Chapter 4

The Processor

Boolean Algebra & Gates

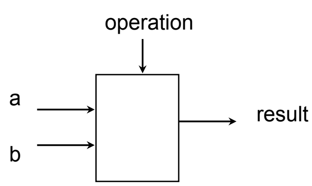
- Problem: Consider a logic function with three inputs: A, B, and C.

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

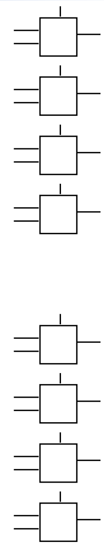
- Show the truth table for these three functions.
- Show the Boolean equations for these three functions.
- Show an implementation consisting of inverters, AND, and OR gates.

An ALU (arithmetic logic unit)

we'll just build a 1 bit ALU, and use 32 of them



op	a	b	res

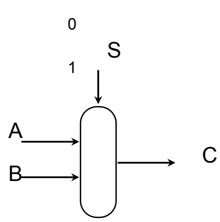


- Possible Implementation (sum-of-products):

Chapter 4 — The Processor — ‹#›

Review: The Multiplexor

Selects one of the inputs to be the output, based on a control input



Operation

C = A if S=0

C = B if S=1

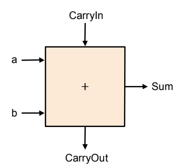
note: we call this a 2-input mux even though it has 3 inputs!

Lets build our ALU using a MUX:

Chapter 4 — The Processor — ‹#›

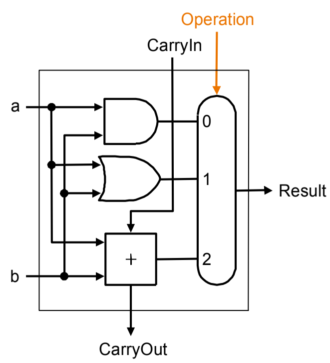
Different Implementations

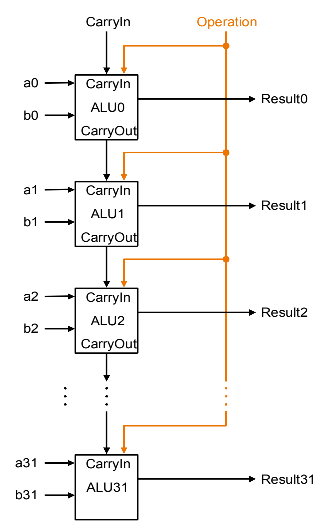
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

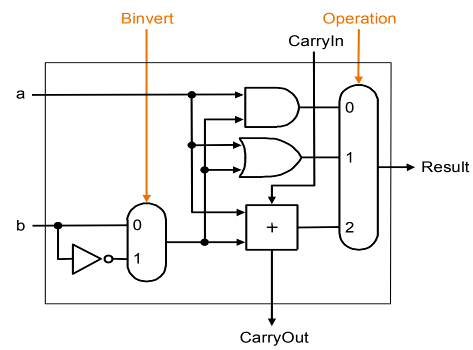
Building a 32 bit ALU





What about subtraction?

- Two's complement approach:
  - just negate b and add.

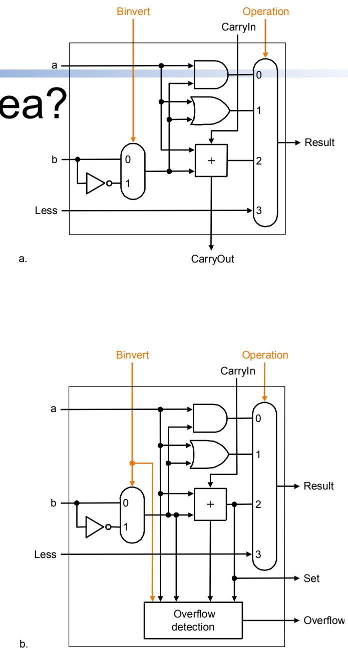


Tailoring the ALU to the MIPS

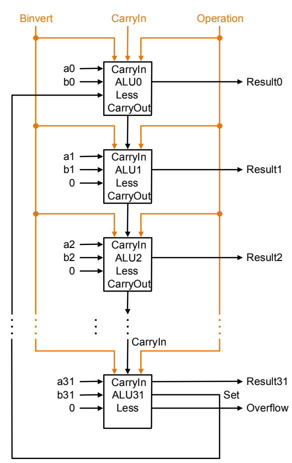
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

- Can we figure out the idea?



Chapter 4 — The Processor — ‹#›



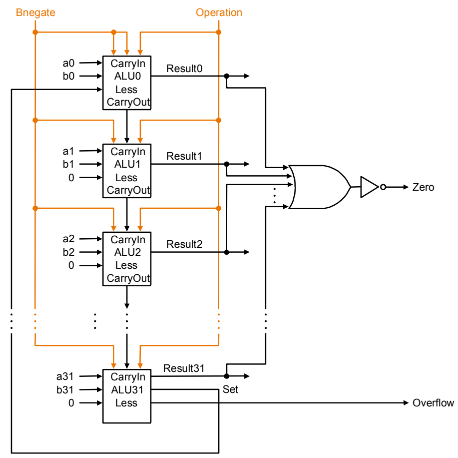
Chapter 4 — The Processor — ‹#›

Test for equality

Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•*Note: zero is a 1 when the result is zero!*



Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$

Not feasible! Why?

