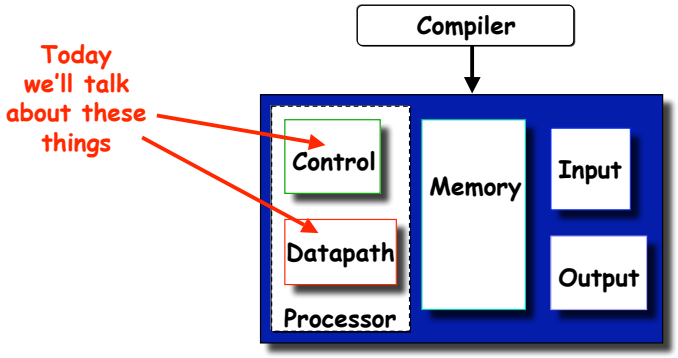
CSE 30321
MIPS Single Cycle Dataflow

The goals of this lecture are…

- …to show how ISAs map to real HW and affect the organization of processing logic…
- …and to set up a discussion of pipelining + other principles of modern processing…

The organization of a computer

Von Neumann Model:
- Stored-program machine instructions are represented as numbers
- Programs can be stored in memory to be read/written just like numbers



Functions of Each Component

- Datapath: performs data manipulation operations
  - arithmetic logic unit (ALU)
  - floating point unit (FPU)
- Control: directs operation of other components
  - finite state machines
  - micro-programming
- Memory: stores instructions and data
  - random access v.s. sequential access
  - volatile v.s. non-volatile
  - RAMs (SRAM, DRAM), ROMs (PROM, EEPROM), disk
  - tradeoff between speed and cost/bit
- Input/Output and I/O devices: interface to the environment
  - mouse, keyboard, display, device drivers

The Performance Perspective

- Performance of a machine determined by
  - Instruction count, clock cycles per instruction, clock cycle time
- Processor design (datapath and control) determines:
  - Clock cycles per instruction
  - Clock cycle time
- We will discuss a simplified MIPS implementation

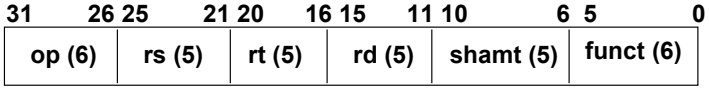
Review of Design Steps

- Instruction Set Architecture => RTL representation
- RTL representation =>
  - Datapath components
  - Datapath interconnects
- Datapath components => Control signals
- Control signals => Control logic

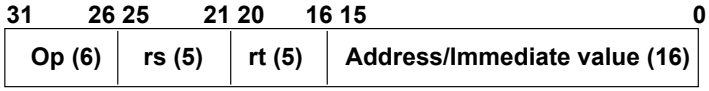
- Writing RTL: How many states (cycles) should an instruction take?
  - CPI
  - Datapath component sharing

MIPS Instruction Formats

- All MIPS instructions are 32 bits (4 bytes) long.
- R-type:



- I-Type:



- J-type

Let's talk about this generally on the board first…

- Let's just look at our instruction formats and "derive" a simple datapath
  - (we need to make all of these instruction formats "work")

The MIPS Subset

- To simplify things a bit we'll just look at a few instructions:
  - memory-reference: lw, sw
  - arithmetic-logical: add, sub, and, or, slt
  - branching: beq, j

- Organizational overview:
  - fetch an instruction based on the content of PC
  - decode the instruction
  - fetch operands
    - (read one or two registers)
  - execute
    - (effective address calculation/arithmetic-logical operations/comparison)
  - store result
    - (write to memory / write to register / update PC)

At simplest level, this is how Von Neumann, RISC model works

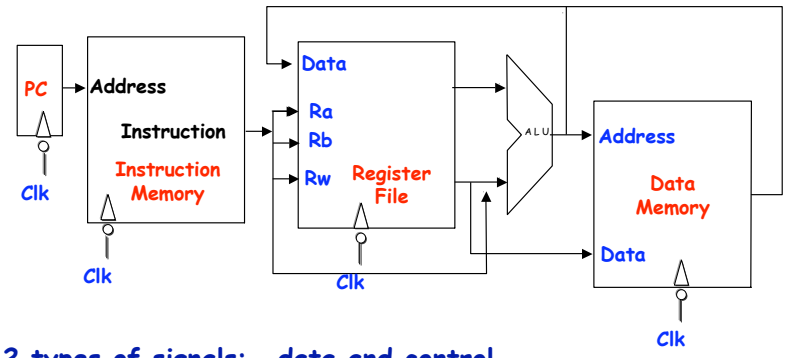
What we'll do…

- …look at instruction encodings…
- …look at datapath development…
- …discuss how we generate the control signals to make the datapath elements work…

Implementation Overview

- Abstract / Simplified View:

simplest view of Von Neumann, RISC μP



- 2 types of signals: data and control
- Clocking strategy: All storage elements clocked by same clock edge.

