Kiến trúc MIPS

Nội dung

- Phần I: Tổng quan dòng vi xử lý MIPS
- Phần II: Mô hình Lập trình

Phần I: Tổng quan dòng vi xử lý MIPS

Lịch sử phát triển:

- MIPS (**Microprocessor without Interlocked Pipeline Stages**) hình thành trên cơ sở RISC.
- Năm 1981: John L. Hennessy đứng đầu một nhóm bắt đầu một công trình nghiên cứu về *bộ xử lý MIPS* đầu tiên tại **Stanford University**
- Một thiết kế chủ chốt trong MIPS là yêu cầu các câu lệnh phải hoàn thành trong 1 chu kì máy.
- Hãng MIPS Technologies (MIPS Computer Systems)

  http://www.mips.com

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

Các thế hệ của MIPS

- Ban đầu MIPS là kiến trúc 32 bit, sau này mở rộng ra 64bit.
- MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS 32 và MIPS 64. Hiện nay tồn tại MIPS 32 và MIPS 64.
- Các dòng vi xử lý thương mại MIPS đã được sản xuất:
  - R2000 năm 1985
  - R3000 năm 1988
  - R4000 năm 1991,mở rộng tập lệnh đầy đủ cho 64bit, 100MHz, 8kB.
  - R4400 năm 1993,16kB.
  - R8000 năm 1994: là thiết kế superscalar đầu tiên của MIPS

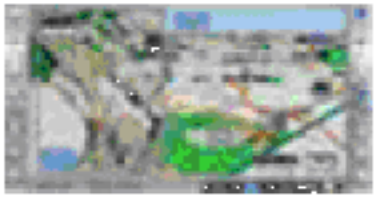
http://en.wikipedia.org/wiki/MIPS_architecture

- Các ứng dụng :

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

Portable Devices

Canon
EOS 10D Digital

JVC
GR-HD1

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

Sony Playstation PSX

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

Sony Playstation Portable

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

Phần II: Mô hình lập trình

- Quản lý bộ nhớ
- Các thanh ghi của MIPS
- Các khuôn dạng lệnh
- Các chế độ địa chỉ
- Một số lệnh cơ bản
- Khung chương trình hợp ngữ
- Sử dụng trình biên dịch và mô phỏng MIPS2000, MIPS

Quản lý bộ nhớ

Bộ nhớ:

32 bit địa chỉ, đánh địa chỉ theo byte

$\Rightarrow$ không gian 2^{32} địa chỉ 0x00000000 đến 0xFFFFFFFF

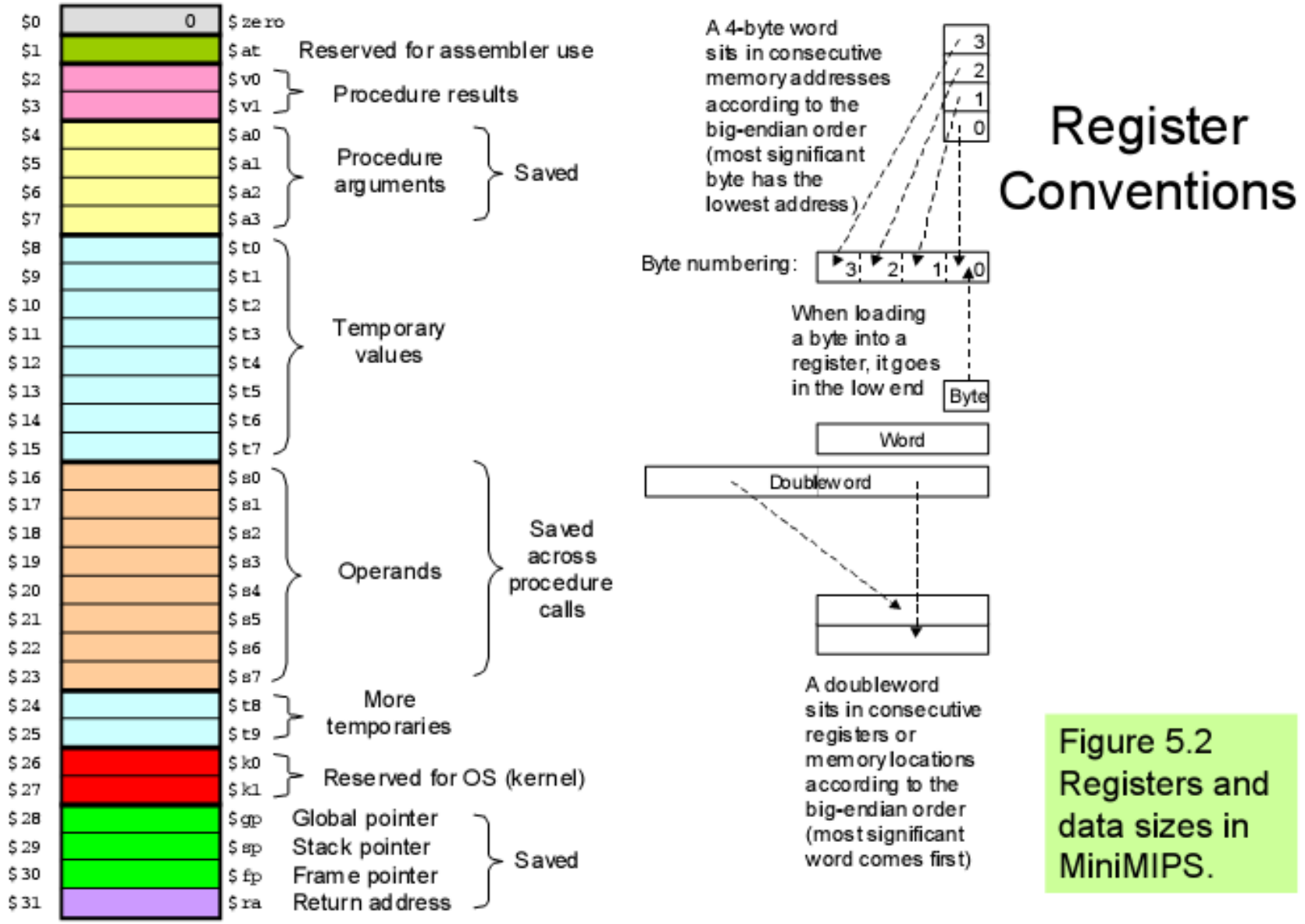
Chia làm các vùng:

0x00400000	Text segment	program instructions
0x10000000	Data segment	
0x7FFFFFFF	decreasing addresses	Stack segment

Tập thanh ghi

a) Thanh ghi đa năng.

