CPU Design Steps

1. Analyze instruction set operations using independent RTN => datapath requirements.

2. Select set of datapath components & establish clock methodology.

3. Assemble datapath meeting the requirements.

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic.
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.

Diagram:

```
Instruction Set =>
Architecture

processor

datapath

control

Reg. File  Mux  ALU  Reg  Mem

Decoder  Sequencer

Cells  Gates
```
Single Cycle MIPS Datapath: CPI = 1, Long Clock Cycle

Inst Memory

<nPC sel

RegDst

Rd | Rt

Equal

ALUctr MemWr MemtoReg

ALUSrc

ExtOp

WrEn

Data In

Clk

32 32-bit Registers

32

32

imm16

32

Data Memory

Clk

Extender

Mux

busB

busW

RegWr

Rs | Rt

PC

4

Adder

Adder

Adder

Mux

32

16

imm16

<21:25>

<16:20>

<11:15>

<0:15>

Rs

Rt

Rd

Imm16

Clk
Drawback of Single Cycle Processor

• Long cycle time.

• 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.
Abstract View of Single Cycle CPU
## Single Cycle Instruction Timing

### Arithmetic & Logical

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>mux</th>
<th>setup</th>
</tr>
</thead>
</table>

### Load

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>Data Mem</th>
<th>mux</th>
<th>setup</th>
</tr>
</thead>
</table>

Critical Path

### Store

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>Data Mem</th>
</tr>
</thead>
</table>

### Branch

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>cmp</th>
<th>mux</th>
</tr>
</thead>
</table>
Reducing Cycle Time: Multi-Cycle Design

- Cut combinational dependency graph by inserting registers / latches.
- The same work is done in two or more fast cycles, rather than one slow cycle.
Clock Cycle Time & Critical Path

- Critical path: the slowest path between any two storage devices
- Cycle time is a function of the critical path
- must be greater than:
  - Clock-to-Q + Longest Path through the Combination Logic + Setup
Instruction Processing Cycles

1. **Instruction Fetch**: Obtain instruction from program storage.
2. **Instruction Decode**: Determine instruction type and 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:
- **Next Instruction**: Update program counter to address of next instruction.
Partitioning The Single Cycle Datapath

Add registers between smallest steps

Diagram showing the datapath with steps labeled:
- Instruction Fetch
- Operand Fetch
- Exec
- Mem Access
- ALU ctr
- RegDst
- ALUSrc
- MemRd
- MemWr
- RegWr
- MemWrite
- Result Store
Example Multi-cycle Datapath

Registers added:

**IR:** Instruction register

**A, B:** Two registers to hold operands read from register file.

**R:** or ALUOut, holds the output of the ALU

**M:** or Memory data register (MDR) to hold data read from data memory
### Operations In Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>Fetch</strong></td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
<td>IR ← Mem[PC]</td>
</tr>
<tr>
<td></td>
<td><strong>Decode</strong></td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
<td>A ← R[rs]</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td></td>
<td>R ← A + B</td>
<td>R ← A OR ZeroExt[imm16]</td>
<td>R ← A + SignEx(Im16)</td>
<td>R ← A + SignEx(Im16)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td>M ← Mem[R]</td>
<td>Mem[R] ← B</td>
<td>PC ← PC + 4</td>
</tr>
</tbody>
</table>
Finite State Machine (FSM) Control Model

- State specifies control points for Register Transfer.
- Transfer occurs upon exiting state (same falling edge).

Diagram:

- Inputs (conditions) flow to Next State Logic.
- Next State Logic leads to Control State.
- Control State leads to Output Logic.
- Output Logic produces outputs (control points).

State $X$:
- Register Transfer Control Points depend on input.

Legend:
- Solid arrows indicate transitions.
- Dashed arrows indicate dependencies.
Control Specification For Multi-cycle CPU
Finite State Machine (FSM)

IR ← MEM[PC]

A ← R[rs]
B ← R[rt]

“instruction fetch”

“decode / operand fetch”

R-type

ORi

LW

SW

BEQ & Equal

BEQ & ~Equal

PC ← PC + 4

PC ← PC + SX || 00

To instruction fetch

Memory

Execute

R ← A fun B
R ← A or ZX
R ← A + SX
R ← A + SX
M ← MEM[R]
MEM[R] ← B
PC ← PC + 4
PC ← PC + 4

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

To instruction fetch
To instruction fetch
To instruction fetch
### Traditional FSM Controller

<table>
<thead>
<tr>
<th>state</th>
<th>op</th>
<th>cond</th>
<th>next state</th>
<th>control points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Truth or Transition Table

Equal

datapath State

11

next State

control points

To datapath

6

State

4

op
Traditional FSM Controller

datapath + state diagram => control

• Translate RTN statements into control points.

• Assign states.

