Chapter 2

Instructions:
Language of the Computer

Instructions:

- **Language of the Machine**
  - **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**

- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **Almost 100 million MIPS processors manufactured in 2002**
  - **used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …**

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `a = b + c`

MIPS 'code': `add a, b, c`

(we'll talk about registers in a bit)

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

MIPS arithmetic

- **Design Principle: simplicity favors regularity.**
- **Of course this complicates some things...**

C code: `a = b + c + d;`

MIPS code:

```
add a, b, c
add a, a, d
```

Example (p.79)

Example (p.79)

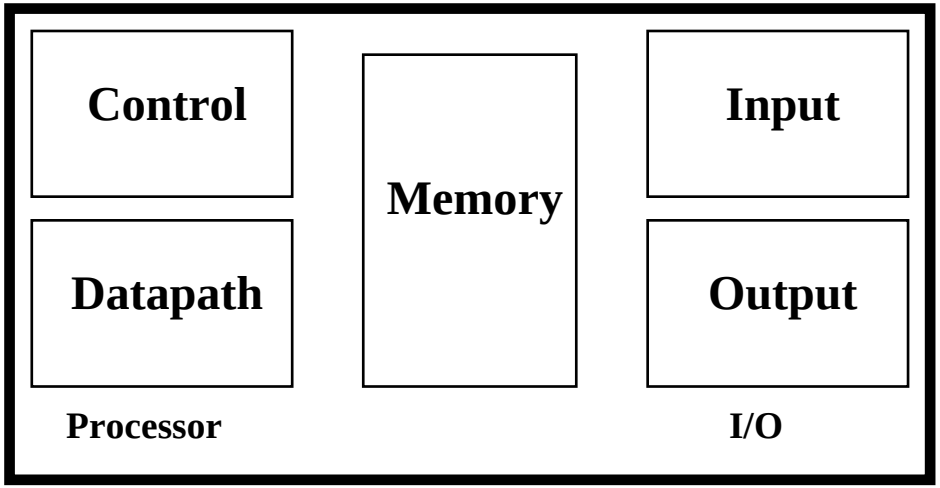
- **Operands must be registers, only 32 registers provided**
- **Each register contains 32 bits**

Example (p.81)

- **Design Principle: smaller is faster.**

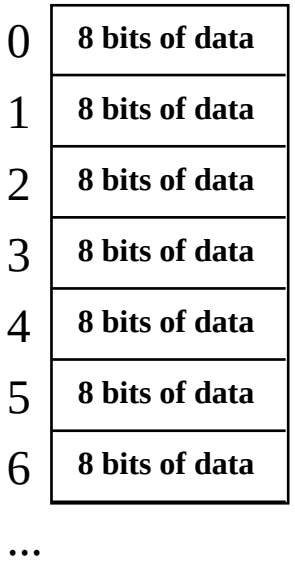
Registers vs. Memory

- **Arithmetic instructions operands must be registers,**
  — only 32 registers provided
- **Compiler associates variables with registers**
- **What about programs with lots of variables**



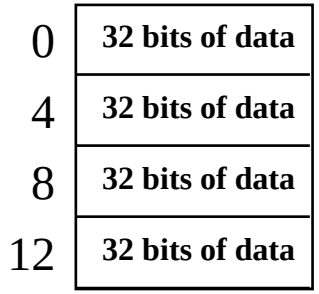
Memory Organization

- **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.**



Memory Organization

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



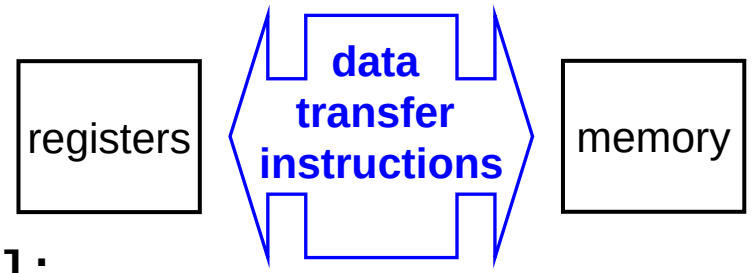
Registers hold 32 bits of data

- **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?

Instructions

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



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

offset *base address*

MIPS code:

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

load word

store word

- **Can refer to registers by name (e.g., $s2, $t2) instead of number**
- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

Can't write: `add 48($s3), $s2, 32($s3)`

Registers vs. Memory

- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - **More instructions to be executed**
- **Compiler must use registers for variables as much as possible**
  - **Only spill to memory for less frequently used variables**
  - **Register optimization is important!**

Constants or Immediate Operands

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

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

- **Solutions? Why not?**
  - **put 'typical constants' in memory and load them.**
  - **create hard-wired registers (like $zero) for constants like one.**

- **MIPS Instructions:**

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

add immediate

slt immediate

and immediate

or immediate

- **Design Principle: Make the common case fast. *Which format?***

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**
  - **— immediate operands**

- **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
addi $29, $29, 4        $29 = $29 + 4
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - Range: 0 to $+2^n - 1$
  - Example
    - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
      $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
      $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
  - Using 32 bits
    - 0 to +4,294,967,295
  - Most significant bit and least significant bit
  - overflow

Possible Representations of Signed Integers

Sign and Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

MIPS

- **32 bit signed numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Signed Integers

- **Given an n-bit number**

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - **Range: -2^{n-1} to $+2^{n-1} - 1$**

  - **Example**
    - **$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$**
      $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
      $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

  - **Using 32 bits**
    - **–2,147,483,648 to +2,147,483,647**

Two's Complement Signed Integers

- **Bit 31 is sign bit**
  - **1 for negative numbers**
  - **0 for non-negative numbers**
- **$-(-2^{n-1})$ can't be represented**
- **Non-negative numbers have the same unsigned and two's complement representation**
- **Some specific numbers**
  - **0: 0000 0000 … 0000**
  - **–1: 1111 1111 … 1111**
  - **Most-negative: 1000 0000 … 0000**
  - **Most-positive: 0111 1111 … 1111**

