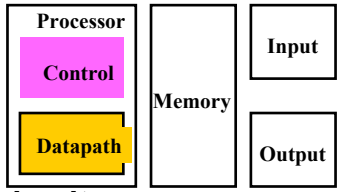
Chapter Five

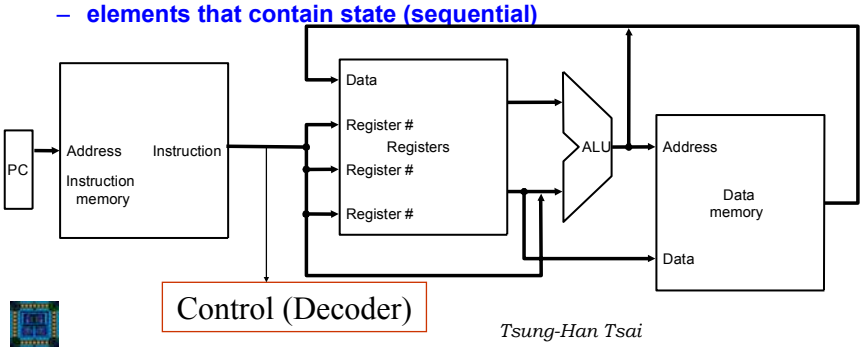
The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers

5.1 Introduction

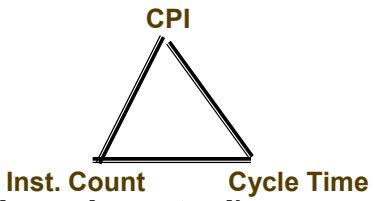
- The Five Classic Components of a Computer
- An abstract view of major functions of MIPS is shown in Fig.5.1. Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)





The Big Picture: The Performance Perspective

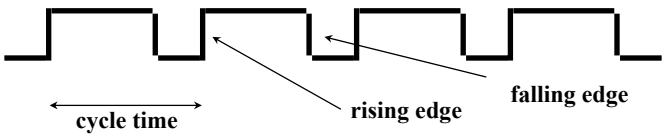
- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction



- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time

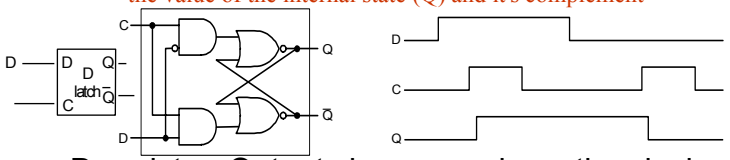
5.2 State Elements: Latches and Registers

- Unclocked (seldom used) vs. Clocked state elements
- Output is equal to the stored value inside the element
  (don't need to ask for permission to look at the value)
- Clocks are used in synchronous logic
  - when should an element that contains state be updated?
  - Change of state (value) is based on the clock
  - Latches: whenever the inputs change, and the clock is asserted
  - Flip-Flop: state changes only on a clock edge
    (edge-triggered methodology)

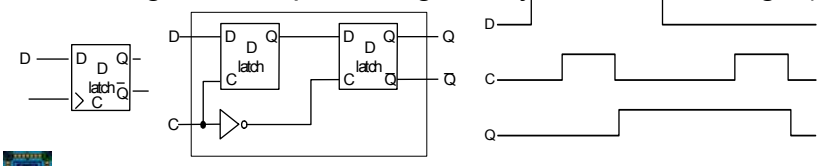


D-latch and D-register

- **D-latch (6.3.3 of Digital Logic)**
  - **Two inputs:**
    - the data value to be stored (D)
    - the clock signal (C) indicating when to read & store D
  - **Two outputs:**
    - the value of the internal state (Q) and it's complement

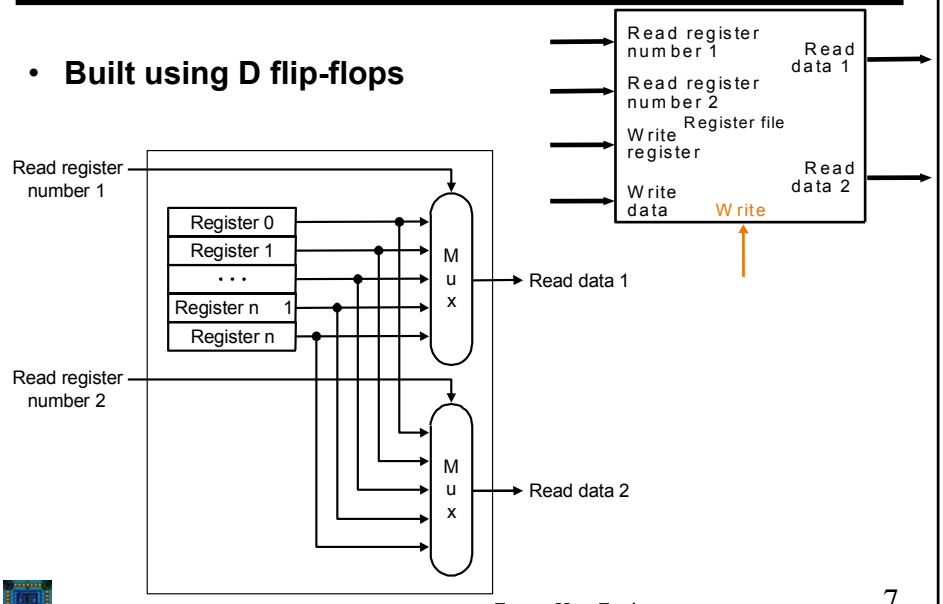


- D-register: Output changes only on the clock edge (6.4.2)



