# CSE 141L: Introduction to Computer Architecture Lab
# Microprocessor Architecture & ISAs

**Pat Pannuto**, UC San Diego
ppannuto@ucsd.edu

## Logistics Update: Waitlists

- This is a big, hard project class
  - You now have the full scope of the big, hard project

*If you are considering dropping this course, please do so ASAP*
*Please do not wait until the deadline [Friday!]*

- If you are far back on the waitlist for 141, then please make room in 141L
- 141 will be offered next quarter
  - (I'm teaching it)

## Logistics Updates

- Project spec released for the quarter
  - Skim the whole document
  - Read the all the requirements
  - Read Milestone 1 in depth
  - Read the all the requirements again
  - Focus on the programs to start — what must your processor ***do***?
- Milestone 1 is due in 16 days
- Viva la Zoom
  - Full remote through Jan 31 at least
  - Remote participation will always be an option for 141L this quarter

## Logistics Advice

- Use Version Control
  - This is how your group should share across machines
    - Shouldn't matter if you use Questa/ModelSim locally, CloudLabs, etc…
  - Good feedback from folks using VSCode to edit & manage code
    - Especially as it has built-in git support
  - Please no public repositories!

## ISA Design and Processor Architecture are Interrelated

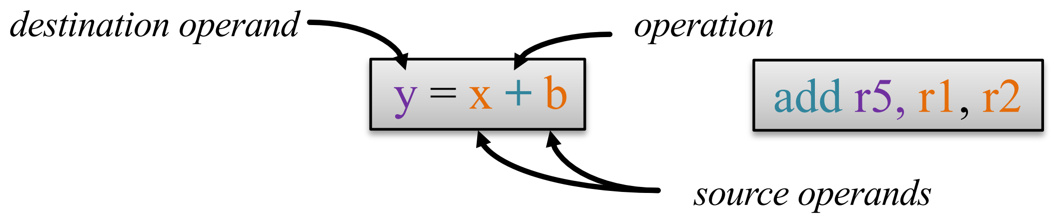
- Your ISA expresses what your processor can do
  - So your architecture has to be able to do it!

## The Instruction Set Architecture

- that part of the architecture that is visible to the programmer
  - available instructions ("opcodes")
  - number and types of registers
  - instruction formats
  - storage access, addressing modes
  - exceptional conditions
- How do each of these affect your ISA design?

# Key questions to ask when designing an ISA

- operations
  - how many?
  - which ones?
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?



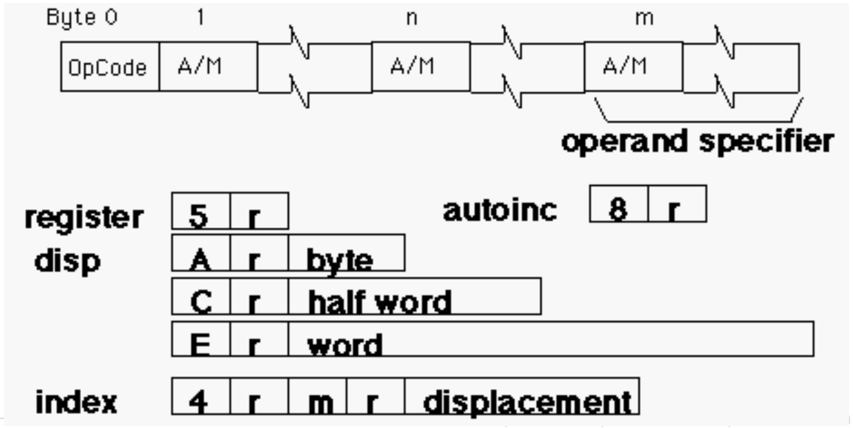
| Syntax choice | Design choice |
|---|---|
| add r5, r1, r2 | add r5, r1– r4 |
| add [r1, r2], r5 | |

*how does the computer know what* `0001 0101 0001 0010` *means?*

## Instruction Formats: What does each bit mean?

- Having many different instruction formats…
  - complicates decoding
  - uses more instruction bits (to specify the format)
  - ~~Could allow us to take full advantage of a variable-length ISA~~ not in 141L!

