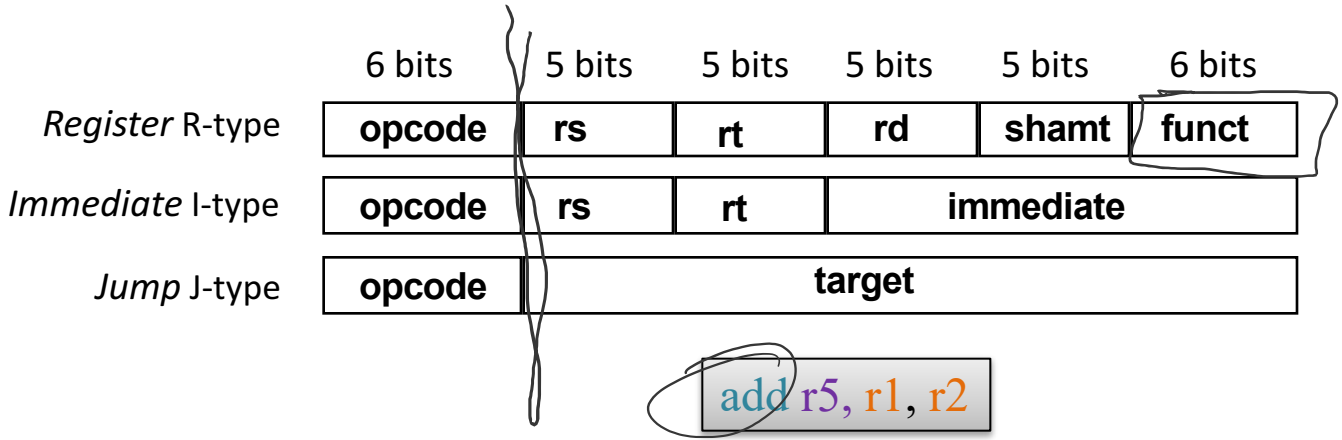
Example of instruction encoding:



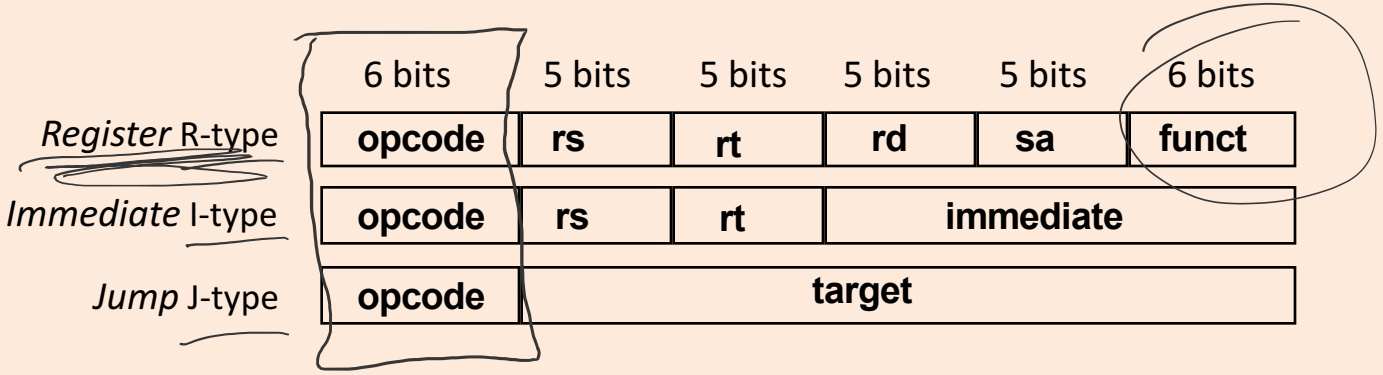
opcode=0, rs=1, rt=2, rd=5, sa=0, funct=32

000000 00001 00010 00101 00000 100000

00000000001000100010100000100000

0x00222420

Poll Q: Implications of the MIPS instruction format



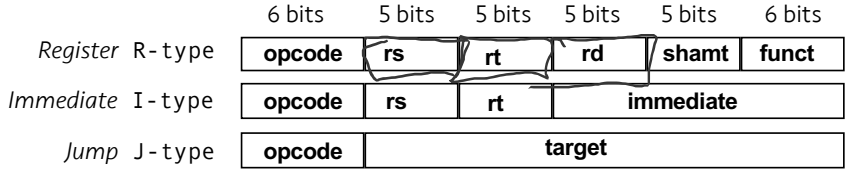
What is the maximum number of unique operations MIPS can encode?

3 $2^6 = 64$ 127 128

Accessing the Operands

aka, what's allowed to go here

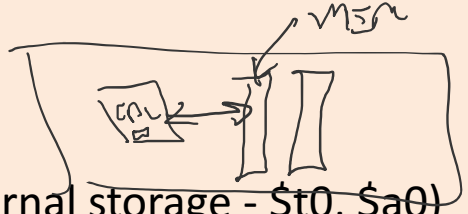
- operands are generally in one of two places:
  - registers (32 options)
  - memory (2^{32} locations)
- registers are
  - easy to specify
  - close to the processor (fast access)
- the idea that we want to use registers whenever possible led to *load-store architectures*.
  - normal arithmetic instructions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



lw

sw

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)

can't do

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

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

— more instructions

\+ fast implementation

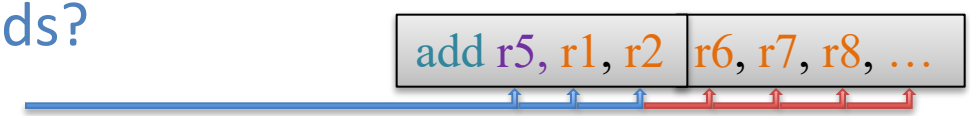
(e.g., easy pipelining)

else

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 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 ← acc + mem[A]
General Purpose Register	3	add A B Rc	mem[A] ← mem[B] + Rc
	2	add A Rc	mem[A] ← mem[A] + Rc
Load/Store:	3	add Ra Rb Rc	Ra ← Rb + Rc
		load Ra Rb	Ra ← mem[Rb]
		store Ra A	mem[A] ← 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