Chapter 4 — The Processor — ‹#›

Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i\, b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1c_1 \qquad c_2 =$

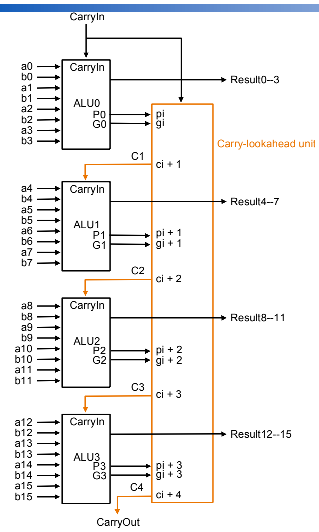
$c_3 = g_2 + p_2c_2 \qquad c_3 =$

$c_4 = g_3 + p_3c_3 \qquad c_4 =$

Feasible! Why?

Chapter 4 — The Processor — ‹#›

Use principle to build bigger adders



- Can’t build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

Introduction

§4.1 Introduction

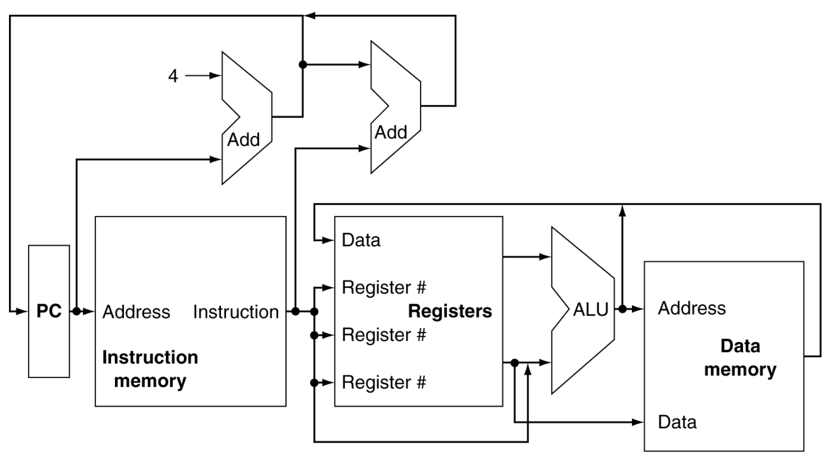
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine three MIPS implementations
  - A simple single cycle version
  - A multi-cycle version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

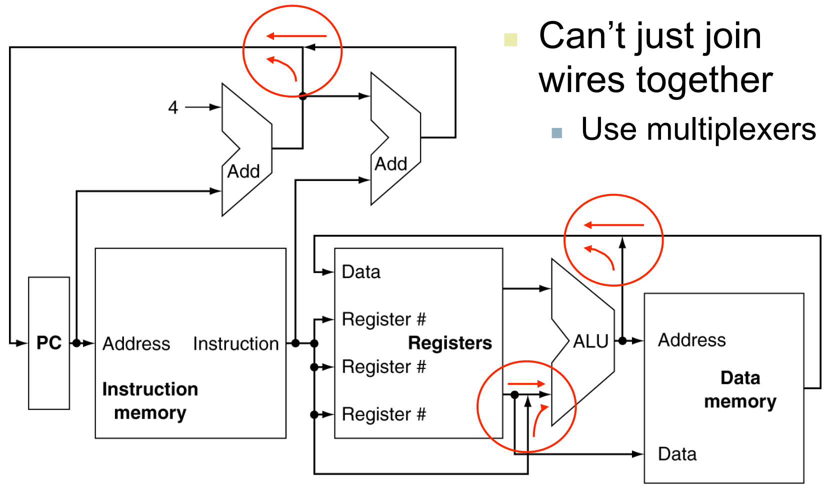
Chapter 4 — The Processor — ‹#›

CPU Overview (Back of the envelope design)