*VAX 11 instruction format*

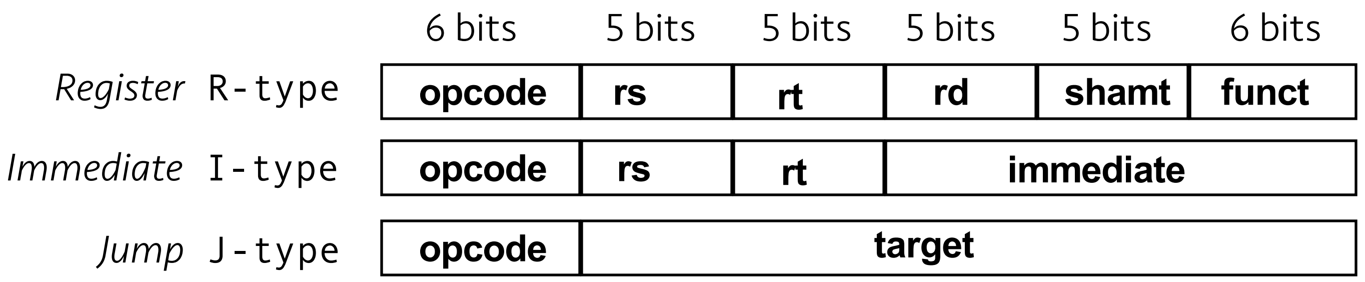


## The MIPS Instruction Format

| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| *Register* `R-type` | **opcode** | **rs** | **rt** | **rd** | **shamt** | **funct** |
| *Immediate* `I-type` | **opcode** | **rs** | **rt** | **immediate** | | |
| *Jump* `J-type` | **opcode** | **target** | | | | |

- the opcode tells the machine which format

## Example of instruction encoding:



add r5, r1, r2

```
opcode=0,    rs=1,   rt=2,   rd=5,   sa=0,   funct=32
000000       00001   00010   00101   00000   100000

00000000001000100010100000100000
0x00222420
```

## Accessing the Operands

*aka, what's allowed to go here*

add r5, r1, r2

- operands are generally in one of two places:
  - registers (32 options)
  - memory ($2^{32}$ locations)
- registers are
  - easy to specify
  - close to the processor (fast access)
- the idea that we want to use registers whenever possible led to *load-store architectures*.
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

| | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|---|
| *Register* | `R-type` | **opcode** | **rs** | **rt** | **rd** | **shamt** | **funct** |
| *Immediate* | `I-type` | **opcode** | **rs** | **rt** | **immediate** | | |
| *Jump* | `J-type` | **opcode** | **target** | | | | |

## Poll Q: Accessing the Operands

There are typically two locations for operands: registers (internal storage - $t0, $a0) and memory. In each column we have which (reg or mem) is better.

*Which row is correct?*

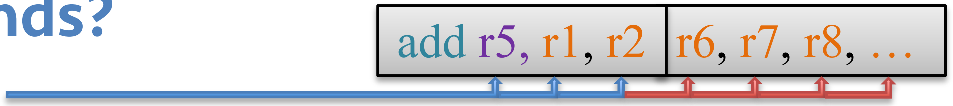
| | Faster access | Fewer bits to specify | More locations |
|---|---|---|---|
| A | Mem | Mem | Reg |
| B | Mem | Reg | Mem |
| C | Reg | Mem | Reg |
| D | Reg | Reg | Mem |
| E | None of the above | | |

## Q: How does all of this align with the project restrictions?

- [After class], re-read the restrictions with these slides in mind
- **Design Question you must answer:**
  - How will your ISA encode operations and operands?
  - And how will that impact how your machine operates?

## How Many Operands?

*aka how many of these?*



- Most instructions have three operands (e.g., z = x + y).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicity.

## Historically, many classes of ISAs have been explored, and trade off compactness, performance, and complexity

| *Style* | *# Operands* | *Example* | *Operation* |
|---|---|---|---|
| Stack | 0 | `add` | $tos_{(N-1)} \leftarrow tos_{(N)} + tos_{(N-1)}$ |
| Accumulator | 1 | `add A` | $acc \leftarrow acc + mem[A]$ |
| General Purpose Register | 3 | `add A B Rc` | $mem[A] \leftarrow mem[B] + Rc$ |
| | 2 | `add A Rc` | $mem[A] \leftarrow mem[A] + Rc$ |
| Load/Store: | 3 | `add Ra Rb Rc` | $Ra \leftarrow Rb + Rc$ |
| | | `load Ra Rb` | $Ra \leftarrow mem[Rb]$ |
| | | `store Ra A` | $mem[A] \leftarrow Ra$ |

## Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

| Stack | Accumulator | GP Register (register-memory) | GP Register (load-store) |
|---|---|---|---|

## Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

| Stack | Accumulator | GP Register (register-memory) | GP Register (load-store) |
|---|---|---|---|
| **Push A** | | | |
| **Push B** | | | |
| **Add** | | | |
| **Pop C** | | | |

## Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

| Stack | Accumulator | GP Register (register-memory) | GP Register (load-store) |
|---|---|---|---|
| **Push A** | **Load A** | | |
| **Push B** | **Add B** | | |
| **Add** | **Store C** | | |
| **Pop C** | | | |

## Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

| Stack | Accumulator | GP Register (register-memory) | GP Register (load-store) |
|---|---|---|---|
| **Push A** | **Load A** | **ADD C, A, B** | |
| **Push B** | **Add B** | | |
| **Add** | **Store C** | | |
| **Pop C** | | | |

# Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

| Stack | Accumulator | GP Register (register-memory) | GP Register (load-store) |
|---|---|---|---|
| **Push A** | **Load A** | **ADD C, A, B** | **Load R1,A** |
| **Push B** | **Add B** | | **Load R2,B** |
| **Add** | **Store C** | | **Add R3,R1,R2** |
| **Pop C** | | | **Store C,R3** |

## Exercise: Working through alternative ISAs

### [if time]

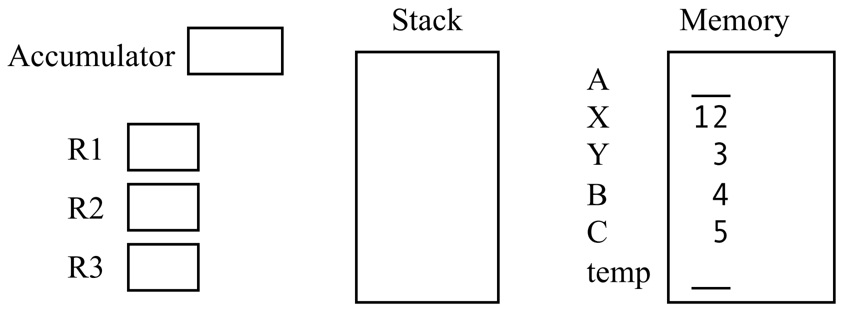
$A = X*Y - B*C$

Stack Architecture

Accumulator

GPR

GPR (Load-store)



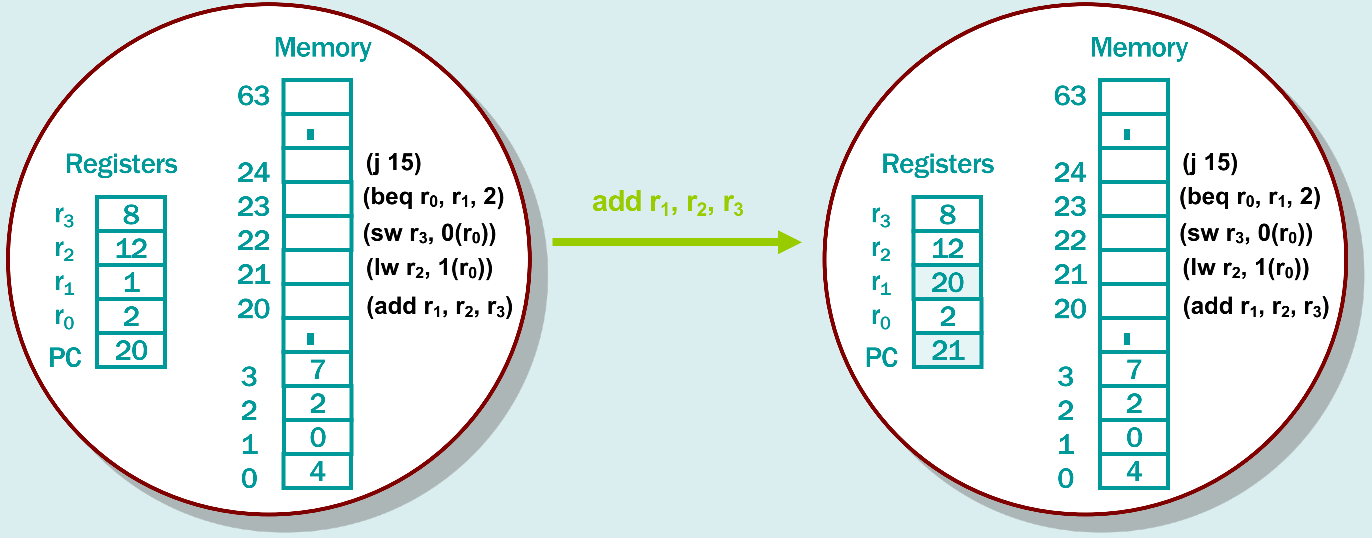
## Example: load-store (aka register-register) ISA

- load words from memory to reg file
- operate in reg file
- store results into memory from reg file