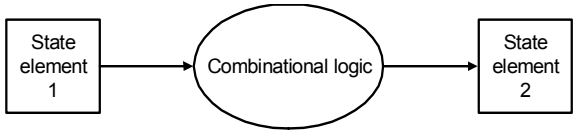
Register File

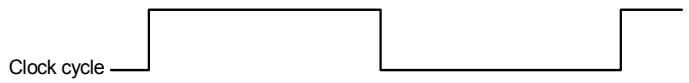
- **Built using D flip-flops**



Our Implementation

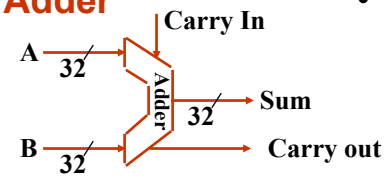
- **An edge triggered methodology**
- **Typical execution:**
  - **read contents of some state elements,**
  - **send values through some combinational logic**
  - **write results to one or more state elements**



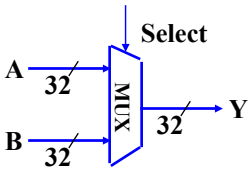


5.2 Building a Datapath

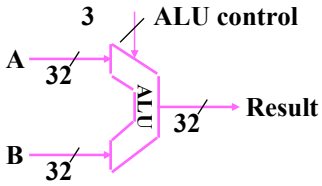
- **Adder**



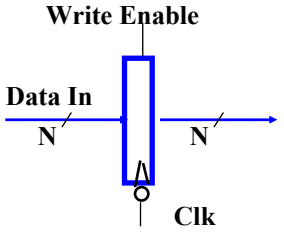
- **MUX**



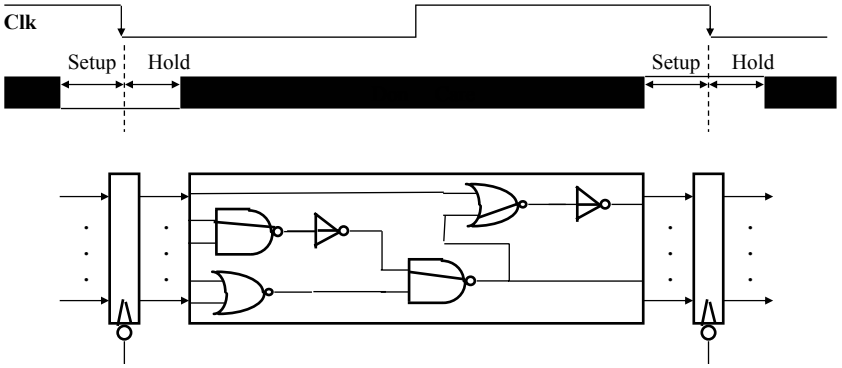
- **ALU**



- **N-bit Register**
  - Consist of N D Flip-Flop
  - N-bit input and output
  - Write Enable:asserted (1): Data Out will become Data In

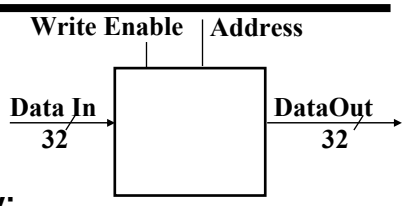


Clocking Methodology



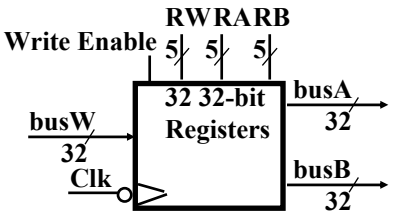
- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

Storage Element: 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
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Address valid => Data Out valid after Access time.
- Fig.5.4a show the abstraction of instruction memory and Fig.5.8a shows abstraction for data memory

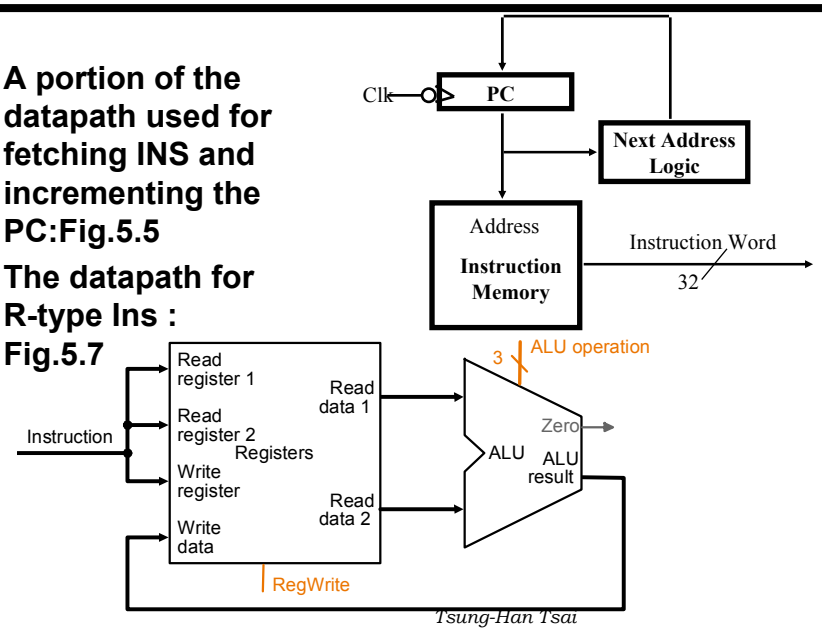
Storage Element: Register File



- 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)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after Access time.

