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

beq $1, $2, LABEL

- All instructions use the ALU after reading the registers

  Why? memory-reference? arithmetic? control flow?

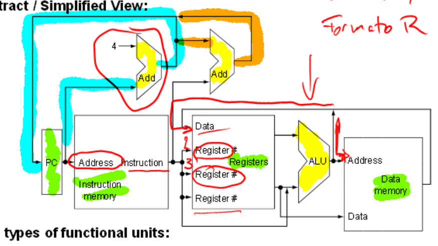
  Endereço Operando

More Implementation Details

- Abstract / Simplified View:

add $1,$2,$3

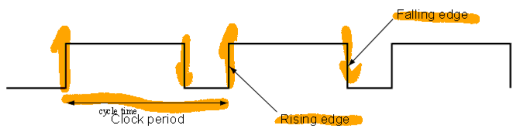
Formato R



Two types of functional units:
- elements that operate on data values (combinational)
- elements that contain state (sequential)

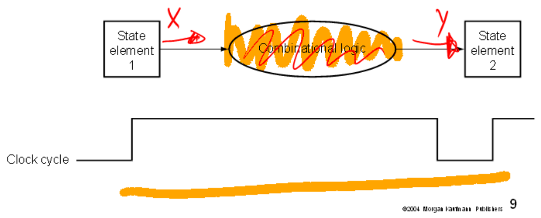
State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



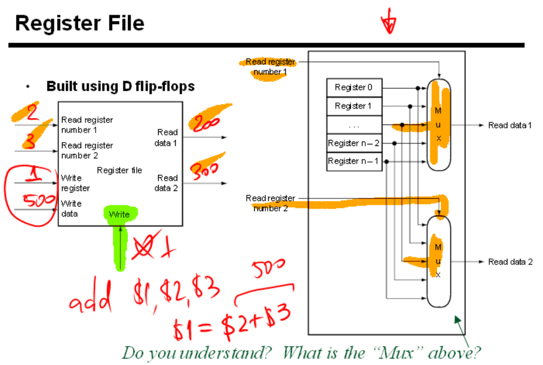
Our Implementation

- 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



Register File

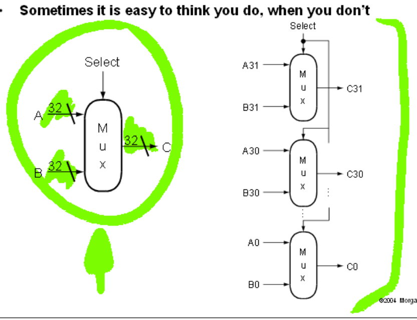
- Built using D flip-flops



Do you understand? What is the "Mux" above?

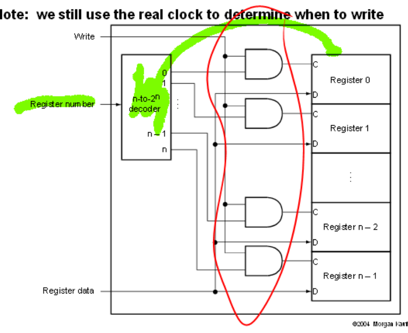
Abstraction

- Make sure you understand the abstractions!
- Sometimes it is easy to think you do, when you don't



