ICT4
Computer Architecture Experiment
MIPS Laboratory

Overview

- I. Introduction to MIPS
- II. MIPS programming model
- III. MIPSIT user guide

I. Introduction to MIPS

- **MIPS** (originally an acronym for **Microprocessor without Interlocked Pipeline Stages**)
- MIPS is a RISC (Reduced Instruction Set Computer) instruction set architecture (ISA) developed by **MIPS Technologies** (formerly MIPS Computer Systems, Inc.)
- In 1981, a team led by John L. Hennessy at Stanford University started work on what would become the first MIPS processor.
- Multiple revisions of the MIPS instruction set exist, including MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32, and MIPS64.

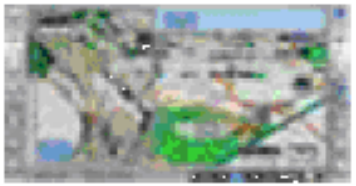
http://en.wikipedia.org/wiki/MIPS_architecture

Applications of MIPS processor

DVD players

Pioneer
DVR-57-H

Kenwood
HDV-810 Car Navigation System



Networking

3COM
3102 Business IP Phone

3COM
3106 Cordless Phone

Apple
Airport Extreme WLAN Access Points

Applications of MIPS processor

Portable Devices

Canon
EOS 10D Digital

JVC
GR-HD1

CPU
Type:MIPS R4000 32bit Core
Clockspeed:333 MHz

Sony Playstation PSX

CPU
Type:LSI/MIPS R3000A
Architecture:32 Bit
Clockspeed:33,8 MHz

Applications of MIPS processor

Residential and Small Office

Samsung
Digital Photo Frame

Sony
Media Server Vaio VGX-X90P

Pioneer
PureVisionÙ Plasma Television 43"
PureVisionÙ Plasma Television 50"

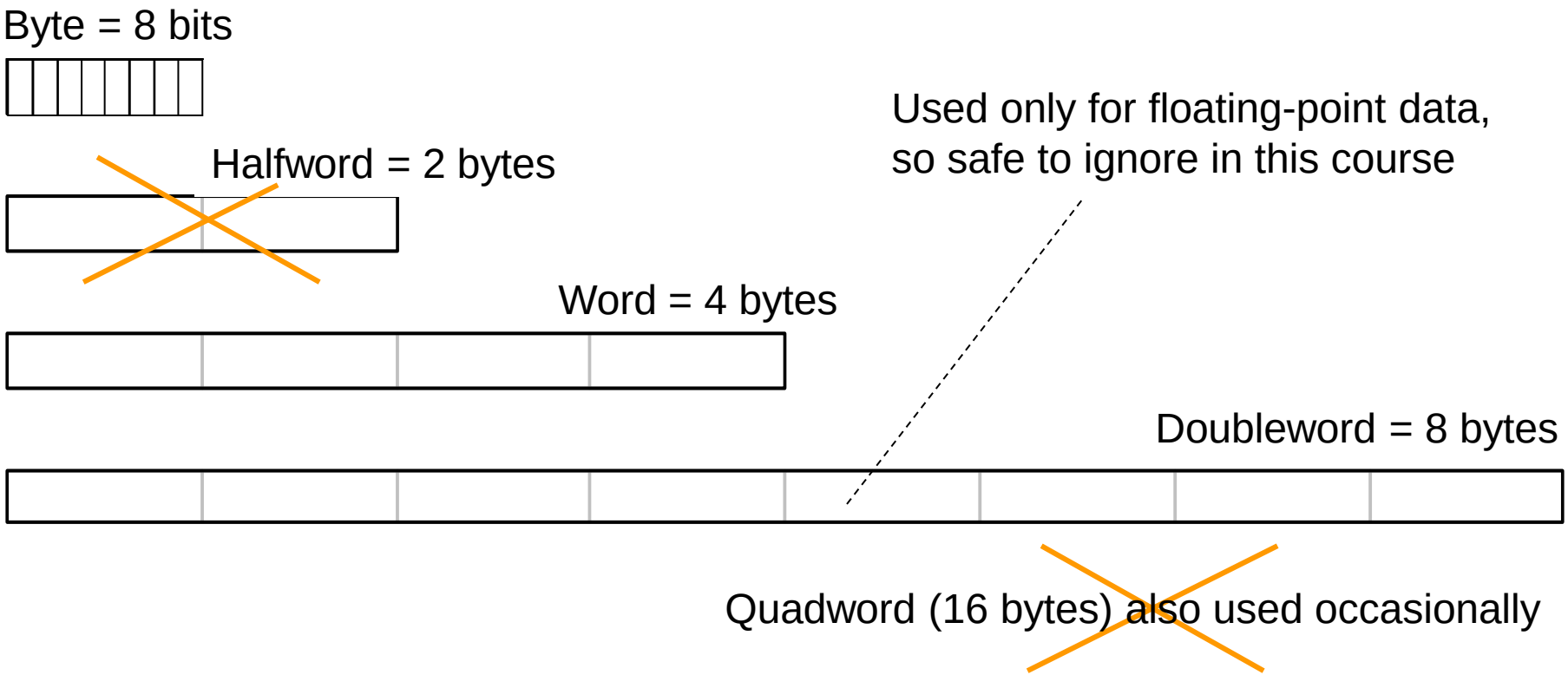
Sony
KDP-51WS550 High Definition TV
KDP-57WS550 High Definition TV
KDP-65WS550 High Definition TV

Hewlet Packard
Color Laser Jet 2500 Laser Printer

II. MIPS Programming Model

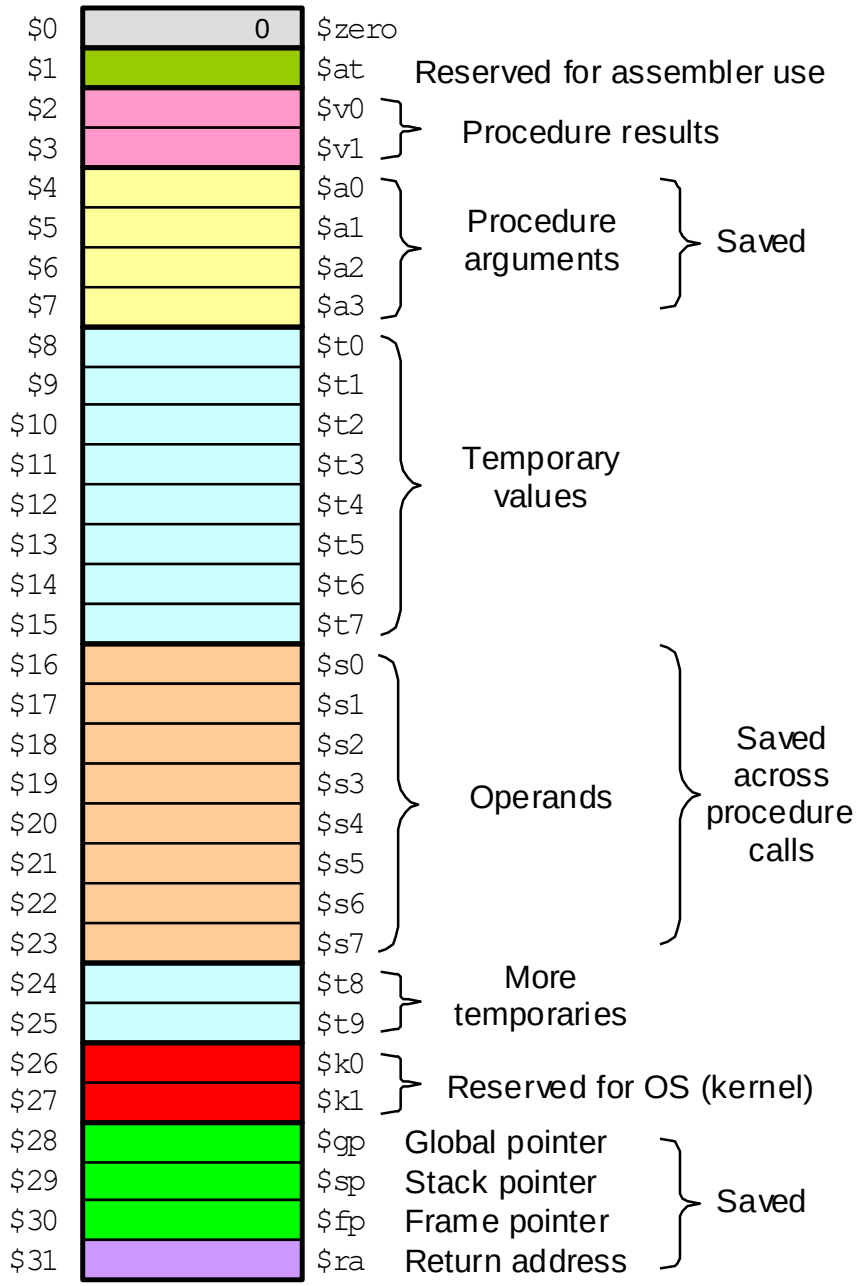
- Data Types
- Registers
- Instruction Formats
- MIPS Instruction
- Addressing Mode
- MIPS Assembly program
- MIPSIT

Data Types



MiniMIPS registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

Register Conventions



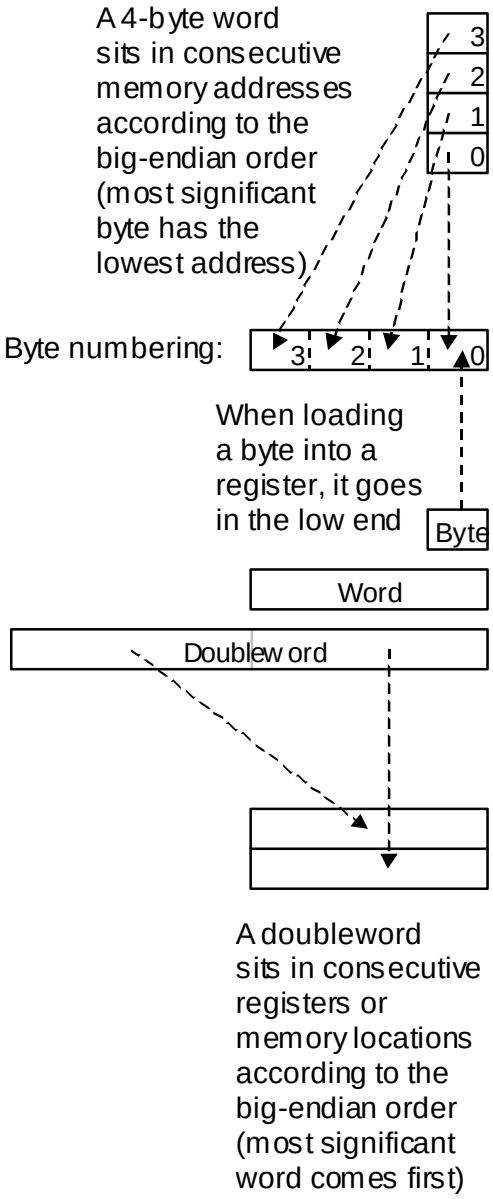


Figure 5.2 Registers and data sizes in MiniMIPS.

