Instructions: Language of the Computer

CSI3102-02: Architecture of Computers
(컴퓨터아키텍쳐)

Youngsok Kim (김영석)

Talking with Computers

- To command a computer's hardware, we need to speak the computer's language.
- A computer's words are called **instructions**, and its vocabulary is called an **instruction set**.

- We will focus on the **MIPS** ISA.
  - Has been developed by MIPS Technologies since the 1980s
- Various ISAs exist; however, they are quite similar.
  - All computers should support a few basic operations.
  - We will discuss some of **ARMv7/ARMv8** ISAs as well.

MIPS: A PL for MIPS CPUs

- MIPS is an **ISA** defining all sorts of things of a CPU.
- MIPS is a **PL** for specifying what the CPU should do.

- By viewing MIPS as a PL, we will discuss
  - Its **programming model** & **paradigm**
  - Its **syntax** (data types, operations, ...)
    - What kinds of strings/values are accepted by the MIPS ISA?
  - Its **semantics** (instructions, ...)
    - How does the MIPS ISA assign meaning to the strings/values?

Collectively, a.k.a. **Programmer-Visible State** (PVS)

MIPS: Imperative Programming

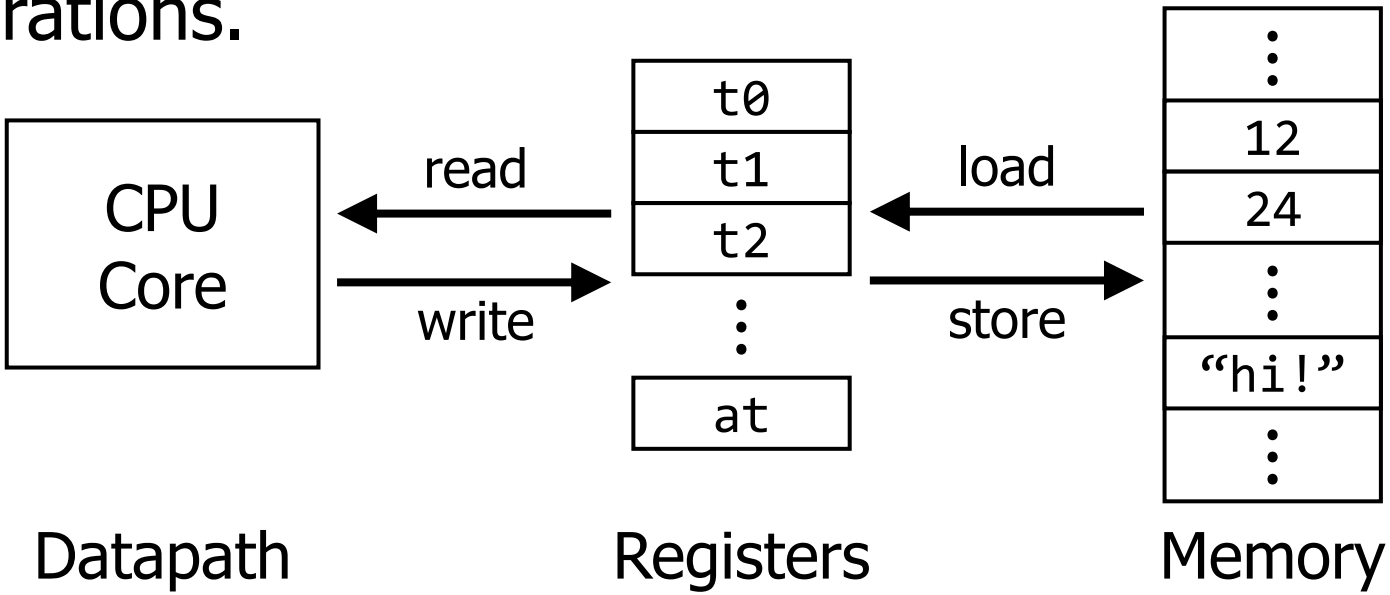
- MIPS instructions change the **CPU's state**!
  - e.g., values stored in some memory

- All instructions are executed **sequentially**.
  - Support jumps/branches which adjust the execution flow
  - Similar to C/C++, Java, Python, ...

- The PVS of the MIPS ISA consists of
  - **A CPU core** for executing MIPS instructions
  - **Registers** which the CPU cores can directly access to
  - **Memory** whose data can be loaded to the registers

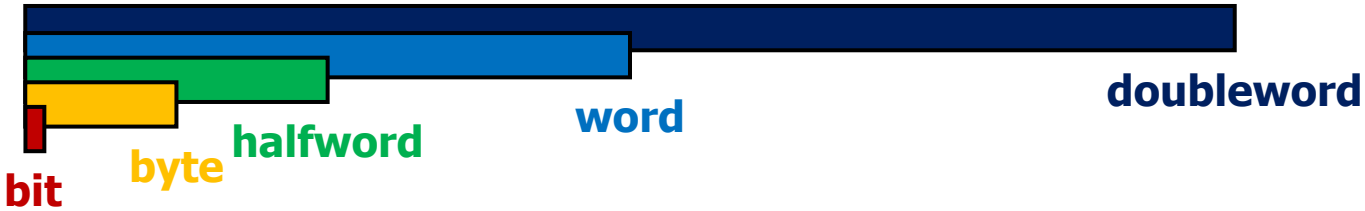
MIPS: Programming Model

- A **load-store** architecture with 32 **registers**
  - Divide instruction into memory access and ALU operations
    - One instruction cannot perform memory access and ALU operations at the same time.
- The CPU core accesses the registers using the ALU operations.



MIPS: Bits & Data Sizes

- All data in a CPU are stored as **binary numbers**.
  - Widely referred to as **bits**; each bit is either **`0`** or **`1`**.
  - e.g., $1100_{(2)}$ instead of $12_{(10)}$
- Popular data sizes have special names!
  - **Bit**: 1-bit data
  - **Byte**: 8-bit data
  - **Halfword**: 16-bit (2-byte) data
  - **Word**: 32-bit (or 4-byte) data
  - **Doubleword**: 64-bit (or 8-byte) data

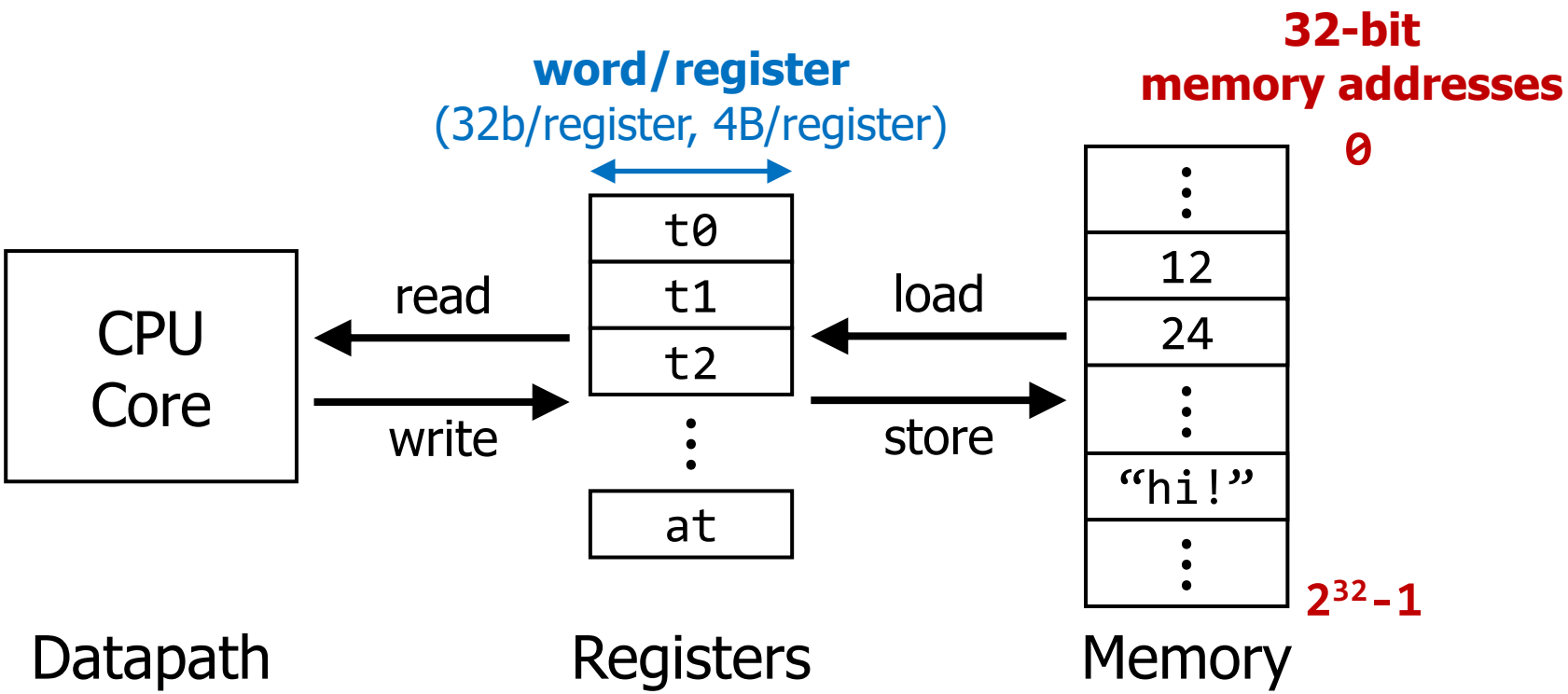


MIPS: Numbers

- **Unsigned**: n-bit-long **positive** binary numbers
  - Range: $0 \sim 2^n-1$
  - With 4 bits, min value: $0000_{(2)}$ (= 0)
    max value: $1111_{(2)}$ (= 8+4+2+1 = 15 = 2^4-1)

