The Von-Neumann Computer Model

- Partitioning of the computing engine into components:
  - Central Processing Unit (CPU): Control Unit (instruction decode, sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
  - Memory: Instruction and operand storage.
  - Input/Output (I/O).
  - The stored program concept: Instructions from an instruction set are fetched from a common memory and executed one at a time.
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 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>
</tr>
<tr>
<td>2</td>
<td>Logic</td>
<td>Registers, ALU’s ...</td>
<td>Gates, FF’s …</td>
<td>Logic Diagrams</td>
</tr>
<tr>
<td>3</td>
<td>Organization</td>
<td>Processors, Memories</td>
<td>Registers, ALU’s …</td>
<td>Register Transfer Notation (RTN)</td>
</tr>
<tr>
<td>4</td>
<td>Microprogramming</td>
<td>Assembly Language</td>
<td>Microinstructions</td>
<td>Microprogram</td>
</tr>
<tr>
<td>5</td>
<td>Assembly language</td>
<td>OS Routines</td>
<td>Assembly language</td>
<td>Assembly Language Programs</td>
</tr>
<tr>
<td>6</td>
<td>Procedural</td>
<td>Applications, Drivers ..</td>
<td>OS Routines</td>
<td>High-level Language Programs</td>
</tr>
<tr>
<td>7</td>
<td>Application</td>
<td>Systems</td>
<td>Procedural Constructs</td>
<td>Problem-Oriented Programs</td>
</tr>
</tbody>
</table>

- **Low Level - Hardware**
- **Firmware**
- **High Level - Software**
Instruction Set Architecture (ISA)

“... the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.”
– Amdahl, Blaaw, and Brooks, 1964.

The instruction set architecture is concerned with:

- Organization of programmable storage (memory & registers): Includes the amount of addressable memory and number of available registers.
- Data Types & Data Structures: Encodings & representations.
- Instruction Set: What operations are specified.
- Instruction formats and encoding.
- Modes of addressing and accessing data items and instructions
- Exceptional conditions.
Instruction Set Architecture (ISA) Specification Requirements

- Instruction Format or Encoding:
  - How is it decoded?
- Location of operands and result (addressing modes):
  - Where other than memory?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
- Data type and Size.
- Operations
  - What are supported
- Successor instruction:
  - Jumps, conditions, branches.
- Fetch-decode-execute is implicit.
Types of Instruction Set Architectures
According To Operand Addressing Fields

Memory-To-Memory Machines:
- Operands obtained from memory and results stored back in memory by any instruction that requires operands.
- No local CPU registers are used in the CPU datapath.
- Include:
  • The 4 Address Machine.
  • The 3-address Machine.
  • The 2-address Machine.

The 1-address (Accumulator) Machine:
- A single local CPU special-purpose register (accumulator) is used as the source of one operand and as the result destination.

The 0-address or Stack Machine:
- A push-down stack is used in the CPU.

General Purpose Register (GPR) Machines:
- The CPU datapath contains several local general-purpose registers which can be used as operand sources and as result destinations.
- A large number of possible addressing modes.
- Load-Store or Register-To-Register Machines: GPR machines where only data movement instructions (loads, stores) can obtain operands from memory and store results to memory.
Types of Instruction Set Architectures
Memory-To-Memory Machines: The 4-Address Machine

- No program counter (PC) or other CPU registers are used.
- Instructions specify:
  - Location of first operand.
  - Location of second operand.
  - Place to store the result.
  - Location of next instruction.

\[
\begin{array}{c}
\text{Op1Addr: Op1} \\
\text{Op2Addr: Op2} \\
\text{ResAddr: Res} \\
\text{NextiAddr: Nexti}
\end{array}
\]

### Instruction:

```
add Res, Op1, Op2, Nexti
```

### Meaning:

\[
(\text{Res} \leftarrow \text{Op1} + \text{Op2})
\]

### Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ResAddr</th>
<th>Op1Addr</th>
<th>Op2Addr</th>
<th>NextiAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
</tbody>
</table>

 Opcode
 Which operation
 Where to put result
 Where to find operands
 Where to find next instruction
Types of Instruction Set Architectures

Memory-To-Memory Machines: The 3-Address Machine

- A program counter is included within the CPU which points to the next instruction.
- No CPU storage (general-purpose registers).

### Instruction Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bits:</th>
<th>Opcode</th>
<th>ResAddr</th>
<th>Op1Addr</th>
<th>Op2Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
<td>8</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

**Meaning:**

\[(\text{Res} \leftarrow \text{Op1} + \text{Op2})\]
Types of Instruction Set Architectures
Memory-To-Memory Machines: The 2-Address Machine

- The 2-address Machine: Result is stored in the memory address of one of the operands.

