs: 2 x 32-bit operands A, B, 4-bit mode
outputs: 32-bit result S, 1-bit carry, 1 bit overflow, 1 bit zero
operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

(2) Block Diagram:

10 operations thus 4 control bits

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>add</td>
</tr>
<tr>
<td>01</td>
<td>addU</td>
</tr>
<tr>
<td>02</td>
<td>sub</td>
</tr>
<tr>
<td>03</td>
<td>subU</td>
</tr>
<tr>
<td>04</td>
<td>and</td>
</tr>
<tr>
<td>05</td>
<td>or</td>
</tr>
<tr>
<td>06</td>
<td>xor</td>
</tr>
<tr>
<td>07</td>
<td>nor</td>
</tr>
<tr>
<td>12</td>
<td>slt</td>
</tr>
<tr>
<td>13</td>
<td>sltU</td>
</tr>
</tbody>
</table>
Building Block: 1-bit ALU

Performs: AND, OR,
addition on A, B or A, B inverted

invertB

CarryIn

Operation

A

and

or

B

1-bit Full Adder

add

Result

CarryOut
32-Bit ALU Using 32 1-Bit ALUs

32-bit rippled-carry adder
(operation/invertB lines not shown)

Addition/Subtraction Performance:

Assume gate delay = T
Total delay = 32 x (1-Bit ALU Delay)
= 32 x 2 x gate delay
= 64 x gate delay
= 64 T
MIPS ALU With SLT Support Added
Improving ALU Performance: Carry Look Ahead (CLA)

\[ C_1 = G_0 + C_0 \cdot P_0 \]

\[ C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]

\[ C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

- \( P = A \text{ and } B \)
- \( G = A \text{ xor } B \)

Truth Table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C-in</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C-in</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- "kill"
- "propagate"
- "generate"
Cascaded Carry Look-ahead
16-Bit Example

\[ C_1 = G_0 + C_0 \cdot P_0 \]
\[ C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]
\[ C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

Delay = \( 2 + 2 + 1 = 5 \) gate delays = \( 5T \)

Assuming all gates have equal delay \( T \)
Unsigned Multiplication Example

- Paper and pencil example (unsigned):
  
  
  Multiplicand      1000  
  Multiplier        1001  
  
  1000
  0000
  0000
  
  1000
  01001000
  
  Product

- $m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}$, $m = 32, n = 32, 64 \text{ bit product}$.

- The binary number system simplifies multiplication:
  
  0 => place 0 (0 x multiplicand).
  1 => place a copy (1 x multiplicand).

- We will examine 4 versions of multiplication hardware & algorithm:
  
  –Successive refinement of design.
At each stage shift A left (x 2).
Use next bit of B to determine whether to add in shifted multiplicand.
Accumulate 2n bit partial product at each stage.
MULTIPLY HARDWARE Version 3

- Combine Multiplier register and Product register:
  - 32-bit Multiplicand register.
  - 32-bit ALU.
  - 64-bit Product register, (0-bit Multiplier register).
Multiply Algorithm
Version 3

Product0 = 1

1. Test Product0

Product0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit.

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done
Booth’s Algorithm

```
<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>000111\textcolor{red}{1000}</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>00011\textcolor{red}{11000}</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>00\textcolor{red}{01}111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>000111\textcolor{red}{11000}</td>
<td>none</td>
</tr>
</tbody>
</table>
```

- Originally designed for Speed (when shift was faster than add).
- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one.
# Booths Example (2 x 7)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 0111 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. $P = P - m$</td>
<td>1110</td>
<td>+ 1110</td>
<td>1110 0111 0</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0011 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>2.</td>
<td>0010</td>
<td>1111 1001 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>3.</td>
<td>0010</td>
<td>1111 1100 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td>4a.</td>
<td>0010</td>
<td>+ 0010</td>
<td>0001 1100 1</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>0000 1110 0</td>
<td>done</td>
</tr>
</tbody>
</table>
Combinational Shifter from MUXes