# Instruction Set Architecture

**Assumptions**

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B

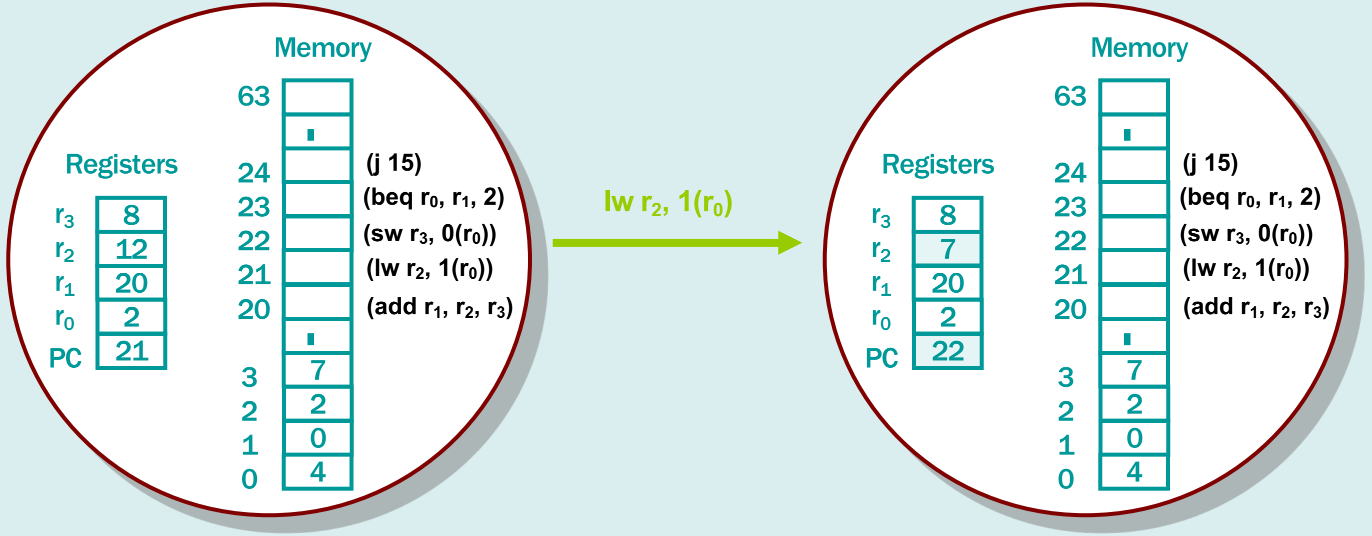


**Before Register and Memory** **After Register and Memory**

# Instruction Set Architecture

**Assumptions**

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B

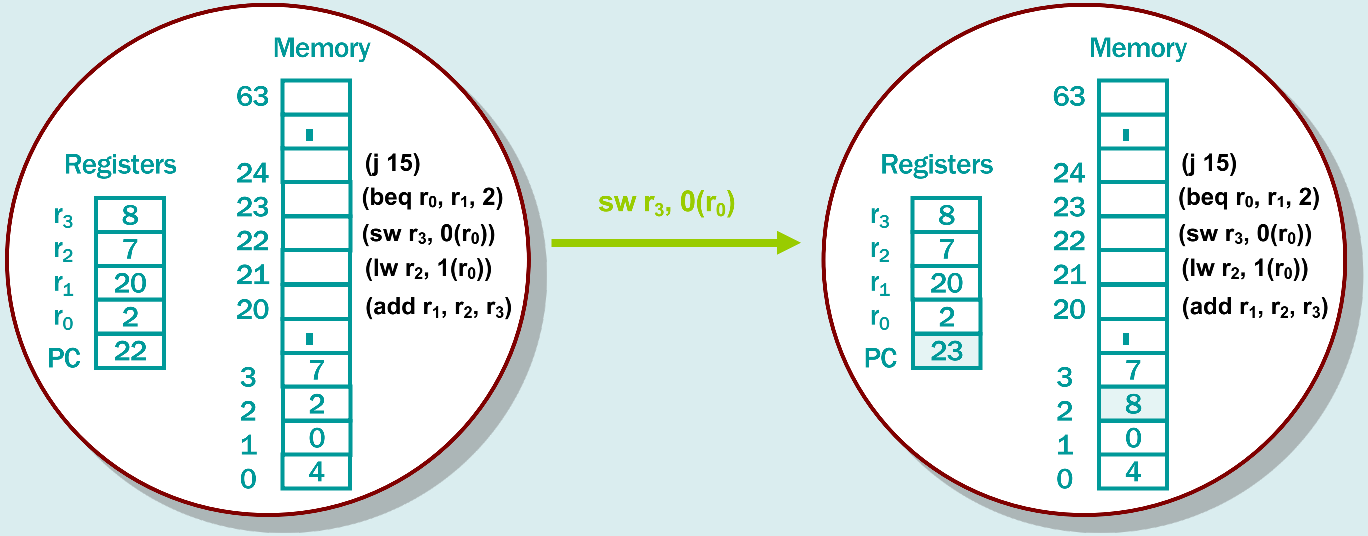