- **Signed**: n-bit-long **two's complement**
  - Use the first bit as a **sign** bit (0: positive, 1: negative)
  - Range: $-2^{n-1} \sim 2^{n-1}-1$
  - With 4 bits, min value: $1000_{(2)}$ (= -8 = -2^3)
    max value: $0111_{(2)}$ (= 4+2+1 = 7 = 2^3-1)

Programming Model w/ Data Sizes



32 * 4 Bytes
= 128 B = 1 Kib

MIPS: Registers & Memory

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

- **32 registers**
  - 32 bits/register
  - Some registers have special purposes (e.g., `$zero`).
- **2^{30} memory words**
  - Equivalent to **2^{32} memory bytes** (1 word = 4 bytes)

MIPS: Data Placement

- MIPS supports both **big-** and **little-endian**.
  - Big-endian places the Most-Significant Byte (MSB) first.
  - Little-endian places the Least-Significant Byte (LSB) first.
    - e.g., $0x111001_{(16)}$ is stored as "$011011_{(16)}$", not "$111001_{(16)}$"

Big-Endian	11	00	01
Byte Address	0	1	2

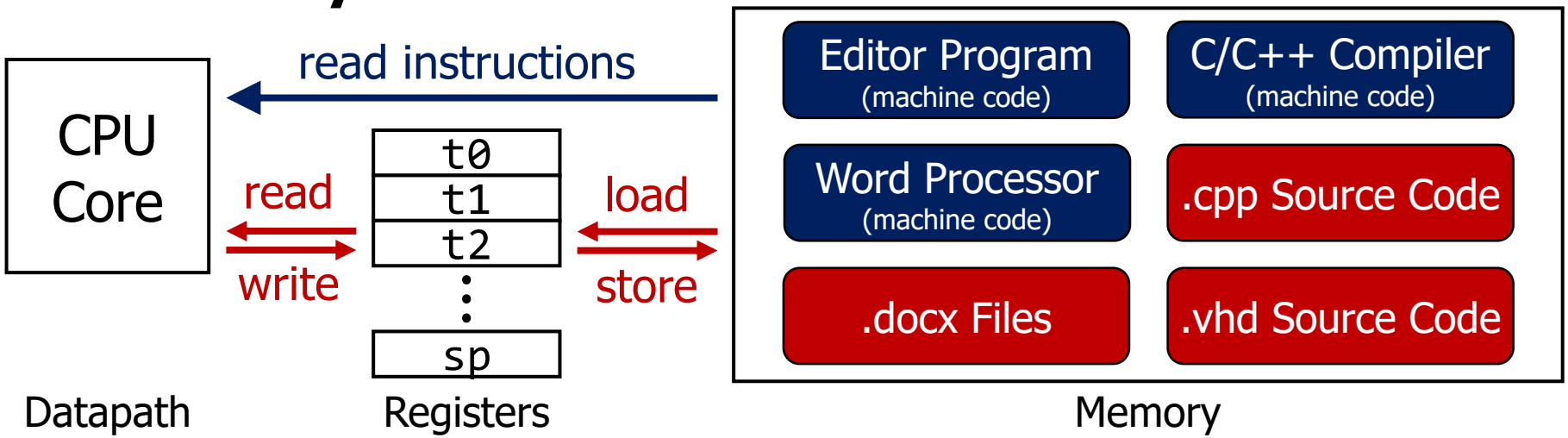
Little-Endian	01	10	11
Byte Address	0	1	2

- Many ISAs support little-endian only.
  - Reading more bits == reading larger data sizes
  - e.g., read 2/4 bytes from some memory address X
    → read halfword/word values from memory address X

MIPS: Two Key Principles

- Instructions are represented **as numbers**.
  - With respect to the appropriate instruction formats

- **Stored-program concept**
  - MIPS ISA stores both instructions and data **in the main memory**.



MIPS: Design Principles

- **Simplicity favors regularity**.
  - e.g., Require every arithmetic/logic instruction to have exactly three operands.
- **Smaller is faster**.
  - e.g., The limit of 32 word-long registers (not 64 registers, 128 registers, ...)
- **Good design demands good compromises**.
  - e.g., Keep all instructions the same length (32 bits) rather than having a single instruction format for all.

These properties greatly simplify the hardware complexity of MIPS processors!

MIPS: Instructions

- **Arithmetic** instructions
  - Perform an arithmetic operation using the 32 registers
  - Two input operands (either a register or an **immediate**)
  - One output operand (a register)

- **Data transfer** instructions
  - Transfer data between registers and memory
  - Involve the size of the data (e.g., word)

Arithmetic Instructions

- Each arithmetic instruction performs only one op, and it must always have exactly three variables.
  - `add t0, t1, t2` adds two registers `t1` and `t2`, and puts the sum in variable `t0`.

- They can use constants as an input operand.
  - Constants in instructions are called **immediates**.
  - e.g., `addi t0, t1, 20` performs `t0 = t1 + 20`

MIPS assembly language

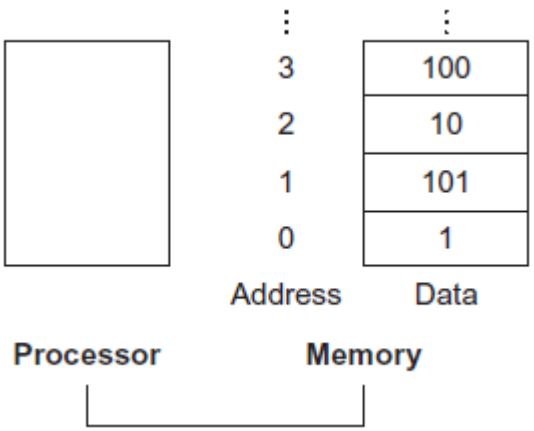
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants

Data Transfer Instructions (1/3)

- How can we access large arrays, structures, etc.?
  - Arithmetic instructions can only access the 32 registers.

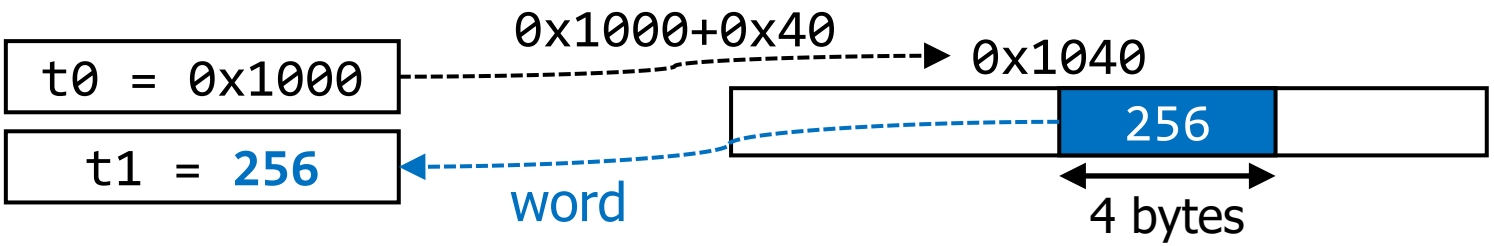
- **Data transfer instructions** & memory **addresses**
  - Load/store data from/to main memory using addresses
  - e.g., The values of a 1-D array get stored in a sequential manner in the main memory from the starting address of the array.

    memory[0..2] = {1,101,10} →



Data Transfer Instructions (2/3)

- Transfer data **between registers and memory**
  - The access granularity of memory is **byte** (= 8 bits).
- Need to specify the **size** of the data to load/store
  - e.g., `lw` loads a **word** (= 32 bits = 4 bytes)
- Also, the **offset** to be added to the base register
  - address-to-access = the base register's value + offset
  - e.g., `lw t1, 0x40(t0)` loads a word starting from memory address `0x1040`, and stores the word to register `t1`.



Data Transfer Instructions (3/3)

- Combinations of load/store and data widths

Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

- Some special load/store instructions (i.e., `ll`, `sc`) perform an atomic swap operations as well.

MIPS: Instruction Format (1/2)

- Define the **layout** of an instruction in the memory
  - e.g., $00000010010010000100000000100000_{(2)}$
    represents `add $t0, $s2, $t0`

- All instructions are **32 bits** and consist of **fields**.

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

  - **op**: opcode; basic operation of the instruction (e.g., add)
  - **rs**, **rt**: the 1st and 2nd register source operands
  - **rd**: the register destination operand
  - **shamt**: shift amount
  - **funct**: function; selects the specific variant of the op

MIPS: Instruction Format (2/2)

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

- e.g., Converting `add $t0, $s2, $t0` to
  `000000 10010 01000 01000 00000 100000`$_{(2)}$
  - **op** = `add` (`000000110011`)
  - **rs** = `$s2` (`10010`)
  - **rt** = `$t0` (`01000`)
  - **rd** = `$t0` (`01000`)
  - **shamt** = 0 (`00000`)
  - **funct** = 32 (`100000`)

MIPS: Instruction Types

- Different instructions require different formats!
  - e.g., `adds` must specify three registers,
    `loads` must specify two registers and an immediate,
    `stores` must specify two registers and a doubleword

- Provide different instruction formats to instructions depending on their needs (or **types**)
  - **R-type**: two source registers & one destination register
  - **I-type**: one source register, one destination register, and an immediate
  - **J-type**: the instruction format for `jump` instructions

MIPS: R- & I-Type Instructions

- **R-type**: two source registers (rs, rt), one destination register (rd)

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

- **I-type**: one register and an immediate as input

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

  - One source register (rs), one destination register (rt), and a 16-bit signed immediate (constant or address)
  - Used for arithmetic operations with one constant operand and load instructions

MIPS: Logical Operations (1/3)

- **Shift** operations (e.g., `sll`, `srl`)
  - Shift all the bits in a doubleword to the left or right, filling the emptied bits with **0s**.
    - e.g., `slli x11, x19, 4 // <-- x11 = x11 << 4 bits`
      If x19 = ...$00001111_{(2)}$, then x11 = ...$00001111\mathbf{0000}_{(2)}$.
  - Useful for multiplying/dividing unsigned numbers by 2^i
    - e.g., $0010_{(2)} * 2^2_{(10)} = 1000_{(2)}$, $1000_{(2)} / 2^3_{(10)} = 0001_{(2)}$
  - Utilize the **R-type** instruction format
    - e.g., `sll $t2, $s0, 4`

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

`sll` → op = 0, funct = 0
$t2 → rd = 10, $s0 → rt = 16, 4 → shamt = 4

MIPS: Logical Operations (2/3)

- Bitwise/Bit-by-bit operations
  - **`and`** and **`andi`** instructions: bitwise logical AND
    - Useful for isolating a field from an instruction
    - e.g., Isolate the 6-bit op field from an instruction through
      `instr & 11111100...00` (2)
      → All the bits except the op field become 0!