Basic Building Block

A \quad B

sel \quad 1 \quad 0

D

8-bit right shifter

A_7 \quad A_6 \quad A_5 \quad A_4 \quad A_3 \quad A_2 \quad A_1 \quad A_0

1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0

\quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0

\quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0

\quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0

R_7 \quad R_6 \quad R_5 \quad R_4 \quad R_3 \quad R_2 \quad R_1 \quad R_0

- What comes in the MSBs?
- How many levels for 32-bit shifter?
Division

\[
\begin{array}{c|c}
\text{Divisor} & 1001 \\
\hline
\text{Dividend} & 1001010 \\
-1000 & 1001010 \\
10 & 101 \\
101 & 1010 \\
1010 & 101010 \\
-1000 & 101010 \\
10 & 101010 \\
\end{array}
\]

\begin{itemize}
\item See how big a number can be subtracted, creating quotient bit on each step:

\[
\text{Binary } \Rightarrow 1 \times \text{divisor} \text{ or } 0 \times \text{divisor}
\]

\[
\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}
\]

\[
\Rightarrow |\text{Dividend}| = |\text{Quotient}| + |\text{Divisor}|
\]

\item 3 versions of divide, successive refinement
\end{itemize}
DIVIDE HARDWARE Version 3

- 32-bit Divisor register.
- 32-bit ALU.
- 64-bit Remainder register (0-bit Quotient register).

Diagram:
- Divisor
  - 32 bits
- 32-bit ALU
  - "HI"
  - "LO"
- Remainder (Quotient)
  - 64 bits
- Control
  - Write
  - Shift Left

Diagram flow:
1. Divisor (32 bits)
2. 32-bit ALU
   - "HI"
   - "LO"
3. Remainder (Quotient)
   - 64 bits
4. Control
   - Write
   - Shift Left
Divide Algorithm
Version 3

Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Remainder >= 0

Remainder < 0

Test Remainder

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

n\text{th repetition}?

No: < n repetitions

Yes: n repetitions (n = 4 here)

Done. Shift left half of Remainder right 1 bit.
Representation of Floating Point Numbers in Single Precision  *IEEE 754 Standard*

Value = \( N = (-1)^S \times 2^{E-127} \times (1.M) \)

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent: excess 127 binary integer added</th>
<th>mantissa: sign + magnitude, normalized binary significand with a hidden integer bit: 1.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 < \( E < 255 \)  
Actual exponent is: \( e = E - 127 \)

Example:  
0 = 0 00000000 0 ... 0  
-1.5 = 1 01111111 10 ... 0

Magnitude of numbers that can be represented is in the range:

2\(^{-126}\) (1.0) to 2\(^{127}\) (2 - 2\(^{-23}\))

Which is approximately:

1.8 \times 10^{-38} to 3.40 \times 10^{38}
Representation of Floating Point Numbers in Single Precision \textit{IEEE 754 Standard}

Value = N = (-1)^S \times 2^{E-127} \times (1.M)

- \text{Sign:} S
- \text{Exponent:} E
- \text{Mantissa:} M

0 < E < 255
Actual exponent is: e = E - 127

Example: 0 = 0.00000000 0 \ldots 0
-1.5 = 1.01111111 10 \ldots 0

Magnitude of numbers that can be represented is in the range:

\[ 2^{-126} \text{(1.0)} \text{ to } 2^{127} \left(2 - 2^{-23}\right) \]

Which is approximately:

\[ 1.8 \times 10^{-38} \text{ to } 3.40 \times 10^{38} \]
Representation of Floating Point Numbers in **IEEE 754 Standard**

Value = \( N = (-1)^S \times 2^{E-1023} \times (1.M) \)

- **Sign** \( S \)
- **Exponent** \( E \)
- **Mantissa** \( M \)

0 < \( E < 2047 \)

Actual exponent is:
\( e = E - 1023 \)

Example:
0 = 0 00000000000 0 \ldots 0

-1.5 = 1 01111111111 10 \ldots 0

Magnitude of numbers that can be represented is in the range:

\( 2^{-1022} (1.0) \) to \( 2^{1023} (2 - 2^{-52}) \)

Which is approximately:

\( 2.23 \times 10^{308} \) to \( 1.8 \times 10^{308} \)
Floating Point Conversion Example

- The decimal number $-2345.125_{10}$ is to be represented in the **IEEE 754** 32-bit single precision format:

  $$-2345.125_{10} = -100100101001.001_2$$
  
  (converted to binary)
  
  $$= -1.00100101001001 \times 2^{11}$$
  
  (normalized binary)

- The mantissa is negative so the sign $S$ is given by:

  $$S = 1$$

- The biased exponent $E$ is given by $E = e + 127$

  $$E = 11 + 127 = 138_{10} = 10001010_2$$

- Fractional part of mantissa $M$:

  $$M = .0010010100100100000000$$
  
  (in 23 bits)