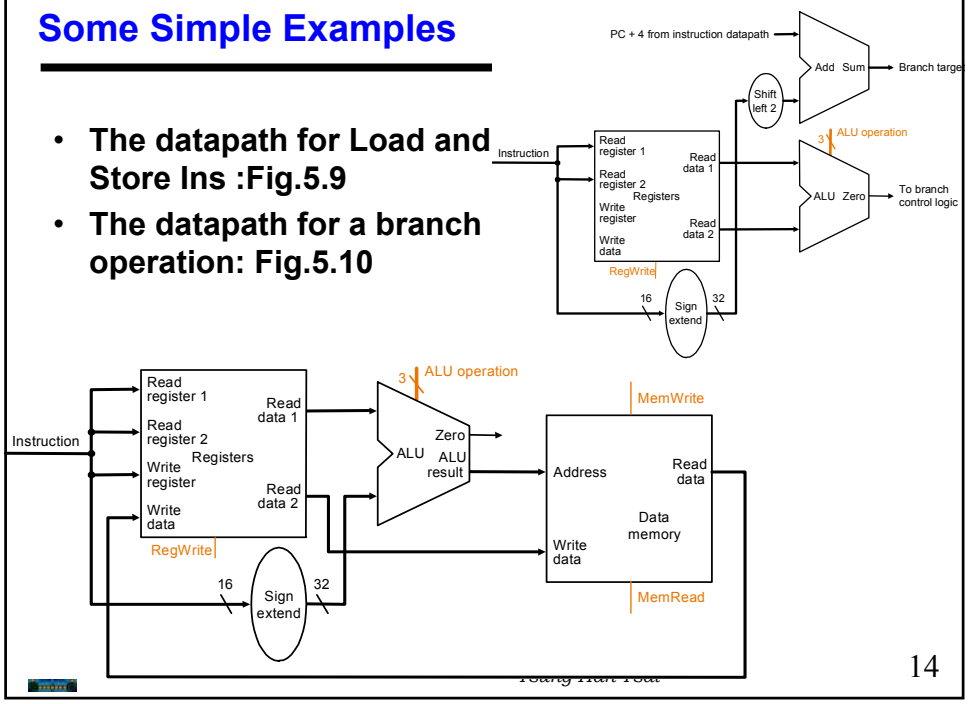
Some Simple Examples

- **A portion of the datapath used for fetching INS and incrementing the PC:Fig.5.5**
- **The datapath for R-type Ins : Fig.5.7**



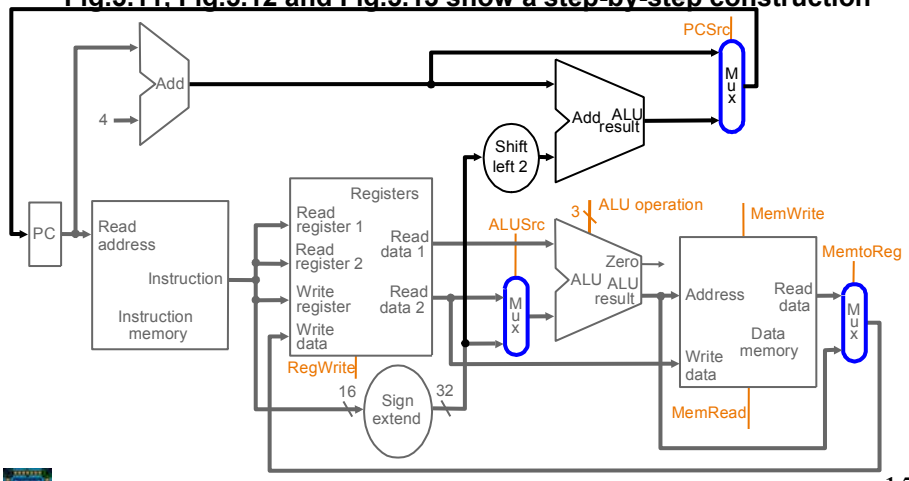
Some Simple Examples

- **The datapath for Load and Store Ins :Fig.5.9**
- **The datapath for a branch operation: Fig.5.10**



5.3 A Simple Implementation Scheme

- **Example: P351 ;Use multiplexors to select different data for an input of a block**
- **Fig.5.11, Fig.5.12 and Fig.5.13 show a step-by-step construction**



ALU Control

- **Selecting the operations to perform (ALU, read/write, etc.)**
- **Controlling the flow of data (multiplexor inputs)**
- **Information comes from the 32 bits of the instruction**
- **Example:**

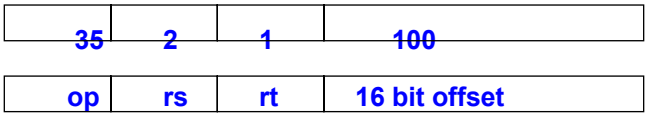
  add $8, $17, $18 Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- **ALU's operation based on instruction type and function code**

ALU Control

- Example: lw $1, 100($2)



- ALU control input

```
000   AND
001   OR
010   add
110   subtract
111   set-on-less-than
```

- Why is the code for subtract 110 and not 011?

