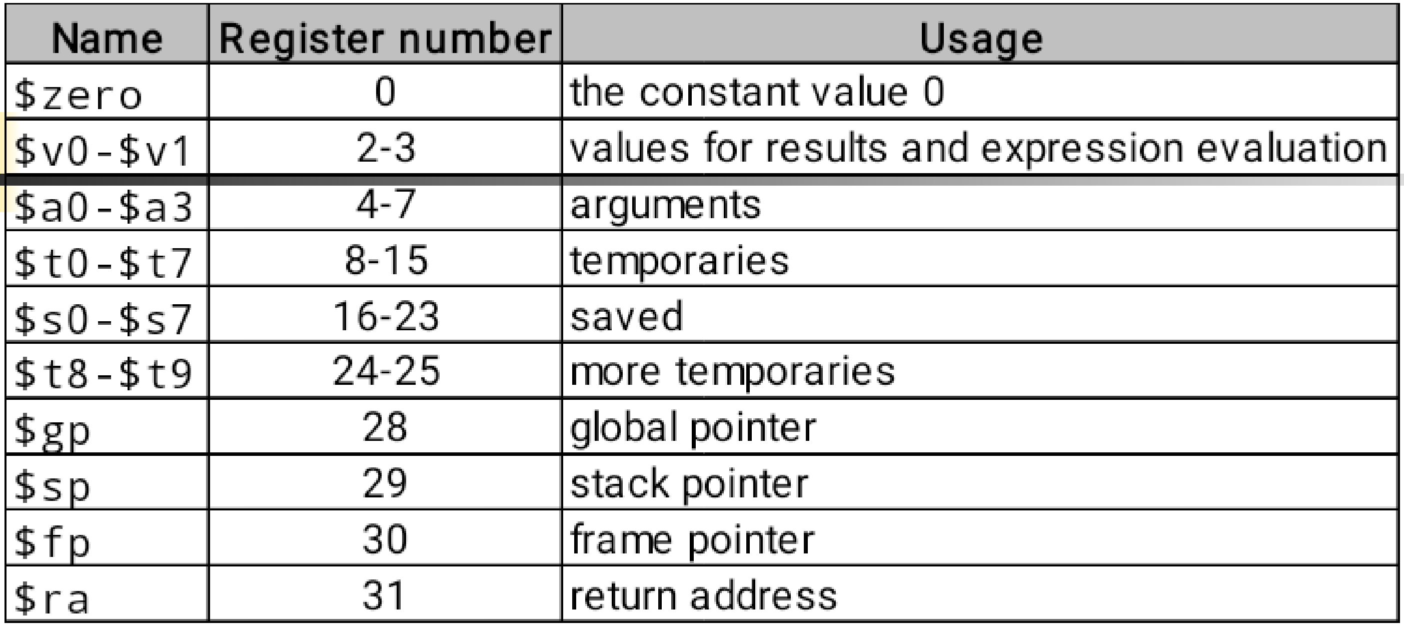
EE204: Computer Architecture

Chapter 2: MIPS Instructions

Instructions: Overview

- Language of the machine
- More primitive than higher level languages, e.g., no sophisticated control flow such as *while* or *for* loops
- Very restrictive
  - e.g., MIPS arithmetic instructions
- We'll be working with the MIPS instruction set architecture
  - inspired most architectures developed since the 80's
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - the name is not related to *millions of instructions per second* !
  - it stands for **m***icrocomputer without* **i***nterlocked* **p***ipeline* **s***tages* !
- <u>Design goals</u>: *maximize performance* and *minimize cost* and *reduce design time*