Instruction Formats

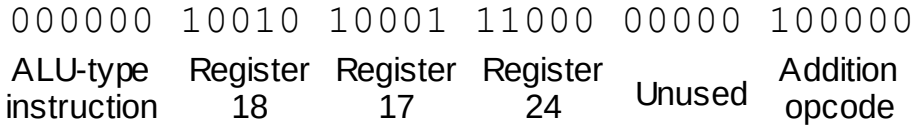
High-level language statement:

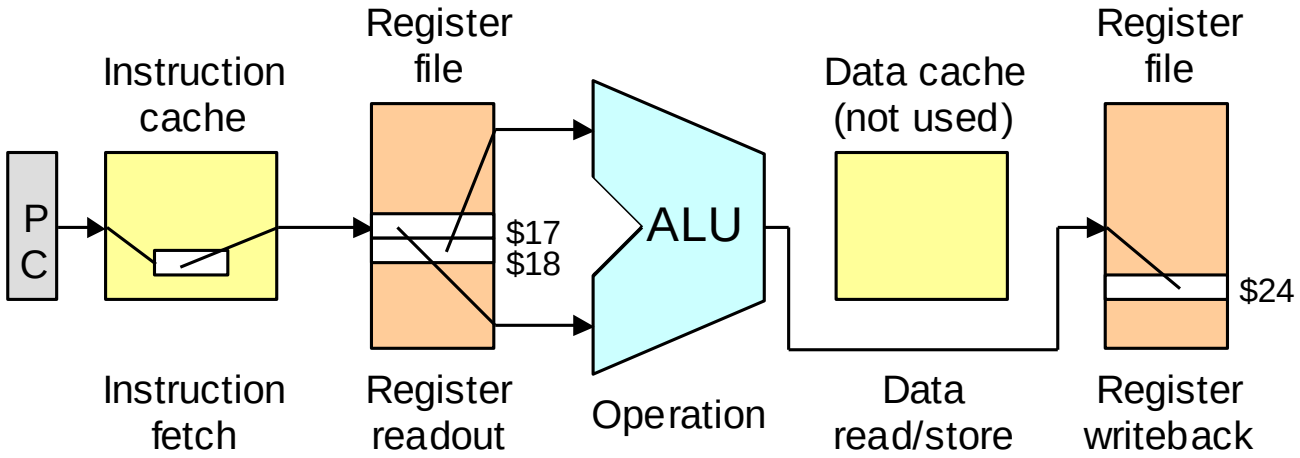
```
a = b + c
```

Assembly language instruction:

```
add $t8, $s2, $s1
```

Machine language instruction:





A typical instruction for MiniMIPS and steps in its execution.

MiniMIPS Instruction Formats

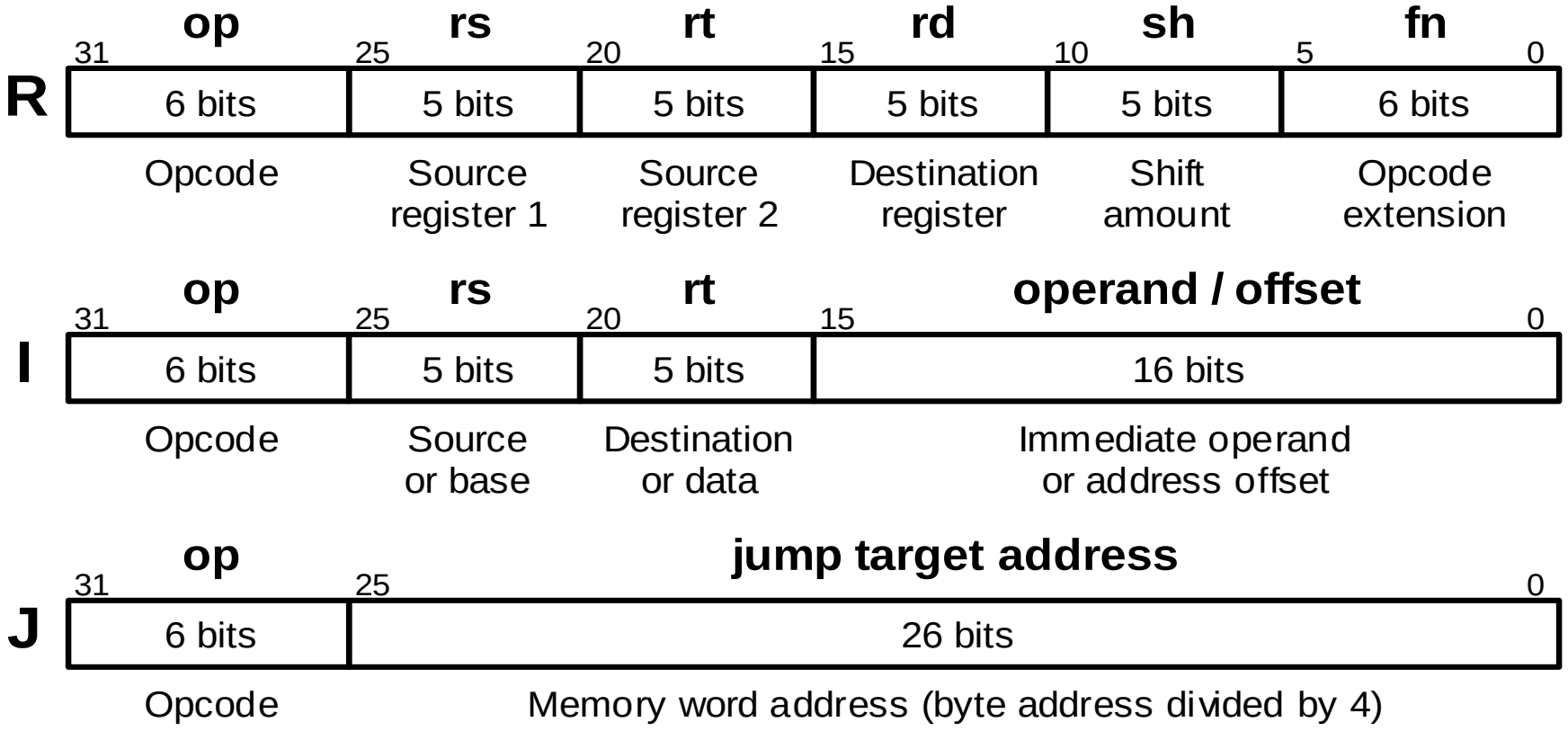


Figure 5.4 MiniMIPS instructions come in only three formats: register (R), immediate (I), and jump (J).

Simple Arithmetic/Logic Instructions

Add and subtract already discussed; logical instructions are similar

```
add   $t0,$s0,$s1      # set $t0 to ($s0)+($s1)
sub   $t0,$s0,$s1      # set $t0 to ($s0)-($s1)
and   $t0,$s0,$s1      # set $t0 to ($s0)∧($s1)
or    $t0,$s0,$s1      # set $t0 to ($s0)∨($s1)
xor   $t0,$s0,$s1      # set $t0 to ($s0)⊕($s1)
nor   $t0,$s0,$s1      # set $t0 to (($s0)∨($s1))′
```

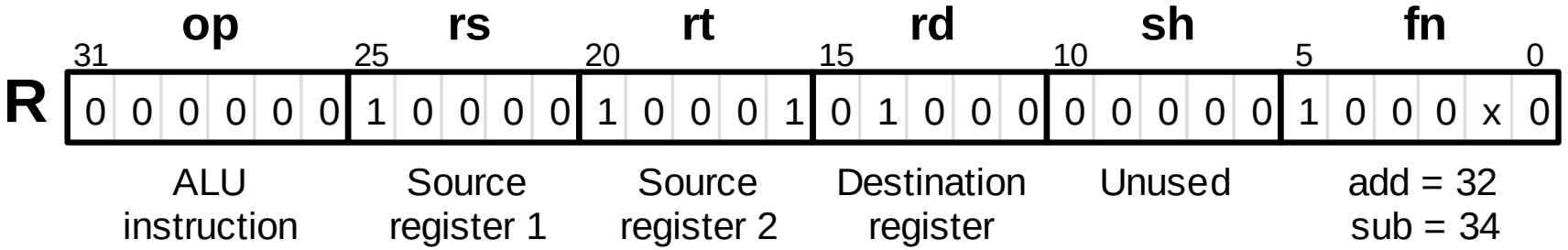


Figure 5.5 The arithmetic instructions `add` and `sub` have a format that is common to all two-operand ALU instructions. For these, the `fn` field specifies the arithmetic/logic operation to be performed.

Arithmetic/Logic with One Immediate Operand

An operand in the range [–32 768, 32 767], or [`0x0000`, `0xffff`], can be specified in the immediate field.

```
addi   $t0,$s0,61      # set $t0 to ($s0)+61
andi   $t0,$s0,61      # set $t0 to ($s0)∧61
ori    $t0,$s0,61      # set $t0 to ($s0)∨61
xori   $t0,$s0,0x00ff  # set $t0 to ($s0)⊕ 0x00ff
```

For arithmetic instructions, the immediate operand is sign-extended

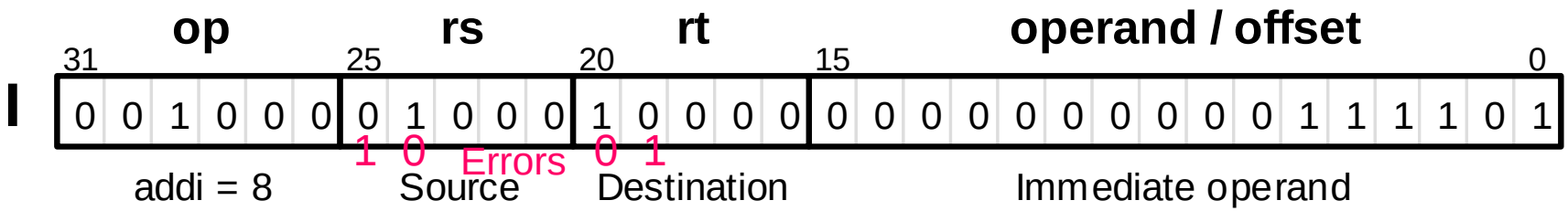
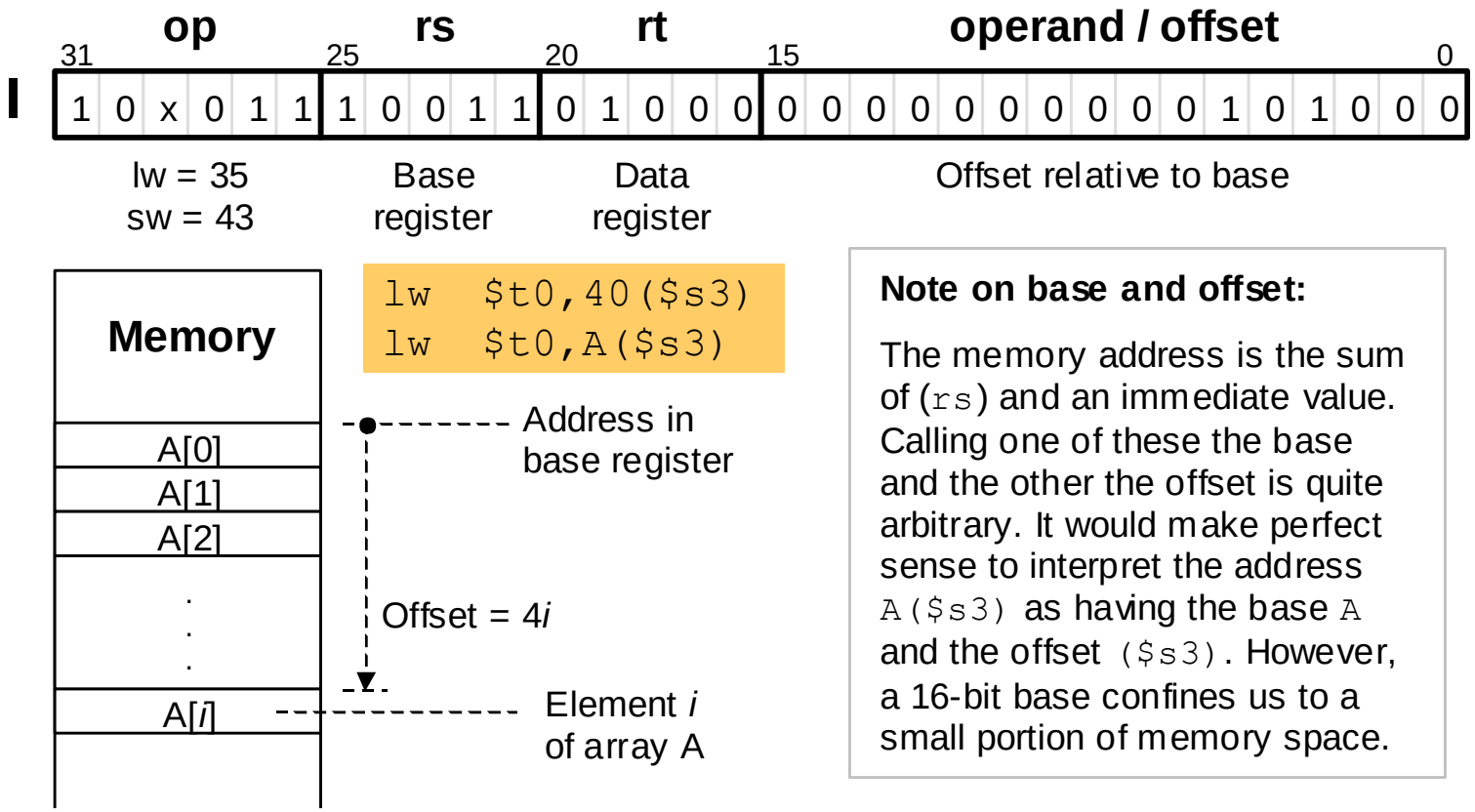


Figure 5.6 Instructions such as `addi` allow us to perform an arithmetic or logic operation for which one operand is a small constant.