Control

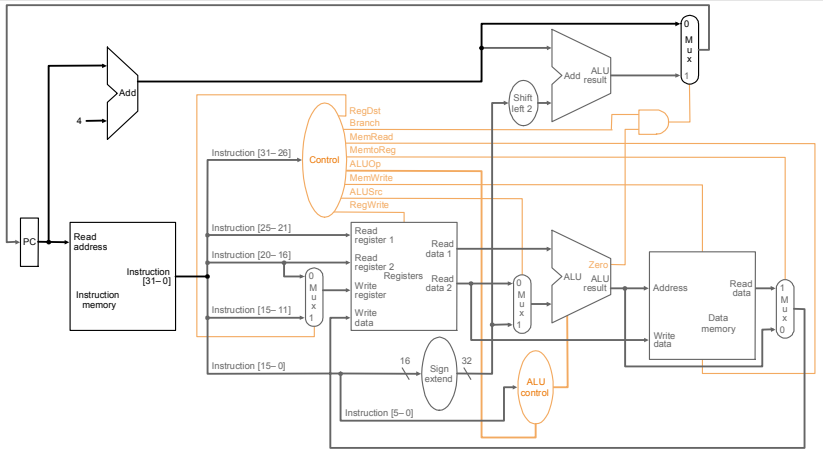
- Must describe hardware to compute 3-bit ALU conrol input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic
  - function code for arithmetic

ALUOp computed from instruction type

- Describe it using a truth table (can turn into gates):

Instruction	ALUOp		Funct field					
opcode	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
LW or SW	0	0	X	X	X	X	X	X
Branch Equal	0	1	X	X	X	X	X	X
R-type(add)	1	X	X	X	0	0	0	0
R-type(subtract)	1	X	X	X	0	0	1	0
R-type(AND)	1	X	X	X	0	1	0	0
R-type(OR)	1	X	X	X	0	1	0	1
R-type(set on less)	1	X	X	X	1	0	1	0

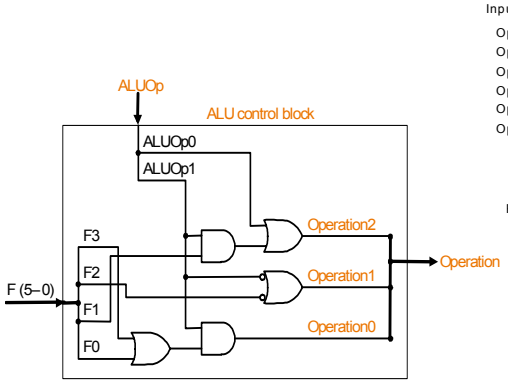
Control

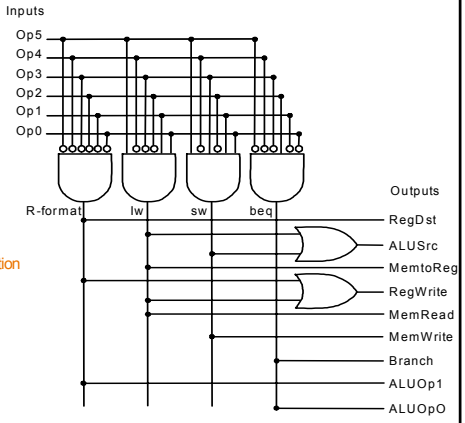


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Control Circuits

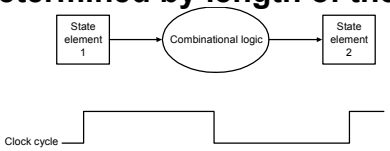
- **Simple combinational logic (truth tables)**



