Kiến trúc Máy tính

Khoa học & Kỹ thuật Máy tính

Chương 2

Ngôn ngữ Máy: Tập lệnh

BK TP.HCM

Các thành phần & Cấu trúc

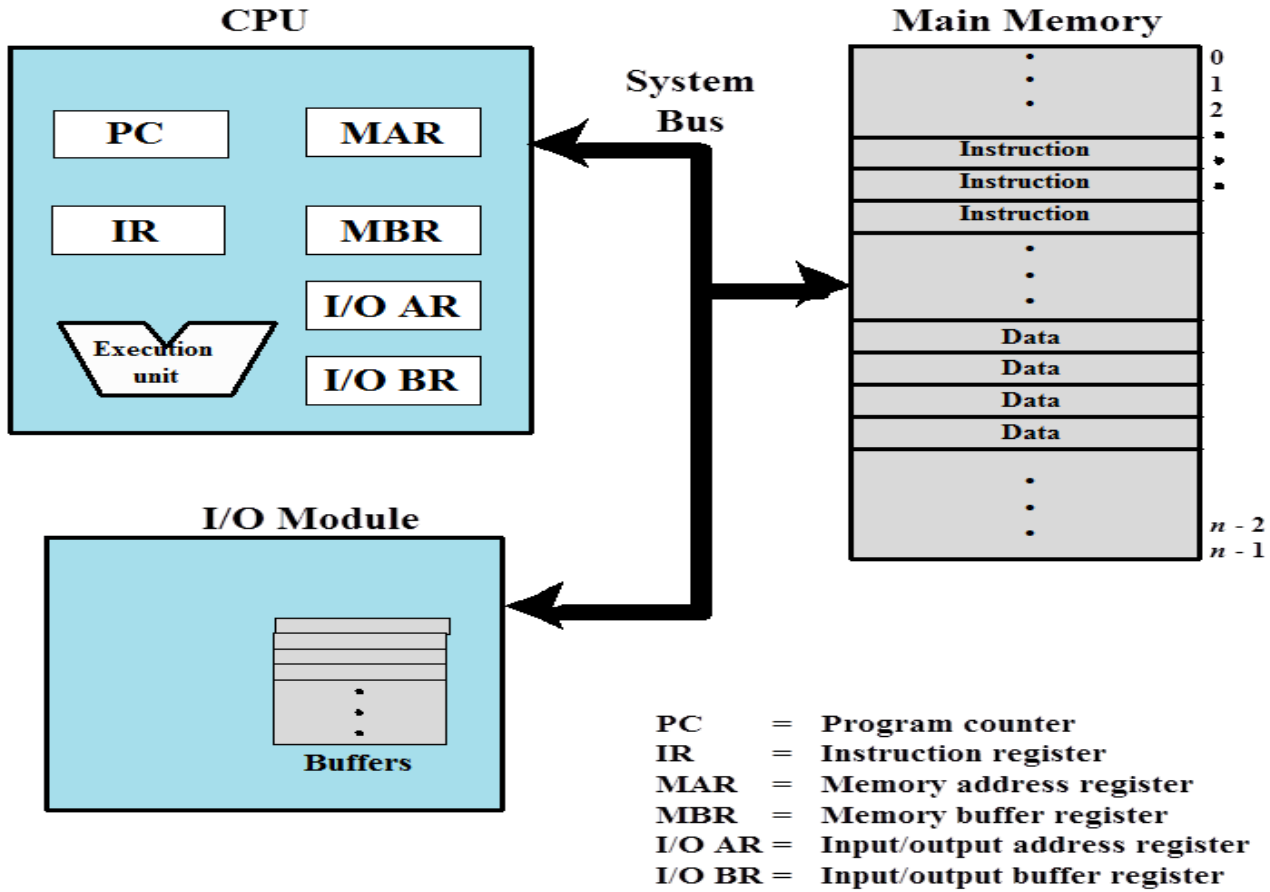


Figure 1.1 Computer Components: Top-Level View

Các bước thực hiện lệnh

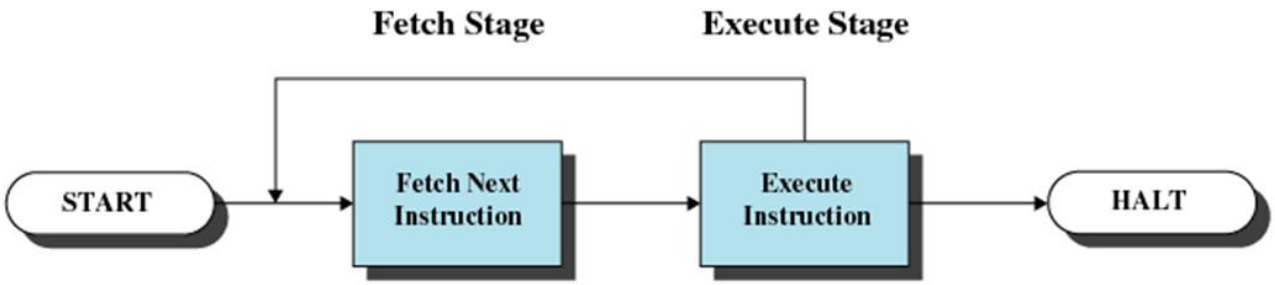


Figure 1.2 Basic Instruction Cycle

- Nạp lệnh: từ bộ nhớ
  - PC tăng lên sau mỗi lần nạp lệnh
  - PC lưu địa chỉ lệnh kế tiếp
- Thực hiện lệnh: giải mã & thực hiện lệnh

Tập lệnh (Instruction Set)

- Tập các lệnh của 1 máy tính
- Máy tính khác nhau có các tập lệnh khác nhau
  - Tuy vậy, có thể có nhiều điểm giống nhau
- Máy tính ở các thế hệ trước thường có tập lệnh rất đơn giản
  - Lý do: dễ thực hiện
- Một số máy tính hiện nay cũng có tập lệnh đơn giản

Tập lệnh MIPS

- Được sử dụng trong môn học này
- Stanford MIPS được thương mại hóa bởi MIPS Technologies (www.mips.com)
- Có thị phần lớn với lõi nhúng (embedded core)
  - Ứng dụng trong thiết bị điện tử, Mạng, lưu trữ, Camera, máy in, v.v., …
