Midterm Questions Overview

Four questions from the following:

• Performance Evaluation:
  – Given MIPS code, estimate performance on a given CPU.
  – Compare performance of different CPU/compiler changes for a given program. May involve computing execution time, speedup, CPI, MIPS rating, etc.
  – Single or multiple enhancement Amdahl’s Law given parameters before or after the enhancements are applied.

• Adding support for a new instruction to the textbook versions of:
  – Single cycle MIPS CPU
  – Multi cycle MIPS CPU

Dependant RTN for the new instruction and changes to datapath/control.
CPU Organization (Design)

• **Datapath Design:** Components & their connections needed by ISA instructions
  - Capabilities & performance characteristics of principal Functional Units (FUs) needed by ISA instructions
  - (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:** Control/sequencing of operations of datapath components to realize ISA instructions
  - 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).**
Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine (ISA).
- 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.
- Load-Store: 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-RISC, SPARC, Alpha, PowerPC.
RISC ISA Example:

MIPS R3000 (32-bit)

- Memory: Can address $2^{32}$ bytes or $2^{30}$ words (32-bits).
- Instruction Categories:
  - Load/Store.
  - Computational: ALU.
  - Jump and Branch.
  - Floating Point.
    - Using coprocessor
  - Memory Management.
  - Special.

5 Addressing Modes:

- Register direct (arithmetic).
- Immediate (arithmetic).
- Base register + immediate offset (loads and stores).
- PC relative (branches).
- Pseudodirect (jumps)

Registrars

- R0 - R31
- 31 GPRs
- R0 = 0
- (each 32 bits)

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</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store, Branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-Type: Jumps</td>
<td>OP</td>
<td>jump target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Word = 4 bytes = 32 bits
### 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 Five Addressing Modes

1. **Register Addressing:**
   Where the operand is a register (R-Type)

2. **Immediate Addressing:**
   Where the operand is a constant in the instruction (I-Type, ALU)

3. **Base or Displacement Addressing:**
   Where the operand is at the memory location whose address is the sum of a register and a constant in the instruction (I-Type, load/store)

4. **PC-Relative Addressing:**
   Where the address is the sum of the PC and the 16-address field in the instruction shifted left 2 bits. (I-Type, branches)

5. **Pseudodirect Addressing:**
   Where the jump address is the 26-bit jump target from the instruction shifted left 2 bits concatenated with the 4 upper bits of the PC (J-Type)
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`

Rs, rt, rd are register specifier fields

Independent RTN:
- `R[rd] ← R[rs] funct R[rt]`
- `PC ← PC + 4`
# 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>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</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`

Independent RTN for addi:

- `R[rt] ← R[rs] + immediate`
- `PC ← PC + 4`
MIPS Load/Store I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address (e.g. offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
  - For load word op = 35, for store word 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 $3, 500($4)
  - source register in rt
  - Offset
  - base register in rs
  - Mem[R[rs] + address] ← R[rt]
  - PC ← PC + 4

- **Load word**: lw $1, 32($2)
  - Destination register in rt
  - Offset
  - base register in rs
  - R[rt] ← Mem[R[rs] + address]
  - PC ← PC + 4

Base or Displacement Addressing used (Mode 3)
MIPS Branch I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address (e.g. offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</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`

Independent RTN for `beq`:

\[
\begin{align*}
R[rs] &= R[rt] : \quad PC \leftarrow PC + 4 + \text{address x 4} \\
R[rs] &\neq R[rt] : \quad PC \leftarrow PC + 4
\end{align*}
\]
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>
<tr>
<td>[31:26]</td>
<td>[25:0]</td>
</tr>
</tbody>
</table>

Jump target in words

- **op**: Opcode, operation of the instruction.
  - **Jump j**: op = 2
  - **Jump and link jal**: op = 3

- **jump target**: Jump memory address in words.