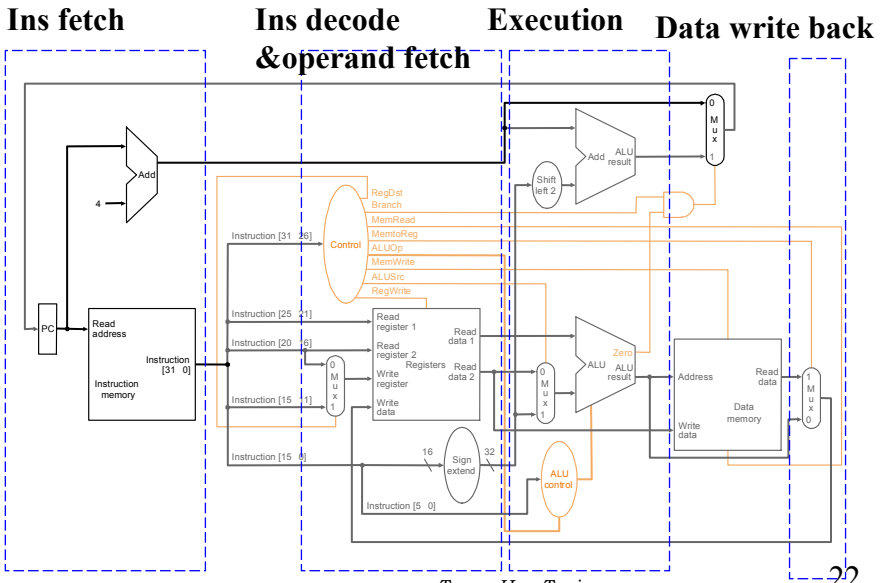


Our Simple Control Structure

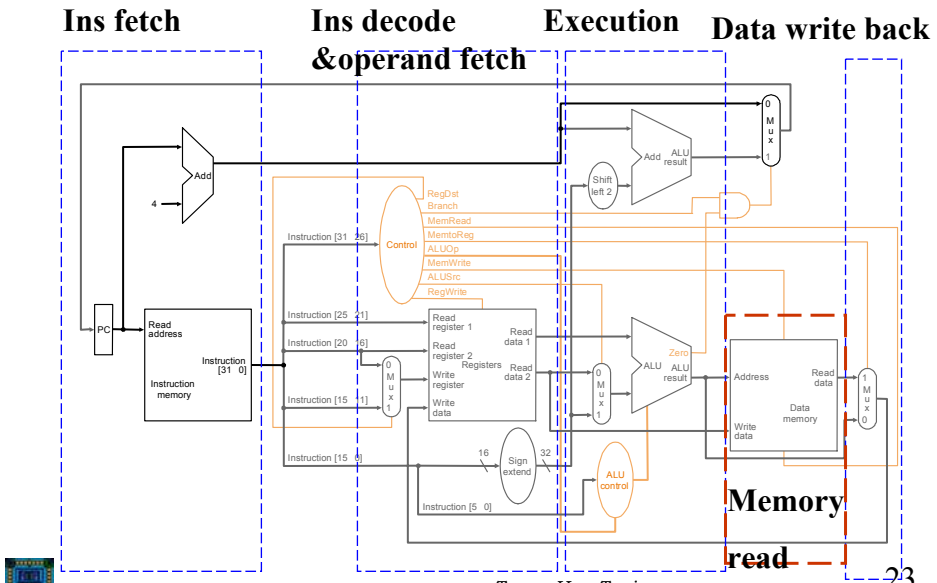
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



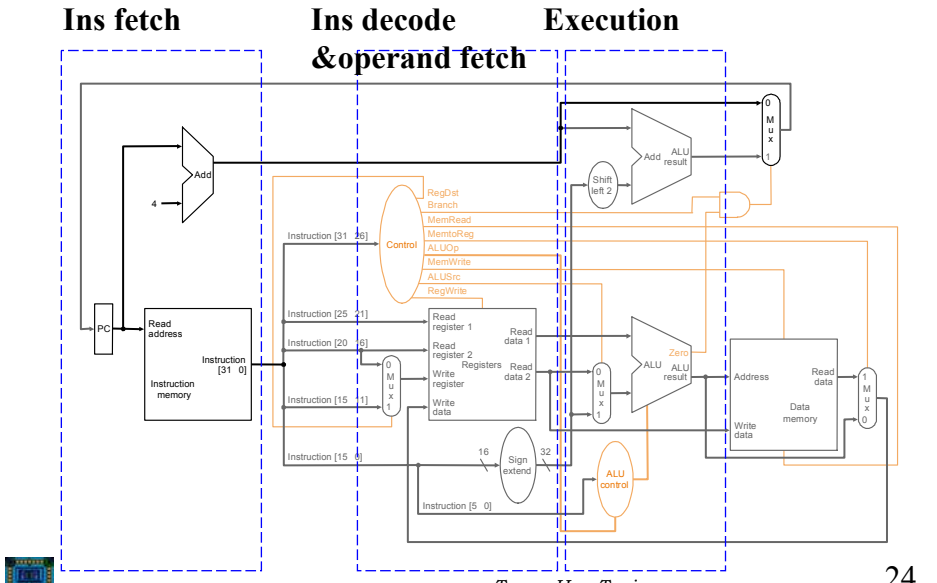
The four steps of R-type Ins (Fig.5.21,22,23,24)



The five steps of lw Ins (Fig.5.25)

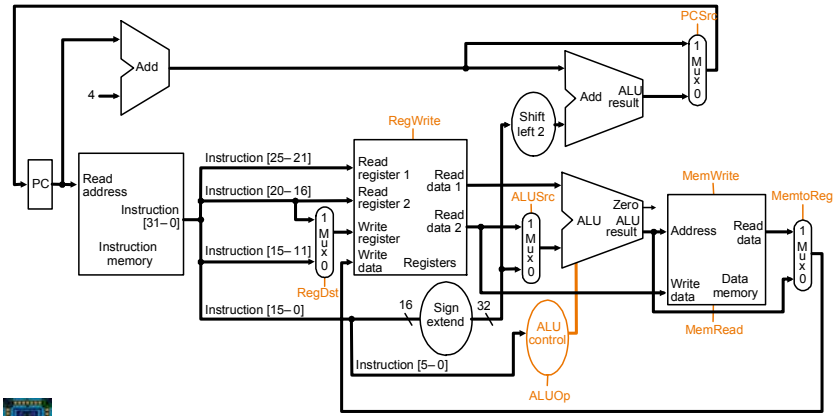


The three steps of beq-type Ins (Fig.5.26)



Single Cycle Implementation

- Calculate cycle time assuming negligible delays in line and other small blocks except:
  - memory (2ns), ALU and adders (2ns), register file (1ns)

