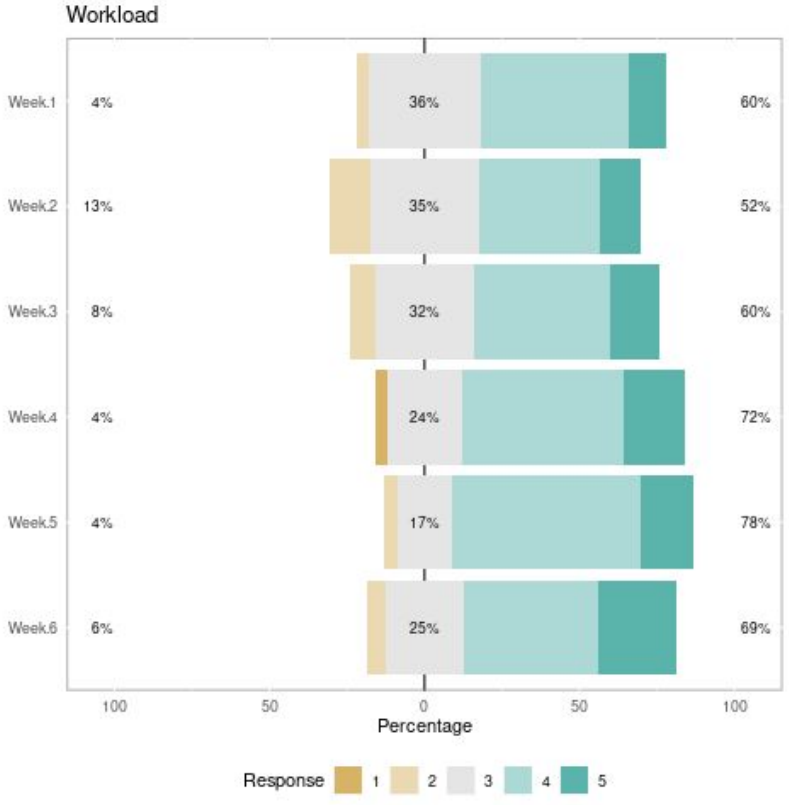
0x0B - Instruction Encoding

ENGR 3410: Computer Architecture

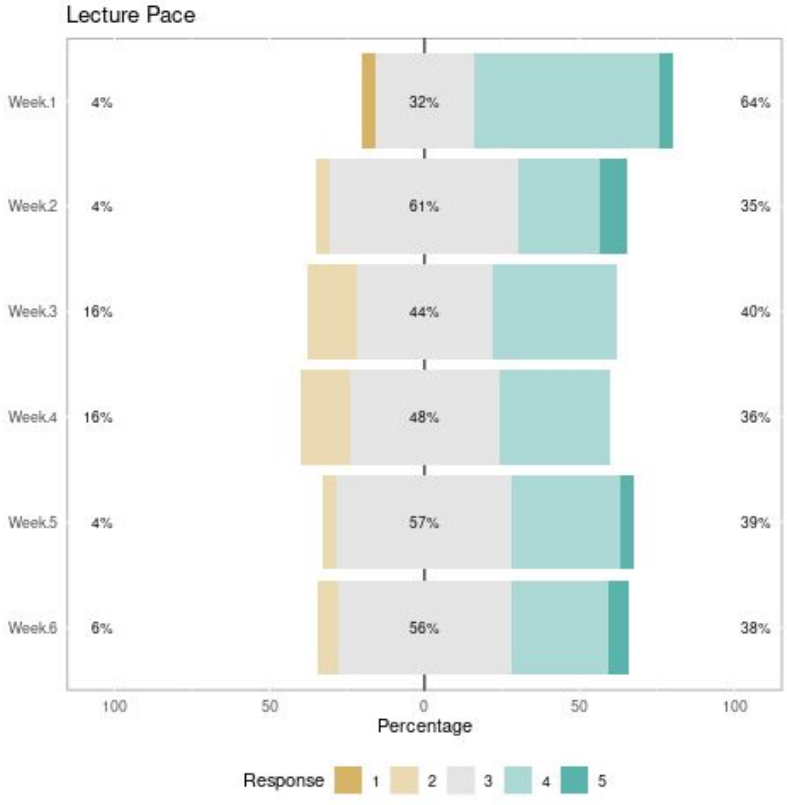
Jon Tse

Fall 2020

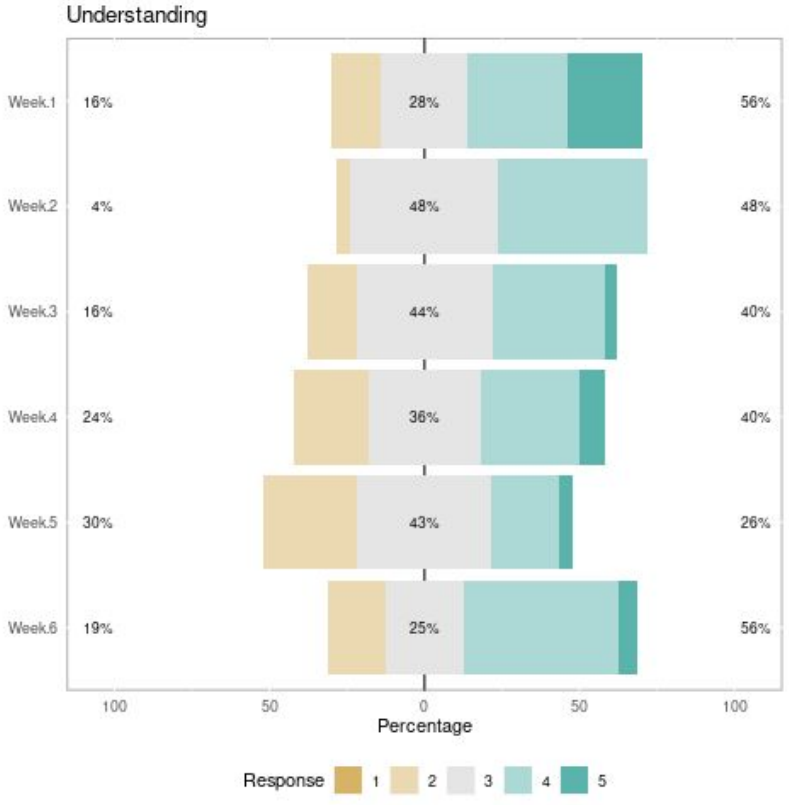
Feedback - Workload



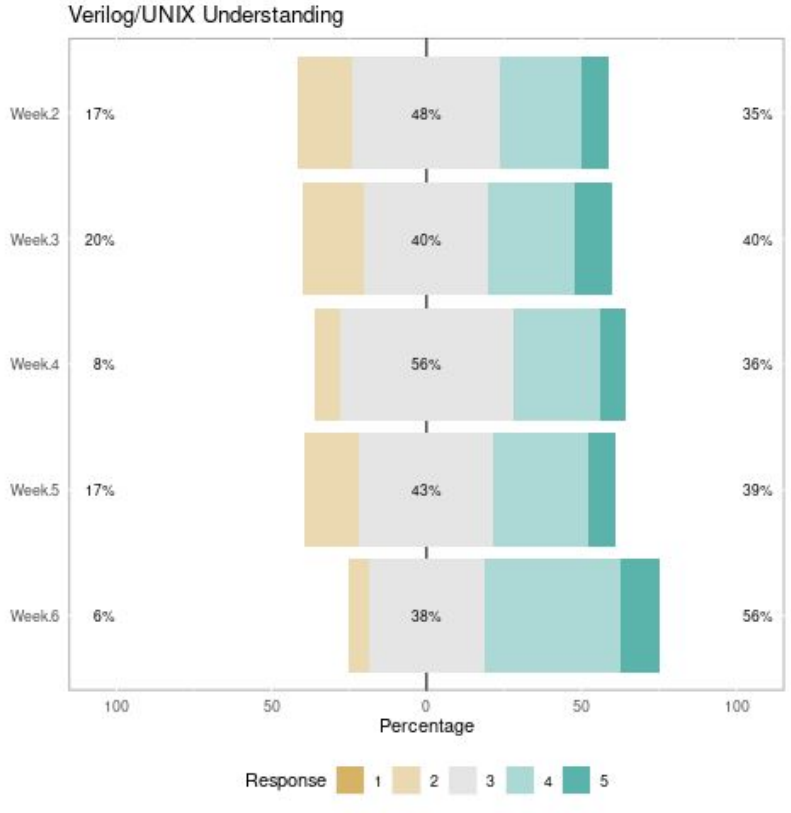
Feedback - Pace



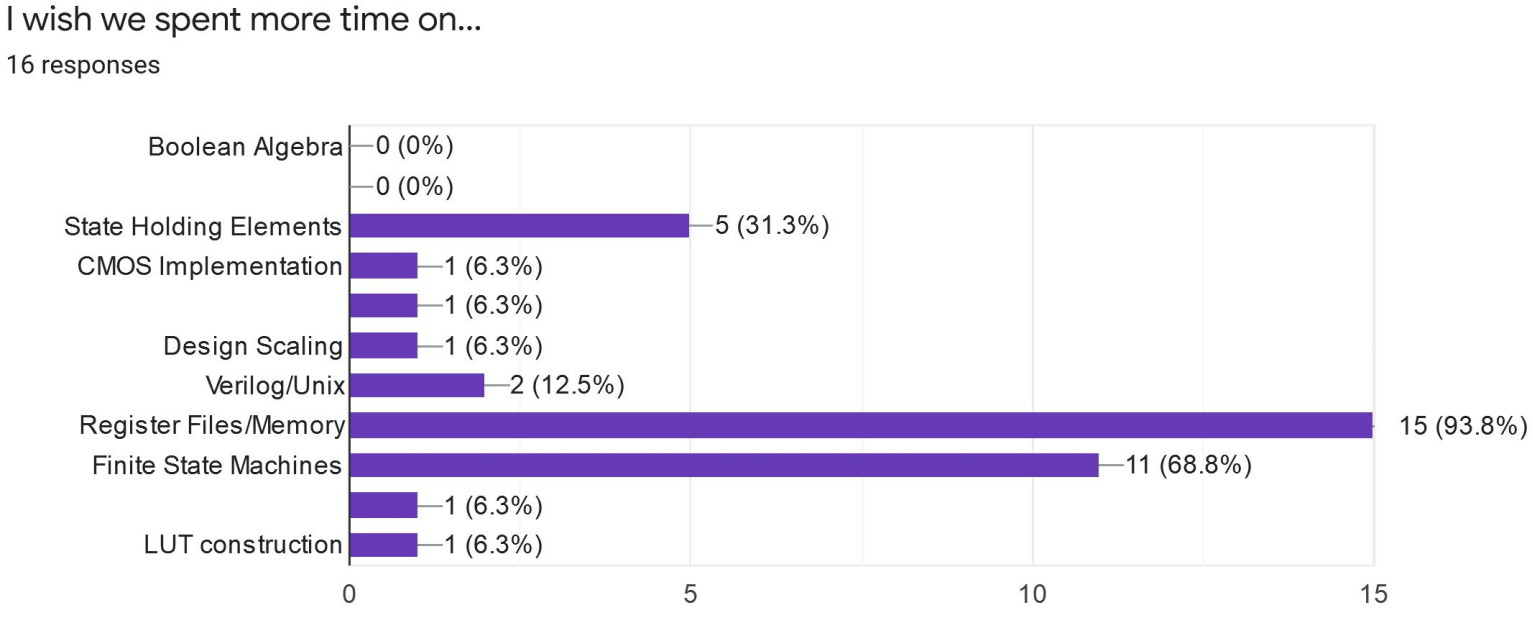
Feedback - Understanding



Feedback - Tools



Feedback - More time on...