**Examples:**

Jump

```
j 10000
```

Jump and link

```
jal 10000
```

Effective 32-bit jump address: PC(31-28), jump_target, 00

Jump memory address in bytes equal to instruction field jump target x 4

From PC(31-28)

PC(31-28) → jump target = 2500 → 0 0

4 bits 26 bits 2 bits

Independent RTN for j:

```
PC ← PC + 4
PC ← PC(31-28), jump_target, 00
```

J-Type = Jump Type Pseudodirect Addressing used (Mode 5)
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

1. Register (direct)
   R-Type
   
   First Operand | Second Operand | Destination
   --- | --- | ---
   op | rs | rt | rd

2. Immediate
   I-Type
   
   First Operand | Destination | Second Operand
   --- | --- | ---
   op | rs | rt | immed

3. Displacement: Base+index (load/store)
   
   First Operand | Destination | Second Operand
   --- | --- | ---
   op | rs | rt | immed

4. PC-relative (branches)
   
   First Operand | Destination
   --- | ---
   op | rs | rt | immed
   +
   PC

Shifted left 2 bits

Pseudodirect Addressing (Mode 5) not shown here, illustrated in the last slide for J-Type
# MIPS Arithmetic Instructions Examples

## Integer

<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,</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 Logic/Shift Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~( $2</td>
<td>$3 )</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = ~$2 &amp;~10</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sllv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
### MIPS Data Transfer Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw $3, 500($4)</td>
<td>Store word</td>
</tr>
<tr>
<td>sh $3, 502($2)</td>
<td>Store half</td>
</tr>
<tr>
<td>sb $2, 41($3)</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>

![Instruction Diagram](image)
### 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>Equal test; PC relative branch</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>Not equal test; PC relative branch</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>Compare less than; 2’s comp.</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>Compare &lt; constant; 2’s comp.</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>Compare less than; natural numbers</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>Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jump to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For switch, procedure return</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>For procedure call</td>
</tr>
</tbody>
</table>
Example: Simple C Loop to MIPS

• Simple loop in C:

```
Loop:    g = g + A[i];
i = i + j;
if (i != h) goto Loop;
```

• Assume MIPS register mapping:

```
g: $s1,   h: $s2,   i: $s3,   j: $s4,   base of A[]: $s5
```

• MIPS Instructions:

```
Loop:    add $t1,$s3,$s3    # $t1= 2*i
add $t1,$t1,$t1    # $t1= 4*i
add $t1,$t1,$s5    # $t1=address of A[I]
lw  $t1,0($t1)    # $t1= A[i]
add $s1,$s1,$t1    # g = g + A[i]
add $s3,$s3,$s4    # I = i + j
bne $s3,$s2,Loop    # goto Loop if i!=h
```

Word = 4 bytes
CPU Performance Evaluation: Cycles Per Instruction (CPI)

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

- The CPU clock rate depends on the specific CPU organization (design) and hardware implementation technology (VLSI) used.

- A computer machine (ISA) 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 (Design):
  
  - A micro operation is an elementary hardware operation that can be performed during one CPU 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 CPU cycles to complete termed as the Cycles Per Instruction (CPI).

- Average CPI of a program: The average CPI of all instructions executed in the program on a given CPU design.
Computer Performance Measures: Program Execution Time

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

• How can one measure the performance of this machine (CPU) running this program?
  – Intuitively the machine (or CPU) 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 (or CPUs) “A”, “B” running a given specific 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 (or slower? if \( n < 1 \)):

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

- Example:
  (i.e Speedup is ratio of performance, no units)

  For a given program:

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

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

  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.

The two CPUs may target different ISAs provided the program is written in a high level language (HLL).
CPU Execution Time: The CPU Equation

- A program is comprised of a number of instructions executed, \( I \)
  - Measured in: instructions/program

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

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

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

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

(This equation is commonly known as the CPU performance equation)
CPU Execution Time: Example

- A Program is running on a specific machine (CPU) with the following parameters:
  - Total executed instruction count: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz. (clock cycle = 5x10^{-9} seconds)

- What is the execution time for this program:

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

\[
= \frac{10,000,000 \times 2.5 \times 1}{200 \times 10^{-9}}
\]

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

\[
= 0.125 \text{ seconds}
\]

\[
T = I \times \text{CPI} \times C
\]
## Factors Affecting CPU Performance

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

\[
T = I \times CPI \times C
\]

- **CPU time** = Seconds
- **Program** = Instructions
- **Instruction** = Cycles
- **Cycle** = Seconds

\[
T = \frac{Program}{Instruction} \times \frac{Program}{Instruction} \times \frac{Instruction}{Cycles} \times \frac{Cycles}{Cycle} \times \frac{Seconds}{Cycle}
\]

- **Factors** affecting CPU performance include:
  - Program
  - Compiler
  - Instruction Set Architecture (ISA)
  - Organization (CPU Design)
  - Technology (VLSI)
Performance Comparison: Example

• From the previous example: A Program is running on a specific machine (CPU) with the following parameters:
  – Total executed instruction count, I: 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 executed instruction count, I: 9,500,000
    New CPI: 3.0
  – Faster CPU implementation: New clock rate = 300 MHz

• What is the speedup with the changes?