Instruction:
add Op2, Op1

Meaning:
(Op2 ← Op1 + Op2)

Instruction Format
Bits: 8 24 24

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op2Addr</th>
<th>Op1Addr</th>
</tr>
</thead>
</table>

Op1Addr: Op1
Op2Addr: Op2, Res
NextiAddr: Nexti

Where to find operands
Where to find next instruction
CPU

Program Counter (PC)
Types of Instruction Set Architectures
The 1-address (Accumulator) Machine

- A single accumulator in the CPU is used as the source of one operand and result destination.

Instruction:
add Op1

Meaning:
(Acc ← Acc + Op1)

Instruction Format

<table>
<thead>
<tr>
<th>Bits:</th>
<th>8</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>add</td>
<td>Op1Addr</td>
</tr>
</tbody>
</table>

Where to find operand2, and where to put result

Where to find next instruction

Program Counter (PC)
Types of Instruction Set Architectures

The 0-address (Stack) Machine

- A push-down stack is used in the CPU.

```
<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU</th>
<th>Instruction Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1Addr: Op1</td>
<td>Op1</td>
<td>push Op1</td>
</tr>
<tr>
<td>ResAddr: Res</td>
<td>etc.</td>
<td>add (TOS ← TOS + SOS)</td>
</tr>
<tr>
<td>NextiAddr: Nexti</td>
<td>Stack</td>
<td>pop Res</td>
</tr>
<tr>
<td></td>
<td>Program Counter (PC)</td>
<td>pop (Res ← TOS)</td>
</tr>
</tbody>
</table>
```
Types of Instruction Set Architectures
General Purpose Register (GPR) Machines

- CPU contains several general-purpose registers which can be used as operand sources and result destination.

![Diagram showing CPU, registers, and instruction format]

### Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Where to find operand1</th>
<th>Opcode</th>
<th>Where to find operands</th>
<th>Opcode</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Op1</td>
<td>add</td>
<td>R2, R4, R6</td>
<td>store</td>
<td>R2</td>
</tr>
<tr>
<td>Op1Addr</td>
<td></td>
<td>R2</td>
<td></td>
<td>R2</td>
<td></td>
</tr>
</tbody>
</table>

- Instruction Format:
  - **load R8, Op1**
    - Meaning: \( R8 \leftarrow Op1 \)
  - Instruction Format: 
    - Bits: 8 3 24
    - Opcode: load
    - Op1 Addr

- Instruction Format:
  - **add R2, R4, R6**
    - Meaning: \( R2 \leftarrow R4 + R6 \)
  - Instruction Format: 
    - Bits: 8 3 3 3
    - Opcode: add
    - R2, R4, R6

- Instruction Format:
  - **store R2, Op2**
    - Meaning: \( Op2 \leftarrow R2 \)
  - Instruction Format: 
    - Bits: 8 3 24
    - Opcode: store
    - R2, ResAddr
# Expression Evaluation Example with 3-, 2-, 1-, 0-Address, And GPR Machines

For the expression \[ A = (B + C) \times D - E \] where A-E are in memory

<table>
<thead>
<tr>
<th>3-Address</th>
<th>2-Address</th>
<th>1-Address Accumulator</th>
<th>0-Address Stack</th>
<th>GPR Register-Memory</th>
<th>Load-Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>add A, B, C</td>
<td>load A, B</td>
<td>load B</td>
<td>push B</td>
<td>load R1, B</td>
<td></td>
</tr>
<tr>
<td>mul A, A, D</td>
<td>add A, C</td>
<td>add C</td>
<td>push C</td>
<td>add R1, C</td>
<td></td>
</tr>
<tr>
<td>sub A, A, E</td>
<td>mul A, D</td>
<td>add</td>
<td>push D</td>
<td>add R1, D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub E</td>
<td>mul</td>
<td>push E</td>
<td>sub R1, E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store A</td>
<td>push</td>
<td>sub</td>
<td>store A, R1</td>
<td></td>
</tr>
</tbody>
</table>

3 instructions 4 instructions 5 instructions 8 instructions 5 instructions 8 instructions
Code size: 30 bytes 28 bytes 20 bytes 23 bytes Code size: about 22 bytes
Code size: 20 bytes Code size: about 29 bytes
Code size: 5 memory accesses 5 memory accesses 5 memory accesses 5 memory accesses
9 memory accesses 12 memory accesses 5 memory accesses 5 memory accesses