Signed Negation

- **Complement and add 1**
  - **Complement means 1 → 0, 0 → 1**

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- **Example: negate +2**
  - **+2 = 0000 0000 … 0010_2**
  - **–2 = 1111 1111 … 1101_2 + 1**
    = 1111 1111 … 1110_2

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - Signed values:
    - Positive: extend with 0s
    - Negative: extend with 1s
  - Unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Machine Language

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example: `add $t1, $s1, $s2`**
  - **registers have numbers, `$t1=9, $s1=17, $s2=18`**

- **Instruction Format:**

000000	10001	10010	01001	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

- ***Can you guess what the field names stand for?***

op: Basic operation of the instruction (opcode)

rs: The first register source operand

rt: The second register source operand

rd: The register destination operand

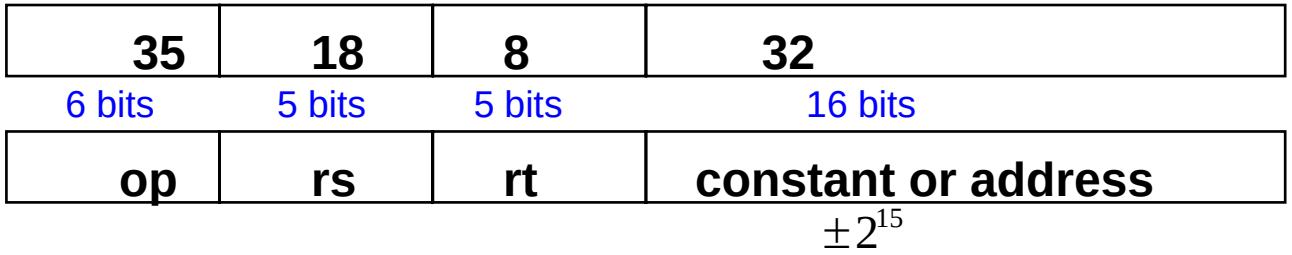
shamt: Shift amount

funct: Function (function code)

Hexadecimal-binary conversion (p.96)

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`



- Where's the compromise?

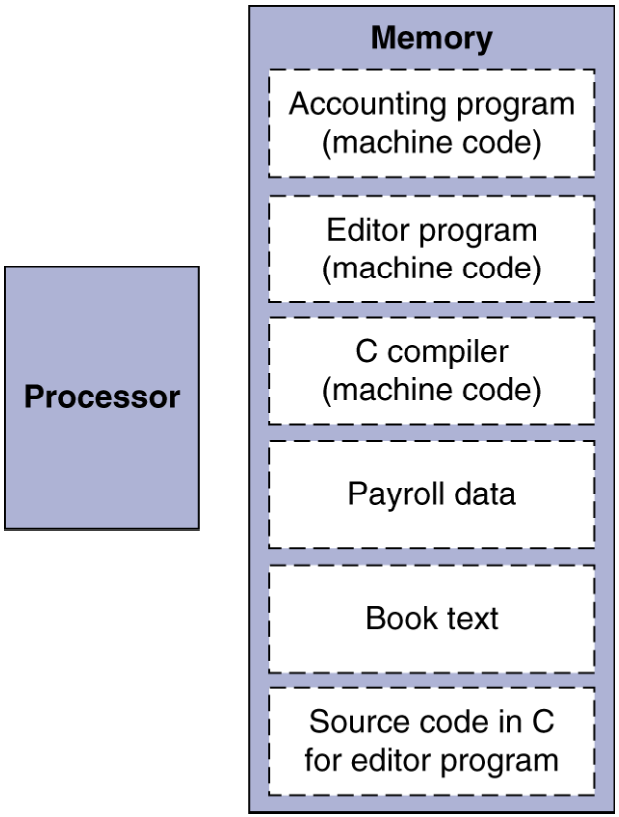
MIPS Instruction Encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub	R	0	reg	reg	reg	0	34_{ten}	n.a.
addi	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Example (p.98)

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

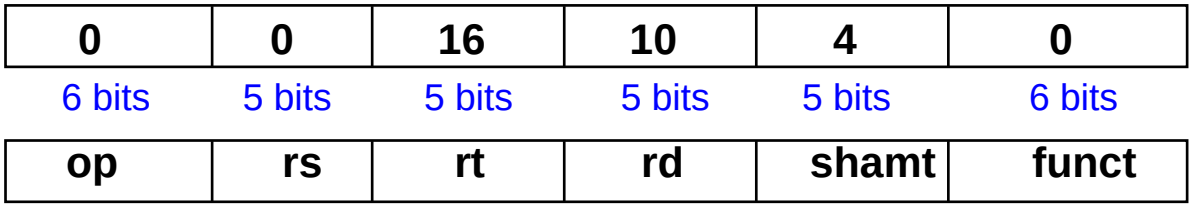
Logical Operations

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Register $s0: 0000 0000 0000 0000 0000 0000 0000 $1001_{two} = 9_{ten}$

sll $t2, $s0, 4 *shift left logical*

Register $t2: 0000 0000 0000 0000 0000 0000 1001 $0000_{two} = 144_{ten}$



Shift left by *i* bits = multiply by 2^i

Logical Operations

Register $t1: 0000 0000 0000 0000 0011 1100 0000 0000$_{two}$
Register $t2: 0000 0000 0000 0000 0000 1101 0000 0000$_{two}$
Register $t3: 0000 0000 0000 0000 0000 0000 0000 0000$_{two}$

