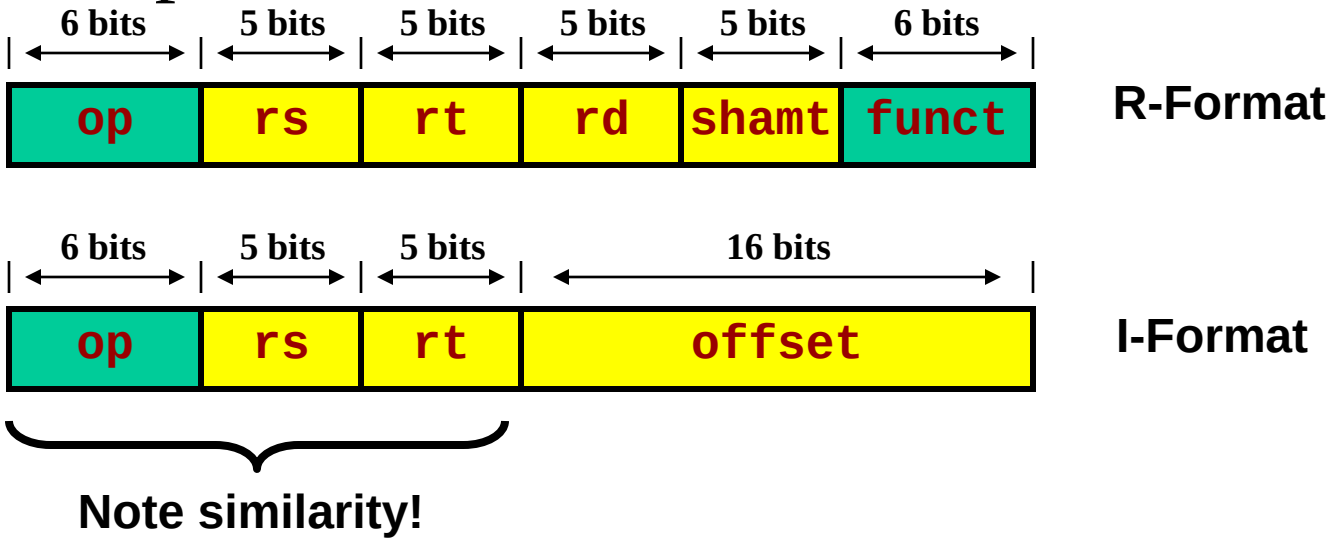
Attention

- Midterm Exam is scheduled on Oct. 23 Monday in class (75 minutes)

- SDSU is ranked No. 140 in this year's list of national universities by *US News & World Report's*. It was ranked No. 183 in 2011.

I-Format vs. R-Format Instructions

- Compare with R-Format

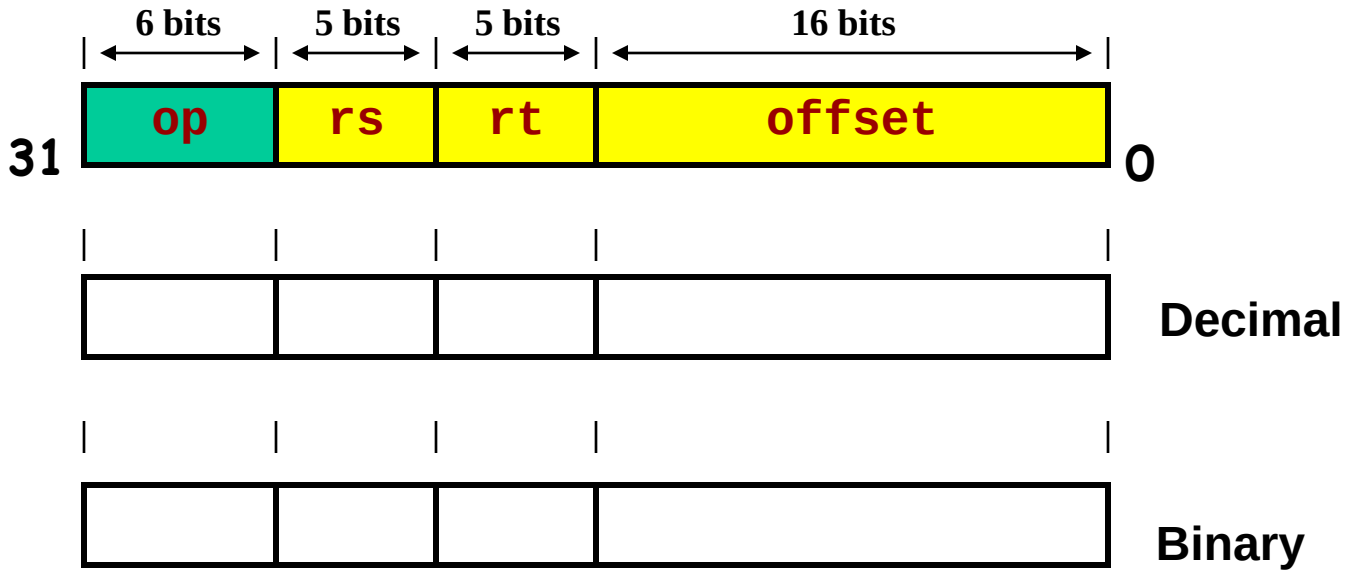


What's the difference?