Additional Feedback

- Discrete “midterm,” where are my lab grades?
- FSM and Regfile need attention
- Request to *not* pre-define Verilog module
  - That’s how it’s done industry -- agree on portlist
  - Just wait till Final Project
- “HW wasn’t too bad once I realized I over complicated. A little frustrated that things were not clarified”

Final Project Ideas

- Survey Paper
  - Research a subject
  - Write a summary paper exploring subject
- CPU-based Project
  - Extend your CPU in some way
- Implement something
  - Game of Life
  - Minecraft CPU
  - Something on FPGA

Housekeeping

- Midterm is live
- HW6 and Lab 3 Done!
- Lab 4 next…
- Git Resource: https://git-scm.com/

Today

- Instruction Decoding
- Instruction Encoding
- Continue Single Cycle CPU
- Jumps and Branches

Single Cycle Design Process

- Instructions
- RTL – Register Transfer Level
  - Describes how the instruction is executed
  - How information flows between registers
- Schematic
  - Arranges building blocks to implement RTL

Review

- Math with 2 Variables

```
Reg[rd] = Reg[rs] op Reg[rt];
```

- Math with 1 variable and 1 constant

```
Reg[rt] = Reg[rs] op SignExtend(imm);
```

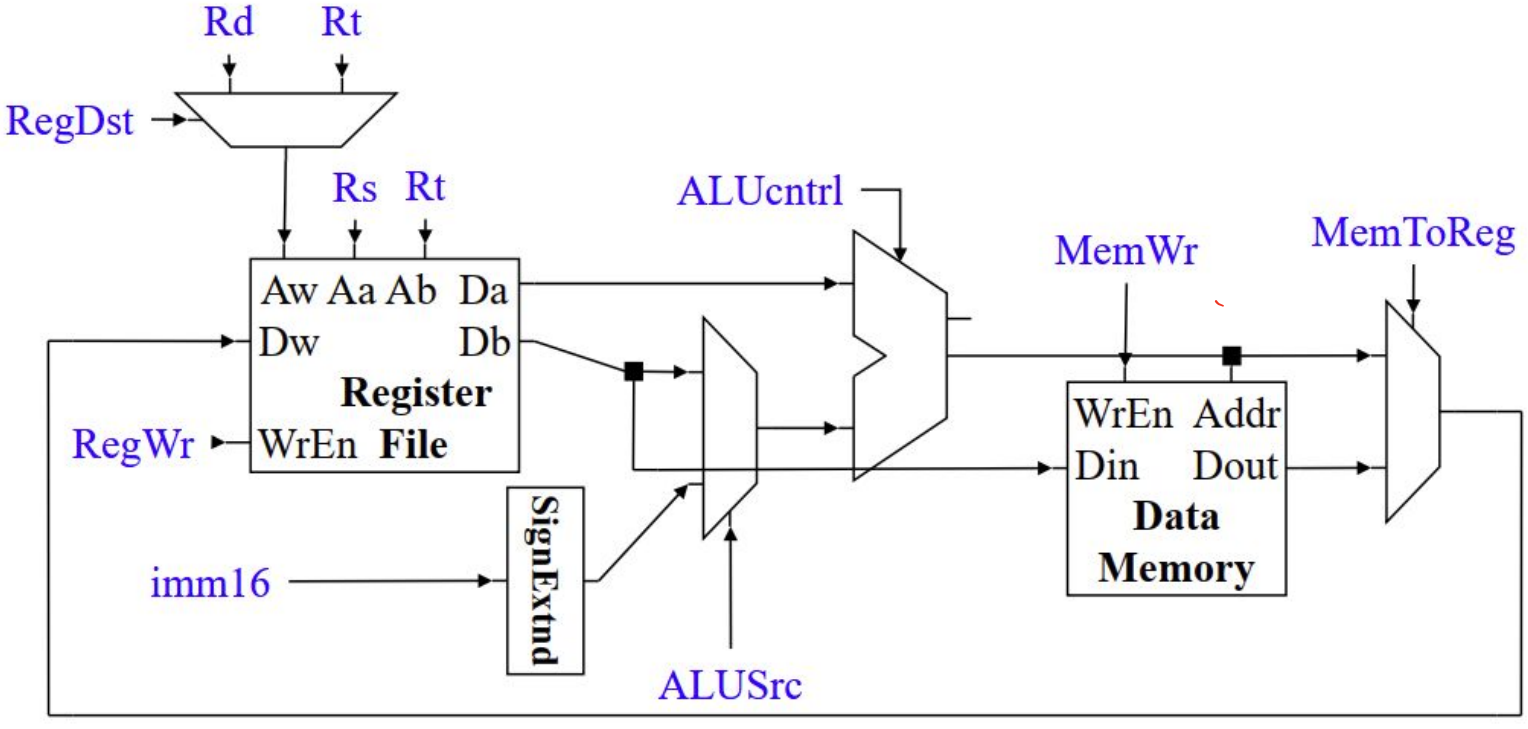
- Load from data memory

```
Addr = Reg[rs] + SignExtend(imm);
Reg[rt] = Mem[Addr];
```

- Store to data memory

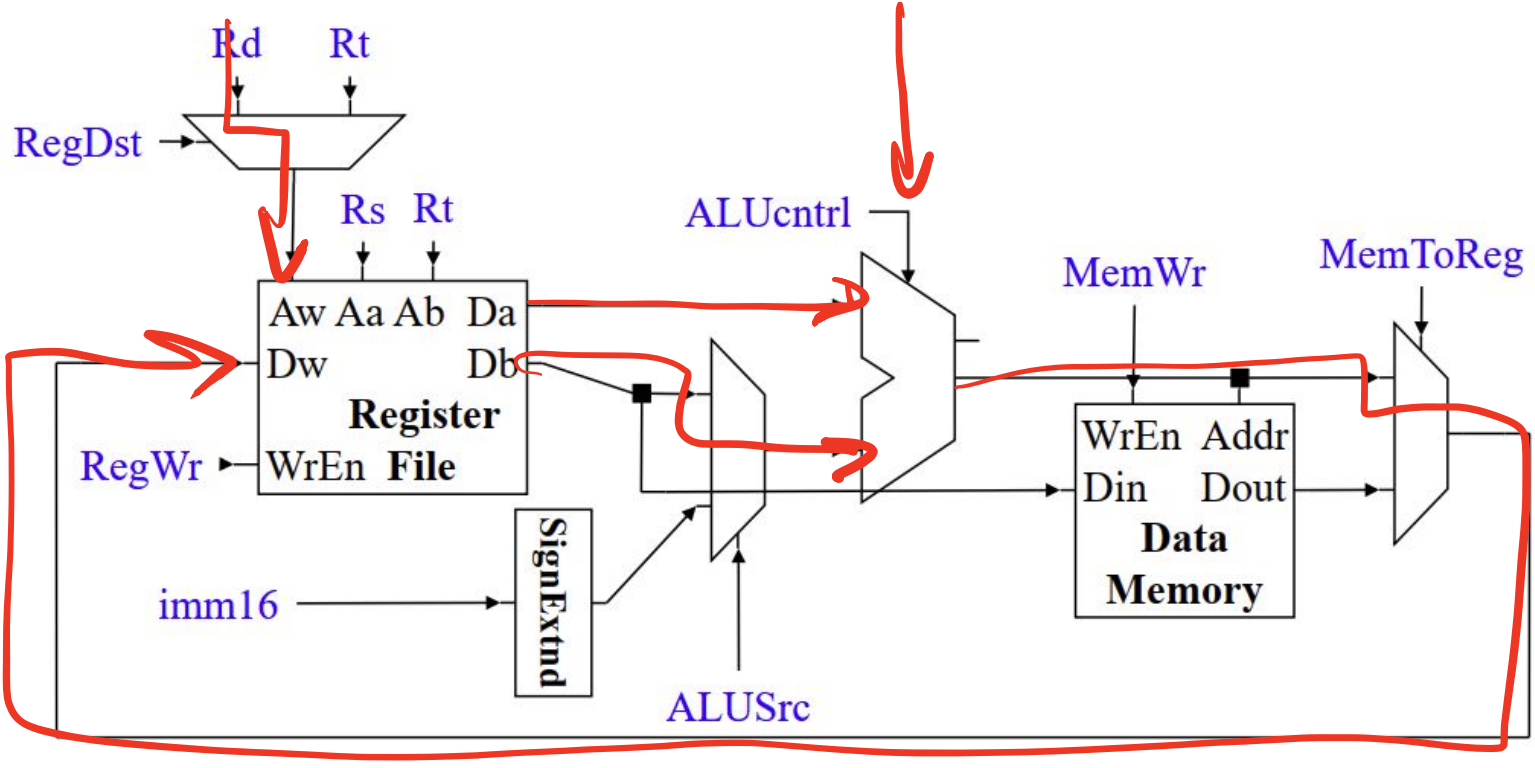
```
Addr = Reg[rs] + SignExtend(imm);
Mem[Addr] = Reg[rt];
```

