CPU Organization (Design)

• **Datapath Design:**
  - 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:**
  - 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).

Chapter 5.1-5.4
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 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.
CPU Design & Implantation Process

- **Top-down Design:**
  - Specify component behavior from high-level requirements (ISA).
- **Bottom-up Design:**
  - Assemble components in target technology to establish critical timing (hardware delays, critical path timing).
- **Iterative refinement:**
  - Establish a partial solution, expand and improve.

Instruction Set Architecture (ISA):
Provides Requirements

Datapath

Processor

Control

Reg. File Mux ALU Reg Mem Decoder Sequencer

Target VLSI implementation Technology

Cells Gates
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.
# MIPS Instruction Formats

## R-Type

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
</tr>
<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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `op`: Opcode, operation of the instruction.
- `rs`, `rt`, `rd`: The source and destination register specifiers.
- `shamt`: Shift amount.
- `funct`: Selects the variant of the operation in the “op” field.

## I-Type: ALU

### Load/Store, Branch

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `op`: Opcode, operation of the instruction.
- `rs`, `rt`: The source and destination register specifiers.
- `immediate`: Address offset or immediate value.

## J-Type: Jumps

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `op`: Opcode, operation of the instruction.
- `target address`: Target address of the jump instruction.
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>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</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

R-Type = Register Type
Register Addressing used (Mode 1)
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: `$3`
  - Offset: `500`
  - Base register in rs: `$4`
  - Signed address offset in bytes: `0x1F0`
  - `Mem[R[rs] + address] ← R[rt]`
  - `PC ← PC + 4`

- **Load word**: `lw $1, 32($2)`
  - Destination register in rt: `$1`
  - Offset: `32`
  - Base register in rs: `$2`
  - Signed address offset in bytes: `0x0020`
  - `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:

\[
R[rs] = R[rt] : \quad PC \leftarrow PC + 4 + \text{address x 4}
\]
\[
R[rs] \neq R[rt] : \quad PC \leftarrow PC + 4
\]
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>

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

Examples:
- Jump   \( j 10000 \)
- Jump and link   \( jal 10000 \)

Effective 32-bit jump address:
\[
\text{PC(31-28)}, \text{jump_target}, 00
\]

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

From \( \text{PC(31-28)} \)

\[
\begin{array}{c|c|c|c}
\text{jump target} & 2500 & 0 & 0 \\
\end{array}
\]

4 bits 26 bits 2 bits

Independent RTN for j:
\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{PC} & \leftarrow \text{PC(31-28)}, \text{jump_target}, 00
\end{align*}
\]
A Subset of MIPS Instructions

ADD and SUB:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
<th>Field 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>add rd, rs, rt</td>
<td>31 26 21 16 11 6 0</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</td>
<td></td>
</tr>
</tbody>
</table>

OR Immediate:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ori rt, rs, imm16</td>
<td>31 26 21 16 0 0</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LOAD and STORE Word

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw rt, rs, imm16</td>
<td>31 26 21 16 0 0</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw rt, rs, imm16</td>
<td>31 26 21 16 0 0</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BRANCH:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq rs, rt, imm16</td>
<td>31 26 21 16 0 0</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic MIPS Instruction Processing Steps

1. **Instruction Fetch**
   - Obtain instruction from program storage
     \[ \text{Instruction} \leftarrow \text{Mem}[\text{PC}] \]

2. **Instruction Decode**
   - Determine instruction type
   - Obtain operands from registers
   - Compute result value or status

3. **Execute**
   - Update program counter to address of next instruction
     \[ \text{PC} \leftarrow \text{PC} + 4 \]

4. **Result Store**
   - Store result in register/memory if needed
     (usually called Write Back).

Common steps for all instructions

Done by Control Unit

Instruction Memory
Overview of MIPS Instruction Micro-operations

- All instructions go through these common steps:
  - Send program counter to instruction memory and **fetch** the instruction.  
    *(fetch) Instruction ← Mem[PC]*
  - **Update the program counter** to point to next instruction  
    *PC ← PC + 4*
  - **Read one or two registers**, using instruction fields. *(decode)*  
    - Load reads one register only.

- Additional instruction execution actions *(execution)* depend on the instruction in question, but similarities exist:
  - **All instruction classes use the ALU** after reading the registers:  
    - Memory reference instructions use it for address calculation.
    - Arithmetic and logic instructions (R-Type), use it for the specified operation.
    - Branches use it for comparison.

