Instructions: Language of the Computer

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

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Compute first twelve Fibonacci numbers and put in array, then print

```
.data
fibs: .word   0 : 12        # "array" of 12 words to contain fib values
size: .word  12             # size of "array"
```

```
 .text
         la   $t0, fibs              # load address of array (Fib. source ptr)
         la   $t5, size              # load address of size variable
         lw   $t5, 0($t5)            # $t5 = 12
         li   $t2, 1                 # 1 is first and second Fib. number
         add.d $f0, $f2, $f4         # add.d – double precision add
         sw   $t2, 0($t0)            # F[0] = 1
         sw   $t2, 4($t0)            # F[1] = F[0] = 1
         addi $t1, $t5, -2           # Counter for loop, will execute (size-2) times
loop:    lw   $t3, 0($t0)             # Get value from array F[n]
         lw   $t4, 4($t0)            # Get value from array F[n+1]
         add  $t2, $t3, $t4          # $t2 = F[n] + F[n+1]
         sw   $t2, 8($t0)            # Store F[n+2] = F[n] + F[n+1] in array
         addi $t0, $t0, 4            # increment address of Fib. number source (ptr)
         addi $t1, $t1, -1           # decrement loop counter
         bgtz $t1, loop              # repeat if not finished yet.
         la   $a0, fibs               # first argument for print (array)
         add  $a1, $zero, $t5        # second argument for print (size)
         jal  print                  # call print routine.
         li   $v0, 10                # system call for exit
         syscall                     # we are out of here.
```

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

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.

So far

- Instruction / Format / Meaning

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**				

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c      # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

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

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
  - 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?*

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten

- Useful for common operation, e.g., move between registers

```
add $t2, $s1, $zero
```

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields:
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS Arithmetic

- Design Principle 1: *simplicity favors regularity.*

  Translation: *Regular instructions make for simple hardware*!

- *Simpler hardware reduces design time and manufacturing cost.*

- Of course this complicates some things...

C code:
```
A = B + C + D;
E = F - A;
```

MIPS code (arithmetic):
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

Allowing variable number of operands would simplify the assembly code but complicate the hardware.

- Performance penalty: high-level code translates to denser machine code.

MIPS Arithmetic

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

- Design Principle 2: *smaller is faster.* *Why?*

  - *Electronic signals have to travel further on a physically larger chip increasing clock cycle time.*
  - *Smaller is also cheaper!*

Simple MIPS instructions:

- 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
```

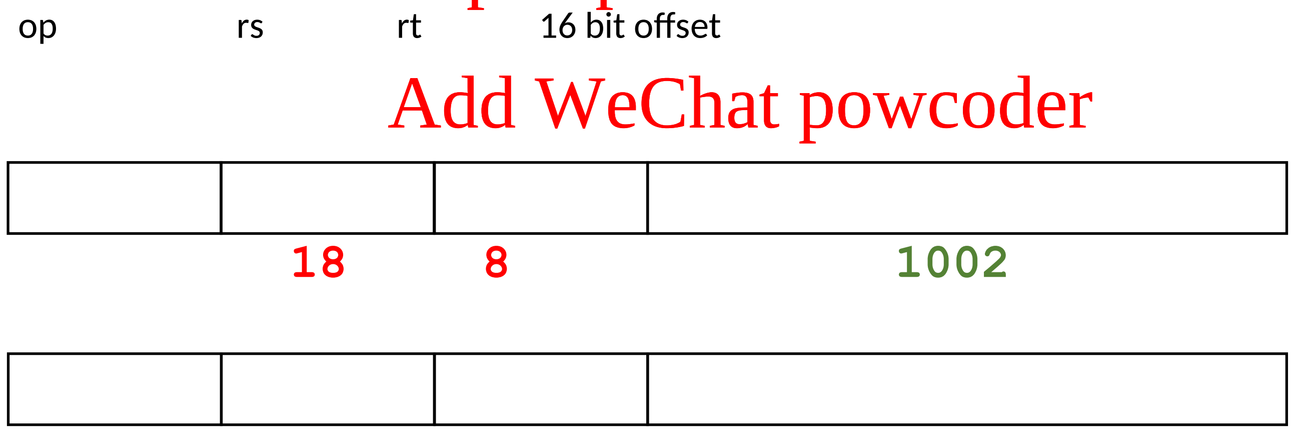
must contain an address

Machine Language

- 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)`   # $t0 → $8 $2 → $18

100011   10010   01000   0000001111101010



MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4*: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Machine Language

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

	17	18	8		
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
opcode – operation	first register source operand	second register source operand	register destin-ation operand	shift amount	function field - selects variant of operation

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

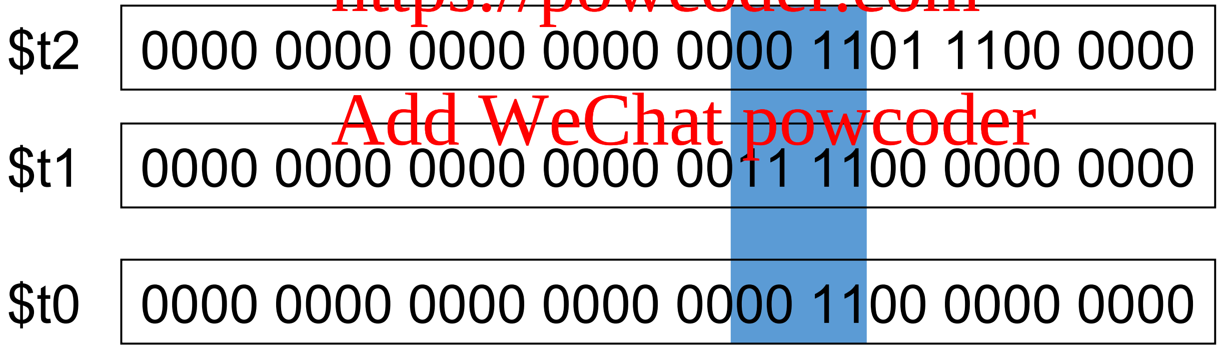
```
addi $s2, $s1, -1
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```