Review - Complete Datapath



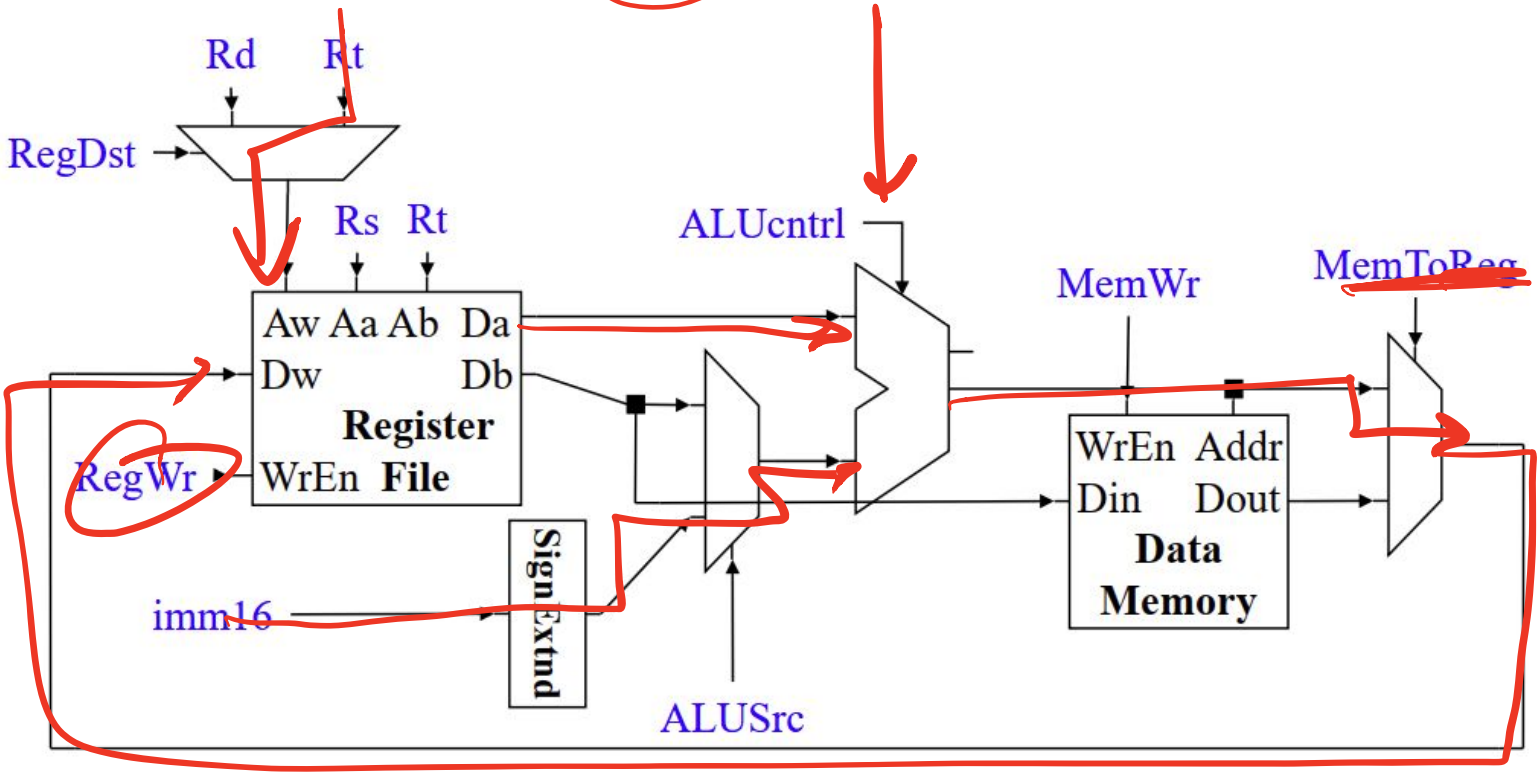
Math with 2 Variables

```
Reg[rd] = Reg[rs] op Reg[rt];
```



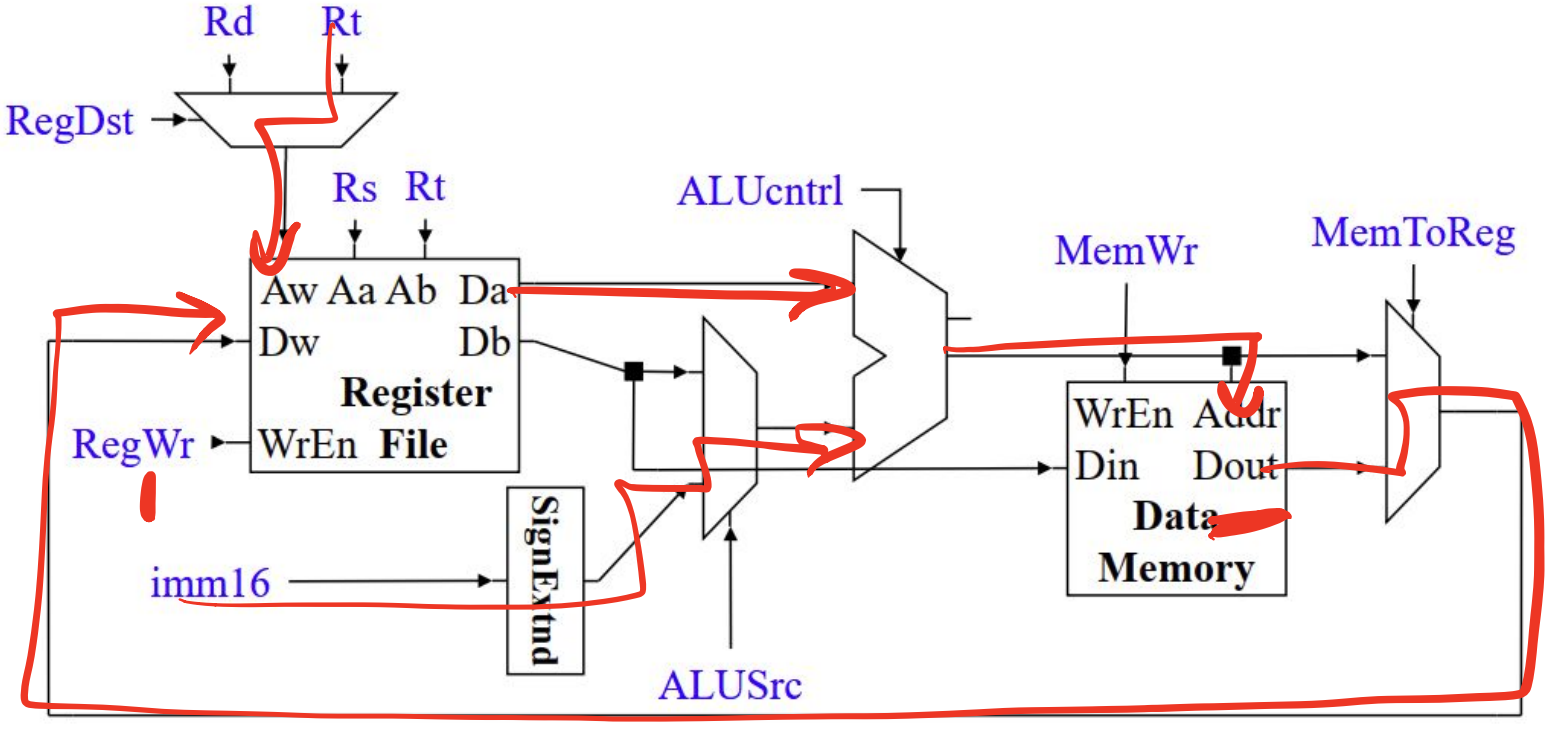
Math with 1 variable and 1 constant

```
Reg[rt] = Reg[rs] op
SignExtend(imm);
```



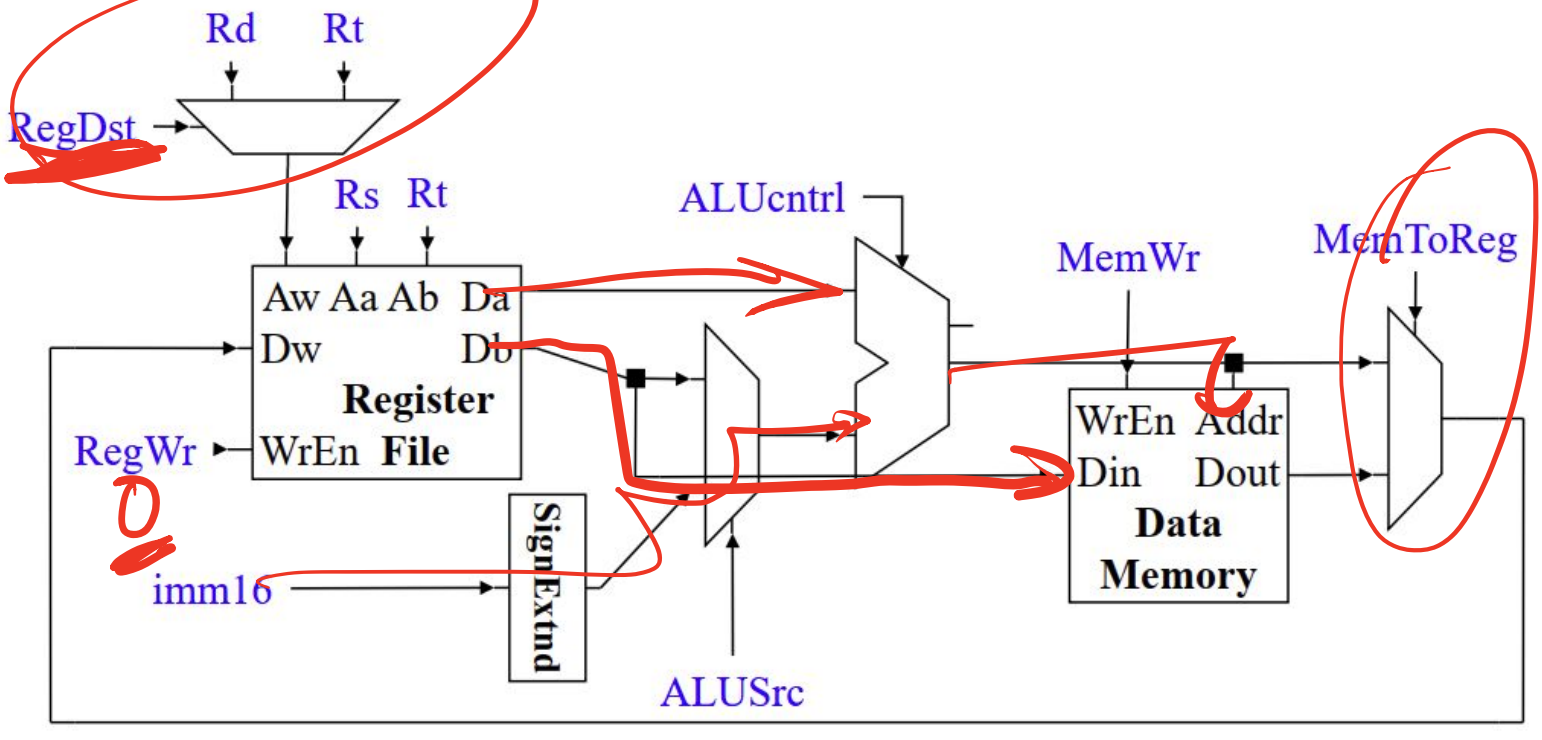
Load from data memory

```
Addr = Reg[rs] + SignExtend(imm);
Reg[rt] = Mem[Addr];
```

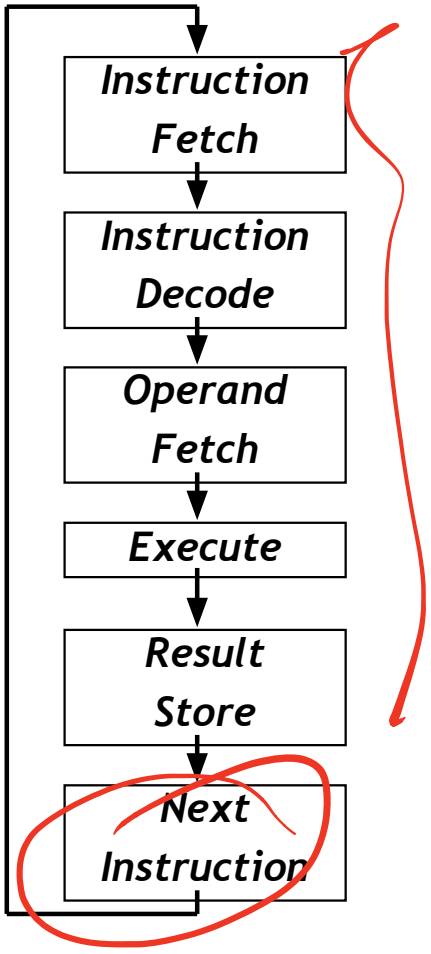


Store to data memory

```
Addr = Reg[rs] + SignExtend(imm);
Mem[Addr] = Reg[rt];
```



