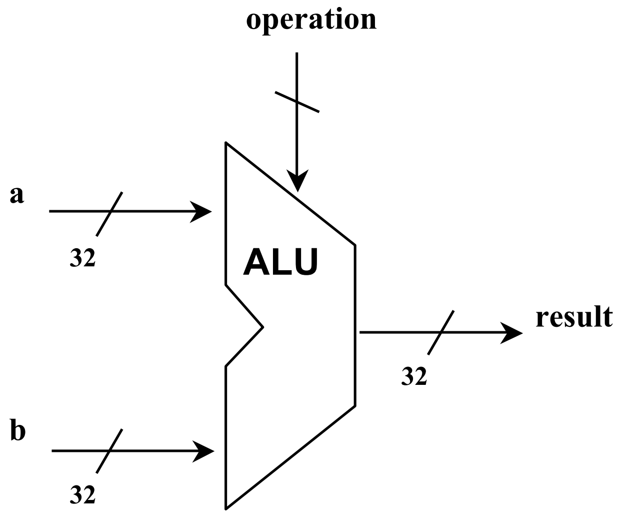
Lets Build a Processor

- Almost ready to move into chapter 5 and start building a processor
- First, let’s review Boolean Logic and build the ALU we’ll need
  (Material from Appendix B)



Review: 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

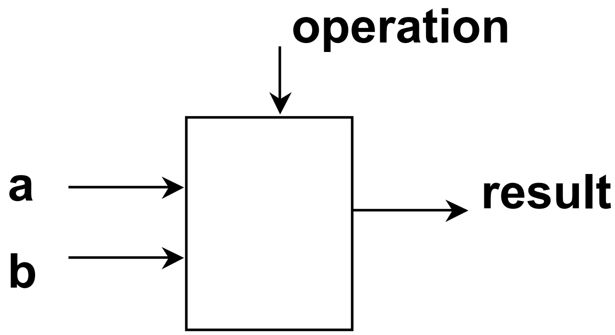
- Show the truth table for these three functions.

- Show the Boolean equations for these three functions.

- Show an implementation consisting of inverters, AND, and OR gates.

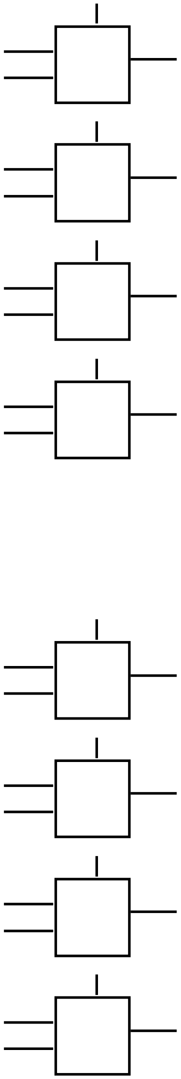
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them



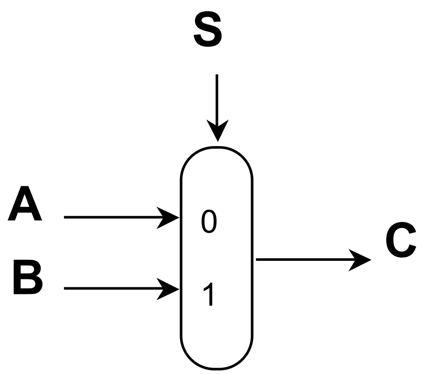
op	a	b	res

- Possible Implementation (sum-of-products):



Review: The Multiplexor

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

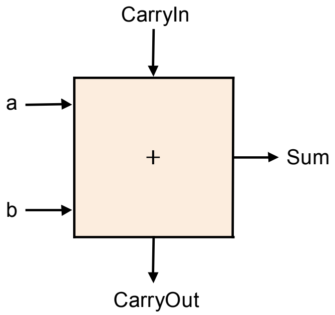


note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

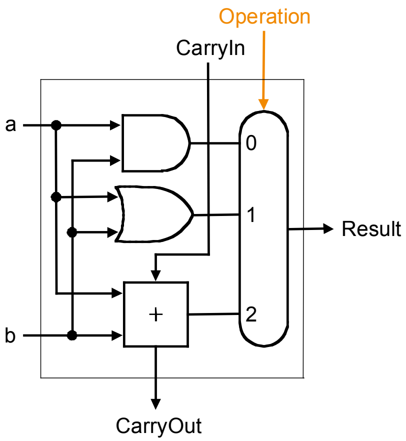
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

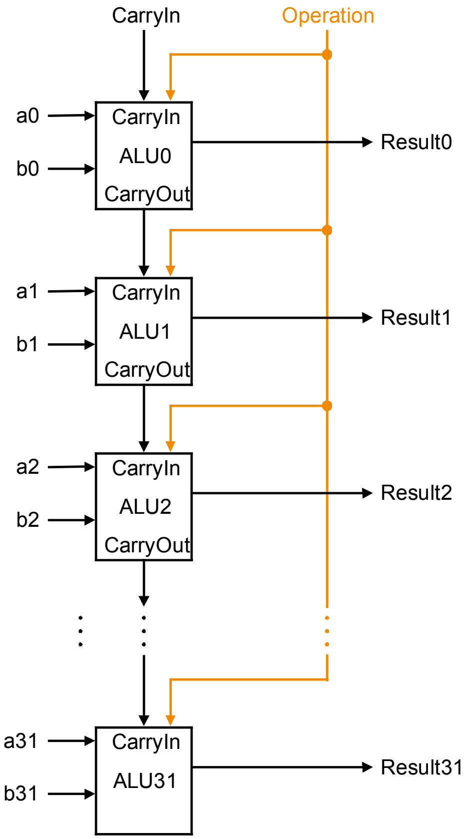


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

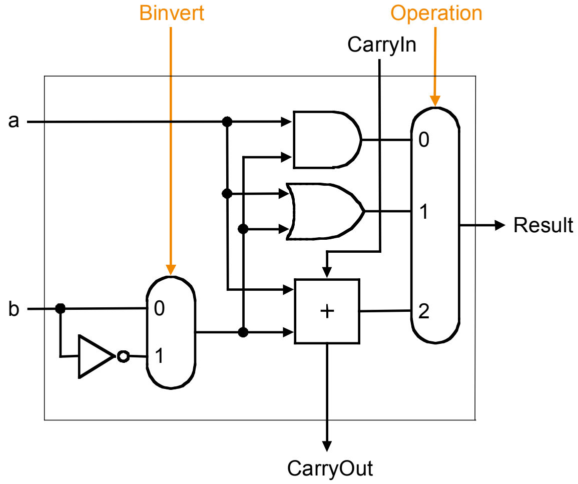




What about subtraction (a – b) ?

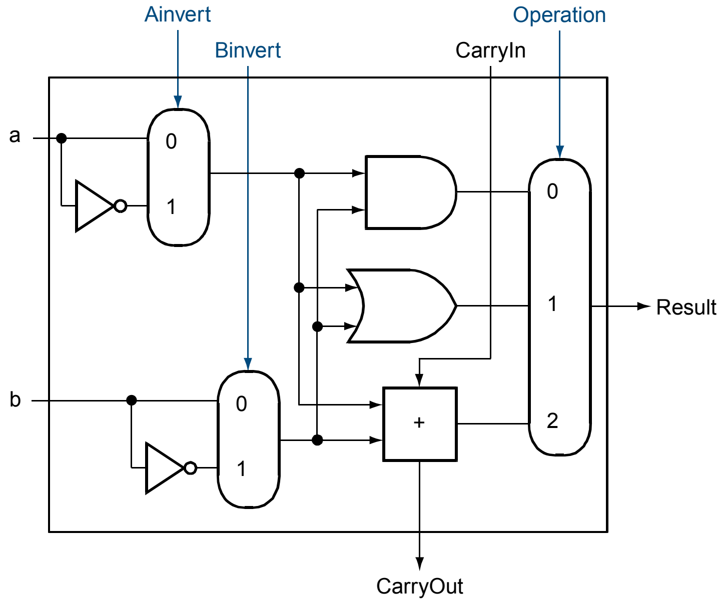
- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution:



Adding a NOR function

- Can also choose to invert a.   How do we get “a NOR b” ?

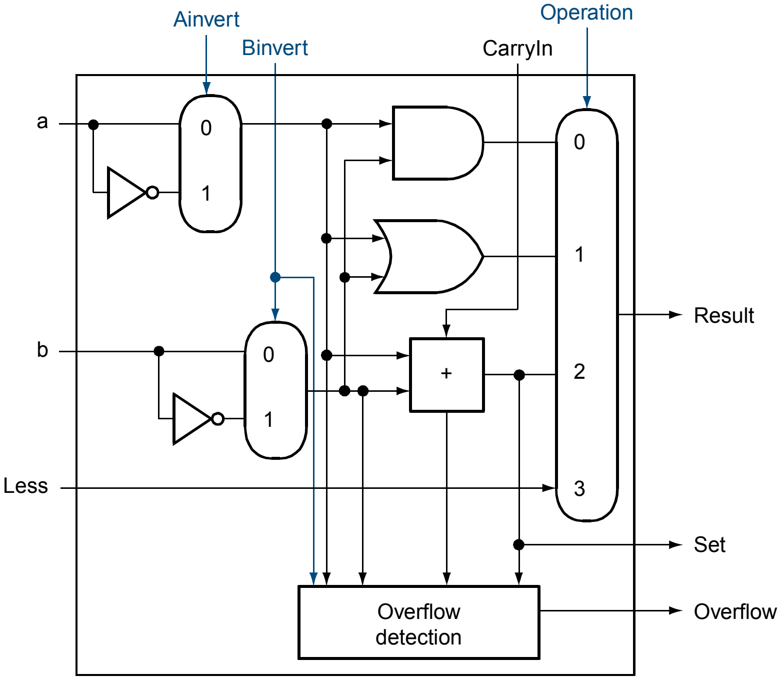


Tailoring the ALU to the MIPS

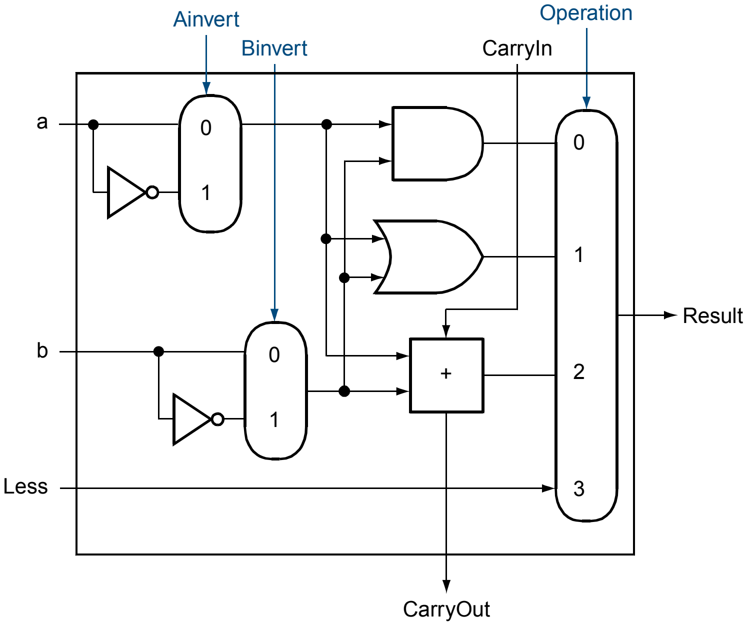
- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0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