$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

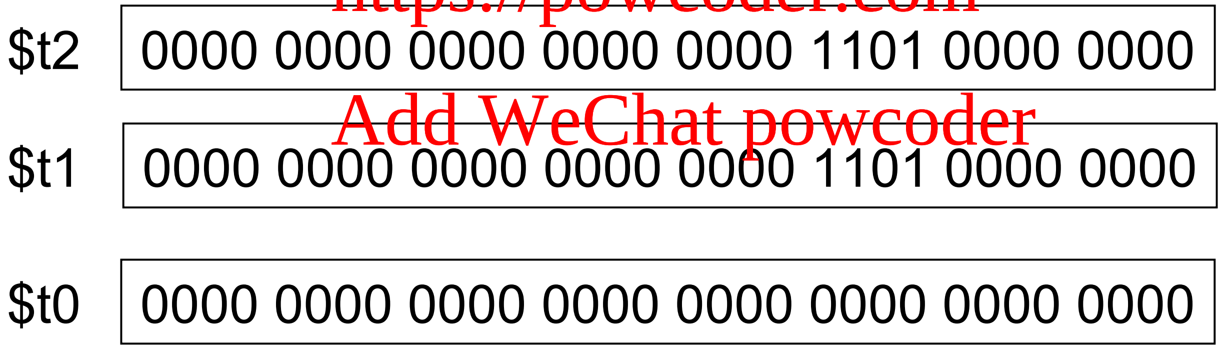
```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

XOR Operations

- Useful to check to see if two words have the same bits
  - Set some bits to 1, leave others unchanged

```
xor $t0, $t1, $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

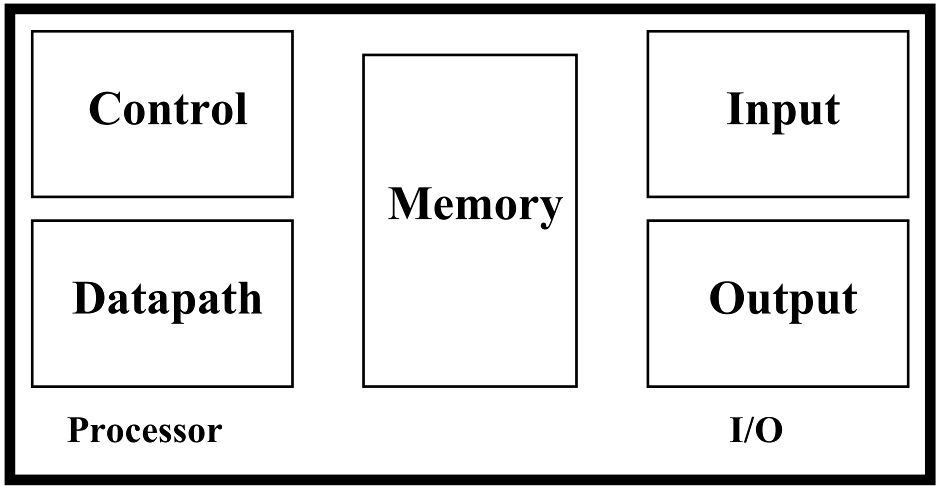
```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

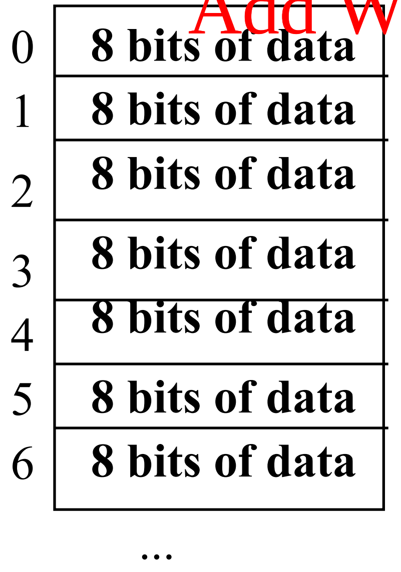
Registers vs. Memory

- 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*



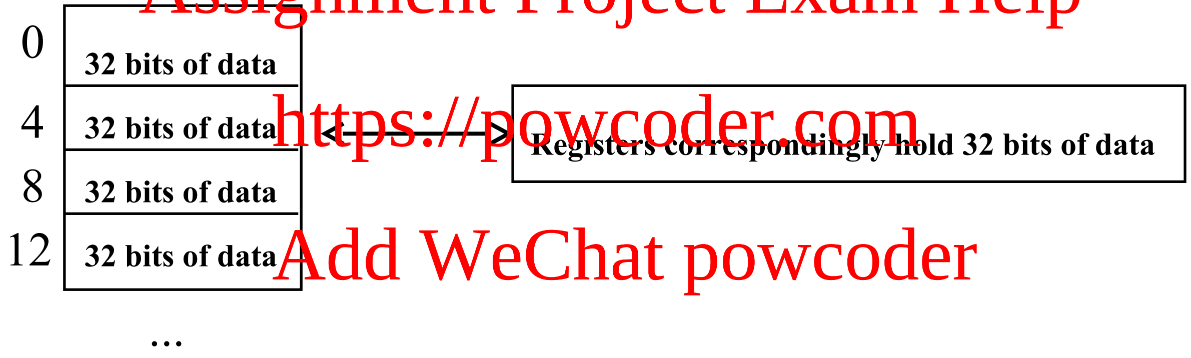
Memory Organization

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- *Byte addressing* means that the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte



Memory Organization

- Bytes are load/store units, but most data items use larger *words*
- For MIPS, a word is 32 bits or 4 bytes.



- 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];`

-

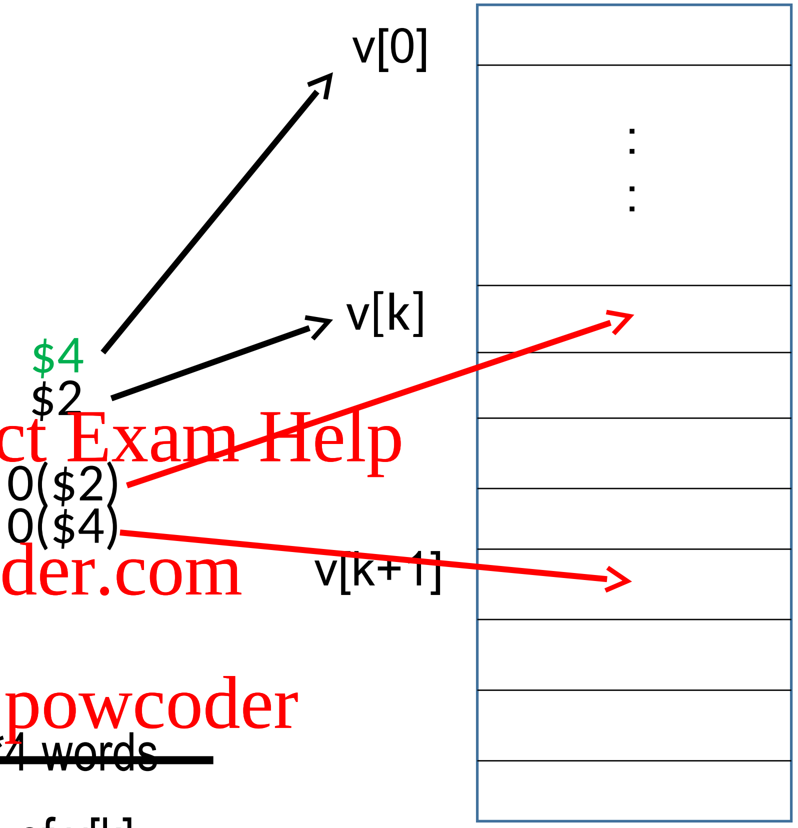
```
                              value      offset   address
MIPS code    (load):        lw   $t0,  32($s3)
             (arithmetic):  add  $t0,  $s2,  $t0
             (store):       sw   $t0,  32($s3)
```

- 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

A MIPS Example

- *Can we figure out the assembly code?*

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