Tên	Số	Ý nghĩa
$zero	$0	Hằng số 0
$at	$1	Assembler Temporary
$v0-$v1	$2-$3	Giá trị trả lại của hàm hoặc biểu thức
$a0-$a3	$4-$7	Các tham số của hàm
$t0-$t7	$8-$15	Thanh ghi tạm (không giữ giá trị trong quá trình gọi hàm)
$s0-$s7	$16-$23	Thanh ghi lưu trữ (giữ giá trị trong suốt quá trình gọi hàm)
$t8-$t9	$24-$25	Thanh ghi tạm
$k0-$k1	$26-$27	Dự trữ cho nhân OS
$gp	$28	Con trỏ toàn cục
$sp	$29	Con trỏ stack
$fp	$30	Con trỏ frame
$ra	$31	Địa chỉ trả về



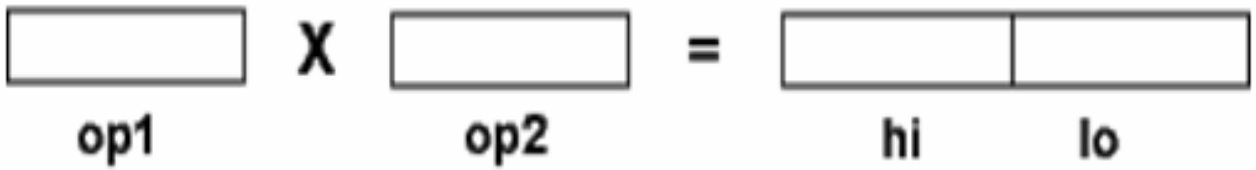
Register Conventions

Figure 5.2 Registers and data sizes in MiniMIPS.

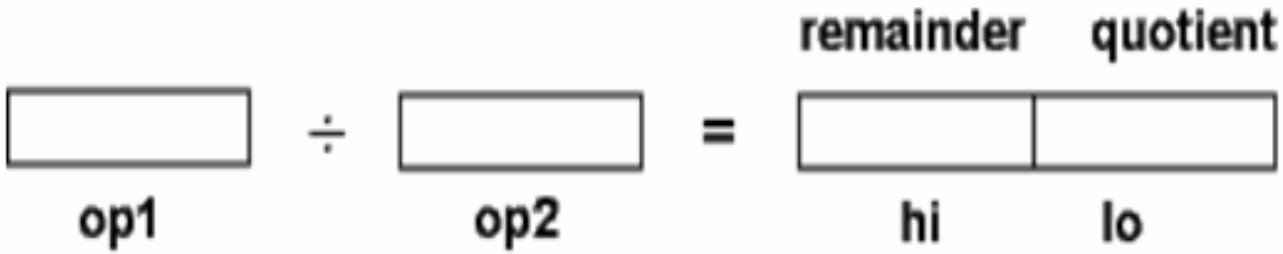
Các thanh ghi của MIPS

b)Thanh ghi HI và LO

Thao tác nhân của MIPS có kết quả chứa 2 thanh ghi HI và LO, đây không phải là thanh ghi đa năng. Bit 32 đến 63 thuộc Hi và 0 đến 31 thuộc LO



Tương tự với phép chia :



Sử dụng các thanh ghi trong MARS

- Phải có kí tự $ ở trước
- Có 2 cách:
  - Địa chỉ thanh ghi. Ví dụ: $8, $19…
  - Tên gợi nhớ. Ví dụ $s0, $t3…
- Ví dụ:
  - add $s0, $6, $zero

Kiến trúc tập lệnh

3 loại lệnh:

- I-Type (Immediate)
- J-Type (Jump and branch)
- R-Type (Register)

