

CSE 141: Introduction to Computer Architecture

Instruction Set Architecture (ISA)

What is Computer Architecture?

Computer Architecture =

Instruction Set Architecture


+

Machine Organization

How you talk to the machine

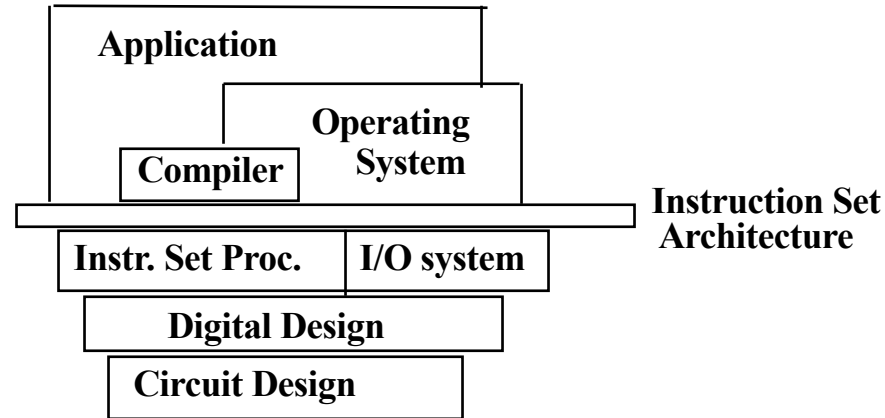


*What the machine
hardware looks like*



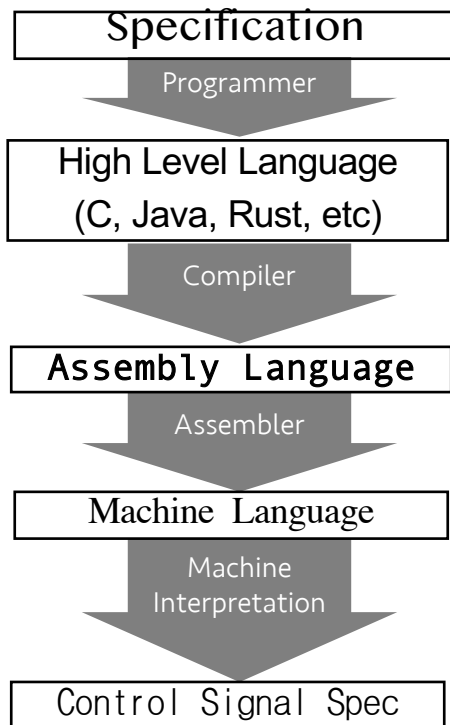
An Instruction Set Architecture is an **abstraction of a computational machine**

- An ISA is “the agreed-upon interface between all the software that runs on the machine and the hardware that executes it.”



Computers do not speak English

And they do not speak C or Java or Python or Haskell (or...) either



“Swap two array elements.”

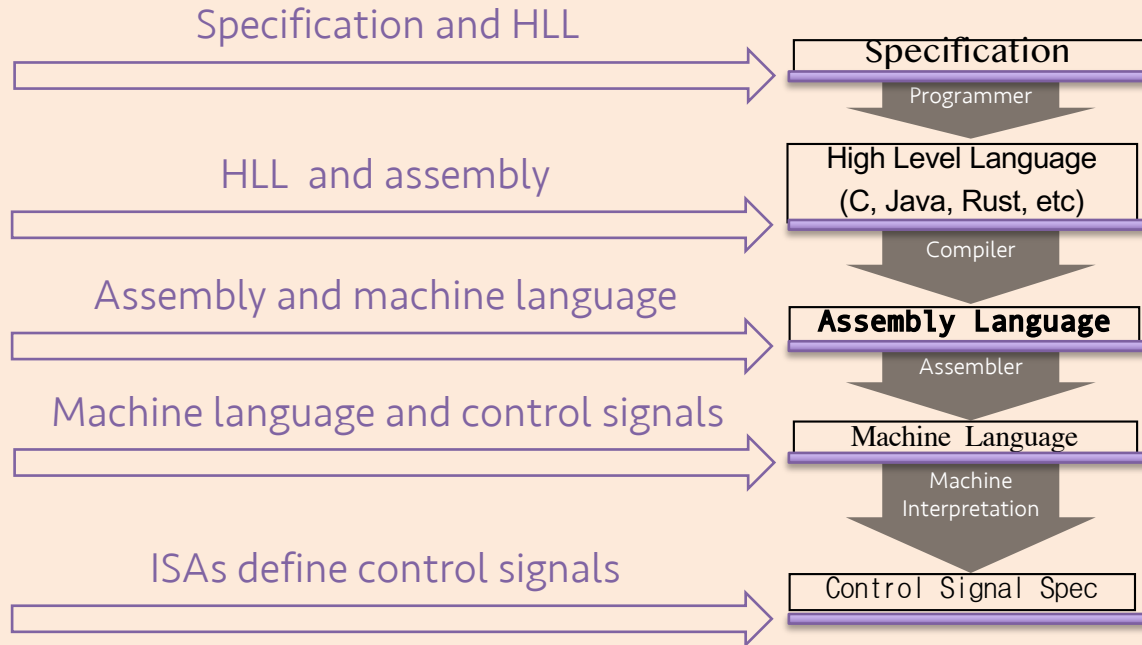
```
int temp = array[index];  
array[index] = array[index + 1];  
array[index + 1] = temp;
```