The **IEEE 754** single precision representation is given by:

<table>
<thead>
<tr>
<th></th>
<th>10001010</th>
<th>0010010100100100100000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>1 bit</td>
</tr>
<tr>
<td>E</td>
<td>10001010</td>
<td>8 bits</td>
</tr>
<tr>
<td>M</td>
<td>001001010010010010000000000</td>
<td>23 bits</td>
</tr>
</tbody>
</table>
Floating Point Addition

Flowchart

Start

(1) Compare the exponents of the two numbers shift the smaller number to the right until its exponent matches the larger exponent

(2) Add the significands (mantissas)

(3) Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

(4) Overflow or Underflow?
   Yes
   Generate exception or return error
   No
   Round the significand to the appropriate number of bits
   If mantissa = 0, set exponent to 0

(5) Still normalized?
   Yes
   Done
   No

Done
Floating Point Addition Hardware

- Sign
- Exponent
- Significand

- Small ALU
  - Exponent difference
  - Control
- Shift right
- Big ALU
  - Increment or decrement
  - Shift left or right
  - Rounding hardware
  - Rounding hardware

- Compare exponents
- Shift smaller number right
- Add
- Normalize
- Round

Sign  Exponent  Significand

Sign  Exponent  Significand
**Floating Point Multiplication Flowchart**

1. Is one/both operands = 0?
   - Set the result to zero: exponent = 0

2. Compute exponent: biased exp.(X) + biased exp.(Y) - bias

3. Compute sign of result: Xs XOR Ys

4. Multiply the mantissas

5. Normalize mantissa if needed

6. Overflow or Underflow?
   - Yes: Generate exception or return error
   - No: Round or truncate the result mantissa

7. Still normalized?
   - Yes: Done
   - No: Continue flowchart
Extra Bits for Rounding

Extra bits used to prevent or minimize rounding errors.

How many extra bits?
IEEE: As if computed the result exactly and rounded.

Addition:

\[
\begin{array}{ccc}
1.xxxxx & 1.xxxxx & 1.xxxxx \\
+ 1.xxxxx & 0.001xxxxx & 0.01xxxxx \\
1x.xxxxxy & 1.xxxxxxyyy & 1x.xxxxxxyyy \\
\end{array}
\]

post-normalization   pre-normalization   pre and post

- **Guard Digits**: digits to the right of the first \( p \) digits of significand to guard against loss of digits – can later be shifted left into first \( P \) places during normalization.
- Addition: carry-out shifted in
- Subtraction: borrow digit and guard
- Multiplication: carry and guard, Division requires guard
Rounding Digits

Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., B = 10, p = 3:

\[
\begin{array}{ccc}
0 & 2 & 1.69 \\
- & 0 & 0 7.85 \\
0 & 2 & 1.61 \\
\end{array}
\]

= \[1.6900 \times 10^{2-\text{bias}}\]

= \[-.0785 \times 10^{2-\text{bias}}\]

= \[1.6115 \times 10^{2-\text{bias}}\]

One round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit.

\textit{IEEE Standard:}

four rounding modes: round to nearest (default)
round towards plus infinity
round towards minus infinity
round towards 0

round to nearest:
round digit < B/2 then truncate
> B/2 then round up (add 1 to ULP: unit in last place)
= B/2 then round to nearest even digit

\textit{it can be shown that this strategy minimizes the mean error introduced by rounding.}
Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

\[ d_0 \cdot d_1 d_2 d_3 \ldots dp-1 \ 0 \ 0 \ 0 \]
\[ 0 \cdot 0 \ 0 \ X \ldots X \ X X \ S \]
\[ X X \ S \]

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

\[ d_0 \cdot d_1 d_2 d_3 \ldots dp-1 \ 0 \ 0 \ 0 \]
\[ 0 \cdot 0 \ 0 \ X \ldots X \ X X \ 0 \]
\[ d_0 \cdot d_1 d_2 d_3 \ldots dp-1 \ 0 \ 0 \ 0 \]
\[ 0 \cdot 0 \ 0 \ X \ldots X \ X X \ 1 \]

generates a borrow

Rounding Summary:

Radix 2 minimizes wobble in precision.

Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit.

One round digit needed for correct rounding.

Sticky bit needed when round digit is B/2 for max accuracy.