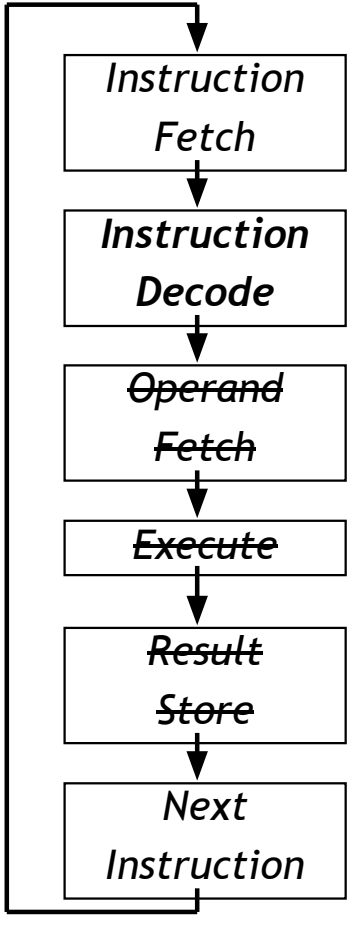
What boxes do we need?



5 stage pipeline

- Memory for Instructions
- Something to "unpack" instructions
- Memory for data
- Something to compute stuff

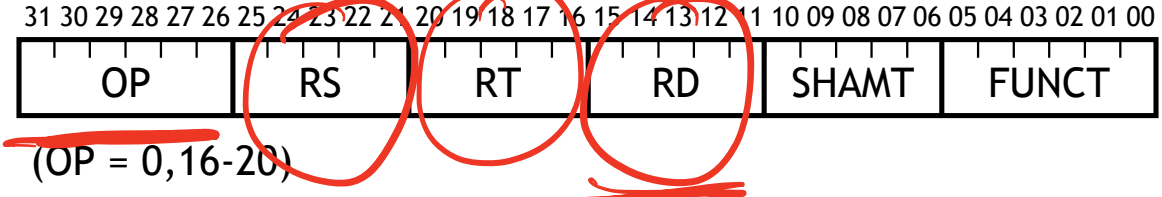
What boxes do we need?



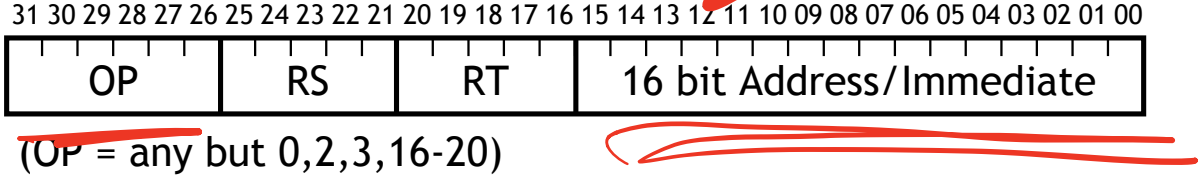
- Memory for Instructions
- **Something to "unpack" instructions**
- ~~Memory for data~~
- ~~Something to compute stuff~~

MIPS Code Encoding Formats

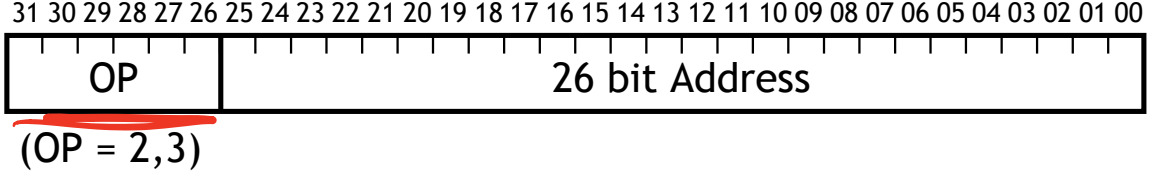
- All instructions encoded in 32 bits
- Register (R-type) instructions



- Immediate (I-type) instructions

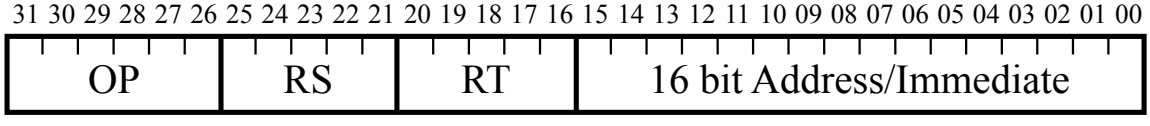


- Jump (J-type) instructions



I-Type

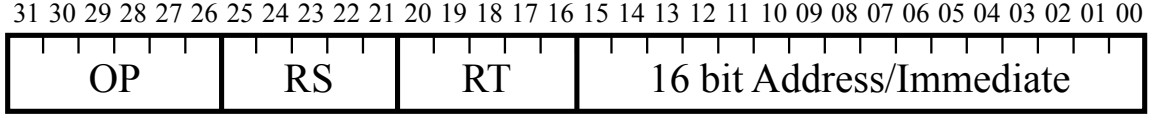
- Used for ops with an immediate operand



- One Op Field (Enumeration)
- Two register address fields
- One Signed/Unsigned field

I-Type Example

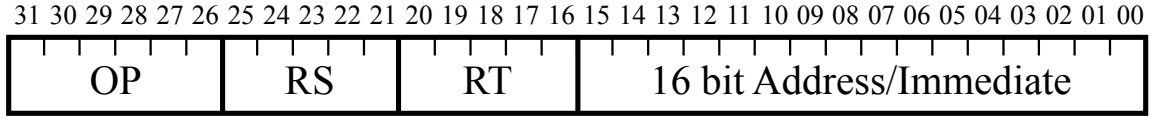
Register 7 = Register 2 + 15;

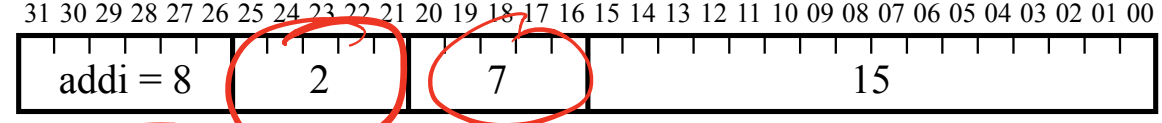


I-Type Example

RT RS

Register 7 = Register 2 + 15;

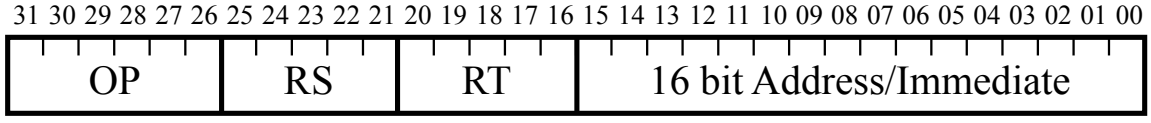


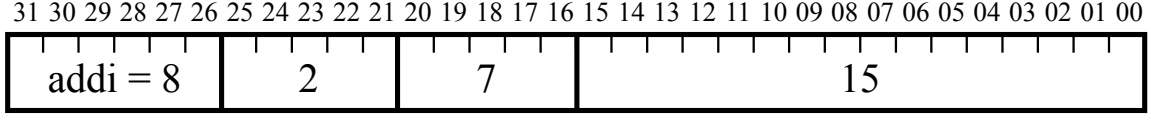


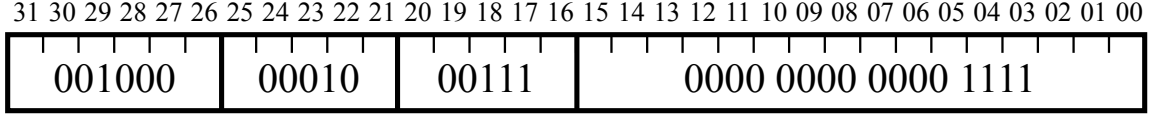
04: beq
05: bne
06: blez
07: bgtz
08: addi
09: addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

I-Type Example

Register 7 = Register 2 + 15;