I-Format Example

- Machine language for

  lw $9, 1200($8)



MIPS Conditional Branch Instructions

- Conditional branches allow decision making

```
beq R1, R2, LABEL    if R1==R2 goto LABEL
bne R3, R4, LABEL    if R3!=R4 goto LABEL
```

- Example

C Code

```
        if (i==j) goto L1;
        f = g + h;
L1:     f = f - i;
```

Assembly

```
        beq $s3, $s4, L1
        add $s0, $s1, $s2
L1:     sub $s0, $s0, $s3
```

MIPS Instruction Encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Example: Compiling C `if-then-else`

- Example

C Code

```
if (i==j) f = g + h;
else f = g - h;
```

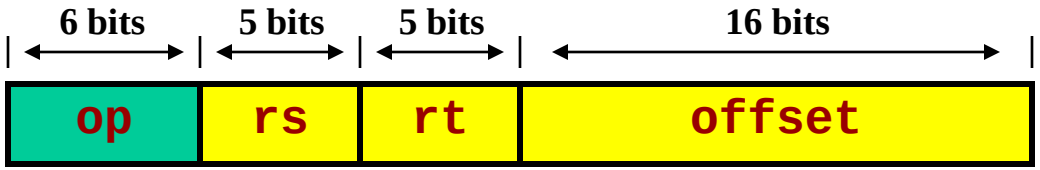
Assembly

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit;          # new: unconditional jump
Else:   sub $s0, $s1, $s2
Exit:
```

- New Instruction: Unconditional jump

```
j LABEL   # goto Label
```

Binary Representation - Branch



- Branch instructions use I-Format
- `offset` is added to PC when branch is taken

```
beq r0, r1, offset
```

has the effect:

Conversion to word offset

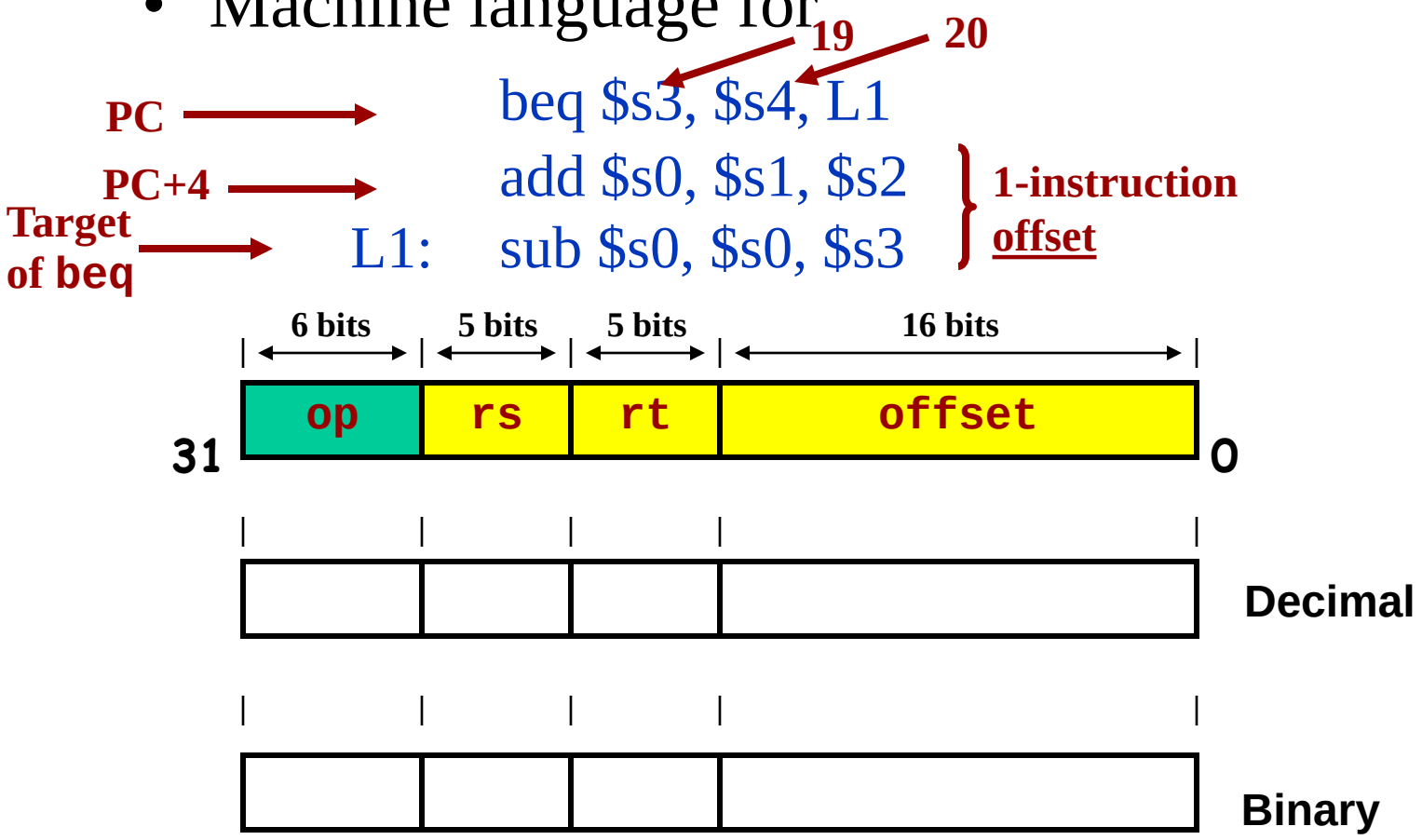
```
if (r0==r1) pc = pc + 4 + (offset << 2)
else pc = pc + 4;
```

- Offset is specified in instruction words (why?)
- What is the range of the branch target addresses?

$(PC+4)-2^{17} <= \text{target address} <= (PC+4)+ 2^{17}$

Branch Example

- Machine language for



Comparisons - What about <, <=, >, >=?

- `bne`, `beq` provide equality comparison
- `slt` provides magnitude comparison