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed.
Pipelining: Design Goals

- The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.
- An important pipeline design consideration is to balance the length of each pipeline stage.
- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):
  
  \[
  \text{Time per instruction on unpipelined machine} \div \text{Number of pipe stages}
  \]

- Under these ideal conditions:
  - Speedup from pipelining = the number of pipeline stages = k
  - One instruction is completed every cycle: CPI = 1.
# Pipelined Instruction Processing Representation

<table>
<thead>
<tr>
<th>Instruction Number</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction I</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pipeline Stages:**
- **IF** = Instruction Fetch
- **ID** = Instruction Decode
- **EX** = Execution
- **MEM** = Memory Access
- **WB** = Write Back

**Time to fill the pipeline** is indicated by the arrows pointing left and right. The first instruction, I, is completed in clock cycle 4, and the last instruction, I+4, is completed in clock cycle 8.
Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

- **Single Cycle Machine:**
  - 8 ns/cycle x 1 CPI x 1000 inst = 8000 ns

- **Multicycle Machine:**
  - 2 ns/cycle x 4.6 CPI (due to inst mix) x 1000 inst = 9200 ns

- **Ideal pipelined machine, 5-stages:**
  - 2 ns/cycle x (1 CPI x 1000 inst + 4 cycle fill) = 2008 ns
Single Cycle, Multi-Cycle, Vs. Pipeline

**Single Cycle Implementation:**
- **Load**: 8 ns
- **Store**: Waste

**Multiple Cycle Implementation:**
- Cycle 1: Load
- Cycle 2: Store
- Cycle 3: Waste
- Cycle 4: Load
- Cycle 5: Store
- Cycle 6: Waste
- Cycle 7: Load
- Cycle 8: Store
- Cycle 9: Waste
- Cycle 10: Load

**Pipeline Implementation:**
- **Load**: IF, ID, EX, MEM, WB
- **Store**: IF, ID, EX, MEM, WB
- **R-type**: IF, ID, EX, MEM, WB
MIPS: A Pipelined Datapath
Pipeline Control

- Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst ALU Op1 ALU Op0 ALU Src Branch Mem Read Mem Write Reg write Mem to Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 0 1 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>0 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 1 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>X 0 0 1 0 0 0 0 0 0 0 0 0 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>X 0 0 1 0 0 0 0 0 0 0 0 0 X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Performance Issues In Pipelining

• Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 1

• Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).

• It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.
Pipelining Performance Example

• **Example:** For an unpipelined machine:
  - Clock cycle = 10ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 1ns to the machine clock cycle then the speedup in instruction execution from pipelining is:

  Non-pipelined Average instruction execution time = Clock cycle x Average CPI
  
  $= 10 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) = 10 \text{ ns} \times 4.4 = 44 \text{ ns}$

  In the pipelined five implementation five stages are used with an average instruction execution time of: 10 ns + 1 ns = 11 ns

  Speedup from pipelining = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}
  
  $= \frac{44 \text{ ns}}{11 \text{ ns}} = 4 \text{ times}$
Pipeline Hazards

• Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle resulting in one or more stall cycles.

• Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  – Structural hazards: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  – Data hazards: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  – Control hazards: Arise from the pipelining of conditional branches and other instructions that change the PC.
Structural hazard Example:
Single Memory For Instructions & Data

Time (clock cycles)

Load
Instr 1
Instr 2
Instr 3
Instr 4

Detection is easy in this case (right half highlight means read, left half write)
Data Hazards Example

- Problem with starting next instruction before first is finished
  - Data dependencies here that “go backward in time” create data hazards.

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Data Hazard Resolution: Stall Cycles

Stall the pipeline by a number of cycles. The control unit must detect the need to insert stall cycles. In this case two stall cycles are needed.

Program execution order (in instructions)

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)

Value of register $2:

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
<th>CC 10</th>
<th>CC 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC 1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>CC 3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>CC 4</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>CC 5</td>
<td>10/20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 6</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 7</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 8</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 9</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 10</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CC 11</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
Data Hazard Resolution: Stall Cycles

Stall the pipeline by a number of cycles. The control unit must detect the need to insert stall cycles. In this case two stall cycles are needed.
Performance of Pipelines with Stalls

- Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:

\[
\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}
\]

- When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}
\]
Data Hazard Resolution: Forwarding

- Register file forwarding to handle read/write to same register
- ALU forwarding
Data Hazard Example With Forwarding