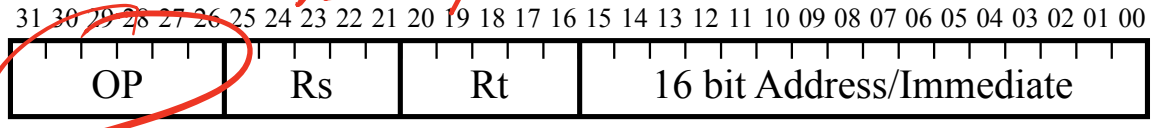


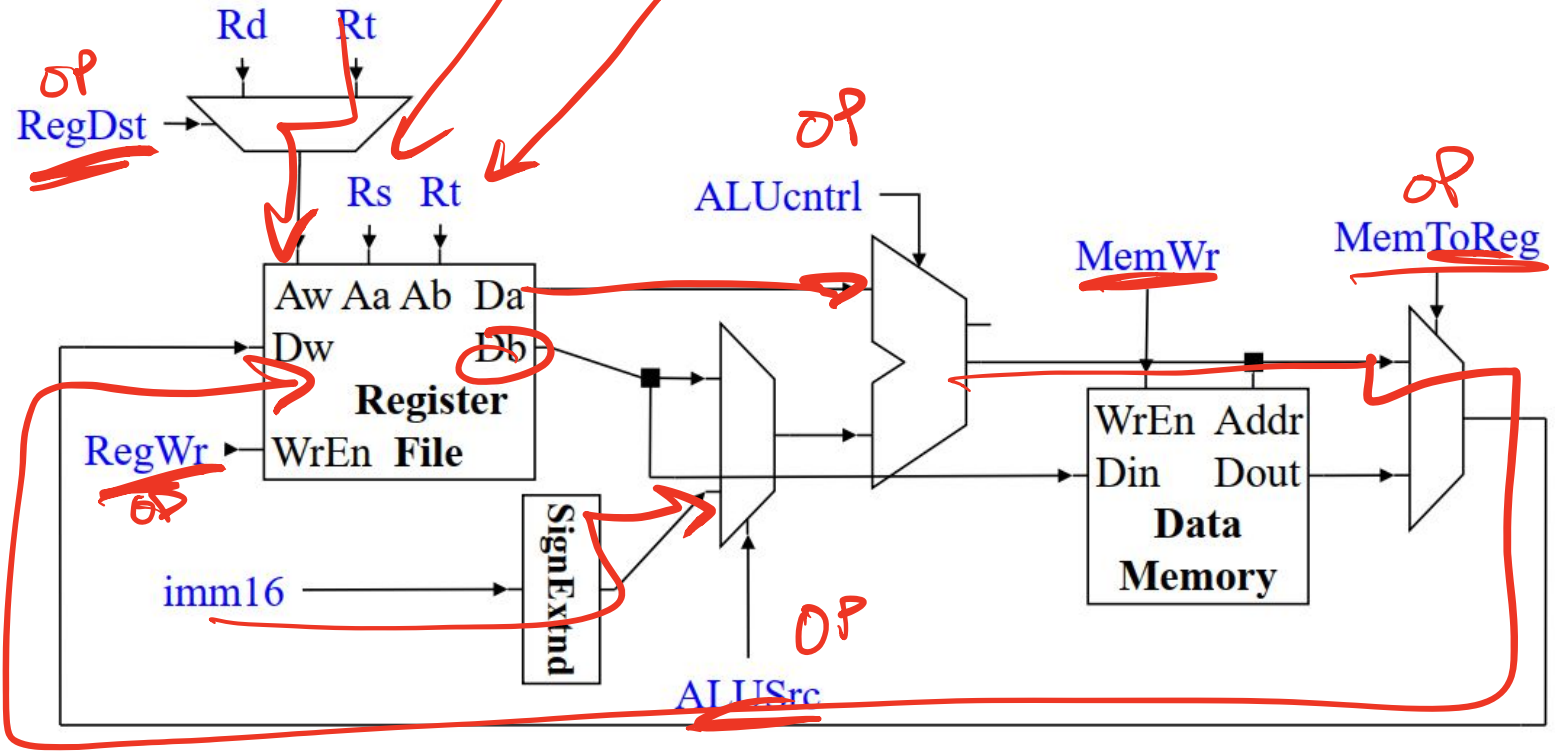




04: beq
05: bne
06: blez
07: bgtz
08: addi
09: addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

Instruction -> CPU Controls



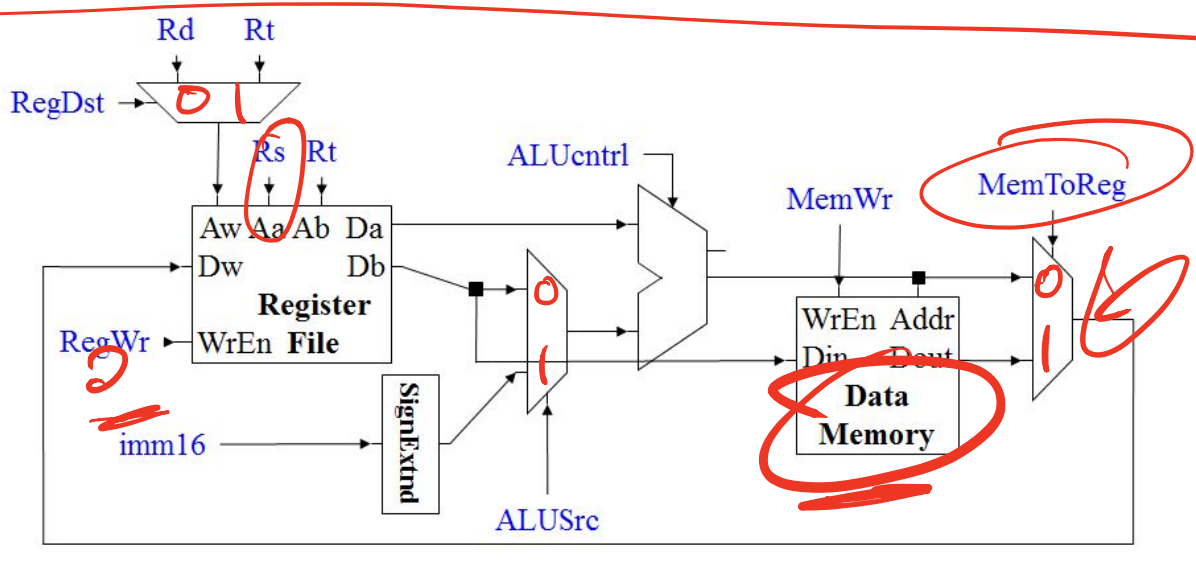


OP Decoding: LUT

- Rs, Rt, Immediate, etc are directly passed
  - Immediate gets sign extended
- Op gets decoded to create other controls
  - Record all signals in a table on paper
  - Translate to machine
  - Implement as LUT

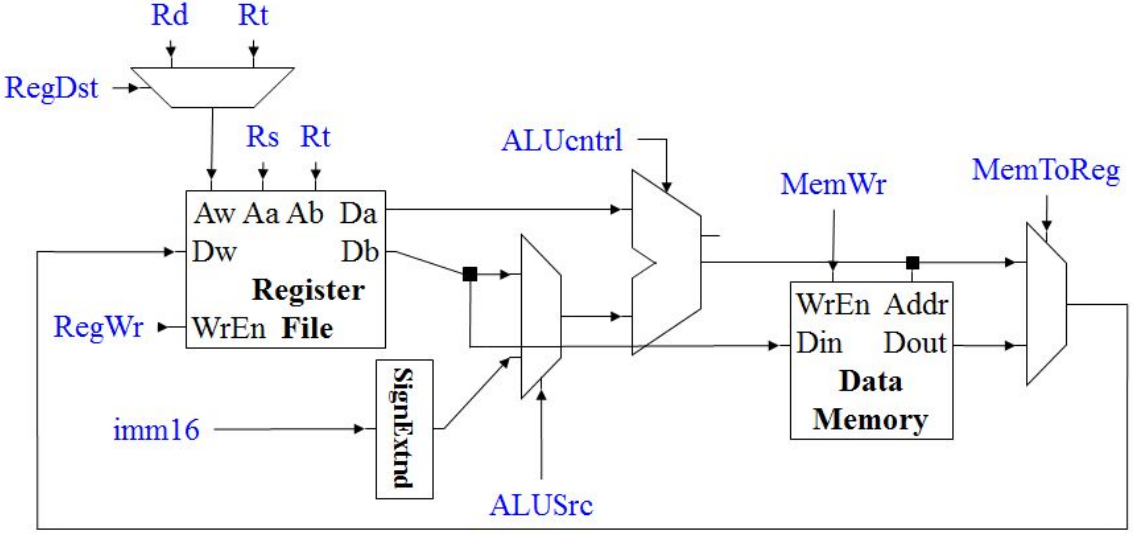
Example Table

	RegDst	RegWr	ALUcntrl	MemWr	MemToReg	ALUsrc
Math w/2	0	1	op	0	0	0
Math w/ Imm	1	1	op	0	0	1
Load	1	1	add	0	1	1
Store	1	0	add	1	X	1



```
Reg[rt] = Reg[rs] op SignExtend(imm);
```

	RegDst	RegWr	ALUcntrl	MemWr	MemToReg	ALUsrc
Math w/2						
Math w/ Imm						
Load						
Store						



(op)?

- Many ops will be extremely similar:
  - Add immediate
  - Add registers
  - Subtract immediate
  - Subtract registers

- Synthesizer will optimize for size
  - Repetition is something it is good at

Behavioral Instruction Decoder LUT

- "always @(…)" begins a behavioral block of code with a "sensitivity list"
- Triggers on sensitivities
- When OP changes, run this block of code

```
always @(OP) begin
   case(OP)
     opADDI: begin
        RegDst = rt
        RegWr = 1
        ALUcntrl = add
        MemWr = 0
        MemToReg = alu
        ALUsrc = imm
      end
   endcase
end
```

Behavioral Instruction Decoder LUT

- Just like any other case statement you know and love
- Equivalent to

```
If(OP==opADDI) {…}
Elseif(OP==???) {…}
```

```
always @(OP) begin
  case(OP)
    opADDI: begin
      RegDst =
      RegWr =
      ALUcntrl =
      MemWr =
      MemToReg =
      ALUsrc =
    end
  endcase
end
```

Behavioral Instruction Decoder LUT

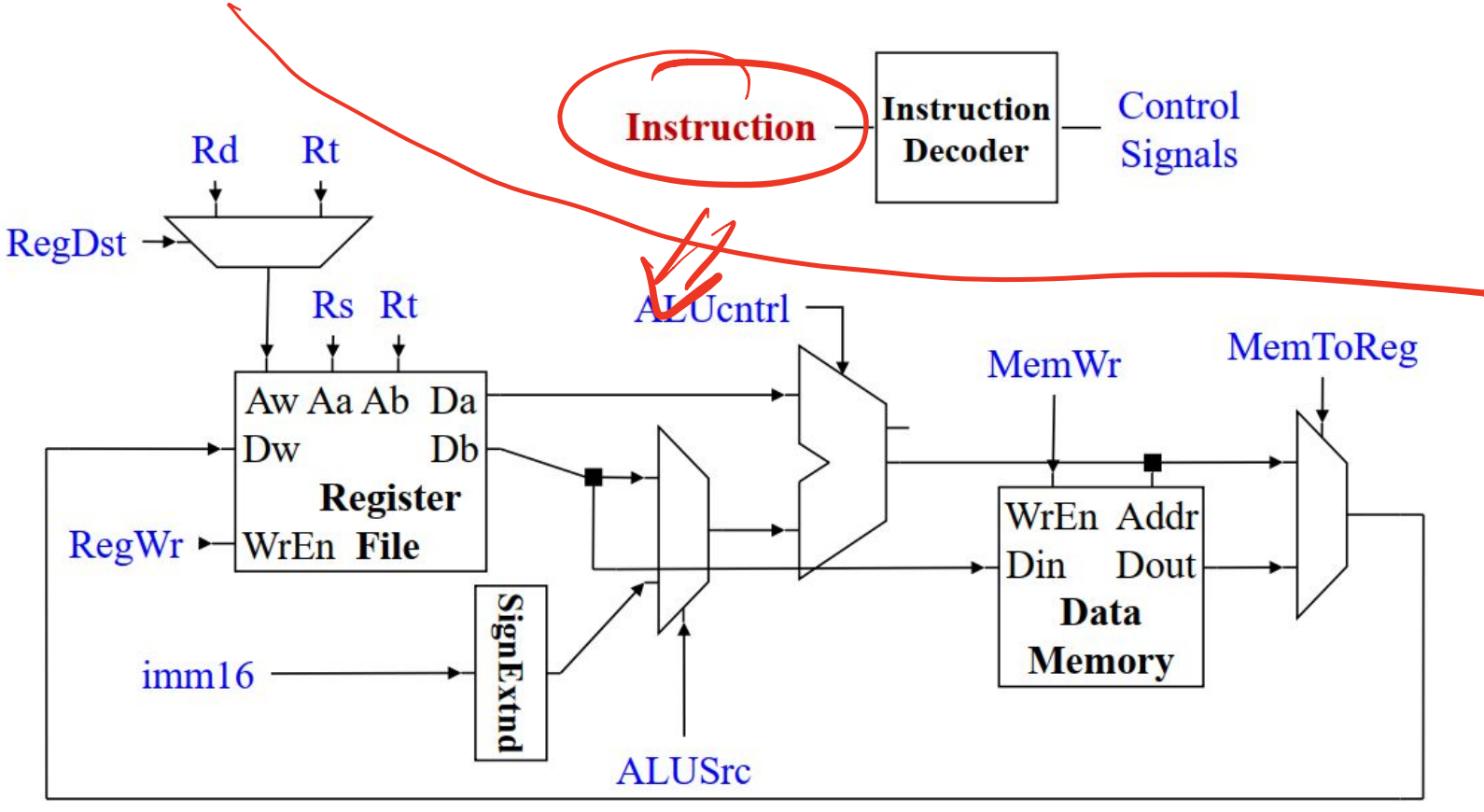
- Assigns each of these control signals the appropriate value for the operation.
- Copy this from your table

```
always @(OP) begin
  case(OP)
    opADDI: begin
      RegDst =
      RegWr =
      ALUcntrl =
      MemWr =
      MemToReg =
      ALUsrc =
    end
  endcase
end
```