- Đặc thù cho nhiều kiến trúc tập lệnh mới
  - Tham khảo MIPS Data tear-out card, và trong phụ lục B, E của sách giáo khoa

Phép tính số học

- Phép cộng (+) và trừ (-): 3 toán hạng
  - 2 nguồn và 1 đích

```
add a, b, c   # a = b + c
```

- Các phép tính số học đều có dạng trên
- *Nguyên tắc thiết kế 1:* Đơn giản dễ tạo tính quy tắc
  - Tính quy tắc sẽ đơn giản hơn việc thực hiện
  - Đơn giản sẽ nâng hiệu xuất, giảm giá thành.

Ví dụ: thực hiện phép số học

- C code:

```
f = (g + h) - (i + j);
```

- Sau khi biên dịch thành MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Toán hạng là thanh ghi

- Có nhiều lệnh số học sử dụng các thanh ghi làm toán hạng
- MIPS có tệp 32 thanh ghi 32-bit
  - Use for frequently accessed data
  - Đánh số từ 0 đến 31
  - 32-bit dữ liệu được gọi là 1 "từ" ("word")
- Được đặt tên gợi nhớ (Ass. Names):
  - $t0, $t1, …, $t9 chứa các giá trị tạm thời
  - $s0, $s1, …, $s7 chứa các biến
- *Nguyên tắc thiết kế 2:* Càng nhỏ, càng nhanh
  - Ngược lại với bộ nhớ chính: hàng triệu ô nhớ.

