Chapter 3
MIPS Assembly Language

Review

- MIPS instruction:fixed instruction size(32bit) and 3 simple formats
- bne or beq: R type or I-type?
- Using slt and beq to simulate ‘branch if less than’
- pseudo instruction

Compiling a While Loop

- **C Source Code:**

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

assuming that I,j,k corresponds to $s3,$s4,$s5 and the base address of the array save is in $s6 (How is this example different from the previous ones?)

- **MIPS assembly code:**

```
Loop: add $t1,$s3,$s3   #reg $t1 = 2*i
      add $t1,$t1,$t1   #reg $t1 = 4*i
      add $t1,$t1,$s6   #$t1 = address of save[i]
      lw $t0,0($t1)     #$t0 = save[i]
      bne $t0,$s5, Exit # goto Exit if save[i]!=k
      add $s3,$s3,$s4   # i= i + j
      j Loop            # goto Loop
Exit:
```

Another Example

- See page 126 of text.
- The loop modifies I, we must multiply its value by 4 each time through the loop.
- Exists a more efficient method (See section 3.11, the pointer version)

Case/Switch Statement

- **C source code:**

```
switch(k){
case 0: f=i+j;break;
case 1: f=g+h;break;
case 2: f=g-h;break;
case 3: f=i-j;break;
}
```

- **What is the MIPS assembly code assuming f-k correspond to registers $s0-$s5 and $t2 contains 4 and $t4 contains base address of JumpTable?**

MIPS Assembly Code for Case/Switch

```
Slt   $t3,$s5,$zero     # test if k<0
bne   $t3,$zero,Exit    # go to Exit if k<0
slt   $t3,$s5,$t2       # test if k<4
beq   $t3,$zero,Exit    # go to Exit if k>=4
add   $t1,$s5,$s5       #$t1 =2*k
add   $t1,$t1,$t1       #$t1 =4*k
add   $t1,$t1,$t4       #$t1=address of JumpTable[k]
lw    $t0,0($t1)        #$to=JumpTable[k]
jr    $t0               #jump based on register $t0
L0:   add $s0,$s3,$s4
      j Exit
L1:   add $s0,$s1,$s2
      j Exit
L2:   sub $s0,$s1,$s2
      j Exit
L3:   sub $s0,$s3,$s4
Exit:
```

Supporting Procedures

- Basic steps:
  - Place parameters in a place where the procedure can access them
  - Transfer control to the procedure
  - Acquire the storage resources needed for the procedure
  - Perform desired task
  - Place the result in a place where the calling program can access it
  - Return control to the point of origin

Registers for Procedure Calling

- **$a0-$a3: four argument registers**
- **$v0-$v1: two value registers**
- **$ra: return address register**

