

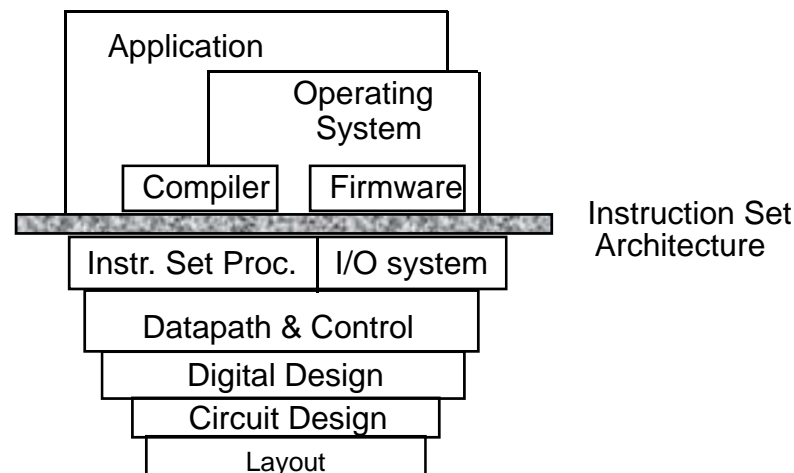
CS M151B / EE M116C

Computer Systems Architecture

Instruction Set Architectures

The Instruction Set Architecture

- That part of the architecture that is visible to the programmer
 - instruction formats
 - opcodes (available instructions)
 - number and types of registers
 - storage access, addressing modes
 - exceptional conditions

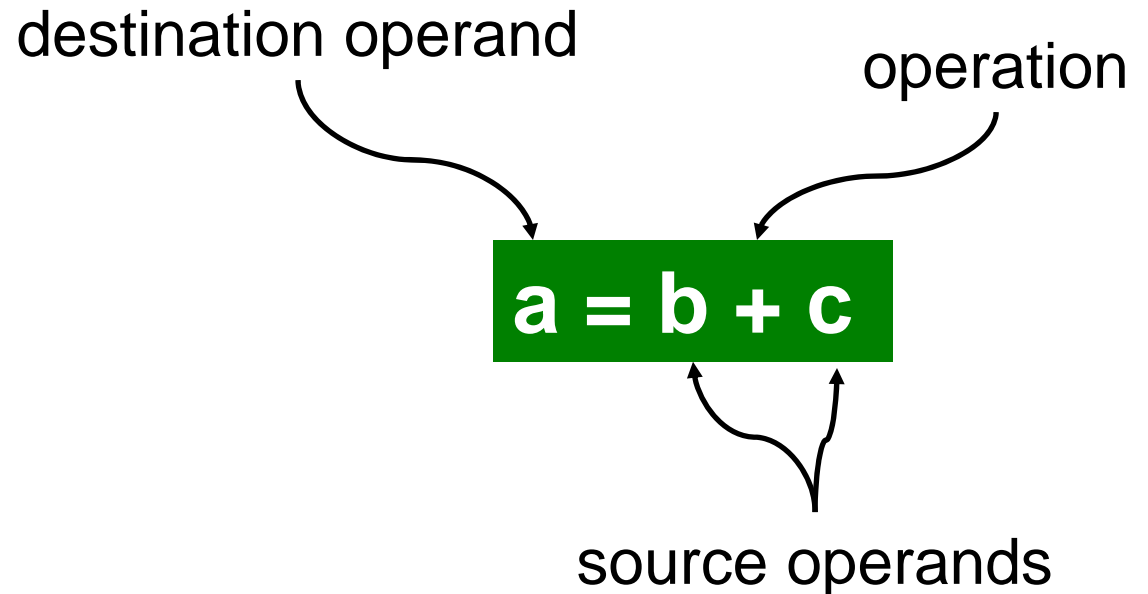


Overall Goals of ISA

- Can be implemented by simple hardware
- Can be implemented by fast hardware
- Instructions do useful things
- Easy to write (or generate) machine code
- We'll be using the MIPS ISA

Key ISA Decisions

- Operations
 - how many?
 - which ones?
 - length?
- Operands
 - how many?
 - location
 - types
 - how to specify?
- Instruction format
 - size
 - how many formats?