Supporting slt

- **Can we figure out the idea?**

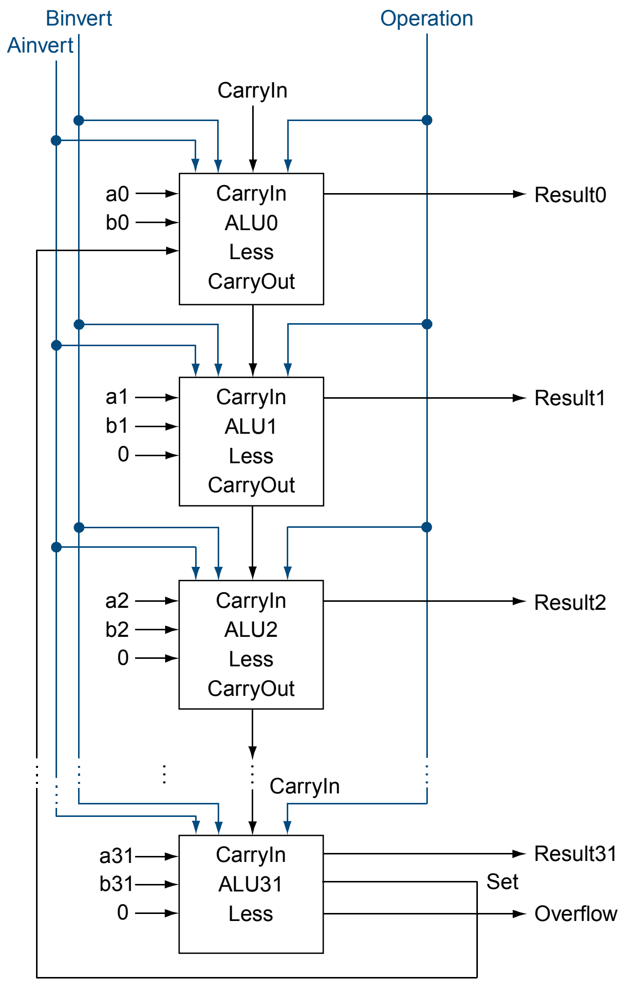


Use this ALU for most significant bit



all other bits

Supporting slt

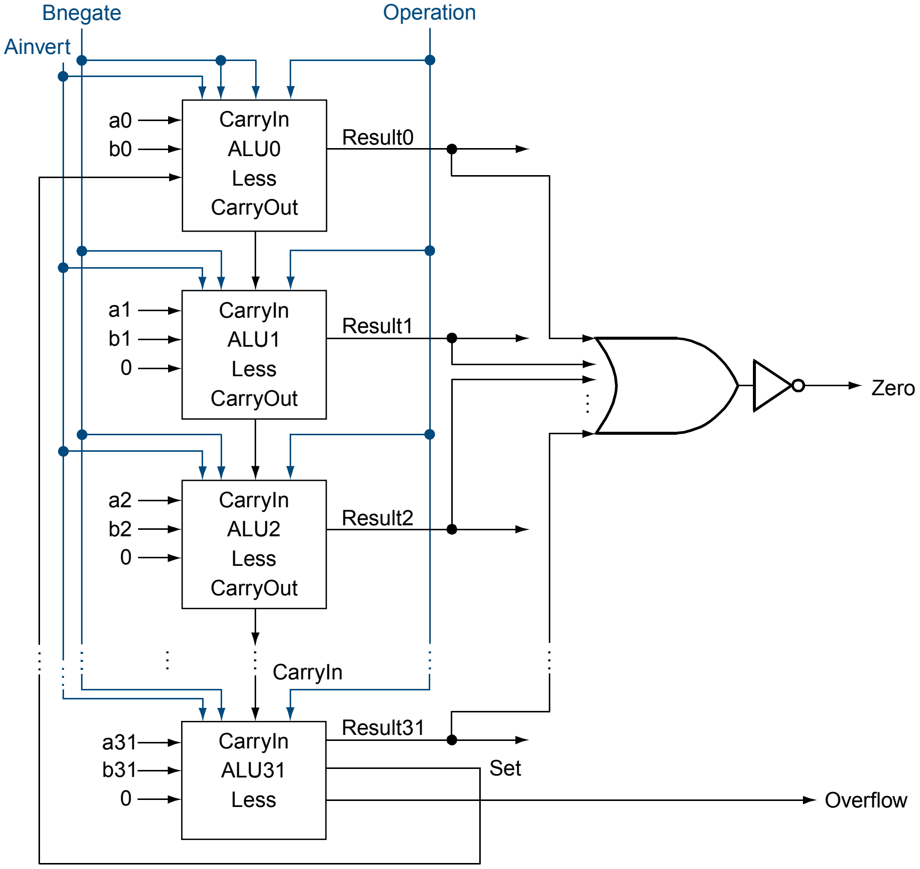


Test for equality

- **Notice control lines:**

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

•*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 saw this in multiplication, let's look at addition now

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?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $c_4 =$

Not feasible! Why?

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?**

$c_1 = g_0 + p_0 c_0$

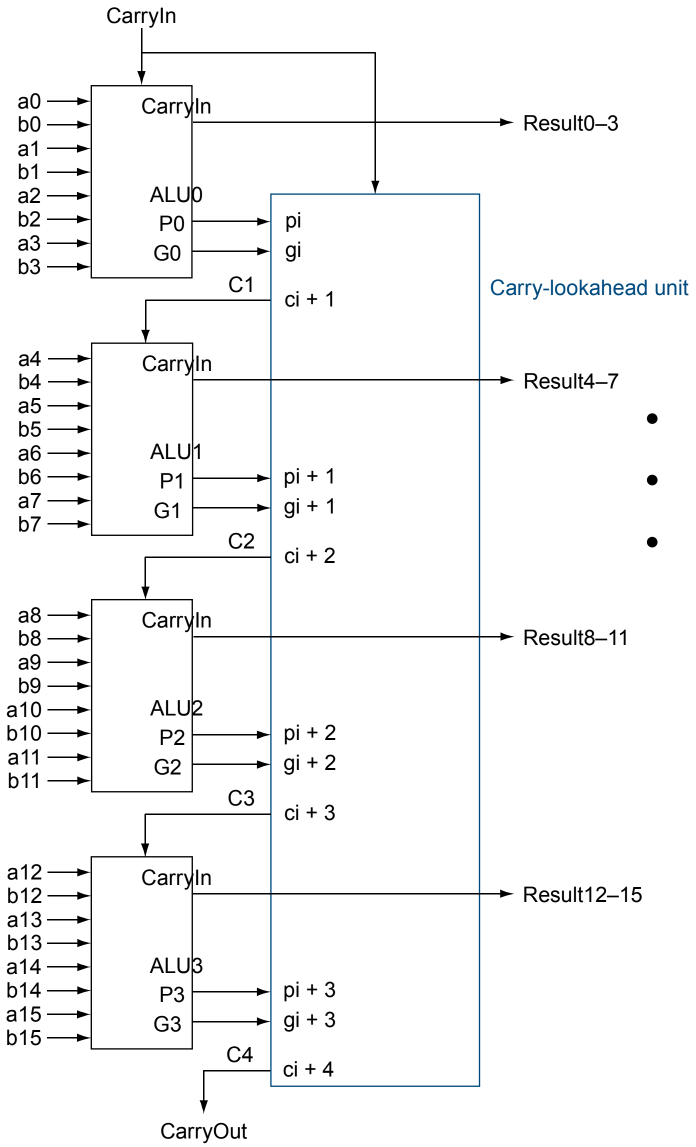
$c_2 = g_1 + p_1 c_1$ $c_2 =$

$c_3 = g_2 + p_2 c_2$ $c_3 =$

$c_4 = g_3 + p_3 c_3$ $c_4 =$

Feasible! Why?

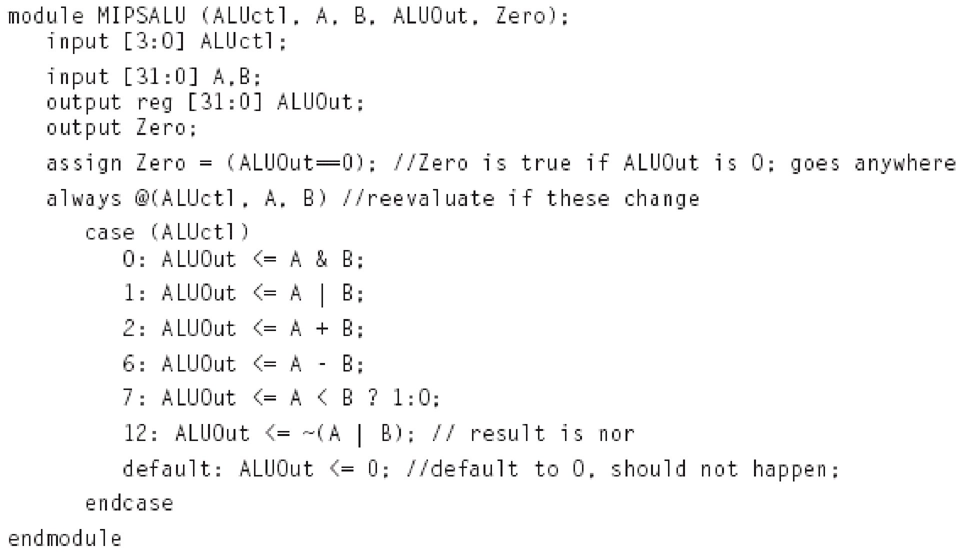
Use principle to build bigger adders



- 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!

ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!



```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

Chapter Five

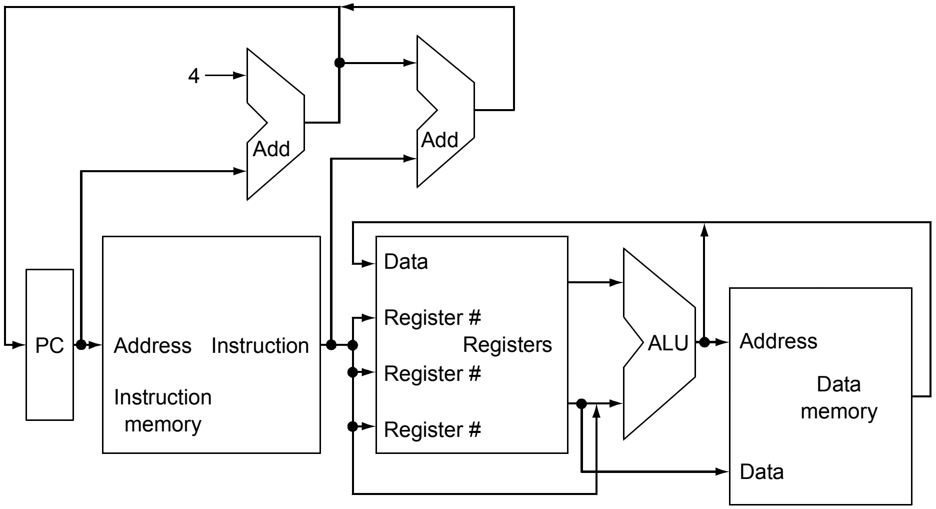
The Processor: Datapath & Control

- 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

  Why? memory-reference? arithmetic? control flow?

More Implementation Details

- **Abstract / Simplified View:**

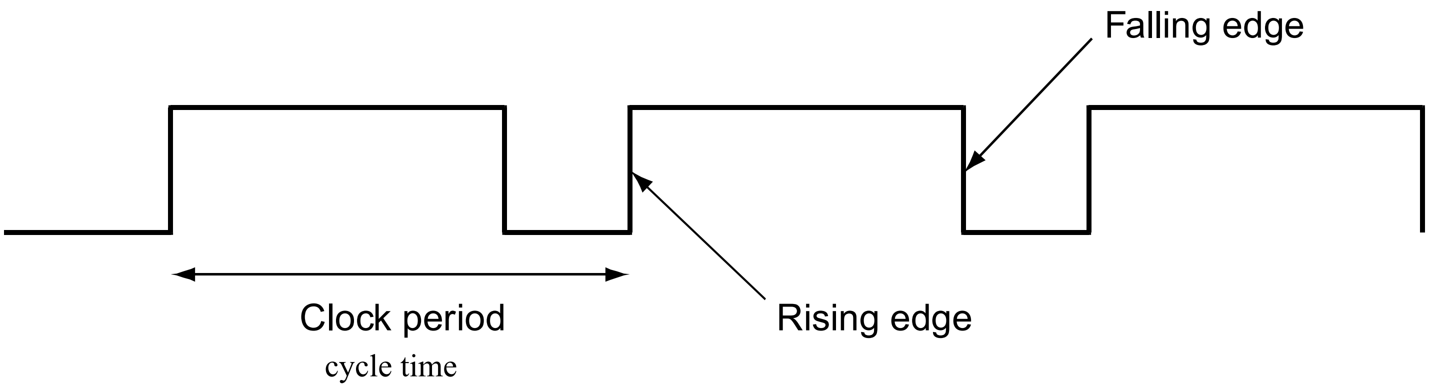


Two types of functional units:

- **elements that operate on data values (combinational)**
- **elements that contain state (sequential)**

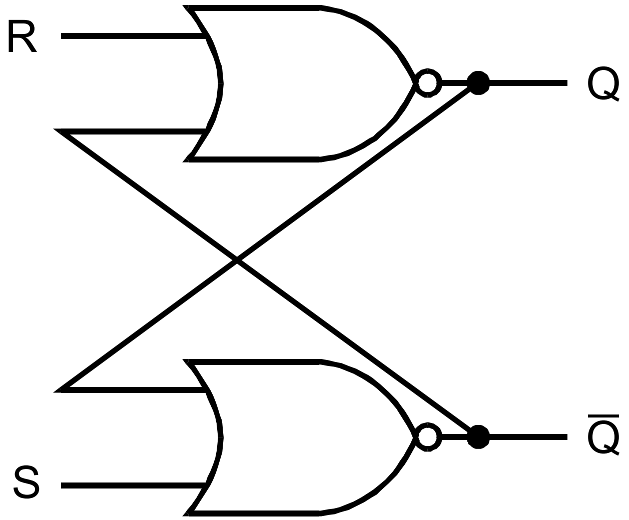
State Elements

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



An unclocked state element

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



Latches and Flip-flops

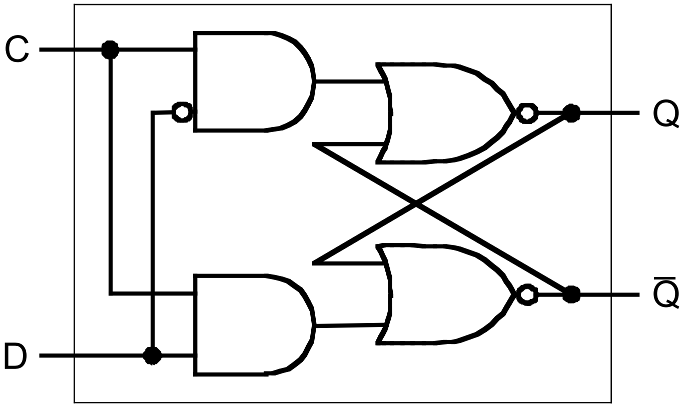
- Output is equal to the stored value inside the element
  (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

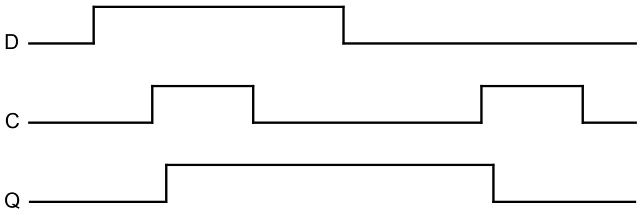
"logically true",
— could mean electrically low

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

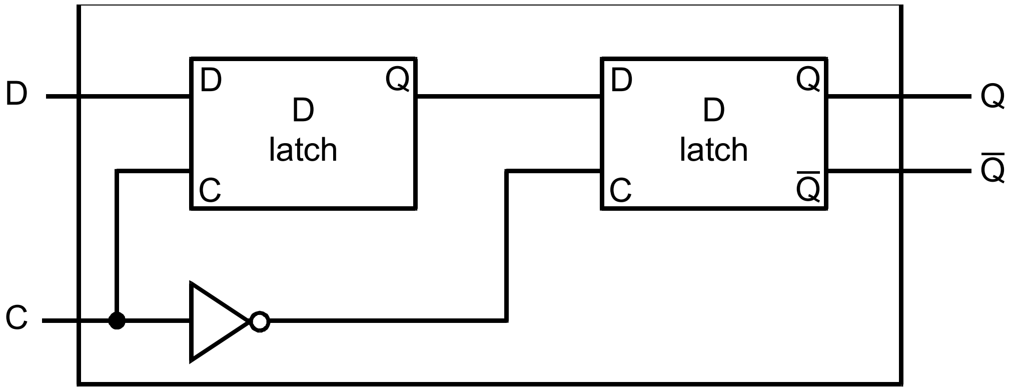
- **Two inputs:**
  - **the data value to be stored (D)**
  - **the clock signal (C) indicating when to read & store D**
- **Two outputs:**
  - **the value of the internal state (Q) and it's complement**

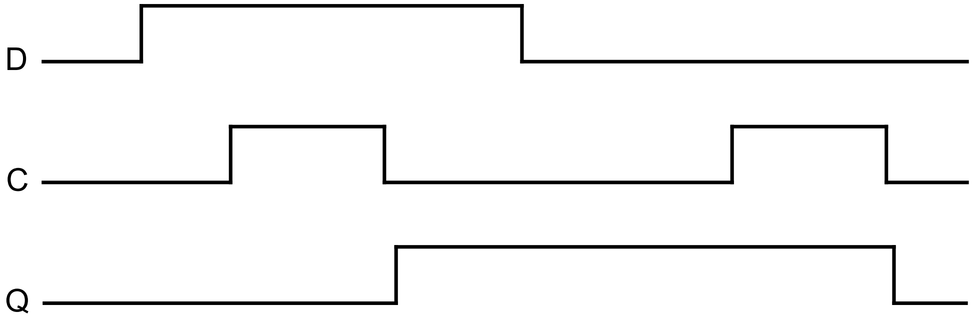




D flip-flop

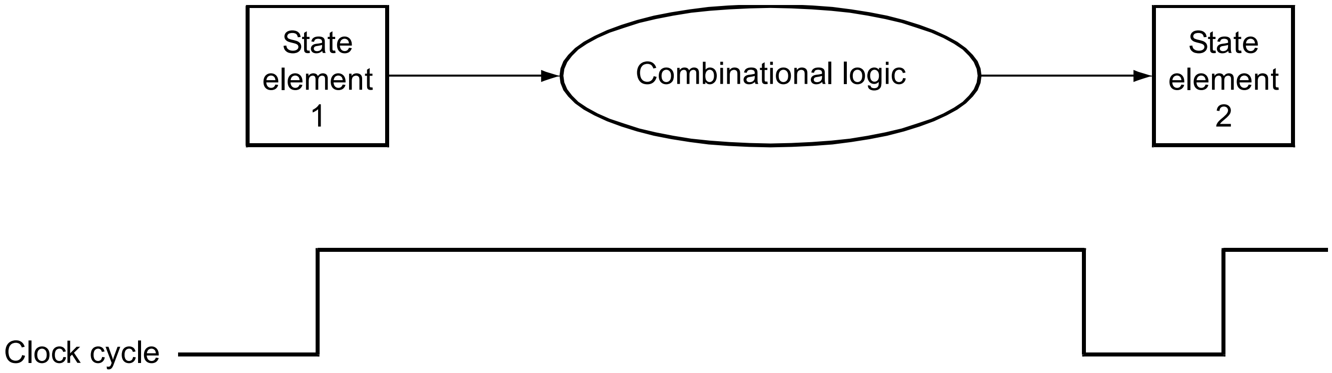
- **Output changes only on the clock edge**





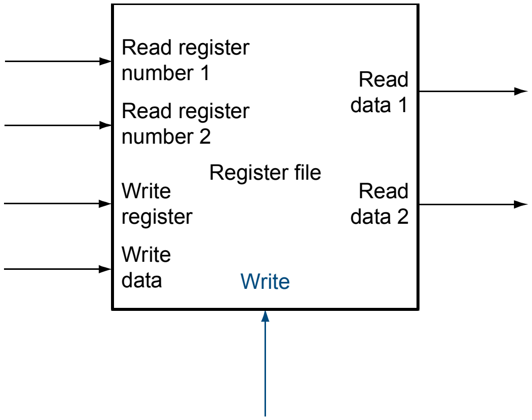
Our Implementation

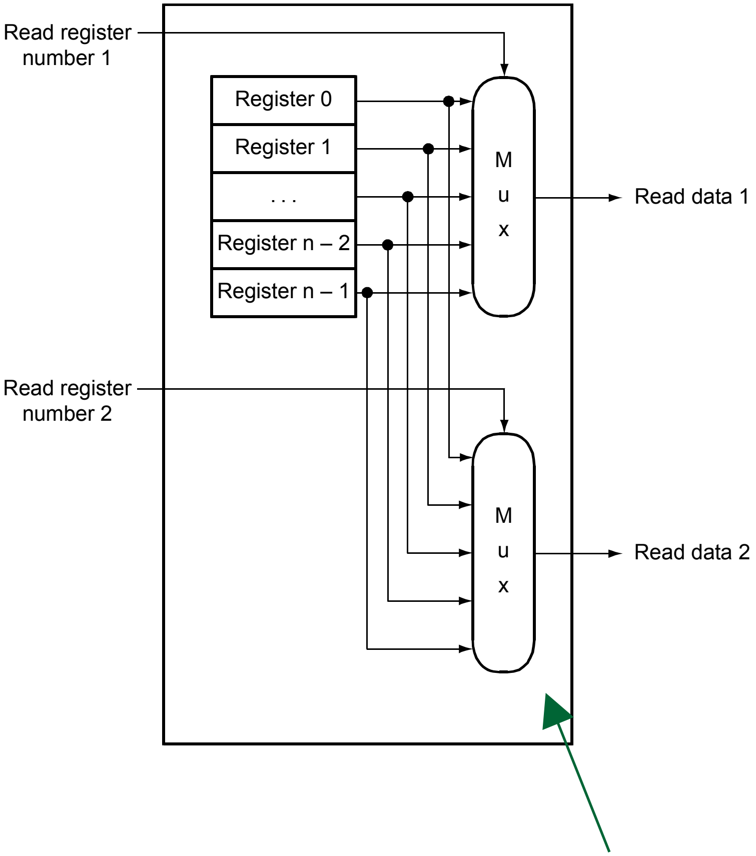
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



Register File

- **Built using D flip-flops**

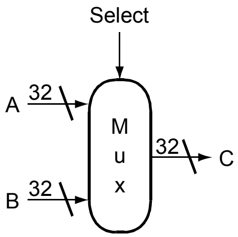


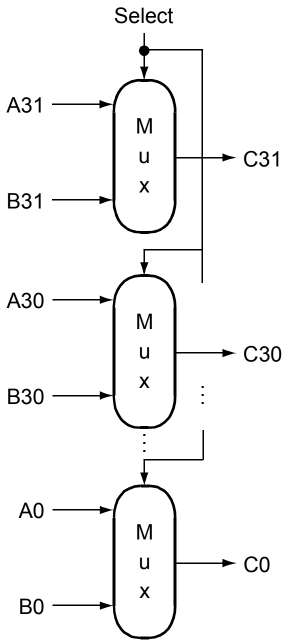


Do you understand? What is the "Mux" above?

Abstraction

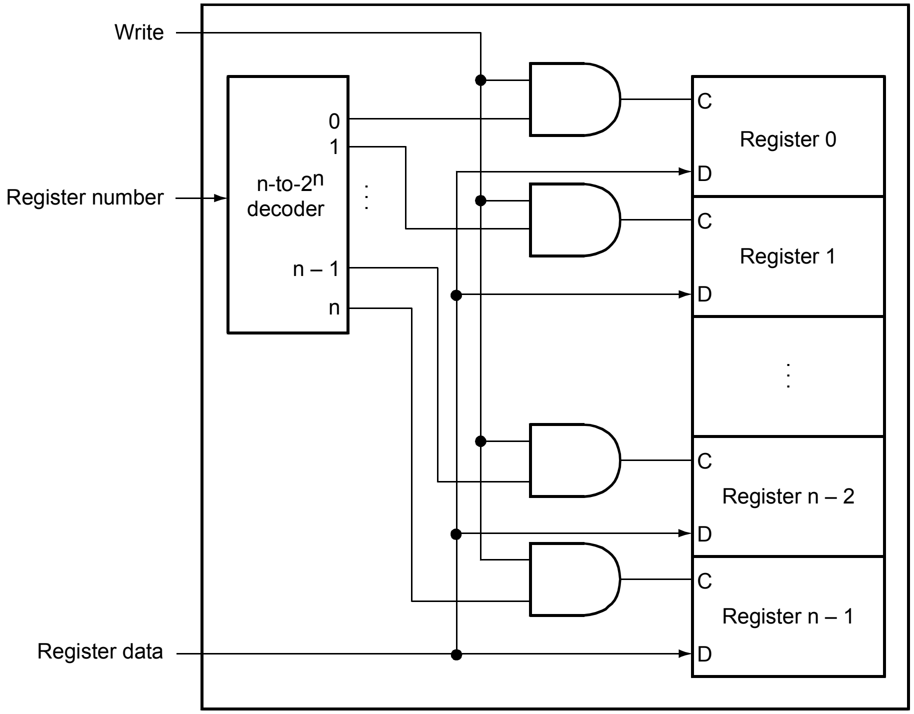
- **Make sure you understand the abstractions!**
- **Sometimes it is easy to think you do, when you don’t**





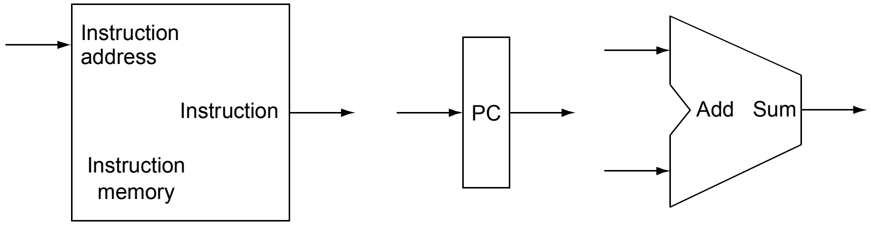
Register File

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