Ví dụ: toán hạng thanh ghi

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j chứa trong $s0, …, $s4

- Sau khi biên dịch thành MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Toán hạng là bộ nhớ

- Bộ nhớ chính dùng để lưu trữ toán hạng có cấu trúc
  - Arrays, structures, dynamic data
- Sử dụng cho các phép số học
  - Nạp các giá trị từ bộ nhớ vào các thanh ghi
  - Lưu giữ các kết quả trong thanh ghi ra bộ nhớ
- Bộ nhớ được định vị theo đơn vị từng byte
  - Mỗi địa chỉ định vị trí cho một 8-bit byte
- 1 từ được sắp xếp gồm 4 bytes trong bộ nhớ
  - Địa chỉ truy xuất = Địa chỉ biểu diễn * 4 byte
- MIPS chứa dữ liệu theo Big Endian
  - Big Endian: Byte có giá trị lớn nằm ở địa chỉ thấp
  - Little Endian: Byte có giá trị nhỏ nhất → Địa chỉ thấp

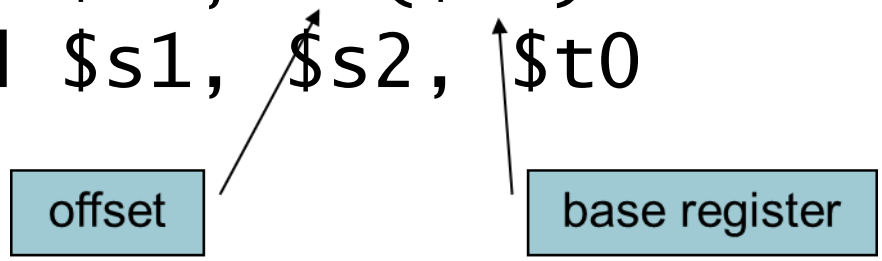
Ví dụ 1: Toán hạng bộ nhớ

- C code:

  `g = h + A[8];`

  - g chứa trong $s1, h trong $s2, địa chỉ cơ sở của A chứa trong $s3
- Sau khi biên dịch thành MIPS code:
  - Chỉ số 8 tương đương với độ dời 32
    - 4 bytes/word

```
lw  $t0, 32($s3)     # Nạp 1 từ (4bytes)
add $s1, $s2, $t0
```



Ví dụ 2: Toán hạng bộ nhớ

- C code:

```
A[12] = h + A[8];
```

  - h chứa trong $s2, địa chỉ cơ sở của A chứa trong $s3

- Sau khi biên dịch thành MIPS code:
  - Chỉ số 8 tương đương với độ dời 32

```
lw  $t0, 32($s3)     # Nạp 1 từ
add $t0, $s2, $t0
sw  $t0, 48($s3)     # Nhớ 1 từ
```

So sánh toán hạng thanh ghi & bộ nhớ

- Truy cập toán hạng thanh ghi nhanh hơn bộ nhớ
- Thực hiện toán hạng bộ nhớ cần nạp và cất dữ liệu → cần nhiều lệnh thực hiện hơn
- Trình biên dịch yêu cầu các biến chứa trong thanh ghi tối đa
  - Chỉ chứa các biến trong bộ nhớ khi chúng ít được dùng đến
  - Tối ưu thanh ghi rất quan trọng!

Toán hạng trực tiếp

- Các dữ liệu hằng trong 1 lệnh, như
  `addi $s3, $s3, 4`
- Không tồn tại lệnh trừ với toán hạng trực tiếp (?????)
  - Tương đương với cộng 1 số âm
    `addi $s2, $s1, -1`
- *Nguyên tắc thiết kế 3:* Làm cho các trường hợp phổ biến thực hiện nhanh
  - Hằng có giá trị nhỏ rất phổ biến
  - Toán hạng trực tiếp trách được lệnh nạp

Thanh ghi Hằng 0 (Zero)

- Thanh ghi MIPS 0 ($zero) là hằng cố định có giá trị 0
  - Giá trị không thay đổi được
- Có ích cho các tác vụ thường gặp như:
  - Ví dụ, gán giá trị một thanh ghi cho thanh ghi khác

```
add $t2, $s1, $zero # $t2 = $s1
```

Số nguyên nhị phân không dấu

- Cho 1 số n-bit, có dạng

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Tầm vực giá trị sẽ là: 0 đến $+2^n - 1$
- Ví dụ:
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Giá trị 1 số nhị phân không dấu 32-bit sẽ là:
  - 0 đến +4,294,967,295 (giá trị thập phân)

Số nguyên có dấu dạng bù 2

- Cho 1 số n-bit như sau:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Tầm giá trị: $-2^{(n-1)}$ đến $+2^{(n-1)} - 1$
- Ví dụ:
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Giá trị 1 số nhị phân có dấu 32-bit sẽ là
  - –2,147,483,648 đến +2,147,483,647

Số nguyên có dấu dạng bù 2 (tt.)