Review of Design Steps

- Instruction set Architecture => RTL representation
- RTL representation =>
  - Datapath components
  - Datapath interconnects
- Datapath components => Control signals
- Control signals => Control logic
- Writing RTL: How many states (cycles) should an instruction take?
  - CPI
  - Datapath component sharing

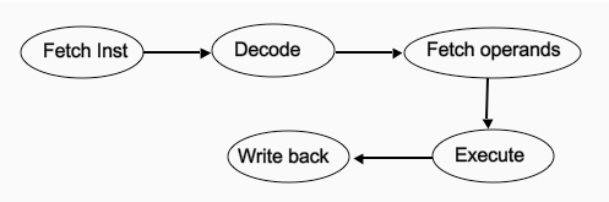
i.e. PC ← PC + 4
(or $4 ← $3 + $2)

need these to do

need these to do

need these to do

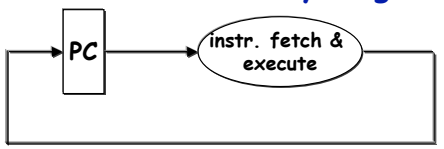
What to be Done for Each Instruction?



- How many cycles should the above take?
- You are the architect so you decide!
- Less cycles => more to be done in one cycle

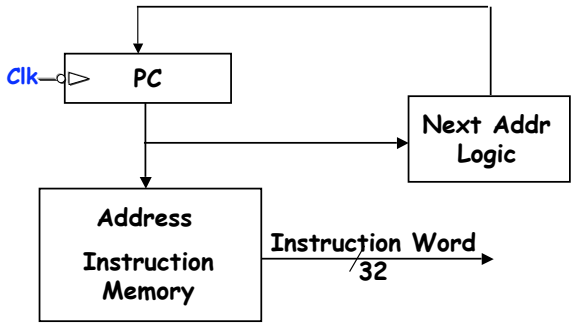
Single Cycle Implementation

- Each instruction takes one cycle to complete.
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce "right answer" right away
    - (why?)
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



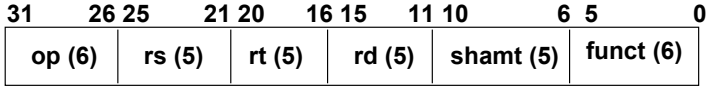
Instruction Fetch Unit

- Fetch the instruction: mem[PC] ,
- Update the program counter:
  - sequential code: PC <- PC+4
  - branch and jump: PC <- "something else"



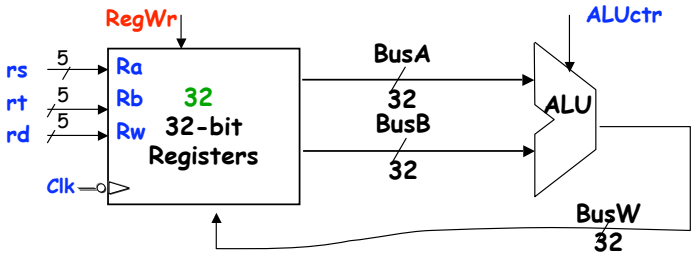
Let's say we want to fetch… …an R-type instruction (arithmetic)

- Instruction format:



- RTL:
  - Instruction fetch: mem[PC]
  - ALU operation: reg[rd] <- reg[rs] op reg[rt]
  - Go to next instruction: Pc <- PC+ 4
- Ra, Rb and Rw are from instruction's rs, rt, rd fields.
- Actual ALU operation and register write should occur after decoding the instruction.