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

Remains
(& 111111)

Zero'ed out
(& 0s)

  - **`or`** and **`ori`** instructions: bitwise logical OR
  - **`nor`** instruction: bitwise logical NOT

MIPS: Logical Operations (3/3)

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

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

MIPS: Representing Text

- Support **ASCII characters**
  - Require **one byte** (or 8 bits) to store **one character**
  - e.g., $65_{(10)}$ represents 'A', $109_{(10)}$ represents 'm'.

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

ASCII representation of characters
(ASCII: the American Standard Code for Information Exchange)

MIPS: Text Processing

- Involves **byte-by-byte** operations
  - e.g., count the number of characters in a string,
    parse a C statement byte-by-byte
- **Load byte** (`lb`) instruction
  - e.g., `lb $t0, 0($sp)`
  - Read a byte from the memory address `$sp+0`, and store the byte in the rightmost 8 bits of `$t0`
- **Store byte** (`sb`) instruction
  - e.g., `sb $t0, 0($gp)`
  - Take a byte from the rightmost 8 bits of register `$t0`, and store the byte at the memory address `$gp+0`

Conditional Branches

- **Alter the execution flow** of instructions
  - e.g., `if-then-else` and `goto` statements in C

- **Branch if equal** (**`beq`**) instruction
  - Syntax: `beq r1, r2, L1`
  - Go to the statement labeled `L1` if the value in register `r1` **is equal to** the value in register `r2` (`r1 == r2`)
- **Branch if not equal** (**`bne`**) instruction
  - Syntax: `bne r1, r2, L1`
  - Go to the statement labeled `L1` if the value in register `r1` **is not equal** to the value in register `r2` (`r1 != r2`)

Example: Conditional Branches

```c
int i, j, f, g, h;


/* ... some code ... */

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

/* ... some code ... */
```

C code

```mips
  // t0: i, t1: j
  // t2: f, t3: g, t4: h

  // ... some code ...

  bne t0, t1, Else
  add t2, t3, t4
  beq zero, zero, Exit
Else:
  sub t2, t3, t4
Exit:

  // ... some code ...
```

MIPS assembly code

Loops

- `beq`/`bne` instructions test equality/inequality.
- Loops demand **more tests** such as ≤, ≥, ‹, ›, …
  - e.g., `for (int i = 0; i < 10; i++) {}` needs **<** test.

- **Set on less than** (**`slt`**) instruction
  - Syntax: `slt rd, rs, rt`
  - Set `rd` = 1 if `rs` < `rt`, `rd` = 0 if `rs` ≥ `rt`
- An immediate version of the slt instruction (`slti`)
  - Syntax: `slti rd, rs, const`
  - Set `rd` = 1 if `rs` < `const`, `rd` = 0 if `rs` ≥ `const`

Case/Switch Statements

- Many PLs have **`case`** or **`switch`** statement.
  - Select one of many paths depending on a single value
  - e.g., `switch (val) { case 0: ...; case 1: ...; }`

- Solution 1: Replace with a chain of `if-then-elses`.
- Solution 2: **Indirect branches**
  - Store target addresses in a branch (address) table
  - Load a target address from the branch table to a register
  - Use a **jump register** (`jr`) instruction on the register
    - Take an unconditional jump to the address stored in the register

Supporting Procedures/Functions

- One way to abstract multiple operations
  - e.g., simply write as function `addTwoNums(a, b)` instead of writing all the code for adding `a` and `b`.
- Involve passing control and data between procedures (or **functions**)
  - e.g., pass `num1 = 1`, `num2 = 2` to function `addTwoSums`, execute `addTwoSums`, pass `c = 3` to function `main`

```
void main() {
  int num1 = 1, num2 = 2, num3;
  num3 = addTwoSums(num1, num2);
  printf("num3 = %d\n", num3);
}
```

```
int addTwoSums(int a, int b) {
    int c = a + b;
    return c;
}
```

Function Calls (1/2)

- Follow the following register allocation convention
  - `a0~a3`: four **argument registers** to pass parameters
  - `v0~v1`: two **value registers** to return values
  - `ra`: **return address register** to store the point of origin

- **Jump-and-link** (`jal`) instruction
  - Syntax: `jal ProcedureAddress`
  - Jump to `ProcedureAddress` & write return address to `ra`
- **Jump register** (`jr`) instruction
  - Syntax: `jr $ra`
  - Jump to the address stored in register `ra`

Function Calls (2/2)

- Invoking a **callee** function from a **caller** function
  - Caller: Put the parameter values in registers `a0 ~ a3`
  - Caller: Use `jal A` to branch to the callee function `A`
  - Callee: Perform calculations, put the results in `v0 ~ v1`
  - Callee: Return control to the caller with `jr $ra`

- **Program Counter (PC)** register
  - Store the memory address of the current instruction
  - Used to store the next instruction address when `jal`'ing,
  - e.g., When `jal`'ing, **store PC+4 to `$ra`** so that we can resume the current function's execution.

Example: Function Calls (1/6)

PC	0x2000
ra	
a0	123
a1	
v0	

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra

        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

Example: Function Calls (2/6)

PC	**0x2004**
ra	
a0	123
a1	**321**
v0	

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra

        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

Example: Function Calls (3/6)

PC	**0x2008**
ra	**0x200C**
a0	123
a1	321
v0	

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra


        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

Example: Function Calls (4/6)

PC	0x1000
ra	0x200C
a0	123
a1	321
v0	**444**

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra

        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

Example: Function Calls (5/6)

PC	**0x1004**
ra	**0x200C**
a0	123
a1	321
v0	444

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra

        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

Example: Function Calls (6/6)

PC	**0x200C**
ra	0x200C
a0	123
a1	321
v0	444

```
        addTwoSums:
0x1000    add $v0, $a0, $a1
0x1004    jr $ra

        main:
          ...
0x2000    addi $a0, $zero, 123
0x2004    addi $a1, $zero, 321
0x2008    jal addTwoSums
0x200C    ...
```

MIPS assembly Code

What If We Need More Registers?

- Argument registers (`$a0`~`$a3`) may not be enough!
  - e.g., functions with more than four arguments

- Use a **stack** to **spill registers**
  - Stack: last-in, first-out data structure
  - Backup a few registers to the stack, use the now-free registers for operations, and then restore the registers
  - Utilize the **stack pointer** (`$sp`)
    - Points to the most recently allocated address in the stack
  - The stack grows from higher addresses to lower addresses.
    - Decrement `$sp` when pushing/spilling a register to the stack
    - Increment `$sp` when popping/restoring a register from the stack

MIPS: Register Spilling

- Previously, the register allocation convention is:
  - `$a0` ~ `$a3`: parameter registers
  - `$ra`: return address register

- Now, we add the followings to the convention:
  - `$t0` ~ `$t9`: **temporary** registers **not preserved** by the callee function upon a function call
    - If the caller uses them, the caller should spill & restore them.
  - `$s0` ~ `$s7`: **saved** registers which **must be preserved** on a function call
    - If the callee uses them, the callee should spill & restore them.