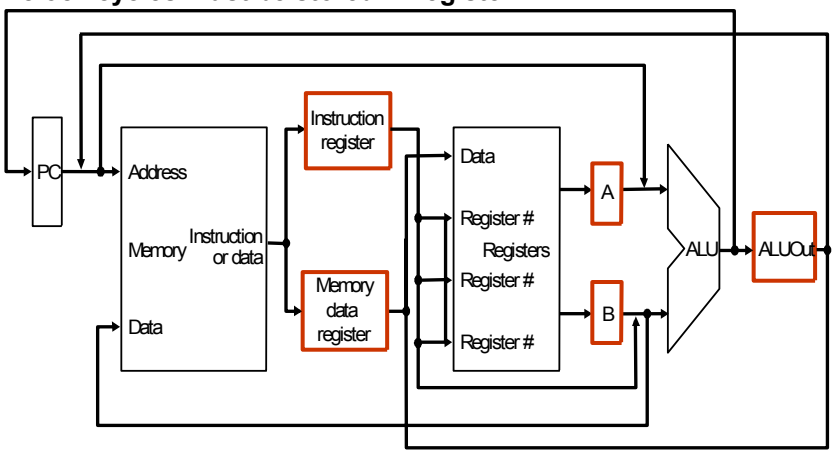


Where we are headed: 5.4 Multicycle Approach

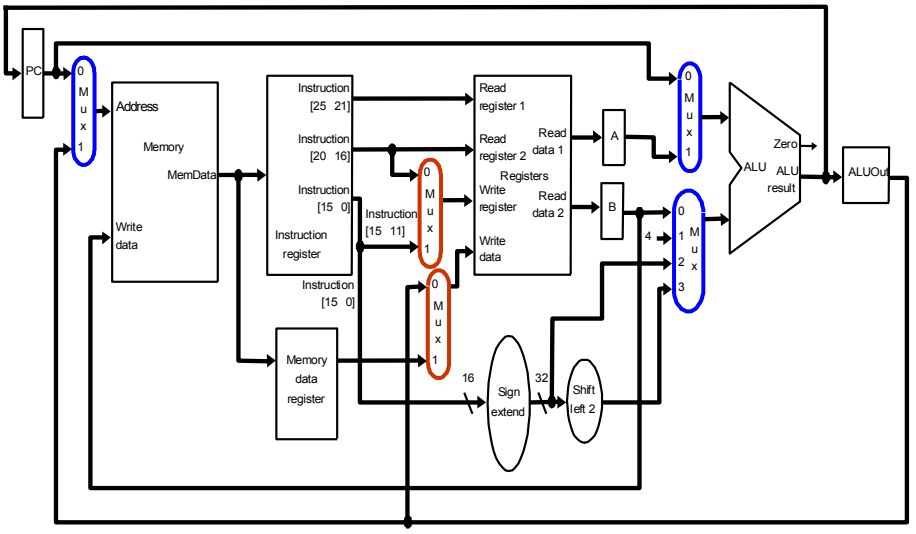
- Single Cycle Problems:
  - What if we had a more complicated instruction like floating point operation? ->The cycle time must consider worst case
  - wasteful of area , Example : one ALU + 2 adders
- Multicycle Solution:
  - use a "smaller" cycle time
    - break up the instructions into steps, each step takes a cycle
    - balance the amount of work to be done
    - have different instructions take different numbers of cycles
  - restrict each cycle to use only one major functional unit
    - At the end of a cycle stores values for use in later cycles (easiest thing to do)
    - A "multicycle datapath": reusing functional units (1)ALU used to compute address and to increment PC (2)Memory used for instruction and data
    - Must introduce additional "internal" registers

Multicycle Approach (Registers shall be added)

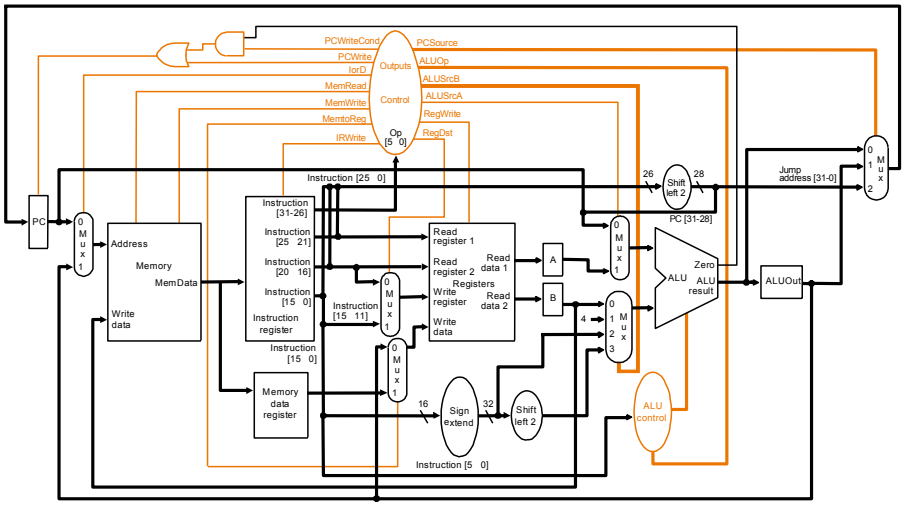
- Our control signals will not be determined solely by instruction
- At the end of a clock cycle, all data that are used in subsequenct clock cycles must be stored in register



Multicycle Approach (Multiplexor is added)



Multicycle with necessary controls



Five Execution Steps

- **Instruction Fetch**
- **Instruction Decode and Register Fetch**
- **Execution, Memory Address Computation, or Branch Completion**
- **Memory Access or R-type instruction completion**
- **Write-back step**

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

Step 3 Execution (instruction dependent)

- ALU is performing one of three functions, based on instruction type
  - Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

  - R-type:

```
ALUOut = A op B;
```

  - Branch:

```
if (A==B) PC = ALUOut;
```

Step 4 R-type or memory-access

- Loads and stores access memory

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Step 5 Write-back step

- `Reg[IR[20-16]]= MDR;`

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Simple Questions

- How many cycles will it take to execute this code?