- and $t0, $t1, $t2

Register $t0: 0000 0000 0000 0000 0000 1100 0000 0000$_{two}$

- or $t0, $t1, $t2

Register $t0: 0000 0000 0000 0000 0011 1101 0000 0000$_{two}$

NOT OR

- A NOR 0 = NOT(A OR 0) = NOT(A)
- nor $t0, $t1, $t3

Register $t0: 1111 1111 1111 1111 1100 0011 1111 1111$_{two}$

Control

- **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
```

branch if not equal

branch if equal

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

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

Control

- **MIPS unconditional branch instructions:**

```
j  label
```

jump

- **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:  ...
```

- ***Can you build a simple for loop?***

Example (p.107)

- ***How about a Case/Switch statement?***
  - ***If-then-else***
  - ***Jump address table & jump register instruction (jr)***

Control Flow

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

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
   $t0 = 1
else
   $t0 = 0
```

set on less than

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers
- Immediate version

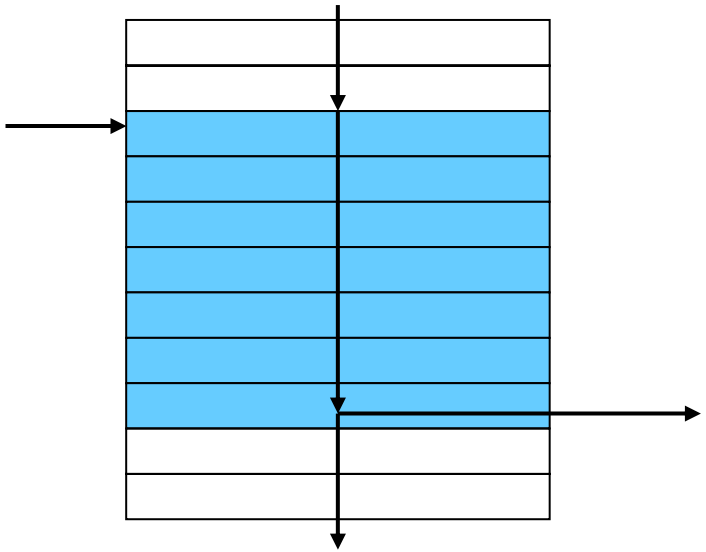
```
slti $t0, $s2, 10          $t0 = 1 if $s2 < 10
```

- Compare with unsigned numbers
  - `sltu` ***(set on less than unsigned)***
  - `sltiu` ***(set on less than immediate unsigned)***

Example (p.110)

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



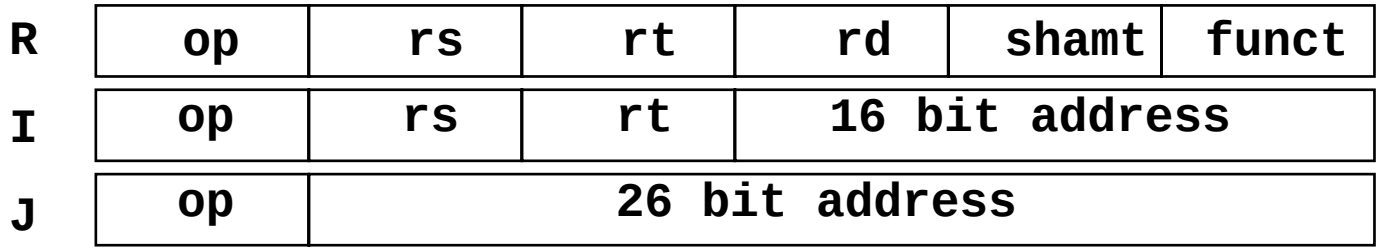
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

So far:

- **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
bne $s4,$s5,L     Next instr. is at Label if $s4 ≠ $s5
beq $s4,$s5,L     Next instr. is at Label if $s4 = $s5
j Label           Next instr. is at Label
slt $t0,$s1,$s2   $t0 = ($s1<$s2)
```

- **Formats:**



Policy of Use Conventions

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 ($at) reserved for assembler, 26-27 for operating system

Supporting Procedures

- **Execution of a procedure:**
  - **Put parameters in a place where the procedure can access them**
  - **Transfer control to the procedure**
  - **Acquire the storage resources needed for the procedure**
  - **Perform the desired task**
  - **Place the result value in a place where the calling program can access it**
  - **Return control to the point of origin, since a procedure can be called from several points in a program**

- **Allocated registers:**
  - **`$a0-$a3`: four argument registers in which to pass parameters**
  - **`$v0-$v1`: two value registers in which to return values**
  - **`$ra`: one return address register to return to the point of origin**

Supporting Procedures

- *Jump-and-link* instruction (`jal`)
  - Jump to an address and simultaneously save the address of the following instruction in register `$ra`
  - `jal ProcedureAddress`

- *Program counter* (PC, *instruction address* register)
  - The register containing the address of current instruction
  - The `jal` instruction saves PC+4 in register `$ra`