Load and Store Instructions



Note on base and offset:

The memory address is the sum of (`rs`) and an immediate value. Calling one of these the base and the other the offset is quite arbitrary. It would make perfect sense to interpret the address `A($s3)` as having the base `A` and the offset `($s3)`. However, a 16-bit base confines us to a small portion of memory space.

MiniMIPS `lw` and `sw` instructions and their memory addressing convention that allows for simple access to array elements via a base address and an offset (offset = 4*i* leads us to the *i* th word).

`lw`, `sw`, and `lui` Instructions

```
lw    $t0,40($s3)      # load mem[40+($s3)] in $t0
sw    $t0,A($s3)       # store ($t0) in mem[A+($s3)]
                       # "($s3)" means "content of $s3"
lui   $s0,61           # The immediate value 61 is
                       # loaded in upper half of $s0
                       # with lower 16b set to 0s
```

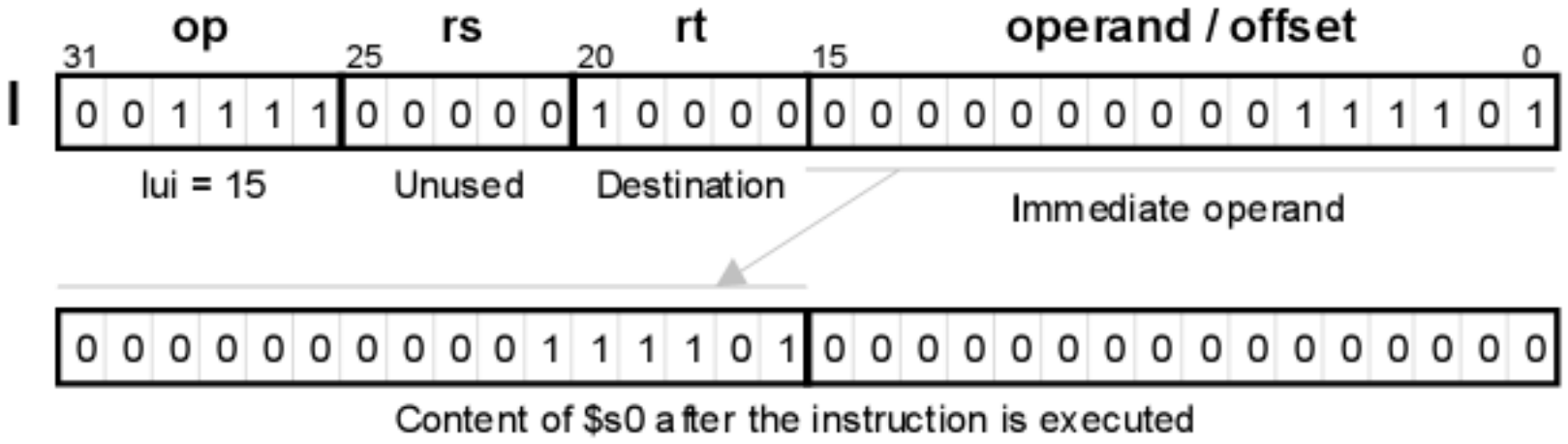


Figure 5.8 The `lui` instruction allows us to load an arbitrary 16-bit value into the upper half of a register while setting its lower half to 0s.

Initializing a Register

Example 5.2

Show how each of these bit patterns can be loaded into `$s0`:

```
0010 0001 0001 0000 0000 0000 0011 1101
1111 1111 1111 1111 1111 1111 1111 1111
```

Solution

The first bit pattern has the hex representation: `0x2110003d`

```
lui   $s0,0x2110      # put the upper half in $s0
ori   $s0,0x003d      # put the lower half in $s0
```

Same can be done, with immediate values changed to `0xffff` for the second bit pattern. But, the following is simpler and faster:

```
nor   $s0,$zero,$zero # because (0 v 0)' = 1
```

5.5 Jump and Branch Instructions

Unconditional jump and jump through register instructions

```
j   verify          # go to mem loc named "verify"
jr  $ra             # go to address that is in $ra;
                    # $ra may hold a return address
```

$ra is the symbolic name for reg. $31 (return address)

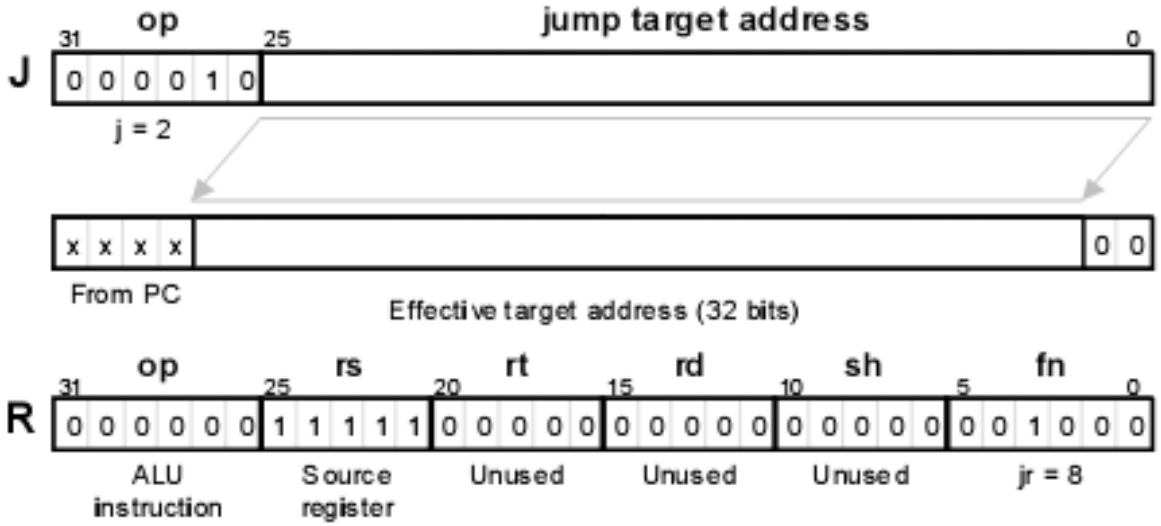


Figure 5.9 The jump instruction `j` of MiniMIPS is a J-type instruction which is shown along with how its effective target address is obtained. The jump register (`jr`) instruction is R-type, with its specified register often being `$ra`.

Conditional Branch Instructions

Conditional branches use PC-relative addressing

```
bltz $s1,L          # branch on ($s1)< 0
beq  $s1,$s2,L      # branch on ($s1)=($s2)
bne  $s1,$s2,L      # branch on ($s1)≠($s2)
```

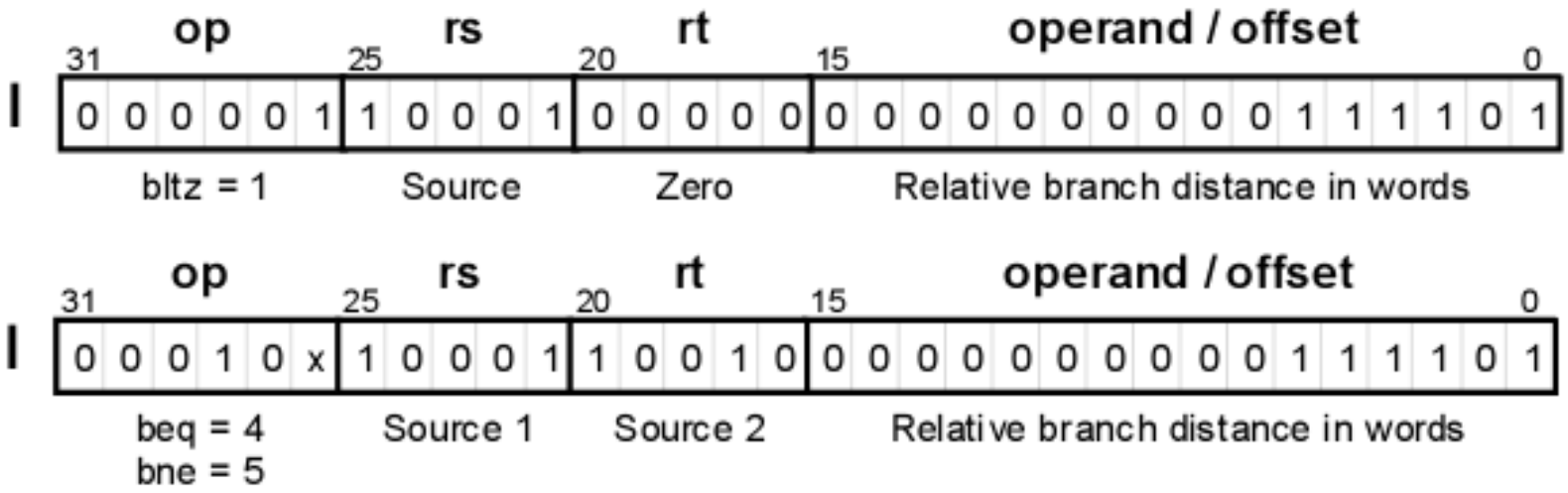


Figure 5.10 (part 1) Conditional branch instructions of MiniMIPS.