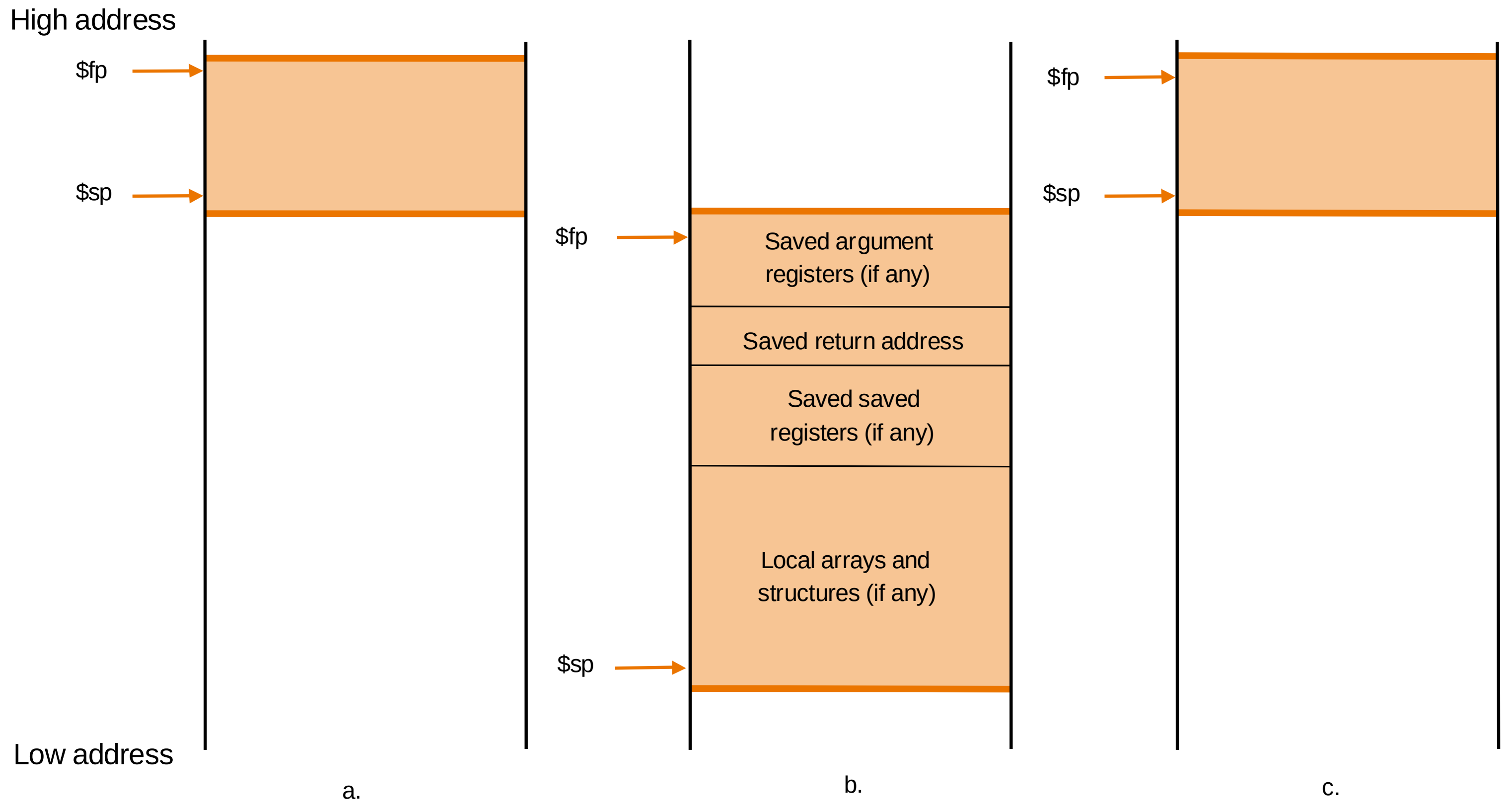
- **jump-and-link:** `jal ProcedureAddress`
- `jal` **instruction actually saves PC+4 in the register $ra**
- **return jump:** `jr $ra`

Using more registers

- What if more than four arguments and two return values are needed?
- Spill register to memory
- use a stack data structure (last-in-first-out) to do this
- that's why there is another register called $sp (stack pointer)
- Example on page 134 of text shows how a procedure call works.
- Nested procedures.

Procedure Frame

- **The segment of the stack containing a procedure's saved registers and local variables is called a procedure frame.**
- **Some MIPS software uses a frame pointer ($fp) to point to the first word of the frame of a procedure. (more stable)**



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.

MIPS Register Convention

Name	Reg#	Usage	Preserved on call?
`$zero`	0	the constant value 0	n.a.
`$v0-$v1`	2-3	values for results	no
`$a0-$a3`	4-7	arguments	yes
`$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

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

Beyond Numbers

- **ASCII code**
- **Loading and saving bytes:**

```
lb $t0,0($sp)  # Read byte from source
sb $t0,0($gp)  # Write byte to dest.
```

- **Example: strcpy (page 143)**

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:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **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:**

I	op	rs	rt	16 bit address

- **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)**
- **Jump instructions just use upper 4 bits of PC**
  - **address boundaries of 256 MB**