Policy-of-Use Convention for Registers

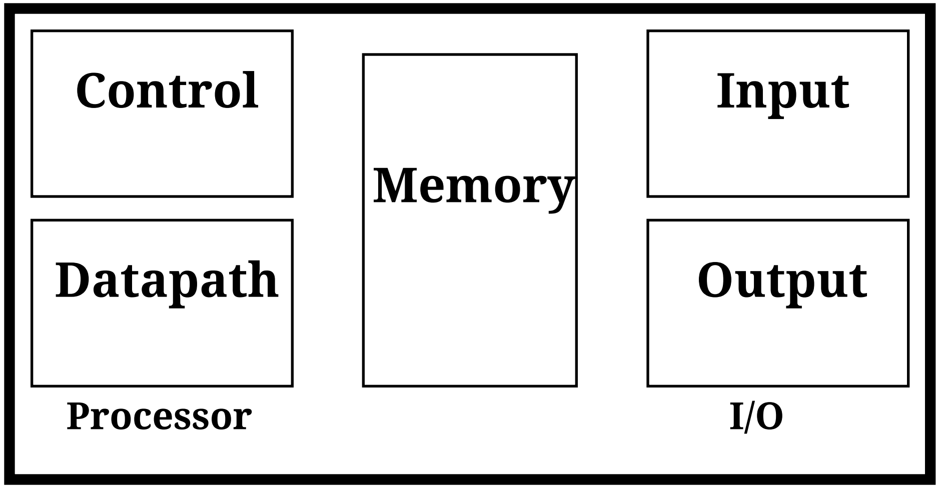


Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Register 1, called $at, is reserved for the assembler; registers 26-27,
called $k0 and $k1 are reserved for the operating system.

Registers vs. Memory

- Arithmetic instructions operands must be in registers
  - MIPS has 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)? Use *memory*, *load/store* operations to transfer data from memory to register – if not enough registers *spill registers* to memory
- *MIPS is a load/store architecture*



Overview of MIPS

- Simple instructions – all 32 bits wide
- Very structured – no unnecessary baggage
- Only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp (29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, 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 register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	three register operands
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch

	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1 = 0
				Compare less than; used with beq, bne

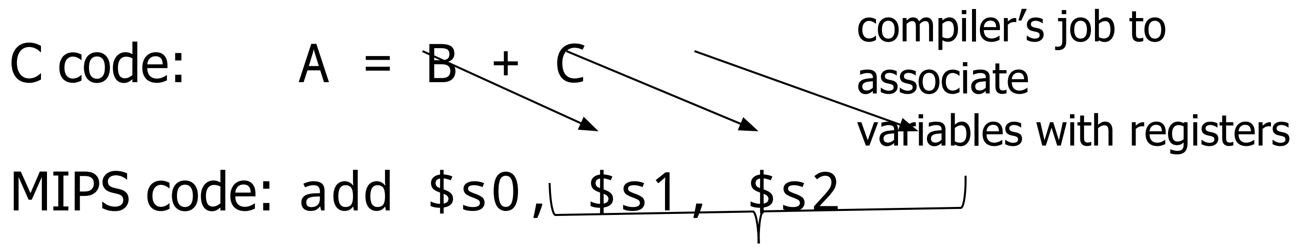
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1 = 0
				Compare less than immediate; used with beq, bne

Unconditional jump	jump	j L	go to L	Jump to target address
	jump register	jr $ra	go to $ra	For procedure return
	jump and link	jal L	$ra = PC + 4; go to L	For procedure call

MIPS Arithmetic

- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)

- *Example* :



C code: A = B + C

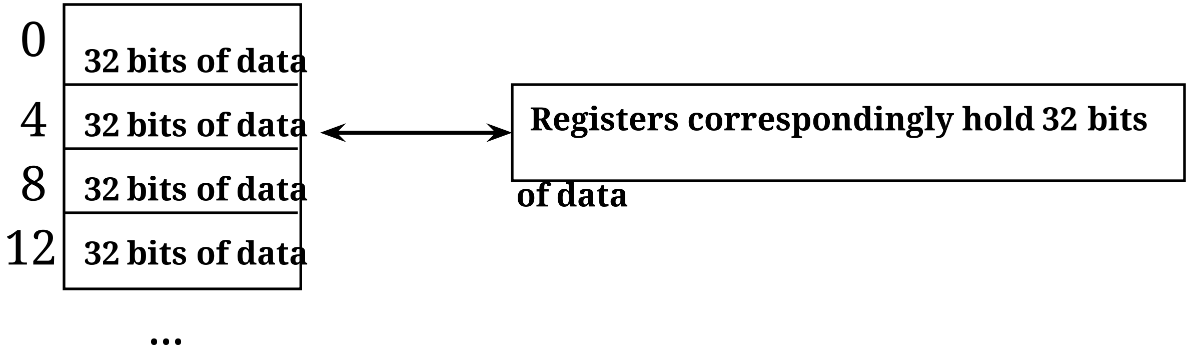
MIPS code: add $s0, $s1, $s2

compiler’s job to associate variables with registers

- Operands must be in registers – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers

Memory Organization

- Bytes are load/store units, 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 correspondingly 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
  - i.e., words are *aligned*
  - *what are the least 2 significant bits of a word address?*