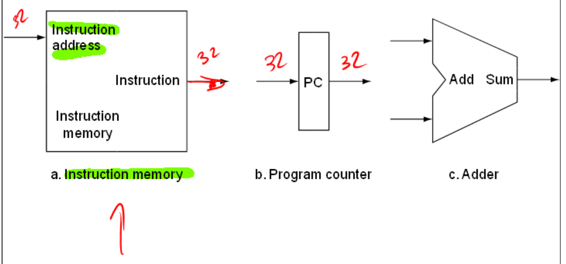
Register File

- Note: we still use the real clock to determine when to write



Simple Implementation

- Include the functional units we need for each instruction



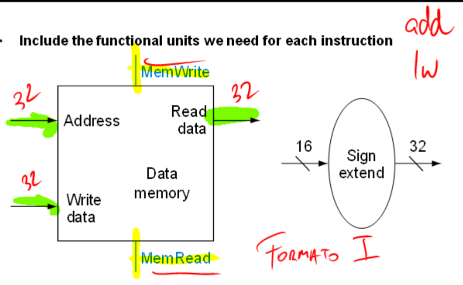
a. Instruction memory

b. Program counter

c. Adder

Simple Implementation

- Include the functional units we need for each instruction

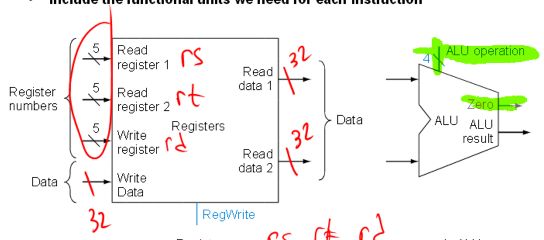


a. Data memory unit

b. Sign-extension unit

Simple Implementation

- Include the functional units we need for each instruction



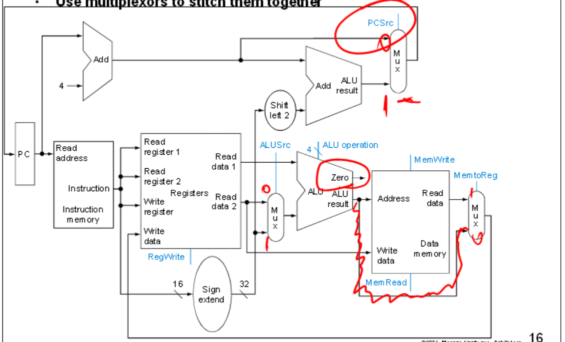
a. Registers

rs, rt, rd

b. ALU

Building the Datapath

- Use multiplexors to stitch them together



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

Control

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100
op	rs	rt	16 bit offset

- ALU control input

```
0000   AND
0001   OR
0010   add
0110   subtract
0111   set-on-less-than
1100   NOR
```

- Why is the code for subtract 0110 and not 0011?

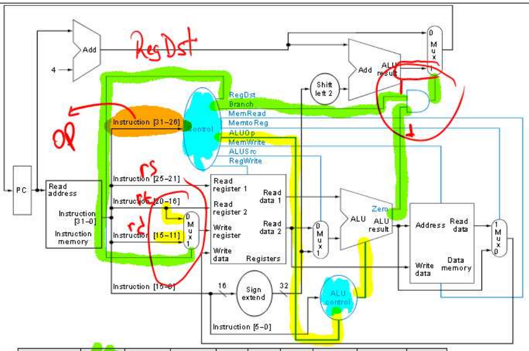
Control

- Must describe hardware to compute 4-bit ALU control input
  - given instruction type
    00 = lw, sw
    01 = beq,
    10 = arithmetic
    ALUOp computed from instruction type
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

instr	RegWrite	Alusrc	MemW	MemRead	MemtoReg	PC src
lw	1	1	0	1	1	0
sw	0	1	1	0	X	
add	1	0	0	0	0	
sub	1	0	0	0	0	
slt	1	0	0	0	0	
or	1	0	0	0	0	
beq	0	0	0	0	X	
j	0	0 X	0	0	X	



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

31 26 25 21 20 16 15 11 0

op	rs	rt	rd	sh	funct
6	5	5	5		

R

31 0

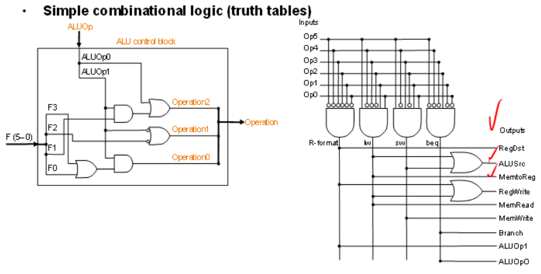
op	rs	rt	Imm

I

~~sw~~
lw

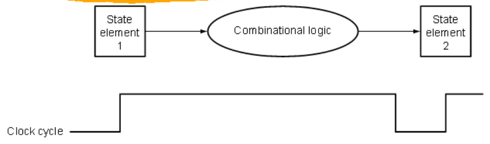
Control

- 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

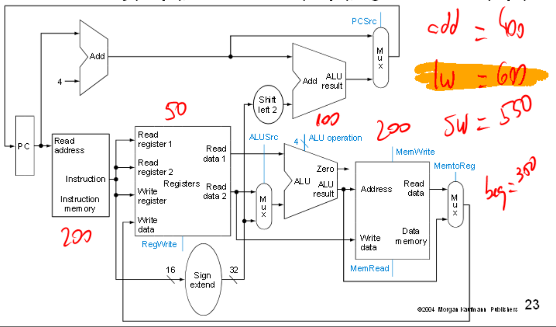


We are ignoring some details like setup and hold times

Single Cycle Implementation

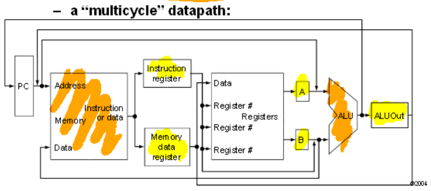
lw $1, 100($2)