```
slt $t0,$s3,$s4      # if $s3<$s4 $t0=1;
                     # else $t0=0;
```

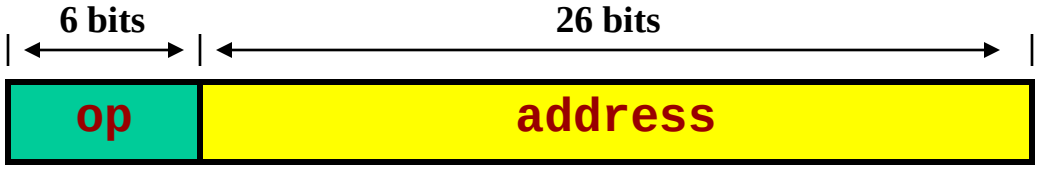
slt: Set on Less Than
blt: Branch Less Than

- Why not include a `blt` instruction in hardware?
  - Supporting in hardware would lower performance
  - Assembler provides this function if desired

- Combine with bne or beq to branch:

```
slt $t0,$s3,$s4          # if (a<b)
bne $t0,$zero, Less      #    goto Less;
```

Binary Representation - Jump



- Jump Instruction uses J-Format (op=2)
- What happens during execution?

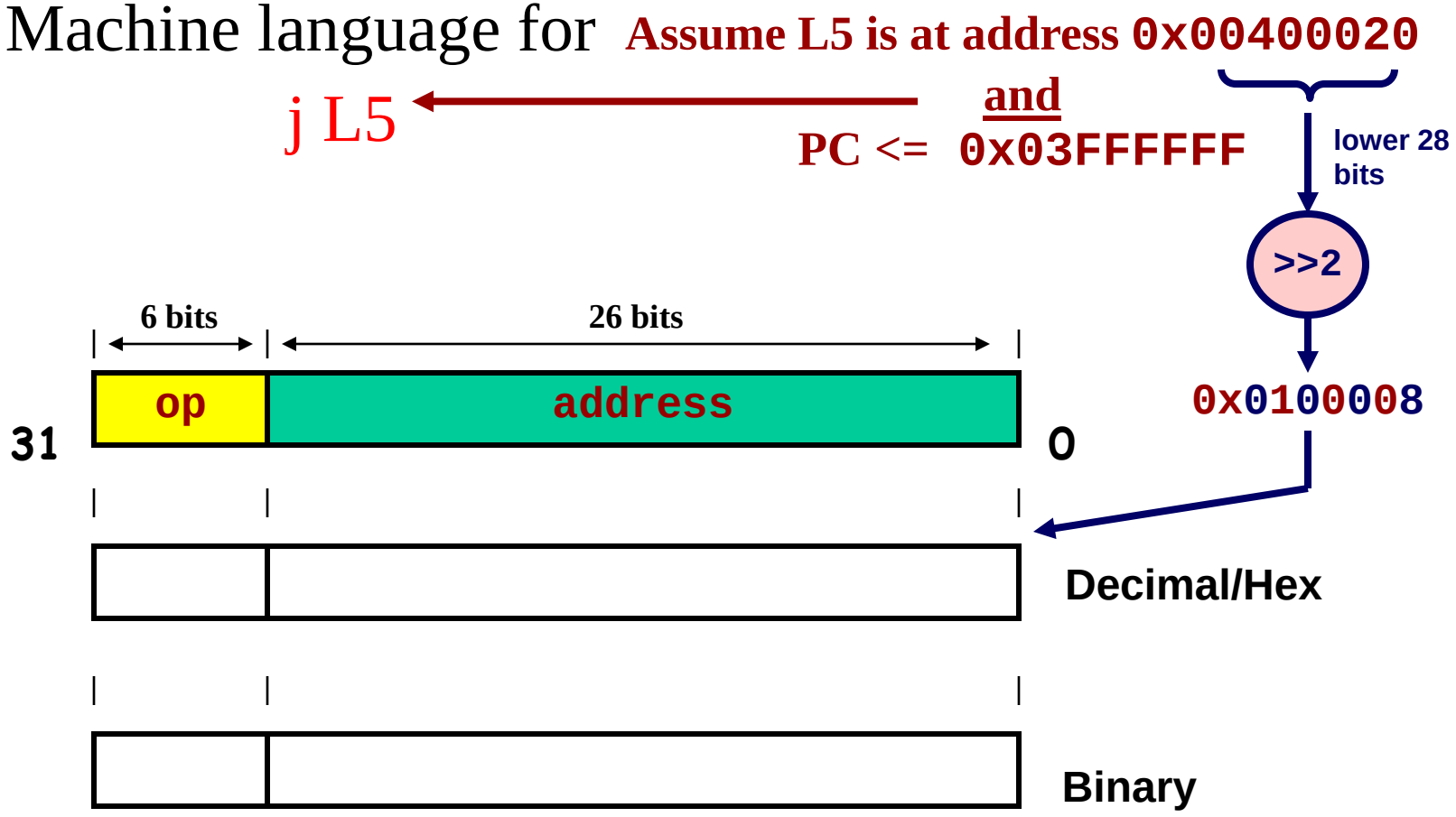
PC = PC[31:28] : (IR[25:0] << 2)

Concatenate upper 4 bits of PC to form complete 32-bit address

Conversion to word offset

Jump Example

- Machine language for



Constants / Immediate Instructions