- Additional execution steps where instruction classes differ:  
  - **Memory reference instructions**: Access memory for a load or store.
  - **Arithmetic and logic instructions**: Write ALU result back in register.
  - **Branch instructions**: Change next instruction address based on comparison.
A Single Cycle MIPS CPU Design

Design target: A single-cycle per instruction MIPS CPU design

All micro-operations of an instruction are to be carried out in a single CPU clock cycle. Cycles Per Instruction = CPI = 1

CPU Performance Equation:

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

Abstract view of single cycle MIPS CPU showing major functional units (components) and major connections between them.
**R-Type Example:**

**Micro-Operation Sequence For ADD**

```
add rd, rs, rt
```

<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>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</td>
</tr>
</tbody>
</table>

Instruction Word $\leftarrow$ Mem[PC]

PC $\leftarrow$ PC + 4

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

Fetch the instruction – Common Steps

Increment PC

Add register rs to register rt result in register rd

Independent RTN?
Initial Datapath Components

Three components needed by: Instruction Fetch: Instruction ← Mem[PC]

Program Counter Update: PC ← PC + 4

Two state elements (memory) needed to store and access instructions:

1 Instruction memory:
   • Only read access (by user code). No read control signal needed.

2 Program counter (PC): 32-bit register.
   • Written at end of every clock cycle (edge-triggered): No write control signal.

3 32-bit Adder: To compute the next instruction address (PC + 4).

Basics of logic design/logic building blocks review in Appendix B (Book CD)
More Datapath Components

ISA Register File

- **Registers**
  - Contains all ISA registers.
  - Two read ports and one write port.
  - Register writes by asserting write control signal
  - **Clocking Methodology:** Writes are edge-triggered.
    - Thus can read and write to the same register in the same clock cycle.

Main 32-bit ALU

- **ALU**

**Register File:**

a. Registers

b. ALU

Basics of logic design/logic building blocks review in Appendix B (Book CD)
Register File consists of 32 registers:
- Two 32-bit output busses: busA and busB
- One 32-bit input bus: busW

Register is selected by:
- RA (number) selects the register to put on busA (data):
  \[ \text{busA} = R[\text{RA}] \]
- RB (number) selects the register to put on busB (data):
  \[ \text{busB} = R[\text{RB}] \]
- RW (number) selects the register to be written via busW (data) when Write Enable is 1
  \[ \text{Write Enable: } R[\text{RW}] \leftarrow \text{busW} \]

Clock input (CLK)
- The CLK input is a factor ONLY during write operations.
- During read operation, it behaves as a combinational logic block:
  - RA or RB valid \(\Rightarrow\) busA or busB valid after “access time.”
A Possible Register File Implementation

Each Register contains 32 edge triggered D-Flip Flops

Write Register RW

5-to-32 Decoder

Register Write Enable (RegWrite)

Write Register 0

Data In

Data Out

Write Register 1

Data In

Data Out

Write Register 30

Data In

Data Out

Write Register 31

Data In

Data Out

32-to-1 MUX

Register Read Data 1 (Bus A)

Read Register 1 (RA)

32-to-1 MUX

Register Read Data 2 (Bus B)

Read Register 2 (RB)

Write Enable

RW RA RB

busA

busB

32-bit Registers

Register Write Data (Bus W)

Write Register Write Data (Bus W)

Clock input to registers not shown in diagram

Also see Appendix B (Book CD) - The Basics of Logic Design
Idealized Memory

• Memory (idealized)
  – One input bus: Data In.
  – One output bus: Data Out.

• Memory word is selected by:
  – Address selects the word to put on Data Out bus.
  – Write Enable = 1: address selects the memory word to be written via the Data In bus.

• Clock input (CLK):
  – The CLK input is a factor ONLY during write operation,
  – During read operation, this memory behaves as a combinational logic block:
    • Address valid => Data Out valid after “access time.”

• Ideal Memory = Short access time.
Clocking Methodology Used: 
Edge Triggered Writes

- All storage elements (e.g., flip-flops, registers, data memory) writes are triggered by the same clock edge.
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

Here writes are triggered on the rising edge of the clock.
Building The Datapath

Instruction Fetch & PC Update:

Instruction ← Mem[PC]

Portion of the datapath used for fetching instructions and incrementing the program counter.