**Before Register and Memory** **After Register and Memory**

# Instruction Set Architecture

**Assumptions**

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B



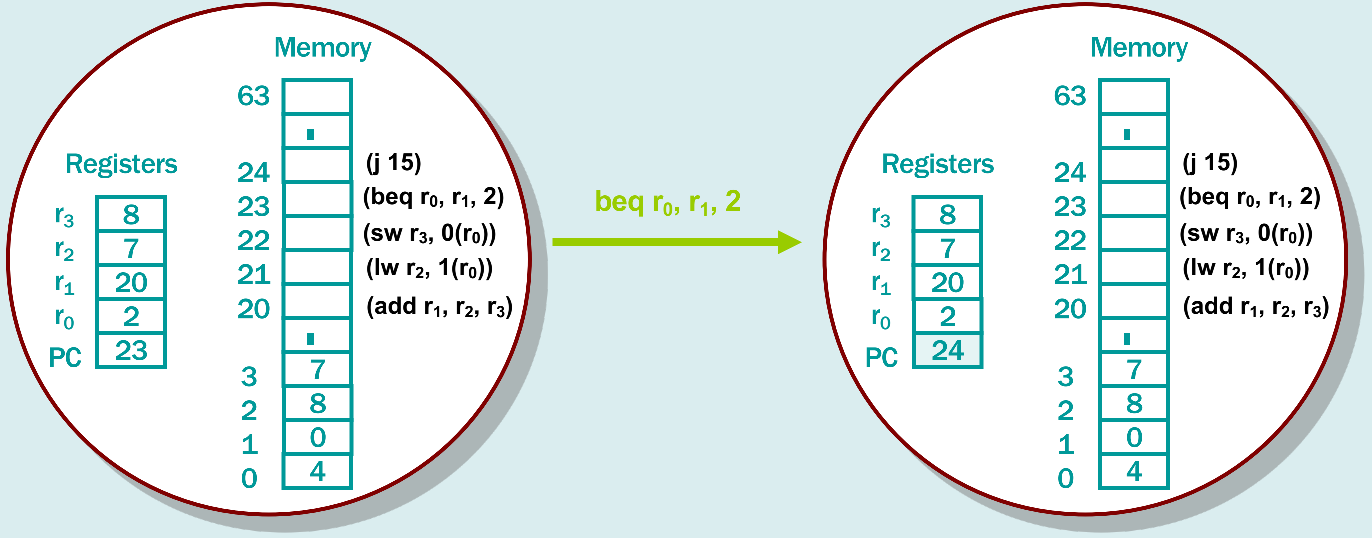
**Before Register and Memory**

**After Register and Memory**

# Instruction Set Architecture

**Assumptions**

- **8 bit ISA**
- **# of registers = 4 + PC (Program Counter)**
- **Memory size = 64B**