Comparison Instructions for Conditional Branching

```
slt   $s1,$s2,$s3     # if ($s2)<($s3), set $s1 to 1
                      # else set $s1 to 0;
                      # often followed by beq/bne
slti  $s1,$s2,61      # if ($s2)<61, set $s1 to 1
                      # else set $s1 to 0
```

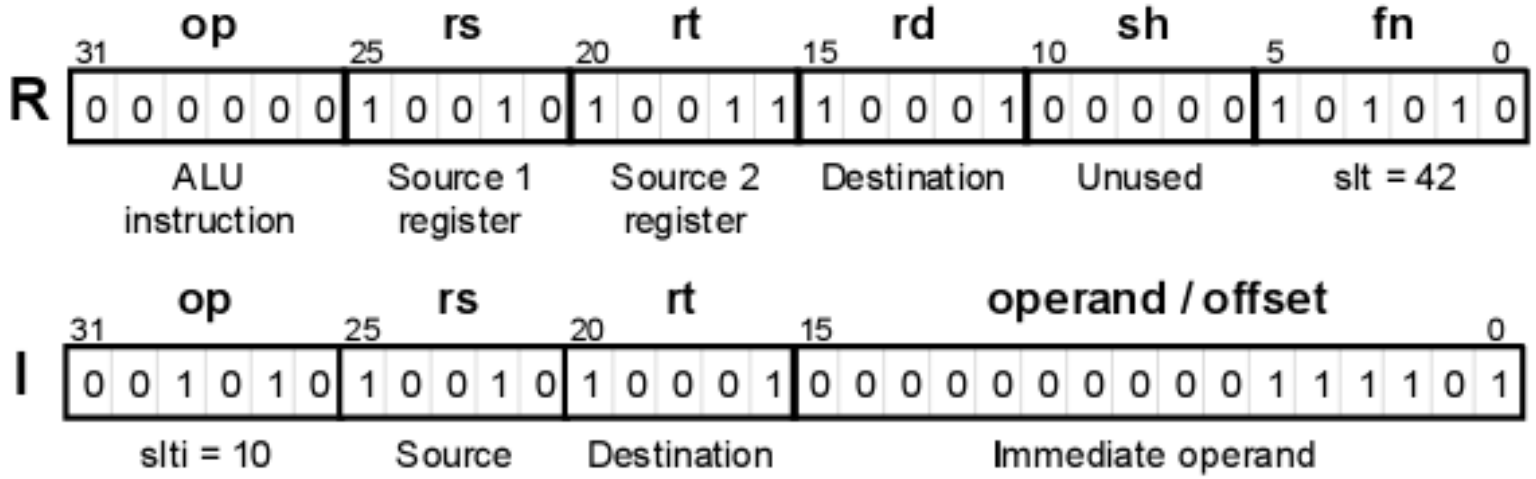


Figure 5.10 (part 2) Comparison instructions of MiniMIPS.

Examples for Conditional Branching

If the branch target is too far to be reachable with a 16-bit offset (rare occurrence), the assembler automatically replaces the branch instruction `beq $s1,$s2,L1` with:

```
      bne  $s1,$s2,L2          # skip jump if (s1)≠(s2)
      j    L1           # goto L1 if (s1)=(s2)
 L2: ...
```

Forming if-then constructs; e.g., `if (i == j) x = x + y`

```
      bne  $s1,$s2,endif  # branch on i≠j
      add  $t1,$t1,$t2        # execute the "then" part
endif: ...
```

If the condition were `(i < j)`, we would change the first line to:

```
      slt  $t0,$s1,$s2        # set $t0 to 1 if i<j
      beq  $t0,$0,endif       # branch if ($t0)=0;
                              # i.e., i not< j or i≥j
```

Compiling if-then-else Statements

Example 5.3

Show a sequence of MiniMIPS instructions corresponding to:

```
if (i<=j) x = x+1; z = 1; else y = y-1; z = 2*z
```

Solution

Similar to the "if-then" statement, but we need instructions for the "else" part and a way of skipping the "else" part after the "then" part.

```
        slt  $t0,$s2,$s1     # j<i? (inverse condition)
        bne  $t0,$zero,else  # if j<i goto else part
        addi $t1,$t1,1       # begin then part: x = x+1
        addi $t3,$zero,1     # z = 1
        j    endif           # skip the else part
else:   addi $t2,$t2,-1      # begin else part: y = y-1
        add  $t3,$t3,$t3     # z = z+z
endif:...
```

`while` Statements

Example

The simple while loop: `while (A[i]==k) i=i+1;`
Assuming that: `i, A, k` are stored in `$s1,$s2,$s3`

Solution

```
loop: add   $t1,$s1,$s1  # t1 = 4*i
      add   $t1,$t1,$t1  #
      add   $t1,$t1,$s2  # t1 = A + 4*i
      lw    $t0,0($t1)           # t0 = A[i]
      bne   $t0,$s3,endwhl       #
      addi $s1,$s1,1             #
      j     loop                 #
endwhl: ...                      #
```

`switch` Statements

Example

The simple switch

```
switch(test) {
   case 0:
        a=a+1; break;
   case 1:
        a=a-1; break;
   case 2:
        b=2*b; break;
   default:
}
```

Assuming that: `test,a,b` are stored in `$s1,$s2,$s3`

```
        beq     s1,t0,case_0
        beq     s1,t1,case_1
        beq     s1,t2,case_2
        b       default
case_0:
        addi    s2,s2,1         #a=a+1
        b       continue
case_1:
        sub     s2,s2,t1        #a=a-1
        b       continue
case_2:
        add     s3,s3,s3        #b=2*b
        b       continue
default:
continue:
```

The 20 MiniMIPS Instructions Covered So Far

	Instruction	Usage	op	fn
Copy	Load upper immediate	`lui rt,imm`	15	
Arithmetic	Add	`add rd,rs,rt`	0	32
	Subtract	`sub rd,rs,rt`	0	34
	Set less than	`slt rd,rs,rt`	0	42
	Add immediate	`addi rt,rs,imm`	8	
	Set less than immediate	`slti rd,rs,imm`	10	
Logic	AND	`and rd,rs,rt`	0	36
	OR	`or rd,rs,rt`	0	37
	XOR	`xor rd,rs,rt`	0	38
	NOR	`nor rd,rs,rt`	0	39
	AND immediate	`andi rt,rs,imm`	12	
	OR immediate	`ori rt,rs,imm`	13	
	XOR immediate	`xori rt,rs,imm`	14	
Memory access	Load word	`lw rt,imm(rs)`	35	
	Store word	`sw rt,imm(rs)`	43	
Control transfer	Jump	`j L`	2	
	Jump register	`jr rs`	0	8
	Branch less than 0	`bltz rs,L`	1	
	Branch equal	`beq rs,rt,L`	4	
	Branch not equal	`bne rs,rt,L`	5	

Table 5.1

Pseudoinstructions

- Pseudoinstructions means “fake instruction”
- Pseudoinstructions do not correspond to real MIPS instructions
- The assembler, that converts assembly language programs to machine code, would then translate pseudoinstructions to real instructions, usually requiring at least one on more instructions.
- Example:
  - **mov $rt, $rs** #Copy contents of register **s** to register **t**, R[t] = R[s]

  => real instruction: **addi $rt, $rs, 0**

5.6 Addressing Modes

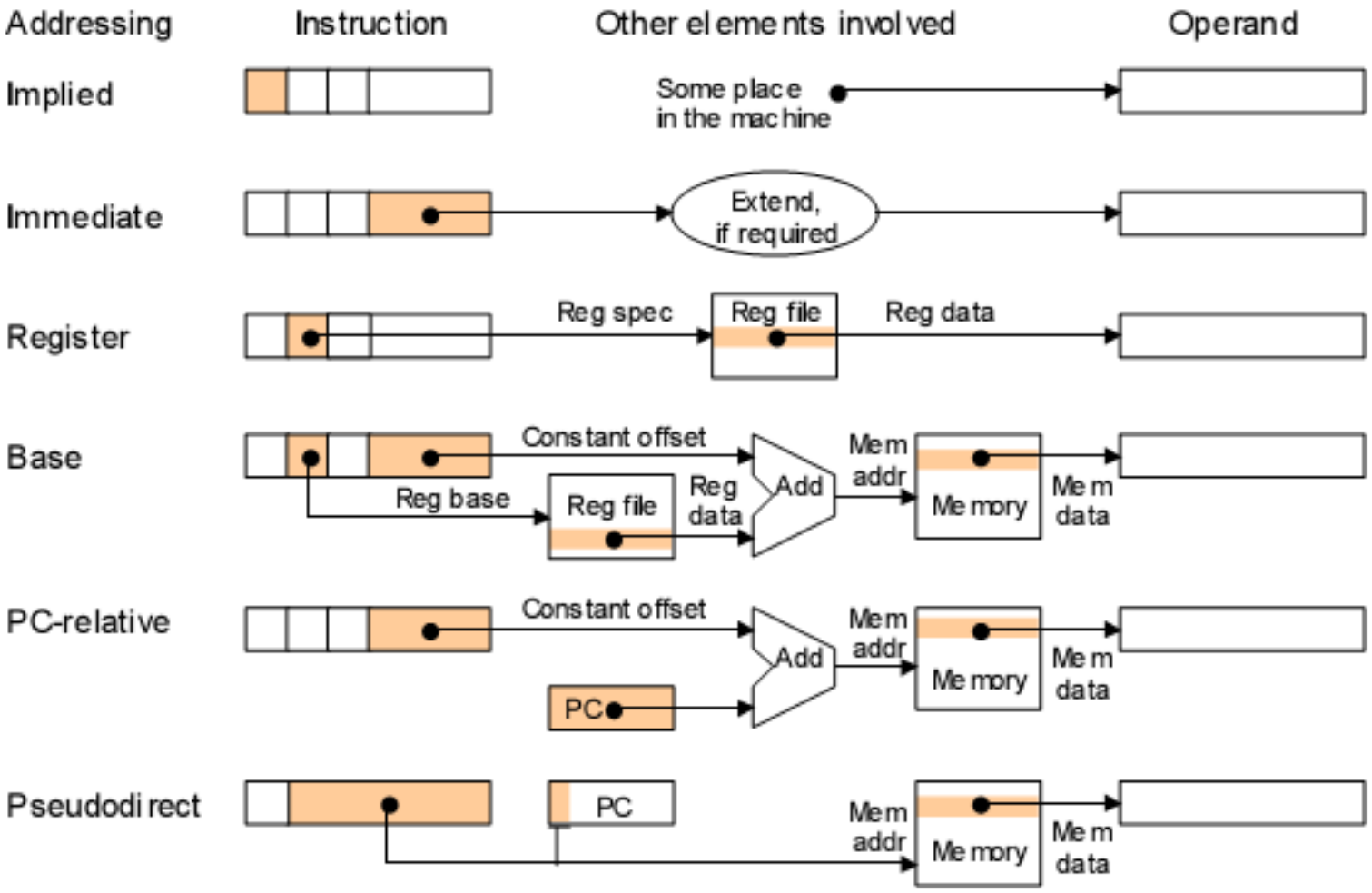


Figure 5.11 Schematic representation of addressing modes in MiniMIPS.