Main ISA Classes

- **CISC** (“Complex Instruction Set Computers”)
 - Digital’s VAX (1977) and Intel’s x86 (1978)
 - large # of instructions
 - many specialized complex instructions
- **RISC** (“Reduced Instruction Set Computers”)
 - almost all machines of 80’s and 90’s are RISC
 - MIPS, PowerPC, DEC Alpha, IA64
 - relatively fewer instructions
 - enable pipelining and parallelism

Instruction Length

Variable:

x86 – Instructions vary from 1 to 17 Bytes long

VAX – from 1 to 54 Bytes

Fixed:

MIPS, PowerPC, and most other RISC's:

all instruction are 4 Bytes long

Instruction Length

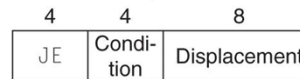
- Variable-length instructions (x86, VAX):
 - require multi-step fetch and decode
 - + allow for a more flexible and compact instruction set (motivated by scarce instruction memory at the time)
- Fixed-length instructions (RISC's)
 - + allow easy fetch and decode
 - + simplify pipelining and parallelism
 - instruction bits are scarce
- All MIPS instructions are 32 bits long

Instruction Formats

- What does each bit mean?
- Having many different instruction formats...
 - complicates decoding
 - uses instruction bits (to specify the format)

Some x86 ISA Formats

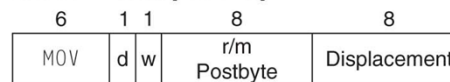
a. JE EIP + displacement



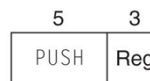
b. CALL



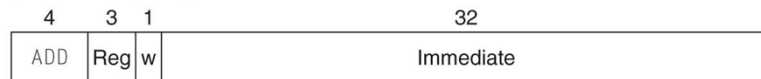
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



MIPS Instruction Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
r format	OP	rs	rt	rd	sa	funct
i format	OP	rs	rt	immediate		
j format	OP	target				

- for instance, “**add r1, r2, r3**” has
 - OP=0, rs=2, rt=3, rd=1, sa=0, funct=32
 - 000000 00010 00011 00001 00000 100000
- opcode (OP) tells the machine which format

Where Are the Operands?

- Operands can be in registers or in memory
- Registers
 - All computers have a small set of registers
 - Memory to hold values that will be used soon
 - Typical instruction will use 2 or 3 register values
 - Some advantages of a small number of registers:
 - It requires fewer bits to specify which one.
 - Less hardware
 - Faster access (shorter wires, fewer gates)
 - Faster context switch (when all registers need saving)
 - Some advantages of a larger number:
 - Fewer load and store instructions needed
 - Easier to do several operations at once

How Many Registers?

- VAX – 16 registers
 - R15 is program counter (PC)
 - Elegant! Loading R15 is a jump instruction
- x86 – 8 (16) general purpose regs (some restrictions apply)
 - Plus floating point and special purpose registers
- Most RISC's have 32 int and 32 floating point regs
 - Plus some special purpose ones
 - PowerPC has 8 four-bit “condition registers”, a “count register” (to hold loop index), and others.
- Itanium (IA64) has 128 fixed, 128 floating point, and 64 “predicate” registers

Where Do Operands Reside?

- Accumulator machine:
 - Only 1 register (called the “accumulator”)
 - Instructions include “store” and “acc \leftarrow acc + mem”
- Stack machine:
 - no registers, just a stack of value locations
 - “Push **A**” loads memory location **A** onto the stack, this value is now the top of stack (1st entry of stack).
 - “Pop **A**” stores the top of stack into memory location **A**, then removes the top of stack.
 - “Add” places the sum of the first two stack entries in the 2nd entry of the stack, then removes the top of stack.
- Register-Memory machine :
 - Arithmetic instructions can use data in registers and/or memory
- Load-Store Machine (aka Register-Register Machine):
 - Arithmetic instructions can only use data in registers.

