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).
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.
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

| 3-Address | 2-Address | 1-Address Accumulator | 0-Address Stack | GPR Register-Memory | Load-Store
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>load R1, B</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>load R2, C</td>
</tr>
<tr>
<td>sub A, A, E</td>
<td>mul A, D</td>
<td>mul D</td>
<td>add</td>
<td>mul R1, D</td>
<td>add R3, R1, R2</td>
</tr>
<tr>
<td></td>
<td>sub E</td>
<td>sub E</td>
<td>push D</td>
<td>sub R1, E</td>
<td>load R3, R3, R1</td>
</tr>
<tr>
<td></td>
<td>store A</td>
<td></td>
<td>push E</td>
<td>store A, R1</td>
<td>load R1, E</td>
</tr>
<tr>
<td>3 instructions</td>
<td>4 instructions</td>
<td>5 instructions</td>
<td>8 instructions</td>
<td>5 instructions</td>
<td>8 instructions</td>
</tr>
<tr>
<td>9 memory accesses</td>
<td>12 memory accesses</td>
<td>5 memory accesses</td>
<td>5 memory accesses</td>
<td>5 memory accesses</td>
<td>5 memory accesses</td>
</tr>
</tbody>
</table>
## 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 ← R4 + R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
<td>R4 ← R4 + 3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 10 (R1)</td>
<td>R4 ← R4 + Mem[10 + R1]</td>
</tr>
<tr>
<td>Indirect</td>
<td>Add R4, (R1)</td>
<td>R4 ← R4 + Mem[R1]</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3, (R1 + R2)</td>
<td>R3 ← R3 + Mem[R1 + R2]</td>
</tr>
<tr>
<td>Absolute</td>
<td>Add R1, (1001)</td>
<td>R1 ← R1 + Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1, @ (R3)</td>
<td>R1 ← R1 + Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1, (R2) +</td>
<td>R1 ← R1 + Mem[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 ← R2 + d</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, - (R2)</td>
<td>R1 ← R1 - Mem[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 ← R2 - d</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1, 100 (R2) [R3]</td>
<td>R1 ← R1 + Mem[100 + R2 + R3*d]</td>
</tr>
</tbody>
</table>

Add R4, R3
Add R4, #3
Add R4, 10 (R1)
Add R4, (R1)
Add R3, (R1 + R2)
Add R1, (1001)
Add R1, @ (R3)
Add R1, (R2) +
Add R1, (R2)
Add R1, 100 (R2) [R3]
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
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.

<table>
<thead>
<tr>
<th>R-Type</th>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Type: ALU Load/Store, Branch</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>J-Type: Jumps</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
<td></td>
</tr>
</tbody>
</table>

Registers

- R0 - R31
- PC
- HI
- LO
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)`

```
  op  rs  rt  rd
  +---+----+---+
    register
```

`Immediate`

```
  op  rs  rt  immed
```

`Displacement: Base+index`

```
  op  rs  rt  immed
```

`PC-relative`

```
  op  rs  rt  immed
```

Memory:

```
  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; <strong>exception possible</strong></td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; <strong>exception possible</strong></td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; <strong>exception possible</strong></td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; <strong>no exceptions</strong></td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; <strong>no exceptions</strong></td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; <strong>no exceptions</strong></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, Hi = $2 mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</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 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, Hi = $2 mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</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

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

```
LUI R5
0000 ... 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>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:
  \[
  \text{add } $t1,$s4,$s4 \# $t1 = 2*i \\
  \text{add } $t1,$t1,$t1 \# $t1 = 4*i \\
  \text{add } $t1,$t1,$s3 \# $t1 = address of A[i] \\
  \text{lw } $t0,0($t1) \# $t0 = A[i] \\
  \text{add } $s1,$s2,$t0 \# g = h + A[i]
  \]
Example: While C Loop to MIPS

• While loop in C:

\[
\text{while (save}[i] == k) \\
i = i + j;
\]

• Assume MIPS register mapping:

\[
i: \$s3, \quad j: \$s4, \quad k: \$s5, \quad \text{base of save[ ]}: \$s6
\]