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

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

or 32 % faster after changes.
Instruction Types & CPI

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

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

\[
CPI_i = \text{Cycles per instruction for type}_i \quad i = 1, 2, \ldots, n
\]

Then:

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

Where:

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

\[
\text{Executed Instruction Count} \ I = \sum_C C_i
\]

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

For a specific CPU design

• 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

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

\[
\text{CPI} = \frac{\text{CPU Cycles}}{I}
\]
Instruction Frequency & CPI

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

\[
C_i = \text{Count of instructions of type}_i \quad i = 1, 2, \ldots, n
\]

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

\[
F_i = \text{Frequency or fraction of instruction type}_i \text{ executed}
\]

\[
= C_i/ \text{total executed instruction count} = C_i/ I
\]

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

Program Profile or Executed Instructions Mix

<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% $= .5/2.2$</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>

Typical Mix

$$\text{CPI} = \sum_{i=1}^{n} \left( \text{CPI}_i \times F_i \right)$$

$$\text{CPI} = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2$$

$$= .5 + 1 + .3 + .4$$

Sum = 2.2
Computer Performance Measures: MIPS (Million Instructions Per Second) Rating

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

  \[
  \text{MIPS Rating} = \frac{\text{Instruction count}}{(\text{Execution Time} \times 10^6)} = \frac{\text{Instruction count}}{(\text{CPU clocks} \times \text{Cycle time} \times 10^6)} = \frac{(\text{Instruction count} \times \text{Clock rate})}{(\text{Instruction count} \times \text{CPI} \times 10^6)} = \frac{\text{Clock rate}}{(\text{CPI} \times 10^6)}
  \]

- Major problem with MIPS rating: As shown above the MIPS rating does not account for the count of instructions executed (I).
  - A higher MIPS rating in many cases may not mean higher performance or better execution time. i.e. due to compiler design variations.

- In addition the MIPS rating:
  - Does not account for the instruction set architecture (ISA) used.
    - Thus it cannot be used to compare computers/CPUs with different instruction sets.
  - Easy to abuse: Program used to get the MIPS rating is often omitted.
    - Often the Peak MIPS rating is provided for a given CPU which is obtained using a program comprised entirely of instructions with the lowest CPI for the given CPU design which does not represent real programs.
Computer Performance Measures:
MIPS (Million Instructions Per Second) Rating

• Under what conditions can the MIPS rating be used to compare performance of different CPUs?

• The MIPS rating is only valid to compare the performance of different CPUs provided that the following conditions are satisfied:

  1. The same program is used
     (actually this applies to all performance metrics)

  2. The same ISA is used

  3. The same compiler is used

⇒ (Thus the resulting programs used to run on the CPUs and obtain the MIPS rating are identical at the machine code level including the same instruction count)
Compiler Variations, MIPS & Performance: An Example

- For a machine with instruction classes:

  Instruction class | CPI
  -----------------|---
  A                | 1
  B                | 2
  C                | 3

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

<table>
<thead>
<tr>
<th>Instruction counts (in millions) for each instruction class</th>
<th>Code from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Compiler 1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Compiler 2</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></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)

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

\[
\text{CPI} = \frac{\text{CPU execution cycles}}{\text{Instructions count}}
\]

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

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

- For compiler 1:
  - \(\text{CPI}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) / (5 + 1 + 1) = 10 / 7 = 1.43\)
  - \(\text{MIPS Rating}_1 = 100 / (1.428 \times 10^6) = 70.0 \text{ MIPS}\)
  - \(\text{CPU time}_1 = (5 + 1 + 1) \times 10^6 \times 1.43) / (100 \times 10^6) = 0.10 \text{ seconds}\)