- Bit 31 là bit dấu
  - 1 có nghĩa là số âm (-)
  - 0 có nghĩa là số không âm (+)
- Dạng $-(-2^{n-1})$ không tồn tại
- Các số không âm biểu diễn giống số không dấu và số bù 2
- Vài số đặc biệt như:
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Số âm nhỏ nhất: 1000 0000 … 0000
  - Số dương lớn nhất: 0111 1111 … 1111

Số âm có dấu

- Đảo giá trị bit và cộng 1
  - Đảo giá trị bit: 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Ví dụ: giá trị (-) 2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

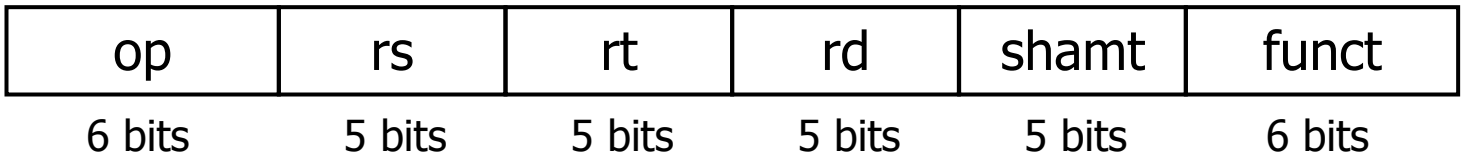
Mở rộng bit với số có dấu

- Biểu diễn với số bit nhiều hơn
  - Dữ nguyên giá trị
- Ví dụ: Trong tập lệnh MIPS
  - `addi`: mở rộng số bit giá trị toán hạng trực tiếp
  - `lb`, `lh`: mở rộng số bit với byte/(1/2 từ) được nạp
  - `beq`, `bne`: mở rộng số bit của độ dời địa chỉ
- Thêm giá bit dấu vào các bit mở rộng bên trái
  - Đối với giá trị không dấu: gán 0s
- Ví dụ: chuyển số 8-bit thành số 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Biểu diễn lệnh

- Lệnh được mã hóa thành giá trị nhị phân
  - Gọi là mã máy
- Các lệnh của MIP
  - Mã hóa thành từ lệnh 32-bit
  - Chia thành các phần nhỏ: Mã lệnh, thanh ghi, ..
  - Theo quy tắc!
- Các thanh ghi MIP được đánh số:
  - $t0 – $t7 tương ứng với thanh ghi 8 – 15
  - $t8 – $t9 tương ứng với thanh ghi 24 – 25
  - $s0 – $s7 tương ứng với thanh ghi 16 – 23

Các lệnh dạng R



- Cấu trúc thành phần của lệnh dạng R
  - op: Mã lệnh (opcode)
  - rs: Chỉ số thanh ghi nguồn thứ nhất
  - rt: Chỉ số thanh ghi nguồn thứ nhì
  - rd: Chỉ số thanh ghi đích
  - shamt: Số bit dịch chuyển
  - funct: mã chức năng mở rộng (extends opcode)