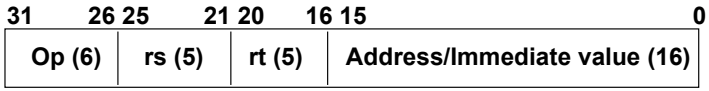
Datapath for R-Type Instructions



- Register timing:
  - Register can always be read.
  - Register write only happens when RegWr is set to high and at the falling edge of the clock

I-Type Arithmetic/Logic Instructions

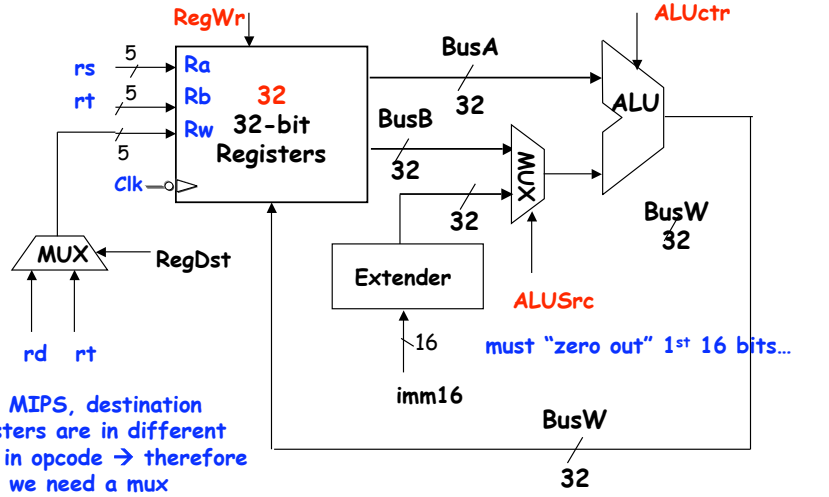
- Instruction format:



- RTL for arithmetic operations: e.g., ADDI
  - Instruction fetch: mem[PC]
  - Add operation: reg[rt] <- reg[rs] + SignExt(imm16)
  - Go to next instruction: Pc <- PC+ 4
- Also, immediate instructions

Datapath for I-Type A/L Instructions

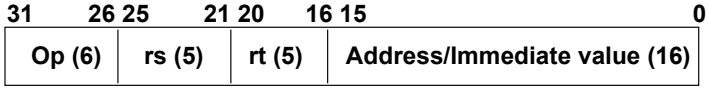
note that we reuse ALU…



In MIPS, destination registers are in different places in opcode → therefore we need a mux

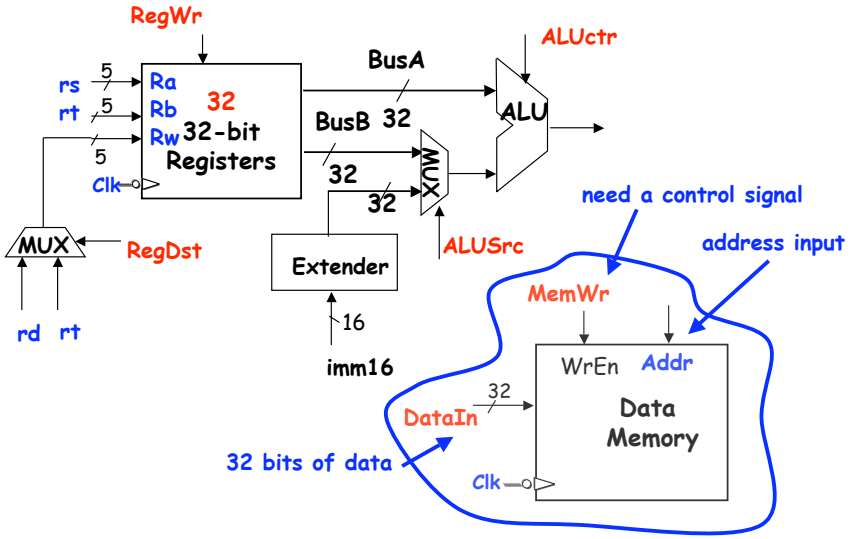
I-Type Load/Store Instructions

- Instruction format:



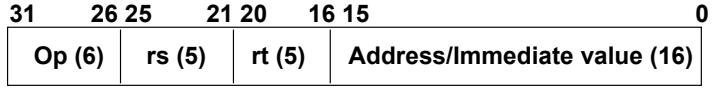
- RTL for load/store operations: e.g., LW
  - Instruction fetch: mem[PC]
  - Compute memory address: Addr <- reg[rs] + SignExt(imm16)
  - Load data into register: reg[rt] <- mem[Addr]
  - Go to next instruction: Pc <- PC+ 4
- How about store?
  - same thing, just skip 3rd step (mem[addr] ← reg[rs])