- For compiler 2:
  - \(\text{CPI}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) / (10 + 1 + 1) = 15 / 12 = 1.25\)
  - \(\text{MIPS Rating}_2 = 100 / (1.25 \times 10^6) = 80.0 \text{ MIPS}\)
  - \(\text{CPU time}_2 = (10 + 1 + 1) \times 10^6 \times 1.25) / (100 \times 10^6) = 0.15 \text{ seconds}\)

MIPS rating indicates that compiler 2 is better while in reality the code produced by compiler 1 is faster.
MIPS (The ISA not the metric) Loop Performance Example

For the loop:

\[
\text{for (i=0; i<1000; i=i+1)}
\]
\[
x[i] = x[i] + s;
\}

MIPS assembly code is given by:

\[
\begin{align*}
\text{lw} & \quad \$3, \quad 8(\$1) \quad ; \text{load } s \text{ in } \$3 \\
\text{addi} & \quad \$6, \quad \$2, \quad 4000 \quad ; \$6 = \text{address of last element} + 4 \\
\text{loop:} & \quad \text{lw} \quad \$4, \quad 0(\$2) \quad ; \text{load } x[i] \text{ in } \$4 \\
& \quad \text{add} \quad \$5, \quad \$4, \quad \$3 \quad ; \$5 \text{ has } x[i] + s \\
& \quad \text{sw} \quad \$5, \quad 0(\$2) \quad ; \text{store computed } x[i] \\
& \quad \text{addi} \quad \$2, \quad \$2, \quad 4 \quad ; \text{increment } \$2 \text{ to point to next } x[ ] \text{ element} \\
& \quad \text{bne} \quad \$6, \quad \$2, \quad \text{loop} \quad ; \text{last loop iteration reached?}
\end{align*}
\]

The MIPS code is executed on a specific CPU that runs at 500 MHz (clock cycle = 2ns = 2x10^{-9} seconds) with following instruction type CPIs:

- **ALU**: 4
- **Load**: 5
- **Store**: 7
- **Branch**: 3

For this MIPS code running on this CPU find:

1. Fraction of total instructions executed for each instruction type
2. Total number of CPU cycles
3. Average CPI
4. Fraction of total execution time for each instruction type
5. Execution time
6. MIPS rating, peak MIPS rating for this CPU

X[ ] array of words in memory, base address in $2, s a constant word value in memory, address in $1
MIPS (The ISA) Loop Performance Example (continued)

- The code has 2 instructions before the loop and 5 instructions in the body of the loop which iterates 1000 times,
- Thus: Total instructions executed, \( I = 5 \times 1000 + 2 = 5002 \) instructions

1. Number of instructions executed/fraction \( F_i \) for each instruction type:
   - ALU instructions = 1 + 2x1000 = 2001  \( \text{CPI}_{\text{ALU}} = 4 \)  \( \text{Fraction}_{\text{ALU}} = F_{\text{ALU}} = 2001/5002 = 0.4 = 40\% \)
   - Load instructions = 1 + 1x1000 = 1001  \( \text{CPI}_{\text{Load}} = 5 \)  \( \text{Fraction}_{\text{Load}} = F_{\text{Load}} = 1001/5002 = 0.2 = 20\% \)
   - Store instructions = 1000  \( \text{CPI}_{\text{Store}} = 7 \)  \( \text{Fraction}_{\text{Store}} = F_{\text{Store}} = 1000/5002 = 0.2 = 20\% \)
   - Branch instructions = 1000  \( \text{CPI}_{\text{Branch}} = 3 \)  \( \text{Fraction}_{\text{Branch}} = F_{\text{Branch}} = 1000/5002 = 0.2 = 20\% \)

2. CPU clock cycles = \( \sum_{i=1}^{n} (\text{CPI}_i \times C_i) \)
   \[ = 2001 \times 4 + 1001 \times 5 + 1000 \times 7 + 1000 \times 3 = 23009 \text{ cycles} \]