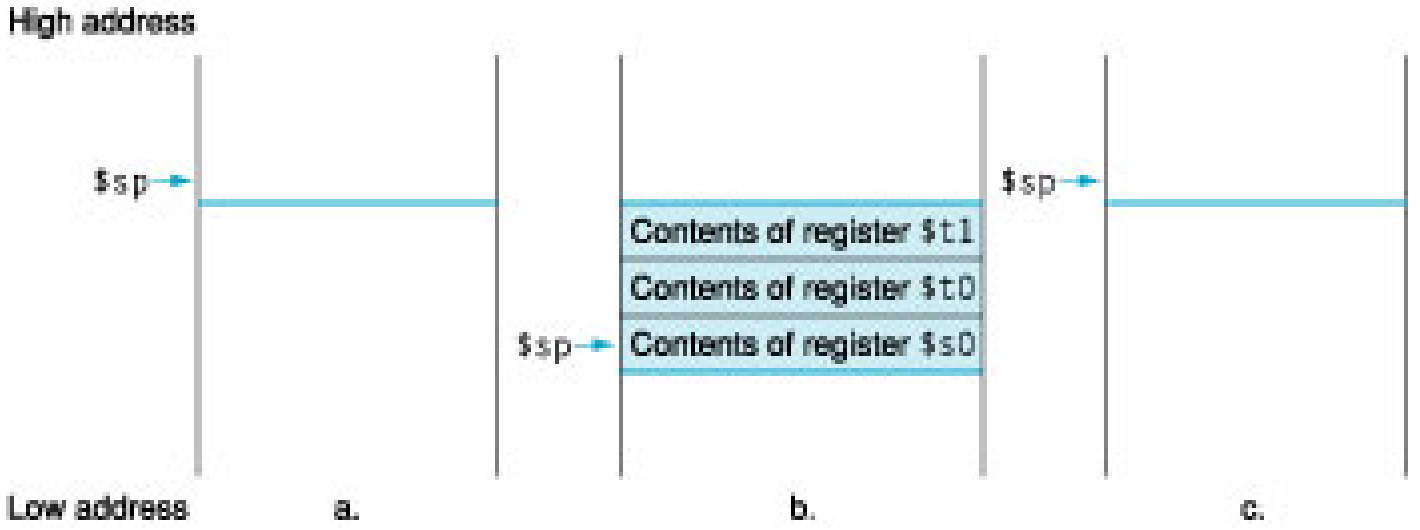
- *Jump register* instruction (`jr`)
  - `jr $ra`

- The calling program (*caller*) puts the parameter values in `$a0-$a3` and uses `jal X` to jump to procedure `X` (*callee*)
- The callee then performs the calculations, places the results in `$v0-$v1`, and returns control to the caller using `jr $ra`

Using More Registers

- If four argument and two return value registers are not enough…
- Stack
  - A last-in-first-out (LIFO) queue
  - "grow" from higher addresses to lower addresses
  - *Push*: subtracting from the stack pointer ($sp)
  - *Pop*: adding to the stack pointer

Example (p.114)



Before the procedure call | During the procedure call | After the procedure call

Nested Procedures

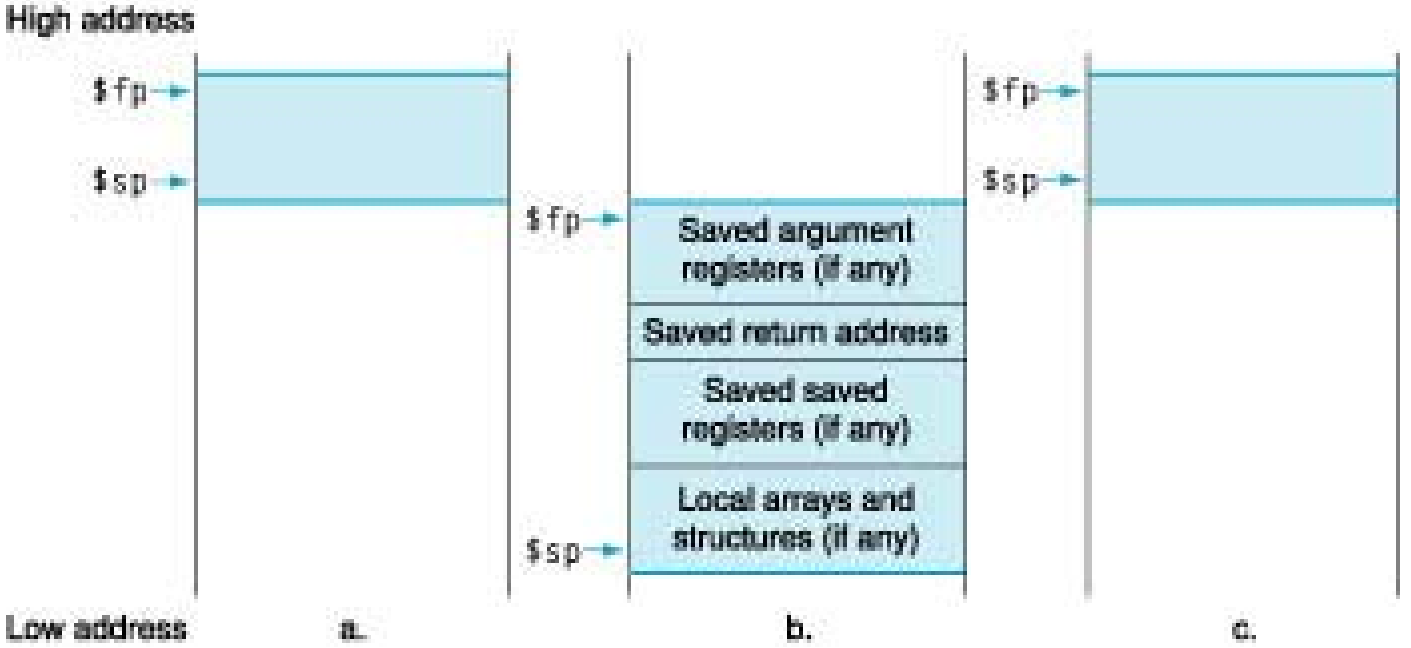
- **Conflict over registers:**
  - **Argument registers**
  - **Return register**
  - **…**

- **Solution: use the stack again**
  - **The caller pushes any argument registers (`$a0-$a3`) or temporary registers (`$t0-$t9`) that are needed after the call**
  - **The callee pushes the return address `$ra` and any saved registers (`$s0-$s7`) used by the callee**
  - **`$sp` is adjusted to account for the number of registers placed on the stack**
  - **Upon the return, the registers are restored from memory and the stack pointer is readjusted**