Datapath for Load/Store Instructions



I-Type Branch Instructions

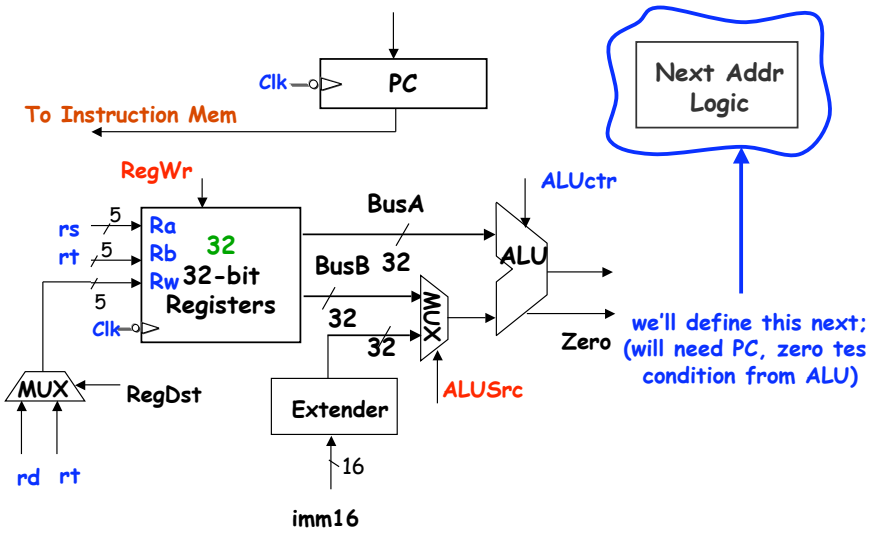
- Instruction format:



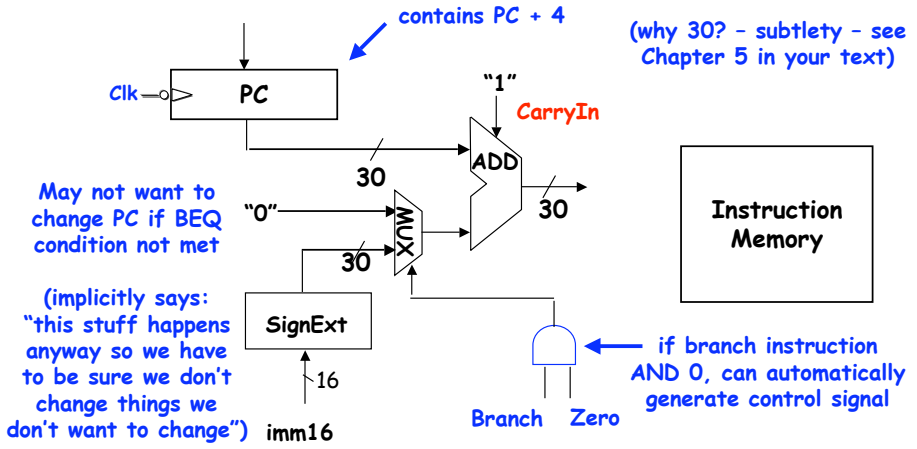
- RTL for branch operations: e.g., BEQ
  - Instruction fetch: mem[PC]
  - Compute conditon: Cond <- reg[rs] - reg[rt]
  - Calculate the next instruction's address:

```
if (Cond eq 0)  then
        PC <- PC+ 4 + (SignExd(imm16) x 4)
else ?
```


Datapath for Branch Instructions



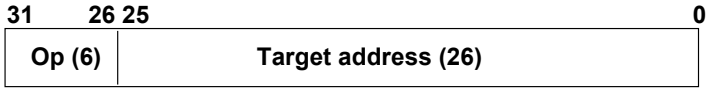
Next Address Logic



When does the correct new PC become available?
Can we do better?