To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

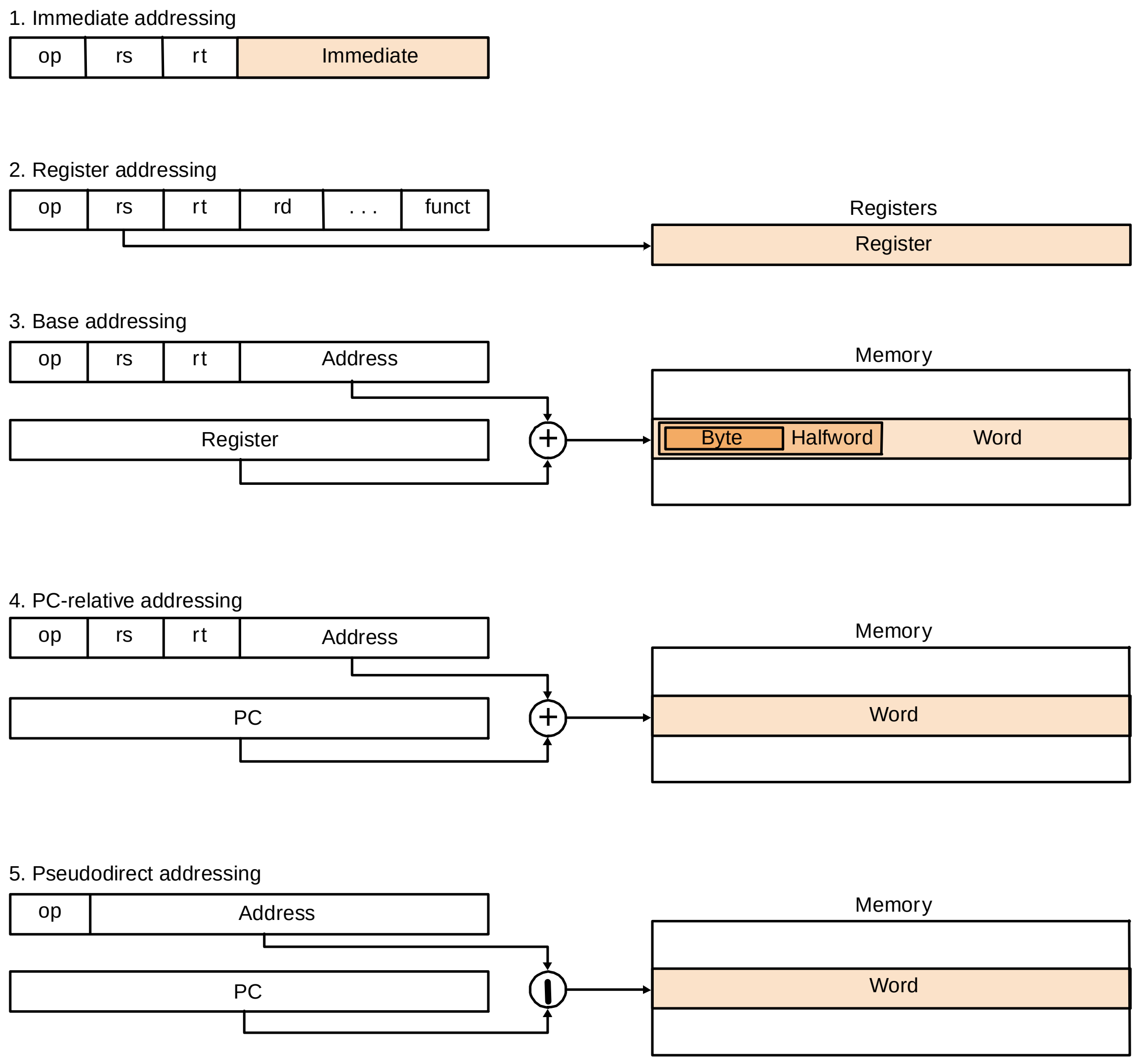
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

MIPS Addressing Mode

- Register addressing: operand is a register
- Base or displacement addressing: example: lw $t0,1200($t1)
- Immediate addressing: addi
- PC-relative addressing: address is the sum of the PC and a constant in the instruction (conditional branch)
- Pseudodirect addressing: the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC (jump)

MIPS Addressing Mode (Cont'd)