Spilling Registers with a Stack

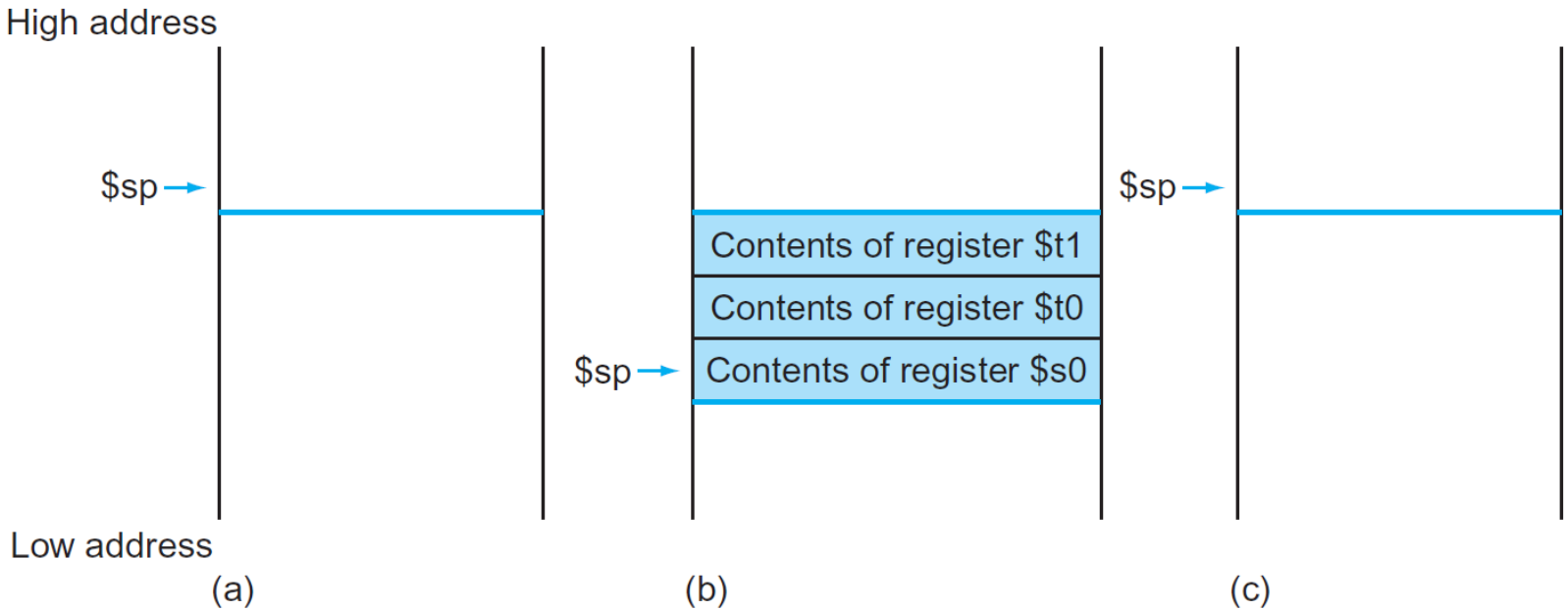
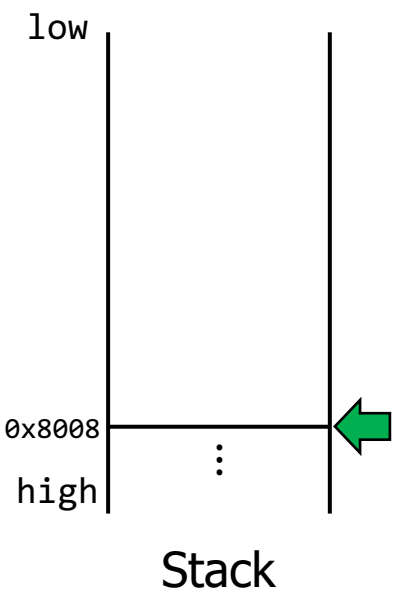


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

- A **caller** needs to backup the **temporary** registers (`$t0` ~ `$t9`), not the saved registers (`$s0` ~ `$s7`).

Example: Register Spilling (1/12)



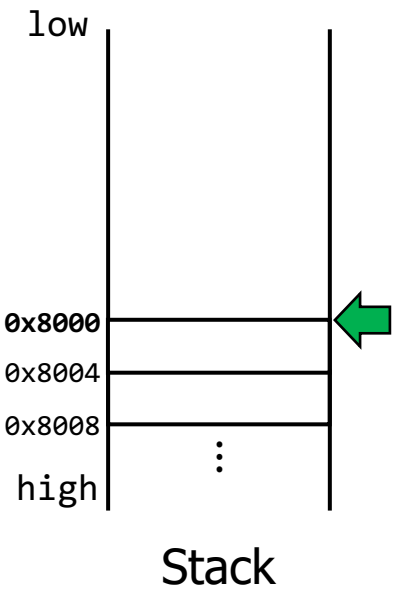
PC	0x1FFC
$ra	
$sp	0x8008
$t0	"LIVE"
$t1	"VALs"
$a0	
$a1	
$v0	

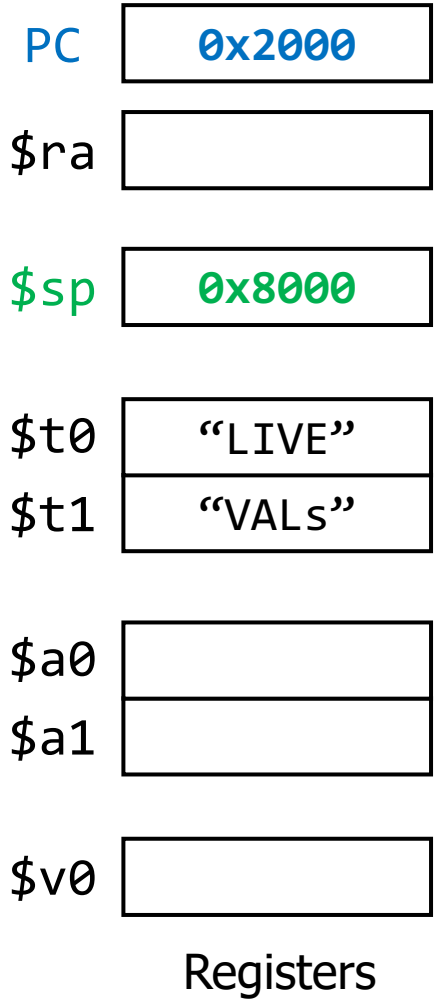
Registers

```
0x1FFC  ...
0x2000  addi $sp, $sp, -8
0x2004  sw $t1, 4($sp)
0x2008  sw $t0, 0($sp)

0x200C  addi $a0, $zero, 123
0x2010  addi $a1, $zero, 321
0x2014  jal complexFunction

0x2018  lw $t1, 4($sp)
0x201C  lw $t0, 0($sp)
0x2020  addi $sp, $sp, 8
0x2038  ...
```

MIPS assembly code

Example: Register Spilling (2/12)