PC ← PC + 4

PC write or update is edge triggered at the end of the cycle.
Simplified Datapath For MIPS
R-Type Instructions

Components and connections as specified by RTN statement

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

Destination register \( R[rd] \) write or update is edge triggered at the end of the cycle.
More Detailed Datapath
For R-Type Instructions
With Control Points Identified

\[ R[rd] \leftarrow R[rs] + R[rt] \]
**R-Type Register-Register Timing**

- **Clk**: Clock signal.
- **PC**: Program Counter.
- **Rs, Rt, Rd, Op, Func**: Register and operation fields.
- **ALUctr**: ALU control.
- **RegWr**: Register write.
- **busA, B**: Bus signals for data transfer.
- **busW**: Bus signal for write.
- **Rw, Ra, Rb**: Destination registers.
- **ALUct**: ALU control function.
- **Result**: ALU result.

**Key Points**
- All register writes occur on the falling edge of the clock.
- Clocking methodology is used.

**Timing Phases**
- Instruction Memory Access Time
- Delay through Control Logic
- Register File Access Time
- ALU Delay

**Register Write Occurs Here**
Logical Operations with Immediate Example:
Micro-Operation Sequence For ORI

ori rt, rs, imm16

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Word ← Mem[PC]

PC ← PC + 4

R[rt] ← R[rs] OR ZeroExt[imm16]

Fetch the instruction

Increment PC

OR register rs with immediate field zero extended to 32 bits, result in register rt
Datapath For Logical Instructions With Immediate

R\[rt\] ← R\[rs\] OR ZeroExt[imm16]
Load Operations Example:
Micro-Operation Sequence For LW

lw rt, rs, imm16

<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>

Instruction Word ← Mem[PC]

PC ← PC + 4

R[rt] ← Mem[R[rs] + SignExt[imm16]]

Fetch the instruction

Increment PC

Immediate field sign extended to 32 bits and added to register rs to form memory load address, write word at load effective address to register rt

Data Memory

Effective Address

Instruction Memory

Address offset in bytes

[31:26] [25:21] [20:16] [15:0]
Additional Datapath Components For Loads & Stores

**a. Data memory unit**
- **Inputs:**
  for address and write (store) data
- **Output**
  for read (load) data

**b. Sign-extension unit**
- **Inputs:**
  16-bit input sign-extended
- **Output**
  into a 32-bit value at the output

Data memory write or update is edge triggered at the end of the cycle (clocking methodology)
R[rt] ← Mem[R[rs] + SignExt[imm16]]
Store Operations Example:
Micro-Operation Sequence For SW

sw rt, rs, imm16

Instruction Word ← Mem[PC]

PC ← PC + 4

Mem[R[rs] + SignExt[imm16]] ← R[rt]

Fetch the instruction
Increment PC
Immediate field sign extended to 32 bits and added to register rs to form memory store effective address, register rt written to memory at store effective address.
Datapath For Stores

Mem[R[rs] + SignExt[imm16]] ← R[rt]
Conditional Branch Example:
Micro-Operation Sequence For BEQ

```
beq rs, rt, imm16
```

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>26</td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
</tr>
</tbody>
</table>

**Instruction Word** ← Mem[PC]

PC ← PC + 4

Zero ← R[rs] - R[rt]

**Condition**

```
Zero : PC ← PC + ( SignExt(imm16) x 4 )
```

**Action**

Fetch the instruction

Increment PC

Calculate the branch condition

R[rs] == R[rt]

(i.e. R[rs] - R[rt] = 0)

Calculate the next instruction’s PC address

“Zero” is zero flag of main ALU
Main ALU evaluates branch condition

New adder to compute branch target:

- Sum of incremented PC and the sign-extended lower 16-bits on the instruction.