MiniMIPS Instruction Formats

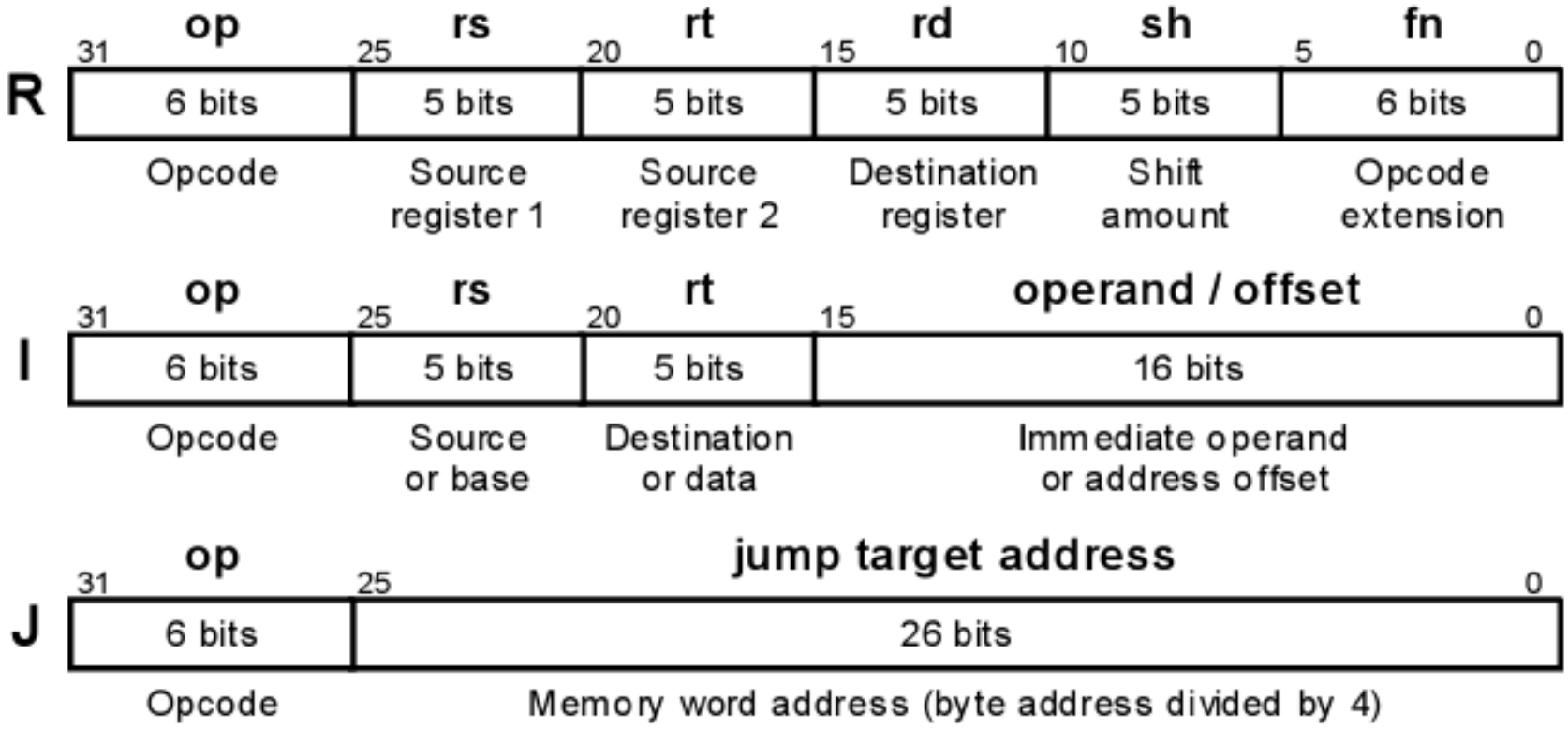


Figure 5.4 MiniMIPS instructions come in only three formats: register (R), immediate (I), and jump (J).

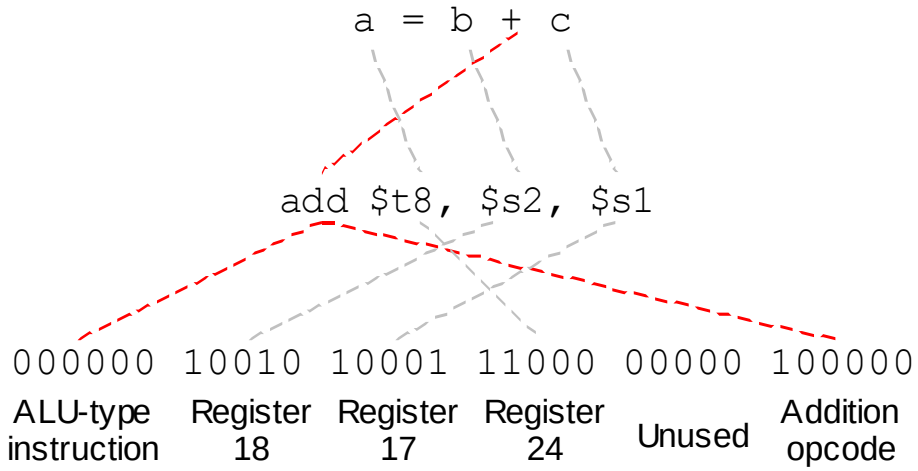
Các khuôn dạng lệnh

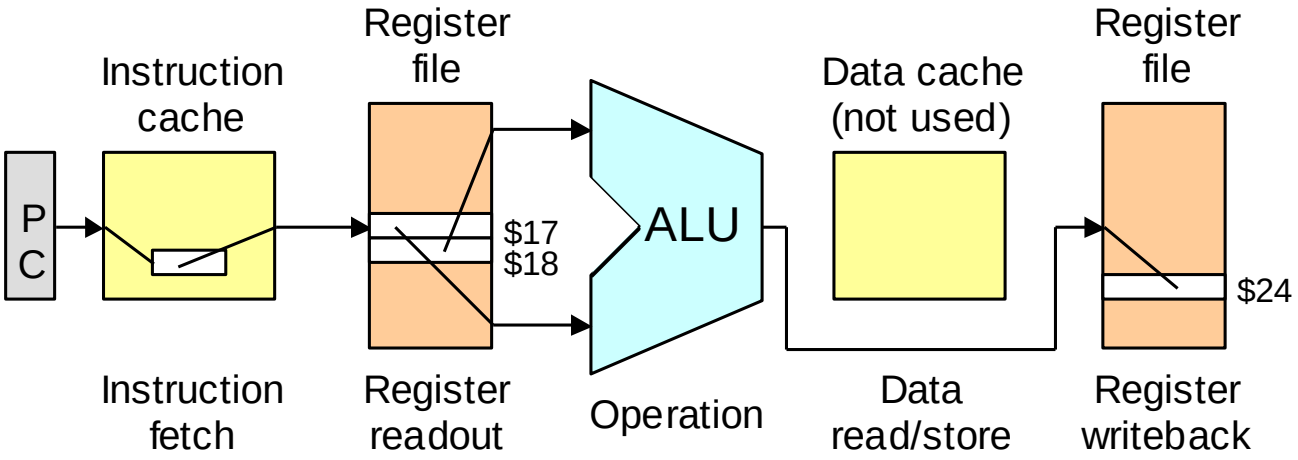
Phân tích khuôn dạng lệnh

High-level language statement:

Assembly language instruction:

Machine language instruction:



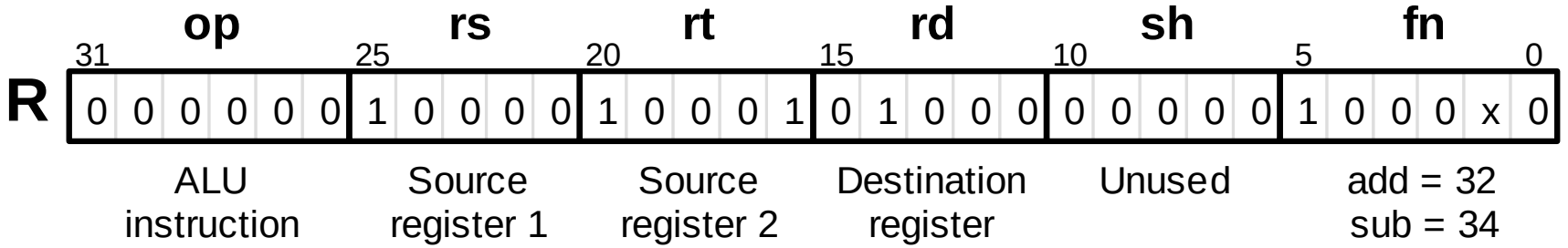


A typical instruction for MiniMIPS and steps in its execution.

