Introduction to Computer Organization and Architecture

Topic 3. Instruction set architecture (cont.)

Han, Nguyen Dinh (han.nguyendinh@hust.edu.vn)

Faculty of Mathematics and Informatics
Hanoi University of Science and Technology

Outline

1

Instructions for making decisions

Instructions for making decisions

Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS includes two decision-making instructions, similar to an *if* statement with a *go to*. The two instructions, called *conditional branches*, are as follows:

```
beq register1, register2, L1
```

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`

```
bne register1, register2, L1
```

The `bne` instruction means go to the statement labeled `L1` if the value in `register1` does not equal the value in `register2`

Instructions for making decisions

MIPS has another kind of branch, often called an *unconditional branch* or *jump*, abbreviated as j. For instance, the instruction `j Exit` means that go to the statement labeled Exit

Decisions are important both for choosing between two alternatives - found in *if* statements - and for iterating a computation - found in loops. The following code segment shows how a loop is implemented in MIPS using conditional and unconditional branches

```
Loop:   $t0, 0($t1)
        bne $t0, $s5, Exit
        j Loop
Exit:
```

Instructions for making decisions

Try to run

pieces of assembly codes

given in Section 2.7, Chapter 2 of the textbook!

2

Procedures and data

Supporting procedures in computer hardware

A *procedure* or *function* is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. In the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Acquire the storage resources needed for the procedure
4. Perform the desired task
5. Put the result value in a place where the calling program can access it
6. Return control to the point of origin, since a procedure can be called from several points in a program

Supporting procedures in computer hardware

In a computer, registers are the fastest place to hold data. So, we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

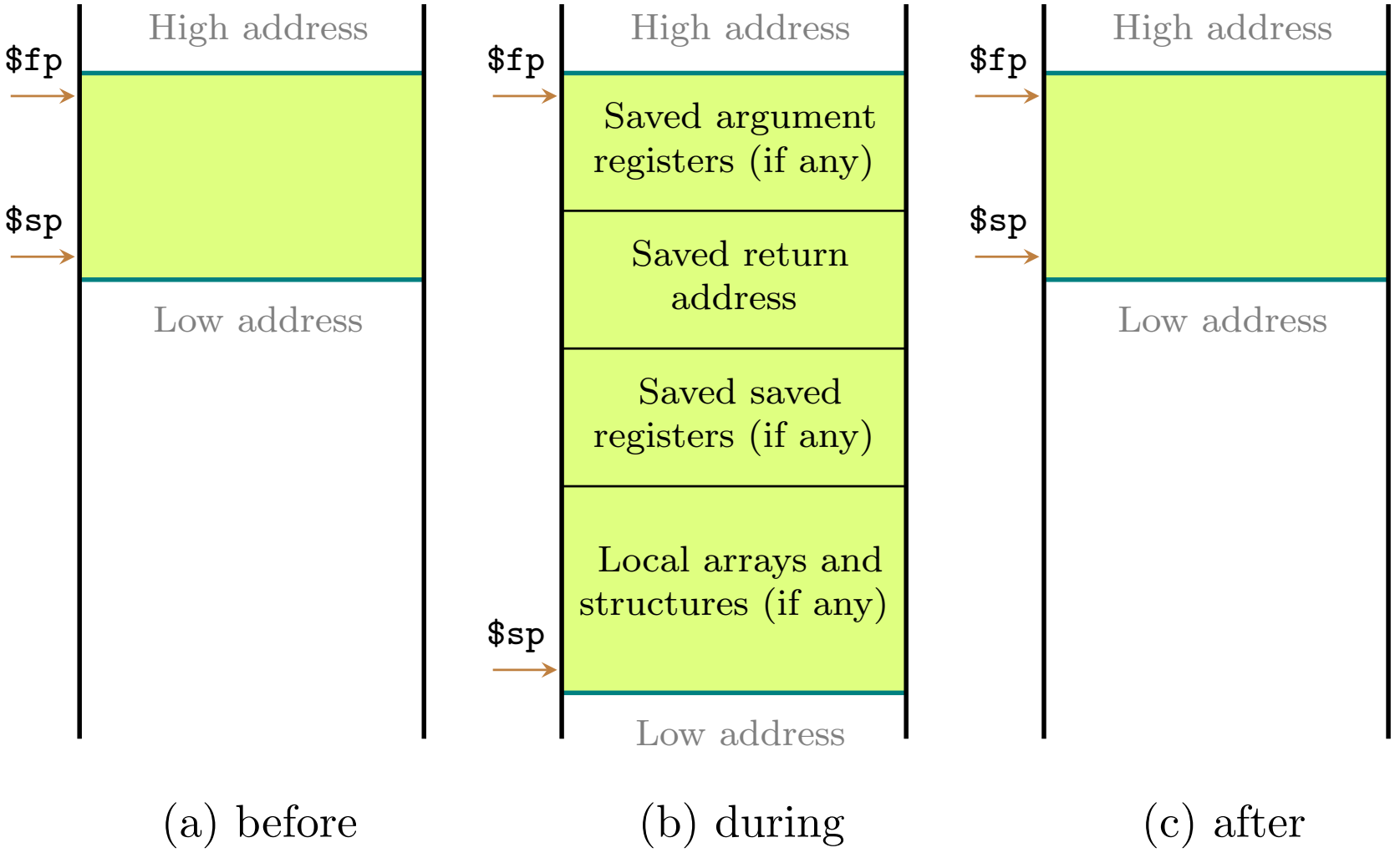
`$a0-$a3`: four argument registers in which to pass parameters