Example (p.117)

Allocating Space for New Data on the Stack

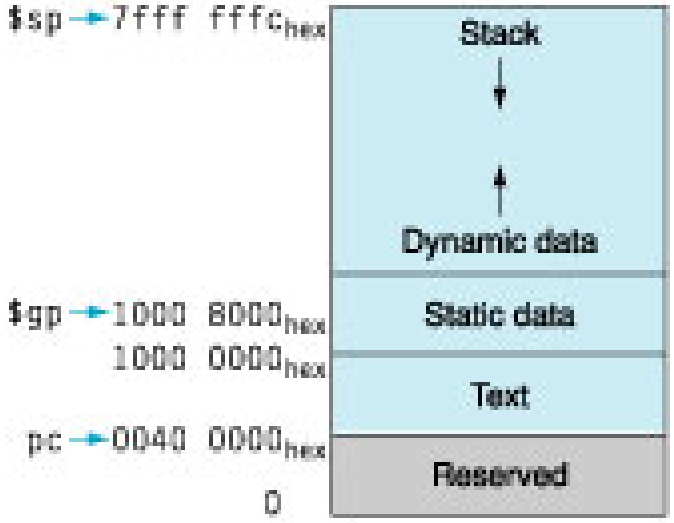
- *Procedure frame*
  - Also called activation record
  - The segment of the stack containing a procedure's saved registers and local variables
- *Frame pointer* (`$fp`)
  - A value denoting the first word of the frame of a procedure



Before the procedure call | During the procedure call | After the procedure call

Allocating Space for New Data on the Heap

- **Text segment**
  - **The home of the MIPS machine code**
- **Static data segment**
  - **For constants and other static variables**
- **Heap**
  - **For dynamic data**



Communicating with people

- **American Stand Code for Information Interchange (ASCII)**
  - **Use 8-bit bytes to represent characters**

Example (p.123)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Communicating with people

- Instructions to move bytes
- *Load byte* (`lb`)
  - Load a byte from memory
  - Place it in the rightmost 8 bits of a register
  - `lb $t0, 0($sp)`

- *Store byte* (`sb`)
  - Take a byte from the rightmost 8 bits of a register
  - Write it to memory
  - `sb $t0, 0($gp)`

Global pointer ($gp): the register that is reserved to point to static data

- Combine characters to strings

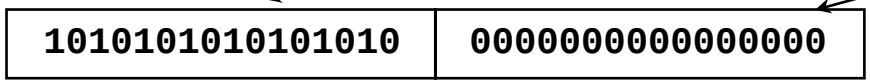
Example (p.124)

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

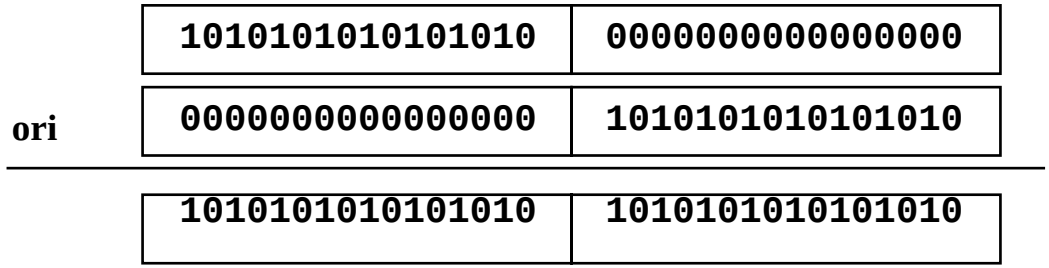
```
lui $t0, 1010101010101010
```

filled with zeros



- Then must get the lower order bits right, i.e.,

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



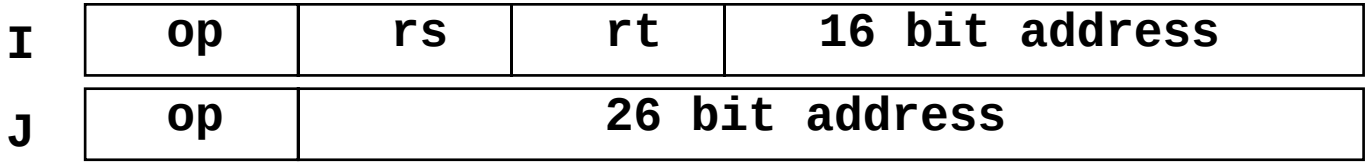
- The machine language version of `lui $to, 255`

Addresses in Branches and Jumps

- **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`
`j Label`	Next instruction is at Label

- **Formats:**



- **Addresses are not 32 bits**
  - **— How do we handle this with load and store instructions?**

Addresses in Branches

- **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

- **Formats:**

- **Could specify a register (like `lw` and `sw`) and add it to address**
  - **use Instruction Address Register (PC = program counter)**
  - **most branches are local (principle of locality)**

PC-relative addressing

- **Jump instructions just use high order bits of PC**
  - **address boundaries of 256 MB**