Load/Store Instructions

- *Load* and *store* instructions
- *Example* :

```
C code:      A[8] = h + A[8];
                              value       offset     address
 MIPS code      (load):        lw   $t0, 32($s3)
             (arithmetic):    add $t0, $s2, $t0
              (store):     sw   $t0, 32($s3)
```

- Load word has destination first, store has destination last
- Remember MIPS arithmetic operands are registers, not memory locations
  - therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory

So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100]= $s1`

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - *Example*: `add $t0, $s1, $s2`
  - registers are numbered, e.g., `$t0 is 8, $s1 is 17, $s2 is 18`
- Instruction Format **R-type** ("R" for aRithmetic):

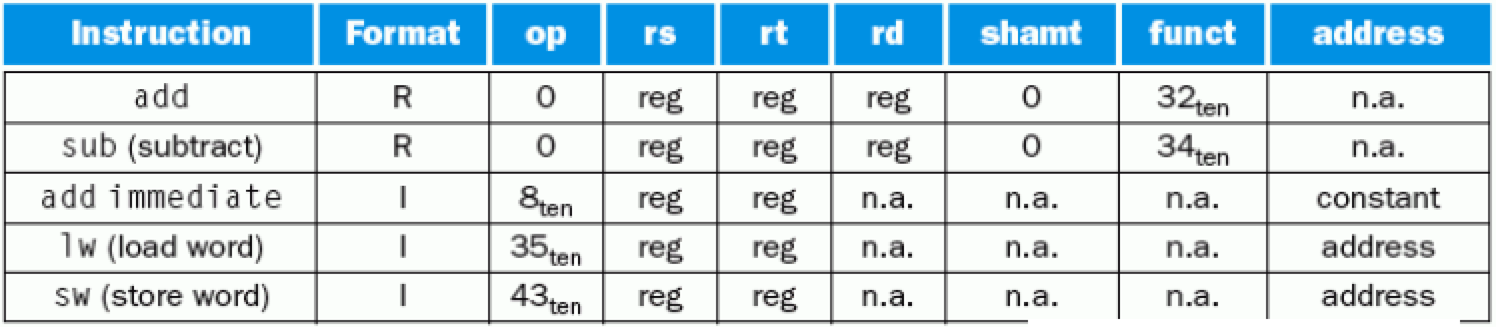
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
opcode – operation	first register source operand	second register source operand	register destination operand	shift amount	function field - selects variant of operation
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-type instr

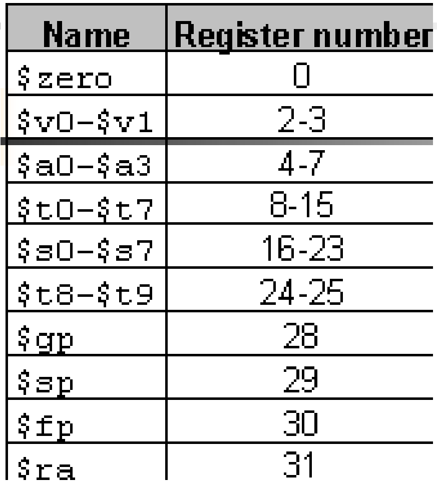
- Consider the load-word and store-word instructions,
  - what would the regularity principle have us do?
    - we would have only 5 or 6 bits to determine the offset from a base register - too little…

- Design Principle 3: *Good design demands a compromise*
- Introduce a new type of instruction format
  - **I-type** ("I" for Immediate) for data transfer instructions
  - *Example*: `lw $t0, 1002($s2)`

100011	10010	01000	0000001111101010
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit offset