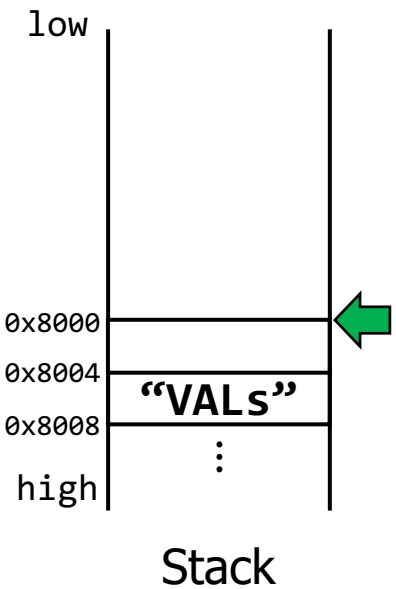


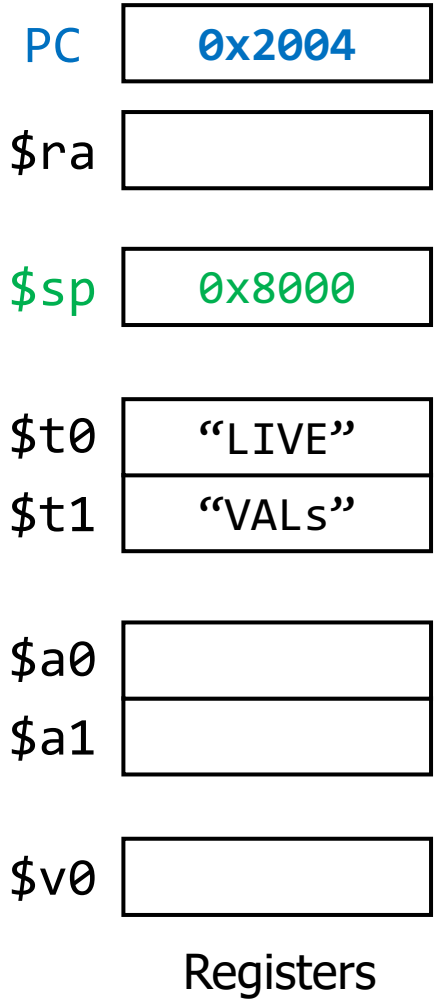



```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (3/12)

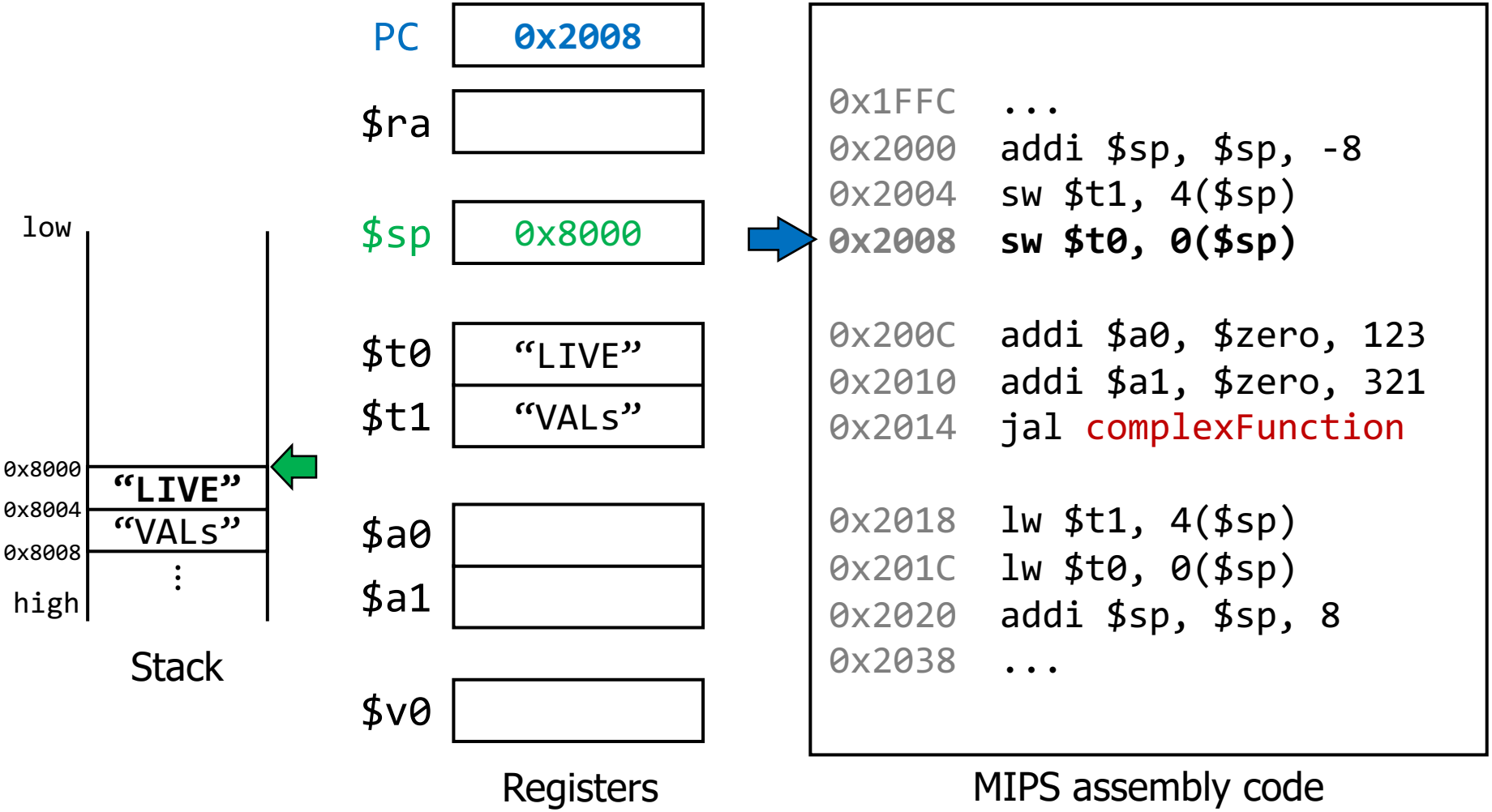




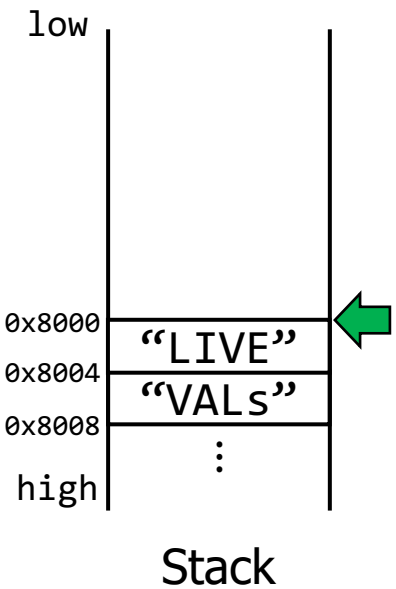

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

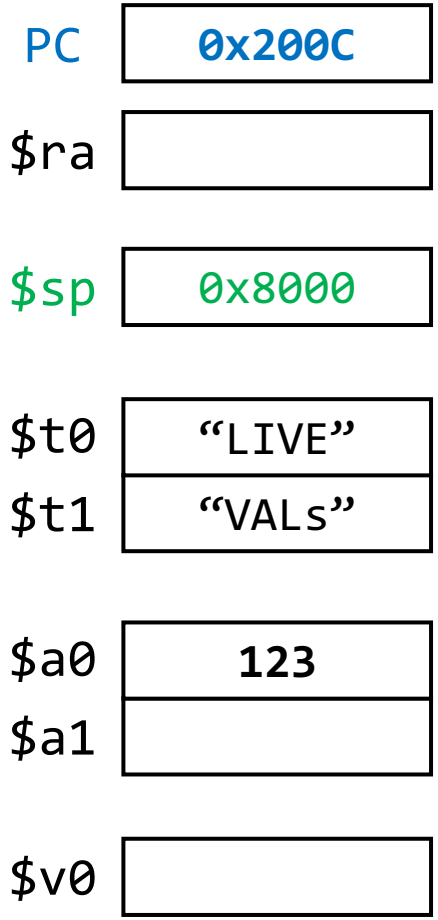
MIPS assembly code

Example: Register Spilling (4/12)



Example: Register Spilling (5/12)

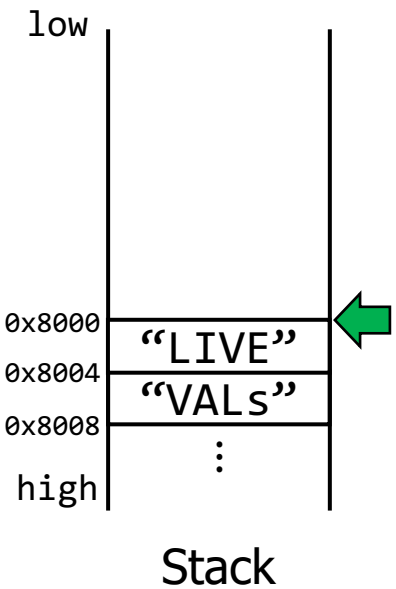





```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (6/12)



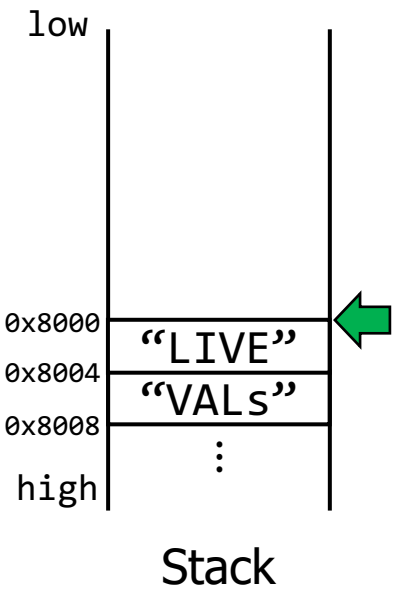
PC	0x2010
$ra	
$sp	0x8000
$t0	"LIVE"
$t1	"VALs"
$a0	123
$a1	**321**
$v0	

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (7/12)



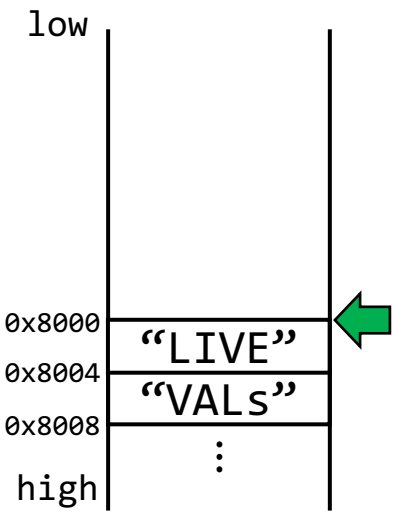
PC	0x2014
$ra	0x2018
$sp	0x8000
$t0	"LIVE"
$t1	"VALs"
$a0	123
$a1	321
$v0	

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (8/12)



Stack

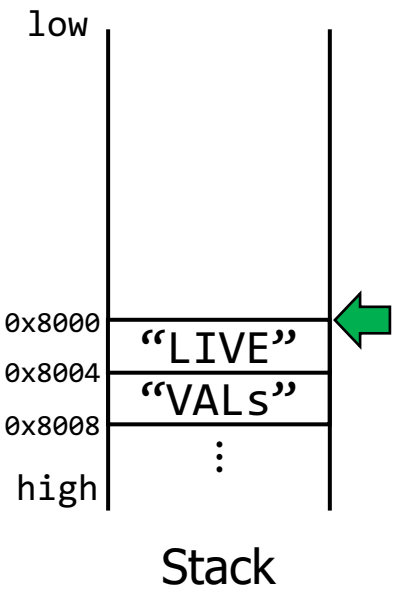
PC	0x....
$ra	0x2018
$sp	0x8000
$t0	"SOME"
$t1	"DATA"
$a0	123
$a1	321
$v0	444

Registers

```
0x1FFC  ...
0x2000  addi $sp, $sp, -8
0x2004  sw $t1, 4($sp)
0x2008  sw $t0, 0($sp)

0x200C  addi $a0, $zero, 123
0x2010  addi $a1, $zero, 321
0x2014  jal complexFunction

0x2018  lw $t1, 4($sp)
0x201C  lw $t0, 0($sp)
0x2020  addi $sp, $sp, 8
0x2038  ...
```

MIPS assembly code

Example: Register Spilling (9/12)



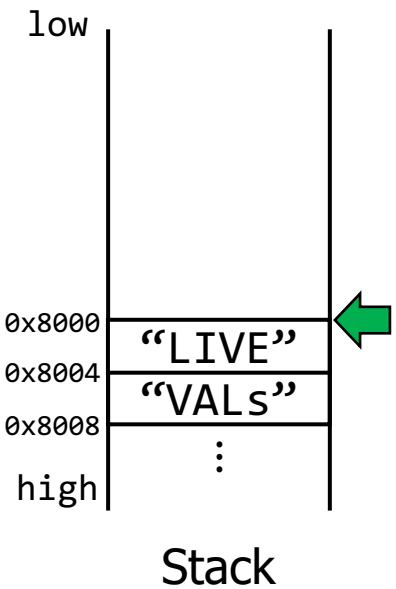
PC	0x2018
$ra	0x2018
$sp	0x8000
$t0	"SOME"
$t1	"VALs"
$a0	123
$a1	321
$v0	444

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (10/12)



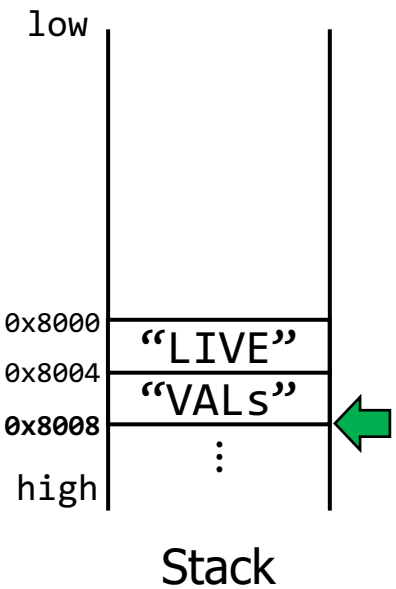
PC	0x201C
$ra	0x2018
$sp	0x8000
$t0	"LIVE"
$t1	"VALs"
$a0	123
$a1	321
$v0	444