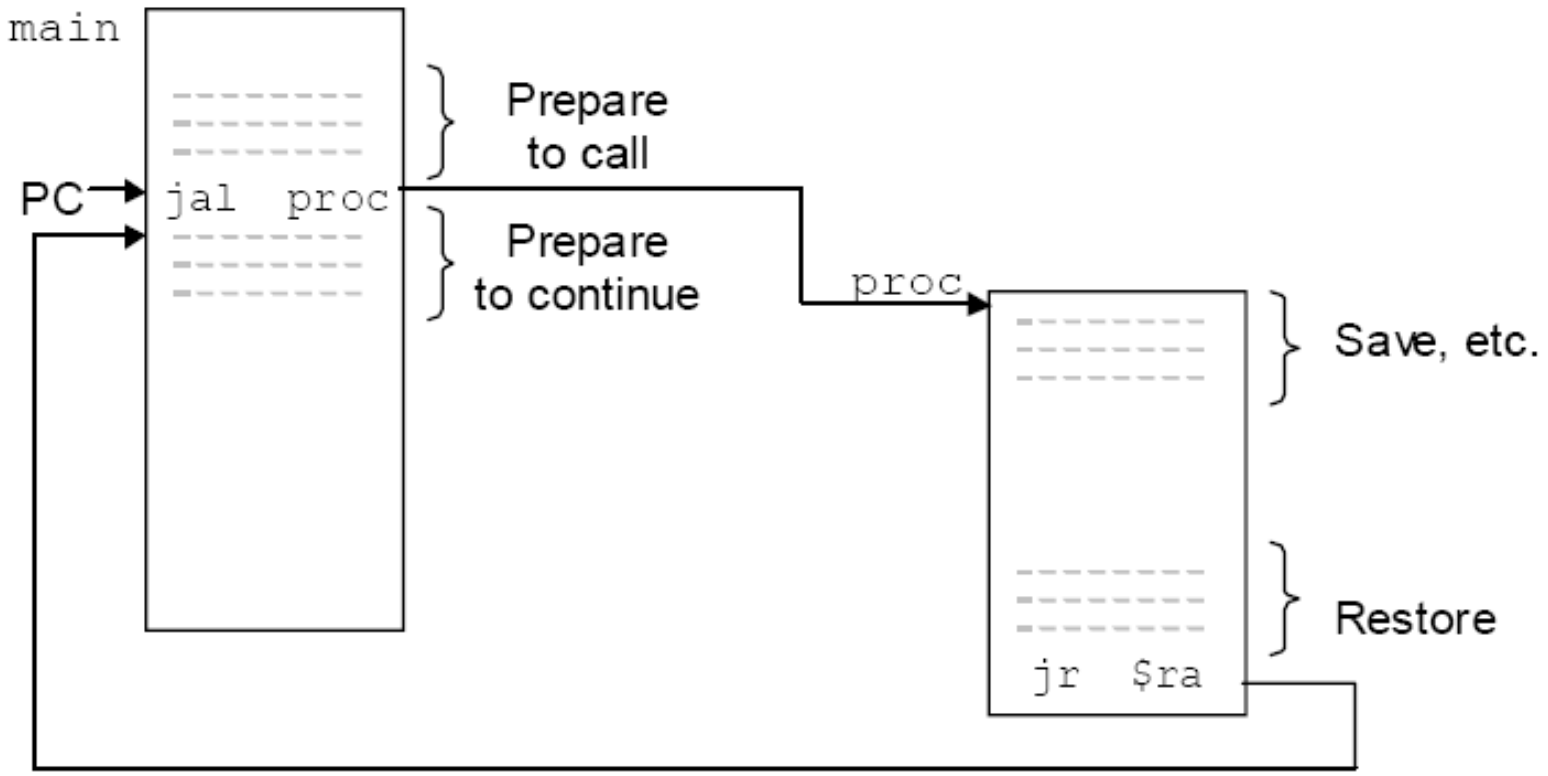
Procedure & Stack

- Procedure call:

```
jal ( jump and link)
```

- Return to call point

```
jr $ra
```

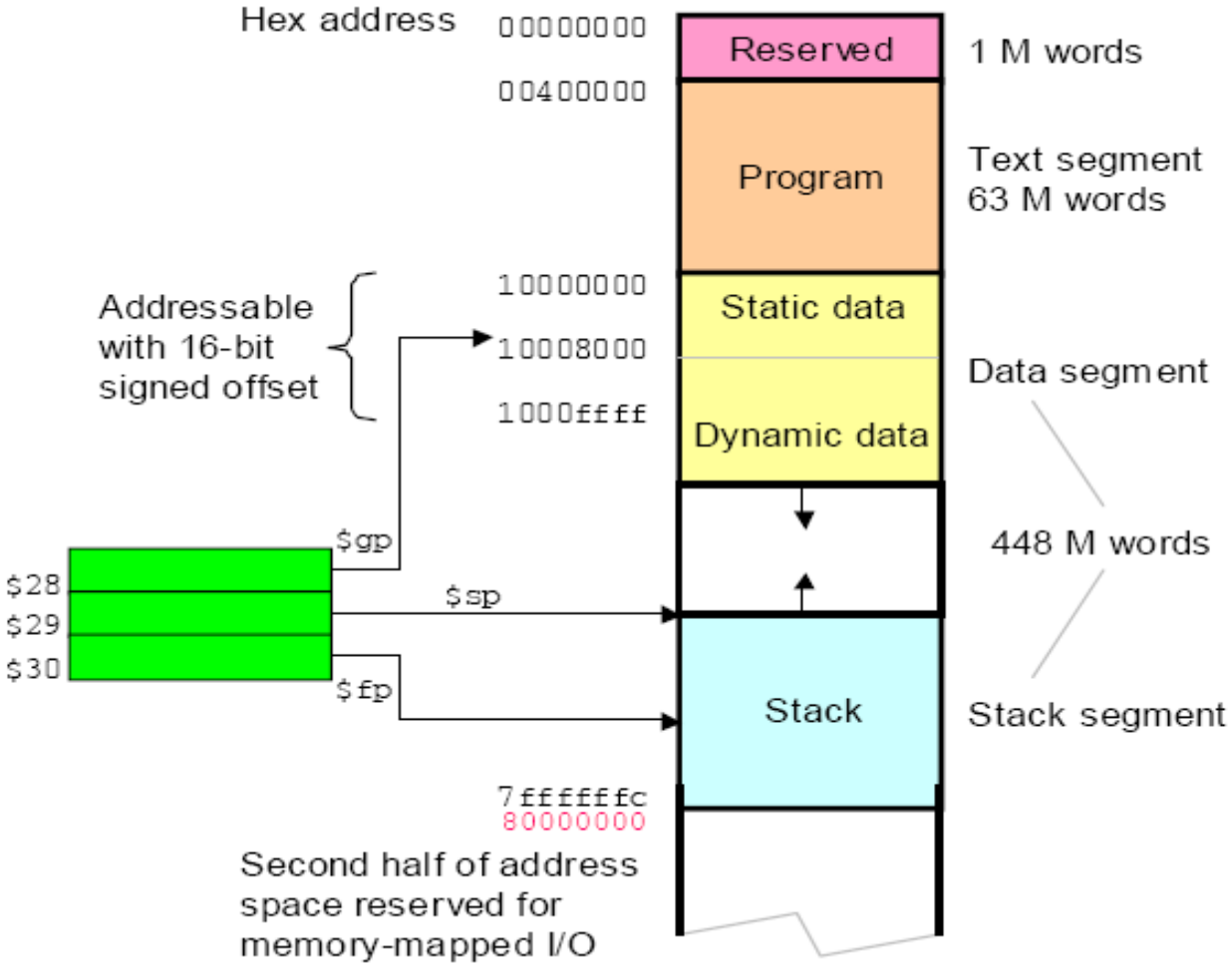


Stack

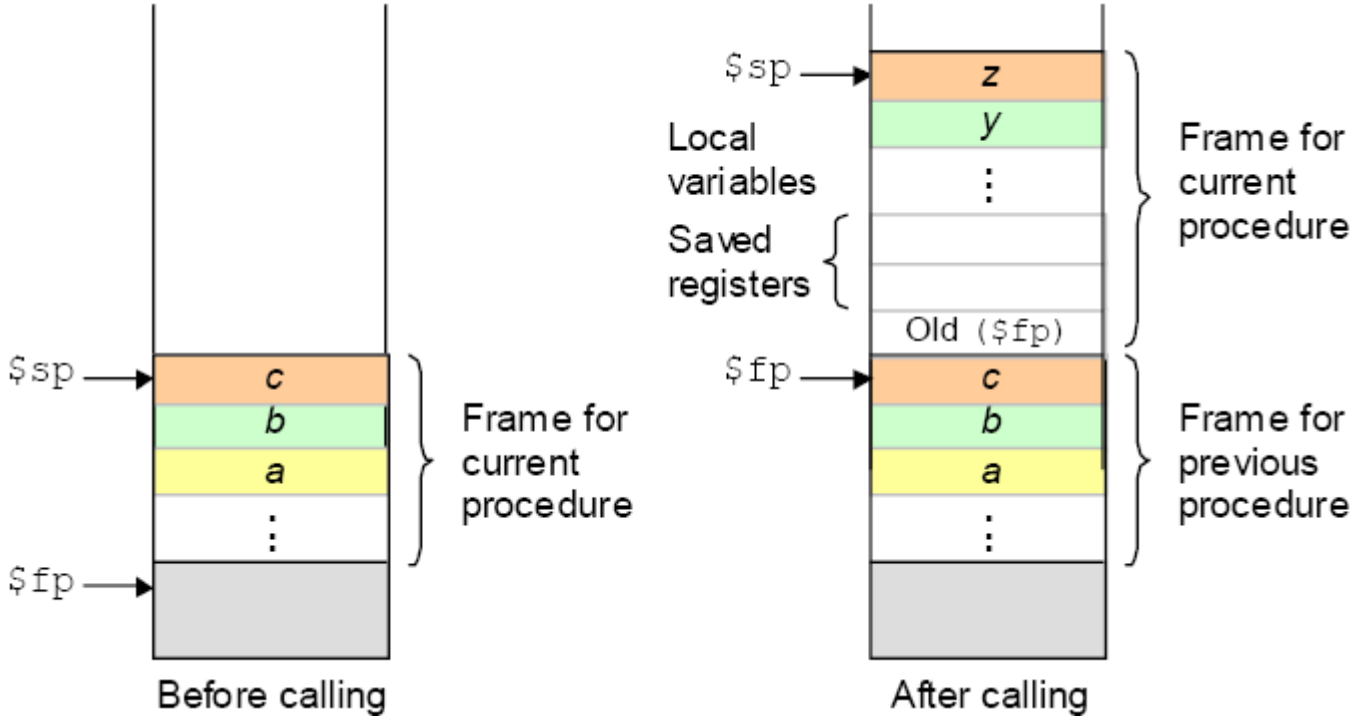

```
push: addi   $sp,$sp,-4
      sw     $t4,0($sp)
```

```
pop: lw     $t5,0($sp)
     addi   $sp,$sp,4
```

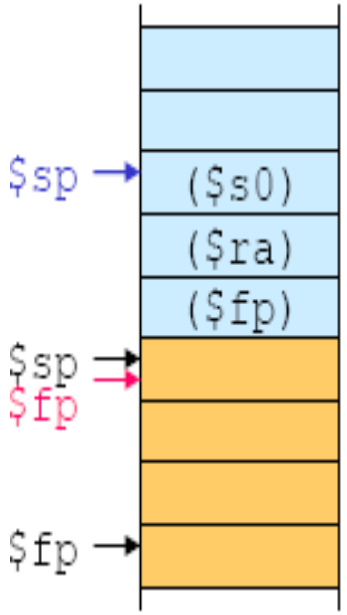
Stack


Hex address
00000000
00400000
10000000
10008000
1000ffff
7fffffffc
80000000
Reserved
Program
Static data
Dynamic data
Stack
1 M words
Text segment
63 M words
Data segment
448 M words
Stack segment
Addressable
with 16-bit
signed offset
$gp
$28
$29
$30
$sp
$fp
Second half of address
space reserved for
memory-mapped I/O

$sp and $fp


$sp
c
b
a
$fp
Frame for current procedure
Before calling
$sp
z
y
Local variables
Saved registers
Old ($fp)
$fp
c
b
a
Frame for current procedure
Frame for previous procedure
After calling

Example: $sp and $fp




```
proc:  sw   $fp,-4($sp)    # save the old frame pointer
       addi $fp,$sp,0      # save ($sp) into $fp
       addi $sp,$sp,-12    # create 3 spaces on top of stack
       sw   $ra,-8($fp)    # save ($ra) in 2nd stack element
       sw   $s0,-12($fp)   # save ($s0) in top stack element
        .
        .
        .
       lw   $s0,-12($fp)   # put top stack element in $s0
       lw   $ra,-8($fp)    # put 2nd stack element in $ra
       addi $sp,$fp, 0     # restore $sp to original state
       lw   $fp,-4($sp)    # restore $fp to original state
       jr   $ra            # return from procedure
```