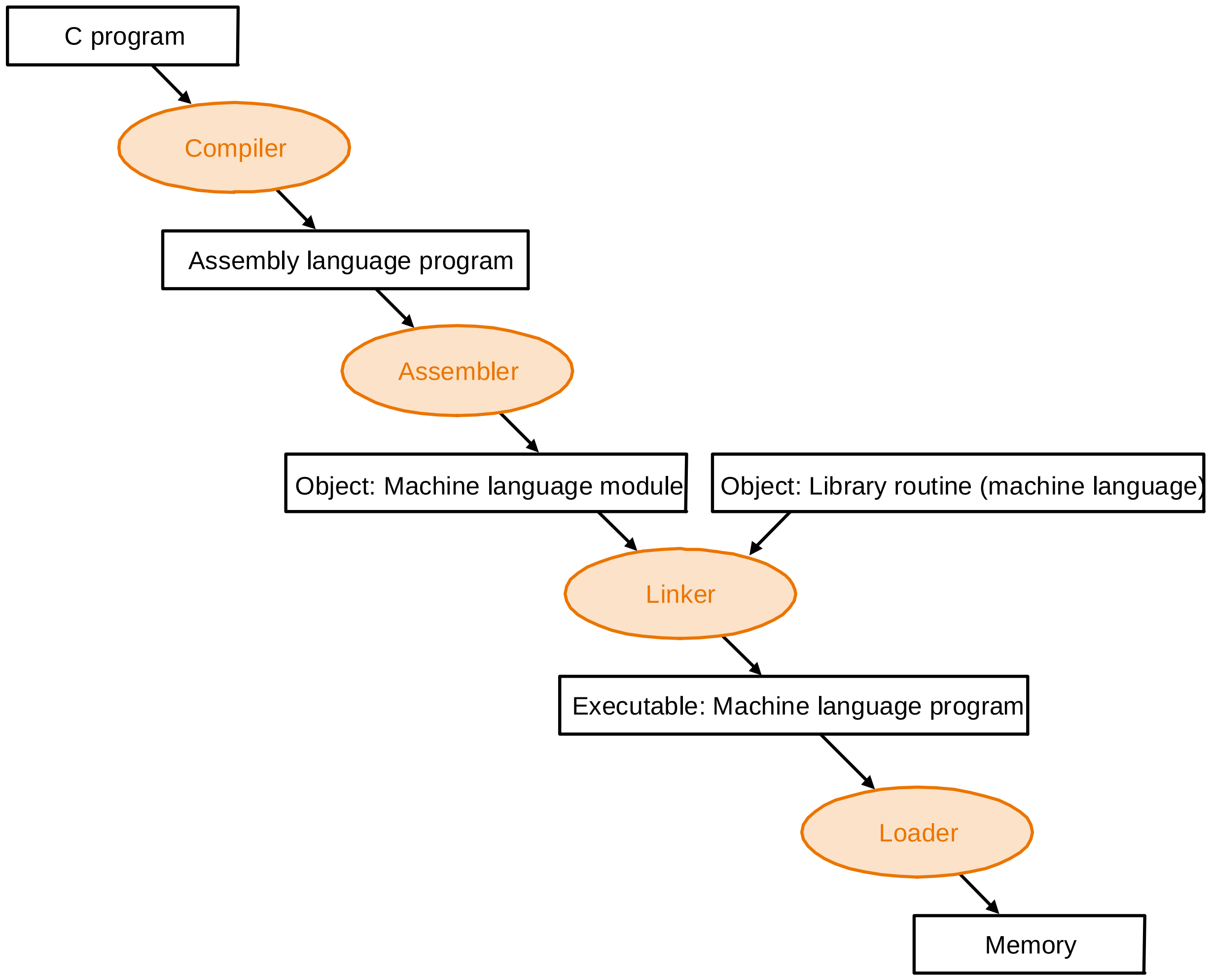
Decoding Machine Code

- With the help of Figure 3.18, you should be able to decode the following code:

  0000 0000 1010 1111 1000 0000 0010 0000

- add $s0,$a1,$t7

Starting a Program


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Using PCSPIM For Windows

- Messages: SPIM messages
- Text Segments (instruction)
- Data Segments: displays the data load to the program’s memory and data on the program’ stack.
- Registers
- Console