`$v0-$v1`: two value registers in which to return values

`$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: `jr $ra`. This jumps to an address and simultaneously saves the address of the following instruction in register `$ra`

Space allocation for a procedure call



(a) before (b) during (c) after

Procedures and data

Let's run

pieces of assembly codes

given in Section 2.8, Chapter 2 of the textbook!

3

Arrays and pointers

Arrays and pointers

Let's run

pieces of assembly codes

given in Section 2.14, Chapter 2 of the textbook!

4

Decoding machine language

MIPS addressing mode

1. Immediate addressing: the operand is a constant

6 bits 5 bits 5 bits 16 bits

Example:

Machine instruction: 0010 0011 1011 1101 0000 0000 0000 0100

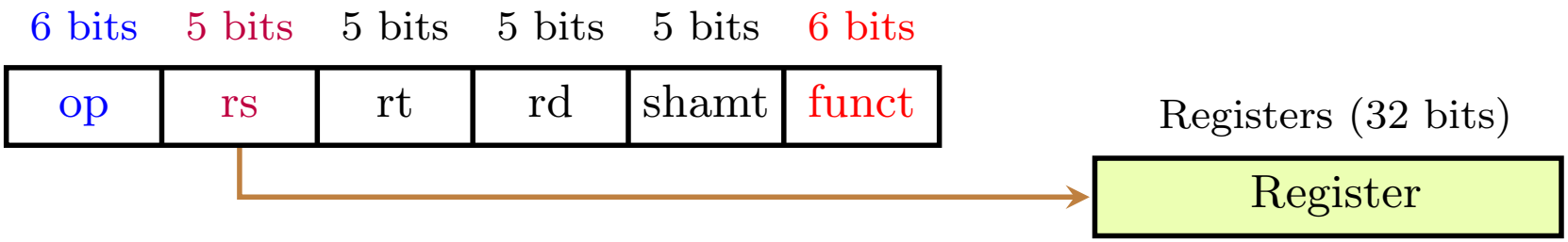
Decoding: $(001000_{op})(11101_{rs})(11101)(0000\ 0000\ 0000\ 0100)$

