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
  - Application
    - Operating System
      - Compiler
      - Firmware
    - I/O system
    - Datapath & Control
      - Digital Design
        - Circuit Design
          - Layout
  - Instruction Set Architecture
  - Assembly Language Programs
  - Machine Language Program
  - Software/Hardware Boundary
  - Hardware
    - Logic Diagrams
    - Circuit Diagrams

- Software
  - Software/Hardware Boundary

- Hardware
  - Logic Diagrams
  - Circuit Diagrams
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.
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>3-Address</th>
<th>2-Address</th>
<th>1-Address Accumulator</th>
<th>0-Address Stack</th>
<th>GPR Register-Memory</th>
<th>GPR 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>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>load R2, C</td>
<td></td>
</tr>
<tr>
<td>sub A, A, E</td>
<td>mul A, D</td>
<td>add C</td>
<td>add D</td>
<td>mul R1, D</td>
<td>add R3, R1, R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub E</td>
<td></td>
<td>push D</td>
<td>sub R1, E</td>
<td>load R1, D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store A</td>
<td></td>
<td>push E</td>
<td>mul R3, R3, R1</td>
<td>sub R1, E</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sub</td>
<td>store A, R1</td>
<td>load R1, E</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pop A</td>
<td>store A, R3</td>
<td>sub R3, R3, R1</td>
<td></td>
</tr>
</tbody>
</table>

3 instructions
Code size: 30 bytes
9 memory accesses

4 instructions
Code size: 28 bytes
12 memory accesses

5 instructions
Code size: 20 bytes
5 memory accesses

8 instructions
Code size: 23 bytes
5 memory accesses

5 instructions
Code size: about 22 bytes
5 memory accesses

8 instructions
Code size: about 29 bytes
5 memory accesses
## 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>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>
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:**
- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.

**4 Addressing Modes:**
- 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></td>
<td>immediate</td>
</tr>
<tr>
<td>J-Type: Jumps</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump target</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
```

Immediate

```
op | rs | rt | immed
```

Displacement:
Base+index

```
op | rs | rt | immed
```

PC-relative

```
op | rs | rt | immed
```

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, 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,</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 data transfer instructions Examples

<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

R5 0000 ... 0000
### MIPS Branch, Compare, Jump Instructions Examples

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

- For the C statement with a variable array index:
  
  \[ g = h + A[i]; \]

- Assume: \( g: \$s1, \ h: \$s2, \ i: \$s4, \ \text{base address of } A[ ]: \$s3 \)

- 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:
  ```
  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)`
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 op = 2
  - Jump and link jal op = 3

- **jump target**: Jump memory address in words.
  - Jump memory address in bytes equal to instruction field: jump target x 4

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

<table>
<thead>
<tr>
<th>PC(31-28)</th>
<th>jump target = 2500</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>26 bits</td>
<td>2 bits</td>
<td></td>
</tr>
</tbody>
</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:

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

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

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

- Example:

  For a given program:

  Execution time on machine A: \(\text{Execution}_A = 1\) second
  Execution time on machine B: \(\text{Execution}_B = 10\) seconds

  \[
  \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{10}{1} = 10
  \]

  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, $I$
  - 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 $C = 1/$clock rate
  - Measured in: seconds/cycle

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

$$ T = I \times CPI \times C $$

<table>
<thead>
<tr>
<th>CPU time = Seconds Program</th>
<th>= Instructions Program x Cycles Instruction x Seconds Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$I \times CPI \times C$</td>
</tr>
</tbody>
</table>
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:

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

\[
= \frac{10,000,000 \times 2.5 \times 1}{200,000,000}
= \frac{10,000,000 \times 2.5 \times 5 \times 10^{-9}}{200,000,000}
= 0.125 \text{ seconds}
\]
## Factors Affecting CPU Performance

The formula for calculating CPU time is:

\[
T = \frac{I \times CPI \times C}{CPI \times C}
\]

Where:
- \( T \) = CPU time
- \( I \) = Instruction Count
- \( CPI \) = Cycles per Instruction
- \( C \) = Clock Cycle Time

### Factors Affecting CPU Performance

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

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

\[
\text{Speedup} = \frac{(10,000,000 \times 2.5 \times 5 \times 10^{-9})}{(9,500,000 \times 3 \times 3.33 \times 10^{-9})}
= \frac{.125}{.095} = 1.32
\]

or 32% faster after changes.
Instruction Types & CPI

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

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

Then:

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

Where:

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

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>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 = $2 \times 1 + 1 \times 2 + 2 \times 3 = 10$ cycles
  
  CPI for sequence 1 = clock cycles / 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:

\[
C_i = \text{Count of instructions of type}_i
\]
\[
CPI_i = \text{Average cycles per instruction of type}_i
\]
\[
F_i = \text{Frequency or fraction of instruction type}_i
= C_i / \text{total instruction count}
\]

Then:

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

Fraction of total execution time for instructions of type \( i \) = \( \frac{CPI_i \times F_i}{CPI} \)
### Instruction Type Frequency & CPI: A RISC Example