3. Average CPI = CPU clock cycles / \( I = 23009/5002 = 4.6 \)

4. Fraction of execution time for each instruction type:
   - Fraction of time for ALU instructions = \( \text{CPI}_{\text{ALU}} \times F_{\text{ALU}} / \text{CPI} = 4 \times 0.4/4.6 = 0.348 = 34.8\% \)
   - Fraction of time for load instructions = \( \text{CPI}_{\text{Load}} \times F_{\text{Load}} / \text{CPI} = 5 \times 0.2/4.6 = 0.217 = 21.7\% \)
   - Fraction of time for store instructions = \( \text{CPI}_{\text{Store}} \times F_{\text{Store}} / \text{CPI} = 7 \times 0.2/4.6 = 0.304 = 30.4\% \)
   - Fraction of time for branch instructions = \( \text{CPI}_{\text{Branch}} \times F_{\text{Branch}} / \text{CPI} = 3 \times 0.2/4.6 = 0.13 = 13\% \)

5. Execution time = \( I \times \text{CPI} \times C = \text{CPU cycles} \times C = 23009 \times 2 \times 10^{-9} = 4.6 \times 10^{-5} \text{ seconds} = 0.046 \text{ msec} = 46 \text{ usec} \)

6. MIPS rating = Clock rate / (\( \text{CPI} \times 10^6 \)) = 500 / 4.6 = 108.7 MIPS
   - The CPU achieves its peak MIPS rating when executing a program that only has instructions of the type with the lowest CPI. In this case branches with \( \text{CPI}_{\text{Branch}} = 3 \)
   - Peak MIPS rating = Clock rate / (\( \text{CPI}_{\text{Branch}} \times 10^6 \)) = 500/3 = 166.67 MIPS
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:

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

$F$ (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied.
Pictorial Depiction of Amdahl’s Law

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

Before:
Execution Time without enhancement $E$: (Before enhancement is applied)
  - shown normalized to $1 = (1-F) + F = 1$

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

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

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:

\[
\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S}} = \frac{1}{.55 + .45/2.5} = 1.37
\]
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\(E\) = \(\frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count} \times \text{old CPI} \times \text{clock cycle}}{\text{Instruction count} \times \text{new CPI} \times \text{clock cycle}}\)

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

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 = 100/4 = 25 seconds

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

→ \( S = \frac{80}{5} = 16 \)

Alternatively, it can also be solved by finding enhanced fraction of execution time:

\[
F = \frac{80}{100} = .8
\]

and then solving Amdahl’s speedup equation for desired enhancement factor \( S \)

\[
\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S}} = 4 = \frac{1}{(1 - .8) + \frac{.8}{S}} = \frac{1}{.2 + \frac{.8}{S}}
\]

Hence multiplication should be 16 times faster to get an overall 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:

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

Unaffected fraction

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

Note: All fractions refer to original execution time before the enhancements are applied.
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}_2 = F_2 = 15\%
  
  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?

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

- \[
\text{Speedup} = 1 / \left[\left(1 - .2 - .15 - .1\right) + \frac{.2}{10} + \frac{.15}{15} + \frac{.1}{30}\right]
\]

- \[
= 1 / \left[ .55 + .0333 \right]
\]

- \[
= 1 / .5833 = 1.71
\]
Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1

Unaffected, fraction: .55

F1 = .2
F2 = .15
F3 = .1

/ 10 / 15 / 30

Unchanged

Unaffected, fraction: .55

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

Speedup = 1 / .5833 = 1.71

What if the fractions given are after the enhancements were applied? How would you solve the problem?

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

1. **Analyze instruction set** to get datapath requirements:
   - Using independent RTN, write the micro-operations required for target ISA instructions.
     - This provides the the required datapath **components** and **how they are connected**.

2. **Select set of datapath components** and establish **clocking methodology** (defines when storage or state elements can read and when they can be written, e.g. clock edge-triggered)

3. **Assemble datapath** meeting the requirements.

4. **Identify and define the function of all control points or signals** needed by the datapath.
   - Analyze implementation of each instruction to determine setting of control points that affects its operations.

5. **Control unit design**, based on micro-operation timing and control signals identified:
   - Combinational logic: For single cycle CPU.  
     - e.g Any instruction completed in one cycle
   - Microprogrammed.
Datapath Design Steps

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

• Independent RTN statements specify: the required datapath components and how they are connected.

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

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

• Complete the micro-operation sequences for all remaining instructions adding datapath components + connections/multiplexors as needed.
Single Cycle MIPS Datapath Extended To Handle Jump with Control Unit Added

(Book figure has an error!)