Assembly instruction: `addi $sp, $sp, 4`

NOTE ⇝ The instruction is in *I-format*. Please refer to Appendix A of the textbook for decoding MIPS instructions

MIPS addressing mode

2. Register addressing: the operand is a register



Example:

Machine instruction: 0000 0000 1010 1111 1000 0000 0010 0000

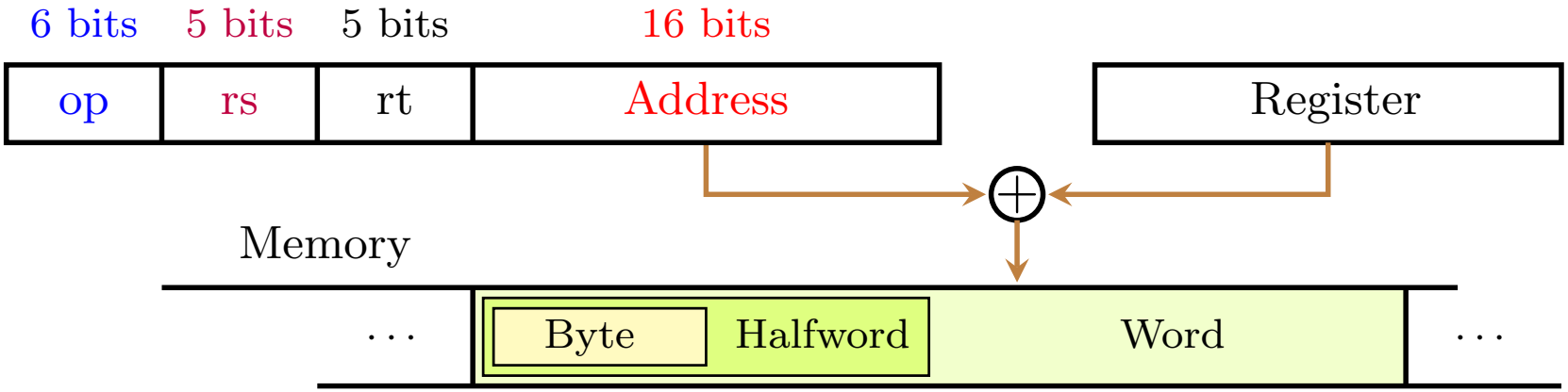
Decoding: $(000000_{op})(00101_{rs})(01111)(10000)(00000)(100000_{fn})$

Assembly instruction: `add $s0, $a1, $t7`

NOTE ⇝ The instruction is in *R-format*, its operand is shaded in color, and **`shamt`** is unused

MIPS addressing mode

3. Base addressing: the operand is at the memory location whose address is the sum of a register and a constant



Example:

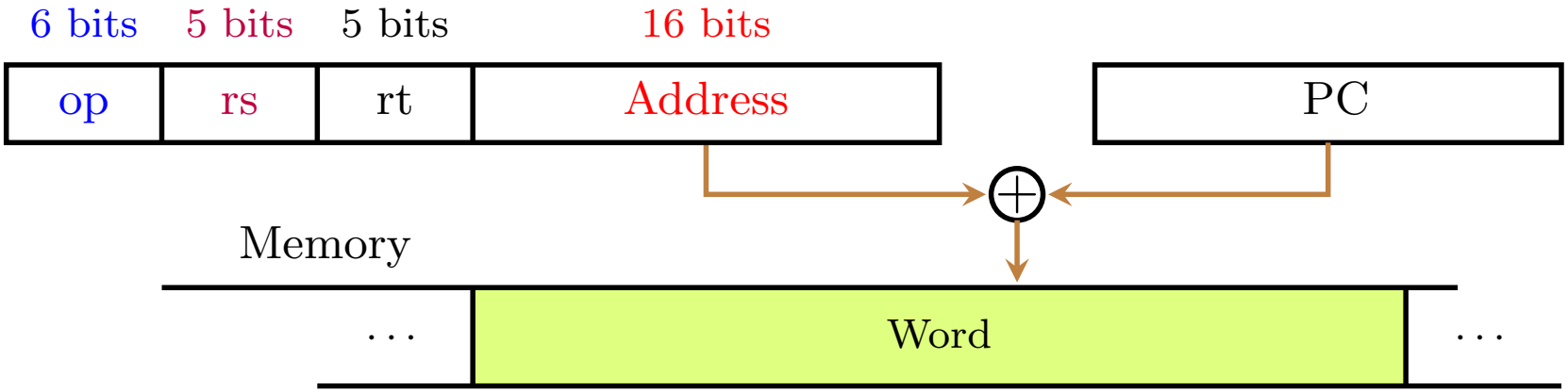
Machine instruction: 1000 1101 0010 1000 0000 0100 1011 0000