GPR Load-Store
## Typical ISA Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Sample Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add $R4, R3$</td>
<td>$R4 \leftarrow R4 + R3$</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add $R4, #3$</td>
<td>$R4 \leftarrow R4 + 3$</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add $R4, 10 (R1)$</td>
<td>$R4 \leftarrow R4 + \text{Mem}[10 + R1]$</td>
</tr>
<tr>
<td>Indirect</td>
<td>Add $R4, (R1)$</td>
<td>$R4 \leftarrow R4 + \text{Mem}[R1]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add $R3, (R1 + R2)$</td>
<td>$R3 \leftarrow R3 + \text{Mem}[R1 + R2]$</td>
</tr>
<tr>
<td>Absolute</td>
<td>Add $R1, (1001)$</td>
<td>$R1 \leftarrow R1 + \text{Mem}[1001]$</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add $R1, @ (R3)$</td>
<td>$R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]]$</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add $R1, (R2) +$</td>
<td>$R1 \leftarrow R1 + \text{Mem}[R2]$</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add $R1, - (R2)$</td>
<td>$R2 \leftarrow R2 - d$</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add $R1, 100 (R2) [R3]$</td>
<td>$R1 \leftarrow R1 + \text{Mem}[100 + R2 + R3*d]$</td>
</tr>
</tbody>
</table>
Three Examples of Instruction Set Encoding

Variable Length Encoding: VAX (1-53 bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

Fixed Length Encoding: DLX, MIPS, PowerPC, SPARC

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address Specifier</th>
<th>Address field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Address Specifier 1</td>
<td>Address Specifier 2</td>
</tr>
<tr>
<td>Operation</td>
<td>Address Specifier</td>
<td>Address field 1</td>
</tr>
</tbody>
</table>

Hybrid Encoding: IBM 360/370, Intel 80x86
Instruction Set Architecture Trade-offs

- 3-address machine: shortest code sequence; a large number of bits per instruction; large number of memory accesses.
- 0-address (stack) machine: Longest code sequence; shortest individual instructions; more complex to program.
- General purpose register machine (GPR):
  - Addressing modified by specifying among a small set of registers with using a short register address (all machines since 1975).
  - Advantages of GPR:
    - Low number of memory accesses. Faster, since register access is currently still much faster than memory access.
    - Registers are easier for compilers to use.
    - Shorter, simpler instructions.
- Load-Store Machines: GPR machines where memory addresses are only included in data movement instructions between memory and registers (all machines after 1980).
Complex Instruction Set Computer (CISC)

- Emphasizes doing more with each instruction.
- Motivated by the high cost of memory and hard disk capacity when original CISC architectures were proposed:
  - When M6800 was introduced: 16K RAM = $500, 40M hard disk = $55,000
  - When MC68000 was introduced: 64K RAM = $200, 10M HD = $5,000
- Original CISC architectures evolved with faster, more complex CPU designs, but backward instruction set compatibility had to be maintained.
- Wide variety of addressing modes:
  - 14 in MC68000, 25 in MC68020
- A number instruction modes for the location and number of operands:
  - The VAX has 0- through 3-address instructions.
- Variable-length or hybrid instruction encoding is used.
Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine.
- Reduced number of cycles needed per instruction.
  - Goal: At least one instruction completed per clock cycle.
- Designed with CPU instruction pipelining in mind.
- Fixed-length instruction encoding.
- Only load and store instructions access memory.
- Simplified addressing modes.
  - Usually limited to immediate, register indirect, register displacement, indexed.
- Delayed loads and branches.
- Prefetch and speculative execution.
- Examples: MIPS, HP-PA, UltraSpark, Alpha, PowerPC.
RISC ISA Example:

MIPS R3000

Instruction Categories: 4 Addressing Modes:
- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.
- Base register + immediate offset (loads and stores).
- Register direct (arithmetic).
- Immediate (jumps).
- PC relative (branches).

Operand Sizes:
- Memory accesses in any multiple between 1 and 4 bytes.

Instruction Encoding: 3 Instruction Formats, all 32 bits wide.

R-Type

I-Type: ALU Load/Store, Branch

J-Type: Jumps

Registers

| R0 - R31 |
| PC |
| HI |
| LO |

OP | rs | rt | rd | sa | funct |

OP | rs | rt | immediate |

OP | jump target |
MIPS Memory Addressing & Alignment

- MIPS uses Big Endian operand storage in memory where the most significant byte is in low memory (this is similar to IBM 360/370, Motorola 68k, Sparc, HP PA).