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2038   ...
```

MIPS assembly code

Example: Register Spilling (11/12)



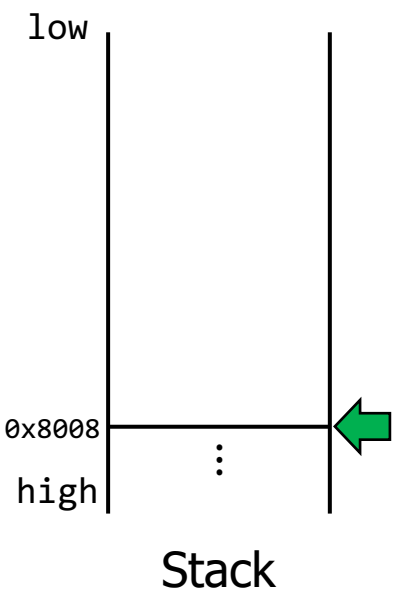
PC	0x2020
$ra	0x2018
$sp	0x8008
$t0	"LIVE"
$t1	"VALs"
$a0	123
$a1	321
$v0	444

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2024   ...
```

MIPS assembly code

Example: Register Spilling (12/12)



PC	0x2024
$ra	0x2018
$sp	0x8008
$t0	"LIVE"
$t1	"VALs"
$a0	123
$a1	321
$v0	444

Registers

```
0x1FFC   ...
0x2000   addi $sp, $sp, -8
0x2004   sw $t1, 4($sp)
0x2008   sw $t0, 0($sp)

0x200C   addi $a0, $zero, 123
0x2010   addi $a1, $zero, 321
0x2014   jal complexFunction

0x2018   lw $t1, 4($sp)
0x201C   lw $t0, 0($sp)
0x2020   addi $sp, $sp, 8
0x2024   ...
```

MIPS assembly code

Nested Procedures

- If a callee function invokes **another function**,
  - `$ra` gets overwritten by the `jal` instruction.
  - The argument registers, `$a0` ~ `$a3`, get overwritten.
- **Spill** `$ra` and `$a0` ~ `$a3` along with other registers!
  - The caller spills the argument & temporary registers.
    - Up to 16 bytes for `$a0`~`$a3`, up to 40 bytes for `$t0`~`$t9`
  - The callee spills the return address and saved registers.
    - 4 bytes for `$ra`, up to 32 bytes for `$s0`~`$s7`

Preserved	Not preserved
Saved registers: `$s0`–`$s7`	Temporary registers: `$t0`–`$t9`
Stack pointer register: `$sp`	Argument registers: `$a0`–`$a3`
Return address register: `$ra`	Return value registers: `$v0`–`$v1`
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

Allocating Space on the Stack

- The segment of the stack containing a function's **saved registers** and **local variables**
  - e.g., local variables whose sizes are larger than registers
  - a.k.a. **activation record** (in the field of compilers)
- **Frame pointer** (**`$fp`**)
  - Point to the first word of the frame
  - Used as a stable base register within a function for local memory references

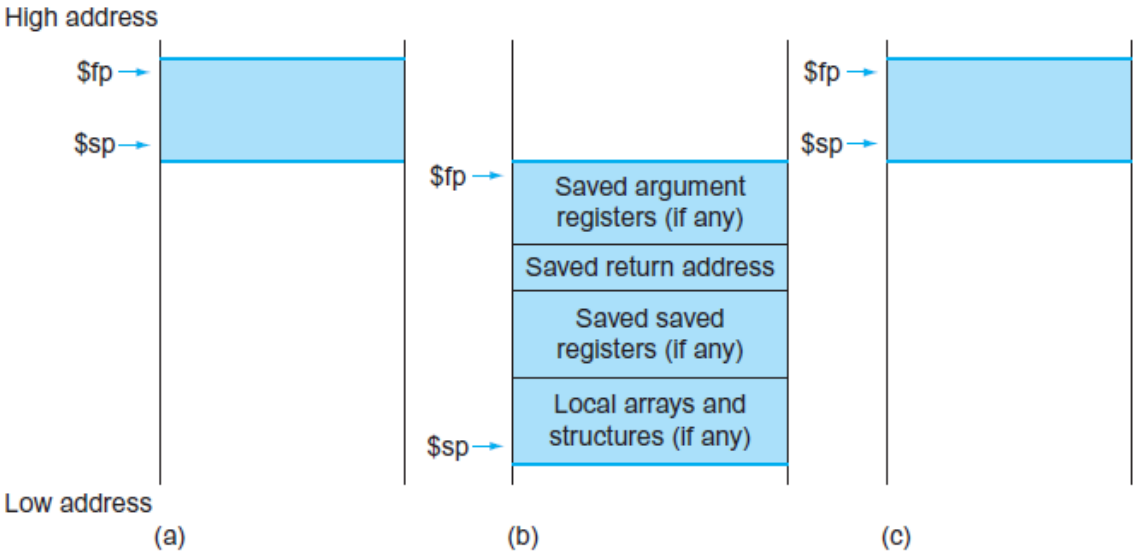
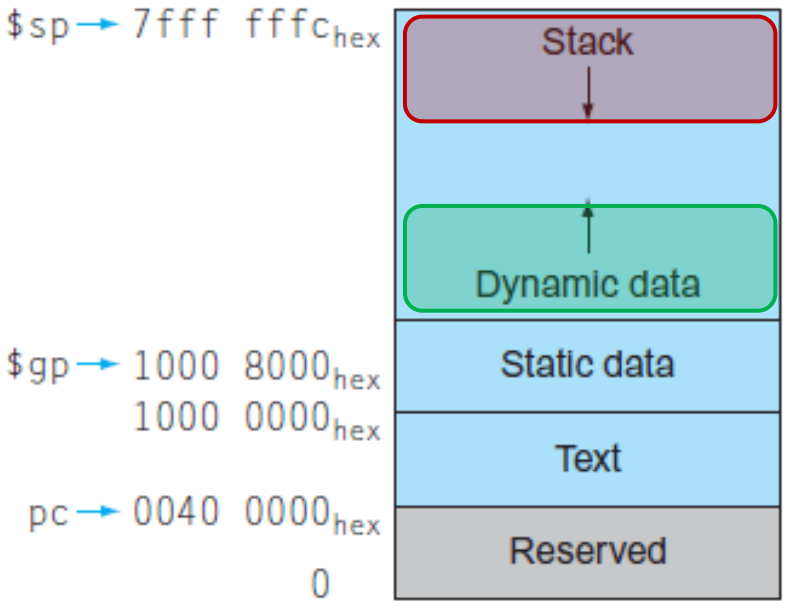


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack.

Allocating Space on the Heap

- The heap supports **dynamic** memory allocations.
  - e.g., `malloc()` and `free()` in C

- The stack and the heap **grow toward each other**!
  - The stack grows from higher addresses to lower addresses.
  - The heap grows from lower addresses to higher addresses.



MIPS Register Conventions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.12), and registers 26–27, called $k0–$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

MIPS Addressing

- **Addressing**
  - Any of several methods of locating and accessing information within storage

- Required by many critical execution scenarios
  - Handling >16-bit immediate operands
  - Locating branch/jump targets
  - Accessing the main memory

Addressing: 32-bit Immediates

- **I-type** instructions support **16-bit** immediates.
- What if an immediate is larger than 16 bits?
  - e.g., 32-bit immediate operands

- **Load upper immediate** (`lui`) instruction
  - **Set the upper 16 bits** of a constant in a register
  - e.g., Assign a value of $003D0900_{(16)}$ to `$s0`
    - **`lui $s0, 61`** followed by **`ori $s0, $s0, 2304`**
      - Let's assume that the initial value of `$s0` is 0.
      - Executing the `lui` instruction results in `$s0` = **003D**$0000_{(16)}$.
      - Executing the `ori` instruction results in `$s0` = 003D**0900**$_{(16)}$.
  - Compilers/assemblers must break large constants into pieces, and then reassemble them into a register!

Addressing: Branches & Jumps

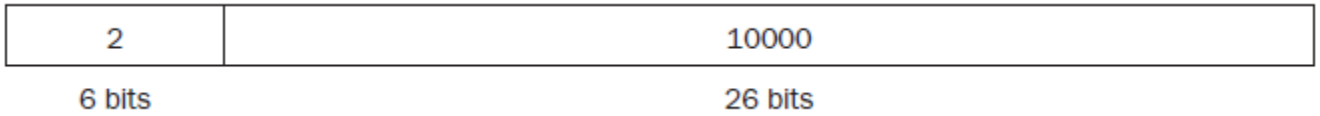
- Jumps utilize the **J-type** instruction format.
  - 6-bit operation field + 26-bit address field
  - e.g., `j 10000`
    - Go to location 10000
    -

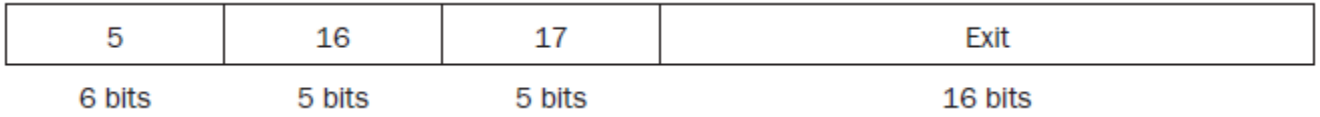
2	10000
6 bits	26 bits

- Branches use the **I-type** instruction format.
  - Specify two operands for tests (e.g., equality)
  - e.g., **bne** `$s0, $s1, Exit`
    - Go to Exit if `$s0 != $s1`
    -

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

Addressing: Branches & Jumps

- Jumps use the **J-type** instruction format.
  - 6-bit operation + **26-bit** address (**2^{26}** addresses)
  - e.g., `j 10000`
    - Go to location 10000
    - 

- Branches use the **I-type** instruction format.
  - **16-bit** immediate as address (**2^{16}** addresses)
  - e.g., **bne** `$s0, $s1, Exit`
    - Go to Exit if `$s0 != $s1`
    - 

PC-Relative Addressing (1/2)

- Storing addresses in the 16-bit field for branches restricts the range of target addresses.
  - **No program can be bigger than 2^{16}** if addresses of a program must fit in the 16-bit field.