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- MIPS Immediate Instructions (I-Format):

  addi $29, $29, 4
  slti $8, $18, 10
  } **Arithmetic instructions sign-extend immed.**

  andi $29, $29, 6
  ori $29, $29, 4
  } **Logical instructions don't sign extend immed.**

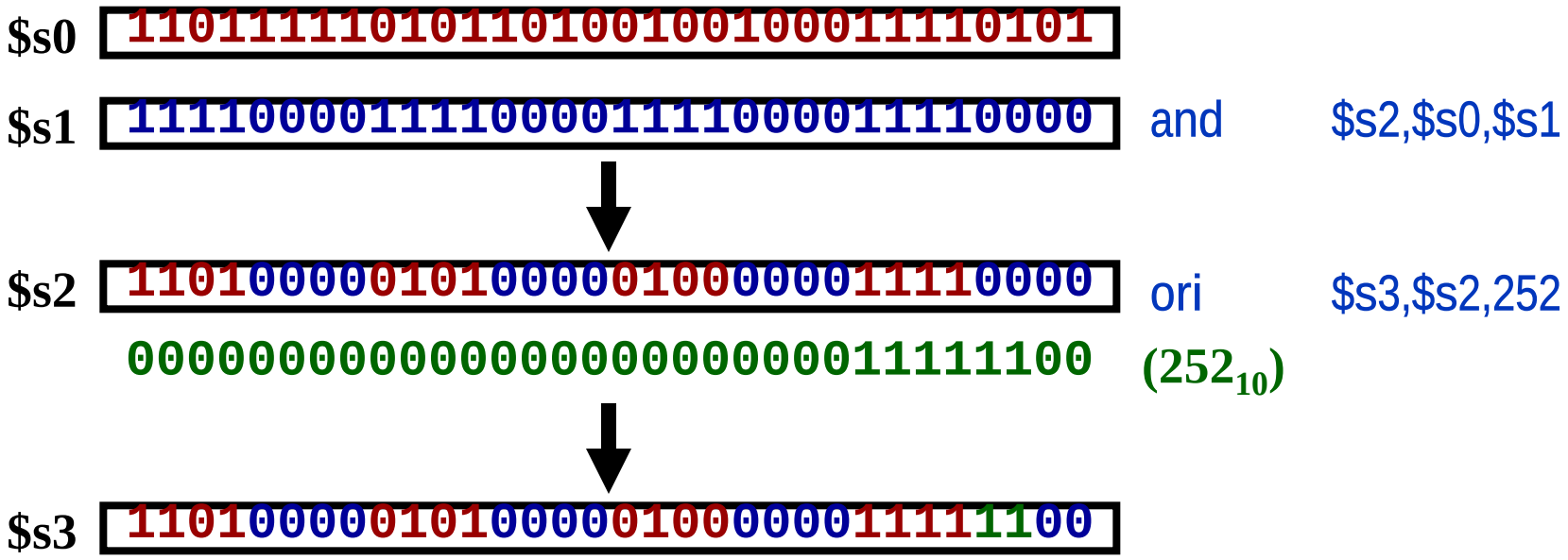
- Allows up to 16-bit constants
- How do you load just a constant into a register?

Why are Immediates only 16 bits?

- Because 16 bits fits neatly in a 32-bit instruction
- Because most constants are small (i.e. < 16 bits)
- Design Principle 4: Make the Common Case Fast

MIPS Logical Instructions

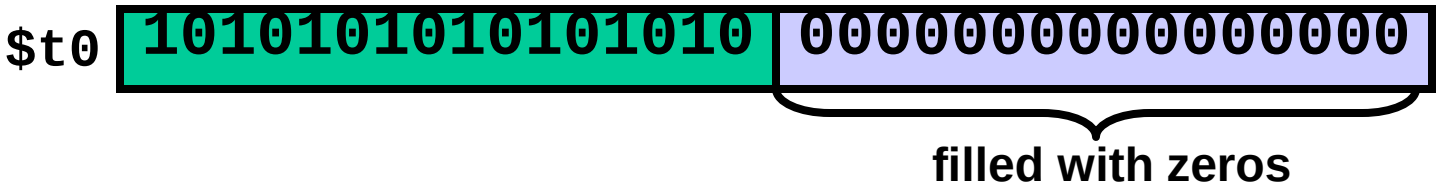
- and, andi - bitwise AND
- or, ori - bitwise OR
- Example



32-Bit Immediates and Address

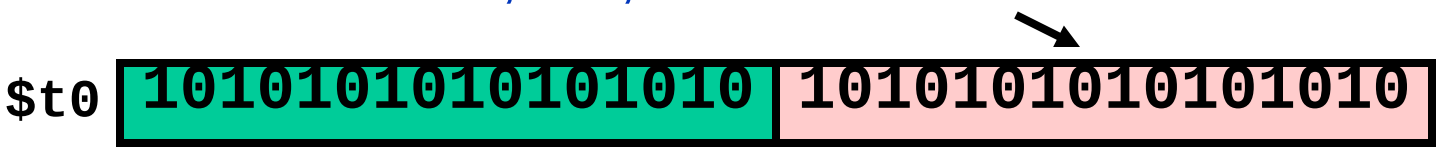