Decoding: $(100011_{op})(01001_{rs})(01000)(0000\ 0100\ 1011\ 0000)$

Assembly instruction: `lw $t0, 1200($t1)`

MIPS addressing mode

4. PC-relative addressing: the address is the sum of the PC and a constant in the instruction



Example:

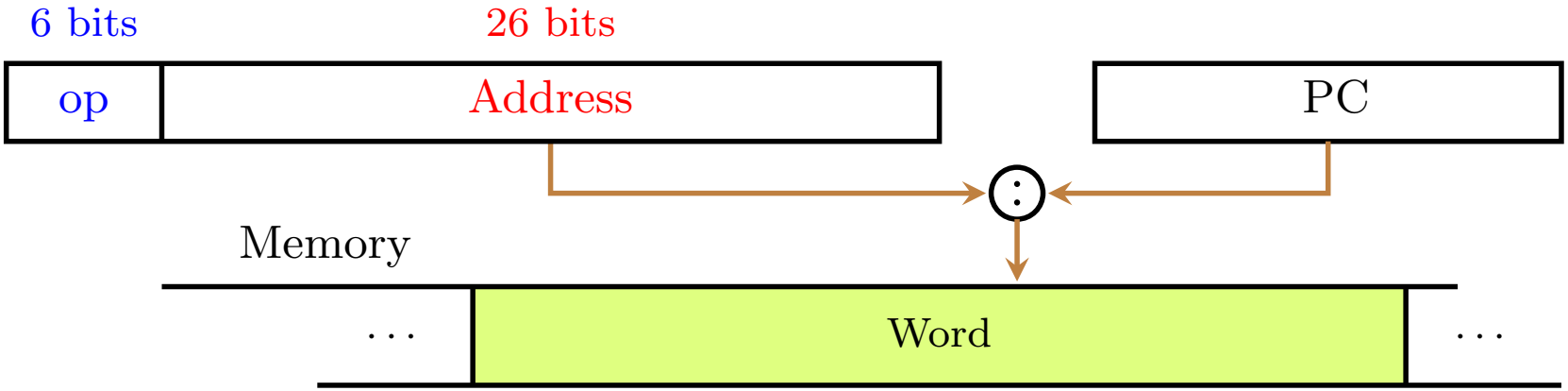
Machine instruction: 0001 0110 0011 0010 0000 0000 0001 1001

Decoding: $(000101_{op})(10001_{rs})(10010)(0000\ 0000\ 0001\ 1001)$

Assembly instruction: `bne $s1, $s2, 100`

MIPS addressing mode

5. Pseudodirect addressing: the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



Example:

Machine instruction: 0000 1000 0000 0000 0000 1001 1100 0100

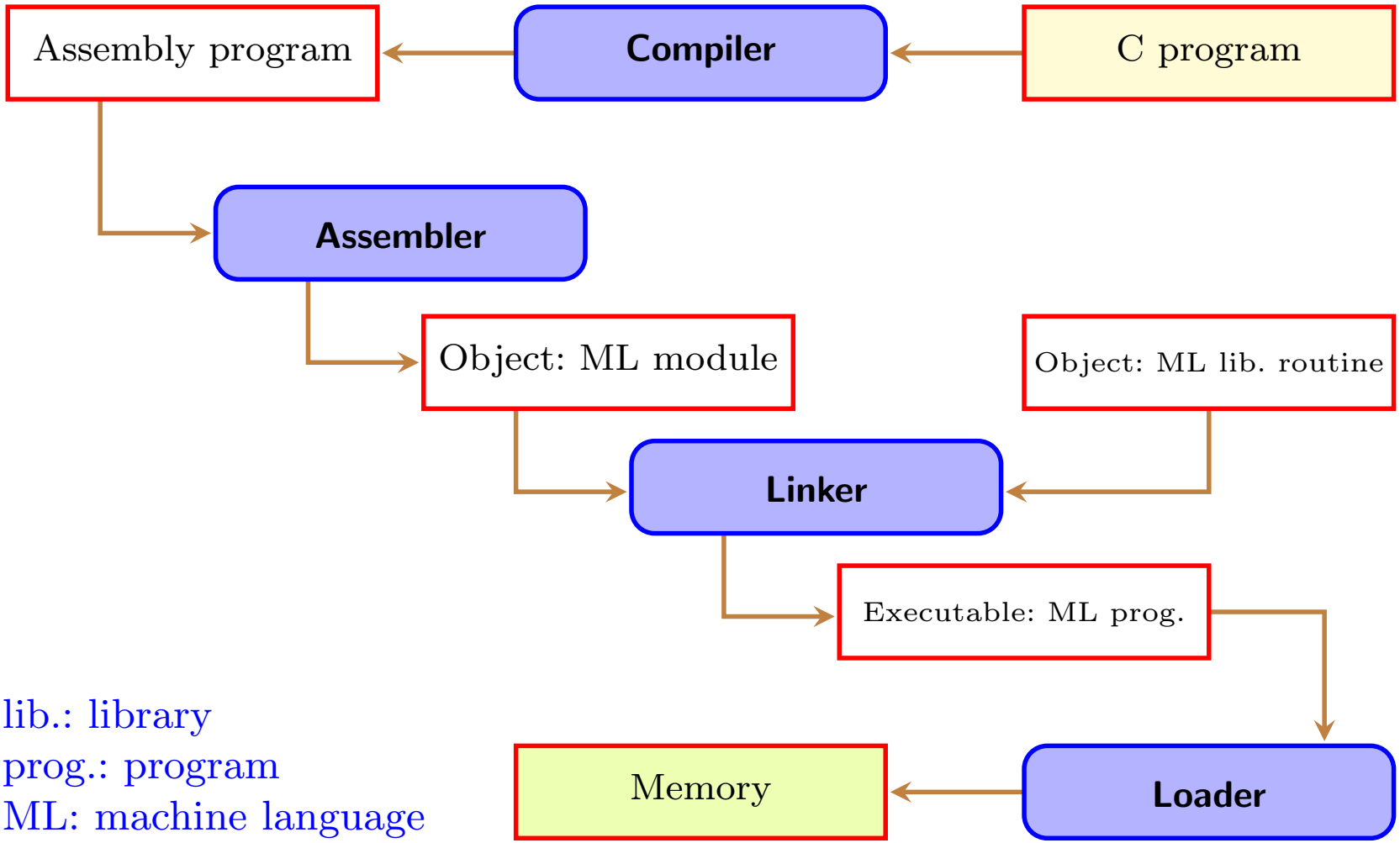
Decoding: (000010_{op})(00 0000 0000 0000 1001 1100 0100)

Assembly instruction: `j 10000`

5

Starting a program

Starting a program



lib.: library
prog.: program
ML: machine language

An example to put it all together

Try to run

the full procedure *sort*

on page 138 of the textbook!

Thank you very much for your attention!