Comparing the Number of Instructions

Code sequence for **$C = A + B$**

<u>Stack</u>	<u>Accumulator</u>	<u>Register-Memory</u>	<u>Load-Store</u>
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

Load-Store Architectures

can do:

`add r1=r2+r3`

`load r3, M(address)`

`store r1, M(address)`

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

can't do:

`add r1=r2+M(address)`

- more instructions
- + fast implementation (e.g.,
easy pipelining)

Alternate ISAs

$$A = X * (Y + Z)$$

Stack	Accumulator	Reg-Mem	Load-store
Push X	Load Y	\$temp1 = Y + Z	Load R1, X
Push Y	Add Z	A = X * \$temp1	Load R2, Y
Push Z	Multiply X		Load R3, Z
Add	Store A		Add R4, R2, R3
Multiply			Mult R5, R4, R1
Pop A			Store A, R5

How Many Operands?

- Two-address code: target is same as one operand
 - E.g., $x = x + y$
- Three-address code: target can be different
 - E.g. $x = y + z$
 - x86 doesn't have three-address instructions; others do
- Some operands are also specified implicitly
 - “condition code” setting shows if result was +, 0, or –
 - PowerPC's “**Branch on count**” uses special “count register”
- Well-known ISA's have 0-4 (explicit) operands
 - PowerPC has “**float multiply add**”, $r = x + y * z$
 - It also has fancy bit-manipulation instructions

Addressing Modes

- Where do we get the operand?
- Immediate #25
 - The operand (25) is part of the instruction
- Register R3 (sometimes written \$3)
 - The operand is the contents of register 3
- Register indirect M[R3]
 - Use contents of R3 as address into memory; find the operand there
- Base+Displacement M[R3 + 160]
 - Add the displacement (160) to contents of R3, look in that memory location for the operand
 - If register is PC, this is “PC-relative addressing”
- All our example ISA's have the above modes

More Addressing Modes

- Base+Index $M[R3 + R4]$
 - Add contents of R3 and R4 to get memory address
- Autoincrement $M[R3++]$ (or $M[R3+=d]$)
 - Find value in memory location designated by R3, but also increment R3
 - Useful for accessing array elements
 - Autodecrement is similar
- Scaled Index $M[R3 + R4*d]$
 - Multiply R4 by d (d is typically 1,2,4, or 8), then add R3 to get memory address
- Memory Indirect $M[M[R3]]$
 - Find number in memory location R3, use THAT as address into memory to find
- None of the above are included in MIPS ISA

VAX Addressing Mode Usage

- **Half** of all references were register-mode
- Remaining half distributed as follows:

Program	Base + Displacement	Immediate	Scaled Index	Memory Indirect	All Others
TEX	56%	43%	0%	1%	0%
Spice	58%	17%	16%	6%	3%
GCC	51%	39%	6%	1%	3%

- Similar measurements show that
 - 16 bits is enough for the immediate field 75-80% of the time
 - 16 bits is enough for displacement 99% of the time

MIPS Addressing Modes

register

OP	rs	rt	rd	sa	funct
----	----	----	----	----	-------

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

immediate

OP	rs	rt	immediate
----	----	----	-----------

addi \$1, \$2, #35

rs (points to \$2)
rt (points to \$1)
immediate value (points to #35)

base + displacement

lw \$1, 24(\$2)

displacement (points to 24)

