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

Application

Operating System

Compiler Firmware

Instr. Set Proc. I/O system

Datapath & Control

Digital Design

Circuit Design

Layout

Machine Language Program

Software/Hardware Boundary

Hardware

Logic Diagrams

Circuit Diagrams

Assembly Language Programs

Instruction Set Architecture

Microprogram

Register Transfer Notation (RTN)
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

The CPU time can be calculated as follows:

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

<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</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Architecture (ISA)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

The table indicates which factors affect each component of the CPU time calculation.
Aspects of CPU Execution Time

CPU Time = Instruction count x CPI x Clock cycle

Instruction Count

CPI

Clock Cycle

Depends on:
- Program Used
- Compiler
- ISA

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

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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>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:

\[
\begin{align*}
C_i & = \text{Count of instructions of type } i \\
CPI_i & = \text{Average cycles per instruction of type } i \\
F_i & = \text{Frequency of instruction type } i \\
    & = C_i / \text{total instruction count}
\end{align*}
\]

Then:

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

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

\[
CPI = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2
\]
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:

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

  Hence speedup is given by:

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

  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>

<table>
<thead>
<tr>
<th>Unchanged</th>
</tr>
</thead>
</table>

| Unaffected, fraction: (1 - F) | F/S |

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$\_2 = F\_2 = 15\%$
  - Speedup$\_3 = S\_3 = 30$  Percentage$\_3 = F\_3 = 10\%$

- 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?

  \[
  \text{Speedup} = \frac{1}{\left(1 - \sum_i F\_i + \sum_i \frac{F\_i}{S\_i}\right)}
  \]

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

## R-Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
</tr>
<tr>
<td>shamt</td>
<td>5</td>
</tr>
<tr>
<td>funct</td>
<td>6</td>
</tr>
</tbody>
</table>

## I-Type: ALU

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td>immediate</td>
<td>16</td>
</tr>
</tbody>
</table>

## I-Type: Load/Store, Branch

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
</tbody>
</table>

## J-Type: Jumps

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td>target address</td>
<td>26</td>
</tr>
</tbody>
</table>

- **op**: Opcode, 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.
Instruction Memory

Adr

Op Fun Rt Rs Rd Imm16 Jump_target

Control Unit

nPC_sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg Jump

Equal

DATA PATH
The Truth Table For The Main Control

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
</tr>
<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>
<tr>
<td>ALUop (Symbolic)</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<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>
Worst Case Timing (Load)

Clk

PC Old Value

Rs, Rt, Rd, Op, Func

Op, Func

ALUctr

ExtOp

ALUSrc

MemtoReg

RegWr

busA

busB

Address

busW

Instruction Memory Access Time

Delay through Control Logic

Register File Access Time

Delay through Extender & Mux

ALU Delay

Data Memory Access Time

Register Write Occurs

Delay through Control Logic
MIPS Single Cycle Instruction Timing Comparison

Arithmetic & Logical

| PC | Inst Memory | Reg File | mux | ALU | mux | setup |

Load

| PC | Inst Memory | Reg File | mux | ALU | Data Mem | mux | setup |

Branch

| PC | Inst Memory | Reg File | cmp | mux |
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.
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

- **Instruction Fetch**: Obtain instruction from program storage
- **Next Instruction**: Update program counter to address of next instruction
- **Instruction Decode**: Determine instruction type, Obtain operands from registers
- **Execute**: Compute result value or status
- **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
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>Instruction Fetch</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-Type</strong></td>
<td><strong>Immediate</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Logic</strong></td>
<td><strong>Load</strong></td>
<td><strong>Store</strong></td>
<td><strong>Branch</strong></td>
</tr>
<tr>
<td>Execution</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>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td>Mem[R] ← B</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</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).
Control Specification For Multi-cycle CPU
Finite State Machine (FSM)

IR ← MEM[PC]  "instruction fetch"

A ← R[rs]
B ← R[rt]  "decode / operand fetch"