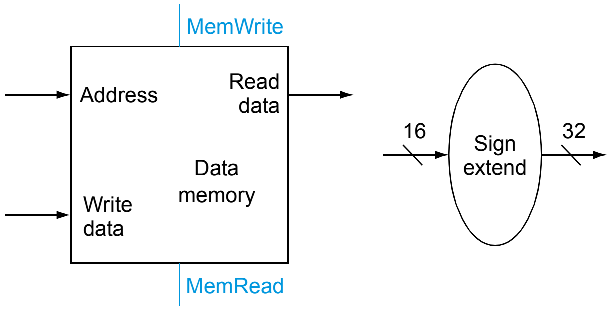


Simple Implementation

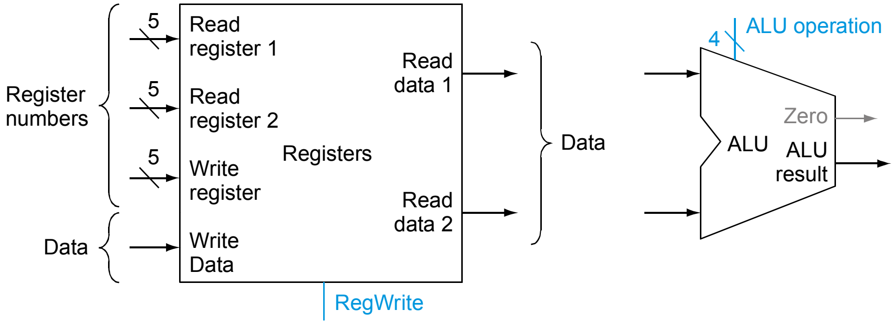
- **Include the functional units we need for each instruction**



a. Instruction memory b. Program counter c. Adder



a. Data memory unit b. Sign-extension unit

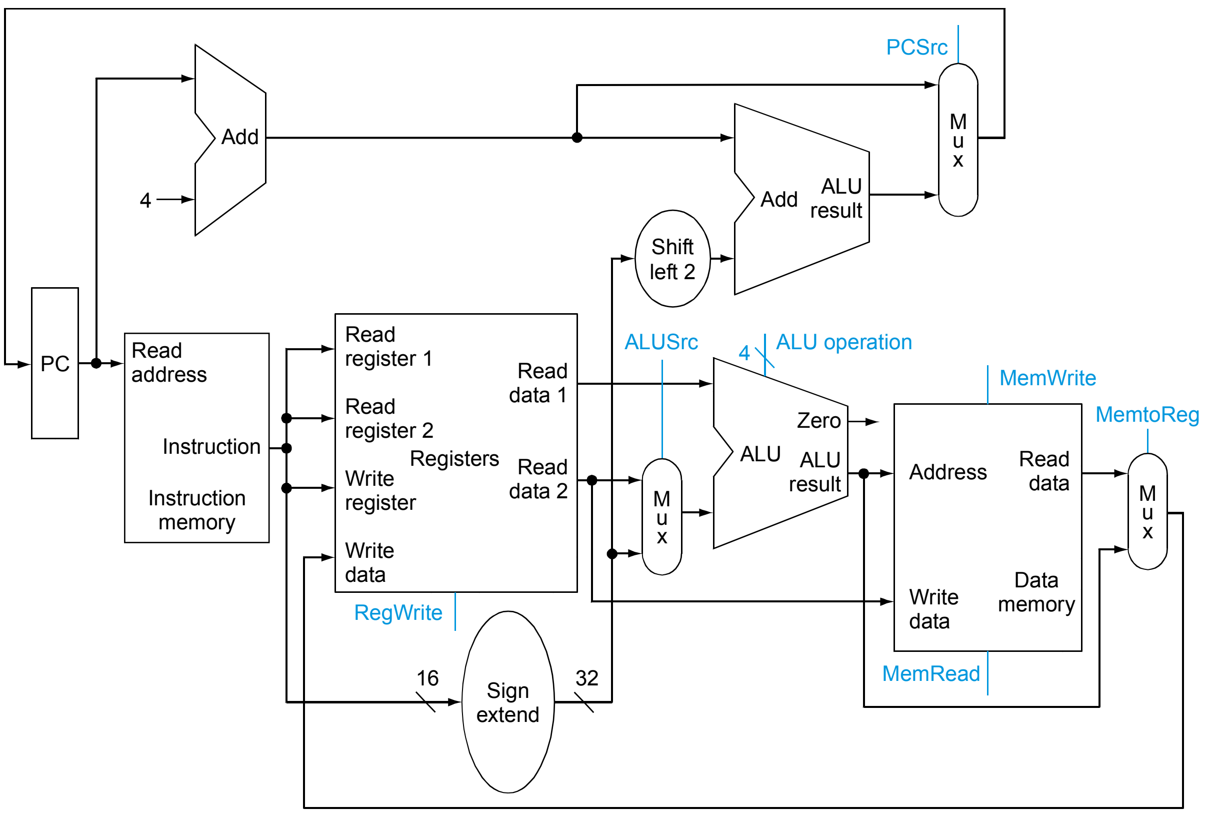


a. Registers b. ALU

Why do we need this stuff?

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?

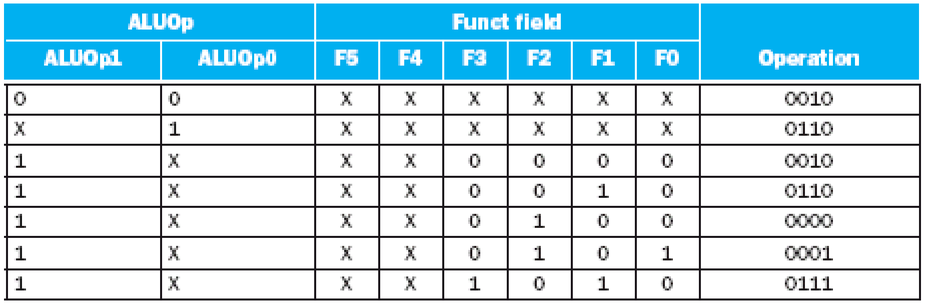
Control

- Must describe hardware to compute 4-bit ALU control input
  - given instruction type

    00 = lw, sw
    01 = beq,
    10 = arithmetic

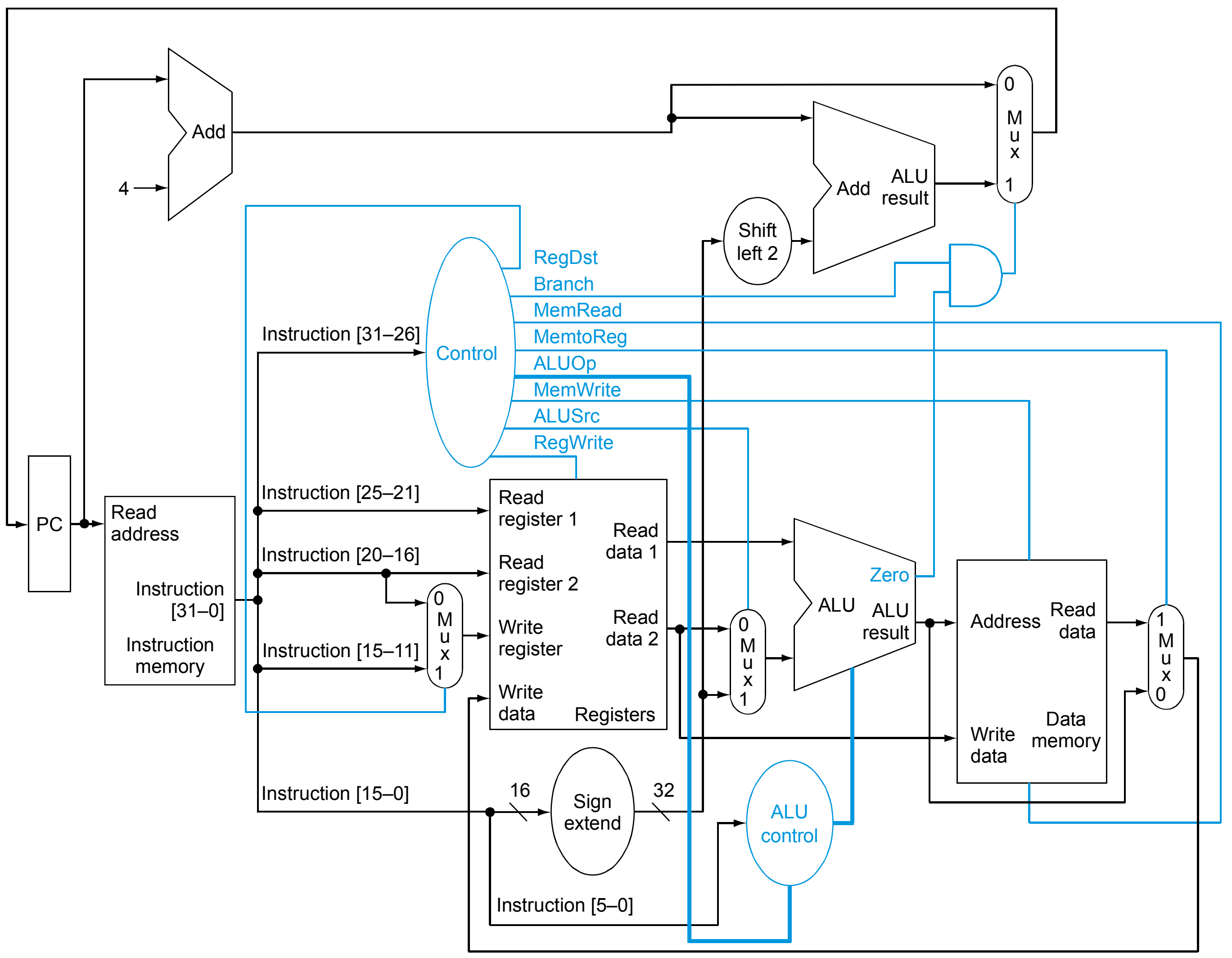
    ALUOp
    computed from instruction type
  - function code for arithmetic

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



ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
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

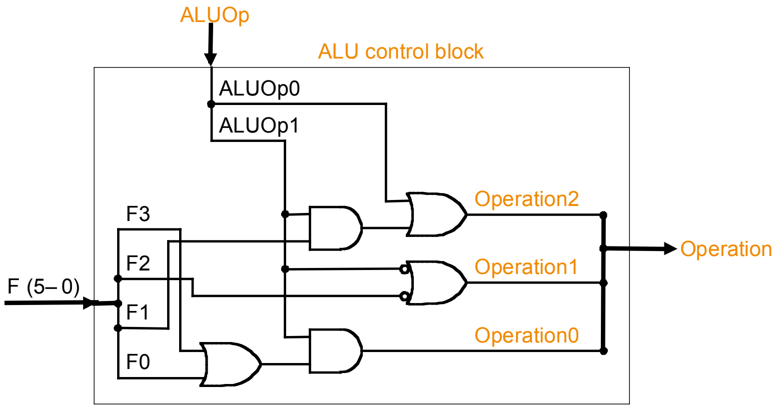
FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth 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.

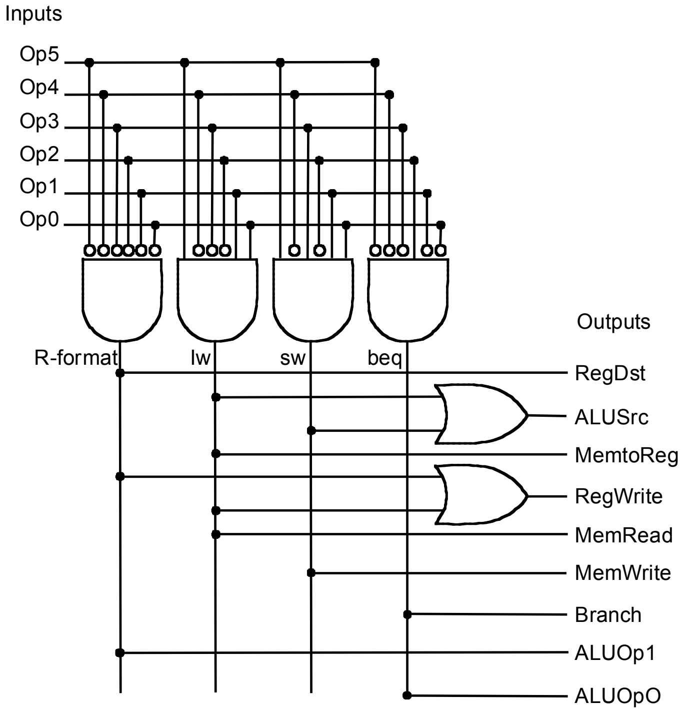


Instruction	RegDst	ALUSrc	Memto- Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control

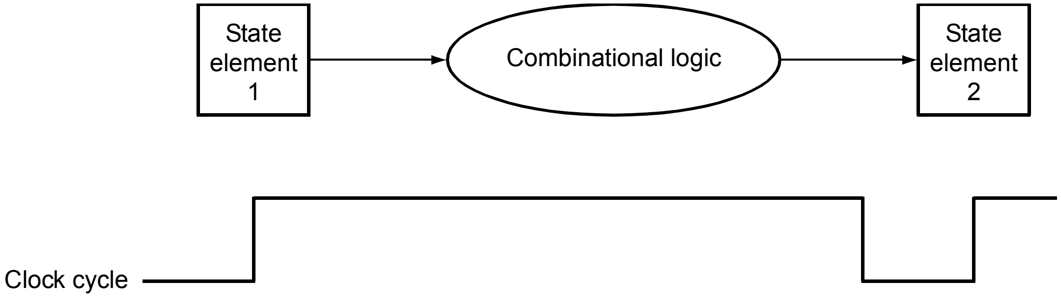
- **Simple combinational logic (truth tables)**





Our Simple Control Structure

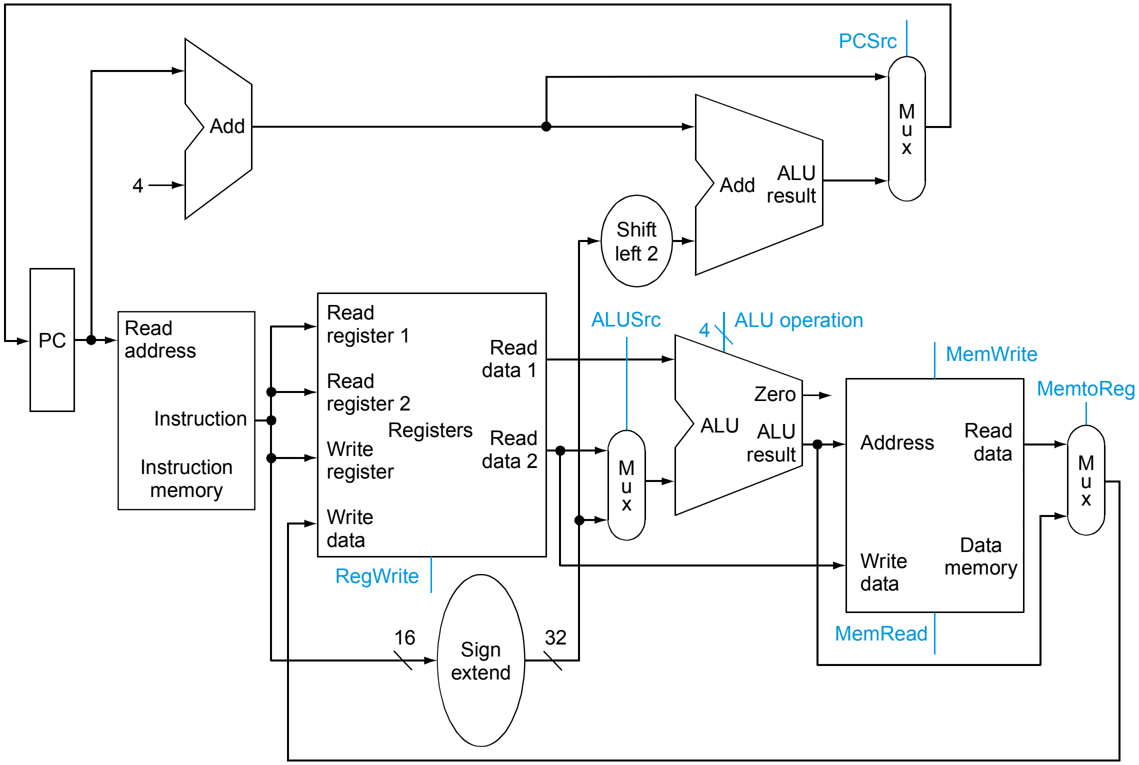
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce "right answer" right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

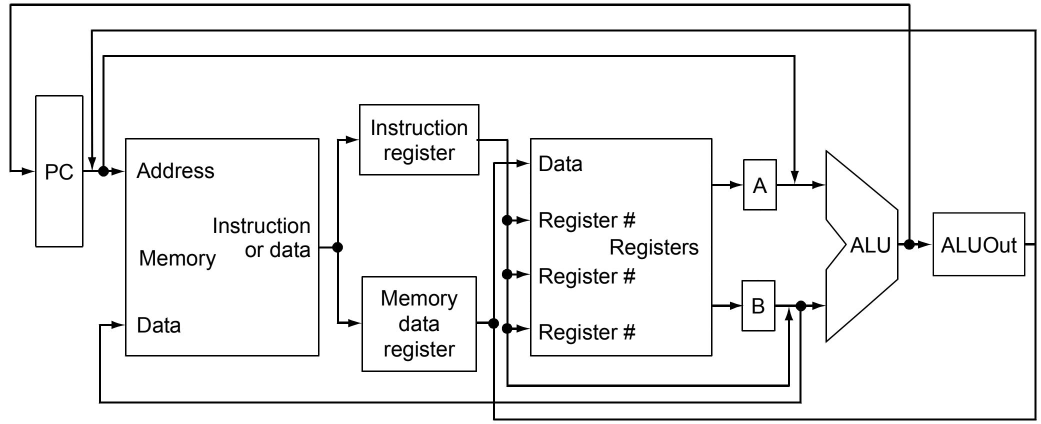
Single Cycle Implementation

- 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:

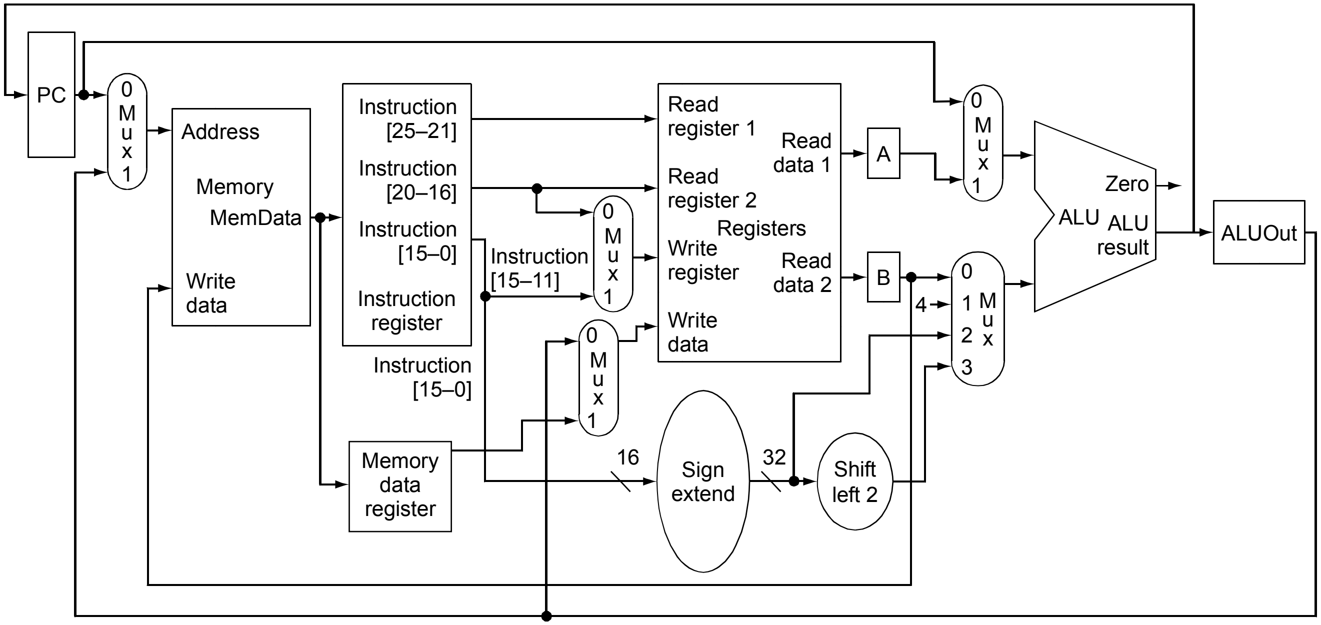


Multicycle Approach

- 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



Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum (“op”) of two registers.
  - 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]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[20:16]] <= ALUOut`

- **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
- 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;
```