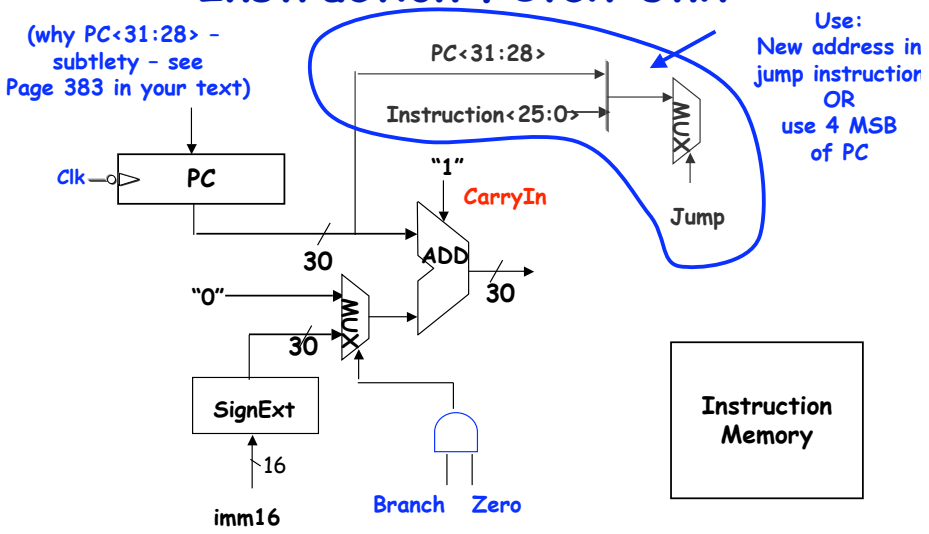
J-Type Jump Instructions

- Instruction format:

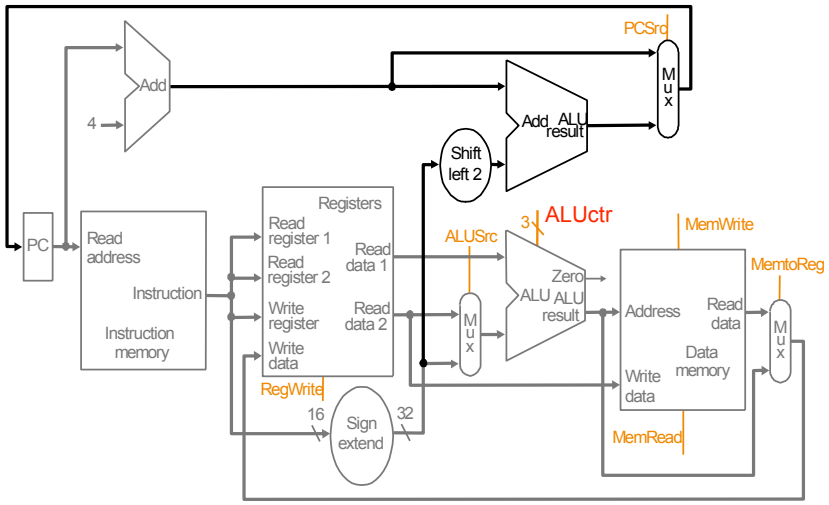


- RTL operations: e.g., BEQ
  - Instruction fetch: mem[PC]
  - Set up PC: PC <- ((PC+ 4)<31:29> CONCAT(target<25:0>) x 4

Instruction Fetch Unit



A Single Cycle Datapath



Add Jump.

Let's trace a few instructions

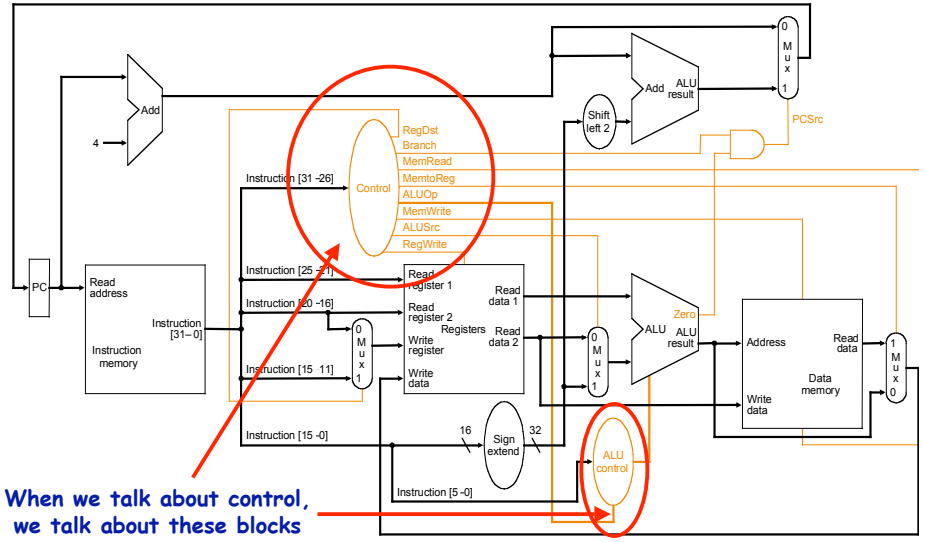
- For example…
  - Add $5, $6, $7
  - SW 0($9), $10
  - Sub $1, $2, $3
  - LW $11, 0($12)

Control Logic

(I.e. *now*, we need to make the HW do what we want it to do - add, subtract, etc. - when we want it to…)

The HW needed, plus control

Single cycle MIPS machine

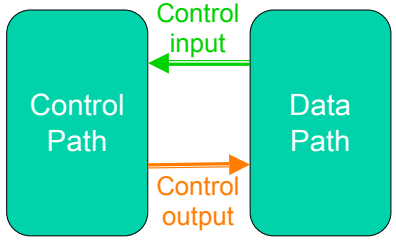


When we talk about control, we talk about these blocks

Implementing Control

- Implementation Steps:
  - Identify control inputs and control output (control words)
  - Make a control signal table for each cycle
  - Derive control logic from the control table
- Do we need a FSM here?