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (100ps), register file access (50ps)



Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:



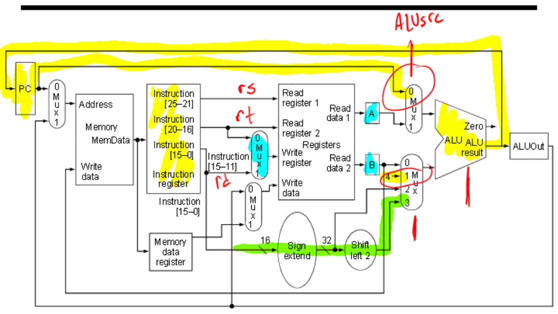
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional "internal" registers

Multicycle Approach



Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register. rd
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC. rs
  - New value is the sum ("op") of two registers. rt
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op
                          Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction. (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op
                          Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]` Busca
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]` } Leitura de Regs
  - `ALUOut <= A op B` Execution
  - `Reg[IR[15:11]] <= ALUOut` A op B
- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step (avoid unnecessary cycles) while also trying to share steps where possible (minimizes control, helps to simplify solution)
- Result: Our book's multicycle Implementation!

Five Execution Steps

- Instruction Fetch Busca
- Instruction Decode and Register Fetch

Genéricos

- 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:

```
        rs
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]]; rt          imm        IR
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)

$(\underbrace{PC+4}_{PC}) + \underline{imm} << 2$

Step 3 (instruction dependent)

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

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

- R-type: add, sub, slt

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

- Branch: beq

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

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR <= Memory[ALUOut];      lw
     or
Memory[ALUOut] <= B;        sw
```

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

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

Write-back step

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

Which instruction needs this?

lw

Summary:

Step name	Tipo R Action for R-type instructions	Memoria Action for memory-reference instructions	Beq Action for branches	J 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) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
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		

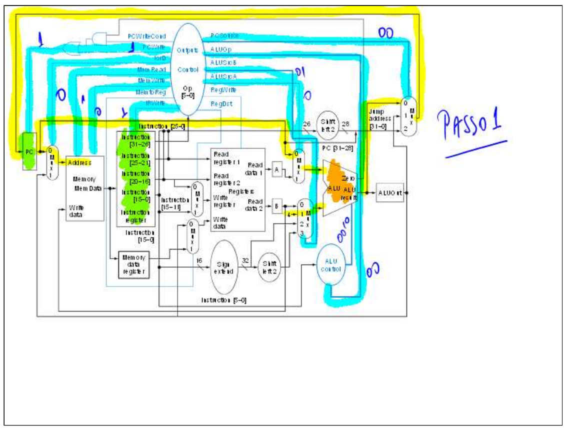
FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