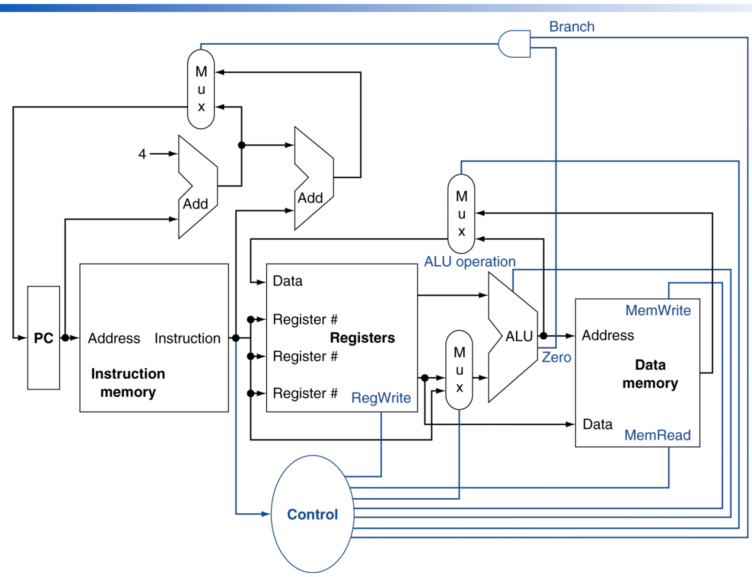
Chapter 4 — The Processor — ‹#›

Multiplexers



- Can't just join wires together
  - Use multiplexers

Control



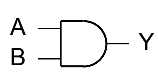
Logic Design Basics

§4.2 Logic Design Conventions

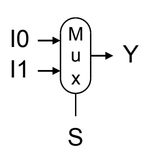
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

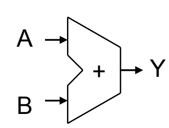
- AND-gate
  - Y = A & B



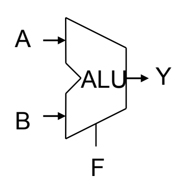
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

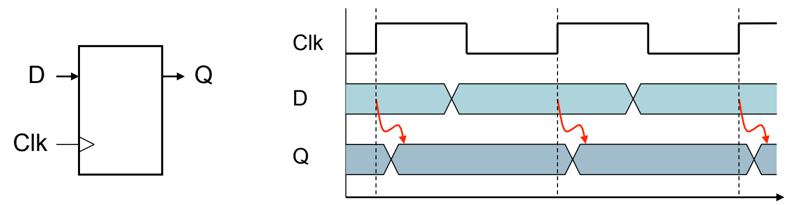


- Arithmetic/Logic Unit
  - Y = F(A, B)



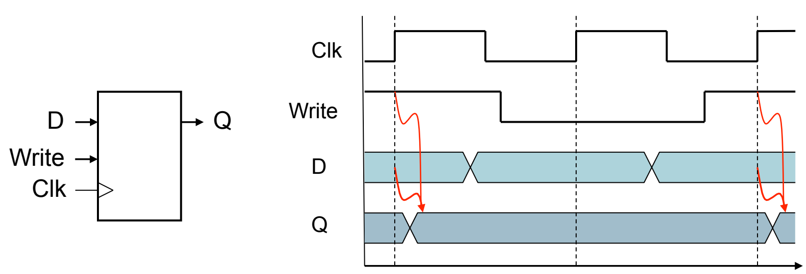
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



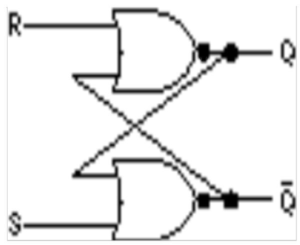
Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



An unclocked state element

- The set-reset latch
  - output depends on present inputs and also on past inputs



Latches and Flip-flops

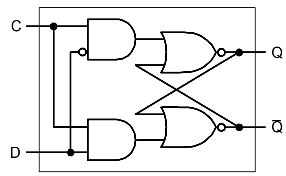
- Output is equal to the stored value inside the element
  (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

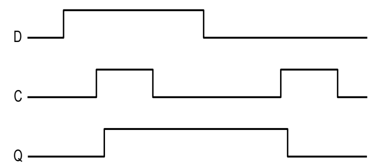
"logically true",
— could mean electrically low

A clocking methodology defines when signals can be read and written
— wouldn't want to read a signal at the same time it was being written