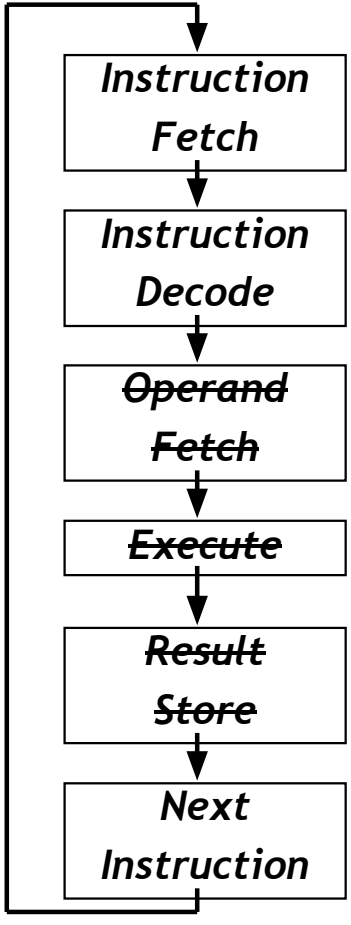
Behavioral LUT

- “reg” for each output
  - These will actually optimize away
  - Only if written correctly!!!
- Registers will stay (this is bad) if:
  - One of your cases is incomplete
  - You do not cover all cases
    - default: helps with this.

Our CPU So Far:



What boxes do we need?



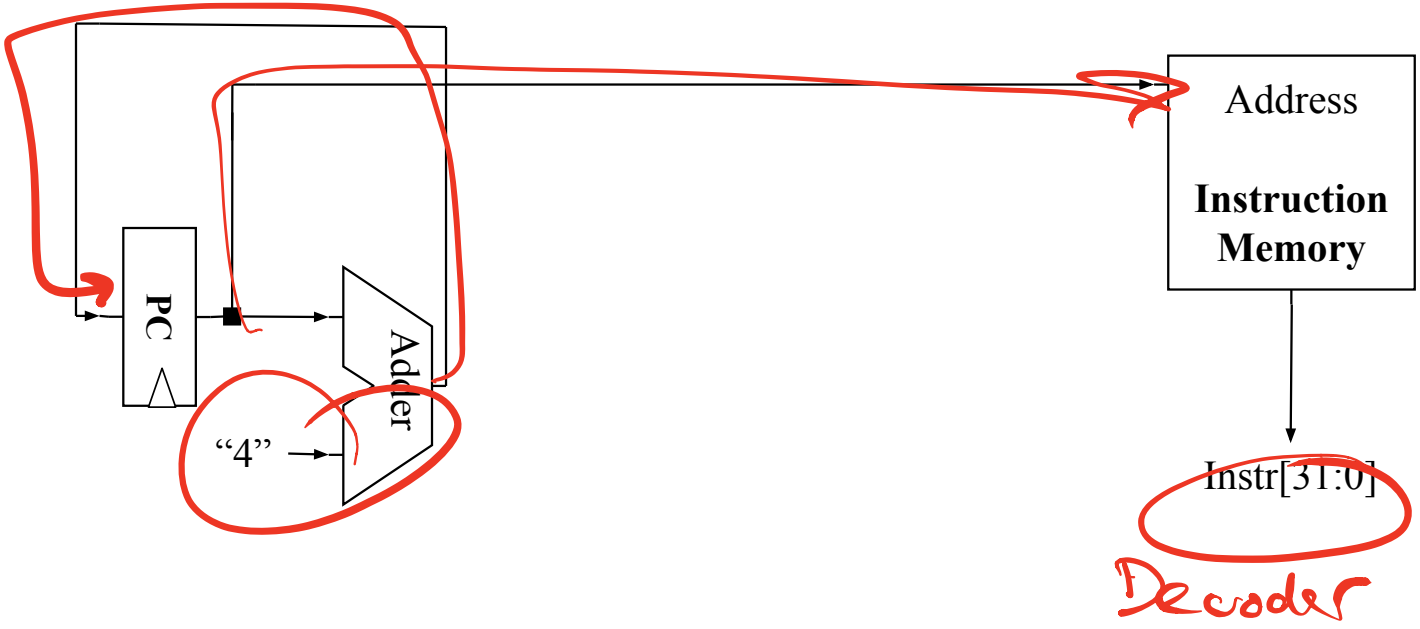
- **Memory for Instructions**
- ~~Something to "unpack" instructions~~
- ~~Memory for data~~
- ~~Something to compute stuff~~

Program Counter

- Where are we in our program?
  - Points to an instruction *in program memory*
- Instructions are 4 bytes long in MIPS
- Increment by 4 to get the next instruction

Fetch Instruction

- Instruction = Mem[PC]; // Fetch Instruction
- PC = PC + 4;// Increment **P**rogram **C**ounter



Flow Control

C code example:

```
for(int i = 0; i<N; i++) {
    a = b + c;
    b *= 3;
}
```

- Jump – Unconditionally change PC
- Branch – Conditionally change PC

Flow Control

```
for(int i = 0; i<N; i++) {
  a = b + c;
  b *= 3;
}
```

- Jump – Unconditionally change PC
  - Jump back to the beginning of the loop
- Branch – Conditionally change the PC
  - Escape loop if i<N fails

Add Flow Control

- Modify the Instruction Fetch Unit to Branch:

Branch RTL

```
if ((Reg[rs] - Reg[rt]) == 0)
     PC = PC +4+ SignExtend(imm)*4;
else
     PC = PC + 4;
```

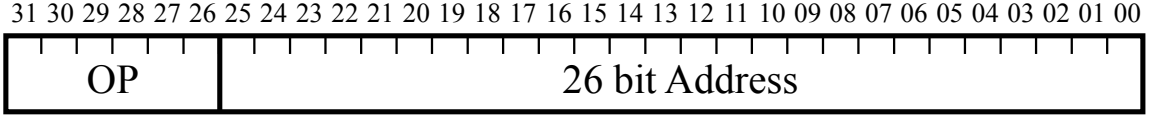
- Write RTL for your version of Jump
  - What does Jump mean to you?
- Implement your RTL
  - What new control lines do you need?
- Create an instruction encoding for this

Difficulties In Jumping

- How did you define your Jump?
  - Absolute: PC = New Value
  - Relative: PC = PC + Offset
- Did you have enough bits?

J-Type

- Used for Unconditional Jumps
- Simplest MIPS encoding



2: j (jump)

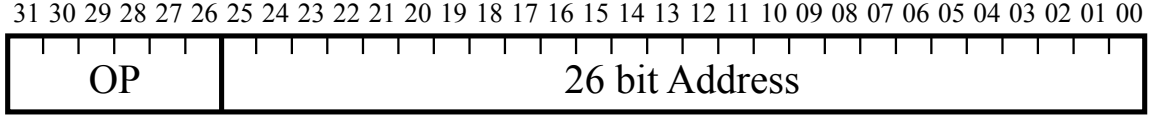
3: jal (jump and link)

- How do we encode “j 100”?

Jump RTL

Jump Instruction: `j target`

```
PC = { PC[31:28], target[25:0], "00" };
```

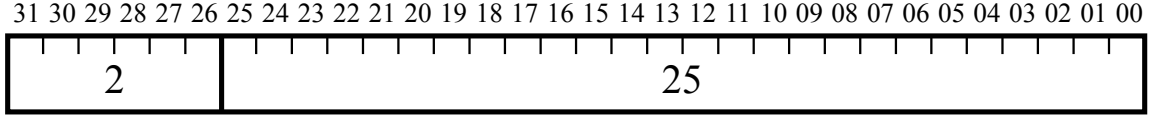


- Weirdness in the Address Field
- Bottom 2 bits are always '00', so drop them
  - Shift everything over by 2
- That's only 28 effective bits…
  - Where are the other 4?
  - How does this limit us?
  - How do we compensate?

Jump RTL

Jump Instruction: `j 100`

```
PC = { PC[31:28], target[25:0], "00" };
```



- Weirdness in the Address Field
- Bottom 2 bits are always ‘00’, so drop them
  - Shift everything over by 2
- That’s only 28 effective bits…
  - Where are the other 4?
  - How does this limit us?
  - How do we compensate?