R-type

R ← A fun B
R[rd] ← R
PC ← PC + 4

R ← A or ZX
R[rt] ← R
PC ← PC + 4

ORi

LW

R ← A + SX
M ← MEM[R]
R[rt] ← M
PC ← PC + 4

SW

R ← A + SX
MEM[R] ← B
PC ← PC + 4

BEQ & Equal

PC ← PC + 4

BEQ & ~Equal

PC ← PC + SX || 00

To instruction fetch

To instruction fetch

To instruction fetch

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 ← MEM[PC]
0000

“instruction fetch”

A ← R[rs]
0001
B ← R[rt]

“decode / operand fetch”

Aen, Ben

imem_rd, IRen

ALUfun, Sen

R-type

R ← A fun B
0100

R ← A or ZX
0110

ORi

LW

R ← A + SX
1000

1001

R ← A + SX
1010

BEQ & ~Equal

SW

MEM[S] ← B

PC ← PC + 4

1011

1001

PC ← PC + 4

0011

BEQ & Equal

0010

PC ← PC + SX ∥ 00

0010

To instruction fetch state 0000

To instruction fetch state 0000

To instruction fetch state 0000

Execute

Memory

Write-back

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben

imem_rd, IRen

Aen, Ben
## Detailed Control Specification

<table>
<thead>
<tr>
<th>State</th>
<th>Op field</th>
<th>Eq</th>
<th>Next</th>
<th>IR</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>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>0</td>
<td>0011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>1</td>
<td>0010</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>R-type</td>
<td>x</td>
<td>0100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>orI</td>
<td>x</td>
<td>0110</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>LW</td>
<td>x</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SW</td>
<td>x</td>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0101</td>
<td>1</td>
<td>0</td>
<td>0 1 fun 1</td>
<td></td>
<td></td>
<td>0 1 1</td>
</tr>
<tr>
<td>0101</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>0 0 or 1</td>
<td></td>
<td></td>
<td>0 1 0</td>
</tr>
<tr>
<td>0110</td>
<td>xxxxxx</td>
<td>x</td>
<td>0111</td>
<td>1</td>
<td>1</td>
<td>0 0 add 1</td>
<td></td>
<td></td>
<td>1 0 0</td>
</tr>
<tr>
<td>0111</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>0 0 add 1</td>
<td></td>
<td></td>
<td>1 1 0</td>
</tr>
<tr>
<td>1000</td>
<td>xxxxxx</td>
<td>x</td>
<td>1001</td>
<td>1</td>
<td>1</td>
<td>0 1 add 1</td>
<td></td>
<td></td>
<td>1 0 0</td>
</tr>
<tr>
<td>1001</td>
<td>xxxxxx</td>
<td>x</td>
<td>1010</td>
<td>1</td>
<td>1</td>
<td>0 1 add 1</td>
<td></td>
<td></td>
<td>1 1 0</td>
</tr>
<tr>
<td>1010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>0 1 add 1</td>
<td></td>
<td></td>
<td>1 1 0</td>
</tr>
<tr>
<td>1011</td>
<td>xxxxxx</td>
<td>x</td>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>0 1 add 1</td>
<td></td>
<td></td>
<td>1 1 0</td>
</tr>
<tr>
<td>1100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>0 1 add 1</td>
<td></td>
<td></td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
• 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>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 +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
</tr>
<tr>
<td></td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(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></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>M ← Mem[ALUout]</td>
<td>Mem[ALUout] ← B</td>
</tr>
</tbody>
</table>

**Logic Immediate**
- IR: Instruction Register
- PC: Program Counter
- Mem[PC]: Memory at PC
- R[rs], R[rt], R[rd]: Register file
- ALUout: ALU output
- A, B: Immediate
- SignExt(imm16): Sign extension of 16-bit immediate
- ZeroExt[imm16]: Zero extension of 16-bit immediate
- SignEx(Im16): Sign extension of 16-bit immediate
- Mem[ALUout]: Memory at ALU output
- ALUout: ALU output
Finite State Machine (FSM) Specification

IR ← MEM[PC]
PC ← PC + 4

“instruction fetch”

A ← R[rs]
B ← R[rt]

ALUout ← PC + SX

“decode”