D-latch

- 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

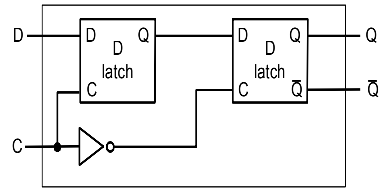


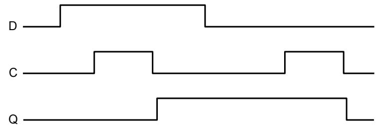


Chapter 4 — The Processor — ‹#›

D flip-flop

- Output changes only on the clock edge

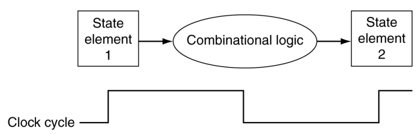


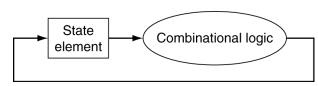


Chapter 4 — The Processor — ‹#›

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



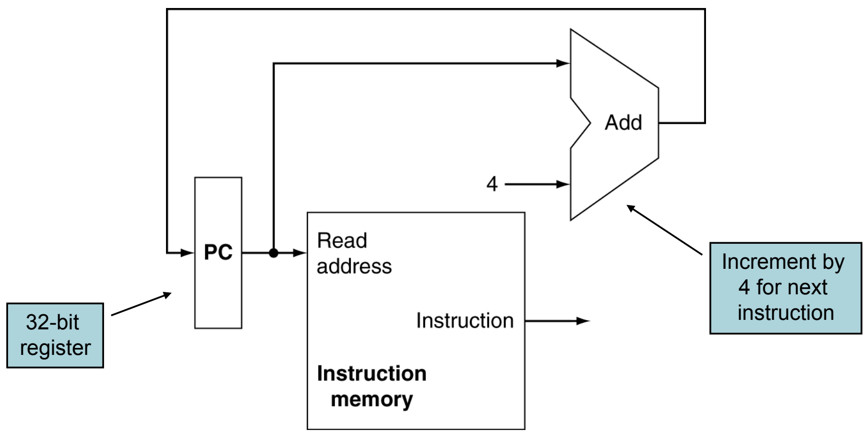


Building a Datapath

§4.3 Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

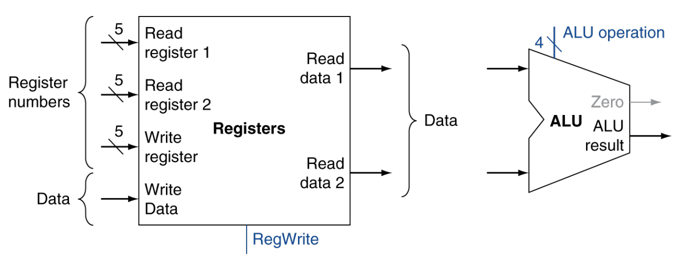
Instruction Fetch



Chapter 4 — The Processor — ‹#›

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

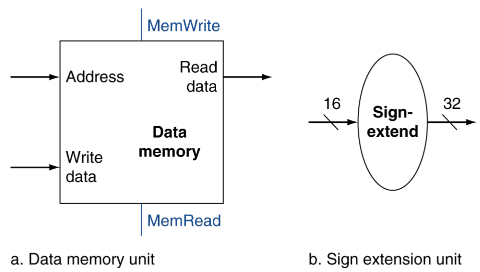


a. Registers b. ALU

Chapter 4 — The Processor — ‹#›

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



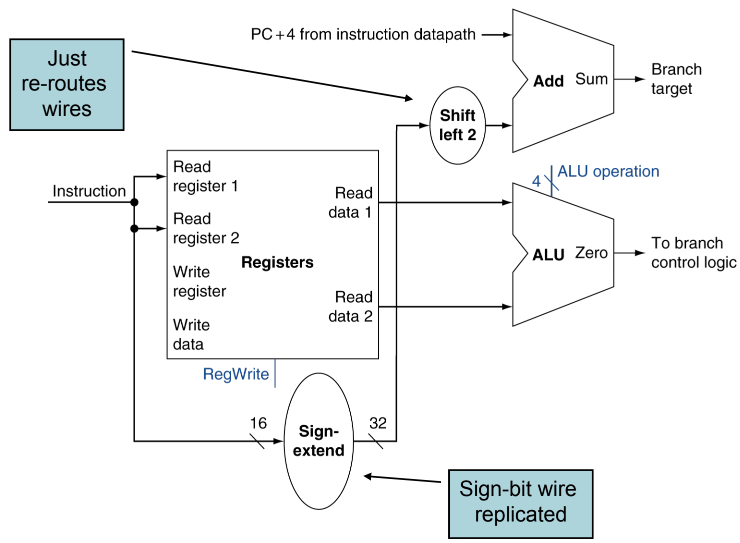
Chapter 4 — The Processor — ‹#›

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Chapter 4 — The Processor — ‹#›