Ví dụ: Lệnh dạng R

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

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Biểu diễn số dạng hệ 16

- Hệ số 16
  - Rút gọn cách biểu diễn chuỗi nhị phân
  - 4 bits cho mỗi số hex

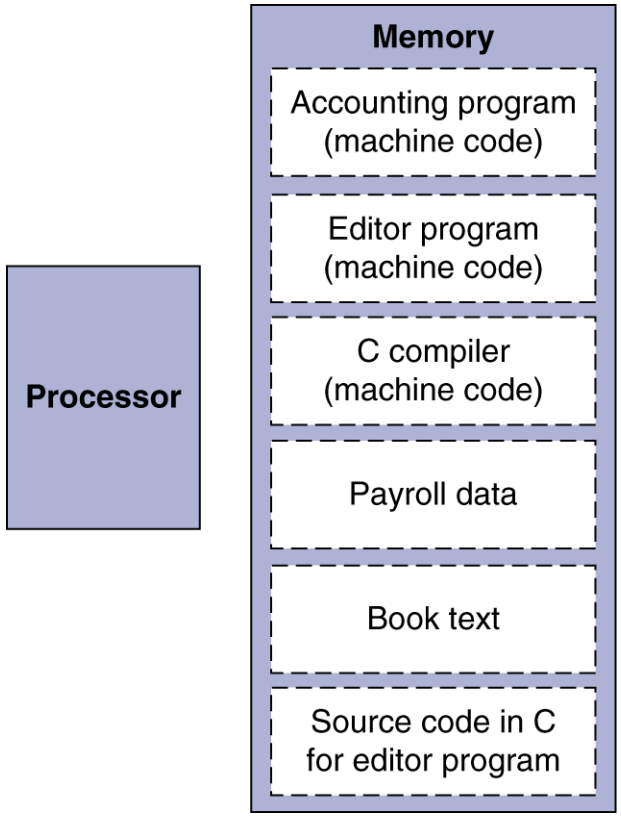
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Ví dụ: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Lệnh MIPS dạng I

- Các lệnh số học trực tiếp hoặc lệnh nạp/cất
  - rt: Thanh ghi đích hoặc nguồn
  - Nếu là hằng: -2^{15} to $+2^{15} - 1$
  - Nếu là địa chỉ: Độ dời + địa chỉ cơ sở chứa trong rs
- *Nguyên tắc thiết kế 4:* Thiết kế tốt yêu cầu sự kết hợp hợp lý
  - Nhiều dạng lệnh làm phức tạp giải mã, nhưng cho phép lệnh chứa đồng nhất chỉ trong 32-bit
  - Giữ dạng lệnh càng giống nhau càng tốt

Tổ chức chương trình



- Lệnh được biểu diễn dạng nhị phân, giống như dữ liệu
- Lệnh và dữ liệu được lưu trong bộ nhớ
- Các chương trình có thể thực hiện trên các chương trình khác, ví dụ: compilers, linkers, …
- Tương thích nhị phân cho phép chương trình thực hiện trên các máy khác nhau → ISA chuẩn

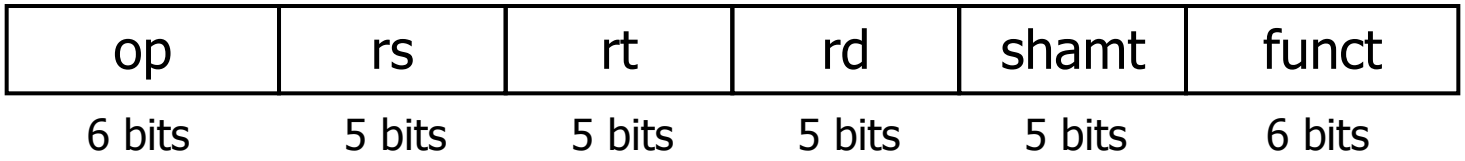
Tác tác vụ luận lý

- Các lệnh xử lý bit

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Có tác dụng rút trích hoặc thêm nhóm bit vào 1 từ

Các tác vụ dịch (shift)