III. MIPSIT User Guide

- MIPSIT IDE + MIPS Simulator
- MIPS assembly program

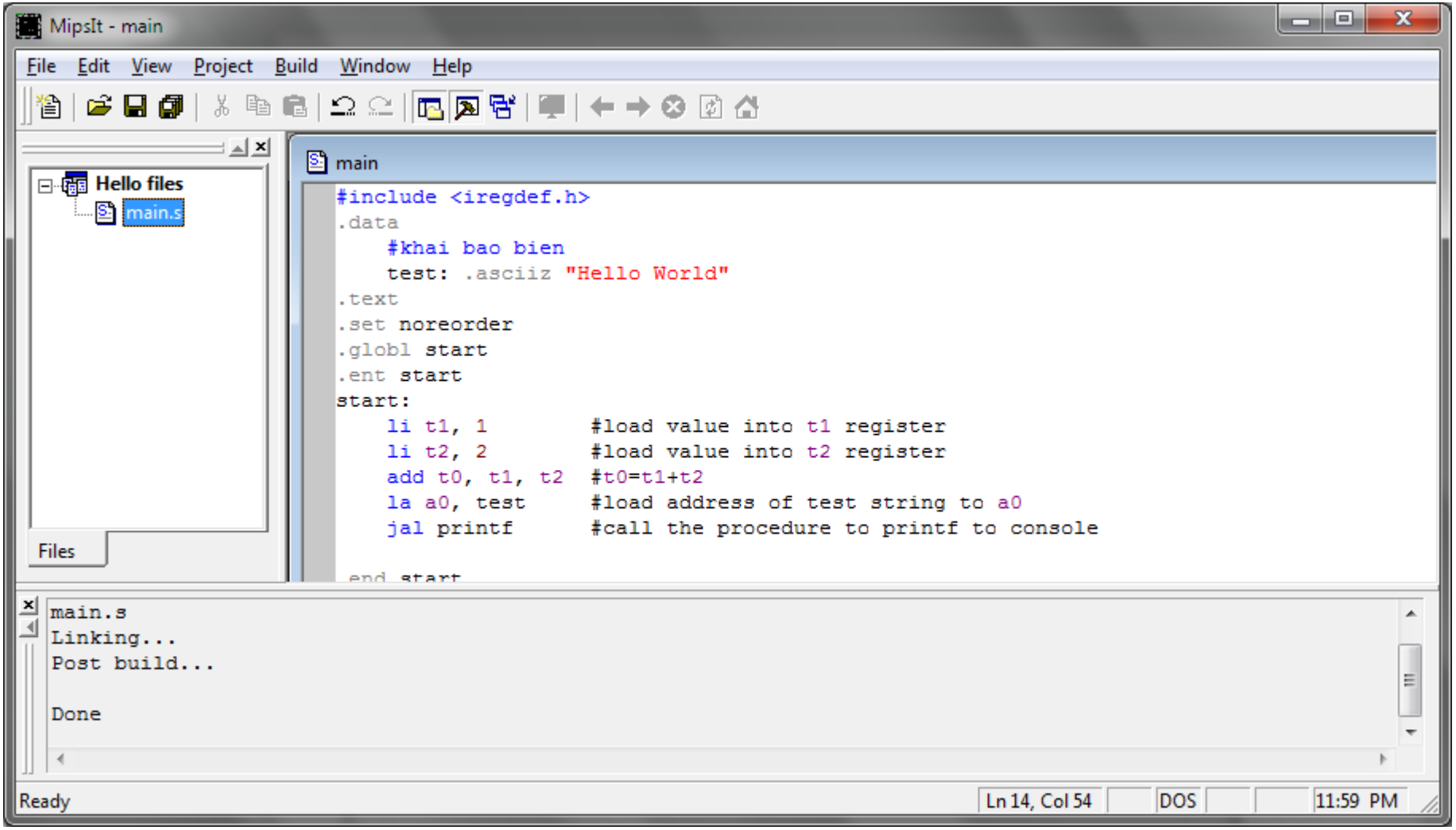
Instroduce to MIPSIT Studio

- MIPSIT Studio: a tool to **edit**, **compile** and **simulate** programs based on MIPS instruction set.

Installation

- Extract MipsIt.rar to installed folder, “C:\MipsIt” for example. After this step, MipsIt folder will have following structure:
- MipsIt\bin: include execution files, such as
  - MipsIt.exe: an editor and compiler program
  - Mips.exe: a simulator program
  - MipsPipeS.exe: a simple pipeline simulator program
  - MipsPipeXL.exe: a more complicated pipeline simulator program
- …
- MipsIt\include: include header files
- MipsIt\lib: include library files
- …

IDE Basics

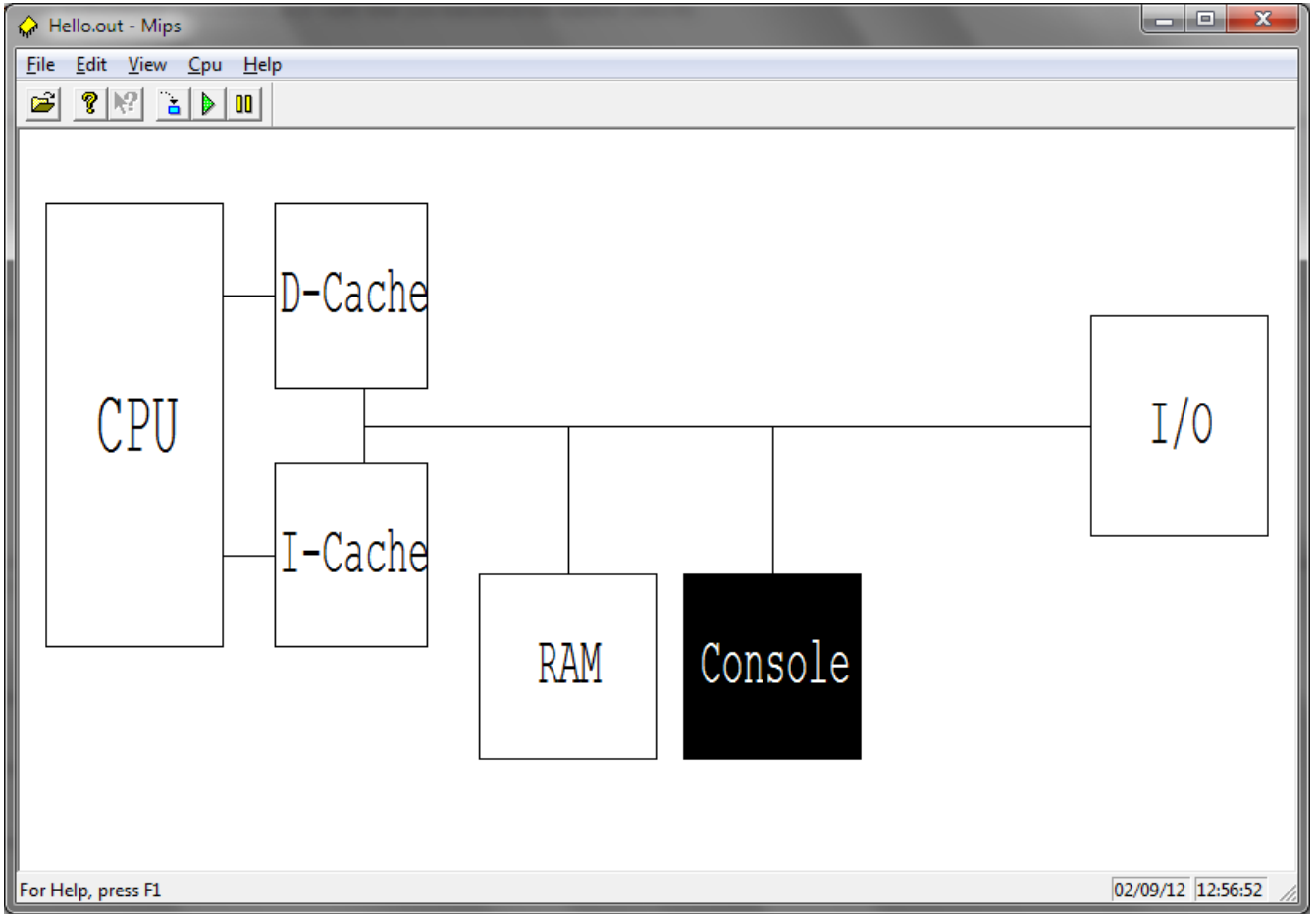

MipsIt - main
File Edit View Project Build Window Help
Hello files
main.s
Files
main
#include <iregdef.h>
.data
#khai bao bien
test: .asciiz "Hello World"
.text
.set noreorder
.globl start
.ent start
start:
li t1, 1 #load value into t1 register
li t2, 2 #load value into t2 register
add t0, t1, t2 #t0=t1+t2
la a0, test #load address of test string to a0
jal printf #call the procedure to printf to console
.end start
main.s
Linking...
Post build...
Done
Ready
Ln 14, Col 54
DOS
11:59 PM