ALUout ← A fun B
0100

ALUout ← A op ZX
0110

ALUout ← A + SX
1000

ALUout ← A + SX
1011

ALUout ← A + SX

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

If A = B then
PC ← ALUout
0010

Execute

Memory

Write-back

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&lt;sub&gt;i&lt;/sub&gt; for type</th>
<th>Frequency</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt; x freq&lt;sub&gt;i&lt;/sub&gt;</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>Sequencing Control</td>
<td>Explicit Next State Function</td>
<td>Microprogram counter + Dispatch ROMs</td>
</tr>
<tr>
<td>Logic Representation</td>
<td>Logic Equations</td>
<td>Truth Tables</td>
</tr>
<tr>
<td>Implementation Technique</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 microprogran 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
# 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>PCSource</td>
<td>PCSource = ALU</td>
<td>PCSource = 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>ALU</td>
<td>Read PC IR ALU</td>
<td>Seq Dispatch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td></td>
<td>Read ALU</td>
<td>Seq</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rt MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td></td>
<td>Write ALU</td>
<td>Seq</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtype:</td>
<td>Func</td>
<td>rs</td>
<td>rt</td>
<td>rd ALU</td>
<td>ALUoutCond.</td>
<td>Seq</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>Seq</td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td></td>
<td>rt ALU</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Microprogram for The Control Unit**

- **Fetch:**
  - Add PC 4 ALU
  - Add PC Extshft

- **Lw:**
  - Add rs Extend

- **Sw:**
  - Add rs Extend

- **Rtype:**
  - Func rs rt rd ALU

- **Beq:**
  - Subt. rs rt ALUoutCond.

- **Ori:**
  - Or rs Extend0 rt ALU
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>Code</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

- A
- B
- invertB
- CarryIn
- Operation
- and
- or
- add
- 1-bit Full Adder
- CarryOut
- Result
- Mux
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

CarryIn0
A0
B0
Less
1-bit ALU
Result0

CarryIn1
A1
B1
Less = 0
1-bit ALU
Result1

CarryIn2
A2
B2
Less = 0
1-bit ALU
Result2

CarryIn3

CarryIn31
A31
B31
Less = 0
1-bit ALU
Result31

C

CarryOut0

CarryOut1

CarryOut2

CarryOut3

CarryOut30

CarryOut31

Zero

Overflow
Improving ALU Performance: Carry Look Ahead (CLA)

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

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

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

\[ G = A \text{ and } B \]

\[ P = A \text{ xor } B \]
Cascaded Carry Look-ahead
16-Bit Example

\[ C1 = G0 + C0 \cdot P0 \]
\[ C2 = G1 + G0 \cdot P1 + C0 \cdot P0 \cdot P1 \]
\[ C3 = G2 + G1 \cdot P2 + G0 \cdot P1 \cdot P2 + C0 \cdot P0 \cdot P1 \cdot P2 \]
\[ \text{Delay} = 2 + 2 + 1 = 5 \text{ gate delays} = 5T \]

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

• Paper and pencil example (unsigned):

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1001</td>
</tr>
</tbody>
</table>

+-----------------+-----------------+-----------------+------------------|
| 1000            | 0000            | 0000            |
| 1000            | 01001000        |

Product

- m bits x n bits = m + n bit product, m = 32, n = 32, 64 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.
Operation of Combinational Multiplier

- 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

- CombineMultiplier register and Product register:
  - 32-bit Multiplicand register.
  - 32-bit ALU.
  - 64-bit Product register, (0-bit Multiplier register).

```
<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit ALU</td>
<td></td>
</tr>
<tr>
<td>Product (= Multiplier )</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
```

Control:
- Write
- Shift Right
Multiply Algorithm
Version 3

Start

1. Test Product0

- Product0 = 1
- 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
Combinational Shifter from MUXes

Basic Building Block

A \hspace{0.5cm} B
\begin{array}{c}
\text{sel} \\
1 \\
0 \\
D
\end{array}

8-bit right shifter

A_7 \hspace{0.5cm} A_6 \hspace{0.5cm} A_5 \hspace{0.5cm} A_4 \hspace{0.5cm} A_3 \hspace{0.5cm} A_2 \hspace{0.5cm} A_1 \hspace{0.5cm} A_0