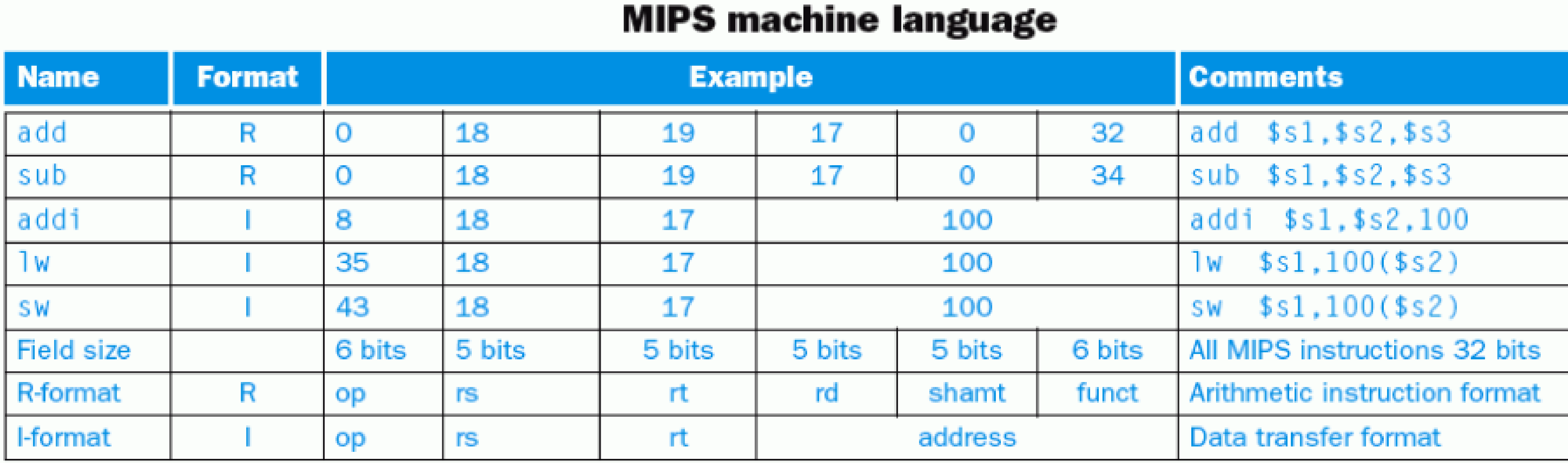
Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address



Name	Register number
$zero	0
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$gp	28
$sp	29
$fp	30
$ra	31

1. lw $t0, 1200($t1)
2. add $t0, $s2, $t0
3. sw $t0, 1200($t1)

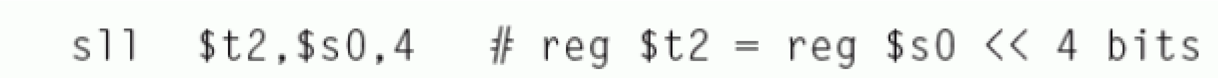
Another look at R-type and I-type instructions



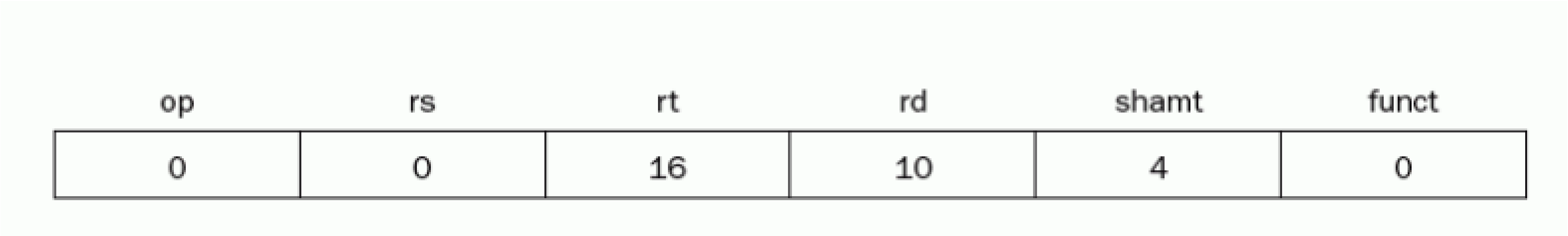
MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Logical Operations