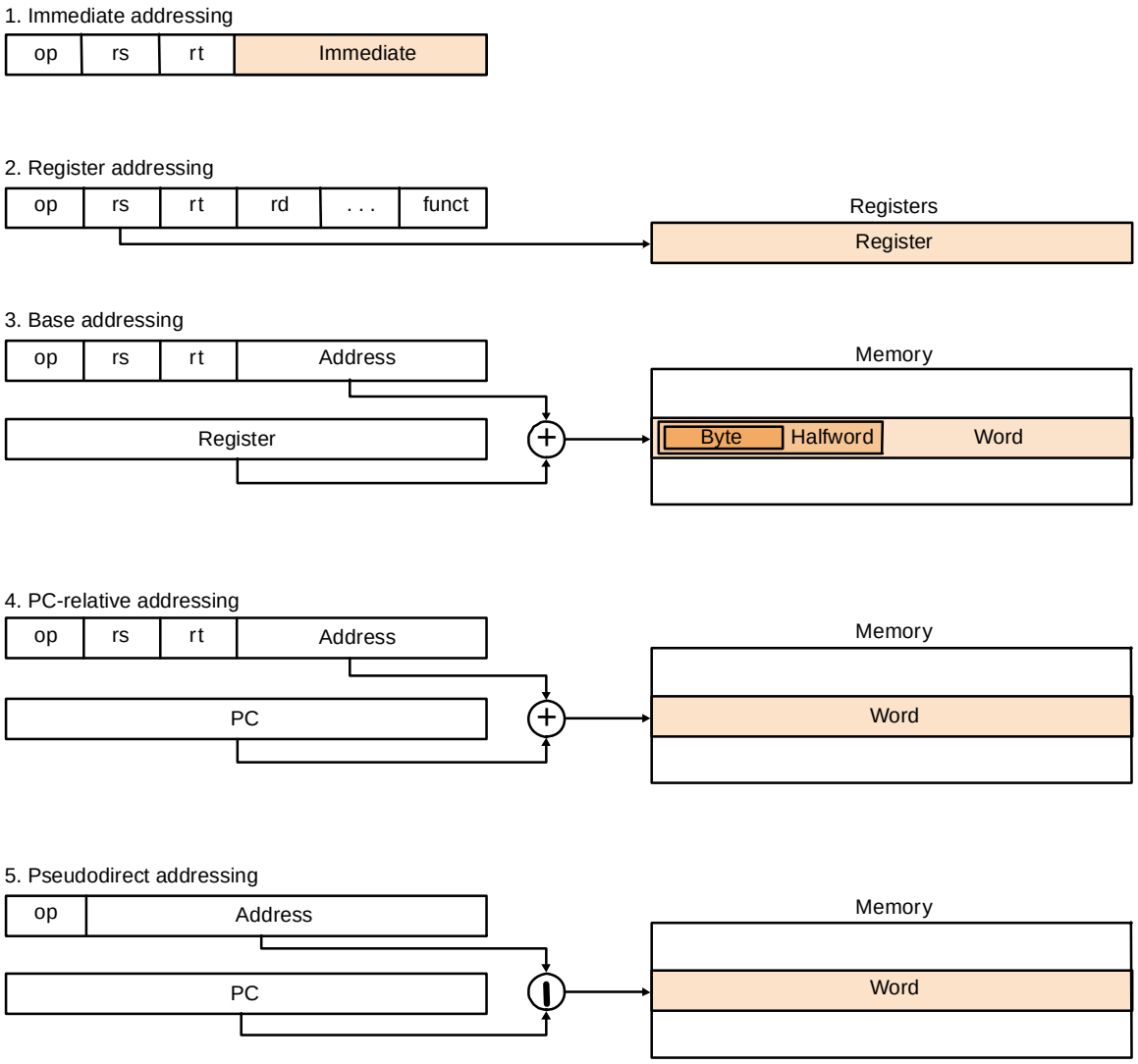
Pseudodirect addressing

Addresses in Branches and Jumps

- **Branch**
  - **PC-relative addressing (I-format)**
  - **16-bit immediate specifies word address**
  - **Due to hardware, add `immediate` to (PC+4), not to PC**
  - **If branch not taken: PC = PC + 4**
  - **If branch taken: PC = (PC+4) + (`immediate`*4)**
  - **Example:**
    `beq $1,$2,25` if ($1 == $2) go to PC+4+100
- **Jump**
  - **Pseudodirect addressing (J-format)**
  - **26-bit immediate specifies word address**
  - **32-bit address = [PC: 4 bits] [immediate: 26 bits] [00]**
  - **Example:**
    `j 10000` go to 10000 26-bit+4-bit of PC

Example (p.131)
Example (p.132)

MIPS addressing modes



Decoding Machine Language

- Machine instruction
  - $00af8020_{hex}$
- Step 1: convert hexadecimal to binary
  - 0000 0000 1010 1111 1000 0000 0010 0000
- Step 2: determine the operation by the op field
  - op = 000000 → R-format
  - op rs rt rd shamt funct
    000000 00101 01111 10000 00000 100000
  - funct = 100000 → add instruction
- Step 3: decode the rest fields
  - rs: $00101_{two} = 5_{ten}$ → $a1
  - rt: $01111_{two} = 15_{ten}$ → $t7
  - rd: $10000_{two} = 16_{ten}$ → $s0
- Assembly instruction
  - add $s0, $a1, $t7

Figure 2.19 (p.135)

Figure 2.20 (p.136)

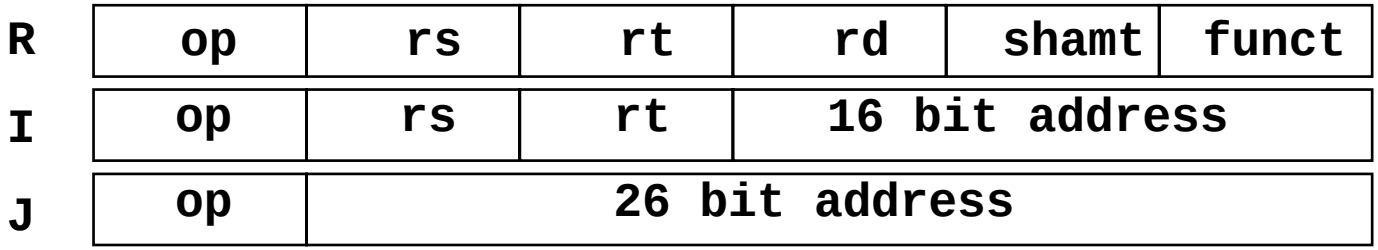
Figure 2.14 (p.121)