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label     #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

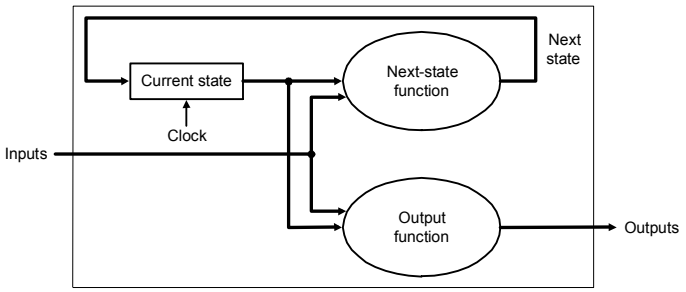
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Review: finite state machines

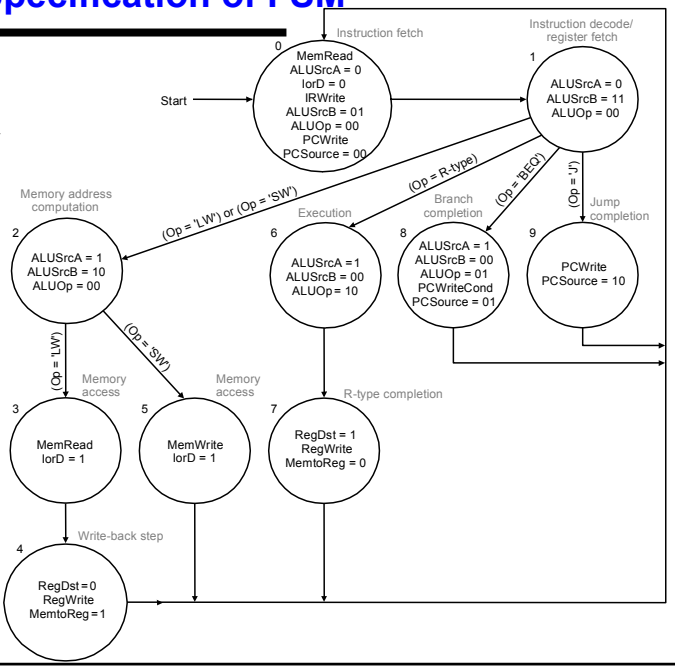
- **Finite state machines:**
  - **a set of states and**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and possibly input)**



  - **We'll use a Moore machine (output based only on current state)**

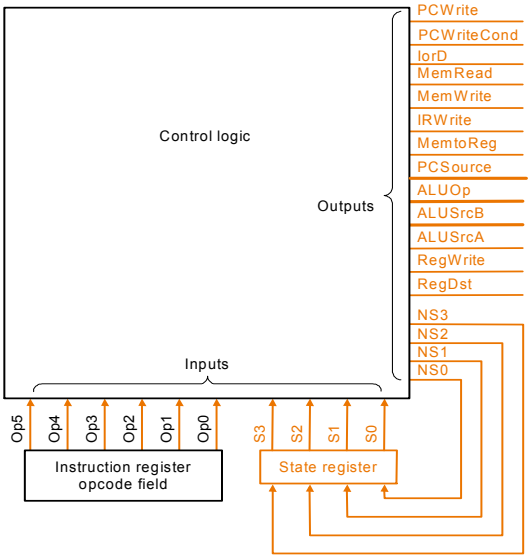
Graphical Specification of FSM

- **How many state bits will we need?**



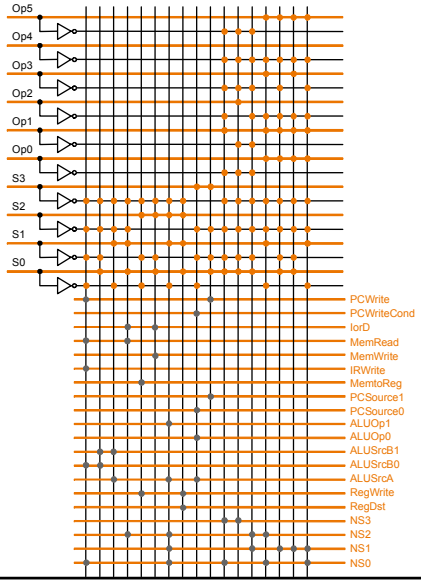
Finite State Machine for Control

- **Implementation:**



Programmable Logic Array (PLA) Implementation

- **If I picked a horizontal or vertical line could you explain it?**



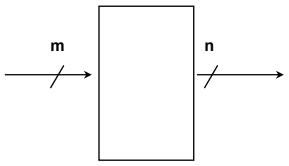
For example:

PCWrite=

$$\overline{s_0}\,\overline{s_1}\,\overline{s_2}\,\overline{s_3} + s_0\,\overline{s_1}\,\overline{s_2}\,\overline{s_3}$$

ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



Address Data

0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

Input Output

m is the "heigth", and n is the "width"

ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

ROM vs PLA

- **Break up the table into two parts**
  - **— 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM**
  - **— 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM**
  - **— Total: 4.3K bits of ROM**
- **PLA is much smaller**
  - **— can share product terms**
  - **— only need entries that produce an active output**
  - **— can take into account don't cares**
- **Size is (#inputs ´ #product-terms) + (#outputs ´ #product-terms)**
  For this example = (10x17)+(20x17) = 460 PLA cells
- **PLA cells usually about the size of a ROM cell (slightly bigger)**

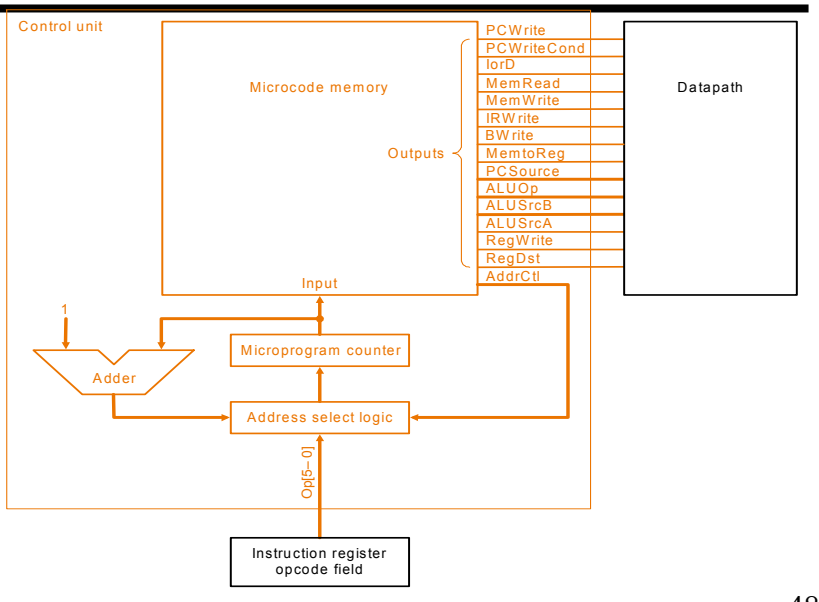
5.5 Microprogramming:Simplifying Control Design

- Control is the hard part of processor design
  - ?Datapath is fairly regular and well-organized
  - ?Memory is highly regular
  - ?Control is irregular and global
- Microprogramming: Designing the control as a program that implements the machine instructions in terms of simpler microinstructions at the level of register transfer operations
  - Think of the control operation as to issue set of control signals (in sequence) that must be asserted in a state as an "microinstruction" to be executed by the datapath.
  - Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction
- A microinstruction is represented as a sequence of fields whose functions are related (example: Fig.5.44)
  - Usually placed in a ROM or a PLA (have address)
  - Three methods to choose next microINS; P.401
- A microprogram is a symbolic representation of the control that will be translated by a program to control logic

Microinstruction format

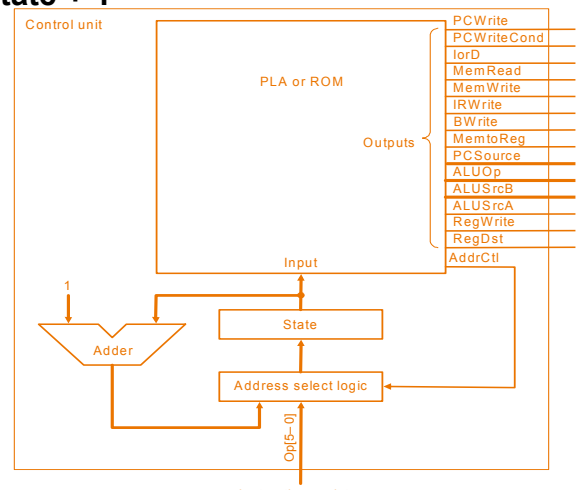
Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Microprogramming



Another Implementation Style

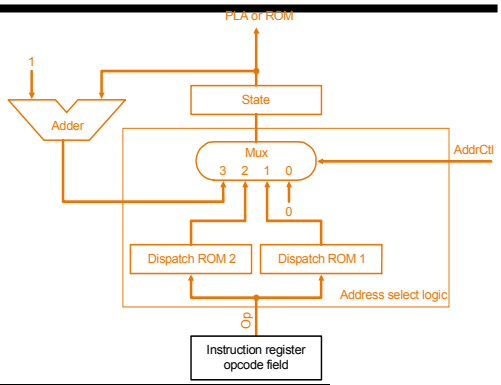
- **Complex instructions: the "next state" is often current state + 1**



Details

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Maximally vs. Minimally Encoded

- **No encoding:**
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- **Lots of encoding:**
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- **Historical context of CISC:**
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Microcode: Trade-offs

- **Specification Advantages:**
  - Easy to design and write
  - Design architecture and microcode in parallel
- **Implementation (off-chip ROM) Advantages**
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- **Implementation Disadvantages, SLOWER now that:**
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

The Big Picture