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))′
```



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

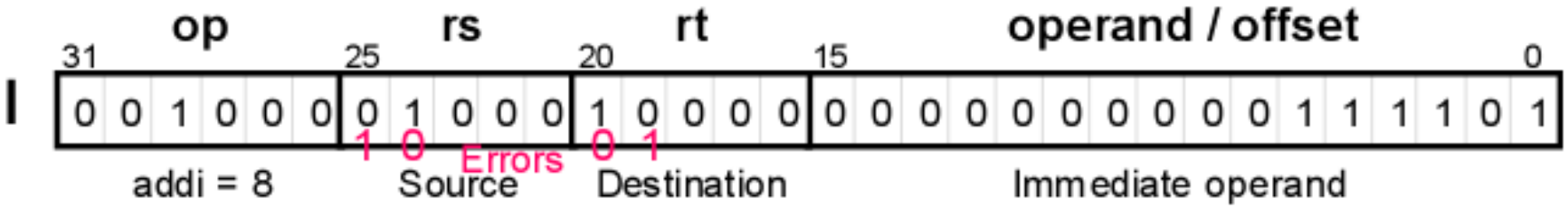
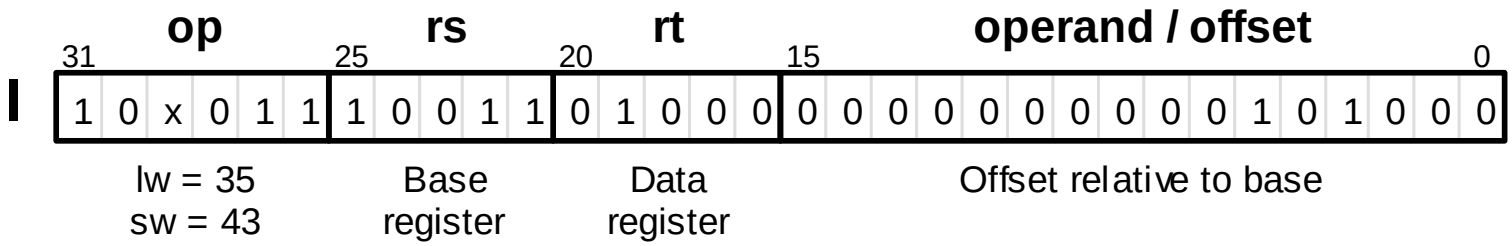
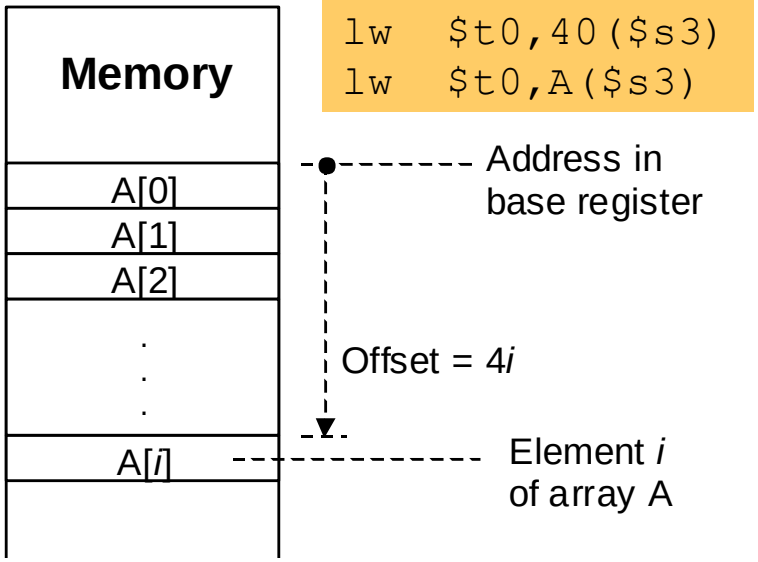


Figure 5.6 Instructions such as `addi` allow us to perform an arithmetic or logic operation for which one operand is a small constant.

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
```

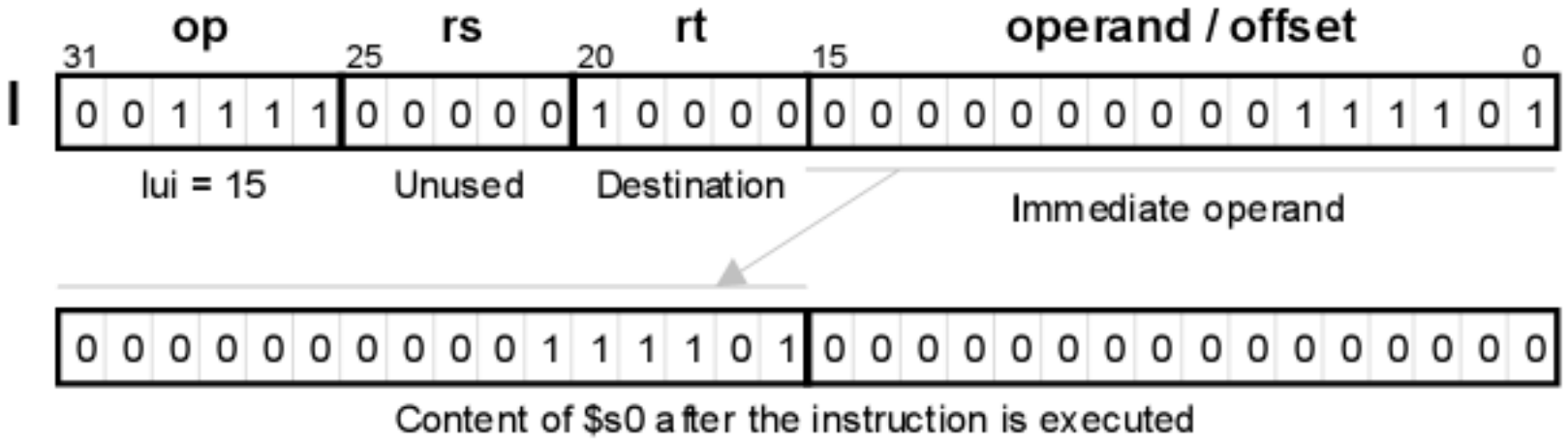


Figure 5.8 The `lui` instruction allows us to load an arbitrary 16-bit value into the upper half of a register while setting its lower half to 0s.