- MIPS requires that all words (32-bits) to start at memory addresses that are multiple of 4.

- In general objects must fall on memory addresses that are multiple of their size.
# MIPS Register Usage/Naming Conventions

- In addition to the usual naming of registers by $ followed with register number, registers are also named according to MIPS register usage convention as follows:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Name</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>Constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>Reserved for assembler</td>
<td>no</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-$v1</td>
<td>Values for result and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-$a3</td>
<td>Arguments</td>
<td>yes</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-$t7</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-$s7</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-$t9</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0-$k1</td>
<td>Reserved for operating system</td>
<td>yes</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

**Register (direct)**

- First Operand: \( \text{op rs rt rd} \)
- Destination: \( \text{register} \)

**Immediate**

- First Operand: \( \text{op rs rt immed} \)

**Displacement: Base+index**

- First Operand: \( \text{op rs rt immed} \)
- Second Operand: \( \text{register} \)
- Destination: \( \text{Memory} \)

**PC-relative**

- First Operand: \( \text{op rs rt immed} \)
- Second Operand: \( \text{PC} \)
- Destination: \( \text{Memory} \)
# MIPS Arithmetic Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
# MIPS Arithmetic Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>mutu$2,$3</td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 \div $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 \mod $3</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 \div $3,</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 \mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
### MIPS data transfer instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sw 500($4), $3</code></td>
<td>Store word</td>
</tr>
<tr>
<td><code>sh 502($2), $3</code></td>
<td>Store half</td>
</tr>
<tr>
<td><code>sb 41($3), $2</code></td>
<td>Store byte</td>
</tr>
<tr>
<td><code>lw $1, 30($2)</code></td>
<td>Load word</td>
</tr>
<tr>
<td><code>lh $1, 40($3)</code></td>
<td>Load halfword</td>
</tr>
<tr>
<td><code>lhu $1, 40($3)</code></td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td><code>lb $1, 40($3)</code></td>
<td>Load byte</td>
</tr>
<tr>
<td><code>lbu $1, 40($3)</code></td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td><code>lui $1, 40</code></td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

#### LUI Example

- **Instruction:** LUI R5
- **Comment:** Load Upper Immediate (16 bits shifted left by 16)

- **Format:**
  - **R5:** 0000 ...
  - **LUI:** 0000
# MIPS Branch, Compare, Jump Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Equal test; PC relative branch</em></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1!= $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Not equal test; PC relative branch</em></td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare less than; 2’s comp.</em></td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare &lt; constant; 2’s comp.</em></td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltu $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare less than; natural numbers</em></td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare &lt; constant; natural numbers</em></td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Jump to target address</em></td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>For switch, procedure return</em></td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>For procedure call</em></td>
</tr>
</tbody>
</table>
Example: C Assignment With Variable Index To MIPS

- For the C statement with a variable array index:
  \[ g = h + A[i]; \]
- Assume: \( g: \$s1, \ h: \$s2, \ i: \$s4, \) base address of \( A[]: \$s3 \)
- Steps:
  - Turn index \( i \) to a byte offset by multiplying by four or by addition as done here: \( i + i = 2i, \ 2i + 2i = 4i \)
  - Next add \( 4i \) to base address of \( A \)
  - Load \( A[i] \) into a temporary register.
  - Finally add to \( h \) and put sum in \( g \)
- MIPS Instructions:
  
  ```
  add $t1,$s4,$s4  # $t1 = 2*i
  add $t1,$t1,$t1  # $t1 = 4*i
  add $t1,$t1,$s3  #$t1 = address of A[i]
  lw $t0,0($t1)   # $t0 = A[i]
  add $s1,$s2,$t0  # g = h + A[i]
  ```
Example: While C Loop to MIPS

• While loop in C:
  ```c
  while (save[i]==k)  
    i = i + j;
  ```

• Assume MIPS register mapping:
  ```
i: $s3,   j: $s4,   k: $s5,   base of save[ ]: $s6
``` 

• MIPS Instructions:
  ```
  Loop:  add $t1,$s3,$s3  # $t1 = 2*i  
         add $t1,$t1,$t1  # $t1 = 4*i  
         add $t1,$t1,$s6  # $t1 = Address  
         lw  $t1,0($t1)  # $t1 = save[i]  
         bne $t1,$s5,Exit  # goto Exit  
          # if save[i]! =k  
         add $s3,$s3,$s4  # i = i + j  
         j   Loop  # goto Loop
  ```

Exit:
**MIPS R-Type (ALU) Instruction Fields**

**R-Type:** All ALU instructions that use three registers

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>

- **op:** Opcode, basic operation of the instruction.
  - For R-Type `op = 0`
- **rs:** The first register source operand.
- **rt:** The second register source operand.
- **rd:** The register destination operand.
- **shamt:** Shift amount used in constant shift operations.
- **funct:** Function, selects the specific variant of operation in the op field.

