Major CPU Design Steps

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

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

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

3. Control unit design, based on micro-operation timing and control signals identified:
   - Microprogrammed.
CPU Design & Implantation Process

- **Bottom-up Design:**
  - Assemble components in target technology to establish critical timing.

- **Top-down Design:**
  - Specify component behavior from high-level requirements.

- **Iterative refinement:**
  - Establish a partial solution, expand and improve.

---

```
Instruction Set
Architecture
---------------------
Datapath
Reg. File  Mux  ALU  Reg  Mem
Cells  Gates

Decoder  Sequencer
Control

Processor
```
Datapath Design Steps

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

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

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

• Complete the micro-operation sequences for all remaining instructions adding connections/multiplexors as needed.
MIPS Instruction Formats

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

- **J-Type:** Jumps
  - op: Opcode, operation of the instruction.
  - target address: Target address of the jump instruction.
  - 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>
</tbody>
</table>

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

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

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

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

<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>

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

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

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

**Examples:**

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

Signed address offset in bytes

Offset

base register in rs

source register in rt

Destination register in rt

Offset

base register in rs
MIPS Branch I-Type Instruction Fields

- **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.

<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>

Examples:
- Branch on equal
  - `beq $1,$2,100`
- Branch on not equal
  - `bne $1,$2,100`
### MIPS J-Type Instruction Fields

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

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

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

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

**Examples:**

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

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

PC(31-28)

<table>
<thead>
<tr>
<th></th>
<th>jump target = 2500</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jump target = 2500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bits</td>
<td>26 bits</td>
<td>2 bits</td>
<td></td>
</tr>
</tbody>
</table>
# A Subset of MIPS Instructions

**ADD and SUB:**

- `addU rd, rs, rt`
- `subU rd, rs, rt`

**OR Immediate:**

- `ori rt, rs, imm16`

**LOAD and STORE Word**

- `lw rt, rs, imm16`
- `sw rt, rs, imm16`

**BRANCH:**

- `beq rs, rt, imm16`

### MIPS Instruction Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
</tr>
<tr>
<td>rd</td>
<td>5 bits</td>
</tr>
<tr>
<td>shamt</td>
<td>5 bits</td>
</tr>
<tr>
<td>funct</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
</tr>
<tr>
<td>immediate</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Instruction Processing Steps

1. **Instruction Fetch**: Obtain instruction from program storage.
2. **Instruction Decode**: Determine instruction type. Obtain operands from registers.
3. **Execute**: Compute result value or status.
4. **Result Store**: Store result in register/memory if needed (usually called Write Back).

Common steps for all instructions: Update program counter to address of next instruction.
Overview of MIPS Instruction Micro-operations

- All instructions go through these two steps:
  - Send program counter to instruction memory and fetch the instruction. *(fetch)*
  - 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 Implementation

Design target: A single-cycle instruction implementation
All micro-operations of an instruction are to be carried out in a single system clock cycle.
Datapath Components

Two state elements needed to store and access instructions:

1. Instruction memory:
   - Only read access.
   - No read control signal.

   - Written at end of every clock cycle: No write control signal.
   - 32-bit Adder: To compute the next instruction address.
Register File:
- Contains all registers.
- Two read ports and one write port.
- Register writes by asserting write control signal
- Writes are edge-triggered.
- Can read and write to the same register in the same clock cycle.
Register File Details

- 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.”
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.”
R-Type Example:
Micro-Operation Sequence For ADDU

addU 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>
</tbody>
</table>

Instruction Word ← Mem[PC]
Fetch the instruction

PC ← PC + 4
Increment PC

R[rd] ← R[rs] + R[rt]
Add register rs to register rt result in register rd
Building The Datapath

Instruction Fetch & PC Update:

Portion of the datapath used for fetching instructions and incrementing the program counter.
Simplified Datapath For R-Type Instructions
More Detailed Datapath For R-Type Instructions
With Control Points Identified
R-Type Register-Register Timing

- **Clk**: Clock signal.
- **PC**: Program Counter.
- **Rs, Rt, Rd, Op, Func**: Source and target registers, operation code.
- **ALUctr**: ALU Control.
- **RegWr**: Register Write.
- **busA, B**: Bus signals A and B.
- **busW**: Bus Writes.