register indirect

⇒ $disp = 0$

absolute

⇒ $(rs) = 0$

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

lhi \$s0, 61

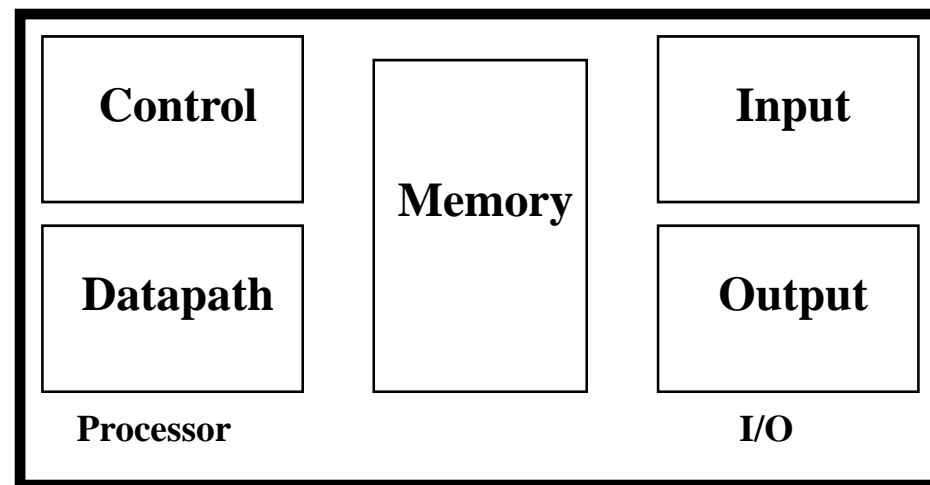
0000 0000 0111 1101	0000 0000 0000 0000
---------------------	---------------------

ori \$s0, \$s0, 2304

0000 0000 0111 1101	0000 1001 0000 0000
---------------------	---------------------

Registers vs. Memory

- Arithmetic instruction operands must be registers,
 - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?



Memory Organization

- 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 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 Organization

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

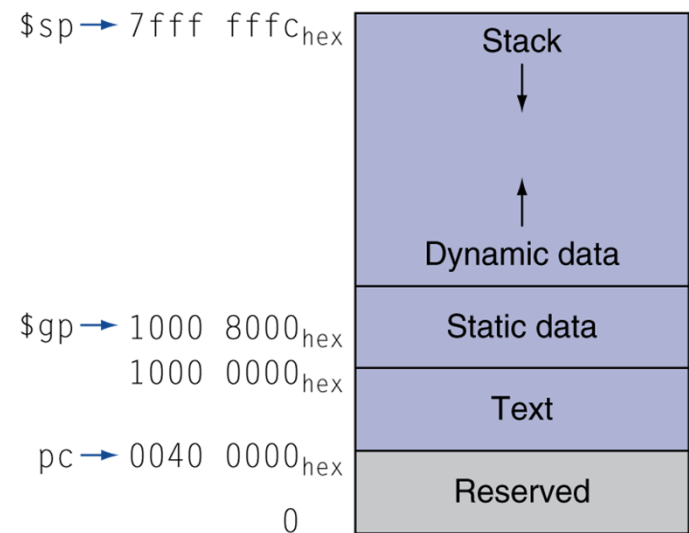
0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
i.e., what are the least 2 significant bits of a word address?