IDE Basics

- Files list
- Editor
- Compile, Build
- Upload to Simulator

The Simulator

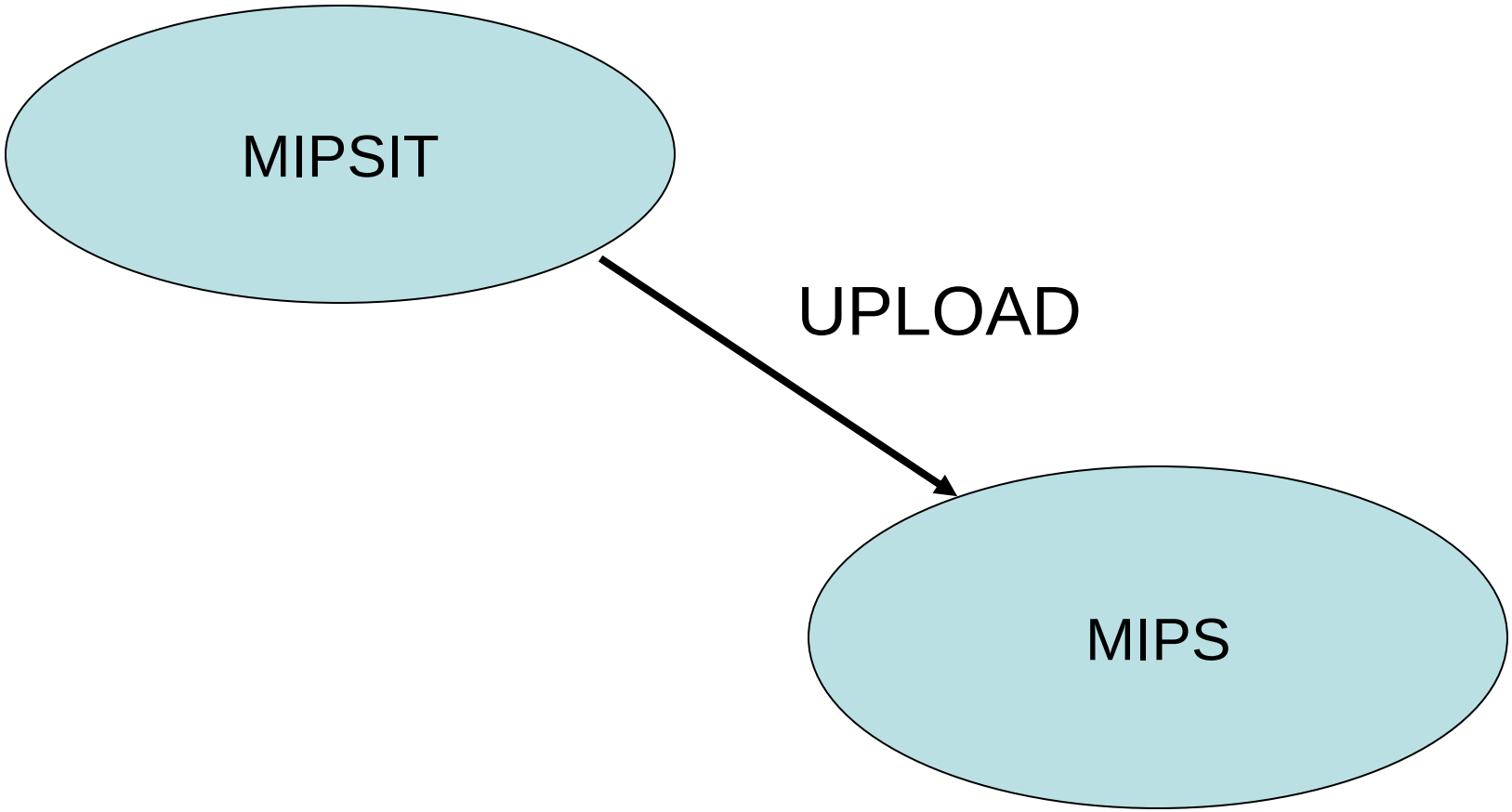
- MIPS.exe



The Simulator

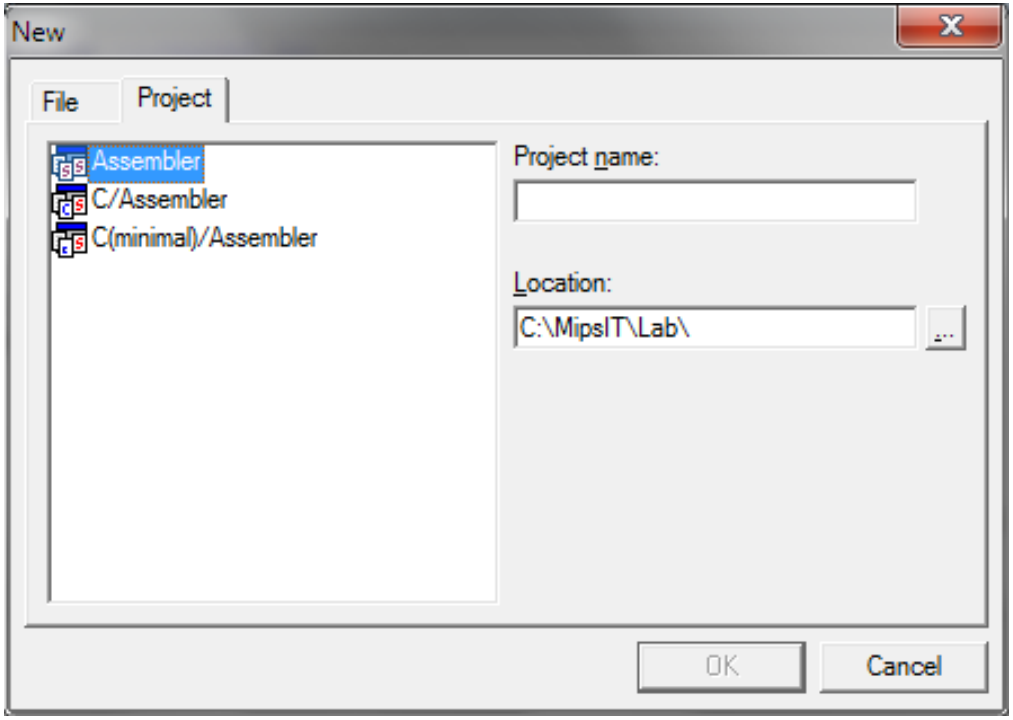
- *CPU*
  View/modify the CPU registers.
  RAM
  View/modify memory, also referred to as the MemView. This unit has most functions of all, for a more detailed description see below.
  Console
  Standard input/output for programs that use it.
  I/O
  Simulates the 8-bit I/O unit, with 8 switches and 8 LEDs.
  D-Cache/I-Cache
  Views of the data and instruction caches.
  Interrupt
  Simulates the interrupt unit, with buttons K1, K2 and the timer.

MIPSIT & MIPS



New project

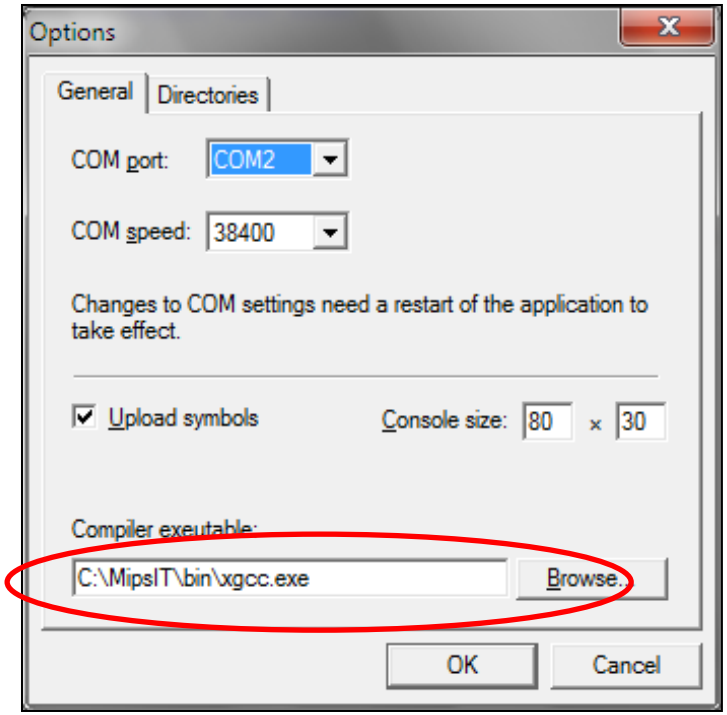
- Add files



- Chú ý: lỗi compile
  - Kiểm tra File/Option
  - Mục Compiler executable

  Trỏ đến $Root\MipsIT\bin\xgcc.exe

  (Không chứa dấu cách)



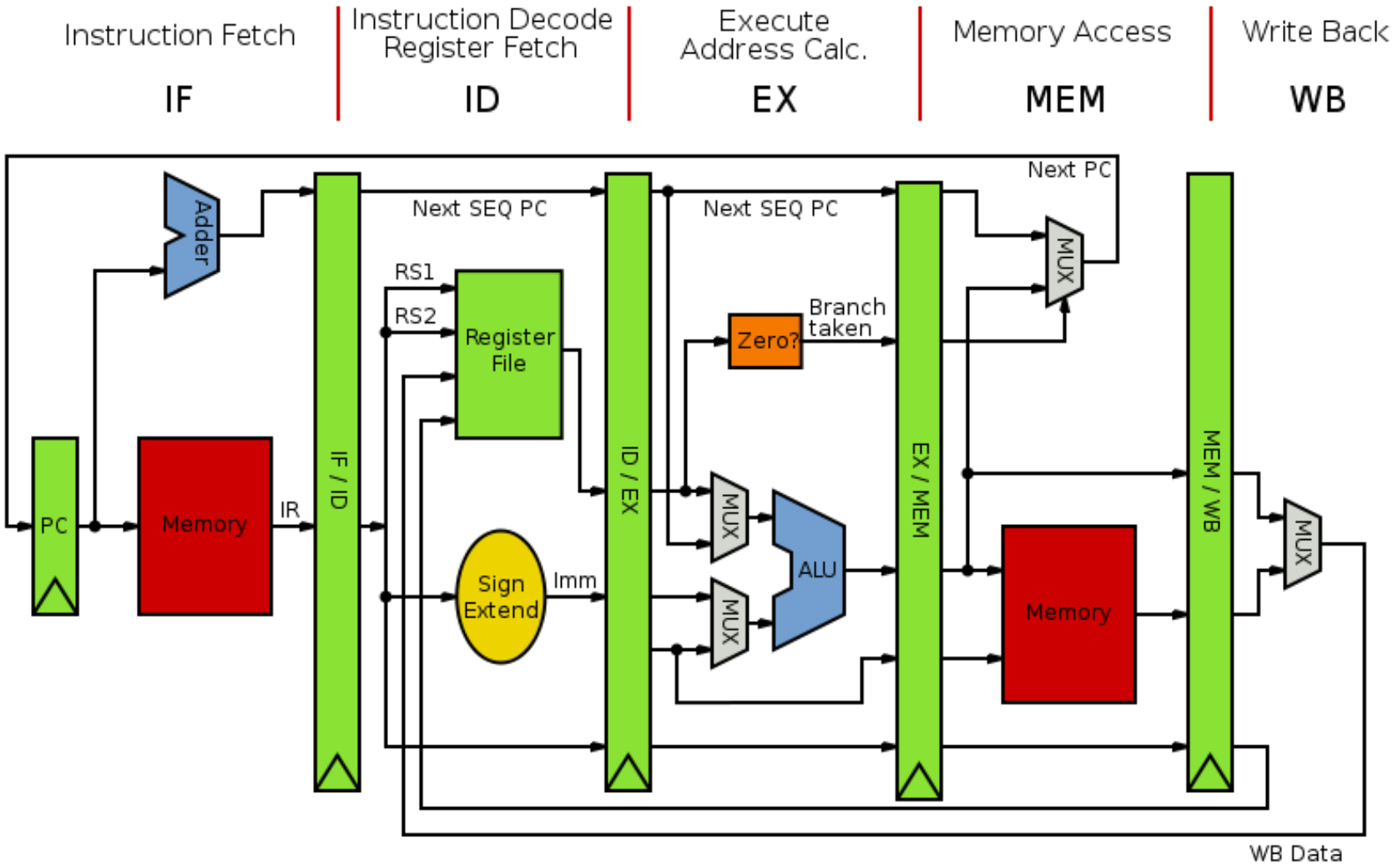
MIPS assembly program

```
#include <iregdef.h>
.data
#declare variables
.text
.globl start
.ent start
start:
#main
.end start
.ent CTCon
CTCon:
#procedure
.end CTCon
```

Example: Hello World

```
#include <iregdef.h>
.data
test: .asciiz "Hello World"
.text
.set noreorder
.globl start
.ent start
start:
    la   a0,test        #load the address of test string to a0
    jal  printf         #print test tring to console
.end start
```

Pipelined MIPS



Lab 3. Arithmetic & Logical Operation

- Bit mask in logical operation

```
li   s0,0x0563              #load test value for these function
```