**Timing Events**:
- **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>
</tbody>
</table>

Instruction Word  ←  Mem[PC]  
Fetch the instruction

PC  ←  PC + 4  
Increment PC

R[rt]  ←  R[rs] OR ZeroExt[imm16]  
OR register rs with immediate field zero extended to 32 bits, result in register rt
Datapath For Logical Instructions With Immediate

- Mux
- RegDst
- RegWr
- busW
- Clk
- imm16
- ZeroExt
- Rw, Ra, Rb
- 32 32-bit Registers
- ALU
- ALUctr
- busA, busB
- Result
- ALUSrc

32-bit Registers
Load Operations Example:
Micro-Operation Sequence For LW

lw 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>
</tbody>
</table>

Instruction Word ← Mem[PC]  
Fetch the instruction

PC ← PC + 4  
Increment PC

R[rt] ← Mem[R[rs] + SignExt[imm16]]  
Immediate field sign extended to 32 bits and added to register rs to form memory load address, word at load address to register rt
Additional Datapath Components For Loads & Stores

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

b. Sign-extension unit
   16-bit input sign-extended into a 32-bit value at the output
Datapath For Loads

RegDst → Mux

RegWr → Mux

Rs → Mux

Rd → Mux

Rt → Mux

busW → 32 32-bit Registers

busA → ALU

busB → MemWr

imm16 → Extender

ExtOp → ALUSrc

ALUctr → ALU

Clk → ALU

WrEn → Memory

Adr → Memory

Data In → Memory

Data → Memory

MemWr → Mux

W_Src → Mux

Rw Ra Rb

ExtOp

Clk

32

32

32

32

32

32

32
Store Operations Example:

**Micro-Operation Sequence For SW**

sw 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]  
Fetch the instruction

PC ← PC + 4  
Increment PC

Mem[R[rs] + SignExt[imm16]] ← R[rt]  
Immediate field sign extended to 32 bits and added to register rs to form memory store address, register rt written to memory at store address.
Datapath For Stores

RegDst → Mux → Rd, Rt → Mux → ALUctr, MemWr, W_Src

RegWr → Mux → Rs, 5 → Mux → ALUctr

busW → Mux → busA, 32-bit Registers

Clk → Mux → 32bit Registers

imm16 → Extender → Mux → ExtOp

ALUSrc → ALU → 32

Data In → Mux → Data Memory

MemWr → Data Memory

WrEn → Data Memory

Adr → Data Memory

W_Src → Mux

Clk → Mux

32 32-bit Registers

Extender

ALU

Data Memory

32

32

32

32

32
Conditional Branch Example:

Micro-Operation Sequence For BEQ

beq rs, rt, imm16

Instruction Word ← Mem[PC]
PC ← PC + 4
Equal ← R[rs] == R[rt]

if (COND eq 0)
    PC ← PC + 4 + (SignExt(imm16) x 4)
else
    PC ← PC + 4

Fetch the instruction
Increment PC
Calculate the branch condition
Calculate the next instruction’s PC address
Datapath For Branch Instructions

ALU to evaluate branch condition
Adder to compute branch target:
- Sum of incremented PC and the sign-extended lower 16-bits on the instruction.
More Detailed Datapath For Branch Operations

Instruction Address

32 32-bit Registers

Equal?
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.
Instruction Fetch Datapath Added to ALU R-Type and Memory Instructions Datapath
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
Single Cycle MIPS Datapath

Necessary multiplexors and control lines are identified here:
Putting It All Together: A Single Cycle Datapath

Instruction<31:0>

Inst Memory

Rs Rt Rd Imm16

Rw Ra Rb

32 32-bit Registers

RegDst

Equal

ALUctr MemWr MemtoReg

ExtOp ALUSrc

Data In

WrEn Adr

Data Memory

MemWr

MemtoReg

ALU

32 32-bit Registers
Adding Support For Jump:

Micro-Operation Sequence For Jump: \( J \)

\[
j \quad \text{jump\_target}
\]

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

Instruction Word \( \leftarrow \) Mem[PC]            Fetch the instruction
PC \( \leftarrow \) PC + 4                            Increment PC
PC \( \leftarrow \) PC(31-28),\text{jump\_target},00  Update PC with jump address
Datapath For Jump

Instruction(15-0)

Instruction(25-0)

Jump target

Shift left 2

Next Instruction Address

PC

Clk
Single Cycle MIPS Datapath Extended To Handle Jump with Control Unit Added
# Control Signal Generation