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

```
10001100011000100000000000000000  
1000110011110010000000000000100  
10101100111100100000000000000000  
1010110001100010000000000000100
```

```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

Poll Q: If you had to put the ISA somewhere in this stack, would you say it sits between...



“Swap two array elements.”

```
int temp = array[index];  
array[index] = array[index + 1];  
array[index + 1] = temp;
```

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

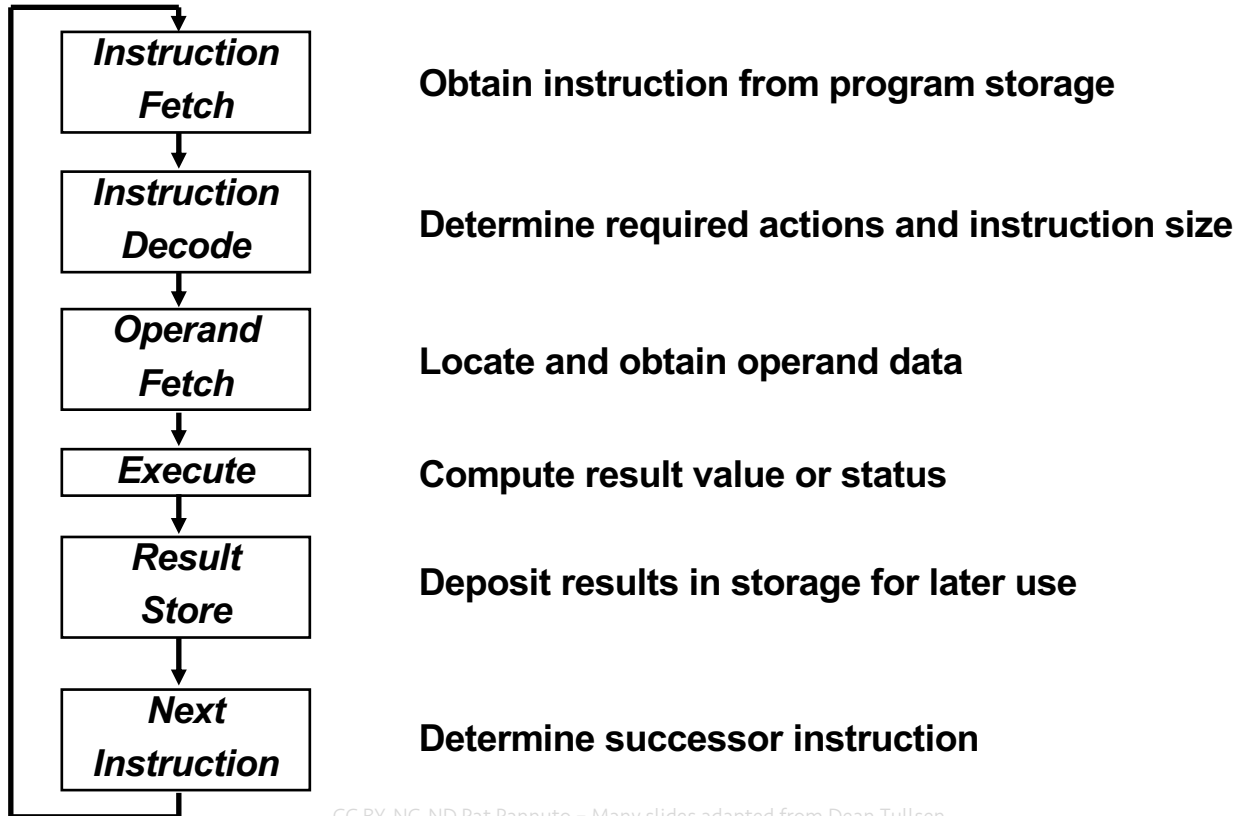
```
10001100011000100000000000000000  
10001100111100100000000000000100  
10101100111100100000000000000000  
10101100011000100000000000000100
```

```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

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

The Instruction Execution Cycle



A brief preview of some machine organization concepts:

Cycle

- The smallest unit of time in a processor



macOS Catalina
Version 10.15.6

iMac (Retina 5K, 27-inch, 2017)
Processor 4.2 GHz Quad-Core Intel Core i7
Memory 40 GB 2400 MHz DDR4
Startup Disk Macintosh HD
Graphics Radeon Pro 580 8 GB

This card displays system information for macOS Catalina Version 10.15.6 on an iMac (Retina 5K, 27-inch, 2017). It includes details on the processor (4.2 GHz Quad-Core Intel Core i7), memory (40 GB 2400 MHz DDR4), startup disk (Macintosh HD), and graphics (Radeon Pro 580 8 GB). A circular image of a mountain landscape is on the left.



macOS Catalina
Version 10.15.7

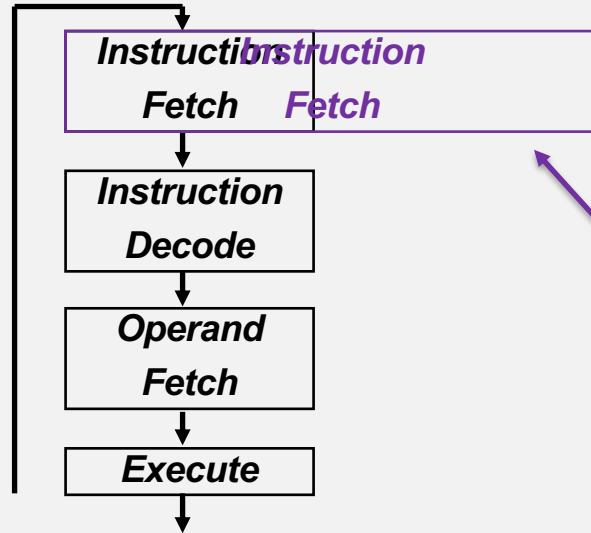
MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)
Processor 2.7 GHz Quad-Core Intel Core i7
Memory 16 GB 2133 MHz LPDDR3
Startup Disk APPLE SSD AP1024M Media
Graphics Intel Iris Plus Graphics 655 1536 MB

This card displays system information for macOS Catalina Version 10.15.7 on a MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports). It includes details on the processor (2.7 GHz Quad-Core Intel Core i7), memory (16 GB 2133 MHz LPDDR3), startup disk (APPLE SSD AP1024M Media), and graphics (Intel Iris Plus Graphics 655 1536 MB). A circular image of a mountain landscape is on the left.

A brief preview of some machine organization concepts:

Parallelism

- The ability to do more than one thing at once



Real-world example

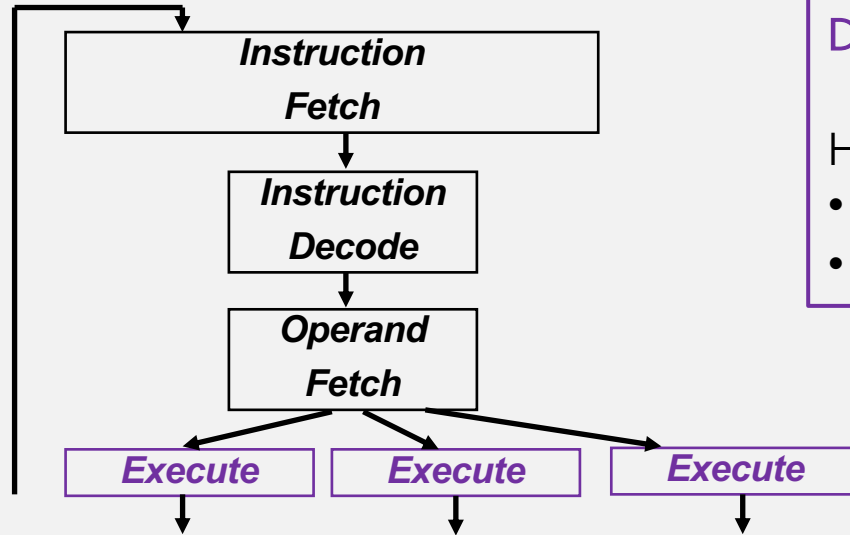
ARM's Thumb instruction set is (mostly) 16-bit instructions on a 32-bit machine

ISA design makes fetch "freely parallel"

A brief preview of some machine organization concepts:

Superscalar Processor

- Can execute more than one instruction per cycle



Duplication is easy but expensive...