```
swap:
   ~~muli $2,   $5,   4~~            ~~# index k*4 words~~
   sll  $2,   $5,   2
   add  $2,   $4,   $2            # address of v[k]
   lw   $15,  0($2)               # $15 = v[k]
   lw   $16,  4($2)               # $16 = v[k+1]
   sw   $16,  0($2)               # $16 → v[k]
   sw   $15,  4($2)               # $15 → v[k+1]

   jr   $31
```

#$4-$7 hold first 5 function arguments, i.e.,
$4 = v[ ] (address)
$5 = k

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

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

$\longleftrightarrow$ beq $t0, $t1, Label
(= addr.100) + PC

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

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

word-relative addressing:
25 words = 100 bytes

Addresses in Branch

- Further extend reach of branch by observing all MIPS instructions are a word (= 4 bytes), therefore *word-relative* addressing:
- MIPS branch destination address = $\underbrace{(PC + 4)}$ + (4 * offset)

Because hardware typically increments PC early in execute cycle to point to next instruction

  - so offset = (branch destination address – PC – 4)/4 → number of words offset
  - *MIPS does*: offset = (branch destination address – PC)/4

Control: Unconditional Branch (Jump)

- MIPS unconditional branch instructions:

```
j Label
```

- *Example*:

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

- **J-type** ("J" for Jump) instruction format
  - *Example*: `j Label` # 25 instructions away

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 (sll by 2)
  - *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

Branching Far Away

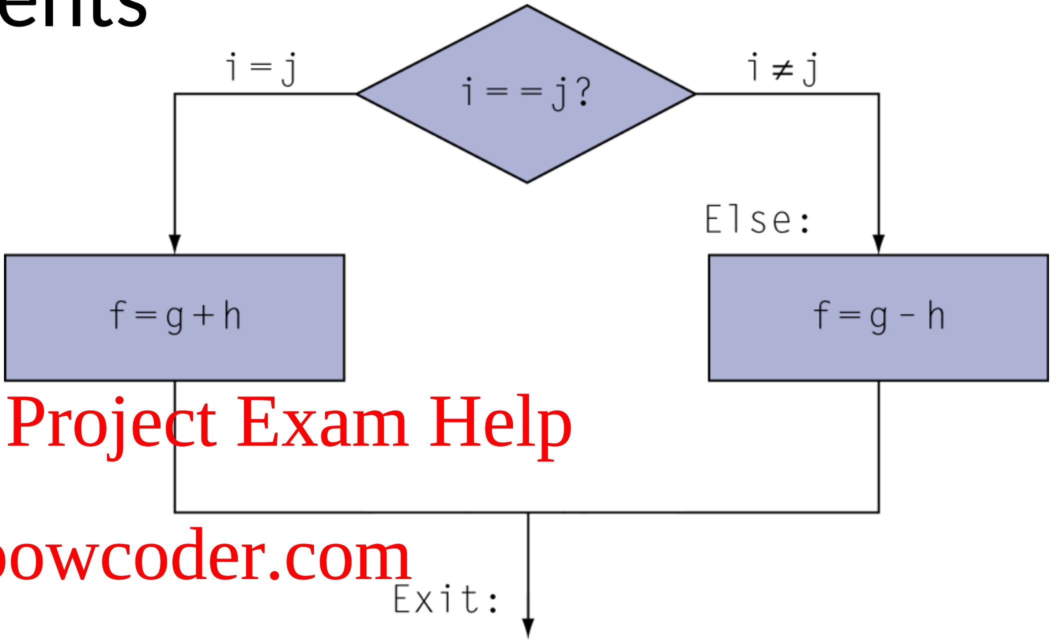
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
                beq $s0,$s1, L1
```

$$\downarrow$$

```
                bne $s0,$s1, L2
                j L1
L2:             …
```

Compiling `if` Statements



- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …

- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save[0] in $s6

- Compiled MIPS code:

```
Loop: sll   $t1, $s3, 2        # shift left logical 2 bits:
                                 # or t1=i*4
      add   $t1, $t1, $s6      # t1 = addr(save[i])
                               #      = addr(save[0]) + 4i
      lw    $t0, 0($t1)        # t0 =save[i] →next element

       bne  $t0, $s5, Exit # if (save[i] != k) exit
      addi $s3, $s3, 1  #  i += 1
      j     Loop
Exit: …
```

Target Addressing Example

Assume Loop at location 80000 (i) $s3 sll 2 (t1)

```
              0000              0000  (0)
while (save[i] == k) i += 1;              0001    0100  (4)
              0010    1000  (8)
  i        k    save[0] addr              0011    1100  (12)
```

Loop: sll $t1, $s3, 2	80000	0	0	19	9	4	0
add $t1, $t1, $s6	80004	0	9	22	9	0	32
lw $t0, 0($t1)	80008	35	9	8	0		
bne $t0, $s5, Exit	80012	5	8	21	2 (PC+i*4)		
addi $s3, $s3, 1	80016	8	19	19	1 PC		
j Loop	80020	2	20000 (x 4)				
Exit: …	80024						

- When bne instruction runs, **PC is already updated by 4**
- When executing bne command: PC = addr(bne) = 80012 + 4 = 80016 = next cmd
- Addr(Exit) = two hops from PC = 80016 + (2 * 4 (bytes per word)) = 80024

More Conditional Operations

- Set result to 1 if a condition is true
  Otherwise, set to 0

- `slt rd, rs, rt`
  if (rs < rt) rd = 1; else rd = 0;

- `slti rt, rs, constant`
  if (rs < constant) rt = 1; else rt = 0;

blt, bne: pseudo-instructions that can be executed as:

```
    slt $t0, $s1, $s2  # if ($s1 < $s2)
```

Why are they not included in the ISA?

Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Signed vs. Unsigned for `slt`

`Slt`: set on less than

```
slt  $t0, $s0, $s1
```
# if $s0 < $s1, then $t0 = 1

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

```
slt  $t0, $s0, $s1
```
# signed

–1 < +1 ⇒ $t0 = 1

```
sltu $t0, $s0, $s1
```
# unsigned

+4,294,967,295 > +1 ⇒ $t0 = 0

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise XOR	^	^	`xor`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift

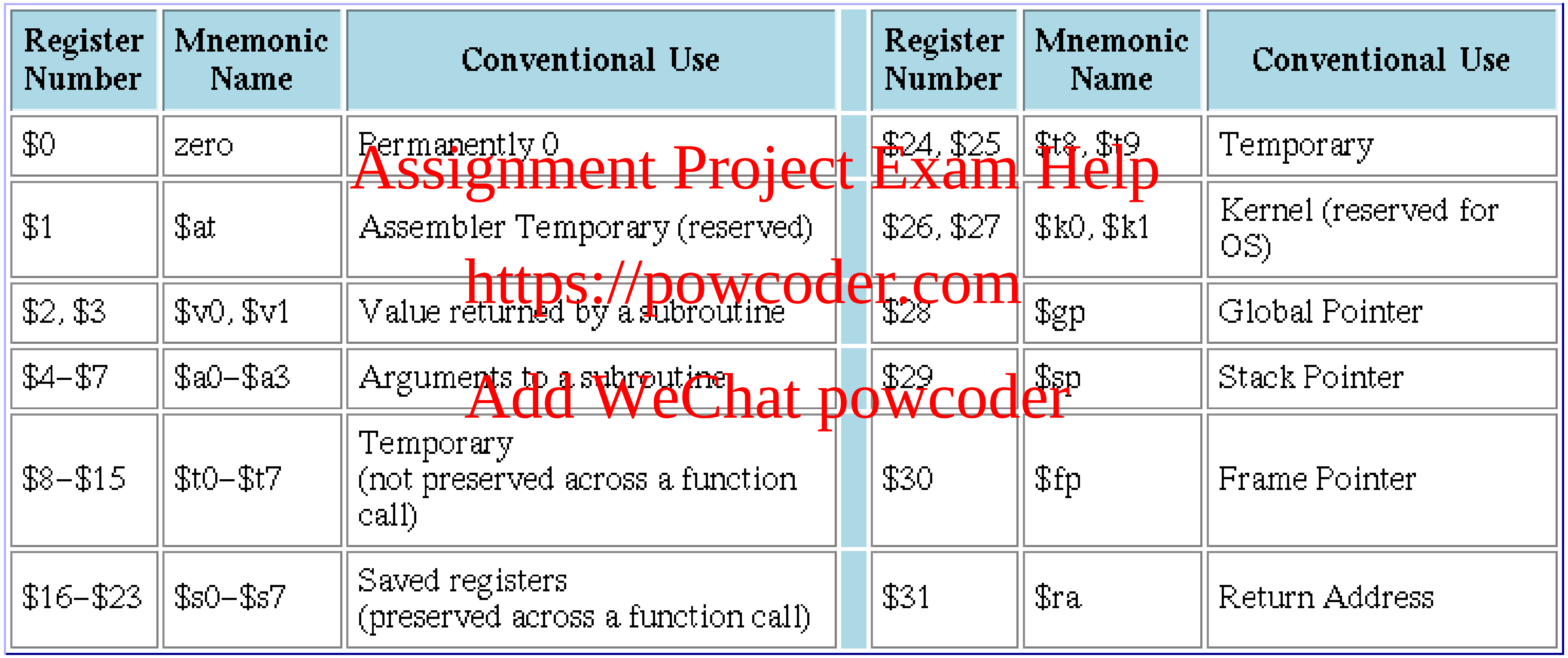
- Shift left logical
  - Shift left and fill with 0 bits