1 \hspace{0.5cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 0

R_7 \hspace{0.5cm} R_6 \hspace{0.5cm} R_5 \hspace{0.5cm} R_4 \hspace{0.5cm} R_3 \hspace{0.5cm} R_2 \hspace{0.5cm} R_1 \hspace{0.5cm} R_0

S_2 \hspace{0.5cm} S_1 \hspace{0.5cm} S_0

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

1001 \[\begin{array}{c|c}
&\text{Quotient} \\
\text{Divisor} & 1000 \\
1001010 & \text{Dividend} \\
-1000 & \\
10 & \\
101 & \\
1010 & \\
-1000 & \\
10 & \text{Remainder (or Modulo result)}\
\end{array}\]

- See how big a number can be subtracted, creating quotient bit on each step:
  
  Binary => \[1 \times \text{divisor} \text{ or } 0 \times \text{divisor}\]

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

- 3 versions of divide, successive refinement
DIVIDE HARDWARE Version 3

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

![Diagram of divide hardware version 3]

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

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.

Remainder >= 0

Test

Remainder < 0

Remainder

nth repetition?

Yes: n repetitions (n = 4 here)

No: < n repetitions

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 &lt; E &lt; 255</td>
<td>Actual exponent is: ( e = E - 127 )</td>
<td></td>
</tr>
</tbody>
</table>

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: 
\[
2^{-126} \times (1.0) \quad \text{to} \quad 2^{127} \times (2 - 2^{-23})
\]

Which is approximately: 
\[
1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}
\]
Representation of Floating Point Numbers in Single Precision *IEEE 754 Standard*

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

- **Sign**: \( \{0, 1\} \)
- **Exponent**: \( 0 < E < 255 \), actual exponent is: \( e = E - 127 \)
- **Mantissa**: sign + magnitude, normalized binary significand with a hidden integer bit: \( 1.M \)

Example:
- \( 0 = 0\underbrace{00000000}_8 0 \ldots 0 \)
- \(-1.5 = 1\underbrace{01111111}_8 10 \ldots 0 \)

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

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

Which is approximately:

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

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

\begin{itemize}
  \item \( 0 < E < 2047 \)
  \item Actual exponent is: \( e = E - 1023 \)
\end{itemize}

Example: \( 0 = 0.00000000000 \ldots 0 \) \quad \text{and} \quad \(-1.5 = 1.01111111111 \ldots 0 \)

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

\begin{itemize}
  \item \( 2^{-1022} \) \text{ (1.0) to } \( 2^{1023} \) \text{ (2 - 2^{-52})} \)
  \item Which is approximately:
    \( 2.23 \times 10^{308} \) \text{ to } \( 1.8 \times 10^{308} \)
\end{itemize}
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:

\[
\begin{array}{ccc}
1 & 10001010 & 00100101001001000000000000 \\
S & E & M \\
1 \text{ bit} & 8 \text{ bits} & 23 \text{ bits}
\end{array}
\]
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. Check if there is overflow or underflow:
   - Yes: Generate exception or return error
   - No: Proceed to next step if still normalized.

5. If still normalized:
   - Yes: Round the significand to the appropriate number of bits. If mantissa = 0, set exponent to 0.
   - No: Done
Floating Point Addition Hardware

- Sign
- Exponent
- Significand
- Sign
- Exponent
- Significand

Small ALU

Exponent difference

0 1

Compare exponents

Shift smaller number right

Add

Normalize

Round

Control

Shift right

Big ALU

Increment or decrement

Shift left or right

Rounding hardware

Sign Exponent Significand

EECC550 - Shaaban

Overflow or Underflow?

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

7. Still Normalized?
   - Yes
   - No
     - Done
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:

\[
\begin{align*}
1.xxxxx & \quad 1.xxxxx & \quad 1.xxxxx \\
+ 1.xxxxx & \quad 0.001xxxxx & \quad 0.01xxxxx \\
1x.xxxxxy & \quad 1.xxxxxyy & \quad 1x.xxxxxyyy \\
\end{align*}
\]

- **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:

\[ \begin{array}{c|c|c} 0 & 2 & 1.69 \\ \hline - & 0 & 0 7.85 \\ \end{array} = \frac{1.6900}{10^{2-bias}} \]

\[ \begin{array}{c|c|c} 0 & 2 & 1.61 \\ \hline - & 0 & 0 7.85 \\ \end{array} = \frac{-0.0785}{10^{2-bias}} \]

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.