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
</tr>
<tr>
<td>Value of EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)
A Data Hazard Requiring A Stall

A load followed by an R-type instruction that uses the loaded value

Even with forwarding in place a stall cycle is needed
This condition must be detected by hardware
Compiler Scheduling Example

• Reorder the instructions to avoid as many pipeline stalls as possible:

\[
\begin{align*}
\text{lw} & \quad $15, 0 (\$2) \\
\text{lw} & \quad $16, 4 (\$2) \\
\text{sw} & \quad $16, 0 (\$2) \\
\text{sw} & \quad $15, 4 (\$2)
\end{align*}
\]

• The data hazard occurs on register $16$ between the second \text{lw} and the first \text{sw} resulting in a stall cycle

• With forwarding we need to find only one independent instructions to place between them, swapping the \text{lw} instructions works:

\[
\begin{align*}
\text{lw} & \quad $15, 0 (\$2) \\
\text{lw} & \quad $16, 4 (\$2) \\
nop & \\
nop & \\
\text{sw} & \quad $15, 0 (\$2) \\
\text{sw} & \quad $16, 4 (\$2)
\end{align*}
\]

• Without forwarding we need three independent instructions to place between them, so in addition two nops are added.

\[
\begin{align*}
\text{lw} & \quad $15, 0 (\$2) \\
\text{lw} & \quad $16, 4 (\$2) \\
nop & \\
nop & \\
\text{sw} & \quad $15, 0 (\$2) \\
\text{sw} & \quad $16, 4 (\$2)
\end{align*}
\]
Control Hazards: Example

- Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.

- In the above diagram, we are predicting “branch not taken”
  - Need to add hardware for flushing the three following instructions if we are wrong losing three cycles.
Reducing Delay of Taken Branches

- Next PC of a branch known in MEM stage: Costs three lost cycles if taken.
- If next PC is known in EX stage, one cycle is saved.
- Branch address calculation can be moved to ID stage using a register comparator, costing only one cycle if branch is taken.
**Pipeline Performance Example**

- Assume the following MIPS instruction mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
</tr>
<tr>
<td>Load</td>
<td>30% of which 25% are followed immediately by an instruction using the loaded value</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>branch</td>
<td>20% of which 45% are taken</td>
</tr>
</tbody>
</table>

- What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage?

\[
\text{CPI} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\
= 1 + \text{stalls by loads} + \text{stalls by branches} \\
= 1 + .3 \times .25 \times 1 + .2 \times .45 \times 1 \\
= 1 + .075 + .09 \\
= 1.165
\]
Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap

- CPU: μProc 60%/yr.
- DRAM: 7%/yr.

Processor-Memory Performance Gap:
(grows 50% / year)
Processor-DRAM Performance Gap Impact: Example

- To illustrate the performance impact, assume a pipelined RISC CPU with CPI = 1 using non-ideal memory.
- Over an 10 year period, ignoring other factors, the cost of a full memory access in terms of number of wasted instructions:

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU speed MZH</th>
<th>CPU cycles ns</th>
<th>Memory Access ns</th>
<th>Minimum CPU cycles or instructions wasted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986:</td>
<td>8</td>
<td>125</td>
<td>190</td>
<td>190/125 = 1.5</td>
</tr>
<tr>
<td>1988:</td>
<td>33</td>
<td>30</td>
<td>175</td>
<td>175/30 = 5.8</td>
</tr>
<tr>
<td>1991:</td>
<td>75</td>
<td>13.3</td>
<td>155</td>
<td>155/13.3 = 11.65</td>
</tr>
<tr>
<td>1994:</td>
<td>200</td>
<td>5</td>
<td>130</td>
<td>130/5 = 26</td>
</tr>
<tr>
<td>1996:</td>
<td>300</td>
<td>3.33</td>
<td>100</td>
<td>110/3.33 = 33</td>
</tr>
</tbody>
</table>
Memory Hierarchy: Motivation

The Principle Of Locality

• Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).

• Two Types of locality:
  – Temporal Locality: If an item is referenced, it will tend to be referenced again soon.
  – Spatial locality: If an item is referenced, items whose addresses are close will tend to be referenced soon.