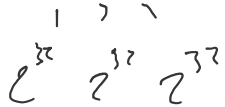
Code sequence for C = A + B for four classes of instruction sets:

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:

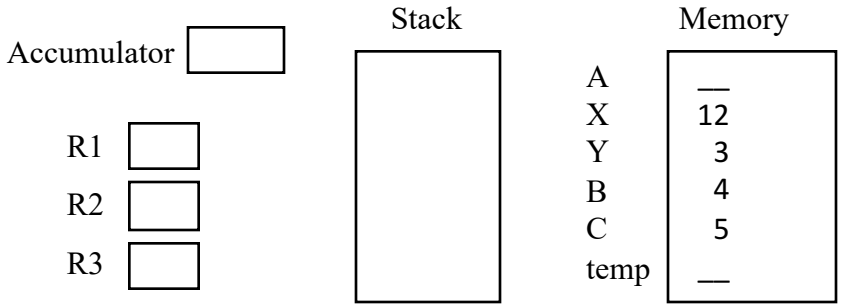
Stack	Accumulator	GP Register (register-memory)	GP Register (load-store)
Push A	**Load A**	**ADD C, A, B**	**Load R1,A**
Push B	**Add B**		**Load R2,B**
Add	**Store C**		**Add R3,R1,R2**
Pop C			**Store C,R3**

Exercise: Working through alternative ISAs

[if time]

A = X*Y - B*C

Stack Architecture | Accumulator | GPR | GPR (Load-store)



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

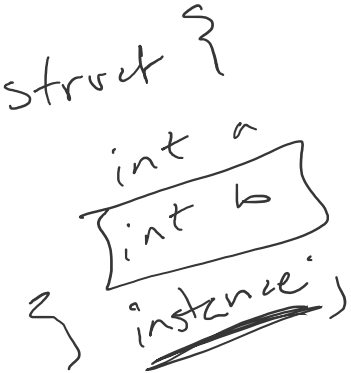
- Only the top of the stack
- Only the accumulator register
- Any general purpose register
- Any general purpose register or anywhere in memory
- Any general purpose register or the top of the stack

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

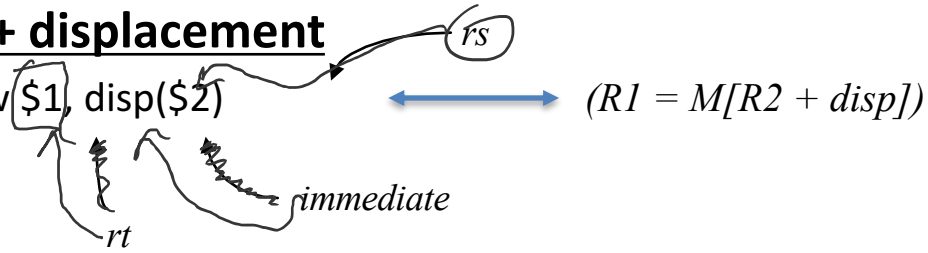
add $1, $2, $3

immediate

add $1, $2, #35

base + displacement

lw $1, disp($2)



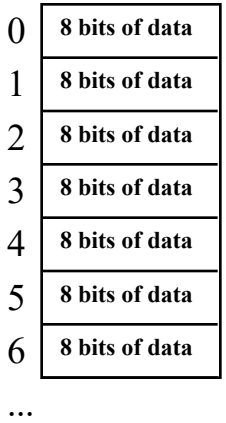
register indirect
⇨ disp = 0
absolute
⇨ (rs) = 0

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)

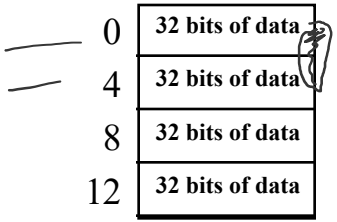
What does memory look like anyway?

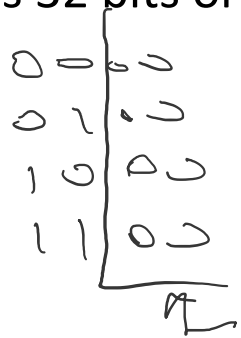
- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index (address) points to a byte of memory.

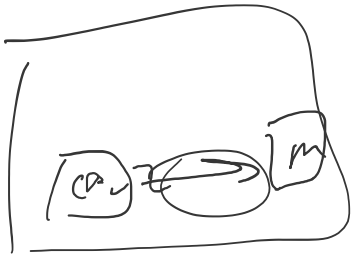


Memory accesses are (often) required to be “word-aligned” because of how buses and memory work

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







bytes 1-5

- 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: SQRT, MOD, SIN COS, ADC add w/ carry
- logical
  - and, or, shift left, shift right, XOR
  - But not: BIC &= &a imm
- data transfer
  - load word, store word
  - But not: lw {R3++}

“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”)