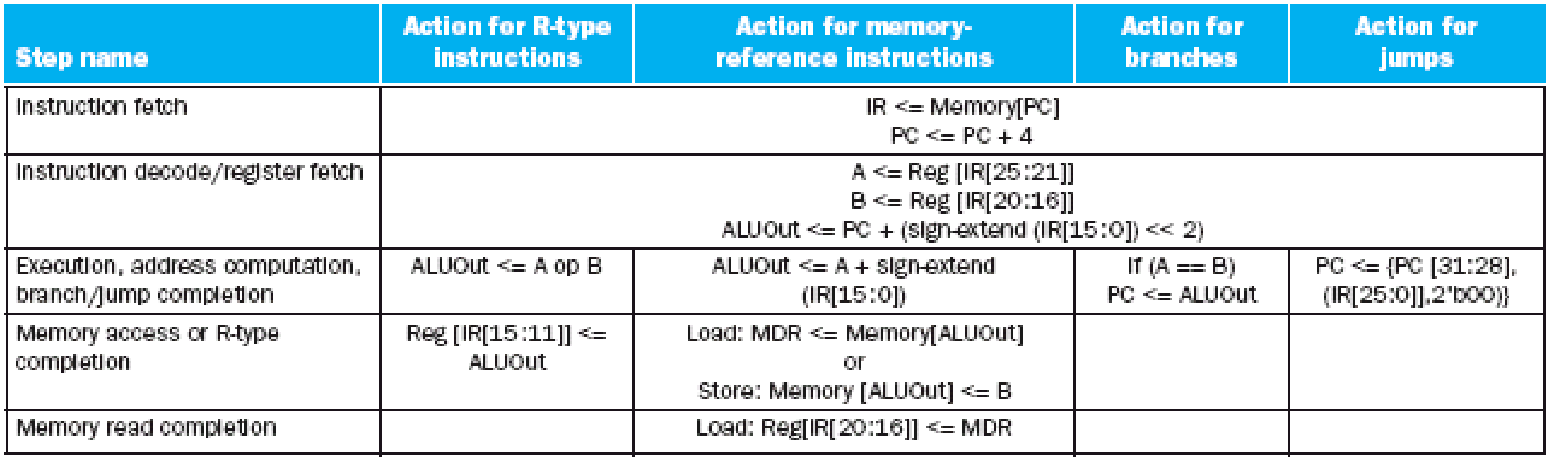
The write actually takes place at the end of the cycle on the edge