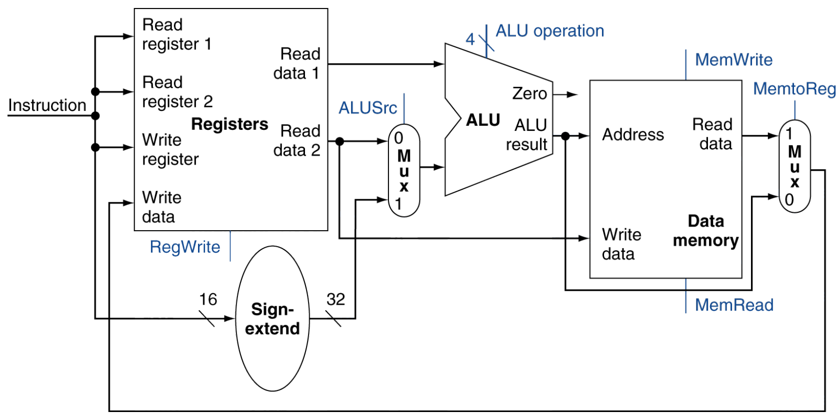
Branch Instructions



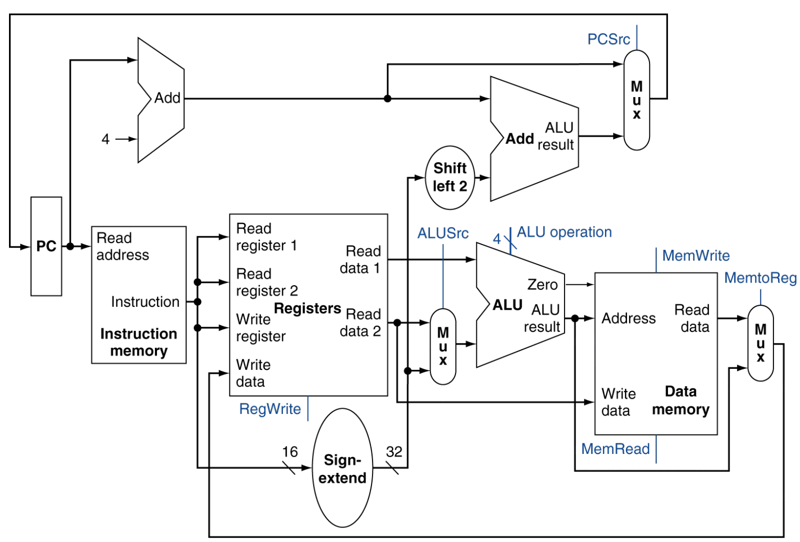
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

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

§4.4 A Simple Implementation Scheme

Chapter 4 — The Processor — ‹#›

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Chapter 4 — The Processor — ‹#›

The Main Control Unit

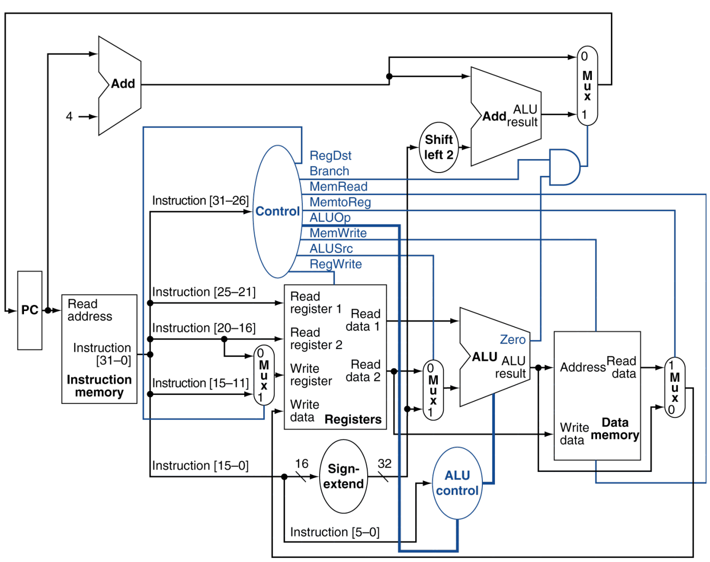
- Control signals derived from instruction

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

31:26	25:21	20:16	rd (15:11) / rt (20:16)	address (15:0)
opcode	always read	read, except for load	write for R-type and load	sign-extend and add