- Immediate operations provide for 16-bit constants.
- What about when we need larger constants?
- Use "load upper immediate - lui" (I-Format) to set the upper 16 bits of a constant in a register.

  lui $t0, 1010101010101010



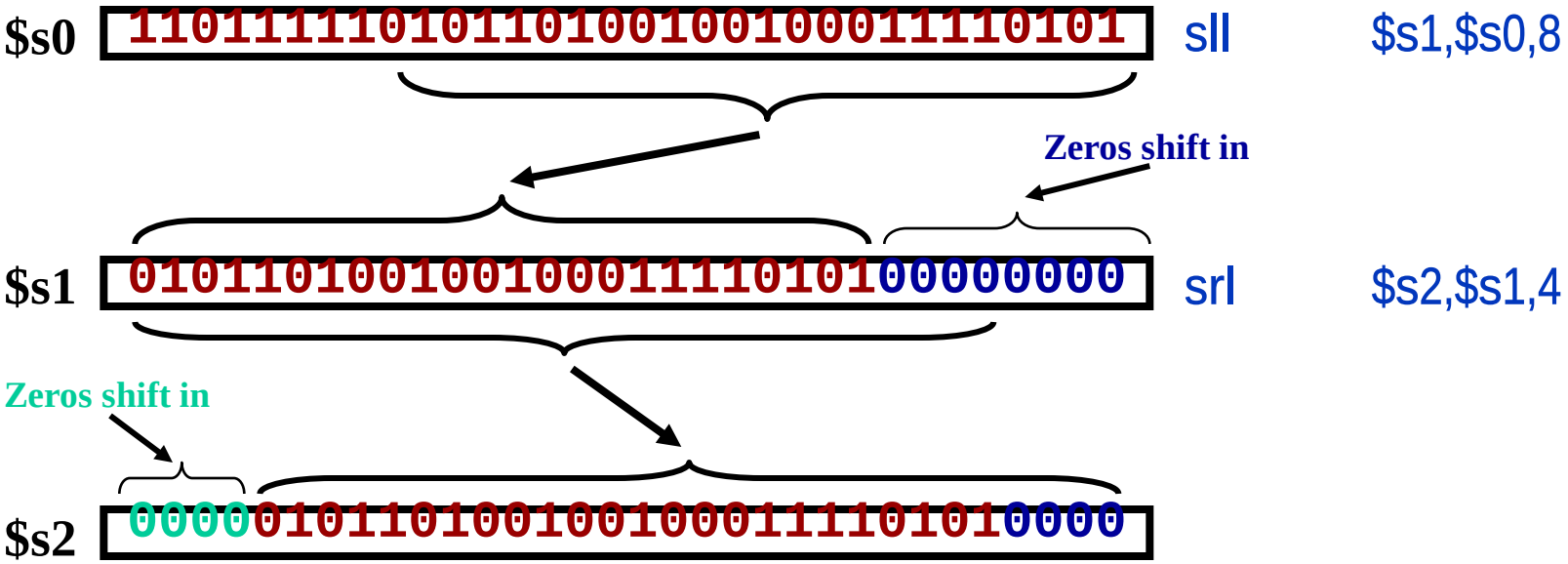
- Then use `ori` to fill in lower 16 bits:

  ori $t0, $t0, 1010101010101010

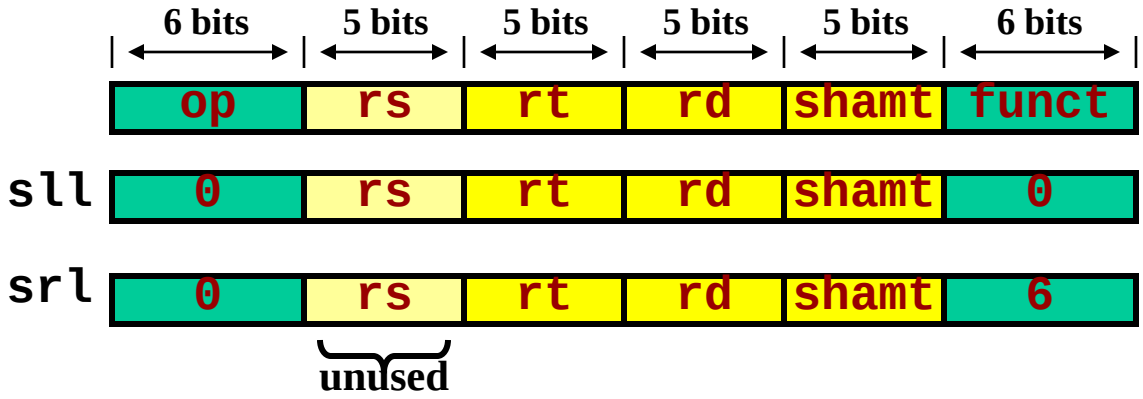


MIPS Shift Instructions

- MIPS Logical Shift Instructions
  - Shift left: sll (shift-left logical) instruction
  - Right shift: srl (shift-right logical) instruction



Shift Instruction Encodings



- Applications
  - Multiplication / Division by power of 2
  - Example: array access

Example: Array Access

- Access the *i*-th element of an array **A** (each element is 32-bit long)