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)

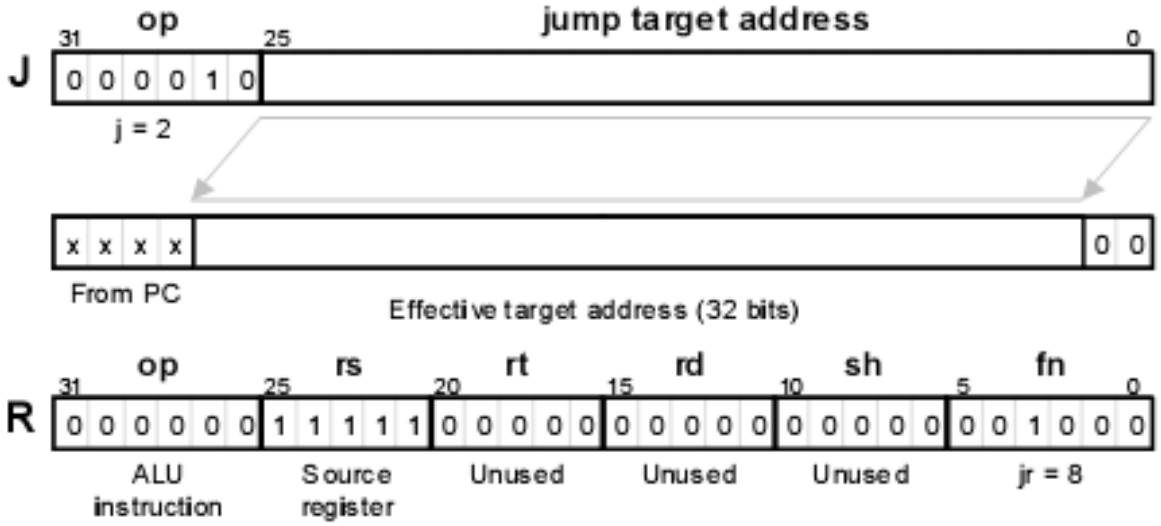


Figure 5.9 The jump instruction `j` of MiniMIPS is a J-type instruction which is shown along with how its effective target address is obtained. The jump register (`jr`) instruction is R-type, with its specified register often being `$ra`.

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)
```

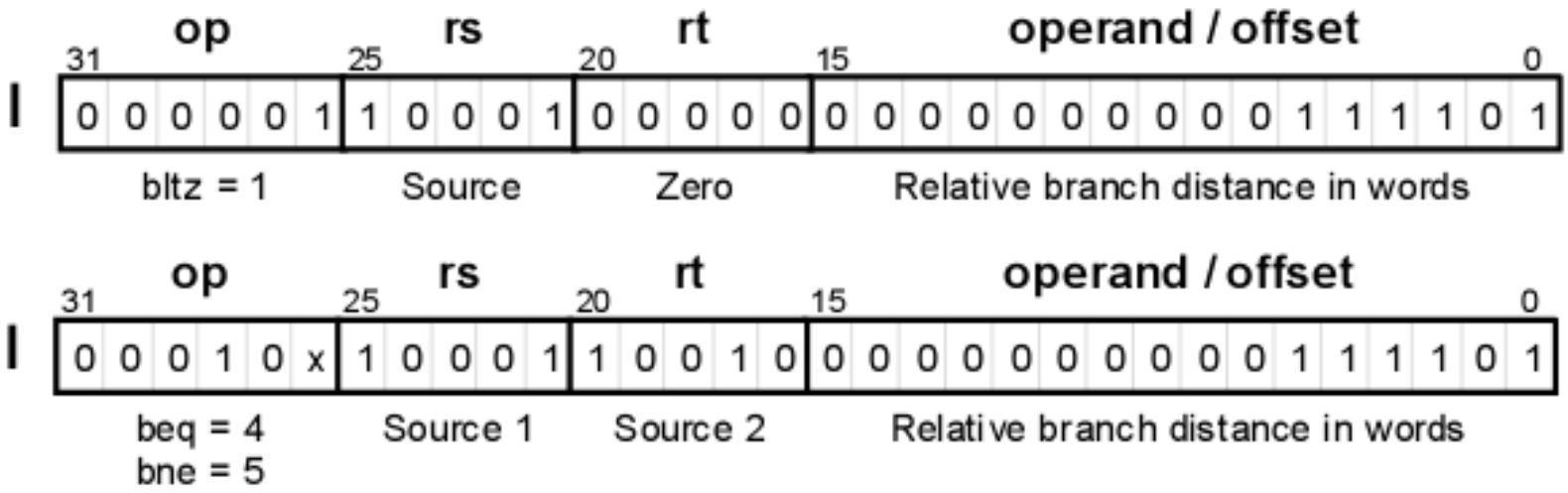


Figure 5.10 (part 1) Conditional branch instructions of MiniMIPS.

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:
```