Examples:
- `add $1,$2,$3`
- `sub $1,$2,$3`
- `and $1,$2,$3`
- `or $1,$2,$3`
MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value Loads/stores, conditional branches.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- op: Opcode, operation of the instruction.
- rs: The register source operand.
- rt: The result destination register.
- immediate: Constant second operand for ALU instruction.

Examples:
- add immediate: `addi $1,$2,100`
- and immediate: `andi $1,$2,10`
## MIPS Load/Store I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
  - For load op = 35, for store op = 43.
- **rs**: The register containing memory base address.
- **rt**: For loads, the destination register. For stores, the source register of value to be stored.
- **address**: 16-bit memory address offset in bytes added to base register.

### Examples:
- **Store word**: `sw 500($4), $3`
- **Load word**: `lw $1, 30($2)`

```
Offset | base register in rs | source register in rt | Destination register in rt | Offset | base register in rs
```

---

EECC550 - Shaaban
MIPS Branch I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
- **rs**: The first register being compared
- **rt**: The second register being compared.
- **address**: 16-bit memory address branch target offset in words added to PC to form branch address.

Examples:
- Branch on equal: \texttt{beq $1,$2,100}
- Branch on not equal: \texttt{bne $1,$2,100}
MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
  - Jump j \( \text{op} = 2 \)
  - Jump and link jal \( \text{op} = 3 \)
- **jump target**: Jump memory address in words.

Jump memory address in bytes equal to instruction field \( \text{jump target} \times 4 \)

**Examples:**
- Branch on equal \( j \ 10000 \)
- Branch on not equal \( \text{jal} \ 10000 \)
Computer Performance Evaluation: Cycles Per Instruction (CPI)

- Most computers run synchronously utilizing a CPU clock running at a constant clock rate:
  
  \[ \text{Clock rate} = \frac{1}{\text{clock cycle}} \]

- A computer machine instruction is comprised of a number of elementary or micro operations which vary in number and complexity depending on the instruction and the exact CPU organization and implementation.
  - A micro operation is an elementary hardware operation that can be performed during one clock cycle.
  - This corresponds to one micro-instruction in microprogrammed CPUs.
  - Examples: register operations: shift, load, clear, increment, ALU operations: add, subtract, etc.

- Thus a single machine instruction may take one or more cycles to complete termed as the Cycles Per Instruction (CPI).
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?
Comparing Computer Performance Using Execution Time

- To compare the performance of two machines “A”, “B” running a given program:

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

- Machine A is \( n \) times faster than machine B means:

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

- Example:

  For a given program:

  \[
  \text{Execution time on machine A: } \text{Execution}_A = 1 \text{ second} \\
  \text{Execution time on machine B: } \text{Execution}_B = 10 \text{ seconds} \\
  \text{Performance}_A / \text{Performance}_B = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{10}{1} = 10
  \]

  The performance of machine A is 10 times the performance of machine B when running this program, or: Machine A is said to be 10 times faster than machine B when running this program.
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:

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

• A Program is running on a specific machine with the following parameters:
  – Total instruction count: 10,000,000 instructions
  – Average CPI for the program: 2.5 cycles/instruction.
  – CPU clock rate: 200 MHz.

• What is the execution time for this program:

  CPU time = Instruction count x CPI x Clock cycle
  
  = 10,000,000 x 2.5 x 1 / clock rate
  
  = 10,000,000 x 2.5 x 5x10^{-9} 
  
  = .125 seconds
Factors Affecting CPU Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</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></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

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

CPU Time = Instruction count x CPI x Clock cycle

Instruction Count

- Depends on:
  - Program Used
  - Compiler
  - ISA

CPI

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

Clock Cycle

- Depends on:
  - CPU Organization
  - Technology
Performance Comparison: Example

- From the previous example: A Program is running on a specific machine with the following parameters:
  - Total instruction count: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz.
- Using the same program with these changes:
  - A new compiler used: New instruction count 9,500,000
    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHZ
- What is the speedup with the changes?