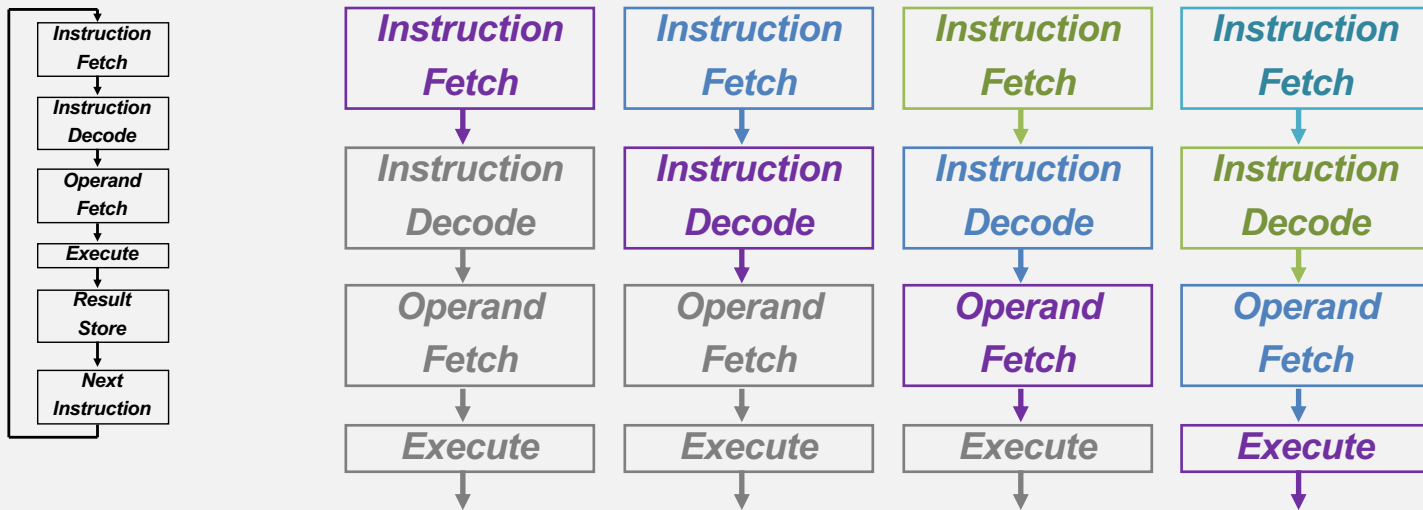
How to do parallelism well?

- Second half of this class
- CSE148

A brief preview of some machine organization concepts:

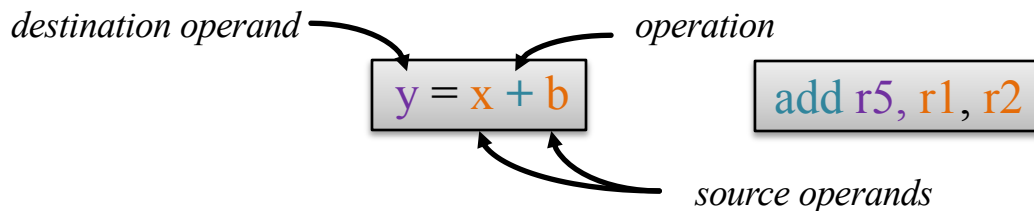
Pipelining

- Overlapping parts of a large task to increase throughput without decreasing latency
 - Key insight: The less work you do in one step, the faster each step can finish



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
<code>add r5, r1, r2</code>	<code>add r5, r1-r4</code>
<code>add [r1, r2], r5</code>	

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

Let us design MIPS together

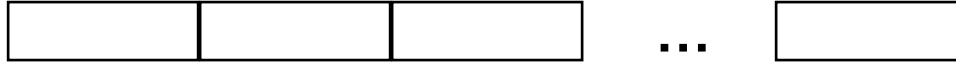
- We will look at several of the key ISA design decisions
- To succeed in 141 you need to understand the how and the why of MIPS
 - The rest of the course builds on MIPS, so need to be comfortable with it
 - But also need to understand the architectural tradeoffs of MIPS
- To succeed in 141L you need to understand the tradeoffs in ISA design

How long should an instruction be?