(This is book version ORI not supported, no zero extend of immediate needed)

Figure 5.24 page 314
Control Lines Settings
(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>

Similar to Figure 5.18 page 308
with Jump instruction control line values included
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:

Obtained from low-level target VLSI implementation technology of components

\[ ns = \text{nanosecond} = 10^{-9} \text{ second} \]
Performance of Single-Cycle (CPI=1) 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 = \( \frac{1}{8 \text{ ns}} = 125 \text{ MHz} \)

- A program with \( I = 1,000,000 \) instructions executed takes:
  \[
  \text{Execution Time} = T = I \times CPI \times C = 10^6 \times 1 \times 8 \times 10^{-9} = 0.008 \text{ s} = 8 \text{ msec}
  \]
The MIPS jump and link instruction, `jal`, is used to support procedure calls by jumping to jump address (similar to `j`) and saving the address of the following instruction `PC+4` in register `$ra ($31)`.

```
jal Address
```

`jal` uses the `j` instruction format:

```
op (6 bits) | Target address (26 bits)
```

- We wish to add `jal` to the single cycle datapath in Figure 5.24 page 314. Add any necessary datapaths and control signals to the single-clock datapath and justify the need for the modifications, if any.
- Specify control line values for this instruction.
Exercise 5.20: jump and link, jal support to Single Cycle Datapath

Instruction Word ← Mem[PC]
R[31] ← PC + 4
PC ← Jump Address

1. Expand the multiplexor controlled by RegDst to include the value 31 as a new input 2.
2. Expand the multiplexor controlled by MemtoReg to have PC+4 as a new input 2.

(For More Practice Exercise 5.20)
Exercise 5.20: jump and link, jal support to Single Cycle Datapath

Adding Control Lines Settings for jal
(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>01</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>00</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>xx</td>
<td>1</td>
<td>xx</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>xx</td>
<td>0</td>
<td>xx</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>xx</td>
<td>x</td>
<td>xx</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>JAL</td>
<td>10</td>
<td>x</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>

RegDst is now 2 bits
MemtoReg is now 2 bits

R[31] = Mem[PC]
Pc + 4 = R[31]
PC ← Jump Address

Instruction Word ← Mem[PC]
R[31] ← PC + 4
PC ← Jump Address

(For More Practice Exercise 5.20)
Adding Support for LWR to Single Cycle Datapath
(For More Practice Exercise 5.22)

- We wish to add a variant of lw (load word) let’s call it LWR to the single cycle datapath in Figure 5.24 page 314.
  
  \[ \text{LWR} \quad \text{\$rd, \$rs, \$rt} \]

- The LWR instruction is similar to lw but it sums two registers (specified by \$rs, \$rt) to obtain the effective load address and uses the R-Type format.

- Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.

- Specify control line values for this instruction.

(For More Practice Exercise 5.22)
Exercise 5.22:  LWR (R-format LW) support to Single Cycle Datapath

Instruction Word ← Mem[PC]
PC ← PC + 4

No new components or connections are needed for the datapath just the proper control line settings

Adding Control Lines Settings for LWR
(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>LWR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(For More Practice Exercise 5.22)
Adding Support for jm to Single Cycle Datapath

(Based on “For More Practice Exercise 5.44” but for single cycle)

• We wish to add a new instruction jm (jump memory) to the single cycle datapath in Figure 5.24 page 314.

  jm offset($rs)

• The jm instruction loads a word from effective address ($rs + offset), this is similar to lw except the loaded word is put in the PC instead of register $rt.

• Jm used the I-format with field rt not used.

<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>

  Not Used

• Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.

• Specify control line values for this instruction.
Adding jump memory, jm support to Single Cycle Datapath

Instruction Word ← Mem[PC]
PC ← Mem[R[rs] + SignExt[imm16]]

1. Expand the multiplexor controlled by Jump to include the Read Data (data memory output) as new input
2. The Jump control signal is now 2 bits

(Based on "For More Practice Exercise 5.44" but for single cycle)
Adding jm support to Single Cycle Datapath

Adding Control Lines Settings for jm
(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>J</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>01</td>
</tr>
<tr>
<td>JAL</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Jump is now 2 bits

PC ← Mem[R[rs] + SignExt[imm16]]

Adding control lines settings for jm
(For Textbook Single Cycle Datapath including Jump)
Reducing Cycle Time: Multi-Cycle Design

- Cut combinational dependency graph by inserting registers / latches.
- The same work is done in two or more shorter cycles, rather than one long cycle.