Control inputs:
Opcode (5 bits)
Func (6 bits)



Control outputs:
RegDst
MemtoReg
RegWrite
MemRead
MemWrite
ALUSrc
ALUctr
Branch
Jump

Implementing Control

- Implementation Steps:
  1. Identify control inputs and control outputs
  2. Make a control signal table for each cycle
  3. Derive control logic from the control table
     - This logic can take on many forms: combinational logic, ROMs, microcode, or combinations…

Single Cycle Control Input/Output

- Control Inputs:
  - Opcode (6 bits)
  - How about R-type instructions?
- Control Outputs:
  - RegDst
  - ALUSrc
  - MemtoReg
  - RegWrite
  - MemRead
  - MemWrite
  - Branch
  - Jump
  - ALUctr

Step 1: Idenitfy inputs & outputs

these are columns

these are rows

Step 2: Make a control signal table for each cycle

Control Signal Table

R-type

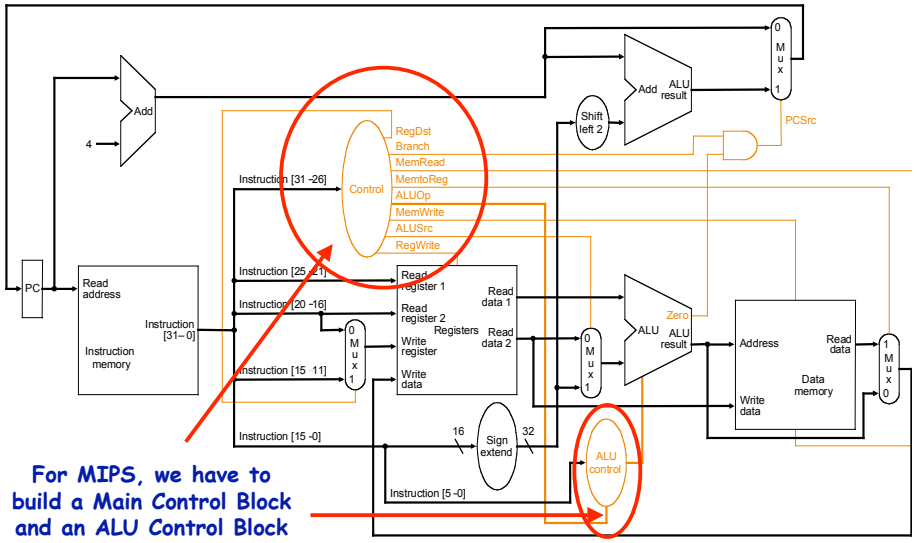
(inputs)

	Add	Sub	LW	SW	BEQ
Func (input)	100000	100010	xxxxxx	xxxxxx	xxxxxx
Op (input)	000000	000000	100011	101011	000100
RegDst	1	1	0	X	X
ALUSrc	0	0	1	1	0
Mem-to-Reg	0	0	1	X	X
Reg. Write	1	1	1	0	0
Mem. Read	0	0	1	0	0
Mem. Write	0	0	0	1	0
Branch	0	0	0	0	1
ALUOp	Add	Sub	00	00	01

(outputs)