Assembly Language vs. Machine Language

- **Assembly provides convenient symbolic representation**
  - **much easier than writing down numbers**
  - **e.g., destination first**
- **Machine language is the underlying reality**
  - **e.g., destination is no longer first**
- **Assembly can provide 'pseudoinstructions'**
  - **e.g., “move $t0, $t1” exists only in Assembly**
  - **would be implemented using “add $t0,$t1,$zero”**
- **When considering performance you should count real instructions**

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

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
Uncondi-tional 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

Synchronization

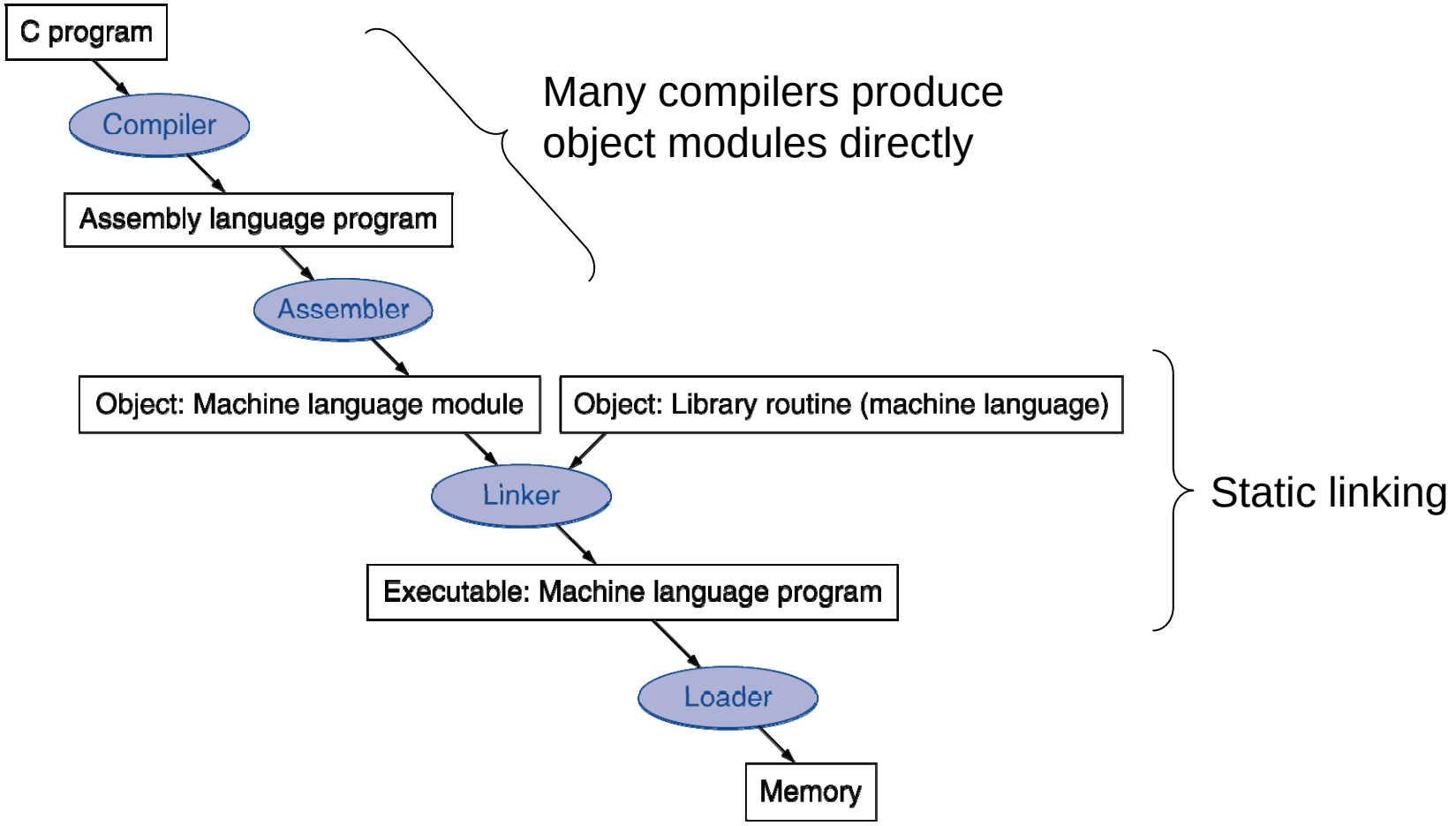
- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Data race: two memory access form a data race if they are from different threads to same location, at least one is a write, and they occur one after another
    - Result depends on order of accesses

- Hardware support required
  - Lock/unlock: create regions where only one processor can operate (mutual exclusion)
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Instruction pair for atomic exchange or swap
  - Atomic swap of register ↔ memory
  - MIPS instruction: load linked (`ll`) and store conditional (`sc`)

Example (p.138)

Translation and Startup



Translating and Starting a Program

- **Compiler**
  - **Transform the C program into an *assembly language program***
  - ***Assembly language***
    - **A symbolic language that can be translated into binary**
- **Assembler**
  - **Convert the assembly language program into an *object file*, which is a combination of *machine language* instructions, data, and information needed to place instructions properly in memory**
  - ***Machine language***
    - **Binary representation used for communication within a computer system**
  - **Accept *pseudoinstructions***