• Implement the controller.
Mapping RTNs To Control Points Examples & State Assignments

IR ← MEM[PC]
0000

“instruction fetch”

A ← R[rs]
B ← R[rt]
0001

“decode / operand fetch”

R-type

imem_rd, IRen

Aen, Ben

ALUfun, Sen

R ← A fun B
0100

R ← A or ZX
0110

R ← A + SX
1000

R ← A + SX
1011

M ← MEM[S]
1001

MEM[S] ← B
PC ← PC + 4
1100

PC ← PC + 4
0011

BEQ & Equal

PC ← PC + SX || 00
0010

To instruction fetch state 0000

To instruction fetch state 0000

To instruction fetch state 0000

Write-back

imem_rd, IRen

Aen, Ben

ALUfun, Sen

R ← A fun B
0100

R ← A or ZX
0110

R ← A + SX
1000

R ← A + SX
1011

M ← MEM[S]
1001

MEM[S] ← B
PC ← PC + 4
1100

PC ← PC + 4
0011

BEQ & Equal

PC ← PC + SX || 00
0010

To instruction fetch state 0000

To instruction fetch state 0000

To instruction fetch state 0000

Write-back
<table>
<thead>
<tr>
<th>State</th>
<th>Op field</th>
<th>Eq</th>
<th>Next</th>
<th>IR</th>
<th>PC en sel</th>
<th>Ops A B</th>
<th>Exec Ex Sr ALU S</th>
<th>Mem R W M</th>
<th>Write-Back M-R Wr Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>??????</td>
<td>?</td>
<td>0001</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>0</td>
<td>0011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>BEQ</td>
<td>1</td>
<td>0010</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>R-type</td>
<td>x</td>
<td>0100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>orI</td>
<td>x</td>
<td>0110</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>LW</td>
<td>x</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SW</td>
<td>x</td>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0101</td>
<td>1</td>
<td>1</td>
<td>1 fun</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>xxxxxx</td>
<td>x</td>
<td>0111</td>
<td>1</td>
<td>1</td>
<td>1 or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>xxxxxx</td>
<td>x</td>
<td>1001</td>
<td>1</td>
<td>1 add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>xxxxxx</td>
<td>x</td>
<td>1010</td>
<td>1</td>
<td>1 add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1 add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>xxxxxx</td>
<td>x</td>
<td>1100</td>
<td>1</td>
<td>1 add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>xxxxxx</td>
<td>x</td>
<td>0000</td>
<td>1</td>
<td>1 add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BEQ**

**R**

**ORI**

**LW**

**SW**

**Lec # 5 Summer 2000**

EECC550 - Shaaban
Alternative Multiple Cycle Datapath (In Textbook)

- Minimizes Hardware: 1 memory, 1 adder
• Shared instruction/data memory unit
• A single ALU shared among instructions
• Shared units require additional or widened multiplexors
• Temporary registers to hold data between clock cycles of the instruction:
  • Additional registers: Instruction Register (IR), Memory Data Register (MDR), A, B, ALUOut
# Operations In Each Cycle

<table>
<thead>
<tr>
<th></th>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUout ← PC +</td>
<td>ALUout ← PC + (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></td>
<td></td>
<td></td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>ALUout ← A + B</td>
<td></td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If Equal = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PC ← ALUout</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td>M ← Mem[ALUout]</td>
<td>Mem[ALUout] ← B</td>
<td></td>
</tr>
</tbody>
</table>
High-Level View of Finite State Machine Control

- First steps are independent of the instruction class
- Then a series of sequences that depend on the instruction opcode
- Then the control returns to fetch a new instruction.
- Each box above represents one or several state.
Instruction Fetch and Decode FSM States

Start

0

MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

Instruction decode/ Register fetch

1

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

(Op = BEQ)

(Op = R-type)

(Op = 'JMP')

Instruction fetch

(Op = 'LW') or (Op = 'SW')

Memory reference FSM (Figure 5.38)

R-type FSM (Figure 5.39)

Branch FSM (Figure 5.40)

Jump FSM (Figure 5.41)
Load/Store Instructions FSM States

From state 1
(Op = 'LW') or (Op = 'SW')
Memory address computation

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Memory access
(Op = 'SW')

MemRead
lorD = 1

Write-back step

RegWrite
MemaReg = 1
RegDest = 0

To state 0
(Figure 5.37)
R-Type Instructions
FSM States

From state 1
(Op = R-type)

Execution

6

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

R-type completion

7

RegDst = 1
RegWrite
MemtoReg = 0

To state 0
(Figure 5.37)
Branch Instruction
Single State

From state 1
(Op = 'BEQ')

aluSrcA = 1
aluSrcB = 00
aluOp = 01
PCWriteCond
PCSource = 01

Branch completion

To state 0
(Figure 5.37)

Jump Instruction
Single State

From state 1
(Op = 'J')

PCWrite
PCSource = 10

Jump completion

To state 0
(Figure 5.37)
Finite State Machine (FSM) Specification

IR ← MEM[PC]
PC ← PC + 4
0000

“instruction fetch”

A ← R[rs]
B ← R[rt]

ALUout ← PC + SX
0001

“decode”

R-type

ALUout ← A fun B
0100

ORi

ALUout ← A op ZX
0110

LW

ALUout ← A + SX
1000

SW

ALUout ← A + SX
1011

BEQ

If A = B then
PC ← ALUout
0010

To instruction fetch

Memory

Execute

R[rd] ← ALUout
0101

To instruction fetch

R[rt] ← ALUout
0111

To instruction fetch

R[rt] ← M
1010

Write-back

MEM[ALUout] ← B
1100

To instruction fetch
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 for type</th>
<th>Frequency</th>
<th>CPI \times freq</th>
<th>Average CPI:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
<td>4.1</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>

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