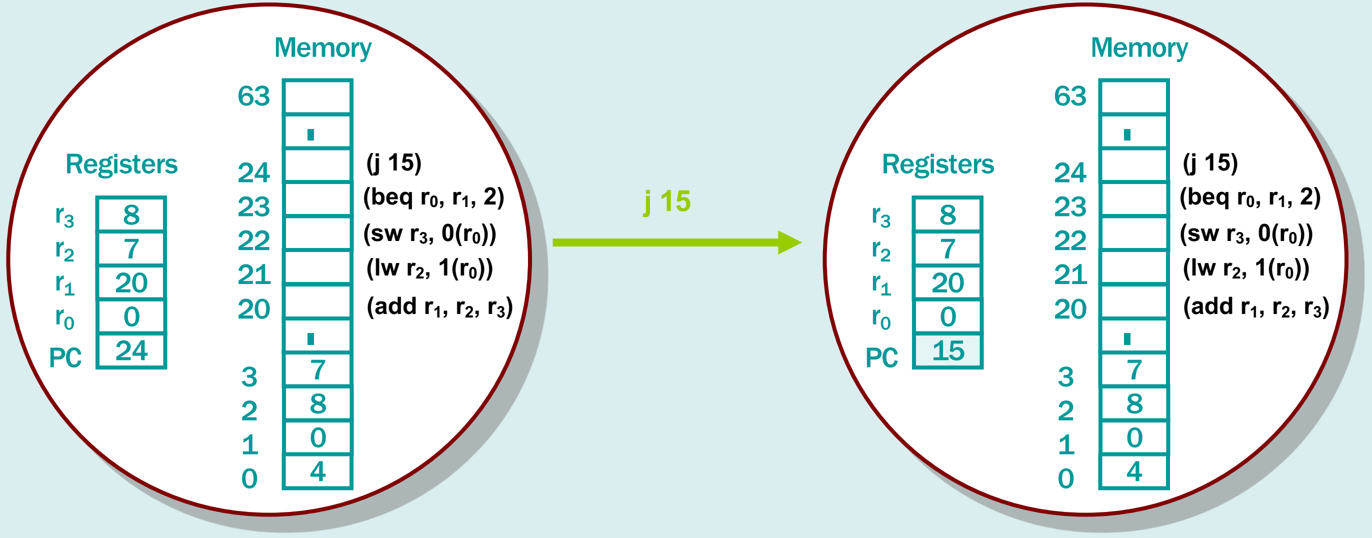


**Before Register and Memory** **After Register and Memory**

# Instruction Set Architecture

**Assumptions**

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B



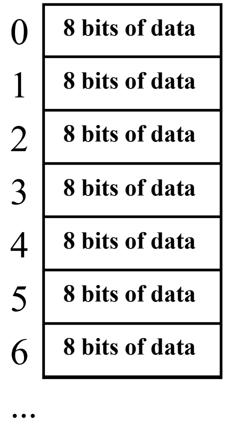
**Before Register and Memory** **After Register and Memory**

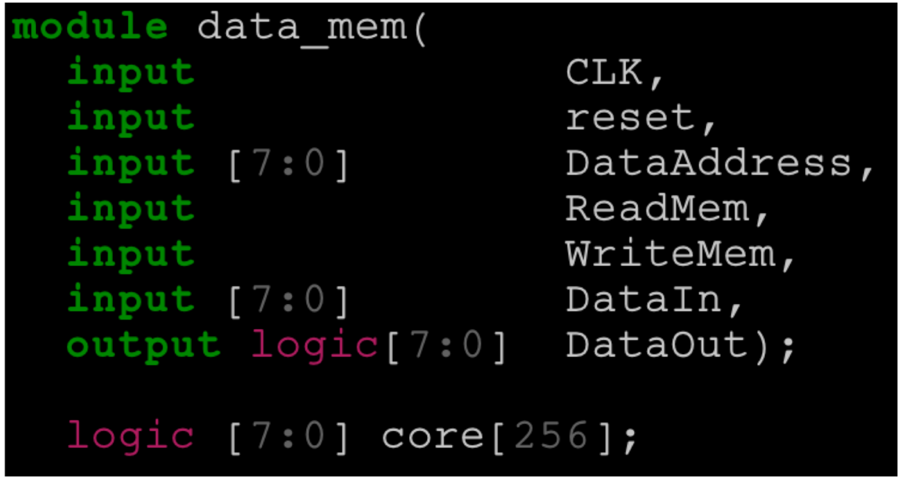
## Addressing Modes
## aka: *how do we specify the operand we want?*

| | |
|---|---|
| • Register direct | R3 |
| • Immediate (literal) | #25 |
| • Direct (absolute) | M[10000] |
| | |
| • Register indirect | M[R3] |
| • Base+Displacement | M[R3 + 10000] |
| • Base+Index | M[R3 + R4] |
| • Scaled Index | M[R3 + R4*d + 10000] |
| • Autoincrement | M[R3++] |
| • Autodecrement | M[R3 - -] |
| | |
| • Memory Indirect | M[ M[R3] ] |

## What does memory look like anyway?

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index (address) points to a byte of memory.





```
module data_mem(
   input                 CLK,
   input                 reset,
   input [7:0]           DataAddress,
   input                 ReadMem,
   input                 WriteMem,
   input [7:0]           DataIn,
   output logic[7:0]     DataOut);

   logic [7:0] core[256];
```

## Which kinds of things can a processor *do*?

- arithmetic
  - add, subtract, multiply, divide
- logical
  - and, or, shift left, shift right
- data transfer
  - load word, store word