```
                    # $t0 = address of start of A
                    # $t1 = i
sll $t1,$t1,2       # $t1 = 4*i
add $t2,$t0,$t1     # add offset to the address of A[0]
                    # now $t2 = address of A[i]
lw $t3,0($t2)       # $t3 = whatever is in A[i]
```

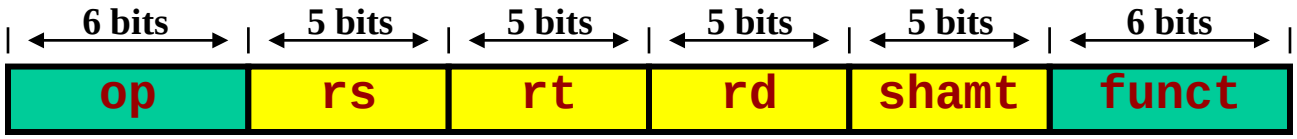
How to Decode?

- What is the assembly language statement corresponding to this machine instruction?

  0x00af8020

- Convert to binary

0000 0000 1010 1111 1000 0000 0010 0000

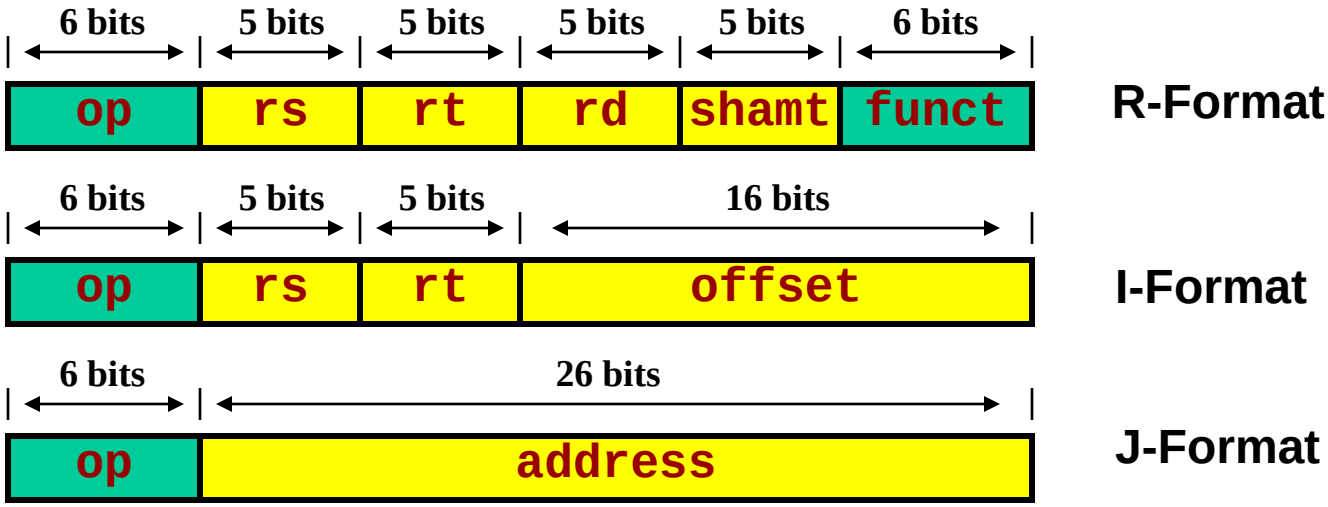


R-Format

- Decode
  - op: 000000
  - rs: 00101
  - rt: 01111
  - rd: 10000
  - shamt: 00000
  - funct: 100000
- Solution: add $s0, $a1, $t7

MIPS Instruction Set - Summary

- simple instructions all 32 bits wide
- very structured
- only three instruction formats



CS572 Micro Architecture

Single-Cycle Processor Implementation

These slides are adapted from notes by Dr. Dave Patterson (UCB)

Processor Design

- *Execution time* is a primary measure of program performance

- CPU time = IC x CPI x clock cycle time

  - IC depends on the compiler and the ISA;
  - CPI and Clock cycle time depend on processor design.

MIPS Instruction Subset