  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

MIPS assembly language

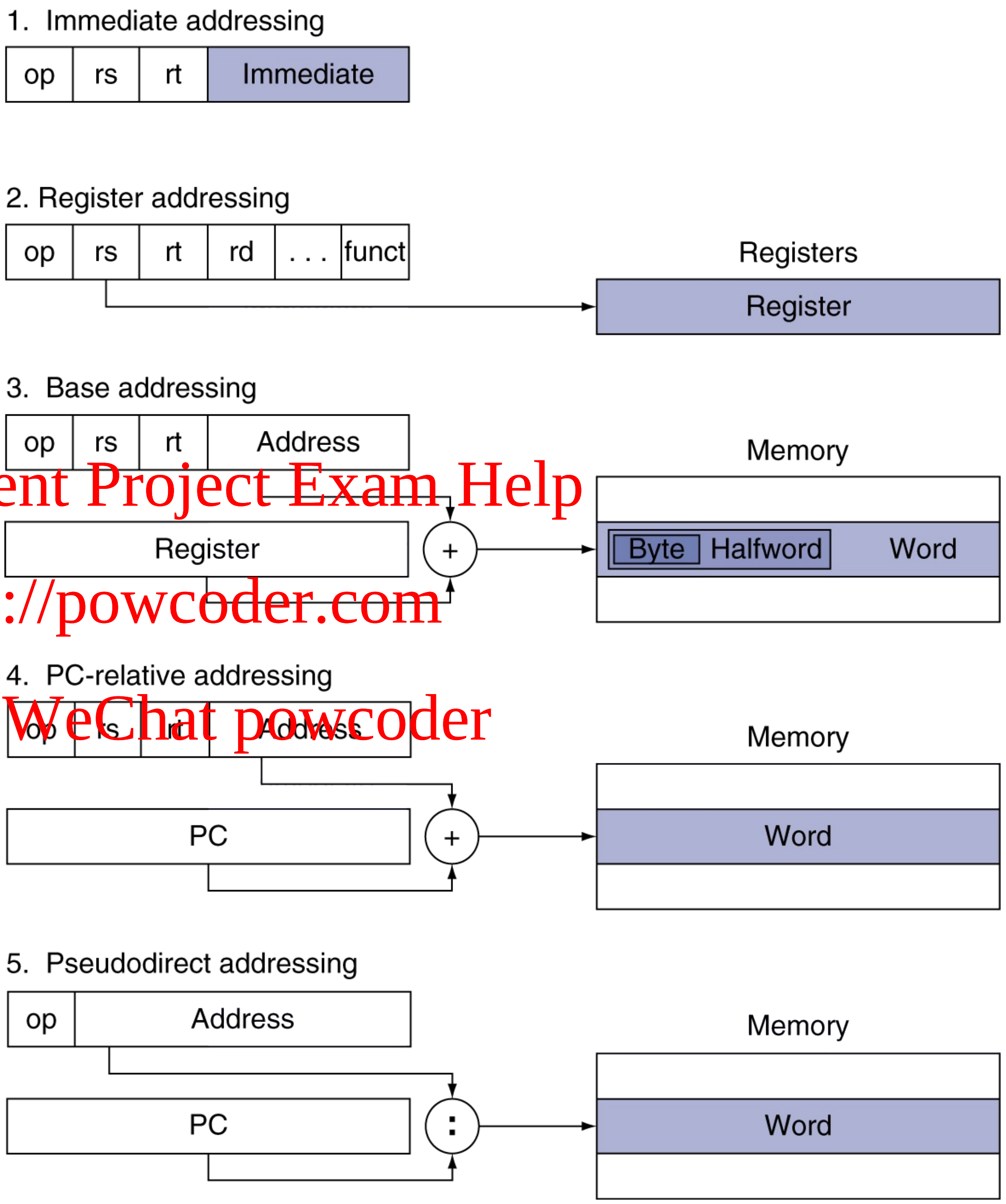
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	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

Register Usage



Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0	$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)	$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine	$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine	$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)	$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)	$31	$ra	Return Address

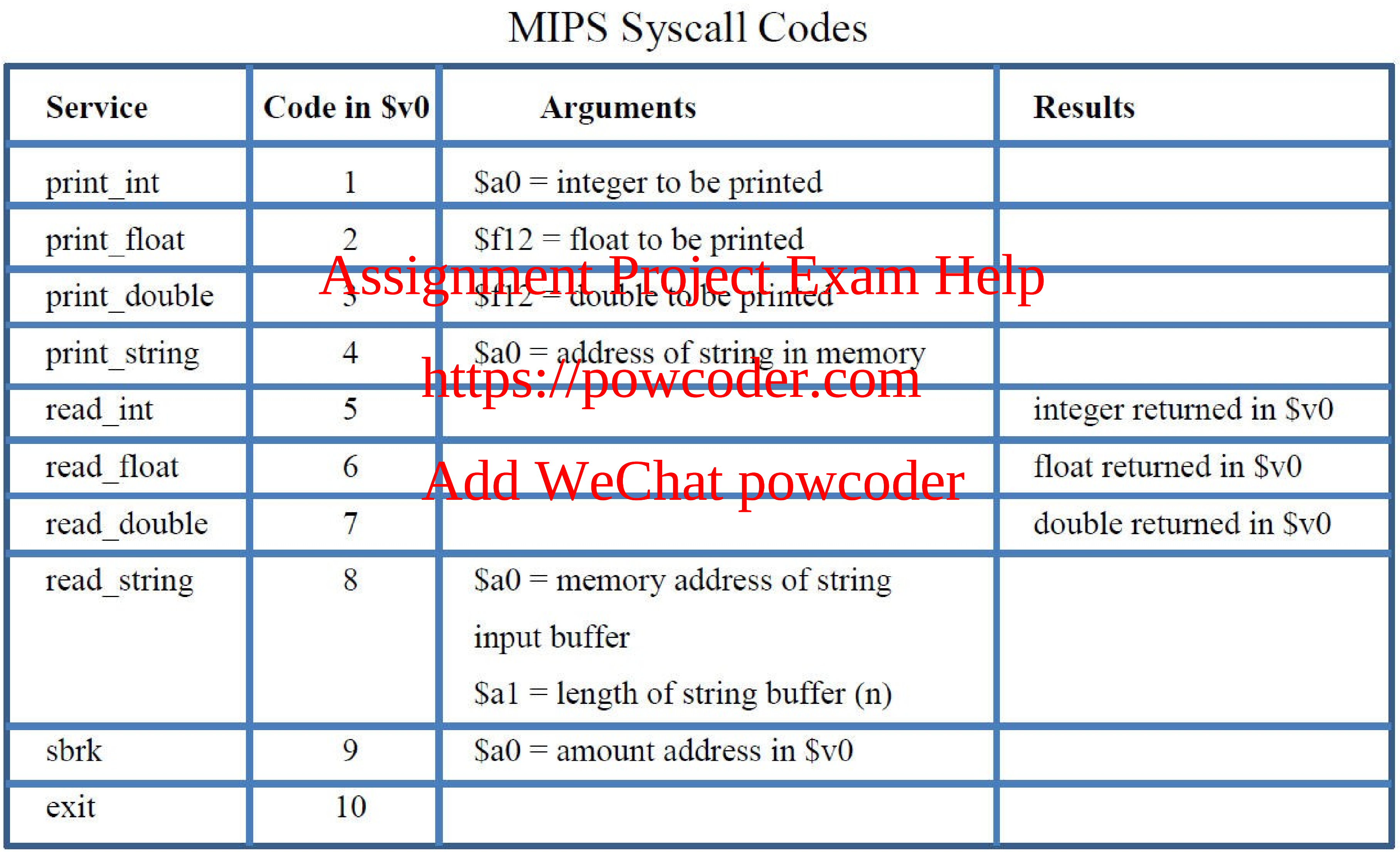
Addressing Mode Summary



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

I/O Service Codes



MIPS Syscall Codes

Service	Code in $v0	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount address in $v0	
exit	10		

Input/Output

A number of system services, mainly for input and output, are available for use by your MIPS program.