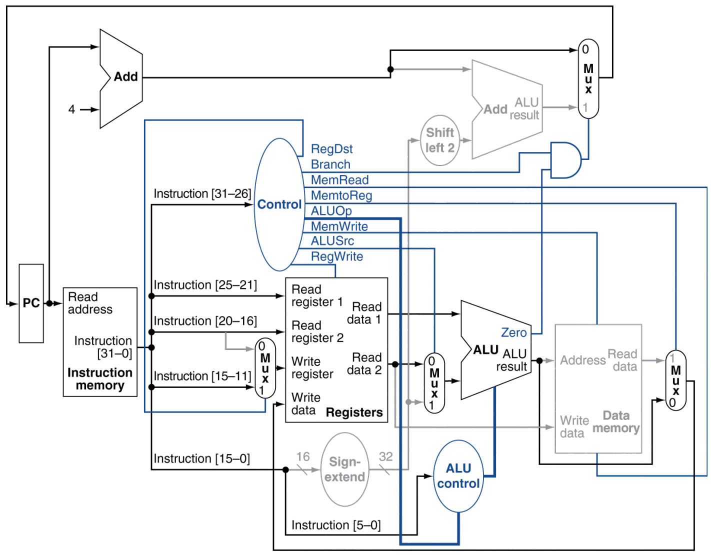
Chapter 4 — The Processor — ‹#›

Datapath With Control



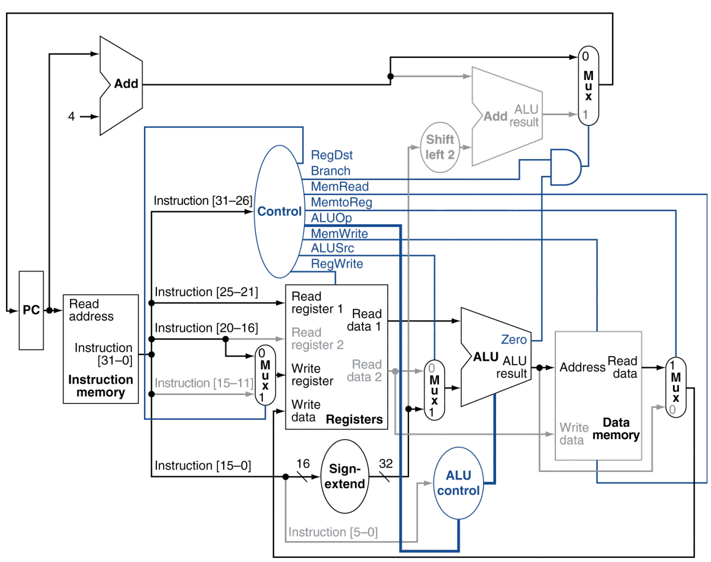
Chapter 4 — The Processor — ‹#›

R-Type Instruction



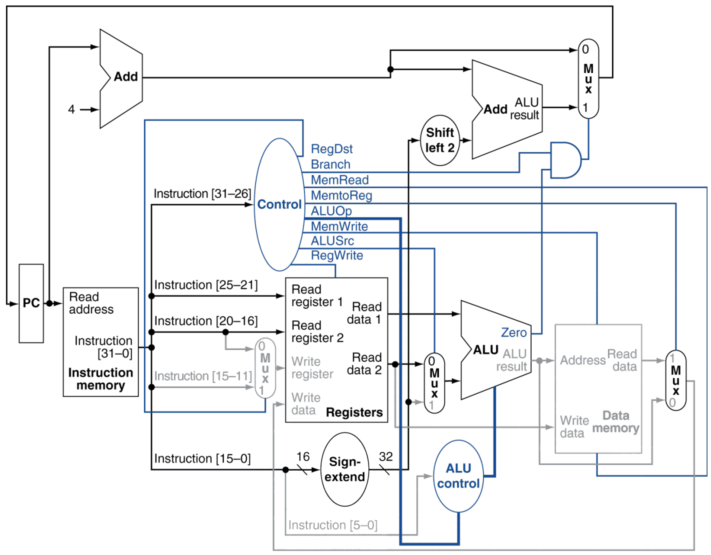
Chapter 4 — The Processor — ‹#›

Load Instruction



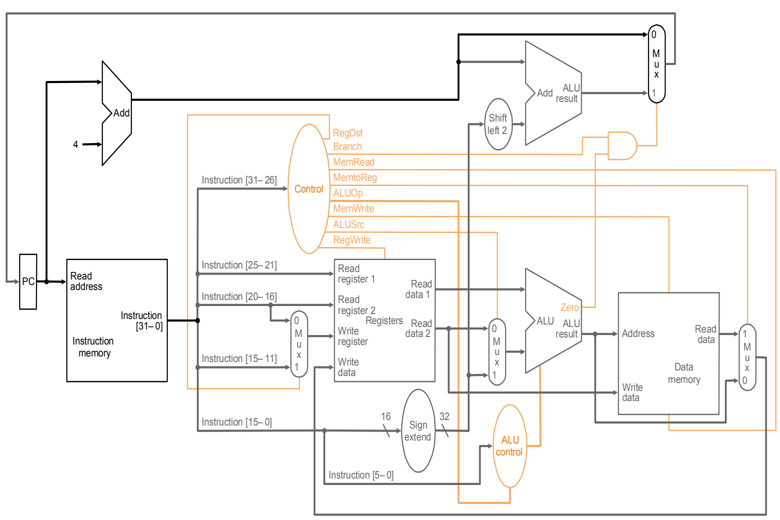
Chapter 4 — The Processor — ‹#›

Branch-on-Equal Instruction



Chapter 4 — The Processor — ‹#›

Single Cycle Implementation



Chapter 4 — The Processor — ‹#›

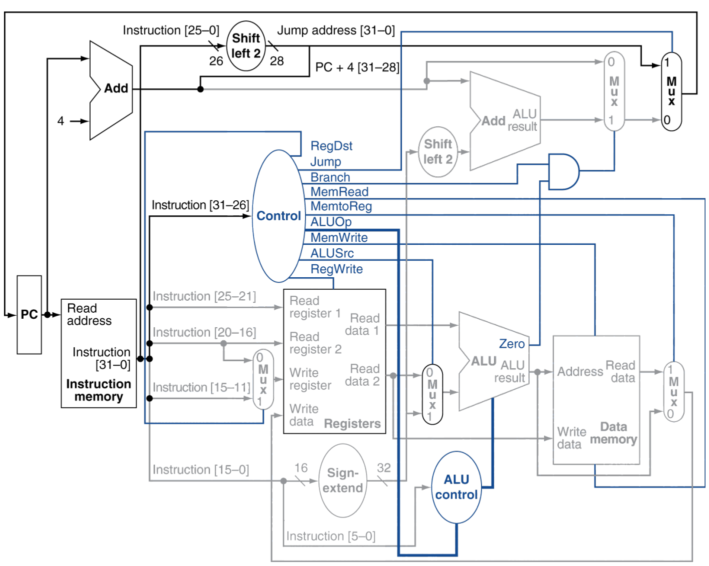
Implementing Jumps

Jump	2	address
	31:26	25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Chapter 4 — The Processor — ‹#›