- Representative instructions from three classes included for processor design:
  - Memory reference:
    - lw, sw
  - Arithmetic-logical:
    - add, sub, and, or, slt
  - Branch:
    - beq, j

Two General Rules

1. Make the common case fast
2. Simplicity favors regularity

According to the first rule, we look for commonality in implementing:

- the three classes of instructions
- instructions within each class

What is Common to All Instructions

- Fetch and decode: Every instruction must be fetched from memory to the processor and decoded to carry out its meaning
- Register use: Every instruction (in the three classes) reads one or two registers
- ALU use: Every instruction uses ALU to do computation Why?

What is Different

- After ALU use

Instructions	Operations
Memory reference	accesses (reads or writes) a memory location
Arithmetic-logical	writes data to a register
Branch	conditionally modify next instruction address

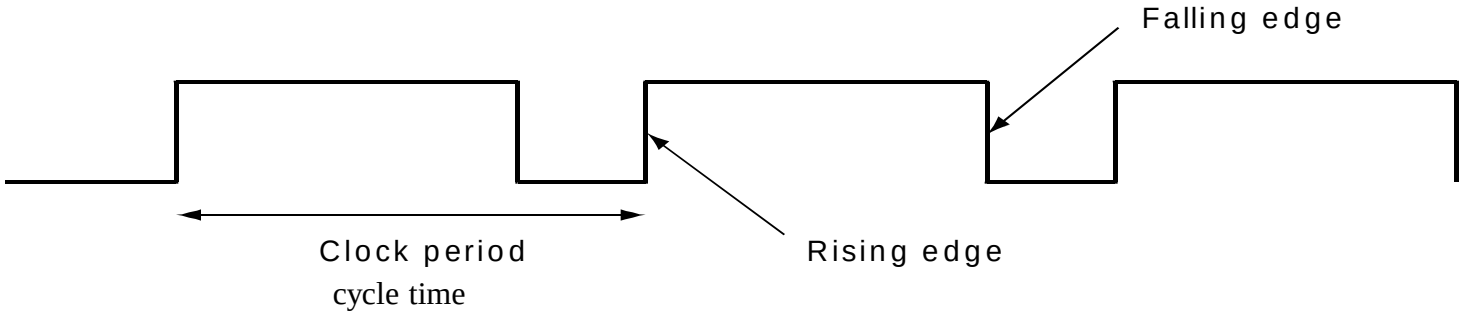
General Procedure

1. Fetch Instruction from memory (IF)
2. Decode Instruction, read register values (ID)
3. If necessary, perform an ALU operation (EX)
4. If load or store, do memory access (MEM)