## “Control Flow” describes how programs execute

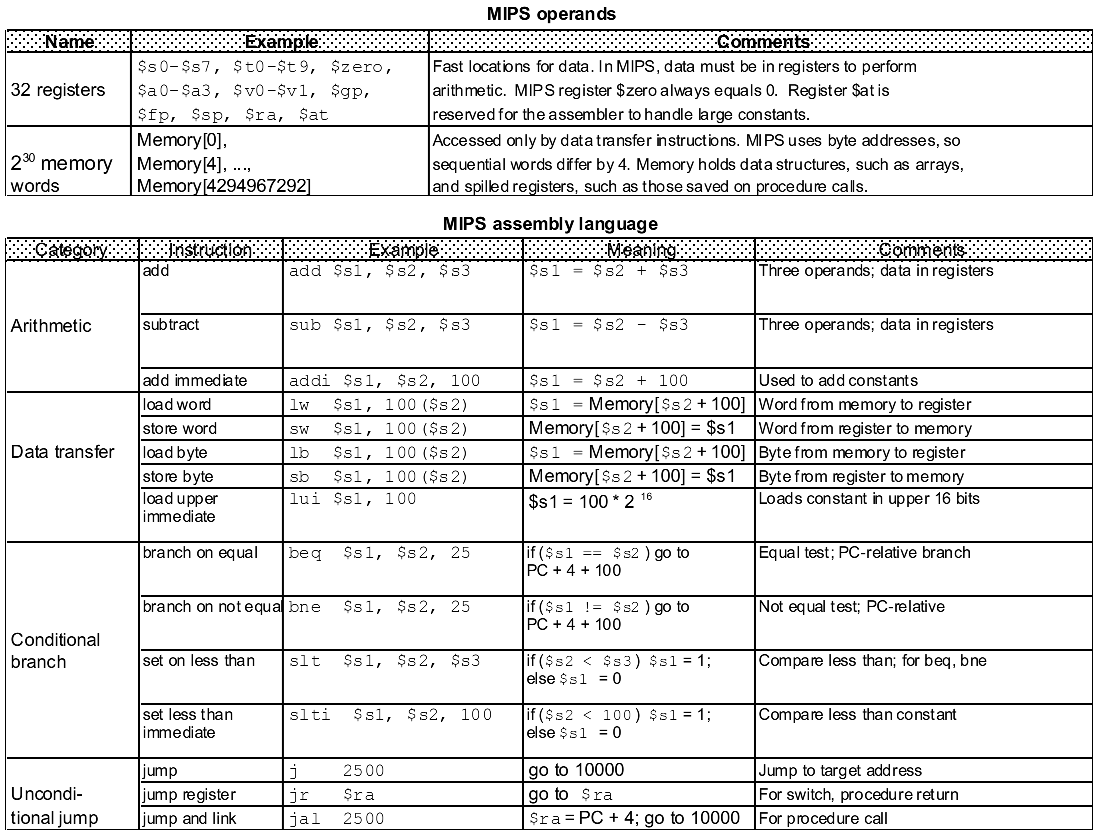
- Jumps
- Procedure call (jump subroutine)
- Conditional Branch
  - Used to implement, for example, if-then-else logic, loops, etc.

- Control flow must specify two things
  - Condition under which the jump or branch is taken
  - If take, the location to read the next instruction from (“target”)

## How do you specify the destination of a branch/jump?

- Unconditional jumps may go long distances
  - Function calls, returns, …
- Studies show that almost all conditional branches go short distances from the current program counter
  - loops, if-then-else, …
- A relative address requires (many) fewer bits than an absolute address
  - e.g., `beq $1, $2, 100  =>  if ($1 == $2):  PC = (PC+4) + 100 * 4`