5.6 Addressing Modes

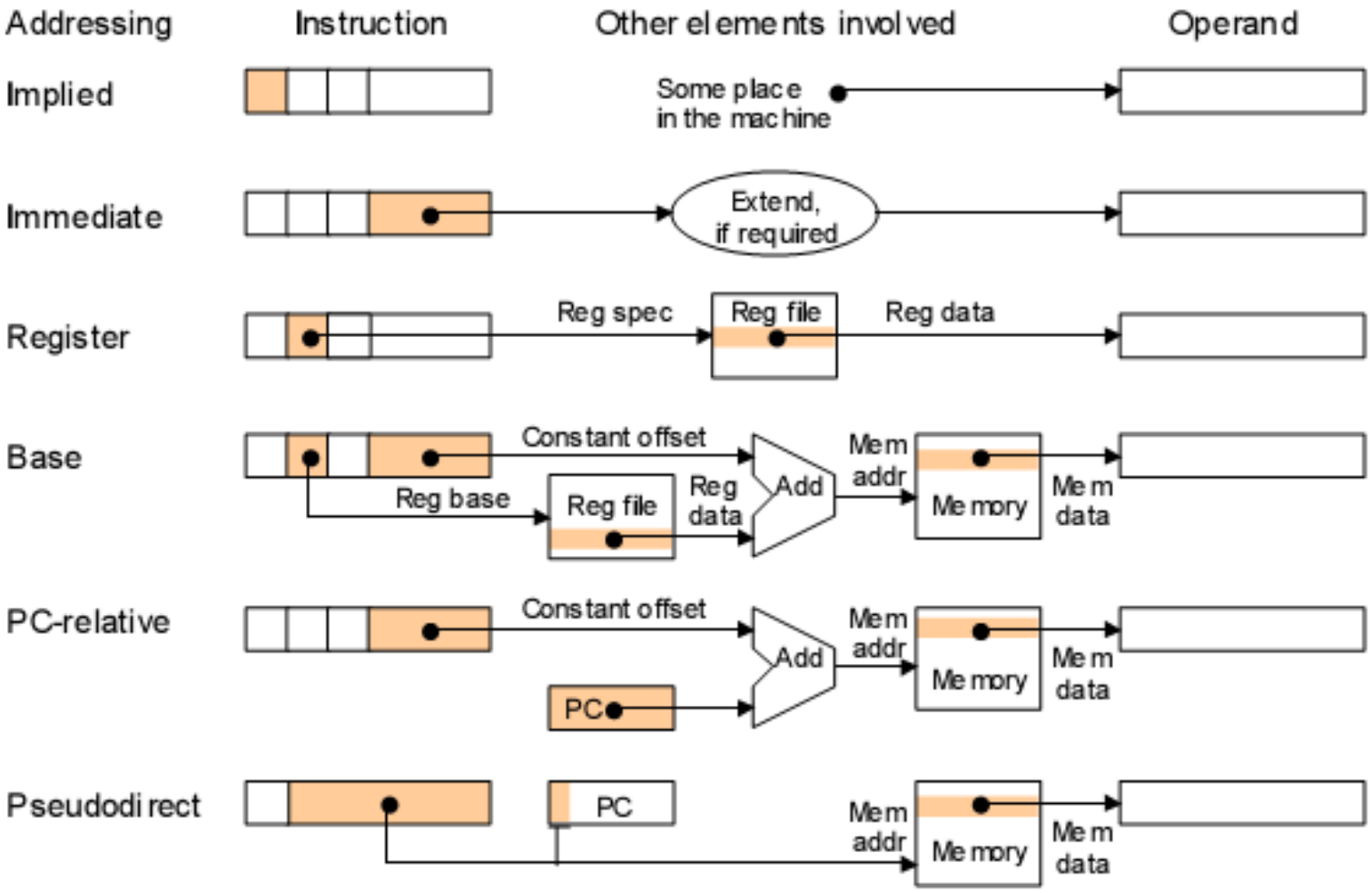


Figure 5.11 Schematic representation of addressing modes in MiniMIPS.

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

PSEUDO INSTRUCTION

- Là “lệnh giả”
- Thực chất khi thực hiện “lệnh giả”, vi xử lý phải thực hiện 1 hay 1 số *Real Instruction* nào đó .
- Ví dụ: `abs $t0,$s0 # $t0=|$s0|`

chính là các lệnh *real* sau:

```
add $t0,$s0,$zero     # lưu giá trị x vào $t0
slt $at,$t0,$zero     # x có là số âm?
beq $at,$zero,+4   # nếu x không âm nhảy đến
lệnh tiếp theo
sub $t0,$zero,$s0     # x có là số dương?
```

Chương trình con và Stack

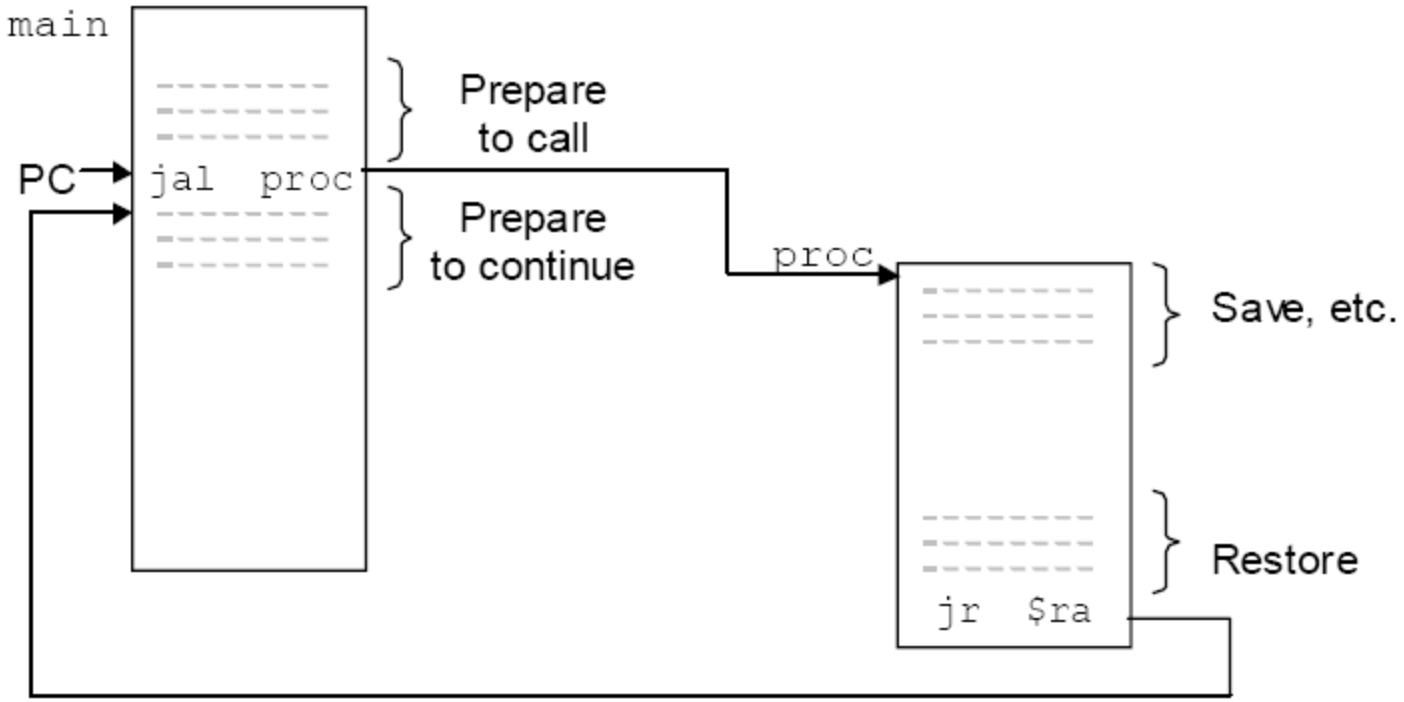
- Để gọi chương trình con: ta sử dụng lệnh