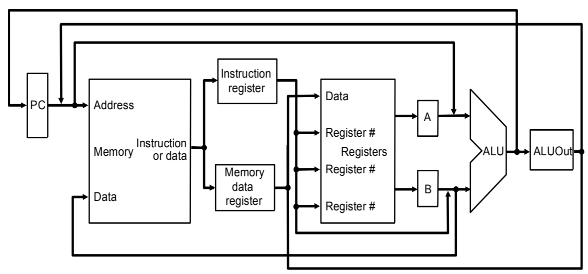
Datapath With Jumps Added



Chapter 4 — The Processor — ‹#›

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:

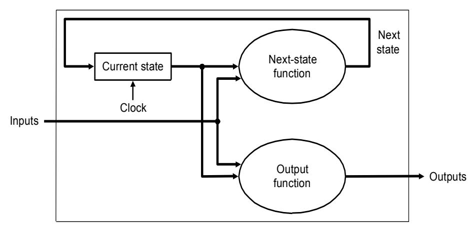


Multicycle Approach

- 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 soley by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Review: finite state machines

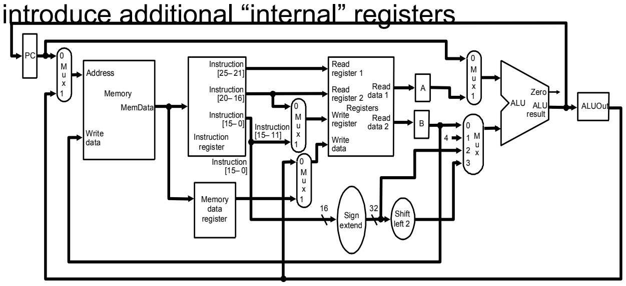
- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



Chapter 4 — The Processor — ‹#›

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



Chapter 4 — The Processor — ‹#›

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 (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

Chapter 4 — The Processor — ‹#›

Write-back step

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

What about all the other instructions?

Chapter 4 — The Processor — ‹#›

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		

Chapter 4 — The Processor — ‹#›

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?

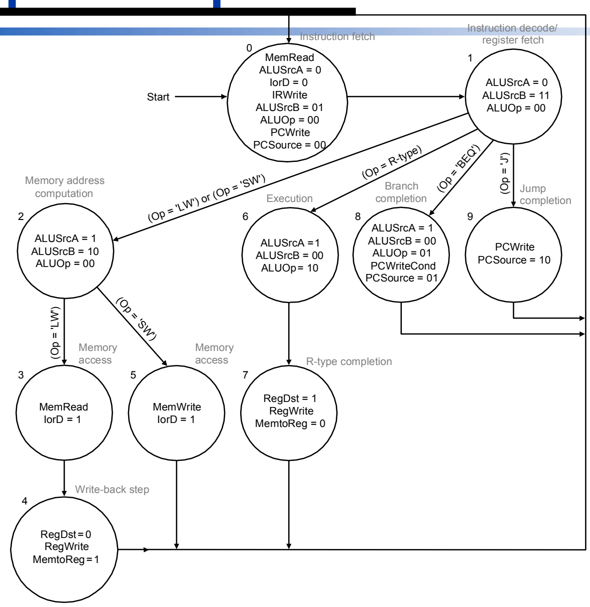
Chapter 4 — The Processor — ‹#›

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 acculumated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Chapter 4 — The Processor — ‹#›

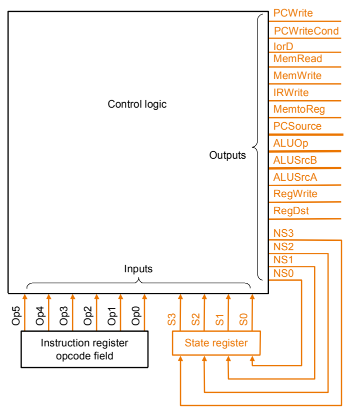
Graphical Specification of FSM



Chapter 4 — The Processor — ‹#›

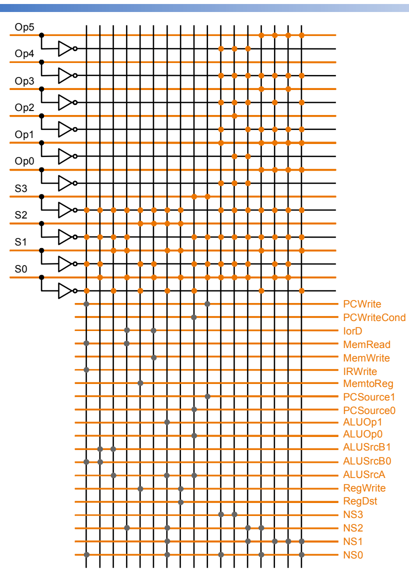
Finite State Machine for Control

- Implementation:



Chapter 4 — The Processor — ‹#›

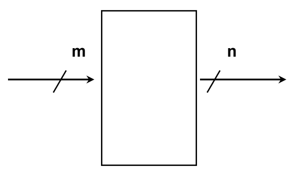
PLA Implementation



Chapter 4 — The Processor — ‹#›

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

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

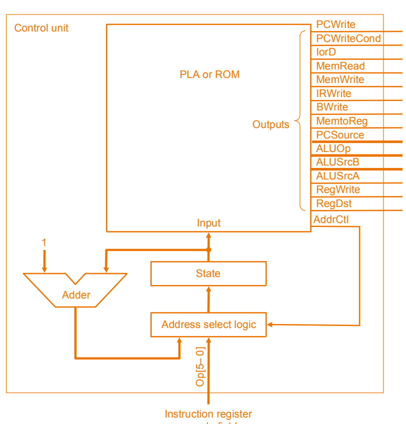
ROM vs PLA

- 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)

Chapter 4 — The Processor — ‹#›

Another Implementation Style

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

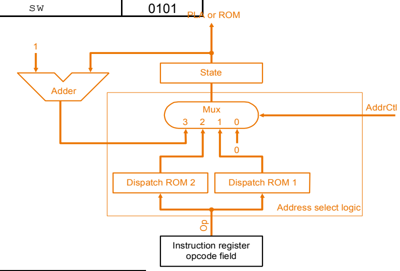


Chapter 4 — The Processor — ‹#›

Details

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

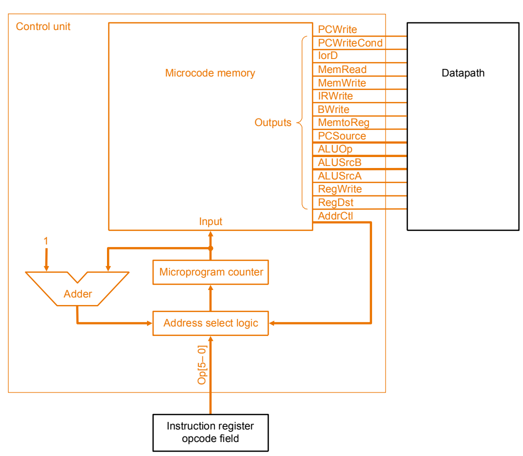
Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	`lw`	0011
101011	`sw`	0101



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

Chapter 4 — The Processor — ‹#›

Microprogramming



Chapter 4 — The Processor — ‹#›

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

Chapter 4 — The Processor — ‹#›

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.

Chapter 4 — The Processor — ‹#›

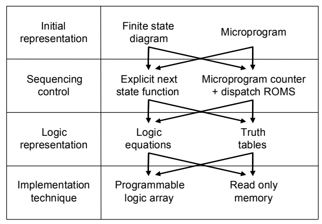
Maximally vs. Minimally Encoded

- 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

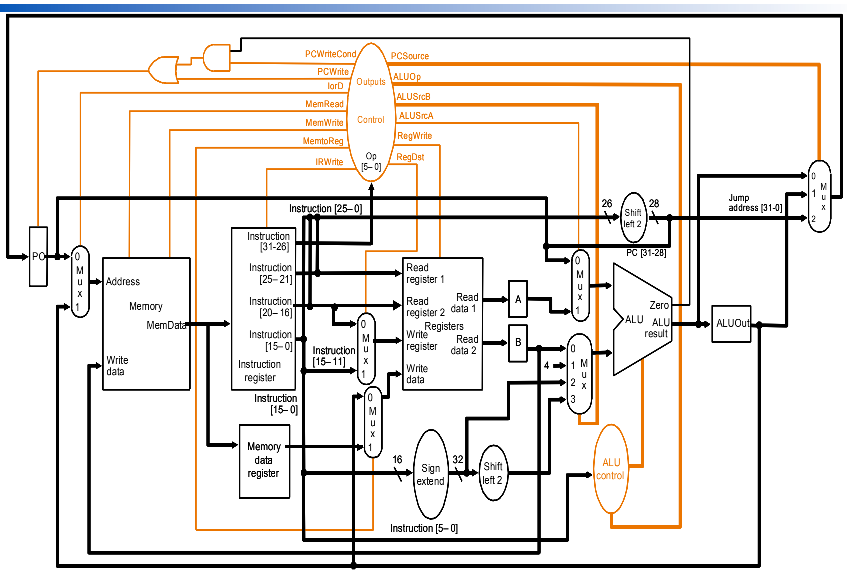
- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - 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

The Big Picture



Chapter 4 — The Processor — ‹#›

Final Multi-Cycle Implementation



Chapter 4 — The Processor — ‹#›