- Fixed



- Variable



- Hybrid



add r5, r1, r2

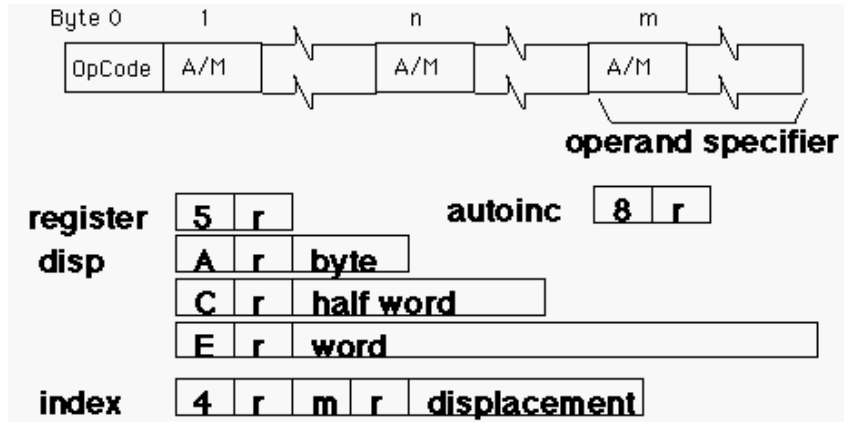
Instruction length tradeoffs

- Fixed-length instructions (MIPS)
 - easy fetch and decode
 - simplify pipelining and parallelism.
 - Variable-length instructions (Intel 80x86, VAX)
 - multi-step fetch and decode
 - much more flexible and compact instruction set.
 - Hybrid instructions (ARM)
 - Middle ground
- ⇒ All MIPS instructions are 32 bits long.
- this decision impacts every other ISA decision we make because it makes instruction bits scarce.

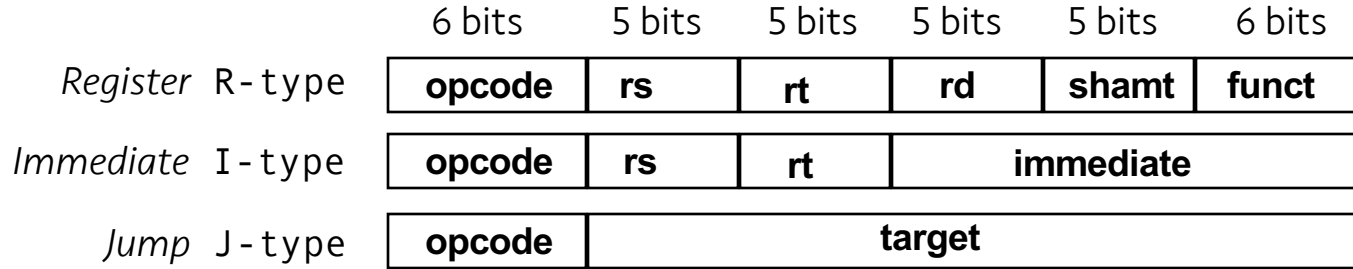
Instruction Formats: What does each bit mean?

- 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

VAX 11 instruction format

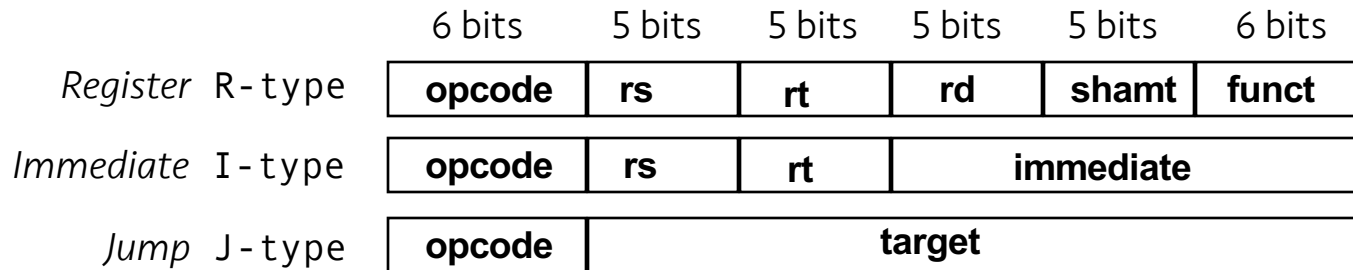


The MIPS Instruction Format



- 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

000000000001000100010100000100000
 0x00222420

Poll Q: Implications of the MIPS instruction format

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
<i>Register</i> R-type	opcode	rs	rt	rd	sa	funct
<i>Immediate</i> I-type	opcode	rs	rt	immediate		
<i>Jump</i> J-type	opcode	target				

What is the maximum number of unique operations MIPS can encode?

3

64

127

128

Accessing the Operands

aka, what's allowed to go here

add r5, r1, r2

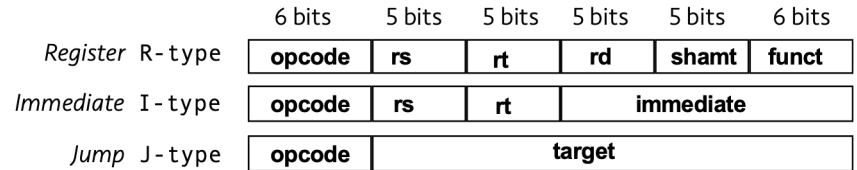
- operands are generally in one of two places:
 - registers (32 options)
 - memory (2^{32} locations)

- 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



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		

MIPS uses a load/store architecture to access operands

can do:

```
add $t0 = $s1 + $s2
```

and

```
lw $t0, 32($s3)
```

➔ forces heavy dependence on registers, which is exactly what you want in today's CPUs

can't do

```
add $t0 = $s1, 32($s3)
```

– more instructions

+ fast implementation

(e.g., easy pipelining)

What pushes MIPS towards a load/store design? (hint: fixed instruction length)

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 explicitly.