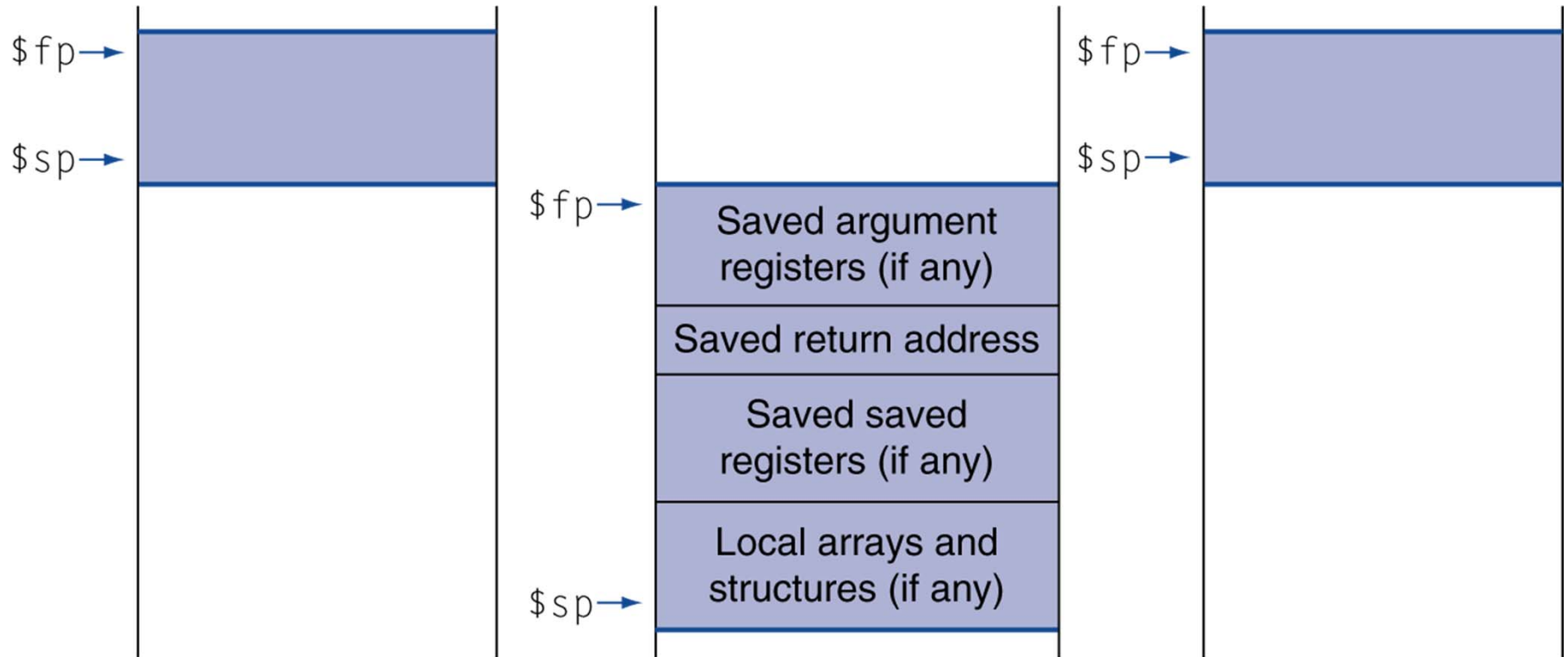
MIPS Memory Allocation

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - \$gp initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage



Stack Allocation

High address



Low address

a.

b.

c.

MIPS ISA Decisions

- Instruction length
 - all instructions are 32 bits long
- How many registers?
 - 32 general purpose registers (R0 always 0)
- Where do operands reside?
 - load-store architecture
- Instruction formats
 - three (r-, i-, and j-format)
- Operands
 - 3-address code
 - immediate, register, and base+displacement modes

MIPS Operations

- arithmetic
 - add, subtract, multiply, divide, ...
- logical
 - and, or, shift left, shift right, ...
- data transfer
 - load word, store word
- conditional branch
 - beq & bne are PC-relative, since most targets are nearby
- unconditional jump
 - jump, jump register, branch and link

Arithmetic and Logical Ops

R-type	OP	rs	rt	rd	sa	funct
--------	----	----	----	----	----	-------

add \$s1, \$s2, \$s3 # \$s1 = \$s2 + \$s3

and \$s1, \$s2, \$s3 # \$s1 = \$s2 AND \$s3

I-type	OP	rs	rt	immediate
--------	----	----	----	-----------

addi \$s1, \$s2, 1 # \$s1 = \$s2 + 1

Load and Store Instructions

- Example:

C code: `A[12] = h + A[8];`

MIPS code: `lw $t0, 32($s3)`
 `add $t0, $s2, $t0`
 `sw $t0, 48($s3)`

- Can refer to registers by name (e.g., \$s2, \$t2) instead of number
- Store word has destination last

I-type

OP	rs	rt	immediate
----	----	----	-----------

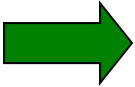
Conditional Branches

- Decision making instructions
 - alter the control flow,
 - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:
 - `bne $t0, $t1, Label`
 - `beq $t0, $t1, Label`
- Example: if (i==j) h = i + j;
 - `bne $s0, $s1, Label`
 - `add $s3, $s0, $s1`
 - `Label: `
- How do we specify the target address of a branch?
 - studies show that almost all conditional branches go relatively short distances from the current PC
 - specify a relative branch target address! (*RELATIVE TO PC+4*)