• MIPS Instructions:

Loop:

\[
\text{add } \$t1, \$s3, \$s3 \quad \# \quad \$t1 = 2*i \\
\text{add } \$t1, \$t1, \$t1 \quad \# \quad \$t1 = 4*i \\
\text{add } \$t1, \$t1, \$s6 \quad \# \quad \$t1 = \text{Address} \\
\text{lw } \$t1, 0(\$t1) \quad \# \quad \$t1 = \text{save}[i] \\
\text{bne } \$t1, \$s5, \text{Exit} \quad \# \quad \text{goto Exit} \\
\text{add } \$s3, \$s3, \$s4 \quad \# \quad i = i + j \\
j \text{ Loop} \quad \# \quad \text{goto Loop}
\]

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:

- Destination register in rd
  - add $1,$2,$3
  - sub $1,$2,$3
- Operand register in rs
  - 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:
- Result register in rt
  - add immediate: `addi $1,$2,100`
  - and immediate: `andi $1,$2,10`
- Source operand register in rs
- Constant operand in immediate
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`
  - Offset
  - Base register in rs
  - Source register in rt

- **Load word**: `lw $1, 30($2)`
  - Destination register in rt
  - Offset
  - Base register in rs

EECC550 - Shaaban

#23  Midterm Review  Winter 2002  1-21-2003
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: `beq $1,$2,100`
- Branch on not equal: `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 \( \text{jal} \) \( \text{op} = 3 \)
- **jump target**: Jump memory address in words.

**Examples:**
- Branch on equal \( \text{j 10000} \)
- Branch on not equal \( \text{jal 10000} \)

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

**Jump target = 2500**

<table>
<thead>
<tr>
<th>4 bits</th>
<th>26 bits</th>
<th>2 bits</th>
</tr>
</thead>
</table>
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:

 Performance_A = 1 / Execution Time_A
 Performance_B = 1 / Execution Time_B

• Machine A is n times faster than machine B means:

 Speedup = n = Performance_A / Performance_B = Execution Time_B / Execution Time_A

• Example:

 For a given program:

 Execution time on machine A: Execution_A = 1 second
 Execution time on machine B: Execution_B = 10 seconds

 Performance_A / Performance_B = Execution Time_B / Execution Time_A
 = 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, \( I \)
  - Measured in: instructions/program

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

- CPU has a fixed clock cycle time \( C = 1/\text{clock rate} \)
  - Measured in: seconds/cycle

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

\[
T = I \times \text{CPI} \times C
\]
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:

\[
\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \frac{1}{\text{Clock cycle}}
\]

\[
= 10,000,000 \times 2.5 \times \frac{1}{200 \text{ MHz}}
\]

\[
= 10,000,000 \times 2.5 \times 5 \times 10^{-9}
\]

\[
= .125 \text{ seconds}
\]
## Factors Affecting CPU Performance

The CPU time can be calculated using the following formula:

\[ T = \frac{I \times CPI \times C}{Cycles per Instruction \times Clock Cycle Time} \]

<table>
<thead>
<tr>
<th>Factors</th>
<th>Instruction Count</th>
<th>Cycles per Instruction</th>
<th>Clock Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Instruction Set Architecture (ISA)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
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?

\[
\text{Speedup} = \frac{10,000,000 \times 2.5 \times 10^{-9}}{9,500,000 \times 3 \times 3.33 \times 10^{-9}} = \frac{.125}{.095} = 1.32
\]

or 32% faster after changes.
Instruction Types & CPI

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

\[
C_i = \text{Count of instructions of type } i \\
CPI_i = \text{Cycles per instruction for type } i
\]

Then:

\[
\text{CPI} = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count } I}
\]

Where:

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

\[
\text{Instruction Count } I = \sum C_i
\]
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>Instruction counts for instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

- CPU cycles for sequence 1 = \(2 \times 1 + 1 \times 2 + 2 \times 3 = 10\) cycles
  
  CPI for sequence 1 = \(\text{clock cycles} / \text{instruction count}\)
  