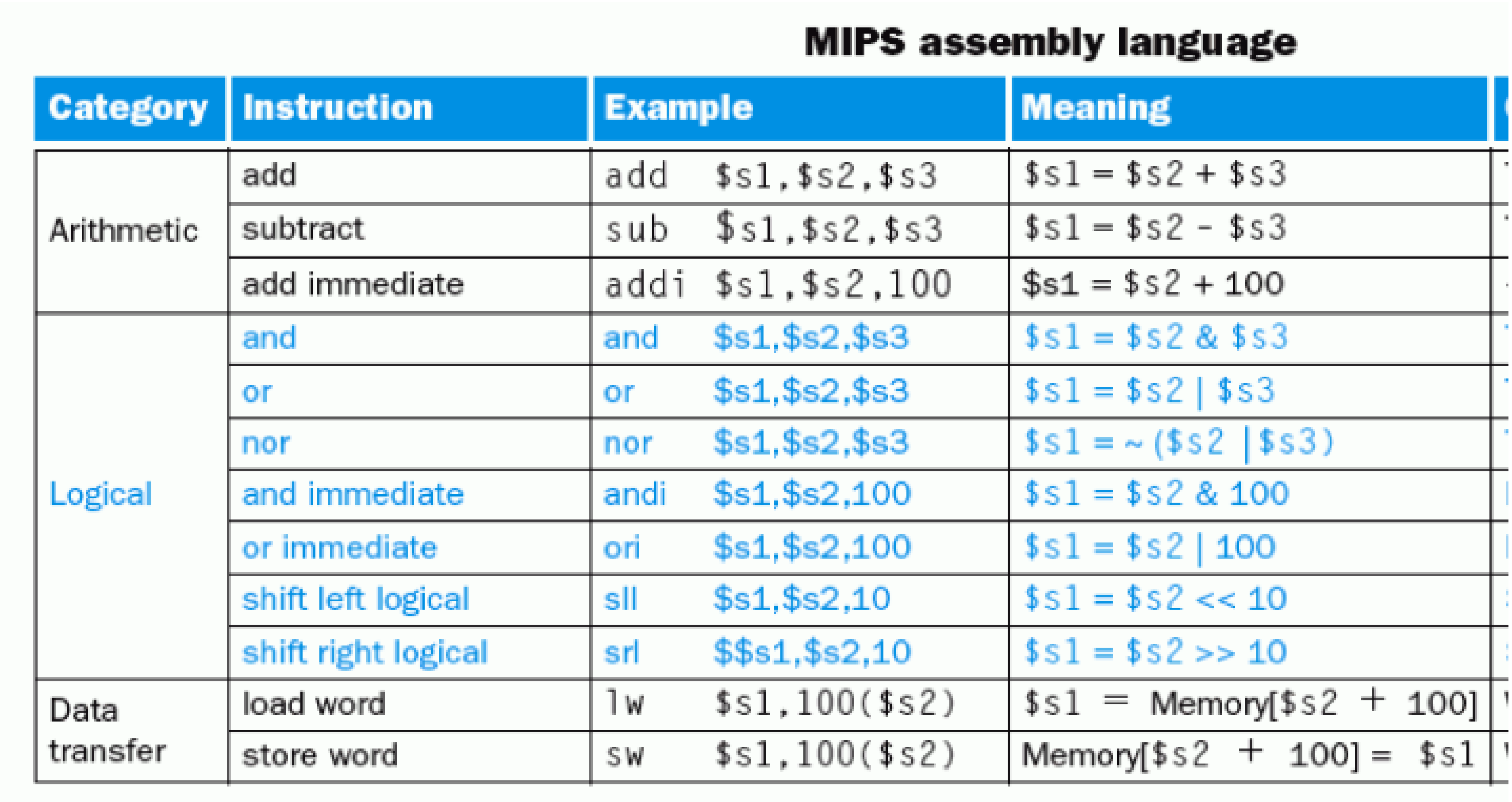


```
sll  $t2,$s0,4    # reg $t2 = reg $s0 << 4 bits
```



op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Logical Operations



MIPS assembly language

Category	Instruction	Example	Meaning
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1

Control: Conditional Branch

- 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
```

I-type instructions

000100	01000	01001	0000000000011001

⟷ `beq $t0, $t1, Label` (= addr.100)

word-relative addressing :
25 words = 100 bytes;
also *PC-relative* (more…)

- *Example* : if (i==j) h = i + j;

```
    bne $s0, $s1, Label
    add $s3, $s0, $s1
Label:  ....
```

Addresses in Branch

- Instructions:

`bne $t4,$t5,Label`	Next instruction is at Label if $t4 != $t5
`beq $t4,$t5,Label`	Next instruction is at Label if $t4 = $t5

- Format:

I	op	rs	rt	16 bit offset

- 16 bits is too small a reach in a 2^{32} address space

- Solution: specify a register (as for `lw` and `sw`) and add it to offset
  - use PC (= program counter), called *PC-relative* addressing, based on
  - *principle of locality* : most branches are to instructions near current instruction (e.g., loops and *if* statements)

Control: Unconditional Branch (Jump)

- MIPS unconditional branch instructions:

```
          j Label
```

- *Example* :