op	rs	rt	16 bit address
----	----	----	----------------

More Conditional Branches

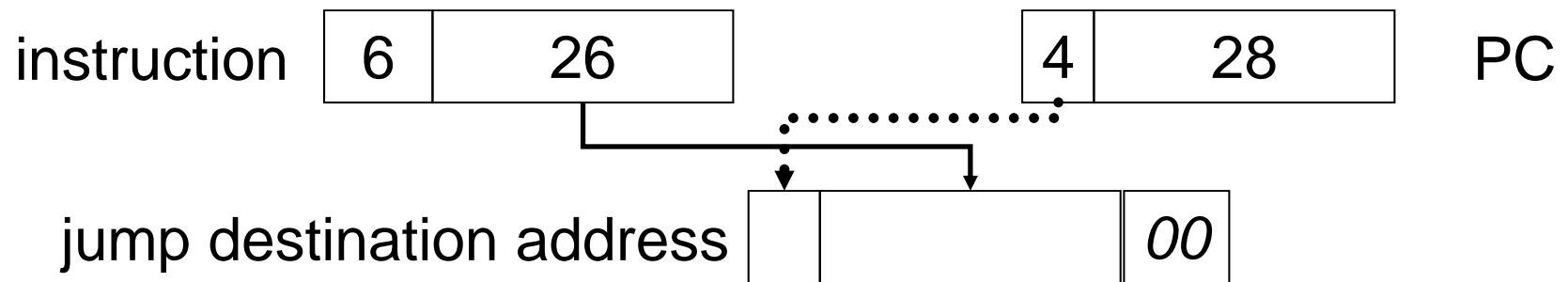
- We have: beq, bne, what about Branch-if-less-than?
- New instruction slt:

<code>slt \$t0, \$s1, \$s2</code>		<code>if \$s1 < \$s2 then</code> <code> \$t0 = 1</code> <code> else</code> <code> \$t0 = 0</code>
-----------------------------------	--	--

- beq, bne, and slt can be used to build all fundamental branch conditions
 - always, !=, ==, >, <=, >=, <, ...
 - note that the assembler needs a register to do this

Unconditional Branches

- jump (j)
 - 26-bit **word address** (represents 28-bit byte address)



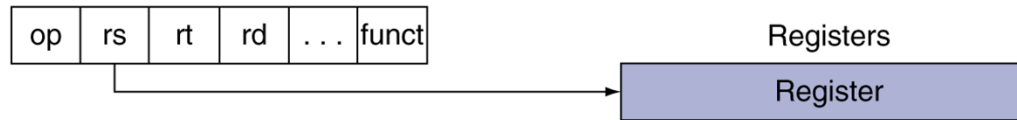
- jump and link (jal)
 - procedure calls - modifies PC; saves PC+4 to \$r31
- jump register (jr)
 - returns
 - jumps larger than the given 26 bits