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

GP Register

(register-memory)

(load-store)

Comparing the Number of Instructions

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

Stack

Accumulator

GP Register

GP Register

(register-memory)

(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:

<u>Stack</u>	<u>Accumulator</u>	<u>GP Register</u> (register-memory)	<u>GP Register</u> (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:

<u>Stack</u>	<u>Accumulator</u>	<u>GP Register</u> (register-memory)	<u>GP Register</u> (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:

<u>Stack</u>	<u>Accumulator</u>	<u>GP Register</u> (register-memory)	<u>GP Register</u> (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]

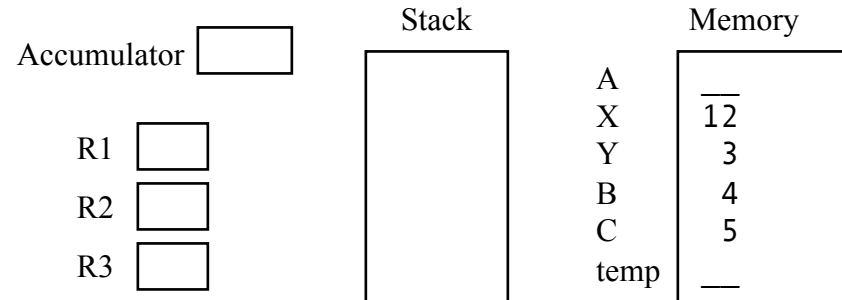
$$A = X * Y - B * C$$

Stack Architecture

Accumulator

GPR

GPR (Load-store)



Poll Q: The destination of a MIPS add operation can be...

- Only the top of the stack
- Only the accumulator register
- Any general purpose register
- Any general purpose register or anywhere in memory
- Any general purpose register or the top of the stack

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]]

MIPS addressing modes and syntax

register direct



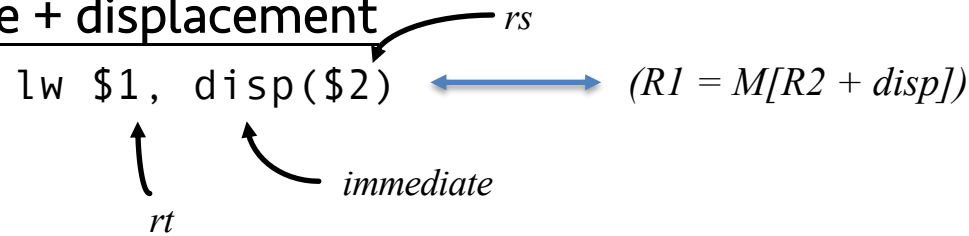
add \$1, \$2, \$3

immediate



add \$1, \$2, #35

base + displacement



register indirect

$\Leftrightarrow disp = 0$

absolute

$\Leftrightarrow (rs) = 0$

Is this sufficient?

- measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
- similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time
- and that 16 bits is enough of a displacement 99% of the time.
- (and when these are not sufficient, it typically means we need one more instruction)

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.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

Memory accesses are (often) required to be “word-aligned” because of how buses and memory work

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

- Words are aligned
i.e., what are the least 2 significant bits of a word address?

The MIPS ISA, so far

- fixed 32-bit instructions
- 3 instruction formats (R, I, J)
- 3-operand, load-store architecture
- 32 general-purpose registers
 - R0 always equals 0.
- 2 additional special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- registers are 32-bits wide (word)
- register, immediate, and base+displacement addressing modes

But what kinds of things do computers actually do?

- arithmetic
- logical
- data transfer
- conditional branch
- unconditional jump

Which kinds of instructions does (and doesn't) the MIPS ISA support?

- arithmetic
 - add, subtract, multiply, divide
 - But not:
- logical
 - and, or, shift left, shift right
 - But not:
- data transfer
 - load word, store word
 - But not:

“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”)

Jumps are unconditional control flow.

What do they look like in MIPS?

- need to be able to jump to an absolute address sometimes
- need to be able to do procedure calls and returns

<i>Jump</i> J-type	opcode	target
--------------------	--------	--------

- Jump j 10000 => PC = 10000
- Jump and Link jal 20000 => \$31 = PC + 4 and PC = 20000
 - used for procedure calls
- Jump register jr \$31 => PC = \$31
 - used for returns, but can be useful for lots of other things
 - Q: how to encode jr instruction?