Datapath For Branch Instructions

New 32-bit Adder (Third ALU) for Branch Target

PC + 4 + (SignExt(imm16) x 4)

Add Sum Branch target

Shift left 2

PC + 4 + (SignExt(imm16) x 4)

ALU operation

R[rs]

ALU Zero

To branch control logic

Zero flag = 1
if R[rs] - R[rt] = 0
(i.e. R[rs] = R[rt])

Main ALU Evaluates Branch Condition (subtract)
More Detailed Datapath For Branch Operations

- Branch Zero
- Instruction Address
- 32-bit Registers
- Clk
- busW
- RegWr
- Rs
- Rt
- busA
- busB
- imm16
- PC Ext
- Sign extend shift left 2
- Branch Target ALU
- New 2X1 32-bit MUX to select next PC value
- Zero
Combining The Datapaths For Memory Instructions and R-Type Instructions

Highlighted multiplexors and connections added to combine the datapaths of memory and R-Type instructions into one datapath
(This is book version ORI not supported)
Instruction Fetch Datapath Added to ALU R-Type and Memory Instructions Datapath

(This is book version ORI not supported, no zero extend of immediate needed)
A Simple Datapath For The MIPS Architecture

Datapath of branches and a program counter multiplexor are added. Resulting datapath can execute in a single cycle the basic MIPS instruction:
- load/store word
- ALU operations
- Branches

Figure 5.11 page 300

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

- The main ALU has four control lines (detailed design in Appendix B) with the following functions:

<table>
<thead>
<tr>
<th>ALU Control Lines</th>
<th>ALU Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>Set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

- For our current subset of MIPS instructions only the top five functions will be used (thus only three control lines will be used)
- For R-type instruction the ALU function depends on both the opcode and the 6-bit “funct” function field
- For other instructions the ALU function depends on the opcode only.
- A local ALU control unit can be designed to accept 2-bit ALUop control lines (from main control unit) and the 6-bit function field and generate the correct 4-bit ALU control lines.
Local ALU Decoding of “func” Field

<table>
<thead>
<tr>
<th>Instruction Opcode</th>
<th>Instruction Operation</th>
<th>ALUOp</th>
<th>Funct Field</th>
<th>Desired ALU Action</th>
<th>ALU Control Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Load word</td>
<td>00</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>SW</td>
<td>Store word</td>
<td>00</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>Branch Equal</td>
<td>branch equal</td>
<td>01</td>
<td>XXXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-Type</td>
<td>add</td>
<td>10</td>
<td>1000000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-Type</td>
<td>subtract</td>
<td>10</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-Type</td>
<td>AND</td>
<td>10</td>
<td>100100</td>
<td>and</td>
<td>0000</td>
</tr>
<tr>
<td>R-Type</td>
<td>OR</td>
<td>10</td>
<td>100101</td>
<td>or</td>
<td>0001</td>
</tr>
<tr>
<td>R-Type</td>
<td>set on less than</td>
<td>10</td>
<td>101010</td>
<td>set on less than</td>
<td>0111</td>
</tr>
</tbody>
</table>
Local ALU Control Unit

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Function Field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1</td>
<td>ALUOp0</td>
<td>F5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

FIGURE C.2.1 The truth table for the three ALU control bits (called Operation) as a function of the ALUOp and function code field. This table is the same as that shown Figure 5.13.

(2 lines From main control unit)

3 ALU Control Lines

More details found in Appendix C (Book CD)
Single Cycle MIPS Datapath

Necessary multiplexors and control lines are identified here and local ALU control added:

Figure 5.15 page 305

(This is book version ORI not supported, no zero extend of immediate needed)
Putting It All Together: A Single Cycle Datapath

Instruction Memory

Branch Zero

PC+4 4

Adder

Mux 1 0

Clk

4

Inst Memory

Rs Rt Rd Imm16

ALUop (2-bits)

ALU Control

MemWr MemtoReg

RegDst

Zero

Function Field

Main ALU

ALUSrc ExtOp

WrEn Adr

Data Memory

Data In

MemWr

MemtoReg

Inst<31:0>

Imm16

(ALU includes ORI not in book version)

32 32-bit Registers

Rw Ra Rb

R[s]

R[rt]

R[rs]

Extender

Mux 1 0

Clk

0

Mux 1

(ExtOp)
Instruction Memory

Adr

Op  Fun  Rt  Rs  Rd  Imm16  Jump_target

Control Unit

Control Lines

RegDst  ALUSrc  MemtoReg  RegWrite  Mem Read  Mem Write  Branch  ALOp (2-bits)