- **Register-relative** addressing
  - Calculate a branch instruction as:

    Program Counter = **Register + Branch Address**
  - Allow the program to be as large as 2^{32}
  - Can use conditional branches

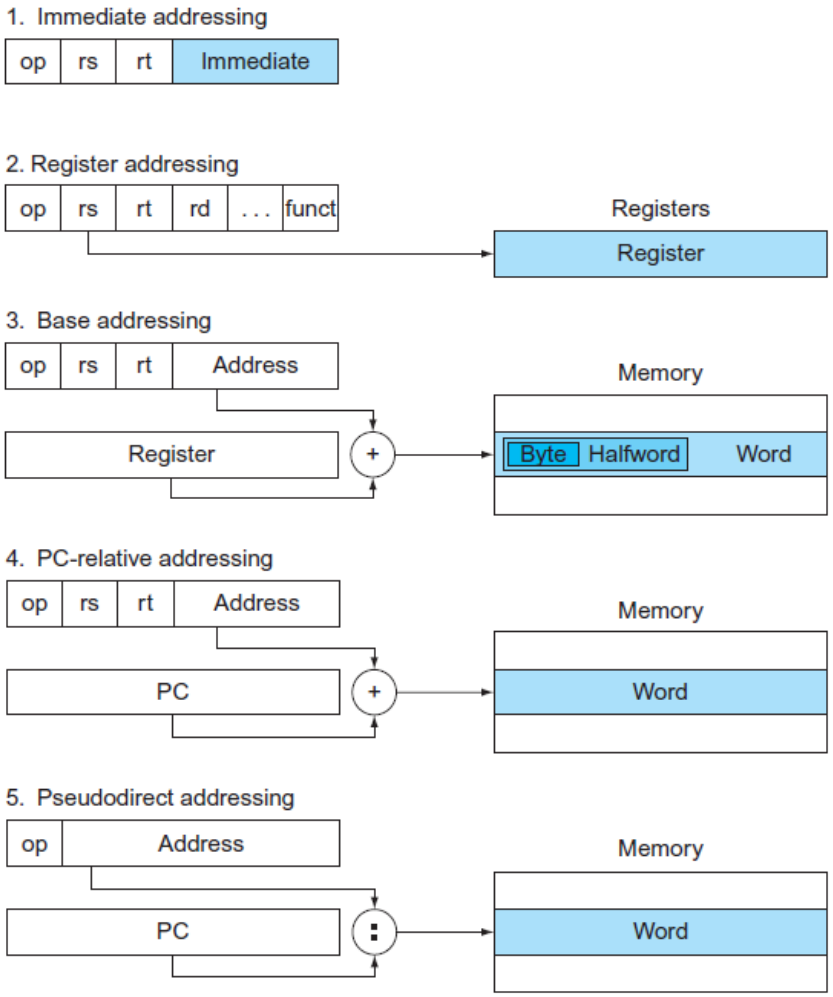
Q: Which register should we use as the base register?

PC-Relative Addressing (2/2)

- **Motivation**: the nature of conditional branches
  - Conditional branches are used in loops and `if` statements.
  - They tend to branch to a **nearby** instruction.
    - e.g., About 50% of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away.
  - Branch target addresses of almost all loops and `if` statements are much smaller than 2^{16} words.
- Thus, let's use **$pc** as the base register!
  - `$pc` stores the address of the **current** instruction.
  - In reality, however, the base address becomes (**$pc + 4**).
  - Also, MIPS interprets the 16-bit field as **word**s, not bytes.
    - 16-bit byte: $-2^{15} \sim 2^{15}-1$, 16-bit word: $-2^{17} \sim 2^{17}-4$

MIPS Addressing Modes

- Immediate
  - The operand is a constant.
- Register
  - The operand is a register.
- Base/Displacement
  - The operand is at memory[reg+constant].
- PC-relative
  - Branch to PC+constant
- Pseudodirect
  - Jump to {$pc[31:26], address[25:0]}



Decoding the Machine Language

- Used by compilers/assemblers and for reverse-engineering MIPS executables

- (Typical) Procedure
  - Obtain the 6-bit op field from a 32-bit instruction
  - Identify the format (i.e., R, I, J) of the instruction
  - Split the 32-bit instruction into the format's fields
  - Calculate the values of the identified fields

- Heavily rely on the MIPS **ISA manual**

MIPS Instruction Encoding

- Figure 2.19 in the textbook

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

FIGURE 2.19 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, the top portion of the figure shows **load word** in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence, subtract in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010_{two} and the op field (bits 31–26) is 000000_{two}. The floating point value in row 2, column 1 is defined in Figure 3.18 in Chapter 3. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. This chapter describes instructions given in full name using color, while Chapter 3 describes instructions given in mnemonics using color. Appendix A covers all instructions.

- Show a mapping from the op field's value to the corresponding instruction and its format

Example: Decoding $00af8020_{(16)}$

- Obtain the 6-bit op field of the instruction

$$00af8020_{(16)} = \boxed{000000}0010101111100000000100000_{(2)}$$

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

Example: Decoding $00af8020_{(16)}$

- Split the instruction using the R-type format

  $00000000101011111000000000100000_{(2)}$

→ $000000\ 00101\ 01111\ 10000\ 00000\ \mathbf{100000}_{(2)}$

op rs rt rd shamt **funct**

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Example: Decoding $00af8020_{(16)}$

- Decode each field of the instruction

```
000000 00101 01111 10000 00000 100000(2)
op     rs    rt    rd    shamt funct
       $a1   $t7   $s0
```

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

```
add $s0, $a1, $t7
```

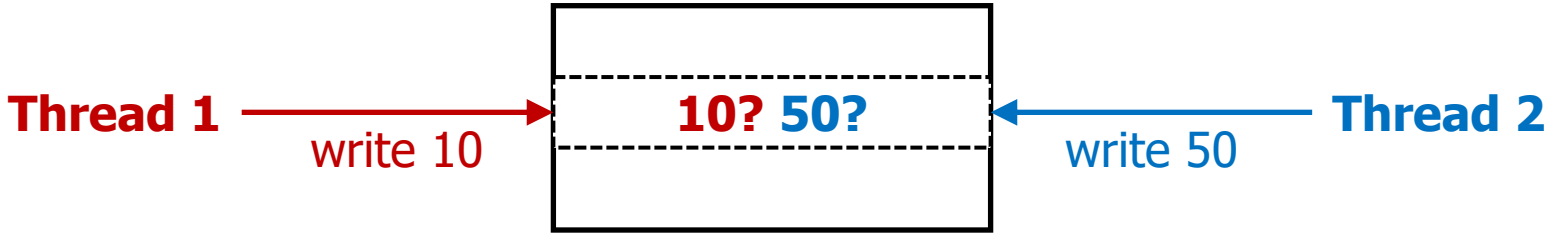
Supporting Parallelism

Instructions for Synchronization

Parallelism & Data Races

- **Parallelism** improves performance by executing multiple threads/processes in parallel.
  - e.g., perform n-by-m matrix multiplication using 4 threads

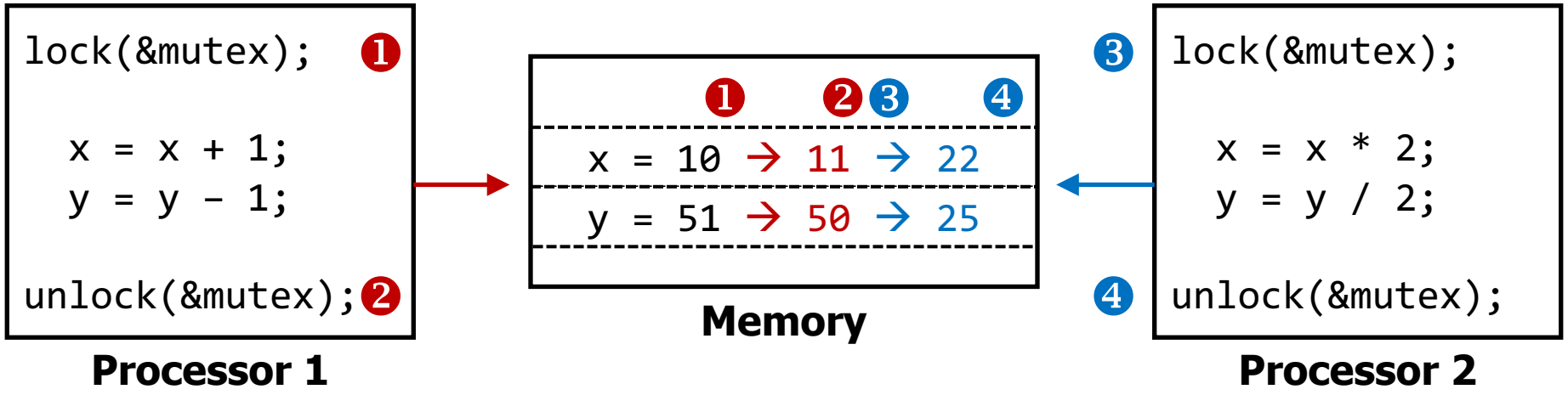
- **Data races** can occur!
  - Two memory accesses from different threads, and at least one is a write
  - Execution results can differ between program executions!