SYSCALL System Services

1. Load the service number in register $v0.
2. Load argument values, if any, in $a0, $a1, $a2, or $f12 as specified.
3. Issue the SYSCALL instruction.
4. Retrieve return values, if any, from result registers as specified.

E.g.,

Print value in $t0 to console

```
li   $v0, 1                # service 1 is print integer
add $a0, $t0, $zero        # load desired value into argument register
                           # $a0, using pseudo-op
syscall
```

I/O Example

Print double value in $t1 to console

```
li $v0, 3                  # service 3 is print double
add $f12, $t1, $zero       # load desired value into argument register
                           # $a0, using pseudo-op
syscall
```

```
.data
getInput: .asciiz "Input a value: "

.text

li $v0,4                      #  Specifies the print string service
la $a0,getInput               #  loads the start string
syscall                       #  displays the string

li $v0,5                      #  specifies the read integer service
syscall                       #  starts the read int service

add $a0,$v0,$0                #  $a0  = value read from read service
li $v0,1                      #  specifies the print integer service
syscall

li $v0,10                     #  specifies the exit service
syscall                       #  exits the program
```

Compute first twelve Fibonacci numbers and put in array, then print

```
.data
fibs: .word   0 : 12       # "array" of 12 words to contain fib values
size: .word  12            # size of "array"
```

```
 .text
         la   $t0, fibs             # load address of array (Fib. source ptr)
         la   $t5, size             # load address of size variable
         lw   $t5, 0($t5)           # load array size
         li   $t2, 1                # 1 is first and second Fib. number
         add.d $f0, $f2, $f4        # add.d – double precision add
         sw   $t2, 0($t0)           # F[0] = 1
         sw   $t2, 4($t0)           # F[1] = F[0] = 1
         addi $t1, $t5, -2          # Counter for loop, will execute (size-2) times
loop:    lw   $t3, 0($t0)            # Get value from array F[n]
         lw   $t4, 4($t0)           # Get value from array F[n+1]
         add  $t2, $t3, $t4         # $t2 = F[n] + F[n+1]
         sw   $t2, 8($t0)           # Store F[n+2] = F[n] + F[n+1] in array
         addi $t0, $t0, 4           # increment address of Fib. number source (ptr)
         addi $t1, $t1, -1          # decrement loop counter
         bgtz $t1, loop             # repeat if not finished yet.
         la   $a0, fibs              # first argument for print (array)
         add  $a1, $zero, $t5       # second argument for print (size)
         jal  print                 # call print routine.
         li   $v0, 10               # system call for exit
         syscall                    # we are out of here.
```

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

- *Example C code*:

// procedure adds 10 to input parameter

```
int main(){
 int i, j;
 i = 5;
 j = add10(i);
 i = j;
 return 0;
}

int add10(int i){
return (i + 10);
}
```

- `$a0` – *contains argument of function*
- `$v0` - *contains return value of function*

```
.text
main:
 addi $s0, $0, 5 # i=5
 add  $a0, $s0, $0 # arg =5

 jal add10

 add $s1, $v0, $0 # j=retValue
 add $s0, $s1, $0 # i=j

 li  $v0, 10
 syscall
```