<table>
<thead>
<tr>
<th>func</th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>10 0010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>Don’t Care</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>00 0000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>00 1101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>10 0011</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 1011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00 0100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00 0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nPCsel</th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jump</th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ExtOp</th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUctr&lt;2:0&gt;</th>
<th>Add</th>
<th>Subtract</th>
<th>Or</th>
<th>Add</th>
<th>Add</th>
<th>Subtract</th>
<th>xxx</th>
</tr>
</thead>
</table>

See Appendix A
The Concept of Local Decoding

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegDst</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop&lt;N:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

Diagram:
- op 6 → Main Control
- func 6 → ALU Control (Local)
- ALUop N → ALU Control (Local)
- ALUctrl 3 → ALU
Local Decoding of “func” Field

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00</td>
<td>“R-type”</td>
<td>0 10</td>
<td>0 00</td>
<td>0 00</td>
<td>0 01</td>
<td>xxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct&lt;5:0&gt;</th>
<th>Instruction Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0000</td>
<td>add</td>
</tr>
<tr>
<td>10 0010</td>
<td>subtract</td>
</tr>
<tr>
<td>10 0100</td>
<td>and</td>
</tr>
<tr>
<td>10 0101</td>
<td>or</td>
</tr>
<tr>
<td>10 1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUctr&lt;2:0&gt;</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Add</td>
</tr>
<tr>
<td>001</td>
<td>Subtract</td>
</tr>
<tr>
<td>010</td>
<td>And</td>
</tr>
<tr>
<td>110</td>
<td>Or</td>
</tr>
<tr>
<td>111</td>
<td>Set-on-less-than</td>
</tr>
</tbody>
</table>
### The Truth Table for ALUctr

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>“R-type”</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 0 0</td>
<td>0 1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>bit&lt;2&gt;</th>
<th>bit&lt;1&gt;</th>
<th>bit&lt;0&gt;</th>
<th>R-type</th>
<th>ALUop func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0 x 1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0 1 x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1 x x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 x x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 x x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 x x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 x x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU Operation</th>
<th>ALUctr bit&lt;2&gt;</th>
<th>ALUctr bit&lt;1&gt;</th>
<th>ALUctr bit&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subtract</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Or</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subtract</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>And</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Or</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Set on &lt;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The Truth Table For The Main Control

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop (Symbolic)</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<tr>
<td>ALUop &lt;2&gt;</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;1&gt;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;0&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>
Example: RegWrite Logic Equation

\[
\text{RegWrite} = \text{R-type} + \text{ori} + \text{lw}
\]

\[
= \neg op<5> \& \neg op<4> \& \neg op<3> \& op<2> \& \neg op<1> \& \neg op<0> \quad \text{(R-type)}
\]

\[
+ \neg op<5> \& \neg op<4> \& op<3> \& op<2> \& \neg op<1> \& op<0> \quad \text{(ori)}
\]

\[
+ op<5> \& \neg op<4> \& \neg op<3> \& \neg op<2> \& op<1> \& op<0> \quad \text{(lw)}
\]
PLA Implementation of the Main Control

R-type, ori, lw, sw, beq, jump

RegWrite, ALUSrc, RegDst, MemtoReg, MemWrite, Branch, Jump, ExtOp, ALUop<2>, ALUop<1>, ALUop<0>
Worst Case Timing (Load)

- Clk
- PC Old Value → New Value
- Rs, Rt, Rd, Op, Func
- ALUctr
- ExtOp
- ALUSrc
- MemtoReg
- RegWr
- busA
- busB
- Address
- busW

Instruction Memory Access Time
Delay through Control Logic
Register File Access Time
Delay through Extender & Mux
ALU Delay
Data Memory Access Time

Register Write Occurs
# 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 |
```
Performance of Single-Cycle CPU

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

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

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

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

• A program with 1,000,000 instructions takes:
  Execution Time = T = I x CPI x C = 10^6 x 1 x 8x10^{-9} = 0.008 s = 8 msec
Drawback of Single Cycle Processor

- Long cycle time:
  - 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

- All instructions must take as much time as the slowest
  - Cycle time for load is longer than needed for all other instructions.

- Real memory is not as well-behaved as idealized memory
  - Cannot always complete data access in one (short) cycle.