DATA PATH
## The Effect of The Control Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Effect when deasserted (=0)</th>
<th>Effect when asserted (=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDst</strong></td>
<td>The register destination number for the write register comes from the rt field (instruction bits 20:16).</td>
<td>The register destination number for the write register comes from the rd field (instruction bits 15:11).</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>None</td>
<td>The register on the write register input is written with the value on the Write data input.</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>The second main ALU operand comes from the second register file output (Read data 2) R[rt]</td>
<td>The second main ALU operand is the sign-extended lower 16 bits on the instruction (imm16)</td>
</tr>
<tr>
<td><strong>PCSrc</strong></td>
<td>The PC is replaced by the output of the adder that computes PC + 4</td>
<td>The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td><strong>MemRead</strong></td>
<td>None</td>
<td>Data memory contents designated by the address input are put on the Read data output.</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>None</td>
<td>Data memory contents designated by the address input are replaced by the value on the Write data input.</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>The value fed to the register write data input comes from the main ALU.</td>
<td>The value fed to the register write data input comes from data memory.</td>
</tr>
</tbody>
</table>
## Control Line Settings

<table>
<thead>
<tr>
<th>Instruction</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

Figure 5.18 page 308
### The Truth Table For The Main Control

<table>
<thead>
<tr>
<th>Control</th>
<th>Signal name</th>
<th>R-format</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Op3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Op1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RegDst</td>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ALUSrc</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td></td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RegWrite</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemRead</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUOp1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOp0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE C.2.4** The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 5.22.
PLA Implementation of the Main Control

Inputs

Op5  Op4  Op3  Op2  Op1  Op0

R-format  lw  sw  beq

Outputs

RegDst  ALUSrc  MemtoReg  RegWrite  MemRead  MemWrite  Branch  ALUOp1  ALUOp0

Figure C.2.5 (Appendix C)

PLA = Programmable Logic Array (Appendix B)
Adding Support For Jump:
**Micro-Operation Sequence For Jump: J**

\[ j \text{ jump}\_\text{target} \]

<table>
<thead>
<tr>
<th>OP</th>
<th>Jump_target</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits [31:26]</td>
<td>26 bits [25:0]</td>
</tr>
</tbody>
</table>

Instruction Word ← Mem[PC]  
Fetch the instruction

PC ← PC + 4  
Increment PC

PC ← PC(31-28), jump_target,00  
Update PC with jump address

**Jump Address**

<table>
<thead>
<tr>
<th>PC(31-28)</th>
<th>[31:26]</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>
<td></td>
</tr>
</tbody>
</table>

4 highest bits from PC + 4
Datapath For Jump

Instruction(25-0)
- jump_target

Instruction(15-0)
- imm16

PC Ext

Adder

Shift left 2

Mux

Adder4

PC Ext

Mux

Branch Target

PC+4(31-28)

JUMP

Next Instruction Address

Jump Address

PC(31-28), jump_target, 00

Branch Zero

Clk

PC
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 Line Settings
(with jump instruction, j added)

<table>
<thead>
<tr>
<th>Instruction</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>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>

Figure 5.18 page 308 modified to include j
### Worst Case Timing (Load)

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

- **Delay through Control Logic**
- **Register File Access Time**
- **Data Memory Access Time**
- **Register Write Occurs**

**Note:**
- Delay through Extender & Mux
- ALU Delay
Instruction Timing Comparison

Arithmetic & Logical

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

Load

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

Critical Path

Store

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

Branch

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

Jump

| PC | Inst Memory | mux |
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 = nanosecond = $10^{-9}$ 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>

Load has longest delay of 8 ns thus determining the clock cycle of the CPU to be 8 ns.

- 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 I = 1,000,000 instructions executed takes:
  Execution Time = T = I x CPI x C = 10^6 x 1 x 8x10^-9 = 0.008 s = 8 msec
Drawbacks of Single Cycle Processor

1. Long cycle time:
   - All instructions must take as much time as the slowest
     - Here, cycle time for load is longer than needed for all other instructions.
     - Cycle time must be long enough for the load instruction:
       - PC’s Clock -to-Q + Instruction Memory Access Time +
         - Register File Access Time + ALU Delay (address calculation) +
         - Data Memory Access Time + Register File Setup Time + Clock Skew
   - Real memory is not as well-behaved as idealized memory
     - Cannot always complete data access in one (short) cycle.

2. Impossible to implement complex, variable-length instructions and complex addressing modes in a single cycle.
   - e.g indirect memory addressing.

3. High and duplicate hardware resource requirements
   - Any hardware functional unit cannot be used more than once in a single cycle (e.g. ALUs).

4. Does not allow overlap of instruction processing (instruction pipelining, chapter 6).