Jump and Branch

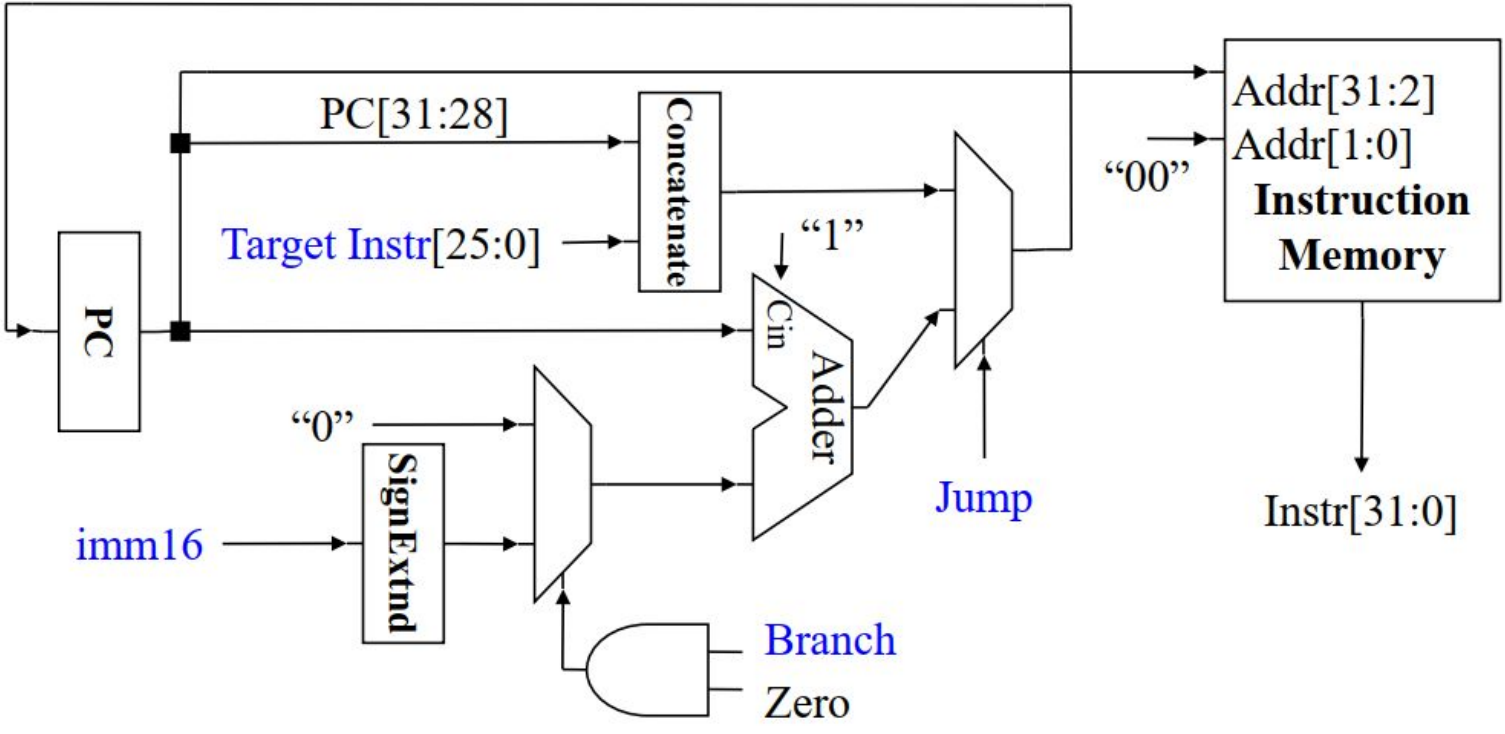
MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

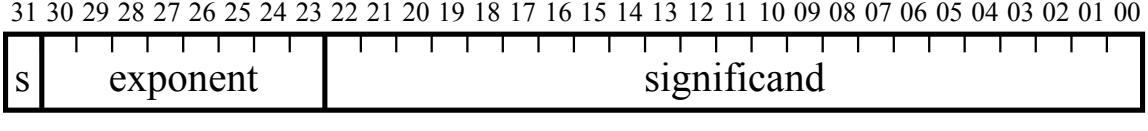
Complete Fetch Unit



Encoding Limitations

- Designing an Encoding Scheme is a Game
  - Best use of limited space?
  - Favor some options over others

- Take Available Space and divide into “Fields”
  - Encoding within an Encoding
  - Fixed width per field
  - How many fields does IEEE-754 use?

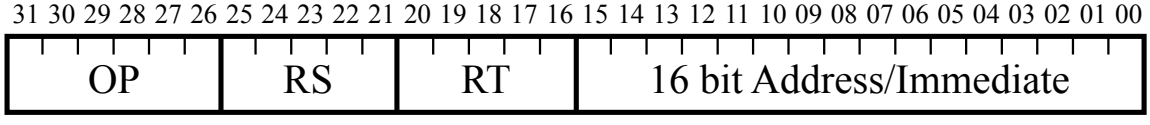


Types of Fields

- Enumerations
  - No mathematical meaning, just enumerate options
- Signed / Unsigned
  - We know these well
- Biased
  - Mathematically offset by a constant

I-Type

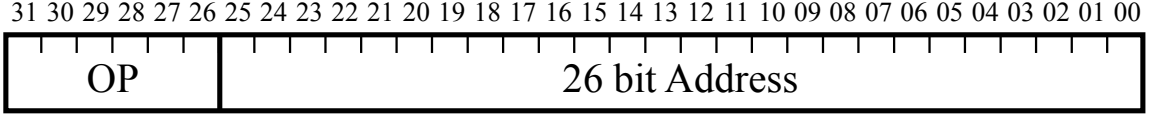
- Used for ops with an immediate operand



- One Op Field (Enumeration)
- Two register address fields
- One Signed/Unsigned field

J-Type

- Used for Unconditional Jumps
- Simplest MIPS encoding

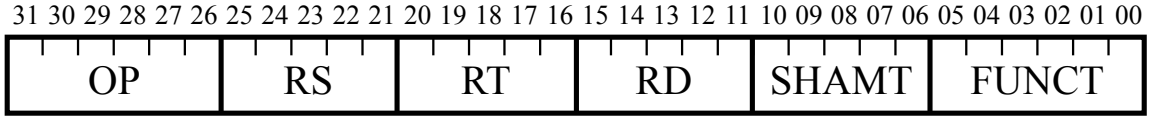


2: j (jump)

3: jal (jump and link)

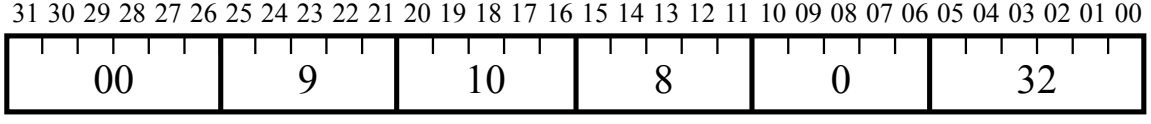
R-Type

- Used for 3 register ALU operations

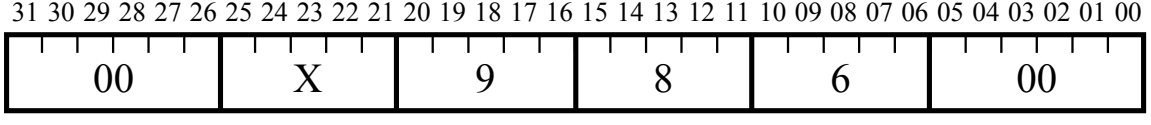


00: sll
02: srl
03: sra
04: sllv
06: srlv
07: srav
08: jr
24: mult
26: div
32: add
33: addu
34: sub
35: subu
36: and
37: or
38: xor
39: nor
42: slt

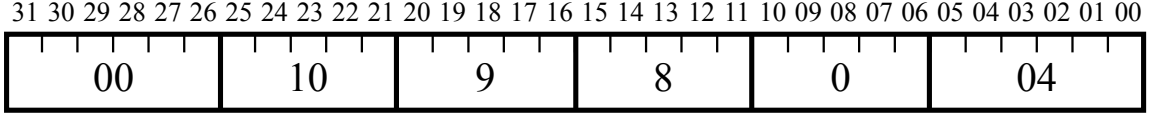
```
add $8, $9, $10  # $8 = $9+$10
```



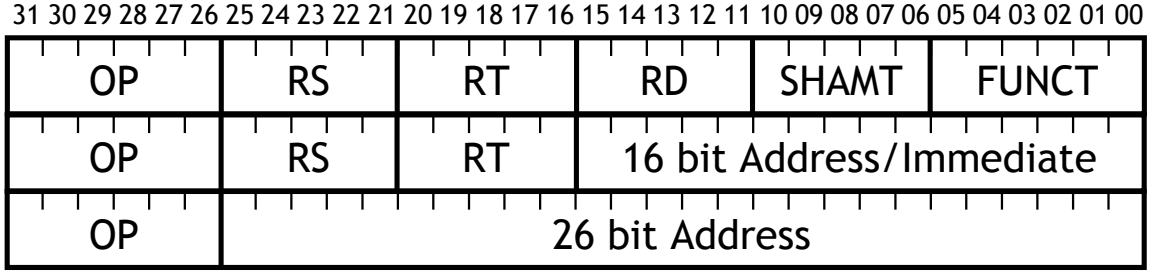

```
sll $8, $9, 6    # $8 = $9<<6
```




```
sllv$8, $9, $10  # $8 = $9<<$10
```



Register addresses (RS RT RD)

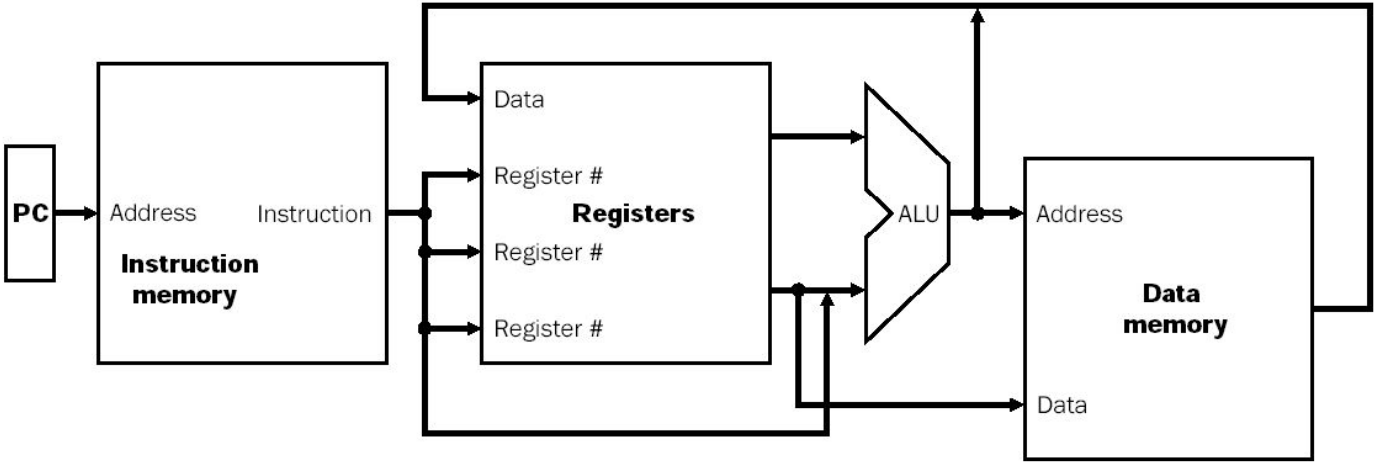


- RD – Bits 11-15 RegFile Aw
  - shares space with IMM, only used in R-type
  - If used, always the destination of the result
- RT – Bits 16-20 RegFile Aw or Ab
  - Operand address in R-type and rarely in I-type (e.g. beq)
  - Result destination address in most I-type
- RS – Bits 21-25 RegFile Aa
  - Operand address in R-type and I-type

Processor Overview

Overall Dataflow

- PC fetches instructions
- Instructions select operand registers, ALU immediate values
- ALU computes values
- Load/Store addresses computed in ALU
- Result goes to register file or Data memory



Instruction Set Design Practice

- Create a 24 bit instruction encoding
  - What will you change to fit?
    - Number of Registers?
    - Types of Operations allowed?
    - Limited Operand s?
- Questions:
  - What did you trim to fit?
  - How do you load a 32 bit immediate value?

- Needs to address these ops:
  - Math
  - Branch if zero
  - Jump
  - Load
  - Store
  - Load Immediate
    - Composite Instruction?