Warning: Some ISAs call jumps “unconditional branches” – useful not to for MIPS

What if we want to condition the control flow? Branches.

```
do { ... ; a++; } while (a < 100);
```

- `beq` and `bne` are the only branches you need
 - `beq r1, r2, addr` => `if (r1 == r2): goto addr`
- But other operations can be combined...
 - `slt $1, $2, $3` => `if ($2 < $3) $1 = 1; else $1 = 0`
- `beq`, `bne`, `slt`, and `$zero`, can implement all fundamental conditions
 - Always, never, `!=`, `==`, `>`, `<=`, `>=`, `<`, `>(unsigned)`, `<= (unsigned)`, ...

```
if (i < j)
```

```
    w = w+1;
```

```
else
```

```
    w = 5;
```

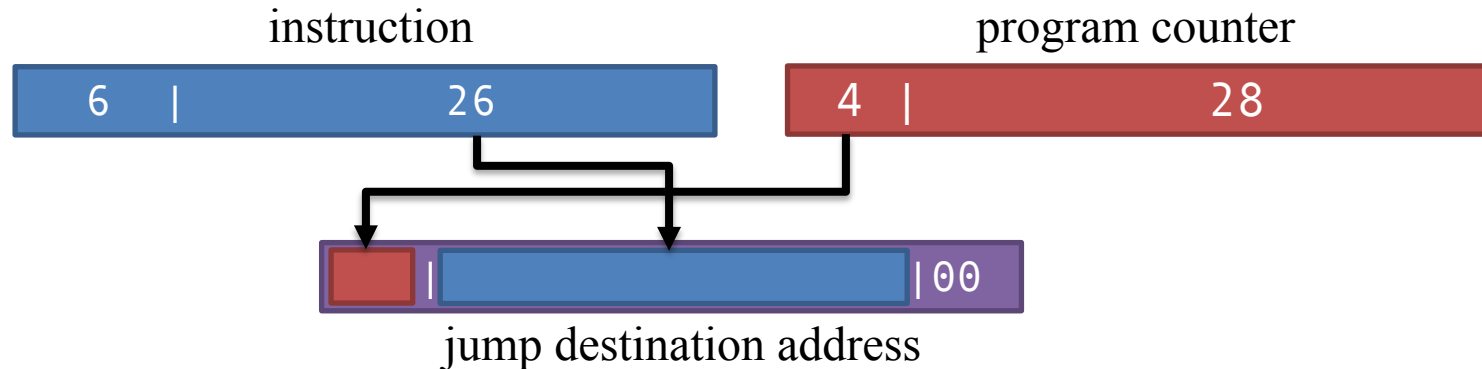


How do you specify the destination of a branch/jump?

- 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 Branch and Jump Addressing Modes

- Branches (e.g., beq) use PC-relative addressing mode
 - uses fewer bits since address typically close
 - Aka: base+displacement mode, with the PC being the base
- Jumps use pseudo-direct addressing mode
 - Recall opcode is 6 bits...
 - How many bits are available for displacement? How far can you jump?
 - 26 bits of the address is in the instruction, the rest is taken from the PC.