  = \(10 / 5 = 2\)

- CPU cycles for sequence 2 = \(4 \times 1 + 1 \times 2 + 1 \times 3 = 9\) cycles
  
  CPI for sequence 2 = \(9 / 6 = 1.5\)
Instruction Frequency & CPI

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

$$C_i = \text{Count of instructions of type}_i$$

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

$$F_i = \text{Frequency of instruction type}_i$$

$$= C_i / \text{total instruction count}$$

Then:

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

## Base Machine (Reg / Reg)

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq, $F_i$</th>
<th>CPI$_i$</th>
<th>CPI$_i$ x $F_i$</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 = \begin{align*}
.5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 &= 2.2
\end{align*}
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.
Compiler Variations, MIPS & Performance: An Example

• For a machine with 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>

• For a given program, two compilers produced the following instruction counts:

<table>
<thead>
<tr>
<th>Instruction counts (in millions) for each instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code from:</td>
</tr>
<tr>
<td>Compiler 1</td>
</tr>
<tr>
<td>Compiler 2</td>
</tr>
</tbody>
</table>

• The machine is assumed to run at a clock rate of 100 MHz.
Compiler Variations, MIPS & Performance: An Example (Continued)

MIPS = \frac{\text{Clock rate}}{(\text{CPI x } 10^6)} = \frac{100 \text{ MHz}}{(\text{CPI x } 10^6)}

\text{CPI = CPU execution cycles / Instructions count}

\text{CPU clock cycles} = \sum_{i=1}^{n} \left( \text{CPI}_i \times C_i \right)

\text{CPU time} = \text{Instruction count} \times \text{CPI} / \text{Clock rate}

• For compiler 1:
  – \text{CPI}_1 = \frac{(5 \times 1 + 1 \times 2 + 1 \times 3)}{(5 + 1 + 1)} = \frac{10}{7} = 1.43
  – \text{MIP}_1 = \frac{100}{(1.428 \times 10^6)} = 70.0
  – \text{CPU time}_1 = \frac{((5 + 1 + 1) \times 10^6 \times 1.43)}{(100 \times 10^6)} = 0.10 \text{ seconds}

• For compiler 2:
  – \text{CPI}_2 = \frac{(10 \times 1 + 1 \times 2 + 1 \times 3)}{(10 + 1 + 1)} = \frac{15}{12} = 1.25
  – \text{MIP}_2 = \frac{100}{(1.25 \times 10^6)} = 80.0
  – \text{CPU time}_2 = \frac{((10 + 1 + 1) \times 10^6 \times 1.25)}{(100 \times 10^6)} = 0.15 \text{ seconds}
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:

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

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:

\[
\begin{array}{c|c|c}
\text{Unaffected, fraction: (1- F)} & \text{Affected fraction: F} \\
\hline
\end{array}
\]

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>

\[ \text{CPI} = 2.2 \]

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

\[
\text{Fraction enhanced} = F = 45\% \text{ or } .45 \\
\text{Unaffected fraction} = 100\% - 45\% = 55\% \text{ or } .55 \\
\text{Factor of enhancement} = 5/2 = 2.5 \\
\text{Using Amdahl’s Law:}
\]

\[
\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S}} = \frac{1}{.55 + .45/2.5} = 1.37
\]
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>

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:**

Old CPI = 2.2

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

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

\[
\frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37
\]

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?

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

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

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

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

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

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

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:

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

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

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:

  Speedup\(_i\) = \(S_i\) = 10
  Percentage\(_i\) = \(F_i\) = 20%
  Speedup\(_i\) = \(S_i\) = 15
  Percentage\(_i\) = \(F_i\) = 15%
  Speedup\(_i\) = \(S_i\) = 30
  Percentage\(_i\) = \(F_i\) = 10%

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

- What is the resulting overall speedup?