Write-back step

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

Which instruction needs this?

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) 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		

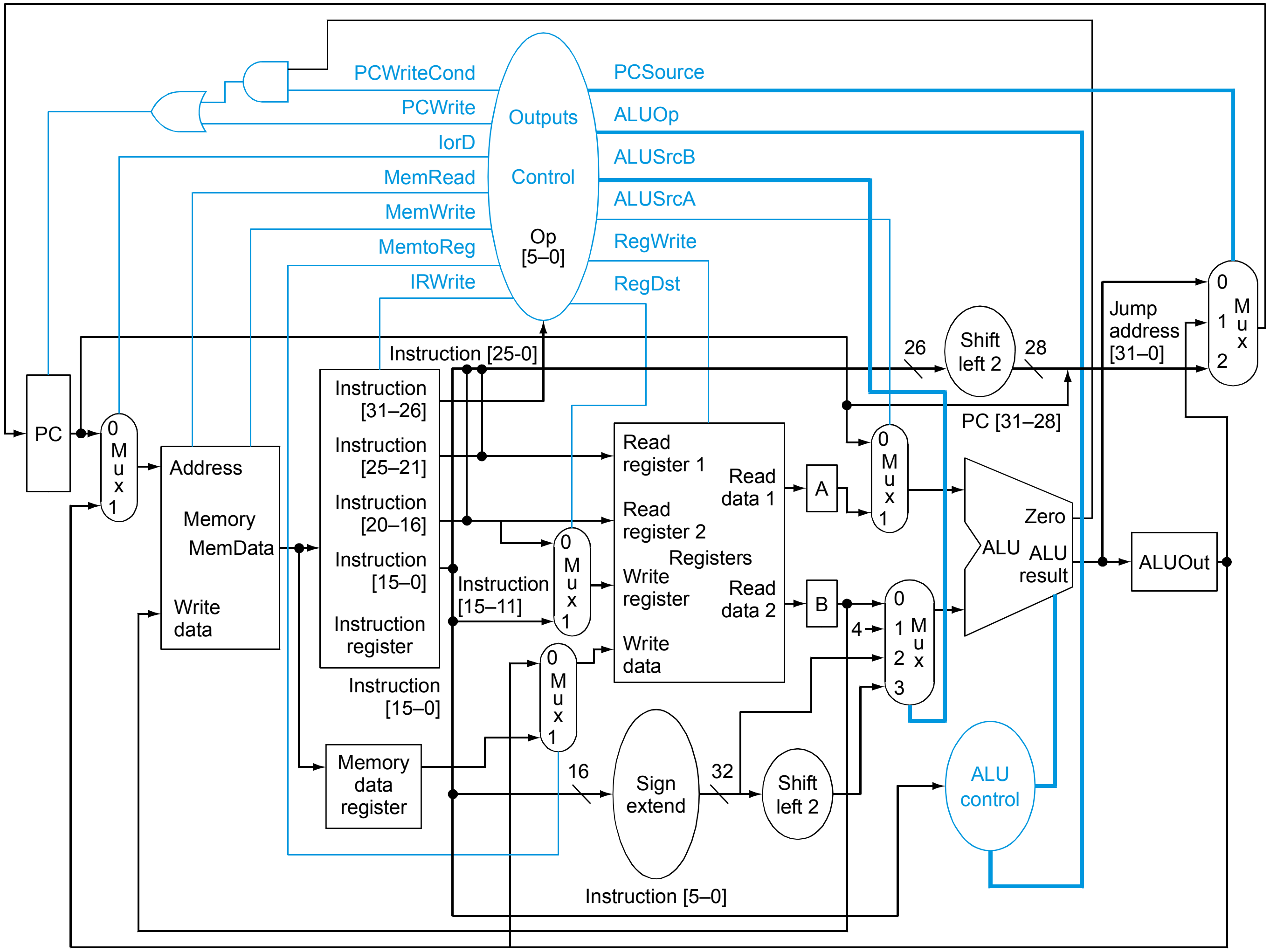
FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction 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?