# MIPS in one slide

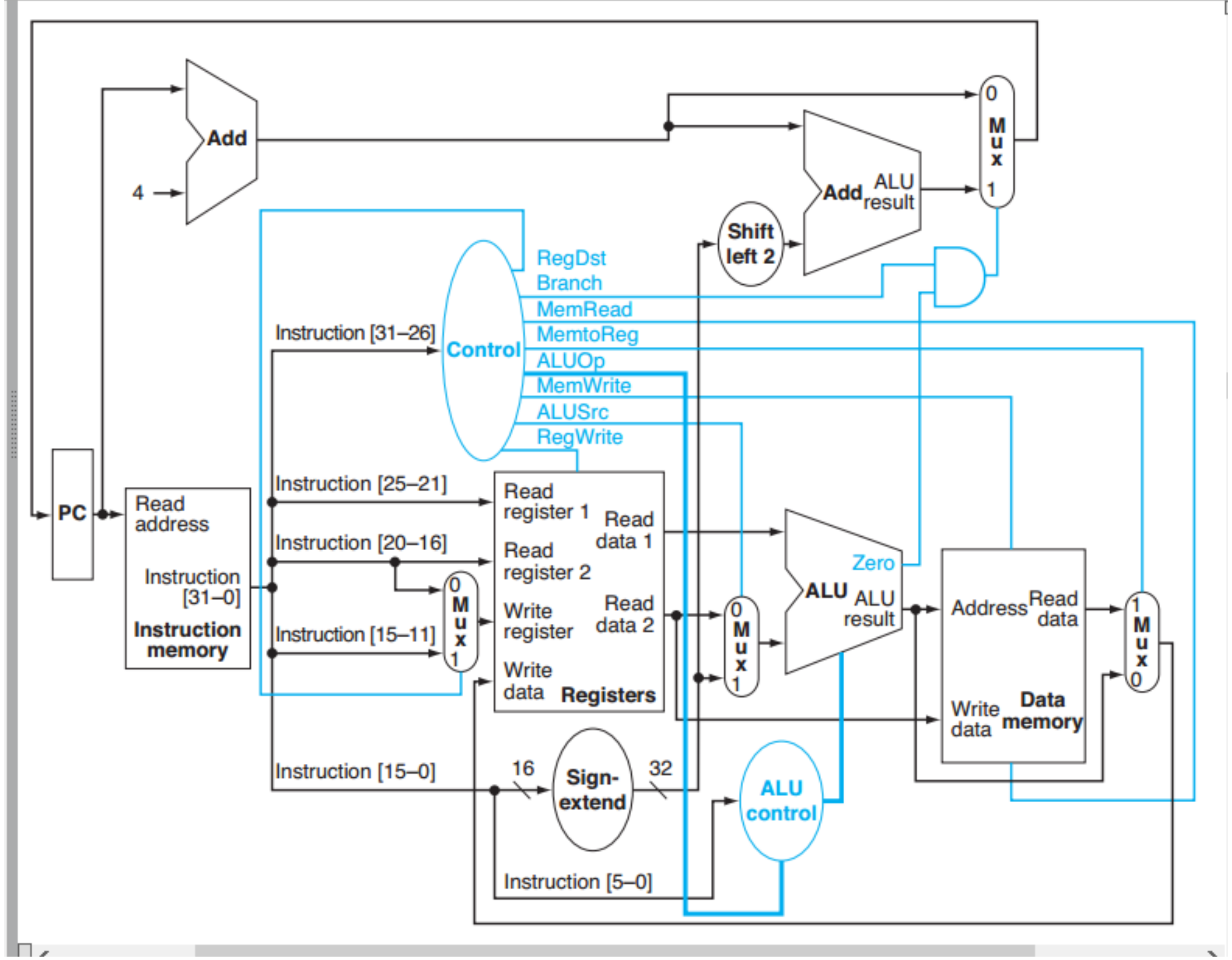


**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero,` `$a0-$a3, $v0-$v1, $gp,` `$fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], ..., Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls. |

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1, $s2, $s3` | `$s1 = $s2 + $s3` | Three operands; data in registers |
| | subtract | `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3` | Three operands; data in registers |
| | add immediate | `addi $s1, $s2, 100` | `$s1 = $s2 + 100` | Used to add constants |
| Data transfer | load word | `lw $s1, 100($s2)` | $s1 = Memory[$s2 + 100] | Word from memory to register |
| | store word | `sw $s1, 100($s2)` | Memory[$s2 + 100] = $s1 | Word from register to memory |
| | load byte | `lb $s1, 100($s2)` | $s1 = Memory[$s2 + 100] | Byte from memory to register |
| | store byte | `sb $s1, 100($s2)` | Memory[$s2 + 100] = $s1 | Byte from register to memory |
| | load upper immediate | `lui $s1, 100` | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | `beq $s1, $s2, 25` | if (`$s1 == $s2`) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equa | `bne $s1, $s2, 25` | if (`$s1 != $s2`) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | `slt $s1, $s2, $s3` | if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0 | Compare less than; for beq, bne |
| | set less than immediate | `slti $s1, $s2, 100` | if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0 | Compare less than constant |
| Uncondi-tional jump | jump | `j 2500` | go to 10000 | Jump to target address |
| | jump register | `jr $ra` | go to `$ra` | For switch, procedure return |
| | jump and link | `jal 2500` | `$ra` = PC + 4; go to 10000 | For procedure call |


Add
4
Add
ALU result
M u x
0
1
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read address
Instruction [31–0]
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

## More Information? More Machine Types?

- 141 will talk some about other machine types
  - The 141 textbook goes into more detail
- I will post a collection of slides and resources from others in Canvas
- Many additional resources online