Place registers to:
- Get a balanced clock cycle length
- Save any results needed for the remaining cycles
Alternative Multiple Cycle Datapath With Control Lines
(Fig 5.28 In Textbook)

(ORI not supported, Jump supported)
(Figure 5.28 page 323)
## Operations (Dependant RTN) for Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>Instruction Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>Instruction Decode</td>
<td></td>
<td></td>
<td></td>
<td></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>EX</strong></td>
<td>Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>Zero ← A - B</td>
<td>PC ← Jump Address</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MDR ← Mem[ALUout]</td>
<td>Mem[ALUout] ← B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>Write Back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R[rd] ← ALUout</td>
<td>R[rt] ← MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fetch (IF) & Instruction Decode cycles are common for all instructions.
Multi-cycle Datapath Instruction CPI

• R-Type/Immediate: Require four cycles, \( \text{CPI} = 4 \)
  – IF, ID, EX, WB

• Loads: Require five cycles, \( \text{CPI} = 5 \)
  – IF, ID, EX, MEM, WB

• Stores: Require four cycles, \( \text{CPI} = 4 \)
  – IF, ID, EX, MEM

• Branches/Jumps: Require three cycles, \( \text{CPI} = 3 \)
  – IF, ID, EX

• Average or effective program CPI: \( 3 \leq \text{CPI} \leq 5 \)
  depending on program profile (instruction mix).
FSM State Transition Diagram (From Book)

IF
- Instruction fetch
  - MemRead
  - ALUSrcA = 0
  - IRD = 0
  - IRWrite
  - ALUSrcB = 01
  - ALUOp = 00
  - PCWrite
  - PCSource = 00

ID
- instruction decode/register fetch
  - ALUSrcA = 0
  - ALUSrcB = 11
  - ALUOp = 00

EX
- Execution
  - ALUout ← A + SignExt(Imm16)
  - ALUout ← ALUout

MEM
- Memory address computation
  - Memory access
    - MemRead
      - IRD = 1
    - MemWrite
      - IRD = 1

WB
- Write-back step
  - Mem[ALUout] ← B

WB
- RegDst = 0
  - RegWrite
  - MemToReg = 1

(Figure 5.38 page 339)
MIPS Multi-cycle Datapath Performance Evaluation

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

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

Average CPI: 4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).
Adding Support for swap to Multi Cycle Datapath

(For More Practice Exercise 5.42)

• You are to add support for a new instruction, swap that exchanges the values of two registers to the MIPS multicycle datapath of Figure 5.28 on page 232
  
  \[
  \text{swap } $rs, $rt
  \]

• Swap used the R-Type format with:
  
  \[
  \text{the value of field } rs = \text{ the value of field } rd
  \]

• Add any necessary datapaths and control signals to the multicycle datapath. Find a solution that minimizes the number of clock cycles required for the new instruction without modifying the register file. Justify the need for the modifications, if any.

• Show the necessary modifications to the multicycle control finite state machine of Figure 5.38 on page 339 when adding the swap instruction. For each new state added, provide the dependent RTN and active control signal values.
Adding swap Instruction Support to Multi Cycle Datapath

We assume here rs = rd in instruction encoding.

Swap $rs, $rt:

\[ R[rt] \leftarrow R[rs] \]

\[ R[rs] \leftarrow R[rt] \]

The outputs of A and B should be connected to the multiplexor controlled by MemtoReg if one of the two fields (rs and rd) contains the name of one of the registers being swapped. The other register is specified by rt. The MemtoReg control signal becomes two bits.

(For More Practice Exercise 5.42)
Adding swap Instruction Support to Multi Cycle Datapath

IF
- Instruction fetch
  - IR ← Mem[PC]
  - PC ← PC + 4

ID
- Instruction decode/register fetch
  - ALUsrcA ← 00
  - ALUsrcB ← 01
  - ALUOp ← 00
  - PCWrite
  - PCSource ← 00

EX
- Execution
  - ALUout ← \( A + \text{SignEx}(\text{imm16}) \)