• The presence of locality in program behavior, makes it possible to satisfy a large percentage of program access needs using memory levels with much less capacity than program address space.
Levels of The Memory Hierarchy

One or more levels (Static RAM):
Level 1: On-chip 16-64K
Level 2: On or Off-chip 128-512K
Level 3: Off-chip 128K-8M

Dynamic RAM (DRAM)
16M-16G

Interface:
SCSI, RAID,
IDE, 1394
4G-100G

Part of The On-chip
CPU Datapath
16-256 Registers

Farther away from
The CPU
Lower Cost/Bit
Higher Capacity
Increased Access
Time/Latency
Lower Throughput

EECC550 - Shaaban
A Typical Memory Hierarchy
(With Two Levels of Cache)

- Faster
- Larger Capacity

Processor
  - Control
  - Datapath
    - Registers
    - On-Chip Level One Cache
  - Second Level Cache (SRAM) \( L_2 \)
  - Main Memory (DRAM)
  - Virtual Memory, Secondary Storage (Disk)
  - Tertiary Storage (Tape)

Speed (ns):
- 1s
- 10s
- 100s
- 1,000,000,000s (10s ms)
- 1,000,000,000,000s (10s sec)

Size (bytes):
- 100s
- Ks
- Ms
- Gs
- Ts
Cache Design & Operation Issues

• Q1: Where can a block be placed cache?  
  *(Block placement strategy & Cache organization)*  
  – Fully Associative, Set Associative, Direct Mapped

• Q2: How is a block found if it is in cache?  
  *(Block identification)*  
  – Tag/Block

• Q3: Which block should be replaced on a miss?  
  *(Block replacement)*  
  – Random, LRU

• Q4: What happens on a write?  
  *(Cache write policy)*  
  – Write through, write back
FIGURE 5.2 This example cache has eight block frames and memory has 32 blocks.
Locating A Data Block in Cache

• Each block frame in cache has an address tag.
• The tags of every cache block that might contain the required data are checked or searched in parallel.
• A valid bit is added to the tag to indicate whether this entry contains a valid address.
• The address from the CPU to cache is divided into:
  – A block address, further divided into:
    • An index field to choose a block set in cache.
      (no index field when fully associative).
    • A tag field to search and match addresses in the selected set.
  – A block offset to select the data from the block.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>
Address Field Sizes

Block Address

- Tag
- Index

Block Offset

Physical Address Generated by CPU

Block offset size = \( \log_2(\text{block size}) \)

Index size = \( \log_2(\text{Total number of blocks/associativity}) \)

Tag size = address size - index size - offset size
Four-Way Set Associative Cache: MIPS Implementation Example

256 sets
1024 block frames
Cache Organization/Addressing Example

- Given the following:
  - A single-level $L_1$ cache with 128 cache block frames
    - Each block frame contains four words (16 bytes)
  - 16-bit memory addresses to be cached (64K bytes main memory or 4096 memory blocks)

- Show the cache organization/mapping and cache address fields for:
  - Fully Associative cache.
  - Direct mapped cache.
  - 2-way set-associative cache.
Cache Example: Fully Associative Case

- Block offset = 4 bits
- Tag = 12 bits
- Block Address = 12 bits

All 128 tags must be checked in parallel by hardware to locate a data block.

Cache (with 128 blocks)
**Cache Example: Direct Mapped Case**

Block Address = 12 bits

- **Tag** = 5 bits
- **Index** = 7 bits
- **Block offset** = 4 bits

Main Memory

<table>
<thead>
<tr>
<th>Block</th>
<th>BLOCK 0</th>
<th>BLOCK 1</th>
<th>BLOCK 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK 127</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 129</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 257</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK 4095</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cache (with 128 blocks)**

- **V** (Valid bit)
- **TAG**

**Block Example:**

- **Block Offset** = 4 bits
- **Tag** = 5 bits
- **Index** = 7 bits

Only a single tag must be checked in parallel to locate a data block.
Cache Example: 2-Way Set-Associative

Block offset = 4 bits
Block Address = 12 bits
Tag = 6 bits
Index = 6 bits

Two tags in a set must be checked in parallel to locate a data block

Main Memory

Valid bits not shown
Cache Replacement Policy

• When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is selected by one of two methods:
  
  – **Random:**
    • Any block is randomly selected for replacement providing uniform allocation.
    • Simple to build in hardware.
    • The most widely used cache replacement strategy.
  