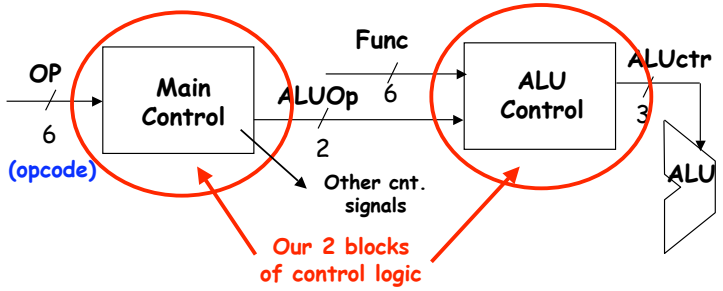
The HW needed, plus control

Single cycle MIPS machine



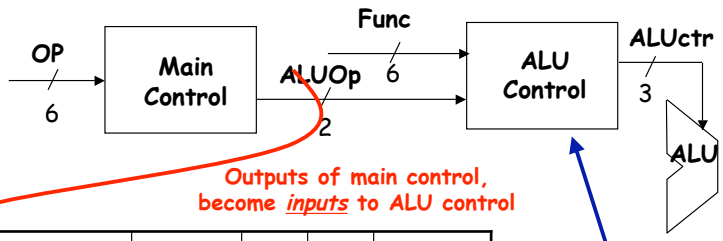
For MIPS, we have to build a Main Control Block and an ALU Control Block

Main control, ALU control



- Use OP field to generate ALUOp (encoding)
  - Control signal fed to ALU control block
- Use Func field and ALUOp to generate ALUctr (decoding)
  - Specifically sets 3 ALU control signals
    - B-Invert, Carry-in, operation

Main control, ALU control



Outputs of main control, become *inputs* to ALU control

	R-type	lw	sw	beq
ALU Operation	"R-type"	add	add	subtract
ALUOp<1:0>	10	00	00	01

We have 8 bits of input to our ALU control block; we need 3 bits of output…

Or in other words…
00 = ALU performs add
01 = ALU performs sub
10 = ALU does what function code says

Generating ALUctr

- We want these outputs:

ALU Operation	and	or	add	sub	slt
ALUctr<2:0>	000	001	010	110	111

ALUctr<2> = B-negate (C-in & B-invert)
ALUctr<1> = Select ALU Output
ALUctr<0> = Select ALU Output

Invert B and C-in must be a 1 for subtract

and - 00
or - 01
adder - 10
less - 11
mux

- We have these inputs…

func<5:0>
36 (and) = 1 0 0 1 0 0
37 (or) = 1 0 0 1 0 1
32 (add) = 1 0 0 0 0 0
34 (sub) = 1 0 0 0 1 0
42 (slt) = 1 0 1 0 1 0

can ignore these (they're the same for all…)

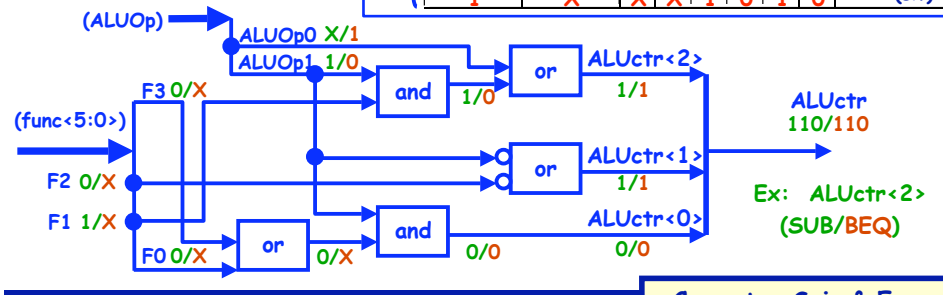
	Inputs: ALUOp		Inputs: Funct field						Outputs
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	ALUctr
lw/sw	0	0	X	X	X	X	X	X	010 (add)
beq	0	1	X	X	X	X	X	X	110 (sub)
R-type	1	X	X	X	0	0	0	0	010 (add)
	1	X	X	X	0	0	1	0	110 (sub)
	1	X	X	X	0	1	0	0	000 (and)
	1	X	X	X	0	1	0	1	001 (or)
	1	X	X	X	1	0	1	0	111 (slt)

The Logic

This table is used to generate the actual Boolean logic gates that produce ALUctr.