\[
\text{Speedup} = \frac{\text{Old Execution Time}}{\text{New Execution Time}} = \frac{I_{\text{old}} \times \text{CPI}_{\text{old}} \times \text{Clock cycle}_{\text{old}}}{I_{\text{new}} \times \text{CPI}_{\text{new}} \times \text{Clock cycle}_{\text{new}}}
\]

\[
\text{Speedup} = \frac{(10,000,000 \times 2.5 \times 5 \times 10^{-9})}{(9,500,000 \times 3 \times 3.33 \times 10^{-9})}
\]

\[
= \frac{.125}{.095} = 1.32
\]

or 32% faster after changes.
Instruction Types & CPI

• Given a program with \( n \) types or classes of instructions with the following characteristics:

\[
C_i = \text{Count of instructions of type}_i
\]

\[
CPI_i = \text{Average cycles per instruction of type}_i
\]

Then:

\[
\text{CPU clock cycles} = \sum_{i=1}^{n} (CPI_i \times C_i)
\]
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:

\[
C_i = \text{Count of instructions of type } i
\]

\[
CPI_i = \text{Average cycles per instruction of type } i
\]

\[
F_i = \text{Frequency of instruction type } i
\]

\[
= \frac{C_i}{\text{total instruction count}}
\]

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.
Types of Benchmarks

Pros

• Representative
  Actual Target Workload

• Portable.
  • Widely used.
  • Measurements useful in reality.
  Full Application Benchmarks

• Easy to run, early in the design cycle.
  Small “Kernel” Benchmarks

• Identify peak performance and potential bottlenecks.
  Microbenchmarks

Cons

• Very specific.
• Non-portable.
• Complex: Difficult to run, or measure.

• Less representative than actual workload.

• Easy to “fool” by designing hardware to run them well.

• Peak performance results may be a long way from real application performance.
Computer Performance Measures: MIPS (Million Instructions Per Second)

- For a specific program running on a specific computer MIPS is a measure of how many millions of instructions are executed per second:

\[
\text{MIPS} = \frac{\text{Instruction count}}{(\text{Execution Time} \times 10^6)}
\]

\[
= \frac{\text{Instruction count}}{(\text{CPU clocks} \times \text{Cycle time} \times 10^6)}
\]

\[
= \frac{(\text{Instruction count} \times \text{Clock rate})}{(\text{Instruction count} \times \text{CPI} \times 10^6)}
\]

\[
= \frac{\text{Clock rate}}{(\text{CPI} \times 10^6)}
\]

- Faster execution time usually means faster MIPS rating.

- Problems with MIPS rating:
  - No account for the instruction set used.
  - Program-dependent: A single machine does not have a single MIPS rating since the MIPS rating may depend on the program used.
  - Easy to abuse: Program used to get the MIPS rating is often omitted.
  - Cannot be used to compare computers with different instruction sets.
  - A higher MIPS rating in some cases may not mean higher performance or better execution time. i.e. due to compiler design variations.
Computer Performance Measures:

**MFOLPS (Million FLOating-Point Operations Per Second)**

- A floating-point operation is an addition, subtraction, multiplication, or division operation applied to numbers represented by a single or a double precision floating-point representation.
- MFLOPS, for a specific program running on a specific computer, is a measure of millions of floating point-operation (megaflops) per second:

  \[
  \text{MFLOPS} = \frac{\text{Number of floating-point operations}}{(\text{Execution time} \times 10^6)}
  \]

- MFLOPS is a better comparison measure between different machines than MIPS.
- Program-dependent: Different programs have different percentages of floating-point operations present. i.e compilers have no floating-point operations and yield a MFLOPS rating of zero.
- Dependent on the type of floating-point operations present in the program.
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:

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

  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

After:
Execution Time with enhancement E:

\[
\text{Speedup}(E) = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{1 - F + \frac{F}{S}}
\]
Performance Enhancement Example

For the RISC machine with the following instruction mix given earlier:

<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>

CPI = 2.2

• If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:

Fraction enhanced = \( F = 45\% \) or \( .45 \)

Unaffected fraction = \( 100\% - 45\% = 55\% \) or \( .55 \)

Factor of enhancement = \( 5/2 = 2.5 \)

Using Amdahl’s Law:

\[
\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S}} = \frac{1}{.55 + \frac{.45}{2.5}} = 1.37
\]
An Alternative Solution Using CPU Equation

<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>

• If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:

Old CPI = 2.2
New CPI = \(0.5 \times 1 + 0.2 \times 2 + 0.1 \times 3 + 0.2 \times 2 = 1.6\)

\[
\text{Speedup}(E) = \frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count} \times \text{old CPI} \times \text{clock cycle}}{\text{Instruction count} \times \text{new CPI} \times \text{clock cycle}}
\]

\[
\text{Speedup}(E) = \frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37
\]

Which is the same speedup obtained from Amdahl’s Law in the first solution.
Performance Enhancement Example

- A program runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program four times faster?

\[
\text{Desired speedup} = 4 = \frac{100}{\text{Execution Time with enhancement}}
\]

\[
\rightarrow \text{Execution time with enhancement} = 25 \text{ seconds}
\]

\[
25 \text{ seconds} = (100 - 80 \text{ seconds}) + \frac{80 \text{ seconds}}{n}
\]

\[
25 \text{ seconds} = 20 \text{ seconds} + \frac{80 \text{ seconds}}{n}
\]

\[
\rightarrow 5 = \frac{80 \text{ seconds}}{n}
\]

\[
\rightarrow n = \frac{80}{5} = 16
\]

Hence multiplication should be 16 times faster to get a speedup of 4.
Extending Amdahl's Law To Multiple Enhancements

- Suppose that enhancement $E_i$ accelerates a fraction $F_i$ of the execution time by a factor $S_i$ and the remainder of the time is unaffected then:

$$\text{Speedup} = \frac{\text{Original Execution Time}}{\left( (1-\sum_i F_i) + \sum_i \frac{F_i}{S_i} \right) \times \text{Original Execution Time}}$$

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

Note: All fractions refer to original execution time.
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:
  
  \[
  \text{Speedup}_1 = S_1 = 10 \quad \text{Percentage}_1 = F_1 = 20\%
  \]
  
  \[
  \text{Speedup}_2 = S_2 = 15 \quad \text{Percentage}_2 = F_2 = 15\%
  \]
  
  \[
  \text{Speedup}_3 = S_3 = 30 \quad \text{Percentage}_3 = F_3 = 10\%
  \]

- 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?

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

- Speedup = \( \frac{1}{[(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30]} \)
  
  = \( \frac{1}{[.55 + .0333]} \)
  
  = \( \frac{1}{.5833} = 1.71 \)
Before:
Execution Time with no enhancements: 1

Unaffected, fraction: .55

F_1 = .2
F_2 = .15
F_3 = .1

Unchanged

S_1 = 10
S_2 = 15
S_3 = 30

Unaffected, fraction: .55

After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

Note: All fractions refer to original execution time.
Major CPU Design Steps

1. Using independent RTN, write the micro-operations required for all target ISA instructions.

2. Construct the datapath required by the micro-operations identified in step 1.

3. Identify and define the function of all control signals needed by the datapath.

3. Control unit design, based on micro-operation timing and control signals identified:
   - Hard-Wired: Finite-state machine implementation
   - Microprogrammed.
Datapath Design Steps

• Write the micro-operation sequences required for a number of representative instructions using independent RTN.

• From the above, create an initial datapath by determining possible destinations for each data source (i.e. registers, ALU).
  – This establishes the connectivity requirements (data paths, or connections) for datapath components.
  – Whenever multiple sources are connected to a single input, a multiplexer of appropriate size is added.

• Find the worst-time propagation delay in the datapath to determine the datapath clock cycle.

• Complete the micro-operation sequences for all remaining instructions adding connections/multiplexers as needed.
Single Cycle MIPS Datapath

Necessary multiplexors and control lines are identified here:
### Worst Case Timing (Load)

<table>
<thead>
<tr>
<th>Event</th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rs, Rt, Rd, Op, Func</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Memory Access Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUctr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtOp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUSrc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegWr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Delay through Control Logic**

**Register Write Occurs**

**Delay through Extender & Mux**

**Register File Access Time**

**ALU Delay**

**Data Memory Access Time**

**Worst Case Timing (Load)**
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.
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>Execution</strong></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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4 +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(SignExt(imm16) x4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td>Mem[R] ← B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4</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>
Control Specification For Multi-cycle CPU
Finite State Machine (FSM)

IR $\leftarrow$ MEM[PC]  \quad \text{“instruction fetch”}

A $\leftarrow$ R[rs]
B $\leftarrow$ R[rt]  \quad \text{“decode / operand fetch”}

- **R-type**
  - R $\leftarrow$ A fun B
  - R $\leftarrow$ A or ZX

- **ORi**
  - R $\leftarrow$ A + SX

- **LW**
  - M $\leftarrow$ MEM[R]