argument to function

Call func

control returns here after call

system code & call to exit

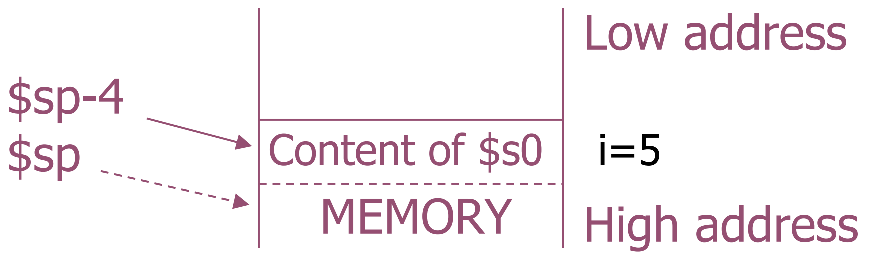
```
add10:
    addi $sp, $sp, -4
    sw $s0,0($sp) #i=5 to memory
    addi $s0,$a0,10 #$s0=15
    add $v0, $s0, $0 # i=15

    lw $s0, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

save register in stack, see figure below

return result to caller

restore $s0 to 5

return



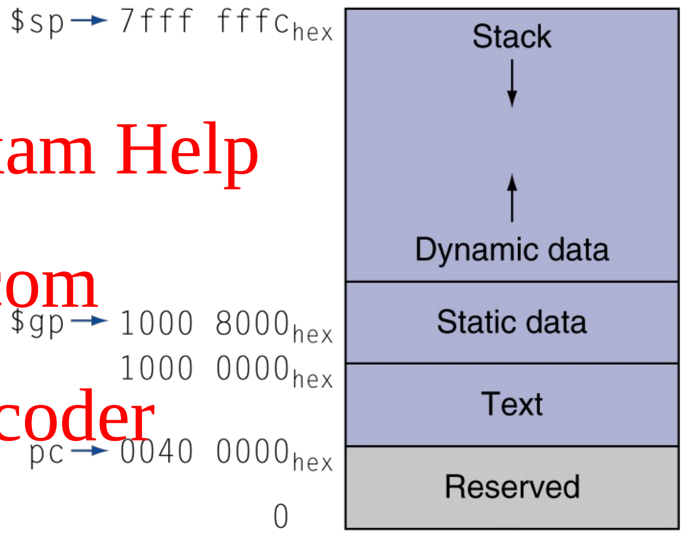
MIPS: Software Conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from function call
3	v1	returned to caller
4	a0	arguments to function
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	

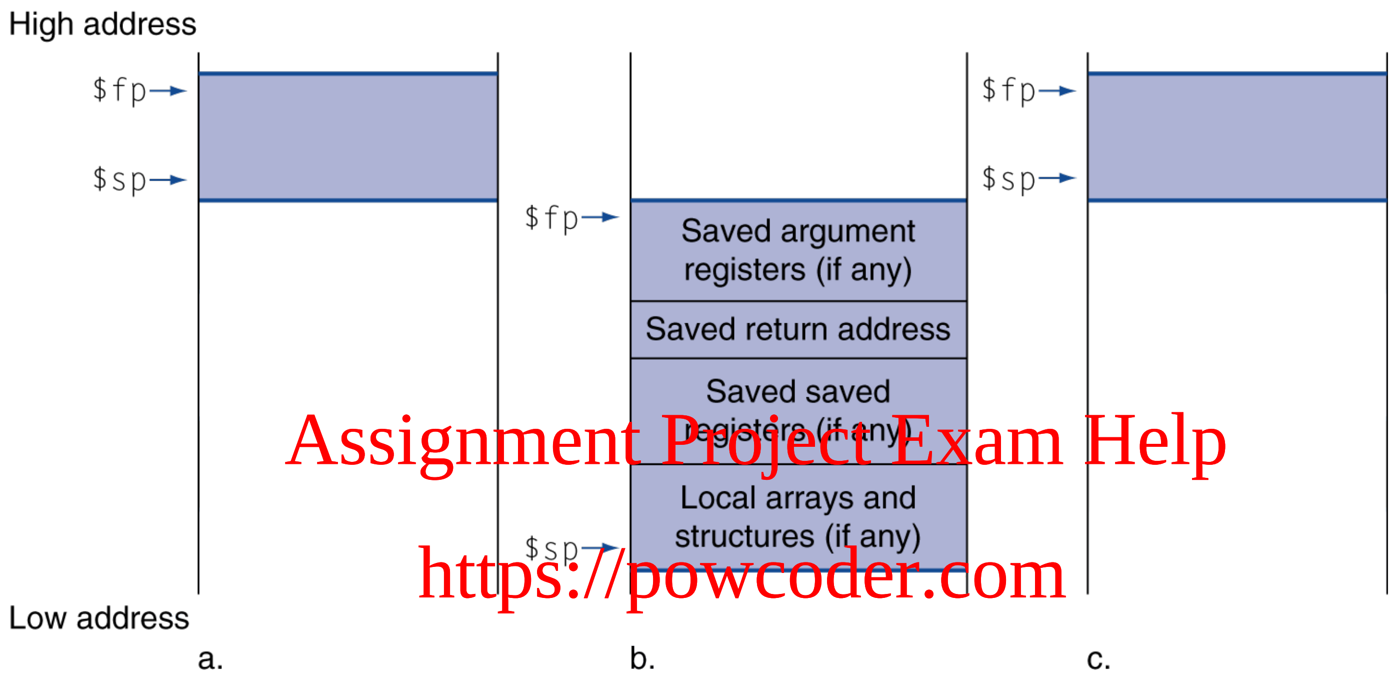
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address: caller saves

Memory Layout

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



Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- Variables that are local to a procedure but do not fit into registers (e.g., local arrays, structures, etc.) are also stored in the stack. This area of the stack is the *frame*. The *frame pointer* $fp points to the top of the frame and the stack pointer to the bottom. $fp does not change during procedure execution, unlike the stack pointer, so it is a stable base register from which to compute offsets to local variables.
- Use of the frame pointer is *optional*. If there are no local variables to store in the stack it is not efficient to use a frame pointer.

Procedures (recursive)

```
int main(){
 int i;
 i = 7;
 j = fact(i);
 return 0;
}
```

```
int fact(int n){
    if (n <= 1) return (1);
    else return (n*fact(n-1));
}
```

- *Translated MIPS assembly*:

```
.text
.globl main

main:
    addi $a0, $0, 7
    jal fact
    nop
```

n = 7

control returns from fact

```
    move $a0, $v0
    li $v0, 1
    syscall
```

print value returned by fact

```
    li  $v0, 10
    syscall
```

exit

```
fact:
    addi $sp, $sp, -8
    sw $ra, 4($sp)
    sw $a0, 0($sp)
```

stack return address and argument

branch to GT0 if n>=1

```
    slti $t0, $a0, 1
    beq $t0, $0, NGE1
    nop
```

if n < 1 return 1

```
    addi $v0, $0, 1
    addi $sp, $sp, 8
    jr $ra
```

```
NGE1:
    addi $a0, $a0, -1
    jal fact
    nop
```

if n>=1 call fact(n-1)

restore return address, argument, and stack pointer

```
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
```

return n*fact(n-1)

```
    mul $v0, $a0, $v0
```

return control

```
    jr $ra
```

Character Data Set

- Byte-encoded character sets
    - ASCII: 128 characters (7 bits)
        - 95 graphic, 33 control
    - Latin-1: 256 characters
        - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
    - Used in Java, C++ wide characters, …
    - Most of the world’s alphabets, plus symbols
    - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)       lh rt, offset(rs)
```

- Sign extend to 32 bits in rt

```
lbu rt, offset(rs)      lhu rt, offset(rs)
```

- Zero extend to 32 bits in rt

```
sb rt, offset(rs)       sh rt, offset(rs)
```

- Store just rightmost byte/halfword

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Race Condition

- count++ could be implemented as

```
register1 = count              // lw $1,0(count)
register1 = register1 + 1      // addi  $1, $1, 1
count = register1              // sw $1, 0(count)
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
step 0: process A execute register1 = count   {register1 = 5}
step 1: process A execute register1 = register1 + 1   {register1 = 6}
 // clock interrupt – context switched (before value is updated to memory)
step 2: process B execute register2 = count   {register2 = 5}
step 3: process B execute register2 = register2 - 1   {register2 = 4}
 // clock interrupt – context switched
step 4: process A execute count = register1   {count = 6 }
 // clock interrupt – context switched
step 5: process B execute count = register2   {count = 4}
```

Atomic Operations in MIPS

Used together to implement lock-free atomic read-modify-write operation:

- Load linked: `ll rt, offset(rs)`
  - Returns current value at memory indicated in offset(rs)