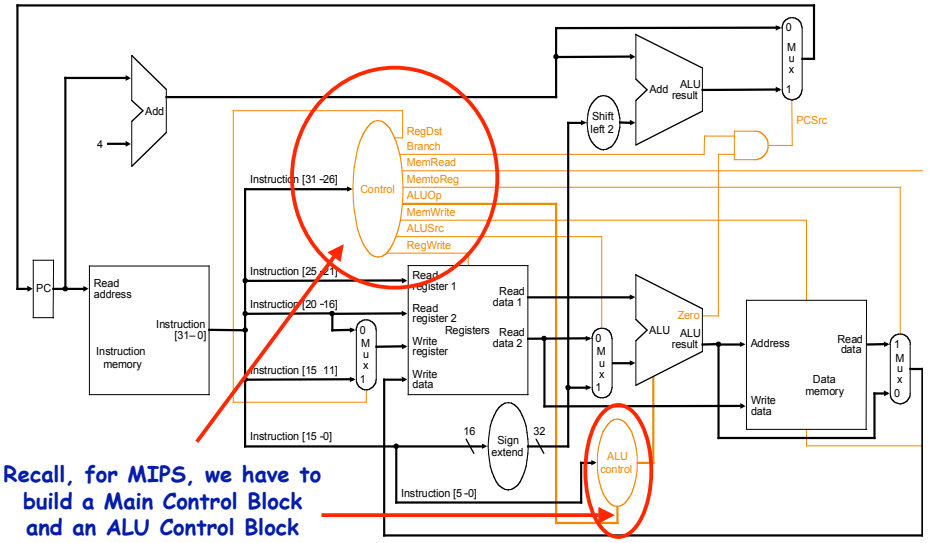
Could generate gates by hand, often done w/SW.

	Inputs: ALUOp		Inputs: Funct field						Outputs
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	ALUctr
lw/sw	0	0	X	X	X	X	X	X	010 (add)
beq	0	1	X	X	X	X	X	X	110 (sub)
R-type	1	X	X	X	0	0	0	0	010 (add)
	1	X	X	X	0	0	1	0	110 (sub)
	1	X	X	X	0	1	0	0	000 (and)
	1	X	X	X	0	1	0	1	001 (or)
	1	X	X	X	1	0	1	0	111 (slt)



Recall…

Single cycle MIPS machine

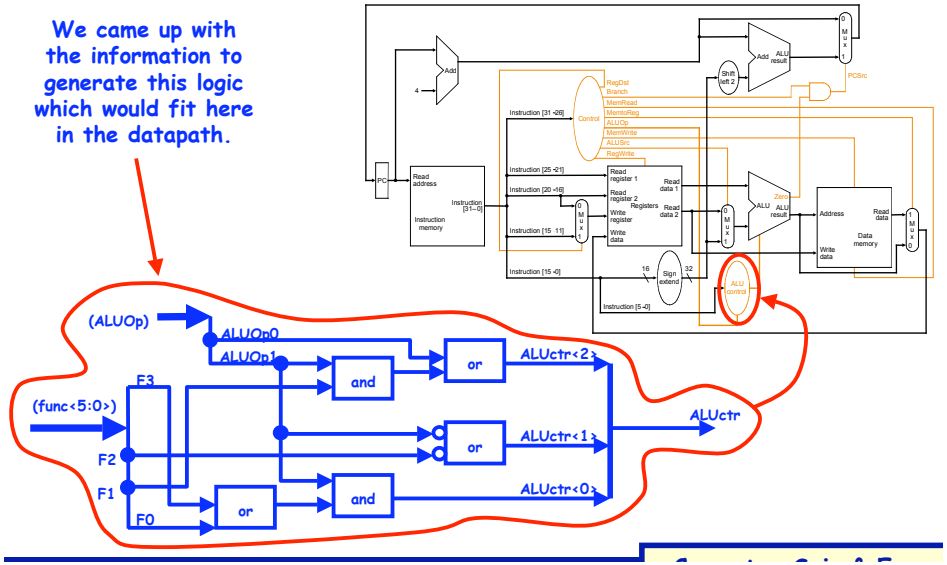


Recall, for MIPS, we have to build a Main Control Block and an ALU Control Block

Well, here's what *we* did…

Single cycle MIPS machine

We came up with the information to generate this logic which would fit here in the datapath.



(and again, remember, realistically logic, ISAs, insturction types, etc. would be *much* more complex)

(we'd also have to route all signals too…which may affect how we'd like to organzie processing logic)

Single cycle versus multi-cycle

Single-Cycle Implementation

- Single-cycle, fixed-length clock:
  - CPI = 1
  - Clock cycle = propagation delay of the longest datapath operations among all instruction types
  - Easy to implement
- Single-cycle, variable-length clock:
  - CPI = 1
  - Clock cycle = Σ (%(type-i instructions) * propagation delay of the type "i" instruction datapath operations)
  - Better than the previous, but impractical to implement
- Disadvantages:
  - What if we have floating-point operations?
  - How about component usage?

Multiple Cycle Alternative

- Break an instruction into smaller steps
- Execute each step in one cycle.
- Execution sequence:
  - Balance 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, why?
    - Introduce additional "internal" registers
- The advantages:
  - Cycle time much shorter
  - Diff. inst. take different # of cycles to complete
  - Functional unit used more than once per instruction