```
if (i!=j)          beq $s4, $s5, Lab1
      h=i+j;       add $s3, $s4, $s5
else           j Lab2
      h=i-j;       Lab1: sub $s3, $s4, $s5
              Lab2: ...
```

- **J-type** ("J" for Jump) instruction format
  - *Example* : `j Label # addr. Label = 100`

word-relative addressing :
25 words = 100 bytes

000010	00000000000000000000011001
6 bits	26 bits
op	26 bit number

Addresses in Jump

- Word-relative addressing also for jump instructions

J	op	26 bit address

- MIPS jump `j` instruction replaces *lower* 28 bits of the PC with A00 where A is the 26 bit address; it *never changes* upper 4 bits
  - *Example* : if `PC = 1011X` (where X = 28 bits), it is replaced with `1011A00`
  - there are 16(=2^4) partitions of the 2^{32} size address space, each partition of size 256 MB (=2^{28}), *such that* , in each partition the upper 4 bits of the address is same.
  - if a program crosses an address partition, then a `j` that reaches a different partition has to be replaced by `jr` with a full 32-bit address first loaded into the jump register
  - therefore, OS should always try to load a program inside a single partition

Constants

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

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

- Solutions? Will these work?
  - create hard-wired registers (like $zero) for constants like 1
  - put program constants in memory and load them as required

- MIPS Instructions:

  ```
  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4
  ```

- *How to make this work?*

Immediate Operands

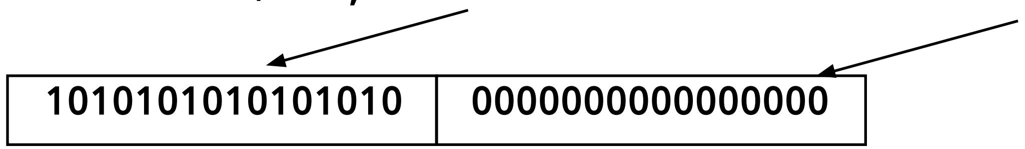
- Make operand part of instruction itself!
- <u>Design Principle 4</u>: *Make the common case fast*
- *Example*: `addi $sp, $sp, 4 # $sp = $sp + 4`

001000	11101	11101	0000000000000100
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

How about larger constants?

- First we need to load a 32 bit constant into a register
- Must use two instructions for this: first new *load upper immediate* instruction for upper 16 bits

```
    lui $t0, 1010101010101010
```

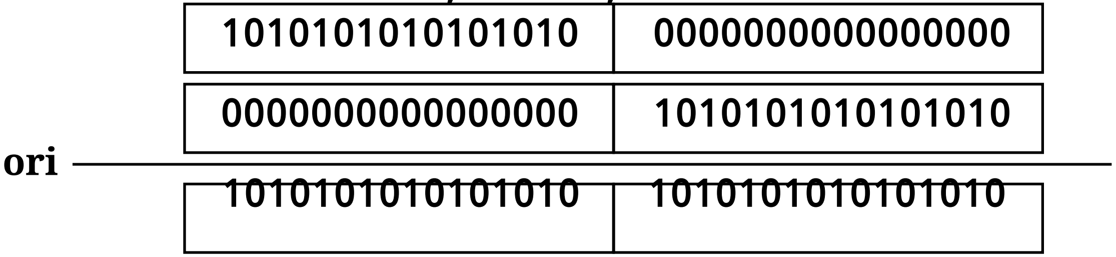


filled with zeros

1010101010101010 | 0000000000000000

- Then get lower 16 bits in place:

```
    ori $t0, $t0, 1010101010101010
```



1010101010101010 | 0000000000000000

0000000000000000 | 1010101010101010

ori

1010101010101010 | 1010101010101010

- Now the constant is in place, use register-register arithmetic

So far

Instruction	Format	Meaning
add $s1,$s2,$s3	R	$s1 = $s2 + $s3
sub $s1,$s2,$s3	R	$s1 = $s2 – $s3
lw $s1,100($s2)	I	$s1 = Memory[$s2+100]
sw $s1,100($s2)	I	Memory[$s2+100] = $s1
bne $s4,$s5,Lab1	I	Next instr. is at Lab1 if $s4 != $s5
beq $s4,$s5,Lab2	I	Next instr. is at Lab2 if $s4 = $s5
j Lab3	J	Next instr. is at Lab3

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Assembly Language vs. Machine Language

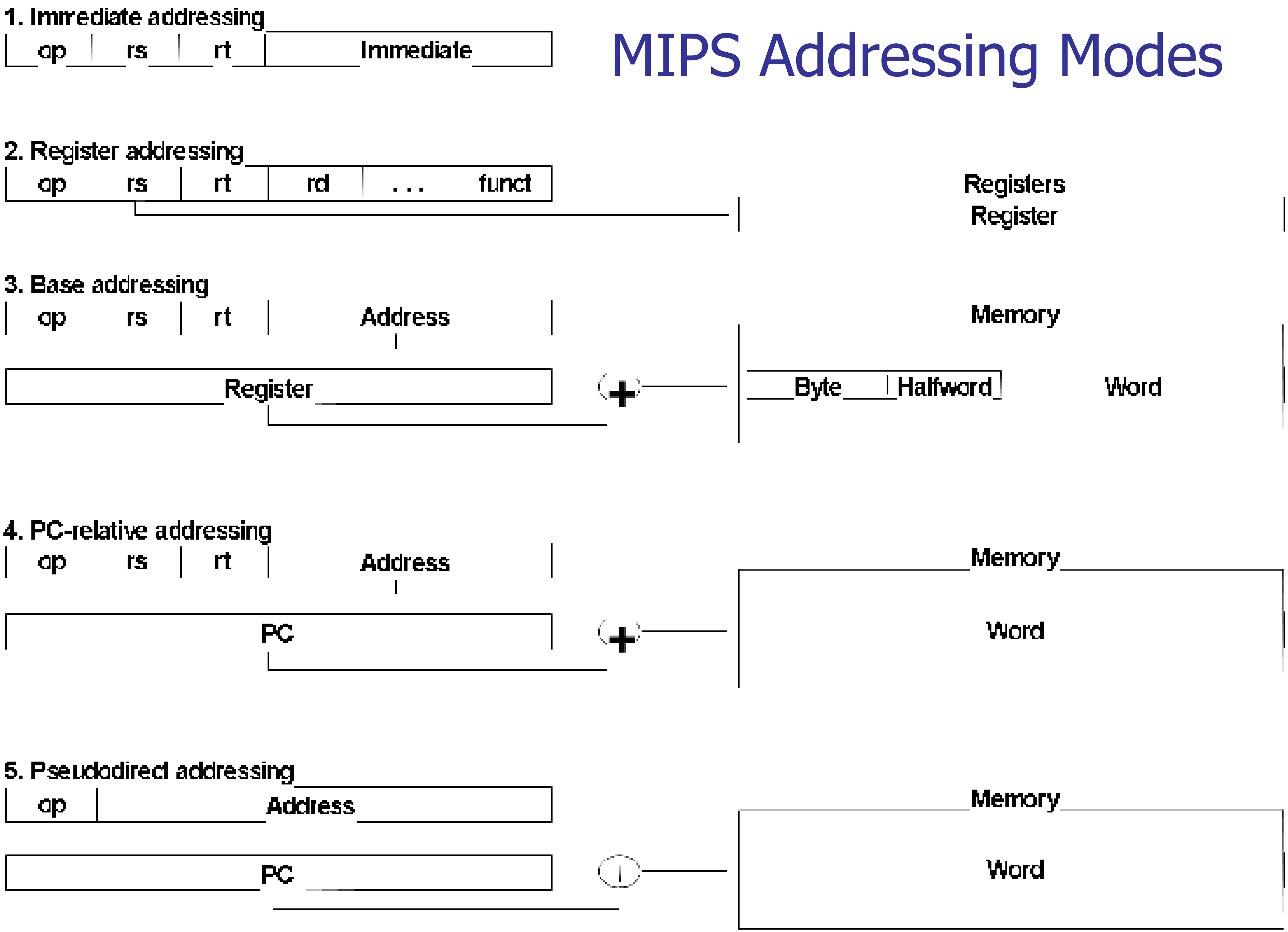
- Assembly provides convenient *symbolic representation*
  - much easier than writing down numbers
  - regular rules: e.g., destination first

- Machine language is the *underlying reality*
  - e.g., destination is no longer first

- Assembly can provide *pseudo-instructions*
  - e.g., `move $t0, $t1` exists only in assembly