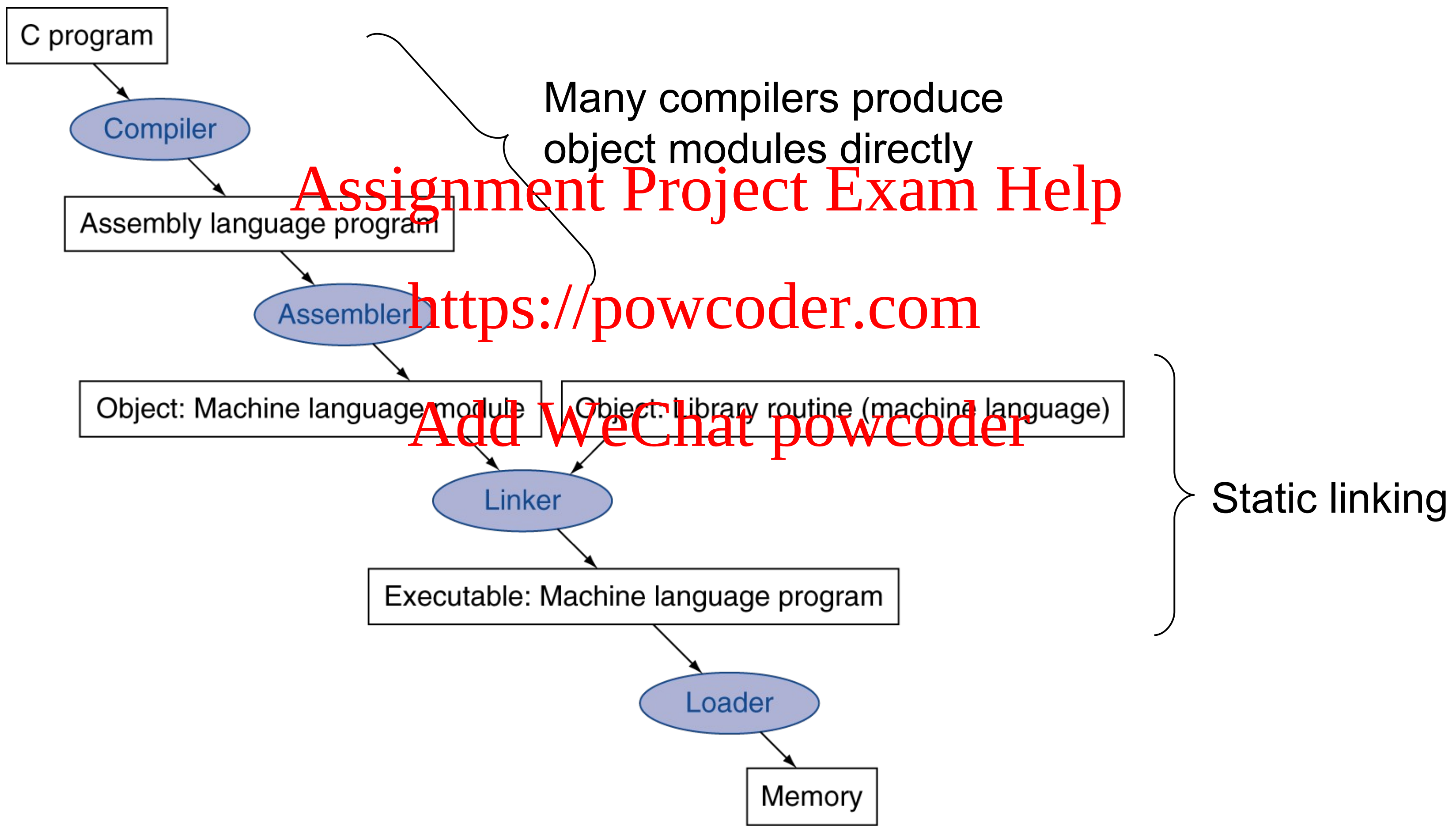
- Store conditional: `sc $s1, offset(rs)`
  - Stores value to offset(rs) location if memory location has not been updated since load-link command. Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt

Synchronization in MIPS

- Example: atomic swap (to test/set lock variable)

```
                                      # swap  $s4 ← → $s1
try: add $t0,$s4,$zero                # copy exchange value
     ll  $t1,0($s1)                   # load linked
     sc  $t0,0($s1)                   # store conditional
     beq $t0,$zero,try                # branch store fails
     add $s4,$t1,$zero                # put load value in $s4
```

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Assembler Pseudoinstructions

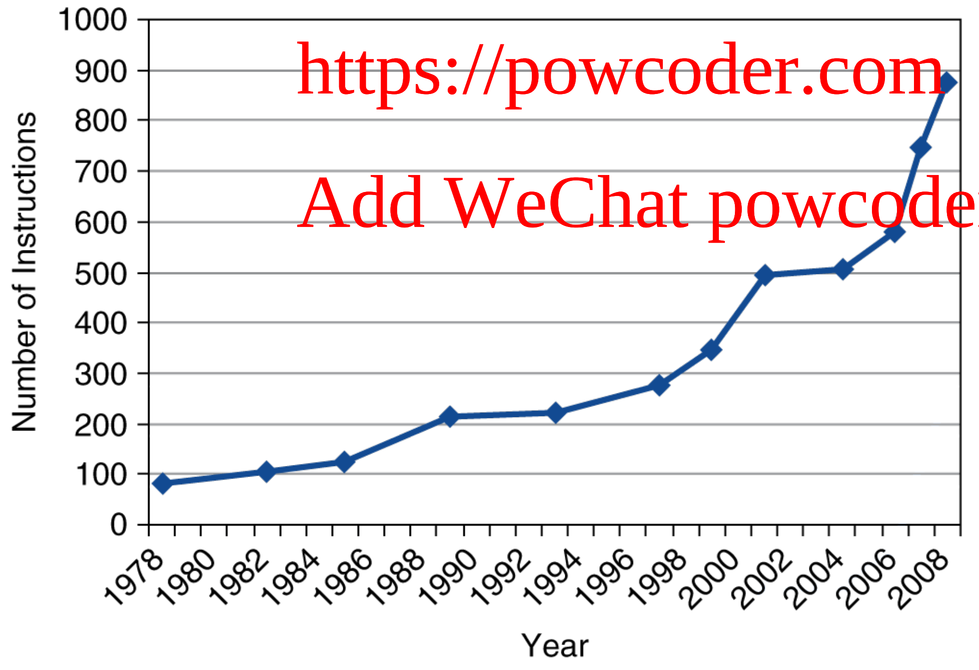
- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1       →  add $t0, $zero, $t1
blt $t0, $t1, L     →  slt $at, $t0, $t1
                       bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do increase gradually.



x86 instruction set

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86