**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

*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

\[
\begin{array}{c}
\text{d0} \cdot \text{d1} \text{d2} \text{d3} \ldots \text{dp-1} 0 0 0 \\
0 \cdot 0 0 \text{X} \ldots \text{X} \text{XX} \text{S} \\
\text{XX} \text{S}
\end{array}
\]

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

\[
\begin{array}{c}
\text{d0} \cdot \text{d1} \text{d2} \text{d3} \ldots \text{dp-1} 0 0 0 \\
0 \cdot 0 0 \text{X} \ldots \text{X} \text{XX} 0
\end{array}
\]

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: \( \text{CPI} = 1 \).
# 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>

<table>
<thead>
<tr>
<th>Clock cycle 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>Time in clock cycles</td>
<td>→</td>
<td></td>
<td></td>
<td></td>
<td></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

*First instruction, I Completed*

*Last instruction, I+4 completed*
Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

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

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

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

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

Multiple Cycle Implementation:
- **Load**
- **Store**
- **R-type**
- 2 ns

Pipeline Implementation:
- **Load**
- **Store**
- **R-type**
MIPS: A Pipelined Datapath

IF
Instruction Fetch

ID
Instruction Decode

EX
Execution

MEM
Memory

WB
Write Back

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execution
- MEM: Memory
- WB: Write Back

Processors are divided into pipeline stages:

1. Fetch
2. Decode
3. Execute
4. Memory Access
5. Write Back

- IF: Fetch instruction from memory
- ID: Decode instruction and fetch registers
- EX: Perform operation and write to ALU
- MEM: Load/store data
- WB: Write back register values

Datapath components:

- Instruction Memory
- Address
- PC
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB
- ALU
- 32-bit ALU
- 16-bit ALU
- Branch
- Load/Store
- Data Memory
- Address
- Write Register
- 32-bit
- Zero
- 16-bit
- Sign Extend
- Shift Left
- Add

Data and control signals flow through the pipeline stages.
Pipeline Control

- Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the data.
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 ns x ((40% + 20%) x 4 + 40% x 5) = 10 ns x 4.4 = 44 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 = Instruction time unpipelined
                          Instruction time pipelined
                          = 44 ns / 11 ns = 4 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

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.

<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>
</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)

Diagram showing pipeline stalls and register values.

Value of register $2$: 10
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>CC</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
</tr>
</thead>
</table>

Time (in clock cycles):

- CC 1: 10
- CC 2: 10
- CC 3: 10
- CC 4: 10
- CC 5: 10
- CC 6: -20
- CC 7: -20
- CC 8: -20
- CC 9: -20
- CC 10: -20
- CC 11: -20
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

Time (in clock cycles)

<table>
<thead>
<tr>
<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>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>
</tbody>
</table>

Value of register $2': 10 10 10 10 10/-20 -20 -20 -20 -20

Value of EX/MEM: X X X -20 X X X X X

Value of MEM/WB: X X X X -20 X X X X X

Program execution order (in instructions)

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

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

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $2, 20($1)</td>
<td>CC 1</td>
</tr>
<tr>
<td>and $4, $2, $5</td>
<td>CC 2</td>
</tr>
<tr>
<td>or $8, $2, $6</td>
<td>CC 3</td>
</tr>
<tr>
<td>add $9, $4, $2</td>
<td>CC 4</td>
</tr>
<tr>
<td>slt $1, $6, $7</td>
<td>CC 5</td>
</tr>
</tbody>
</table>

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 lw and the first sw resulting in a stall cycle.

- With forwarding we need to find only one independent instructions to place between them, swapping the lw instructions works:
  
  \[
  \begin{align*}
  \text{lw} & \quad $15, 0 (\$2) \\
  \text{lw} & \quad $16, 4 (\$2) \\
  \text{nop} & \\
  \text{nop} & \\
  \text{sw} & \quad $15, 0 (\$2) \\
  \text{sw} & \quad $16, 4 (\$2)
  \end{align*}
  \]

- Without forwarding we need three independent instructions to place between them, so in addition two nops are added.
  
  \[
  \begin{align*}
  \text{lw} & \quad $15, 0 (\$2) \\
  \text{lw} & \quad $16, 4 (\$2) \\
  \text{nop} & \\
  \text{nop} & \\
  \text{sw} & \quad $15, 0 (\$2) \\
  \text{sw} & \quad $16, 4 (\$2)
  \end{align*}
  \]
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>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>30%</td>
<td>of which 25% are followed immediately by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>an instruction using the loaded value</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td>20%</td>
<td>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?