  `jal ( jump and link)`

- Khi đó để trở lại thân hàm chính, ta dùng lệnh

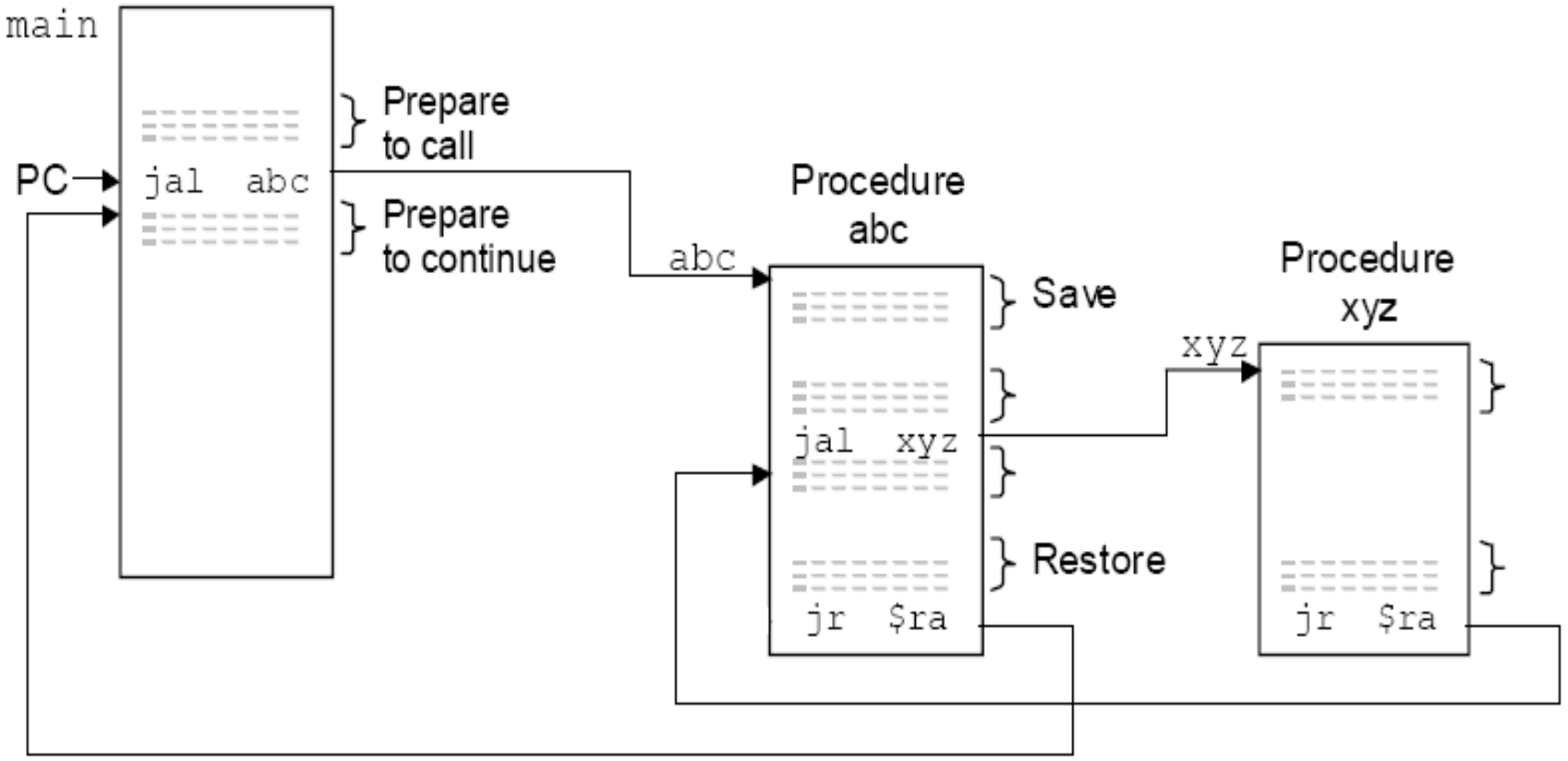
  `jr $ra`

Gọi chương trình con



Hàm lồng trong hàm

Nested Procedure Calls

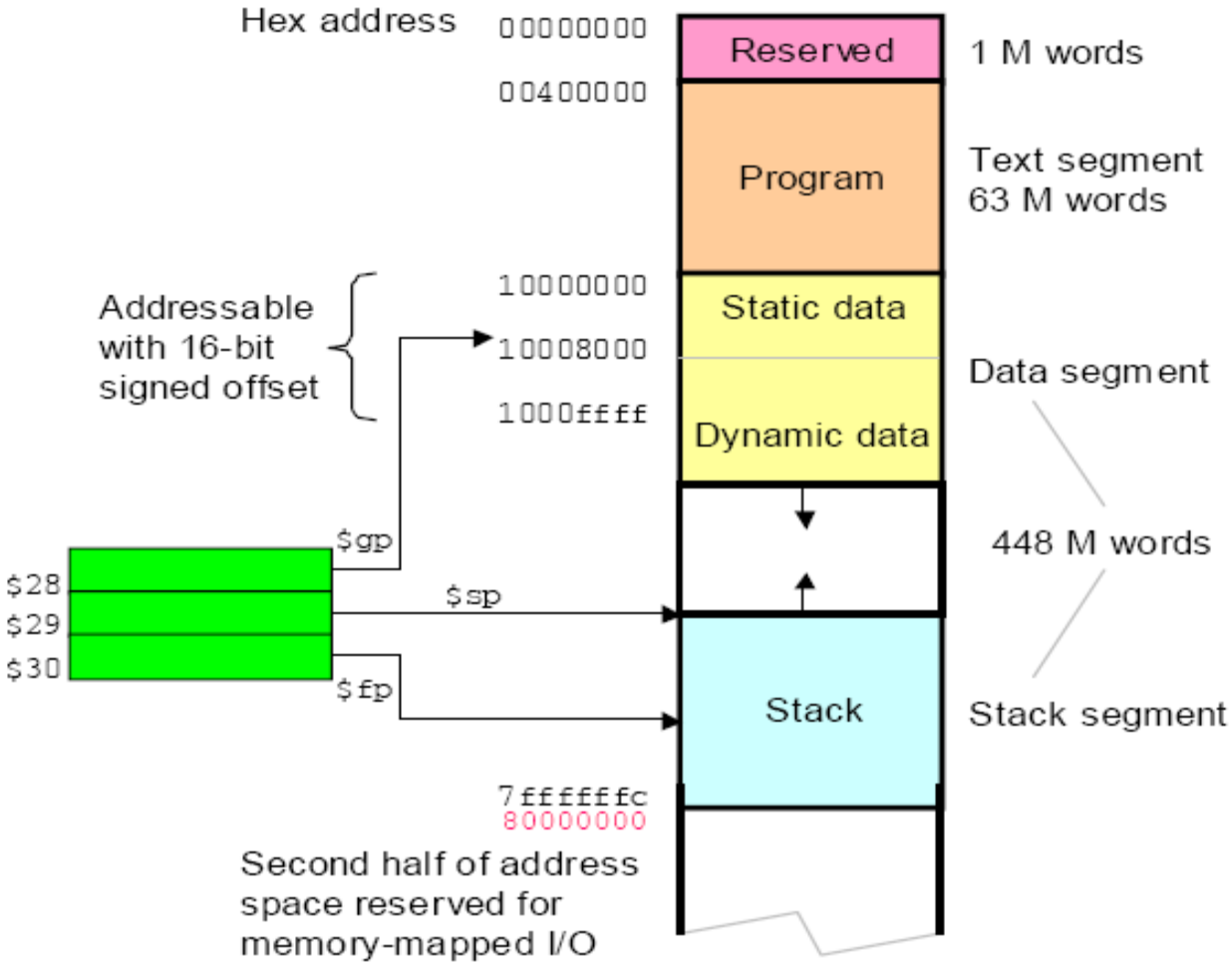


Stack

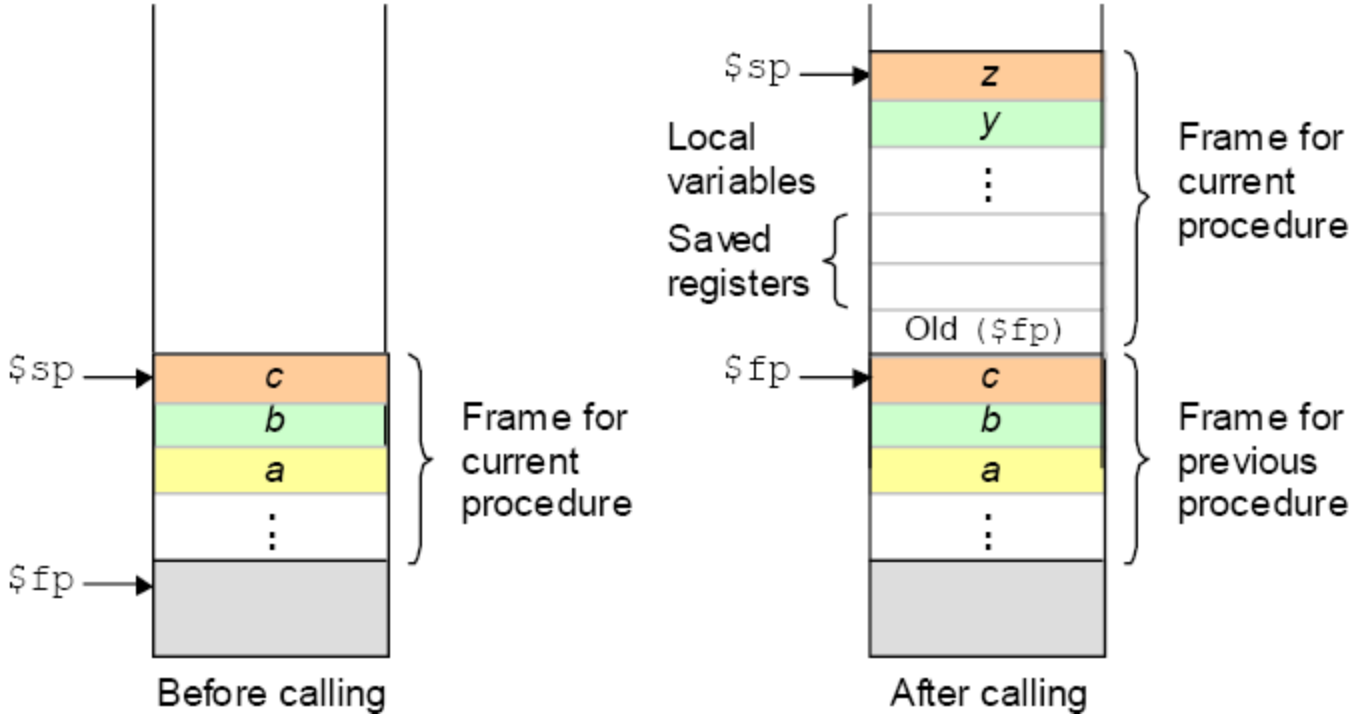

```
push: addi   $sp,$sp,-4
      sw     $t4,0($sp)
```

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

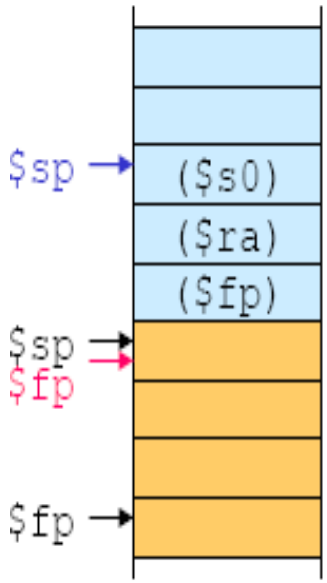
Memory Map


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 và $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

Ví dụ về $sp và $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
```

Khung chương trình hợp ngữ

Giống 8086, chương trình hợp ngữ cho MIPS bao gồm các thành phần

- Định hướng biên dịch
- Lệnh
- Giả lệnh

Khung chương trình hợp ngữ

```
#include <iregdef.h>
.data
#Khai báo biến
.text
.globl start
.ent start
start:
#Nội dung chương trình chính
.end start
.ent CTCon
CTCon:
#Nội dung chương trình con
.end CTCon
```

Chương trình ví dụ

```
#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