Pseudoinstructions

- A common variation of assembly language instructions often treated as it were an instruction in its own right
- Example:
  - **`move $to, $t1` → `add $to, $zero, $t1`**
  - **`blt (branch on less than), bgt, bge, ble, …`**
    - **`slt, bne, beq`**
  - **`Branch to faraway locations`**
    - **`Branch, jump`**

- Provide MIPS a richer set of assembly language instructions than those implemented by the hardware

- The only cost is reserving one register, **`$at`**, for use by the assembler

Object File

- *Object file header*
  - Describe the size and position of the other pieces of the object file
- *Text segment*
  - Contain the machine language code
- *Static data segment*
  - Contain data allocated for the life of the program
- *Relocation information*
  - Identify instructions and data words that depend on absolute addresses when the program is loaded into memory
- *Symbol table*
  - Contain the remaining labels that are not defined, such as external references
- *Debugging information*
  - Contain a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable

Linker

- **Also called *link editor***
- **A system program that combines independently assembled machine language programs and resolves all undefined labels into an executable file**
- **It is much faster to patch code than it is to recompile and reassemble**
- **Three steps:**
  - **Place code and data modules symbolically in memory**
  - **Determine the addresses of data and instruction labels**
  - **Patch both the internal and external references**
- **The linker uses the *relocation information* and *symbol table* in each object module to resolve all undefined labels**
- **When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, much be *relocated* to reflect its true location**
- **Produce an *executable file***

Example (p.143)

Loading a Program

- Load the executable file from disk into memory
  1. Read header to determine segment sizes
  2. Create an address space large enough for the text and data
  3. Copy text (instructions) and data from the executable file into memory
  4. Set up arguments on stack
  5. Initialize registers and stack pointer
  6. Jump to a startup routine
     - Copies arguments to `$a0`, … and calls main
     - When main returns, startup terminates with `exit` system call

Dynamic Linking

- **Only link/load library procedure when it is called**
  - **Requires procedure code to be relocatable**
  - **Avoids image bloat caused by static linking of all (transitively) referenced libraries**
  - **Automatically picks up new library versions**

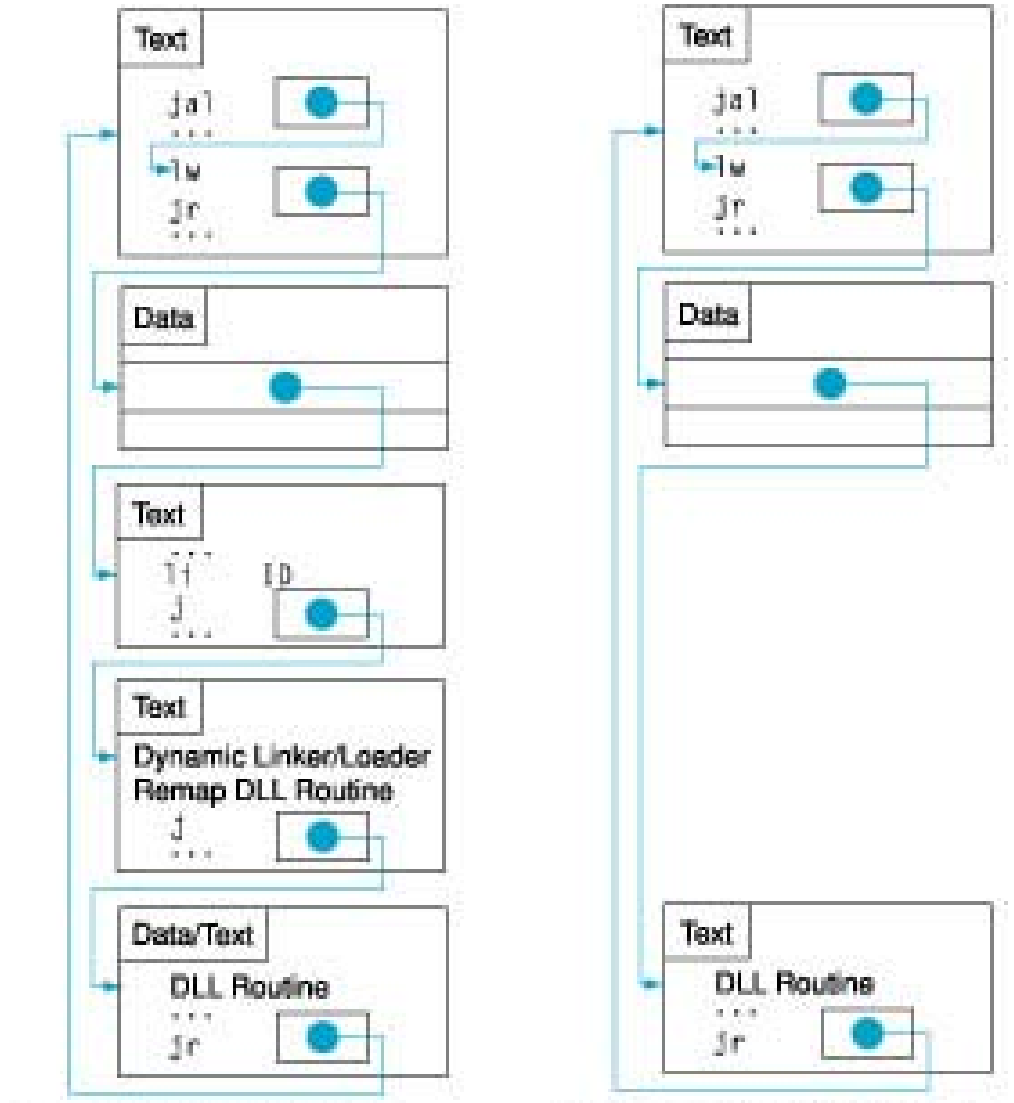
Lazy Linkage

Indirection table

Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



(a) First call to DLL routine (b) Subsequent calls to DLL routine