- CPI = Ideal CPI + 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

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

μProc
60%/yr.

DRAM
7%/yr.
Processor-DRAM Performance Gap Impact: Example

- To illustrate the performance impact, assume a pipelined RISC CPU with CPI = 1 using non-ideal memory.
- Over a 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 MHZ</th>
<th>CPU cycle 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

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

- 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

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

- Optical Disk or Magnetic Tape
- Magnetic Disc
- Main Memory
- Cache
- Registers
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
Cache Organization Example

Fully associative: block 12 can go anywhere

Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

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.
Address Field Sizes

Physical Address Generated by CPU

Block Address
Tag
Index
Block Offset

Block offset size = log2(block size)

Index size = log2(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 Address = 12 bits
Tag = 12 bits
Block offset = 4 bits

All 128 tags must be checked in parallel by hardware to locate a data block.
Cache Example: Direct Mapped Case

Block Address = 12 bits
Tag = 5 bits  Index = 7 bits  Block offset = 4 bits

Main Memory

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

Main Memory

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

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:

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

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

If write and read miss penalties are the same:

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

$CPI_{\text{execution}} = \text{CPI with ideal memory}$

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

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

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

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

Misses per instruction = Memory accesses per instruction x Miss rate

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

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- CPI_{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.

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

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

\[
\text{Mem accesses per instruction} = 1 + 0.3 = 1.3
\]

\[
\text{Mem Stalls per instruction} = 1.3 \times 0.015 \times 50 = 0.975
\]

\[
\text{CPI} = 1.1 + 0.975 = 2.075
\]

The ideal CPU with no misses is \(2.075/1.1 = 1.88\) times faster
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 = $50 \times 2 = 100$ cycles.

CPI = $1.1 + 1.3 \times 0.015 \times 100 = 1.1 + 1.95 = 3.05$

Speedup = $(\text{CPI}_{\text{old}} \times C_{\text{old}}) / (\text{CPI}_{\text{new}} \times C_{\text{new}})$

= $2.075 \times 2 / 3.05 = 1.36$

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

$\rightarrow$ 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

CPU time = IC \times (CPI_{execution} + \text{Mem Stall cycles per instruction}) \times C

Mem Stall cycles per instruction = \text{Mem accesses per instruction} \times \text{Stall cycles per access}

• For a system with 3 levels of cache, assuming no penalty when found in L_1 cache:

Stall cycles per memory access =

\[\text{miss rate } L_1 \times [ \text{Hit rate } L_2 \times \text{Hit time } L_2\]
\[\quad + \text{Miss rate } L_2 \times (\text{Hit rate } L_3 \times \text{Hit time } L_3\]
\[\quad + \text{Miss rate } L_3 \times \text{Memory access penalty} ] =\]

\[\left[1 - H_1 \right] \times \left[ H_2 \times T_2 \right.\]
\[\quad + (1-H_2) \times (H_3 \times (T_2 + T_3)\]
\[\quad + (1 - H_3) \times M \left.\right]\]
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%, \( (T_2 = 2 \text{ cycles}) \)
- \( L_3 \) cache operates at 100 MHZ with miss rate 1.5%, \( (T_3 = 5 \text{ cycles}) \)
- Memory access penalty, \( M = 100 \text{ cycles} \). Find CPI.

- With single \( L_1 \), \( \text{CPI} = 1.1 + 1.3 \times .05 \times 100 = 7.6 \)

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}
\]

\[
\text{Mem Stall cycles per instruction} = \text{Mem accesses per instruction} \times \text{Stall cycles per access}
\]

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

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

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

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

- \( \text{CPI} = 1.1 + 1.3 \times .11 = 1.24 \)