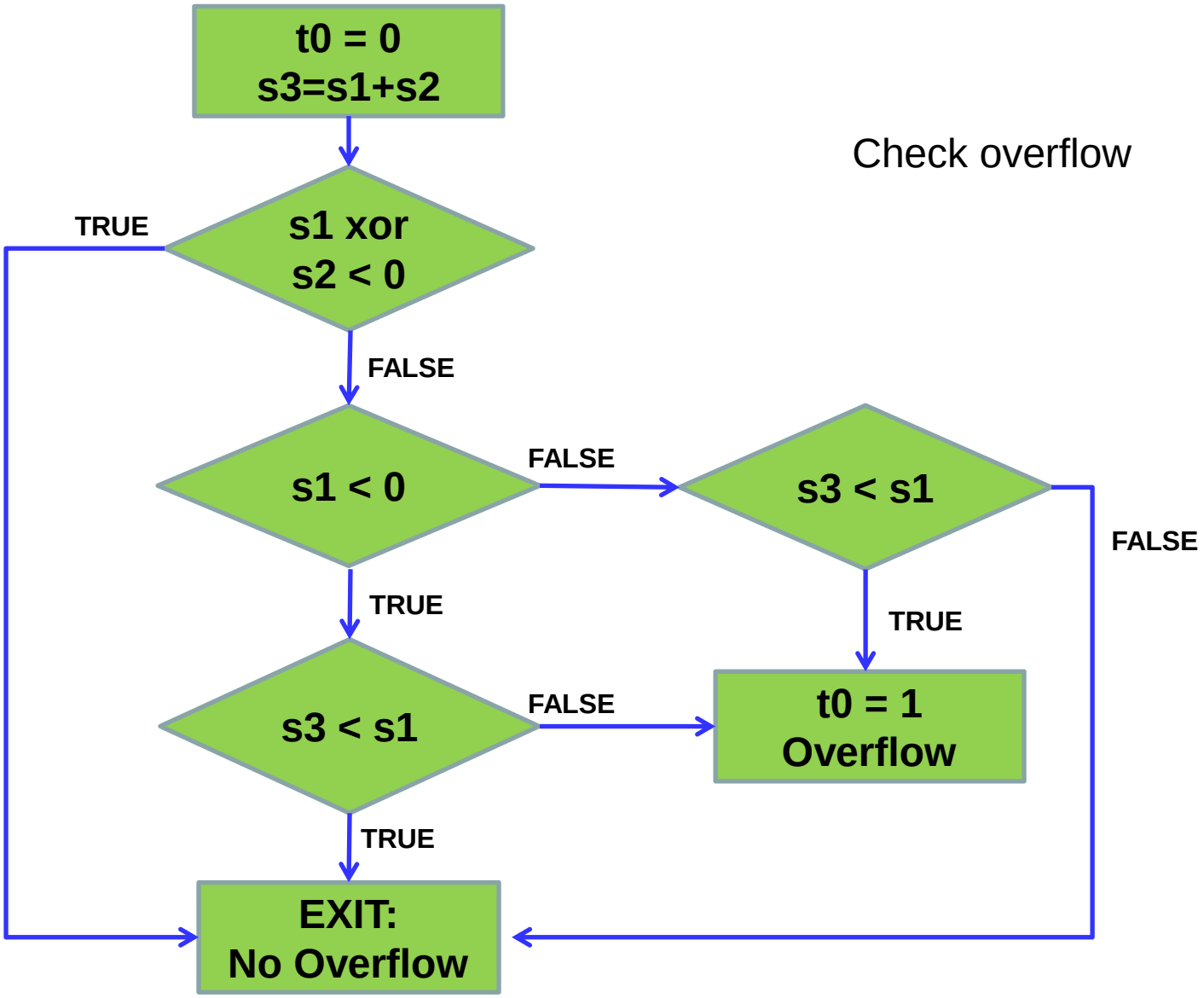
s0 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1

```
andi      t0,s0,0xff                #Extract the LSB of s0
```

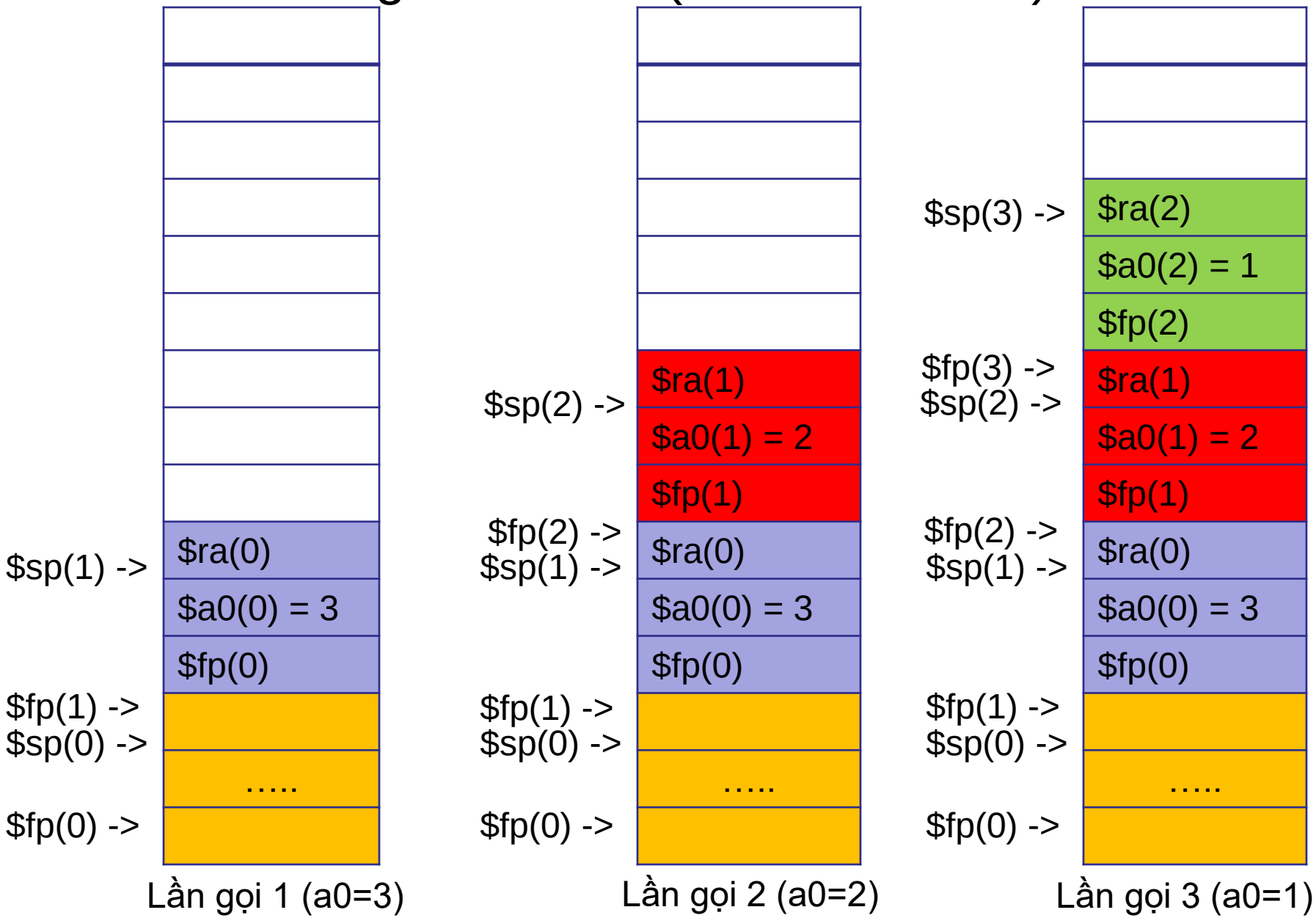
MASK 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

t0 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1

Lab 3. Arithmetic & Logical Operation

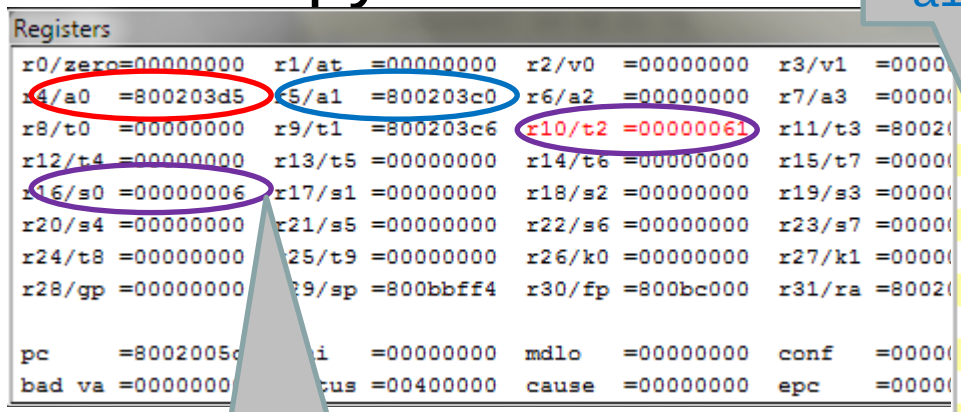


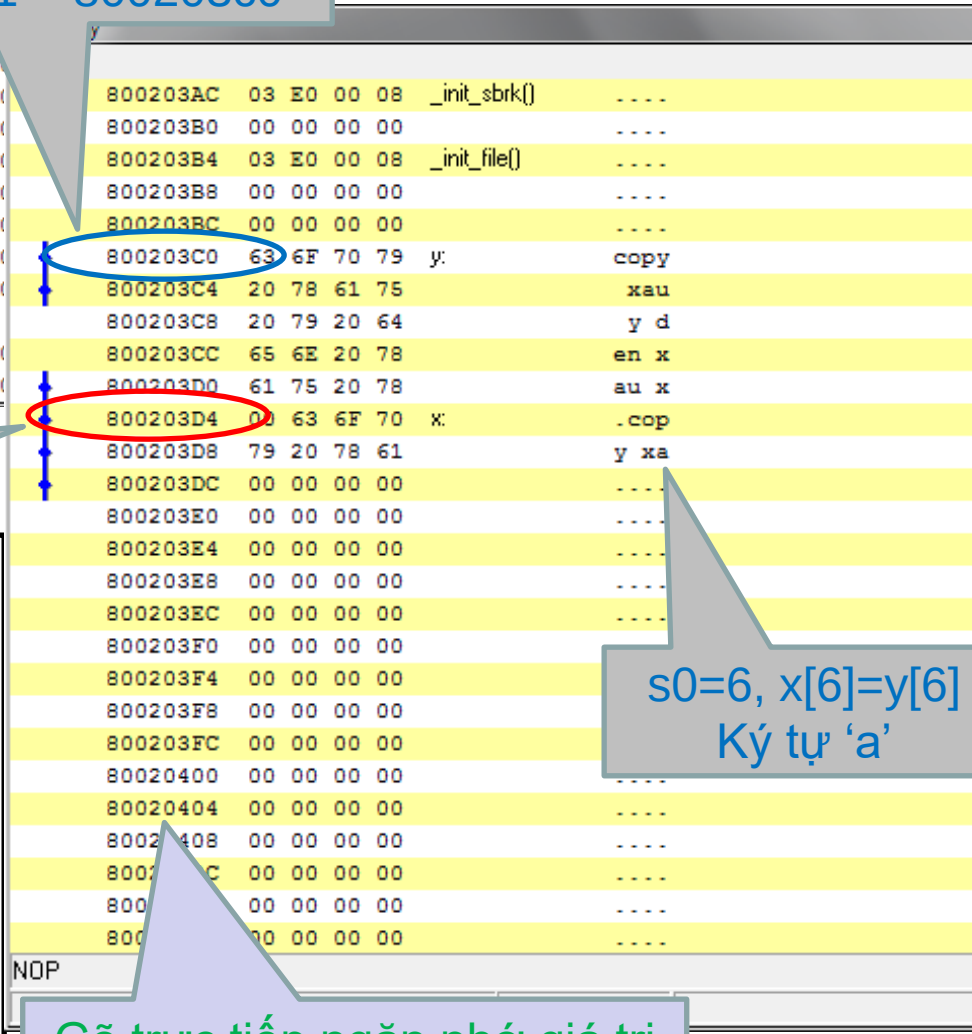
Lab 4. Procedure Calls,
Assigment 4. n! (stack with n=3)



Lab 5. Character string

- strcpy





Địa chỉ xâu y:
a1 = 800203c0

s0=6, x[6]=y[6]
Ký tự ‘a’

Địa chỉ xâu x:
a0 = 800203d5

s0=6, x[6]=y[6]
Ký tự ‘a’

```
L1:
add     t1,s0,a1        #address of y[i] in t1
lb      t2,0(t1)        #t2=y[i]
add     t3,s0,a0        #address of x[i] in t3
sb      t2,0(t3)        #x[i]=y[i]
beq     t2,zero,L2      #if y[i]==0, go to L2
nop
addi    s0,s0,1         #i=i+1
j       L1              #go to L1
nop
L2:
```

Gõ trực tiếp ngăn nhớ giá trị địa chỉ muốn xem. ví dụ: 800203c0 (xâu y)