5. Write results back to register file and increment PC (WB)

Review: Digital Design

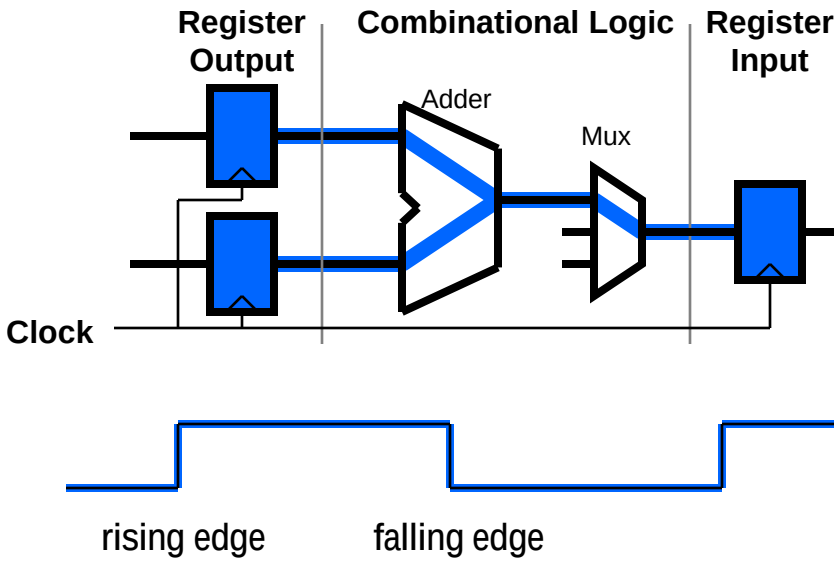
Combinational vs. Sequential Logic

- Combinational is memory-less.
  - Outputs depends only on inputs.
  - Functional abstraction
- Sequential Logic: Can remember.
  - Outputs depends inputs and internal states.
  - At least two inputs: data input and clock



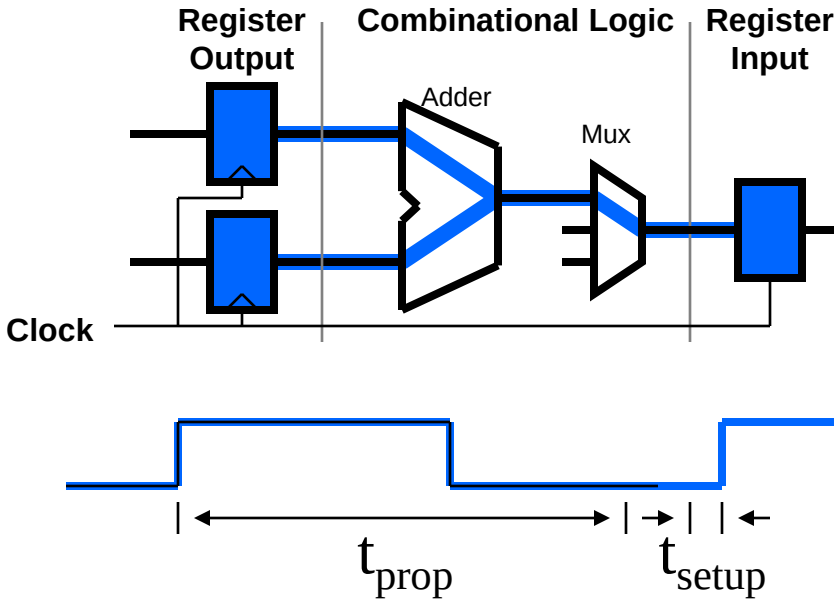
Edge-Triggered Clocking

- Controls sequential circuit operation
  - Register outputs change after first clock edge
  - Combinational logic determines “next state”
  - Storage elements store new state on next clock edge



Edge-Triggered Clocking

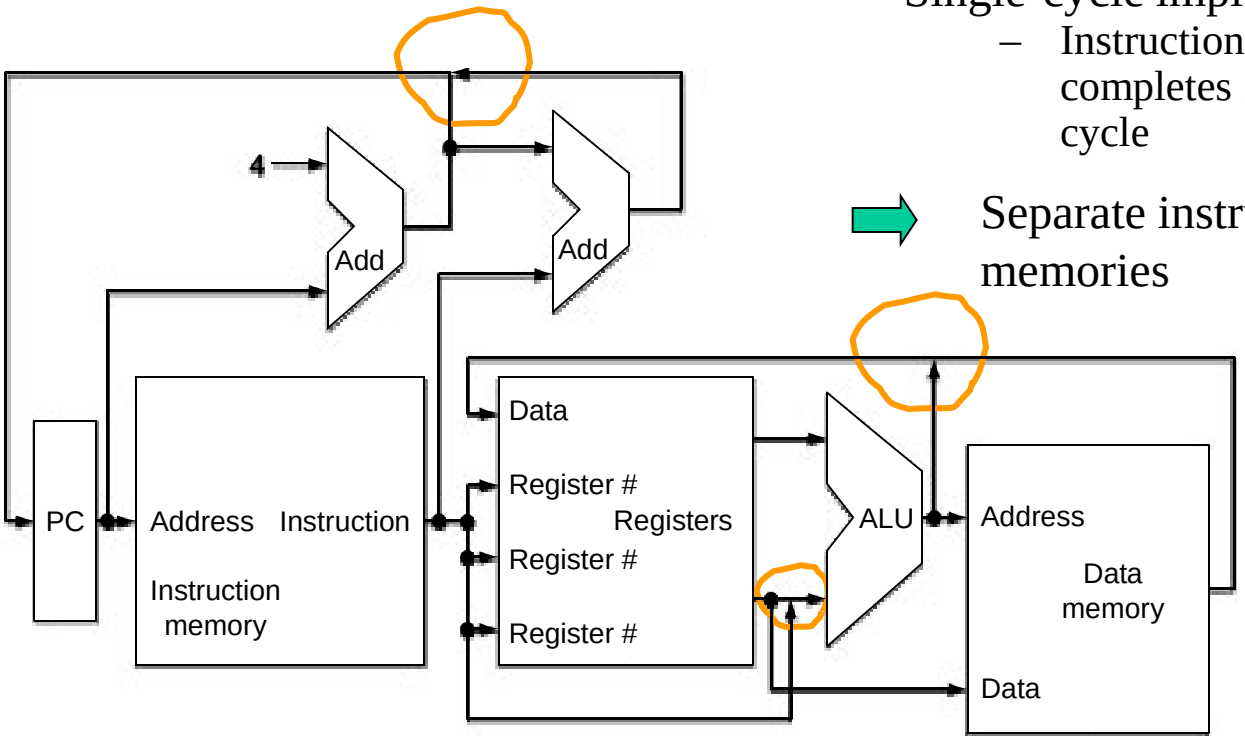
- Propagation delay - t_{prop}
  - Logic (including register outputs)
  - Interconnect
- Register setup time - t_{setup}



$$t_{clock} > t_{prop} + t_{setup}$$

$$t_{clock} = t_{prop} + t_{setup} + t_{slack}$$

Datapath: Abstract View



Single-cycle implementation
- Instruction execution completes in one clock cycle

Separate instruction and data memories

What is missing?

* Mux'es in circled places

* Instruction decoding and control signals