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

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

Unaffected, fraction: .55

F1 = .2

S1 = 10

F2 = .15

S2 = 15

F3 = .1

S3 = 30

Unaffected, fraction: .55

 unchanged

/ 10

/ 15

/ 30

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 Extended To Handle Jump with Control Unit Added
Worst Case Timing (Load)

- **Clk**
  - **Clk-to-Q**

- **PC Old Value** -> **New Value**
  - Instruction Memoey Access Time

- **Rs, Rt, Rd, Op, Func**
  - **Old Value** -> **New Value**
  - **Delay through Control Logic**

- **ALUctr**
  - **Old Value** -> **New Value**

- **ExtOp**
  - **Old Value** -> **New Value**

- **ALUSrc**
  - **Old Value** -> **New Value**

- **MemtoReg**
  - **Old Value** -> **New Value**

- **RegWr**
  - **Old Value** -> **New Value**

- **busA**
  - **Old Value** -> **New Value**
  - **Delay through Extender & Mux**

- **busB**
  - **Old Value** -> **New Value**

- **busW**
  - **Old Value** -> **New Value**

- **Address**
  - **Old Value** -> **New Value**
  - **Data Memory Access Time**

- **Reg Write Occurs**
Simplified Single Cycle Datapath Timing

- Assuming the following datapath/control hardware components delays:
  - Memory Units: 2 ns
  - ALU and adders: 2 ns
  - Register File: 1 ns
  - Control Unit < 1 ns

- Ignoring Mux and clk-to-Q delays, critical path analysis:

```
<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Instruction Memory</td>
<td>2ns</td>
</tr>
<tr>
<td>PC + 4 ALU</td>
<td>3ns</td>
</tr>
<tr>
<td>Branch Target ALU</td>
<td>4ns</td>
</tr>
<tr>
<td>Main ALU</td>
<td>5ns</td>
</tr>
<tr>
<td>Data Memory</td>
<td>7ns</td>
</tr>
<tr>
<td>Register Write</td>
<td>8ns</td>
</tr>
</tbody>
</table>
```

Critical Path: 0 to 8ns
Performance of Single-Cycle CPU

• Assuming the following datapath hardware components delays:
  – Memory Units: 2 ns
  – ALU and adders: 2 ns
  – Register File: 1 ns

• The delays needed for each instruction type can be found:

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Memory</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Data Memory</th>
<th>Register Write</th>
<th>Total Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td>1 ns</td>
<td>6 ns</td>
</tr>
<tr>
<td>Load</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td>1 ns</td>
<td>8 ns</td>
</tr>
<tr>
<td>Store</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
<tr>
<td>Jump</td>
<td>2 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ns</td>
</tr>
</tbody>
</table>

• The clock cycle is determined by the instruction with longest delay: The load in this case which is 8 ns. Clock rate = 1 / 8 ns = 125 MHz

• A program with 1,000,000 instructions takes:
  Execution Time = T = I x CPI x C = 10^6 x 1 x 8x10^-9 = 0.008 s = 8 msec
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>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction Fetch</strong></td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
</tr>
<tr>
<td><strong>Instruction Decode</strong></td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
</tr>
<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>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td>If Equal = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4 +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(SignExt(imm16) x4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4</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>
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 ← A fun B  
R ← A or ZX  
R ← A + SX  
R ← A + SX  

M ← MEM[R]  
MEM[R] ← B  
PC ← PC + 4  
PC ← PC + 4

R[rd] ← R  
P C ← PC + 4  
R[rt] ← R  
P C ← PC + 4  
R[rt] ← M  
P C ← PC + 4

BEQ & Equal  
PC ← PC + 4  
PC ← PC + 4

To instruction fetch
```

---

EECC550 - Shaaban

#58 Midterm Review Winter 2002 1-21-2003
Alternative Multiple Cycle Datapath With Control Lines
(Fig 5.33 In Textbook)
# Operations In Each Cycle

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Instruction Decode</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></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>M ← Mem[ALUout]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mem[ALUout] ← B</td>
</tr>
</tbody>
</table>

**EECC550** - Shaaban
MIPS Multi-cycle Datapath

Performance Evaluation

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

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI_i for type</th>
<th>Frequency</th>
<th>CPI_i \times freq_i</th>
<th>CPI_i x freq_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td></td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td></td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td></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).