  – **Least-recently used (LRU):**
    • Accesses to blocks are recorded and and the block replaced is the one that was not used for the longest period of time.
    • LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated.
## Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

**Sample Data**

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Single Level Cache Performance

For a CPU with a single level (L1) of cache and no stalls for cache hits:

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time}$$

$$\text{Memory stall clock cycles} = (\text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty}) + (\text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty})$$

If write and read miss penalties are the same:

$$\text{Memory stall clock cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}$$
Single Level Cache Performance

CPUtime = IC x CPI x C

$CPI_{execution} = CPI$ with ideal memory

$CPI = CPI_{execution} + \text{Mem Stall cycles per instruction}$

CPUtime = IC x ($CPI_{execution} + \text{Mem Stall cycles per instruction}$) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Miss rate x Miss penalty

CPUtime = IC x ($CPI_{execution} + \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}$) x C

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x ($CPI_{execution} + \text{Misses per instruction} \times \text{Miss penalty}$) x C
Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- \( \text{CPI}_{\text{execution}} = 1.1 \)
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{mem stalls per instruction}
\]

\[
\text{Mem Stalls per instruction} = \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}
\]

\[
\text{Mem accesses per instruction} = 1 + .3 = 1.3
\]

\[
\text{Mem Stalls per instruction} = 1.3 \times .015 \times 50 = 0.975
\]

\[
\text{CPI} = 1.1 + .975 = 2.075
\]

The ideal CPU with no misses is \( 2.075/1.1 = 1.88 \) times faster
Cache Performance Example

- Suppose for the previous example we double the clock rate to 400 MHZ, how much faster is this machine, assuming similar miss rate, instruction mix?
- Since memory speed is not changed, the miss penalty takes more CPU cycles:
  
  Miss penalty = 50 x 2 = 100 cycles.
  CPI = 1.1 + 1.3 x 0.015 x 100 = 1.1 + 1.95 = 3.05
  
  Speedup = (CPI_{old} \times C_{old}) / (CPI_{new} \times C_{new})
  = 2.075 x 2 / 3.05 = 1.36

  The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

  → CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI
3 Levels of Cache

- **CPU**
  - **L1 Cache**
    - Hit Rate = $H_1$, Hit time = 1 cycle
  - **L2 Cache**
    - Hit Rate = $H_2$, Hit time = $T_2$ cycles
  - **L3 Cache**
    - Hit Rate = $H_3$, Hit time = $T_3$ cycles
- **Main Memory**
  - Memory access penalty, $M$
3-Level Cache Performance

\[ \text{CPU time} = \text{IC} \times (\text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times C \]

Mem Stall cycles per instruction = Mem accesses per instruction \times Stall cycles per access

• For a system with 3 levels of cache, assuming no penalty when found in L₁ cache:

\[ \text{Stall cycles per memory access} = \]

\[ [\text{miss rate L}_1] \times [\text{Hit rate L}_2 \times \text{Hit time L}_2 \]
\[ + \text{Miss rate L}_2 \times (\text{Hit rate L}_3 \times \text{Hit time L}_3 \]
\[ + \text{Miss rate L}_3 \times \text{Memory access penalty} ] = \]

\[ [1 - H_1] \times [H_2 \times T_2 \]
\[ + (1 - H_2) \times (H_3 \times (T_2 + T_3) \]
\[ + (1 - H_3) \times M ] \]
Three Level Cache Example

- CPU with $\text{CPI}_{\text{execution}} = 1.1$ running at clock rate = 500 MHZ
- 1.3 memory accesses per instruction.
- $L_1$ cache operates at 500 MHZ with a miss rate of 5%
- $L_2$ cache operates at 250 MHZ with miss rate 3%, $(T_2 = 2$ cycles$)$
- $L_3$ cache operates at 100 MHZ with miss rate 1.5%, $(T_3 = 5$ cycles$)$
- Memory access penalty, $M= 100$ cycles. Find CPI.
- With single $L_1$, $\text{CPI} = 1.1 + 1.3 \times .05 \times 100 = 7.6$

CPI = CPU execution $\text{CPI} + \text{Mem Stall cycles per instruction}$

Mem Stall cycles per instruction = Mem accesses per instruction $\times$ Stall cycles per access

Stall cycles per memory access = $[1 - H_1] \times [H_2 \times T_2 + (1-H_2) \times (H_3 \times (T_2 + T_3) + (1 - H_3) \times M)]$

$$= [.05] \times [ .97 \times 2 + (.03) \times (.985 \times (2+5) + .015 \times 100)]$$

$$= .05 \times [ 1.94 + .03 \times (6.895 + 1.5) ]$$

$$= .05 \times [ 1.94 + .274] = .11$$

- $\text{CPI} = 1.1 + 1.3 \times .11 = 1.24$
Memory Bandwidth Improvement Techniques