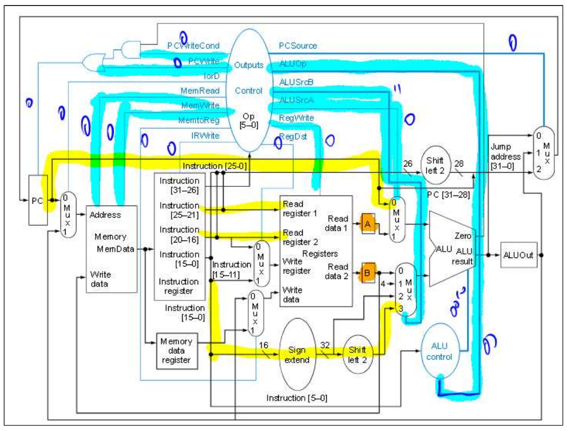
Simple Questions

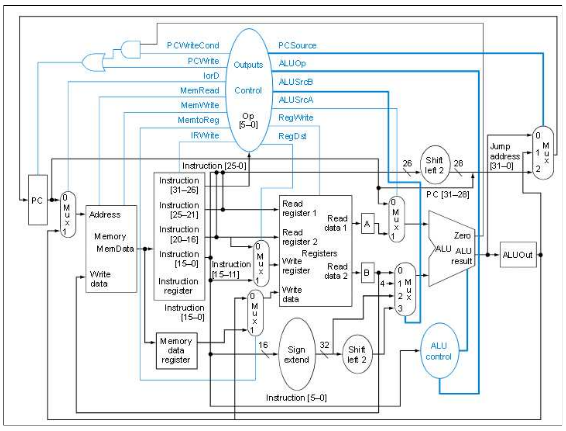
- How many cycles will it take to execute this code?

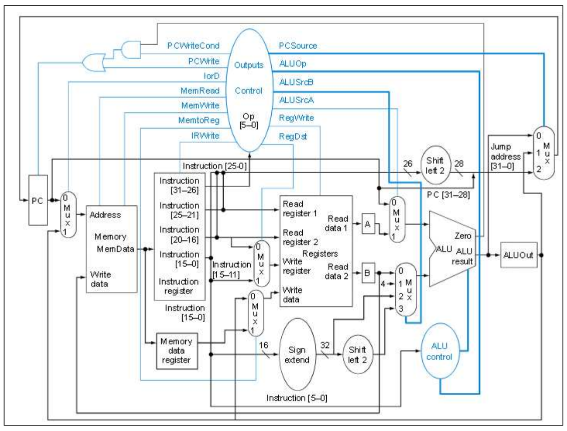
```
21          5  lw $t2, 0($t3)
            5  lw $t3, 4($t3)
            3  beq $t2, $t3, Label     <-   #assume not
            4  add $t5, $t2, $t3
            4  sw $t5, 8($t3)                   16
Label:         ...
```

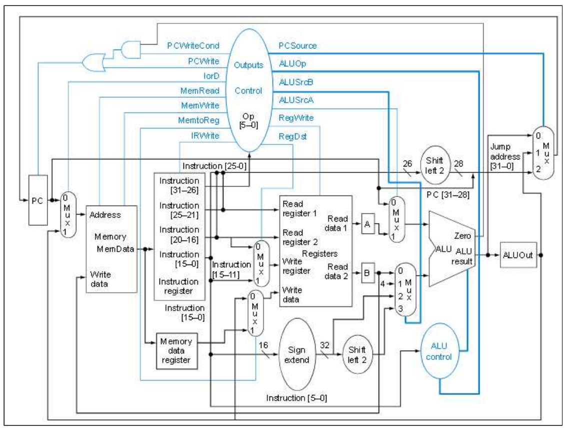
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

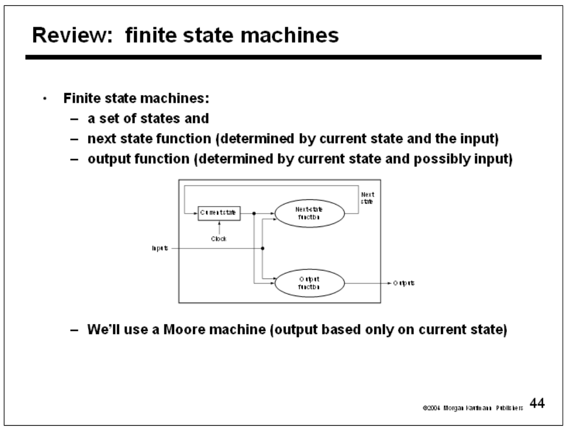






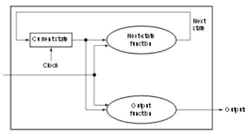






Review: finite state machines

- 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)

©2004 Morgan Kaufmann Publishers 44

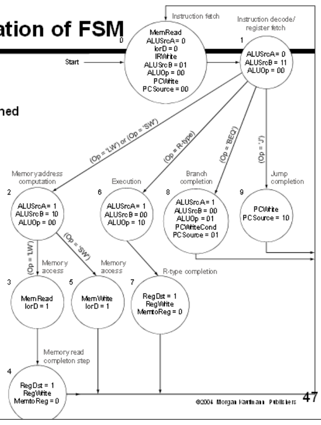
Implementing the Control

- 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

46

Graphical Specification of FSM

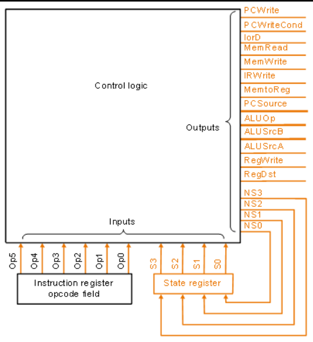
- Note:
  - don't care if not mentioned
  - asserted if name only
  - otherwise exact value
- How many state bits will we need?