- **SW**
  - MEM[R] $\leftarrow$ B
  - PC $\leftarrow$ PC + 4

- **BEQ & Equal**
  - PC $\leftarrow$ PC + 4
  - PC $\leftarrow$ PC + SX || 00

**Execute**

- R[rd] $\leftarrow$ R
- PC $\leftarrow$ PC + 4

**Memory**

- R[rt] $\leftarrow$ R
- PC $\leftarrow$ PC + 4

**Write-back**

- R[rt] $\leftarrow$ M
- PC $\leftarrow$ PC + 4

- To instruction fetch
- To instruction fetch
- To instruction fetch
- To instruction fetch
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

EECC550 - Shaaban

#68 Midterm Review Winter 2001 1-22-2002
## 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></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>Mem[ALUout] ← B</td>
</tr>
</tbody>
</table>
Finite State Machine (FSM) Specification

IR ← MEM[PC]  
PC ← PC + 4  
0000

“instruction fetch”

A ← R[rs]  
B ← R[rt]

ALUout ← PC + SX  
0001

“decode”

R-type

ALUout ← A fun B  
0100

ALUout ← A op ZX  
0110

ALUout ← A + SX  
1000

ALUout ← A + SX  
1011

ORi

ALUout ← A + SX  
0110

MEM[ALUout] ← B  
1100

LW

MEM[ALUout] ← B  
1100

SW

MEM[ALUout] ← B  
1100

BEQ

If A = B then  
PC ← ALUout  
0010

To instruction fetch

Execute

ALUout ← A fun B  
0100

ALUout ← A op ZX  
0110

ALUout ← A + SX  
1000

ALUout ← A + SX  
1011

ALUout ← A + SX  
0110

M ← MEM[ALUout]  
1001

MEM[ALUout] ← B  
1100

R[rd] ← ALUout  
0101

R[rt] ← ALUout  
0111

R[rt] ← M  
1010

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).
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

Microinstruction Address

Inputs

Outputs

Control Signal Fields

Sequencing Control Field

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

Adder

EECC550 - Shaaban

#73 Midterm Review Winter 2001 1-22-2002
# 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>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td></td>
<td>Seq Dispatch</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read ALU</td>
<td></td>
<td>rt MEM</td>
<td></td>
<td>Seq Dispatch</td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Read ALU</td>
<td></td>
<td>rt MEM</td>
<td></td>
<td>Seq Fetch</td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Write ALU</td>
<td></td>
<td></td>
<td></td>
<td>Seq 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 Fetch</td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>ALUoutCond.</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td>rt ALU</td>
<td></td>
<td></td>
<td></td>
<td>Seq Fetch</td>
</tr>
</tbody>
</table>


Exceptions Handling in MIPS

- **Exceptions**: Events Other than branches or jumps that change the normal flow of instruction execution.

- **Two main types**: Interrupts, Traps.
  - An interrupt usually comes from outside the processor (I/O devices) to get the CPU’s attention to start a service routine.
  - A trap usually originates from an event within the CPU (Arithmetic overflow, undefined instruction) and initiates an exception handling routine usually by the operating system.

- The current MIPS implementation being considered can be extended to handle exceptions by adding two additional registers and the associated control lines:
  - **EPC**: A 32 bit register to hold the address of the affected instruction
  - **Cause**: A register used to record the cause of the exception.
    In this implementation only the low-order bit is used to encode the two handled exceptions: undefined instruction = 0
    overflow = 1

- Two additional states are added to the control finite state machine to handle these exceptions.
FSM Control Specification To Handle Exceptions

IR ← MEM[PC]
PC ← PC + 4
0000

“instruction fetch”

“decode”

undefined instruction

EPC ← PC - 4
PC ← exp_addr
cause ← 10 (Ovf)

ALUout ← PC +SX
0001

overflow

EPC ← PC - 4
PC ← exp_addr
cause ← 12 (Ovf)

ALUout ← A fun B
0100

ALUout ← A op ZX
0110

ALUout ← A + SX
1000

ALUout ← A + SX
1011

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

R[rd] ← ALUout
0101

R[rt] ← ALUout
0111

R[rt] ← M
1010

To instruction fetch

To instruction fetch

To instruction fetch

To instruction fetch

If A = B then
PC ← ALUout
0010

To instruction fetch

If A = B then
PC ← ALUout
0010

To instruction fetch

If A = B then
PC ← ALUout
0010

To instruction fetch

If A = B then
PC ← ALUout
0010

To instruction fetch

If A = B then
PC ← ALUout
0010

To instruction fetch