Lock & Unlock Synchronization

- Lock-unlock creates a **mutual exclusion**.
  - Prevent accesses from different processors

- Within a mutual-exclusion region, a processor **atomically** reads from and/or writes to memory.



MIPS: Atomic Instructions

- Basic **synchronization primitives** allow programs to acquire and release a lock.
  - e.g., single atomic exchange/swap operations

- Instead, MIPS uses a **pair of instructions**.
  - **Load linked** (`ll`) instruction
    - Load the value from the target memory address to a register
    - **Register a reservation** on the target memory address
  - **Store conditional** (`sc`) instruction
    - Store the value of a register in memory
    - Change the value of the register to 1 if it succeeds, and to a 0 if it fails.

MIPS: `ll` & `sc` Instructions

- If the contents of the memory location specified by `ll` changes before `sc` occurs, then `sc` fails and does not write to the memory.

- **Example**: an atomic exchange operation on the memory location specified by register `$s1`

```
again: addi $t0,$zero,1       ;copy locked value
       ll   $t1,0($s1)        ;load linked
       sc   $t0,0($s1)        ;store conditional
       beq  $t0,$zero,again   ;branch if store fails
       add  $s4,$zero,$t1     ;put load value in $s4
```

  - `ll` instruction: `$t1 ← memory[$s1]`
  - `sc` instruction: `memory[$s1] ← $t0`
  - `$t0 = 1` if success, `0` if failure

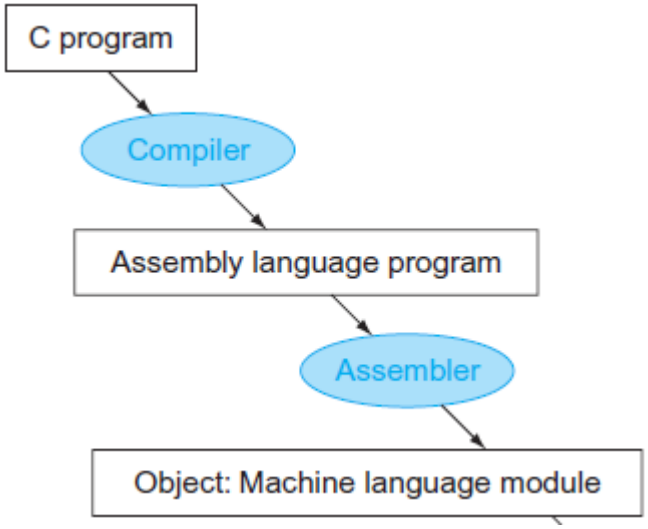
Program-to-Binary

Translating and Starting a Program on MIPS CPUs

Translating a Program

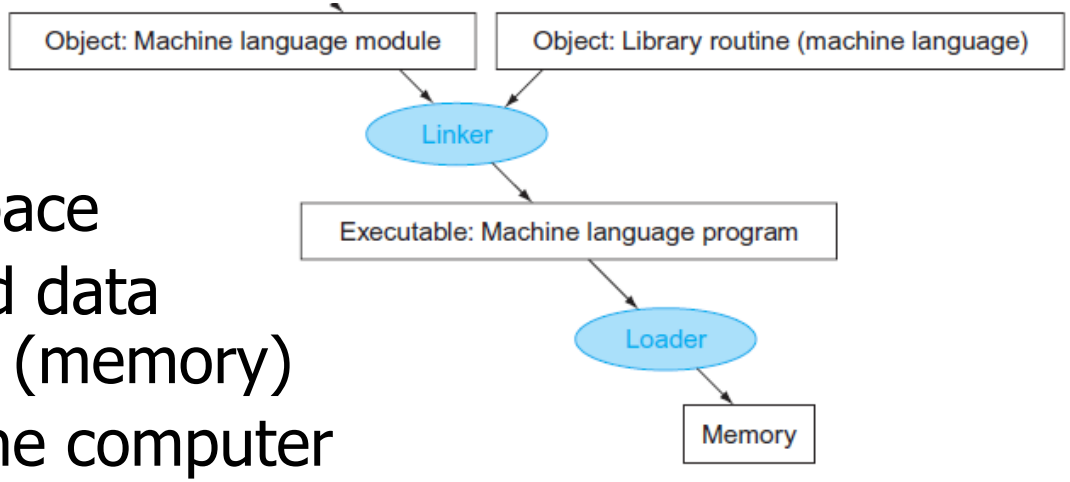
- Executing a C program on a MIPS processor

- **Compiler**
  - C language
    → assembly language
- **Assembler**
  - Assembly language
    → machine language
  - e.g., `li $t0, 123`
    → `addi $t0, $zero, 123`
  - Utilize the **symbol table**



Starting a Program

- **Linker**
  - Link all the independently assembled machine language programs into one large machine language program
    - e.g., resolve `libc` library function calls
  - Produces an **executable** which can run on a computer
- **Loader**
  - Read the executable
  - Create an address space
  - Copy instructions and data to the address space (memory)
  - Initialize & instruct the computer to perform the instructions



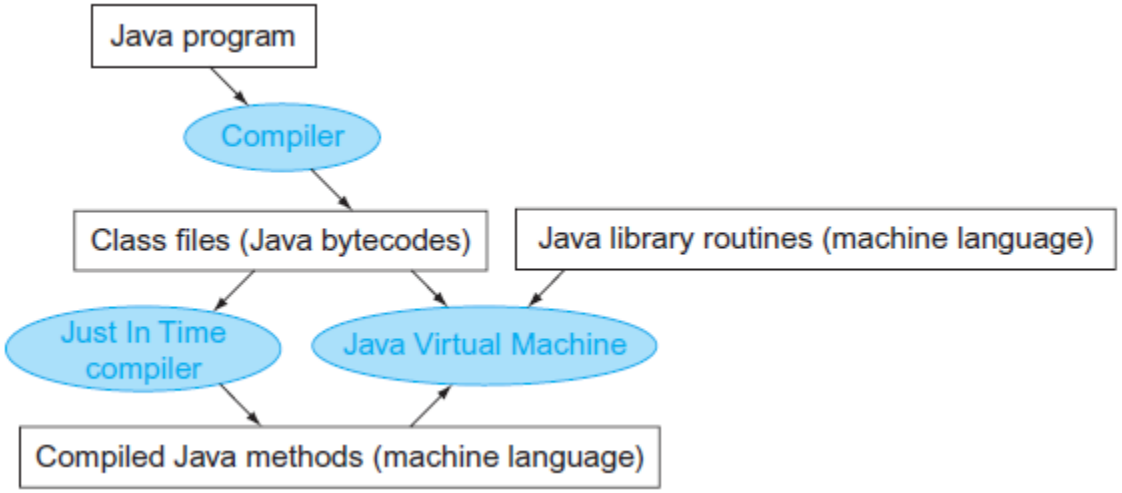
Dynamically Linked Libraries

- Linking libraries before executing a program
  - The library routines become part of the executable.
  - It loads all routines in the library which are called anywhere in the executable, even if not necessary.

- **Dynamically Linked Libraries (DLLs)**
  - The library routines are not linked & loaded **until the program is run**.
  - Enable **lazy** linking & loading of library functions
    - Perform linking & loading of library functions' machine code at the time of the first invocation

Starting a Java Program

- Java employs **Just-In-Time (JIT)** compilation.
  - The machine code gets generated during runtime by a **Java Virtual Machine (JVM)**.



- Achieve **high portability**, but **low performance**

Putting It All Together (1/4)

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

A C function which swaps two locations in memory

- Converting C code to MIPS assembly code
  - Allocate registers to program variables
  - Produce code for the body of the function
  - Preserve registers across function invocations

Putting It All Together (2/4)

```
void swap(word $a0, word $a1) {
  word $t0;
  $t0 = memory[$a0 + 4 * $a1];
  memory[$a0 + 4 * $a1] = memory[$a0 + 4 * $a1 + 4];
  memory[$a0 + 4 * $a1 + 4] = $t0;
}
```

- **Allocate registers to program variables**
  - **`$a0`** for **`v`**, **`$a1`** for **`k`**, and **`$t0`** for **`temp`**
  - Don't forget that memory addresses differ by **4**!

Putting It All Together (3/4)

```
void swap(word $a0, word $a1) {
  sll $t1, $a1, 2   // $t1 = $a1 << 2 = $a1 * 4
  add $t1, $a0, $t1 // $t1 = $a0 + $t1 = $a0 + 4 * $a1

  lw $t0, 0($t1)    // $t0 = memory[$t1] = v[k]
  lw $t2, 4($t1)    // $t2 = memory[$t1 + 4] = v[k + 1]

  sw $t2, 0($t1)    // memory[$t1] = $t2 = v[k + 1]
  sw $t0, 4($t1)    // memory[$t1 + 4] = $t0 = v[k]

  jr $ra
}
```

- **Produce code for the body of the function**
  - **$t1** for the target memory address,
    $t2 for storing the value of `v[k + 1]`

Putting It All Together (4/4)

```
swap:
  sll $t1, $a1, 2   // $t1 = $a1 << 2 = $a1 * 4
  add $t1, $a0, $t1 // $t1 = $a0 + $t1 = $a0 + 4 * $a1
  lw $t0, 0($t1)    // $t0 = memory[$t1] = v[k]
  lw $t2, 4($t1)    // $t2 = memory[$t1 + 4] = v[k + 1]
  sw $t2, 0($t1)    // memory[$t1] = $t2 = v[k + 1]
  sw $t0, 4($t1)    // memory[$t1 + 4] = $t0 = v[k]
  jr $ra
```

- **Preserve registers across function invocations**
  - No need to preserve any registers

Summary: MIPS ISA So Far

- Refer to Figure 2.44 and Appendix A for more details!

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 2.44 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right. Appendix A (Section A.10) describes the full MIPS architecture. Figure 2.1 shows more details of the MIPS architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the MIPS Reference Data Card at the front of the book.