47

Finite State Machine for Control

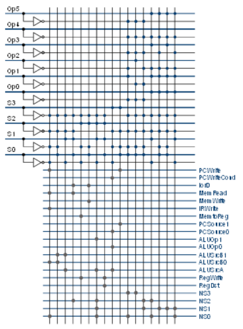
- Implementation:



48

PLA Implementation

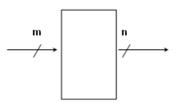
- If I picked a horizontal or vertical line could you explain it?



49

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.



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

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

50

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

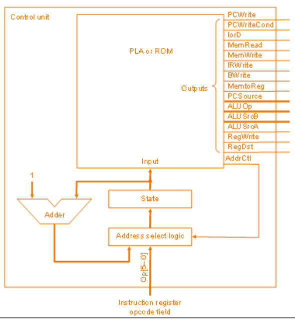
51

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 $\times$ #product-terms) + (#outputs $\times$ #product-terms)
  For this example = (10x17)+(20x17) = 510 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

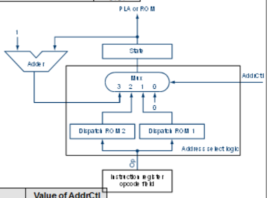
- Complex instructions: the "next state" is often current state + 1



Details

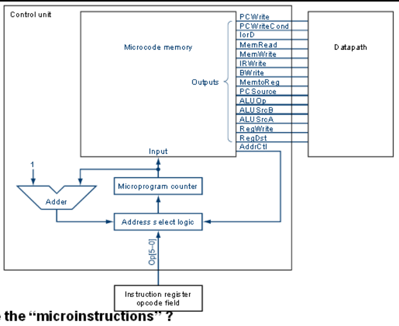
Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

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

Microprogramming



- What are the "microinstructions" ?

Microprogramming

- A specification methodology
  - – appropriate if hundreds of opcodes, modes, cycles, etc.
  - – signals specified symbolically using microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- *Will two implementations of the same architecture have the same microcode?*
- *What would a microassembler do?*

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, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 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.

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

- Distinction between specification and implementation is sometimes blurred
- 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

Chapter 5 Summary

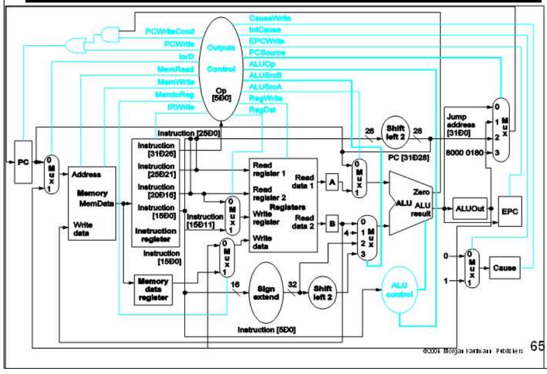
- If we understand the instructions…
  We can build a simple processor!
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation

Exceptions (MIPS)

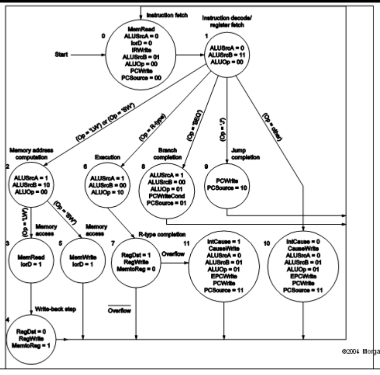
- **Exception**: An unscheduled event that disrupts program execution
- **Interrupt**: An exception that comes from outside of the processor

- **New Registers**: EPC and Cause
- **Two Methods**: Simple and Vectored

Datapath



Control


Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
3
MemRead
IorD = 1
Memory access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
$\overline{\text{Overflow}}$
4
RegDst = 0
RegWrite
MemtoReg = 1