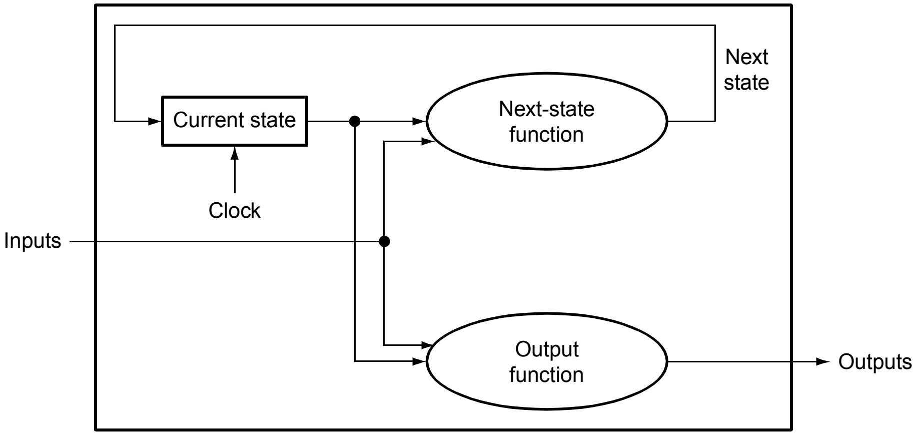
```
              lw $t2, 0($t3)
              lw $t3, 4($t3)
              beq $t2, $t3, Label        #assume not
              add $t5, $t2, $t3
              sw $t5, 8($t3)
Label:        ...
```

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


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25-0]
26
Shift
left 2
28
Jump
address
[31–0]
PC [31–28]
0
M
u
x
1
2
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–11]
0
M
u
x
1
0
M
u
x
1
Instruction
[15–0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
0
M
u
x
1
4
0
1 M
2 u
3 x
Zero
ALU
ALU
result
ALUOut
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]

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)

Review: finite state machines

- **Example:**

  B. 37 ***A friend would like you to build an "electronic eye" for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light "moves" from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite state machine used to specify the electronic eye. Note that the rate of the eye's movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.***

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

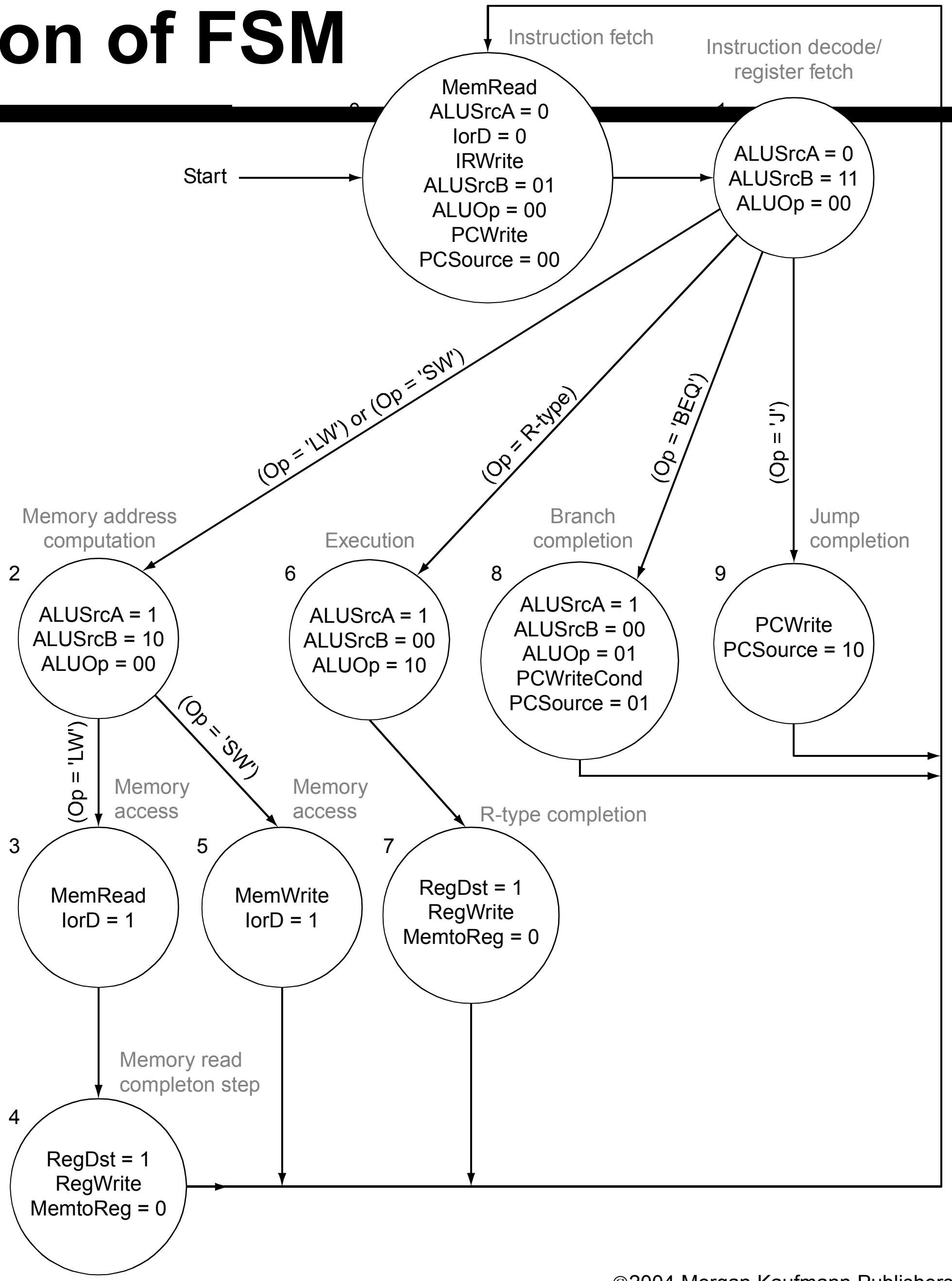
- Use the information we’ve accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification

Graphical Specification of FSM

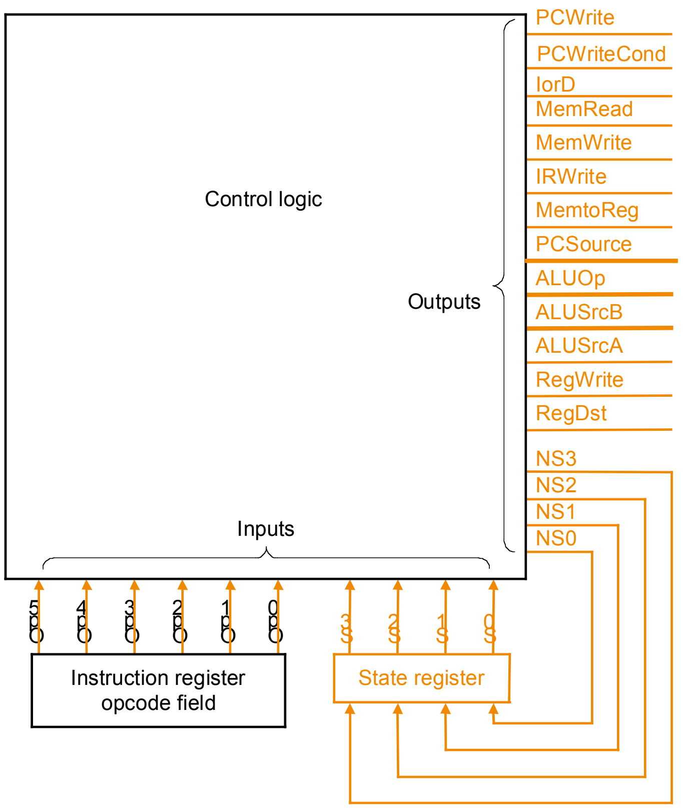
- **Note:**
  - **don’t care if not mentioned**
  - **asserted if name only**
  - **otherwise exact value**

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



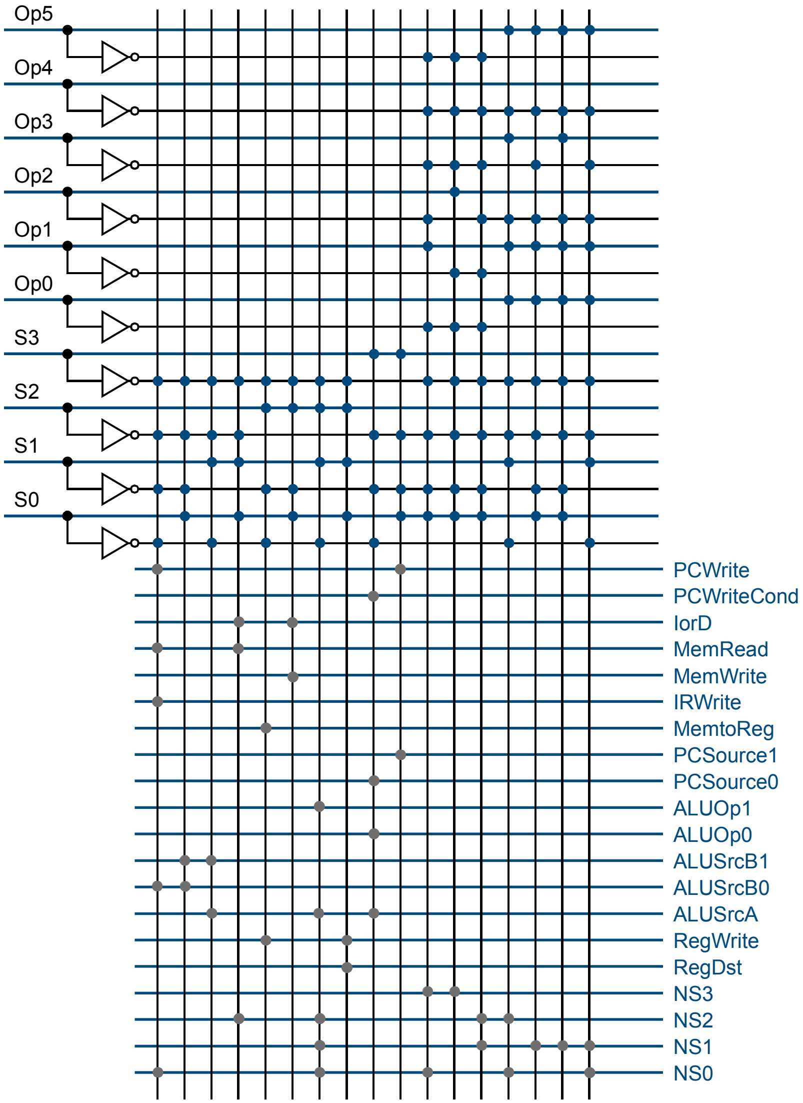
Finite State Machine for Control

- **Implementation:**



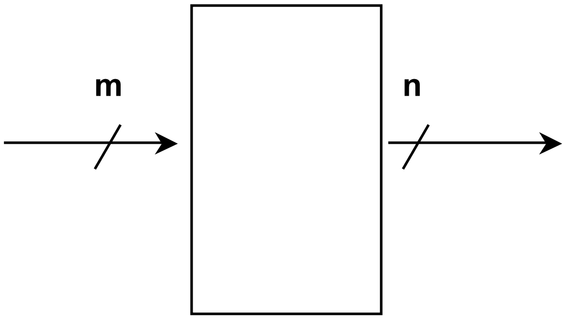
PLA Implementation

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



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"