  - would be implemented using `add $t0, $t1, $zero`

- When considering performance you should count actual number of machine instructions that will execute

MIPS Addressing Modes

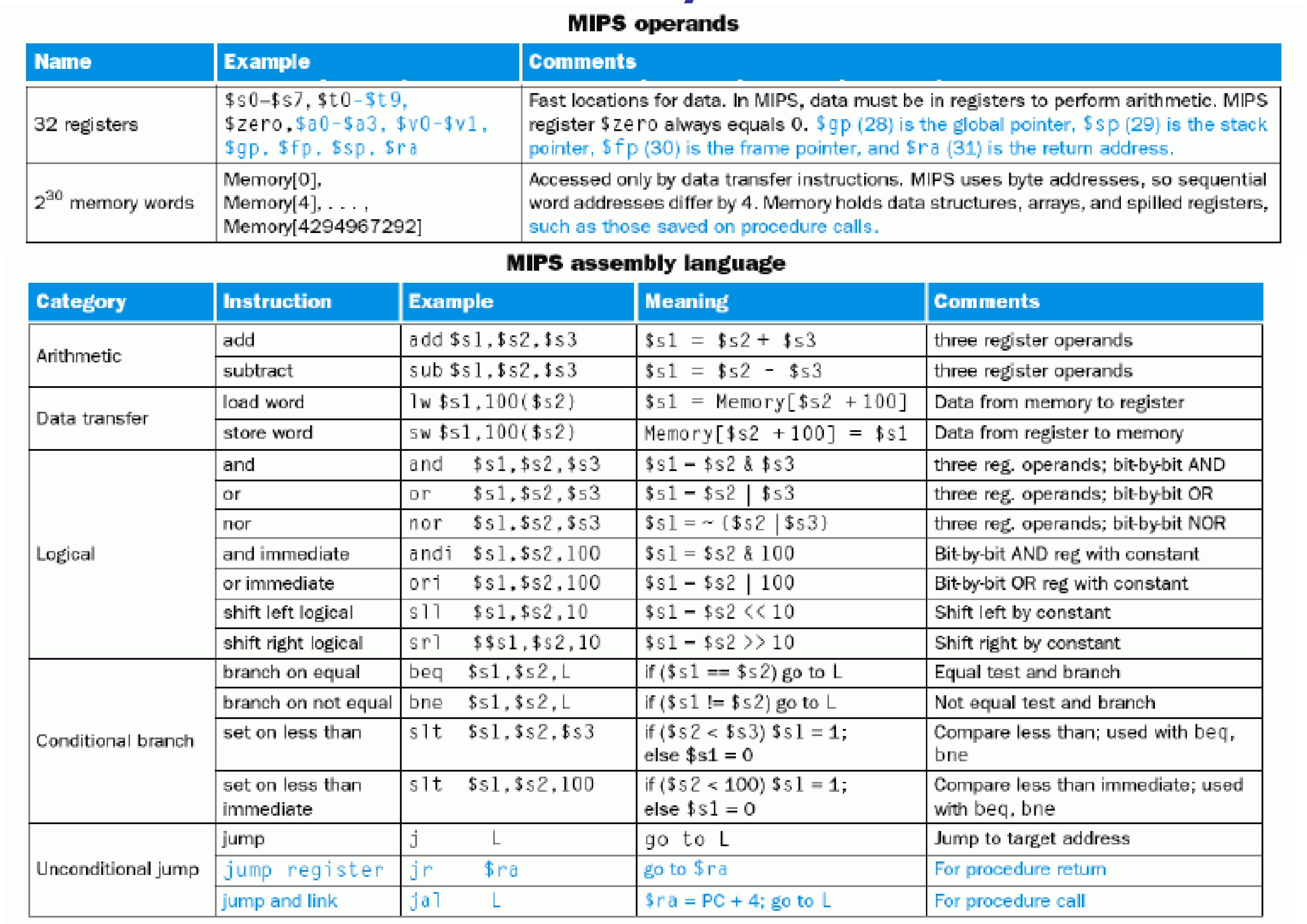


Overview of MIPS

- Simple instructions – all 32 bits wide
- Very structured – no unnecessary baggage
- Only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

MIPS Summary



MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp (29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, 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 register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	three register operands
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address
	jump register	jr $ra	go to $ra	For procedure return
	jump and link	jal L	$ra = PC + 4; go to L	For procedure call

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
- Sometimes referred to as *R(educed)ISC vs. C(omplex)ISC*
  - virtually all new instruction sets since 1982 have been RISC
- We'll look at PowerPC and 80x86