• **Wider Main Memory:**
  Memory width is increased to a number of words (usually the size of a second level cache block).
  ⇒ Memory bandwidth is proportional to memory width.
  e.g. Doubling the width of cache and memory doubles memory bandwidth

• **Simple Interleaved Memory:**
  Memory is organized as a number of banks each one word wide.
  – Simultaneous multiple word memory reads or writes are accomplished by sending memory addresses to several memory banks at once.
  – Interleaving factor: Refers to the mapping of memory addressees to memory banks.
    e.g. using 4 banks, bank 0 has all words whose address is:
    \[(\text{word address}) \mod 4 = 0\]
Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth

Simplest design: Everything is the width of one word

Wider memory, bus and cache

Narrow bus and cache with interleaved memory
Memory Interleaving

Access Pattern without Interleaving:
- D1 available
- Start Access for D1
- Start Access for D2

Access Pattern with 4-way Interleaving:
- Access Bank 0
- Access Bank 1
- Access Bank 2
- Access Bank 3

We can Access Bank 0 again
Memory Width, Interleaving: An Example

Given a base system with following parameters:

Cache Block size = 1 word, Memory bus width = 1 word, Miss rate = 3%
Miss penalty = 32 cycles, broken down as follows:
(4 cycles to send address, 24 cycles access time/word, 4 cycles to send a word)
Memory access/instruction = 1.2 Ideal execution CPI (ignoring cache misses) = 2
Miss rate (block size=2 word) = 2% Miss rate (block size=4 words) = 1%

- The CPI of the base machine with 1-word blocks = 2 + (1.2 x .03 x 32) = 3.15
- Increasing the block size to two words gives the following CPI:
  - 32-bit bus and memory, no interleaving = 2 + (1.2 x .02 x 2 x 32) = 3.54
  - 32-bit bus and memory, interleaved = 2 + (1.2 x .02 x (4 + 24 + 8) = 2.86
  - 64-bit bus and memory, no interleaving = 2 + (1.2 x .02 x 1 x 32) = 2.77
- Increasing the block size to four words; resulting CPI:
  - 32-bit bus and memory, no interleaving = 2 + (1.2 x 1% x 4 x 32) = 3.54
  - 32-bit bus and memory, interleaved = 2 + (1.2 x 1% x (4 +24 + 16) = 2.53
  - 64-bit bus and memory, no interleaving = 2 + (1.2 x 2% x 2 x 32) = 2.77
Virtual Memory

Benefits

- Illusion of having more physical main memory
- Allows program relocation
- Protection from illegal memory access
Two memory accesses needed:
First to page table
Second to item
Virtual Memory Issues/Strategies

- **Main memory block placement:** Fully associative placement is used to lower the miss rate.

- **Block replacement:** The least recently used (LRU) block is replaced when a new block is brought into main memory from disk.

- **Write strategy:** Write back is used and only those pages changed in main memory are written to disk (dirty bit scheme is used).

- To locate blocks in main memory a page table is utilized. The page table is indexed by the virtual page number and contains the physical address of the block.
  - In paging: Offset is concatenated to this physical page address.
  - In segmentation: Offset is added to the physical segment address.

- To limit the size of the page table to the number of physical pages in main memory a hashing scheme is used.

- Utilizing address locality, a translation look-aside buffer (TLB) is usually used to cache recent address translations and prevent a second memory access to read the page table.
Speeding Up Address Translation:
Translation Lookaside Buffer (TLB)

- TLB: A small on-chip fully-associative cache used for address translations.
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.

![Diagram of TLB and Address Translation]

Virtual Page Number

128-256 TLB Entries

Page Table (in main memory)

Physical Page Address

TLB (on-chip) 128-256 Entries

Physical Memory

Disk Storage
TLB & Cache Operation

Virtual address

TLB access

TLB miss use page table

No

TLB hit?

Yes

Physical address

Write?

No

Write access bit on?

Yes

Write protection bit on?

No

Write data into cache, update the tag, and put the data and the address into the write buffer

Yes

Write data into cache, update the tag, and put the data and the address into the write buffer

No

Cache hit?

Yes

Deliver data to the CPU

No

Cache miss stall

Cache is physically-addressed

Cache operation

Try to read data from cache