Revisiting ARM's Immediates

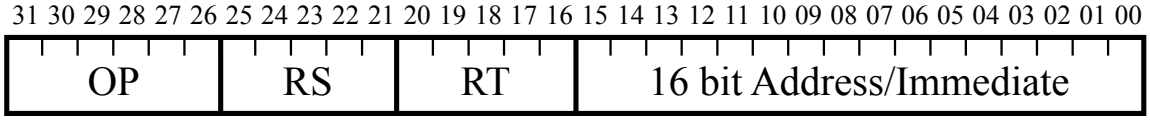
- In Thumb state in ARMv6T2 and later the 32-bit MOV instruction can load:
  - any 8-bit immediate value, giving a range of 0x0-0xFF (0-255)
  - any 8-bit immediate value, shifted left by any number
  - any 8-bit pattern duplicated in all four bytes of a register
  - any 8-bit pattern duplicated in bytes 0 and 2, with bytes 1 and 3 set to 0
  - any 8-bit pattern duplicated in bytes 1 and 3, with bytes 0 and 2 set to 0.
- These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.

- Create an Encoding Scheme to generate these Immediates

Create Your Own Instruction Encodings

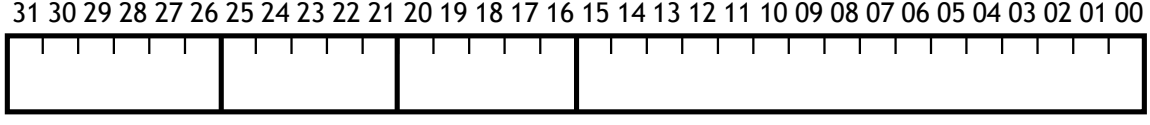
- Create an Encoding Spec that covers:
  - ALU Operations with 2 register operands
  - ALU Operations with immediate for one operand
  - Load from Data Memory, Store to Data Memory
  - Jump (What type?)
  - Branch
- Consider splitting your definition in to ‘types’
- Design an Instruction Decoder for it
  - Probably a LUT (plus other stuff?)

I-Type Examples

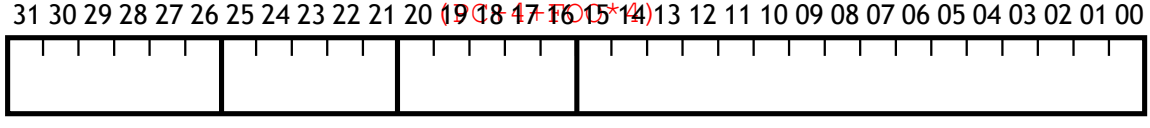


04: beq
05: bne
06: blez
07: bgtz
08: addi
09: addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

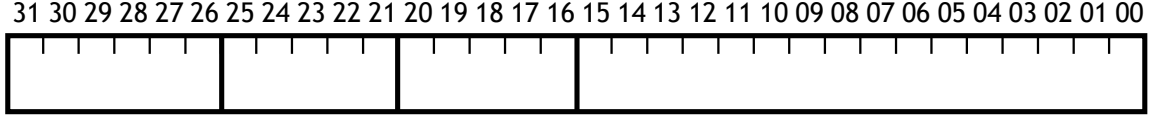
```
addi $t0, $t1, 100     # $t0 = $t1+100
```



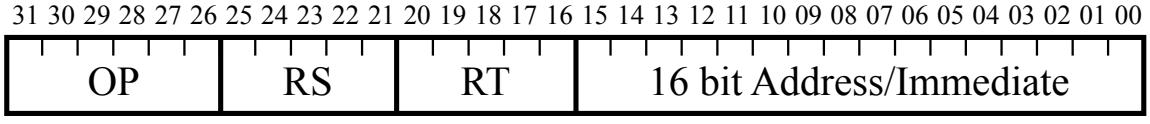

```
beq $a0, $a1, -44     # if $a0 == $a1 GOTO
```




```
lw  $t3, 12($t0) # $t3 = Memory[$t0+12]
```

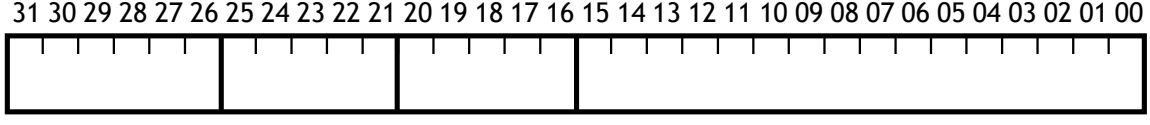


I-Type Examples

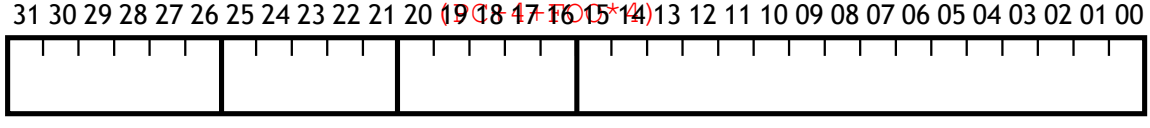


04: beq
05: bne
06: blez
07: bgtz
08: addi
09:
addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

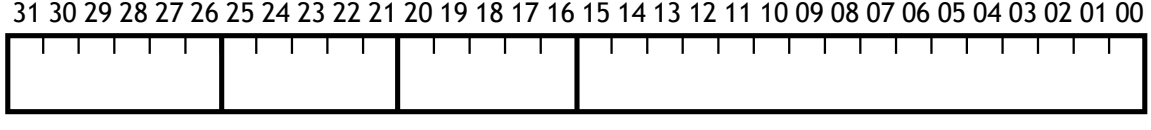
```
addi $t0, $t1, 100     # $t0 = $t1+100
```




```
beq $a0, $a1, -44     # if $a0 == $a1 GOTO
```




```
lw  $t3, 12($t0) # $t3 = Memory[$t0+12]
```

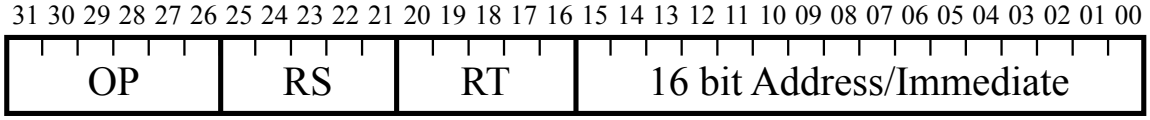


J-Type

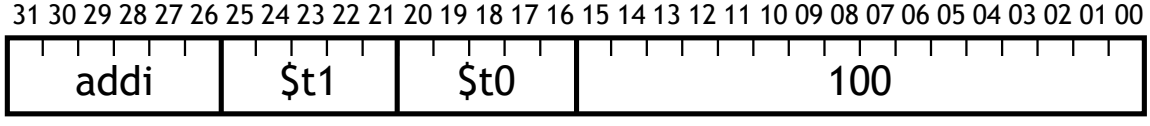
- Weirdness in the Address Field
- Bottom 2 bits are always ‘00’, so drop’em
  - Shift everything over by 2
- That’s only 28 effective bits…
  - Where are the other 4?
  - How does this limit us?
  - How do we compensate?

I-Type Examples

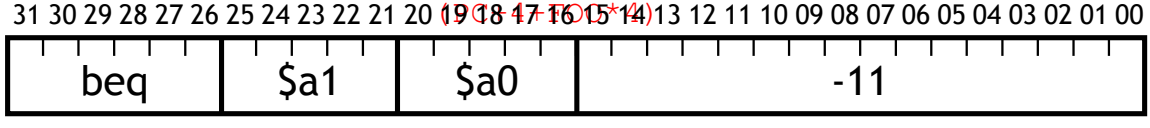
04: beq
05: bne
06: blez
07: bgtz
08: addi
09:
addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw



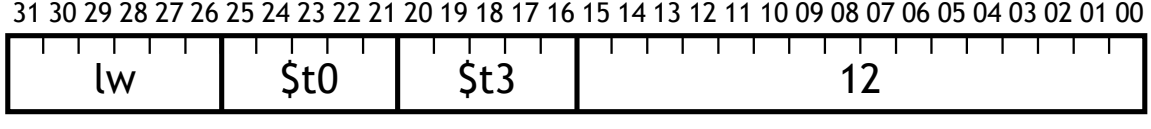

```
addi $t0, $t1, 100     # $t0 = $t1+100
```



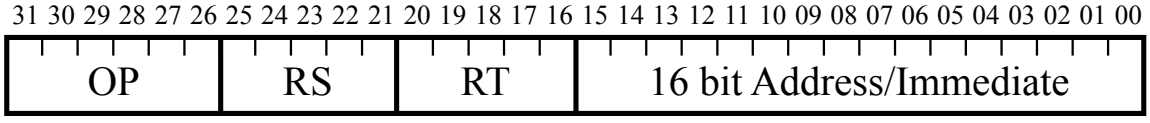

```
beq $a0, $a1, -44     # if $a0 == $a1 GOTO
```




```
lw  $t3, 12($t0) # $t3 = Memory[$t0+12]
```

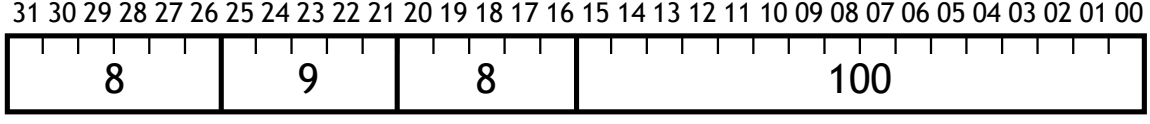


I-Type Examples

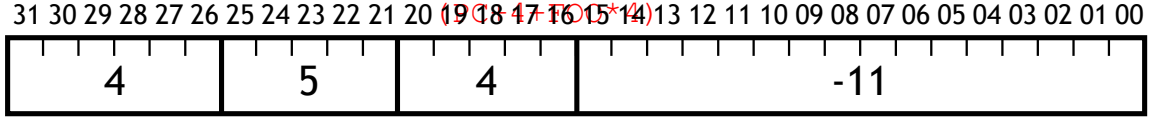


04: beq
05: bne
06: blez
07: bgtz
08: addi
09: addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

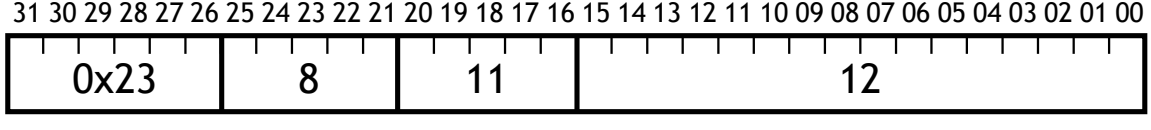
```
addi $t0, $t1, 100     # $t0 = $t1+100
```




```
beq $a0, $a1, -44     # if $a0 == $a1 GOTO
```




```
lw  $t3, 12($t0) # $t3 = Memory[$t0+12]
```



Control Signals

Func	100000	100010	N/A	N/A	N/A	N/A
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst						
ALUSrc						
MemToReg						
RegWr						
MemWr						
Branch						
Jump						
ALUCntrl	Add	Sub	Add	Add	Sub	X