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

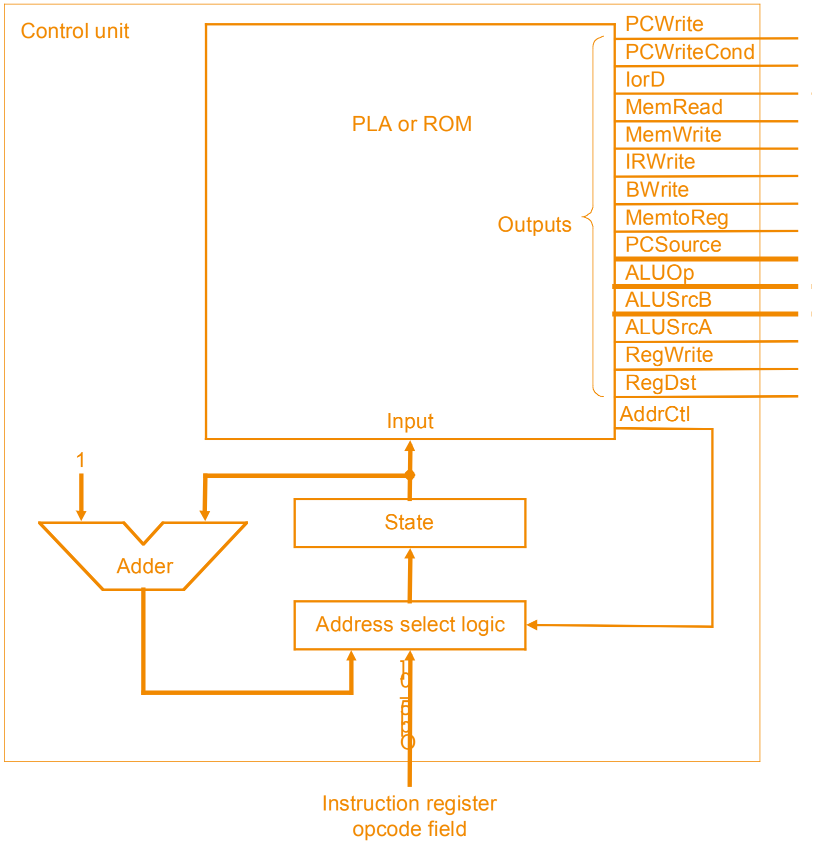
- Break up the table into two parts
  - — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - — Total: 4.3K bits of ROM
- PLA is much smaller
  - — can share product terms
  - — only need entries that produce an active output
  - — can take into account don't cares
- Size is (#inputs × #product-terms) + (#outputs × #product-terms)

  For this example = (10x17)+(20x17) = 510 PLA cells

- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

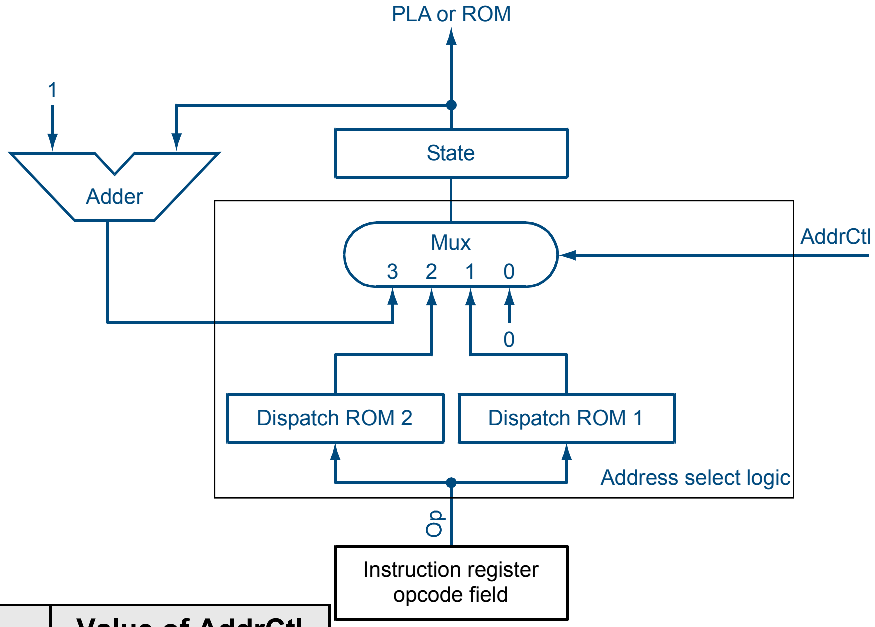
- **Complex instructions: the "next state" is often current state + 1**



Details

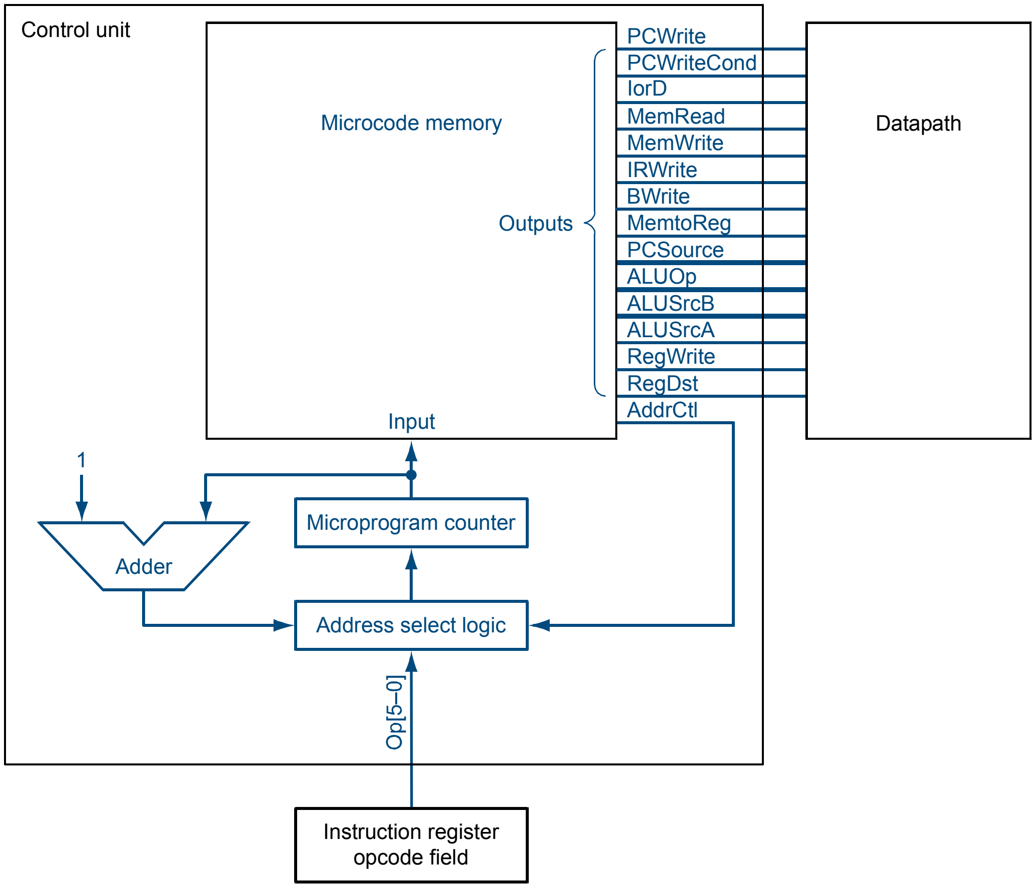
Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	`jmp`	1001
000100	`beq`	1000
100011	`lw`	0010
101011	`sw`	0010

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**

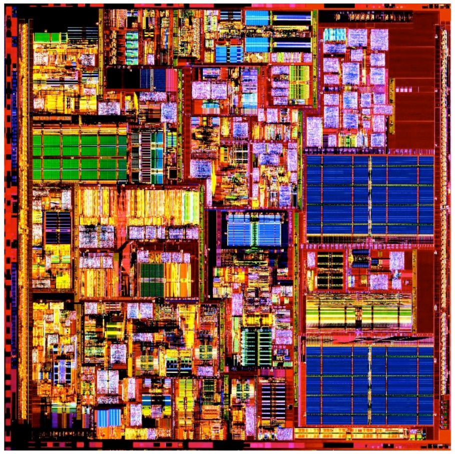
Historical Perspective

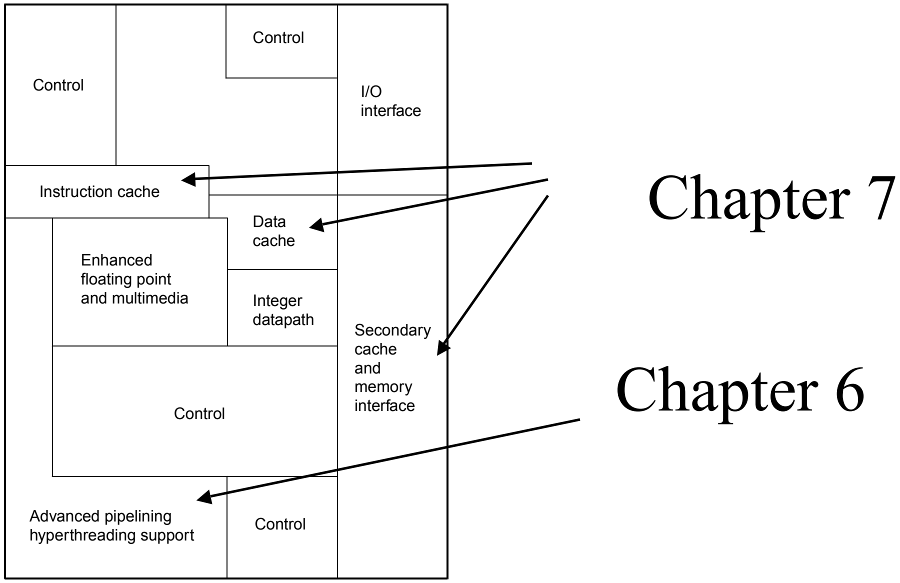
- In the ‘60s and ‘70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the ‘80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - “hardwired control” for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - “microcoded control” for more complex instructions
    (large numbers of cycles, central control store)

- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)



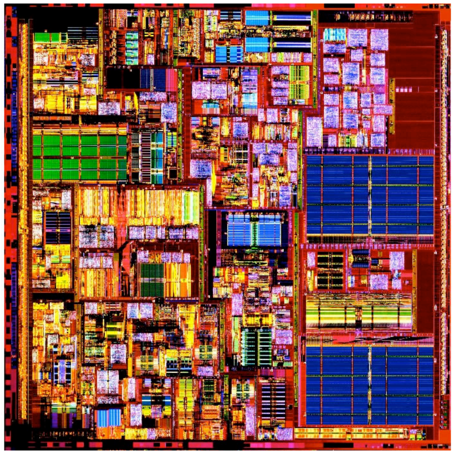

Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control
Chapter 7
Chapter 6

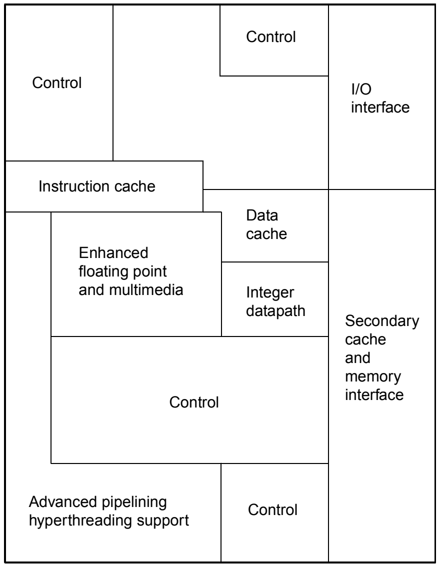
- Pipelining is used for the simple instructions favored by compilers

  “Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions”

Pentium 4

- Somewhere in all that “control we must handle complex instructions




Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- Its complicated!

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