Addressing Modes, Data and Control Transfer

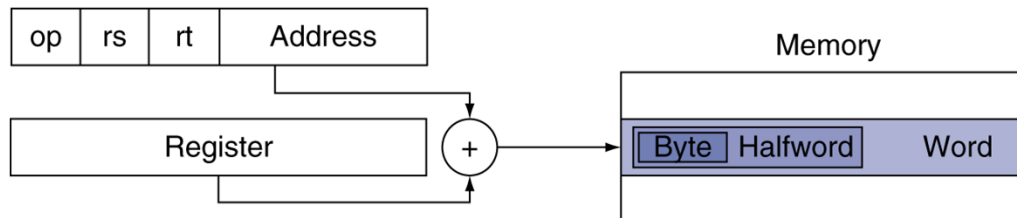
1. Immediate addressing



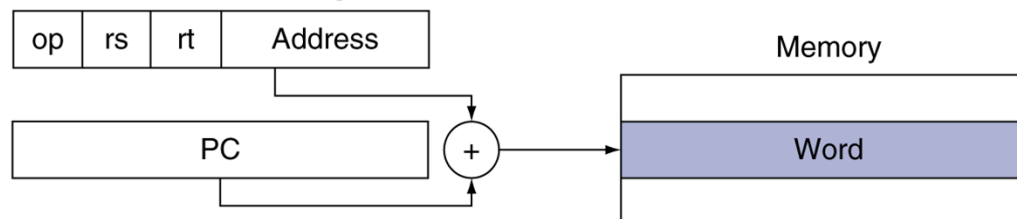
2. Register addressing



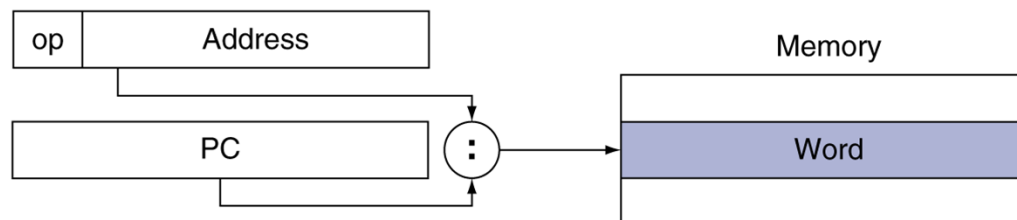
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



To Summarize:

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 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 = PC + 4; go to 10000	For procedure call

Example

assume:

- \$s4 holds the base address of array v[]
- \$s5 holds the integer k
- v is an array of words

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



swap:

```
sll $t2, $s5, 2    # Multiply k by 4 to get byte address  
add $t2, $s4, $t2  # Add this address to base address  
                   # of array v[]  
  
lw $t3, 0($t2)     # load v[k] into $t3  
lw $t4, 4($t2)     # load v[k+1] into $t4  
                   # this is k+1 because we add 4 to $t2  
  
sw $t4, 0($t2)     # store $t4 to v[k]  
sw $t3, 4($t2)     # store $t3 to v[k+1]  
jr $31             # return from the call to swap  
                   # use the contents of $31 as the new  
                   # PC
```

MIPS ISA Tradeoffs

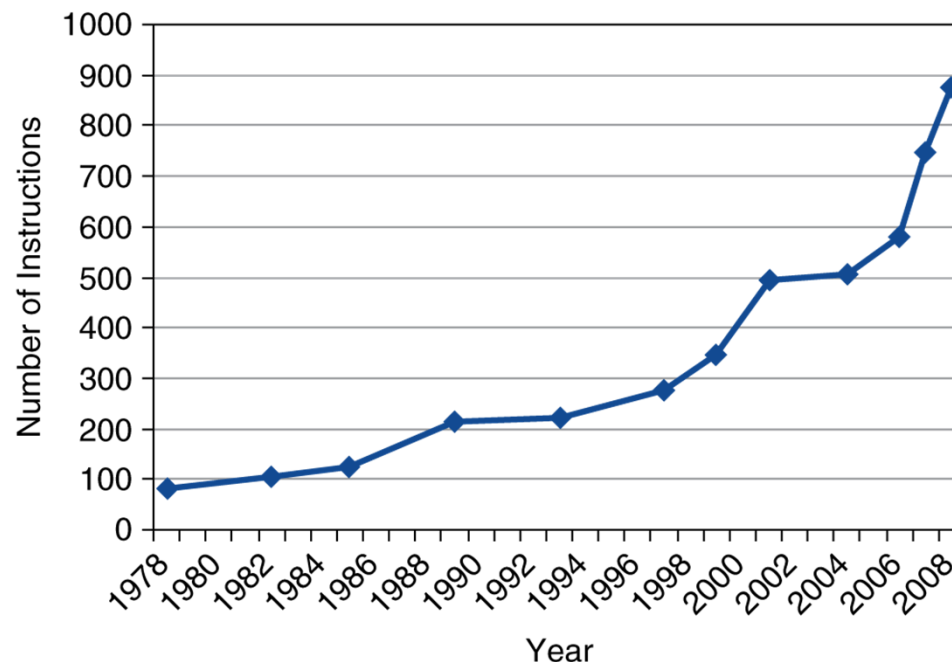
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
OP	rs	rt	rd	sa	funct
OP	rs	rt	immediate		
OP	target				

What if?

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

Fallacies

- Powerful instruction \Rightarrow higher performance?
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Backward compatibility \Rightarrow ISA doesn't change?



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped

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
- Four principles of IS architecture
 - simplicity favors regularity
 - smaller is faster
 - good design demands compromise
 - make the common case fast