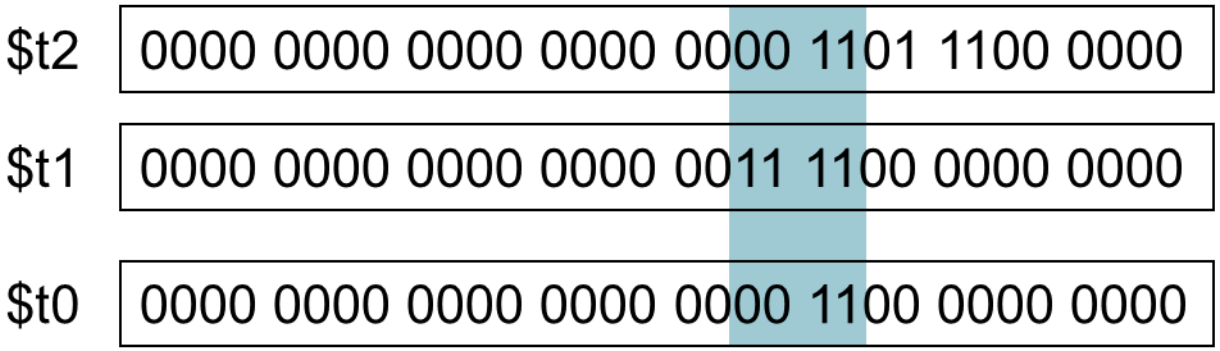


- shamt: dịch vị trí các bits
- Dịch trái
  - Dịch trái các bit *n* vị trí và gán *n* bit bên phải giá trị 0
  - `sll` bởi *i* bits có nghĩa nhân 2^i
- Dịch phải
  - Dịch phải các bit *n* vị trí và gán *n* bit bên trái giá trị 0
  - `srl` bởi *i* bits có nghĩa chia 2^i (chỉ không dấu)

Tác vụ "VÀ" (AND)

- Dùng để đánh dấu các bits trong 1 từ
  - Chọn một số bits, xóa số còn lại về 0

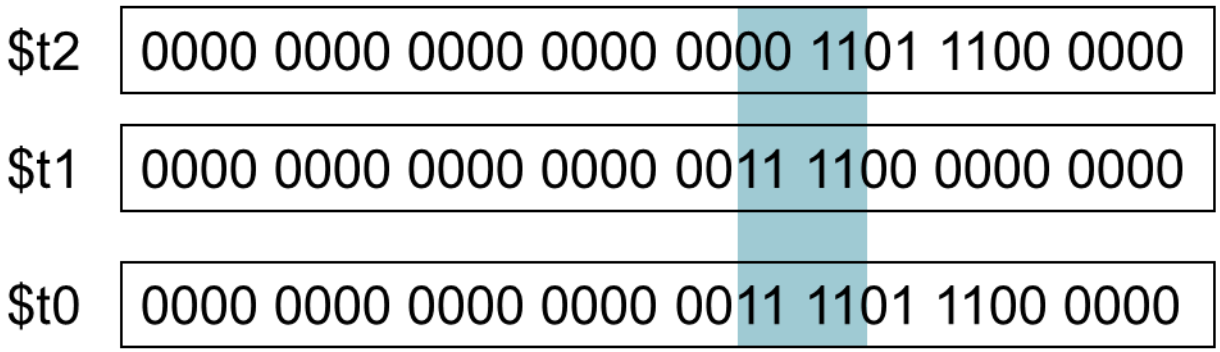
```
and $t0, $t1, $t2
```



Tác vụ "hoặc" (OR)

- Thêm 1 số bit vào 1 từ
  - Gán giá trị 1 nhóm bit thành 1 trong khi giữ nguyên giá trị các bit còn lại

```
or $t0, $t1, $t2
```



Các tác vụ "Not"

- Có tác dụng đảo giá trị các bit trong 1 từ: đổi 0 thành 1, và 1 thành 0
- MIPS có toán tử NOR với 3 toán hạng
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