MIPS in one slide

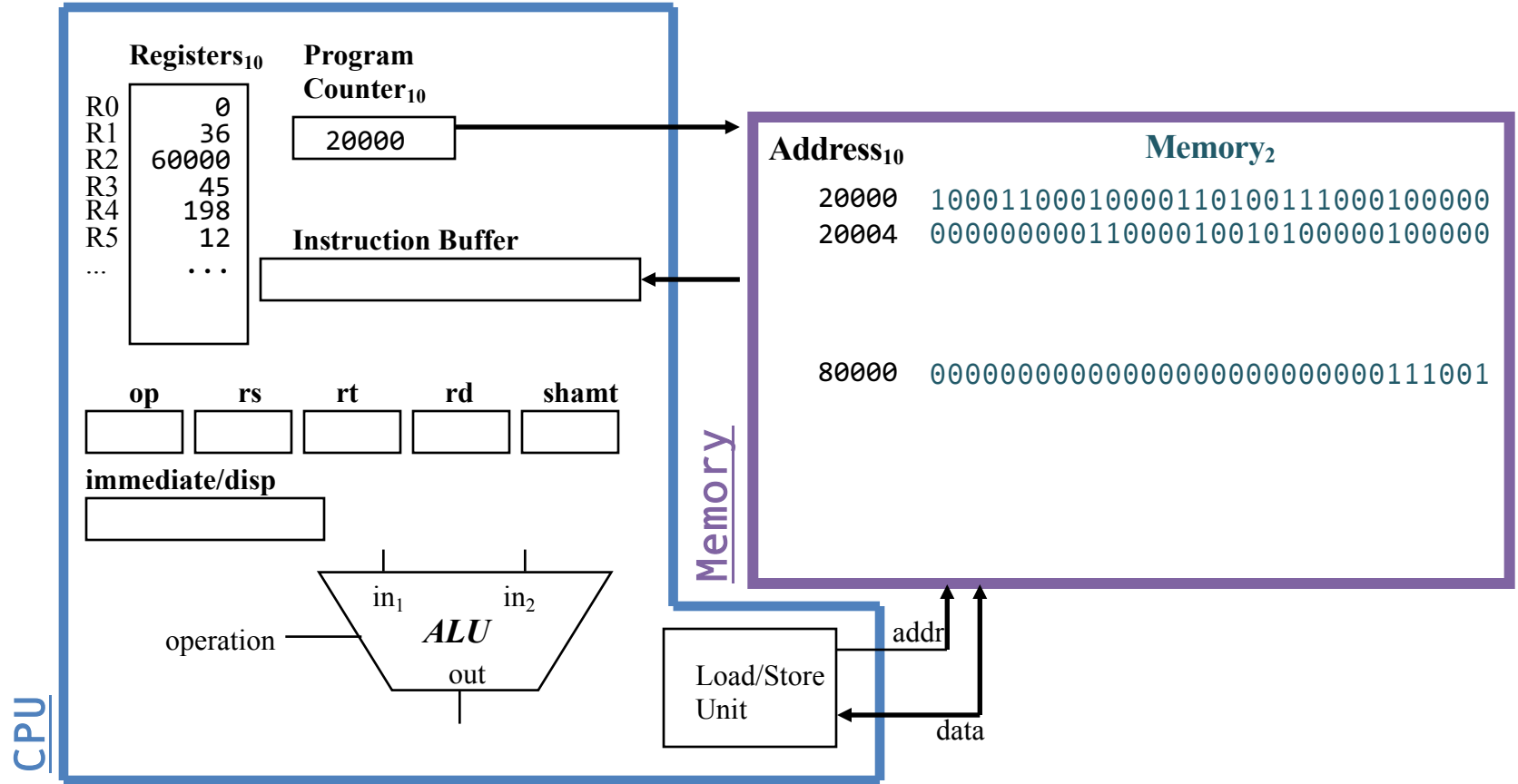
MIPS operands

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($\$s2$)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw $\$s1$, 100($\$s2$)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb $\$s1$, 100($\$s2$)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb $\$s1$, 100($\$s2$)	$\text{Memory}[\$s2 + 100] = \$s1$	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 equal	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
Unconditional 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	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

Review – Instruction Execution in a CPU



Poll Q: Work an Example

- Can we figure out the code?

```
void  
swap(int v[], int k)  
{  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

```
swap:  
muli    $2,    $5, 4  
add     $2,    $4, $2  
lw      $15, 0($2)  
lw      $16, 4($2)  
sw      $16, 0($2)  
sw      $15, 4($2)  
jr      $31
```

	Where is k?
A	\$4
B	\$5
C	\$15
D	\$16
E	None of the above

MIPS ISA Tradeoffs

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-type	OP	rs	rt	rd	sa	funct
I-type	OP	rs	rt	immediate		
J-type	OP	target				

What if?

- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. $Y = AX + B$)

RISC Architectures

- MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
 - fixed instruction length
 - few instruction formats
 - load/store architecture
- RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.

Alternative Architectures

- Design alternative:
 - provide more powerful operations
 - goal is to reduce number of instructions executed
 - danger is a slower cycle time and/or a higher CPI (cycles per instruction)
- Sometimes referred to as “RISC vs. CISC”
 - CISC = Complex Instruction Set Computer (as alt to RISC)
 - virtually all new instruction sets since 1982 have been RISC
 - VAX: minimize code size, make assembly language easy
instructions from 1 to 54 bytes long!
- We’ll look (briefly!) at PowerPC and 80x86
- What is ARM?

PowerPC

- Indexed addressing
 - example: `lw $t1,$a0+$s3 # $t1=Memory[$a0+$s3]`
 - What do we have to do in MIPS?
- Update addressing
 - update a register as part of load (for marching through arrays)
 - example: `lwu $t0,4($s3) # $t0=Memory[$s3+4];$s3=$s3+4`
 - What do we have to do in MIPS?
- Others:
 - load multiple/store multiple
 - a special counter register “bc Loop”
decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added
- 1999: Pentium III (same architecture)
- 2001: Pentium 4 (144 new multimedia instructions), simultaneous multithreading (hyperthreading)
- 2005: dual core Pentium processors
- 2006: quad core (sort of) Pentium processors
- 2009: Nehalem – eight-core multithreaded processors
- 2015: Skylake – 4-core, multithreaded, added hw security features, transactional memory...

80x86

- Complexity:
 - Instructions from 1 to 17 bytes long
 - one operand must act as both a source and destination
 - one operand can come from memory
 - complex addressing modes
 - e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
 - the most frequently used instructions are not too difficult to build
 - compilers avoid the portions of the architecture that are slow

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Historic architectures favored code size over parallelism.
- MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.