<table>
<thead>
<tr>
<th>Oper</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% = .5/2.2</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45% = 1/2.2</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14% = .3/2.2</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18% = .4/2.2</td>
</tr>
</tbody>
</table>

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

\[
CPI = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2
\]
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:
  \[
  \text{MIPS} = \frac{\text{Instruction count}}{(\text{Execution Time} \times 10^6)}
  \]
  \[
  = \frac{\text{Instruction count}}{\left(\text{CPU clocks} \times \text{Cycle time} \times 10^6\right)}
  \]
  \[
  = \frac{(\text{Instruction count} \times \text{Clock rate})}{(\text{Instruction count} \times \text{CPI} \times 10^6)}
  \]
  \[
  = \frac{\text{Clock rate}}{(\text{CPI} \times 10^6)}
  \]

- 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>Code from:</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler 1</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Compiler 2</td>
<td>10</td>
<td>1</td>
<td>1</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 = Clock rate / (CPI x 10^6) = 100 MHz / (CPI x 10^6)

CPI = CPU execution cycles / Instructions count

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

CPU time = Instruction count x CPI / Clock rate

- For compiler 1:
  - CPI_1 = (5 x 1 + 1 x 2 + 1 x 3) / (5 + 1 + 1) = 10 / 7 = 1.43
  - MIP_1 = 100 / (1.428 x 10^6) = 70.0
  - CPU time_1 = ((5 + 1 + 1) x 10^6 x 1.43) / (100 x 10^6) = 0.10 seconds

- For compiler 2:
  - CPI_2 = (10 x 1 + 1 x 2 + 1 x 3) / (10 + 1 + 1) = 15 / 12 = 1.25
  - MIP_2 = 100 / (1.25 x 10^6) = 80.0
  - CPU time_2 = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds
Computer Performance Measures:

MFLOPS (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:

Unaffected, fraction: (1 - F)

Unchanged

After:
Execution Time with enhancement E:

Affected fraction: F

Unaffected, fraction: (1 - F)

F/S

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:

$$\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S}} = \frac{1}{.55 + .45/2.5} = 1.37$$
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>
<th>CPI = 2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
<td>2.2</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
<td></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 = \( .5 \times 1 + .2 \times 2 + .1 \times 3 + .2 \times 2 = 1.6 \)

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

\[
= \frac{2.2}{1.6} = 1.37
\]

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?

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

→ Execution time with enhancement = 25 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:

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

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

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:

  Speedup_1 = S_1 = 10  \quad \text{Percentage}_1 = F_1 = 20\%
  
  Speedup_2 = S_2 = 15  \quad \text{Percentage}_1 = F_2 = 15\%
  
  Speedup_3 = S_3 = 30  \quad \text{Percentage}_1 = F_3 = 10\%

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

- What is the resulting overall speedup?

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

- Speedup = \frac{1}{[(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30]}
  
  = \frac{1}{[0.55 + 0.0333]}
  
  = 1 / 0.5833 = 1.71
Before:
Execution Time with no enhancements: 1

Unaffected, fraction: .55

<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>.2</td>
<td>.15</td>
<td>.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk-to-Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Memory Access Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rs, Rt, Rd, Op, Func</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>Delay through Control Logic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUctr</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>ExtOp</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>RegWr</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>Register File Access Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busA</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>Delay through Extender &amp; Mux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busB</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>ALU Delay</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
<tr>
<td>Data Memory Access Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busW</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>
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:
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>1 ns</td>
<td></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></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Fetch</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Decode</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B ← R[rt]</td>
<td></td>
<td></td>
<td></td>
<td>B ← R[rt]</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>
<td>If Equal = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← PC + 4 + (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>M ← Mem[R]</td>
<td>Mem[R] ← B</td>
</tr>
<tr>
<td></td>
<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>
<td></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  LW
R ← A + SX  SW
M ← MEM[R]  BEQ & Equal
MEM[R] ← B  PC ← PC + 4
PC ← PC + 4  BEQ & Equal
PC ← PC + 4  To instruction fetch

R[rd] ← R  R[rt] ← R  R[rt] ← M  To instruction fetch
PC ← PC + 4  PC ← PC + 4  PC ← PC + 4

To instruction fetch

To instruction fetch

To instruction fetch
Alternative Multiple Cycle Datapath With Control Lines
(Fig 5.33 In Textbook)
## Operations In Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Fetch</strong></td>
<td><strong>Decode</strong></td>
<td><strong>Execute</strong></td>
<td><strong>Memory</strong></td>
<td><strong>Write Back</strong></td>
</tr>
<tr>
<td><strong>PC</strong></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC +</td>
</tr>
<tr>
<td></td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
<td>(SignExt(imm16) x4)</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>ALUout ← A + B</td>
<td>ALUout ← A + ZeroExt[imm16]</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>If Equal = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC ← ALUout</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>M ← Mem[ALUout]</td>
<td>Mem[ALUout] ← B</td>
<td>Mem[ALUout] ← B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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\textsubscript{i} for type</th>
<th>Frequency</th>
<th>CPI\textsubscript{i} x freq\textsubscript{i}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Average CPI: 4.1

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