MEM
- Memory address computation
  - MemRead
  - lorD = 1

MEM
- Memory access
  - MemWrite
  - lorD = 1

WB
- Write back step
  - Mem[ALUout] ← B

WB
- Write back step
  - RegDest = 00
  - RegWrite
  - MemToReg = 00

WB
- Write back step
  - R[rt] ← MDR

WB
- Write back step
  - RegDest = 01
  - RegWrite
  - MemToReg = 10

WB
- Write back step
  - RegDest = 00
  - RegWrite
  - MemToReg = 11

Swap takes 4 cycles

IF
- Instruction fetch
  - IR ← Mem[PC]
  - PC ← PC + 4

ID
- Instruction decode/register fetch
  - ALUsrcA ← 00
  - ALUsrcB ← 11
  - ALUOp ← 00

EX
- Execution
  - ALUout ← \( A \) func \( B \)

MEM
- Memory address computation
  - MemRead
  - lorD = 0

MEM
- Memory access
  - MemWrite
  - lorD = 0

WB
- Write back step
  - Mem[ALUout] ← B

WB
- Write back step
  - RegDest = 01
  - RegWrite
  - MemToReg = 10

WB
- Write back step
  - RegDest = 00
  - RegWrite
  - MemToReg = 11

Swap takes 4 cycles
Adding Support for add3 to Single Cycle Datapath
(For More Practice Exercise 5.45)

• You are to add support for a new instruction, add3, that adds the values of three registers, to the MIPS multicycle datapath of Figure 5.28 on page 232. For example:

add3 $s0,$s1, $s2, $s3

Register $s0 gets the sum of $s1, $s2 and $s3. The instruction encoding uses a modified R-format, with an additional register specifier \( rx \) added replacing the five low bits of the “funct” field.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>rx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s1</td>
<td>$s2</td>
<td>$s0</td>
<td>Not used</td>
</tr>
</tbody>
</table>

• Add necessary datapath components, connections, and control signals to the multicycle datapath without modifying the register bank or adding additional ALUs. Find a solution that minimizes the number of clock cycles required for the new instruction. Justify the need for the modifications, if any.

• Show the necessary modifications to the multicycle control finite state machine of Figure 5.38 on page 339 when adding the add3 instruction. For each new state added, provide the dependent RTN and active control signal values.
Exercise 5.45: add3 instruction support to Multi Cycle Datapath

Add3 $rd, $rs, $rt, $rx

\[ R[rd] \leftarrow R[rs] + R[rt] + R[rx] \]

rx is a new register specifier in field [0-4] of the instruction

No additional register read ports or ALUs allowed

<table>
<thead>
<tr>
<th>Modified R-Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>rx</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31-26]</td>
<td>[25-21]</td>
<td>[20-16]</td>
<td>[10-6]</td>
<td>[4-0]</td>
<td></td>
</tr>
</tbody>
</table>

1. ALUout is added as an extra input to first ALU operand MUX to use the previous ALU result as an input for the second addition.
2. A multiplexor should be added to select between \( rt \) and the new field \( rx \) containing register number of the 3rd operand (bits 4-0 for the instruction) for input for Read Register 2.
   This multiplexor will be controlled by a new one bit control signal called ReadSrc.
3. WriteB control line added to enable writing \( R[rx] \) to B

---

**Diagram Notes:**
- **Instruction [15-6]:**
  - Instruction [15-6] = 0 is taken if it is an instruction.
- **Instruction [13-26]:**
  - Instruction [13-26] = 1 is taken.
- **WriteB Control Line:**
  - Added to enable writing \( R[rx] \) to B.
Exercise 5.45: add3 instruction support to Multi Cycle Datapath

IF

IR ← Mem[PC]
Pc ← PC + 4

ID1

ALUSrcA = 00
ALUSrcB = 01
ALUOp = 00
PcWrite
PCsource = 00

ALUSrcA = 00
ALUSrcB = 11
ALUOp = 00
PcWrite
PCsource = 00

ALUSrcA = 01
ALUSrcB = 00
ALUOp = 01
PcWrite
PCsource = 01

ALUSrcA = 01
ALUSrcB = 00
ALUOp = 01
PcWrite
PCsource = 00

ALUout ← PC + (SignExt(imm16) x 4)

EX

ALUout ← A + SignEx(imm16)

EX1

ALUout ← A + B

B ← R[rx]

EX2

ALUout ← ALUout + B

Add3 takes 5 cycles

(For More Practice Exercise 5.45)