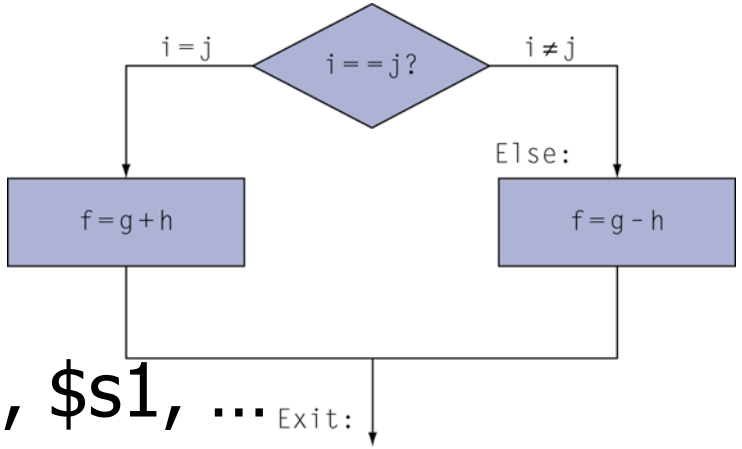
Các tác vụ điều kiện

- Rẽ nhánh đến 1 lệnh có nhãn, nếu điều kiện thỏa
  - Nếu không thỏa, tiếp tục
- `beq rs, rt, L1`
  - Nếu (rs == rt), nhảy đến lệnh có nhãn L1;
- `bne rs, rt, L1`
  - Nếu (rs != rt), nhảy đến lệnh có nhãn L1;
- `j L1`
  - Nhảy vô điều kiện đến lệnh có nhãn L1

Biên dịch các phát biểu if

- C code:

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



  - f, g, … chứa trong $s0, $s1, …
- Sau khi biên dịch thành MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Assembler calculates addresses

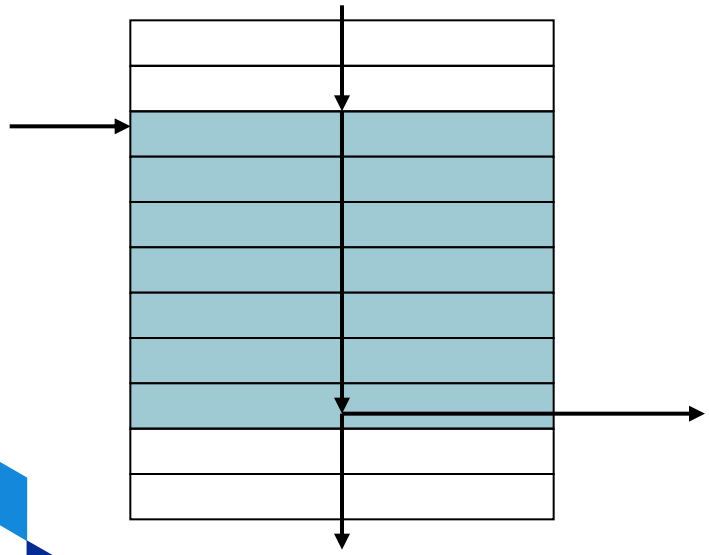
Biên dịch các phát biểu Loop

- C code:
```
while (save[i] == k) i += 1;
```
  - i chứa trong $s3, k trong $s5, địa chỉ của save chứa trong $s6

- Sau khi biên dịch thành MIPS code:
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Khối căn bản (Basic Blocks)

- Một khối chứa tuần tự các lệnh, trong đó
  - Không có rẽ nhánh đi (except at end)
  - Không chứa địa chỉ đích đến (except at beginning)



- Biên dịch sẽ nhận biết khối này để tối ưu kết quả dịch
- Tăng nhanh việc xử lý các lệnh trong khối này

Các tác vụ kiểm tra điều kiện khác

- Gán kết quả là 1, nếu điều kiện thỏa
  - Nếu không thỏa, gán là 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Sử dụng kết hợp với lệnh beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Thiết kế lệnh rẽ nhánh

- Tại sao không có lệnh `blt`, `bge`, etc?
- <, ≥, Thực hiện phần cứng chậm hơn =, ≠
  - Khi kết hợp với rẽ nhánh sẽ phải thực hiện nhiều việc hơn → yêu cầu xung đồng hồ chậm hơn
  - All instructions penalized! (không thống nhất cho các lệnh)
- `beq` và `bne`: trường hợp thường xảy ra
- Đó là sự kết hợp tốt

Dấu và Không dấu

- So sánh có dấu: `slt`, `slti`
- So sánh không dấu: `sltu`, `sltui`
- Ví dụ
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  #` **có đấu**
    - −1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  #` **không dấu**
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Ví dụ: Case/Switch

Dịch đoạn mã C sau đây sang hợp ngữ MIPS

```
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 ;
}
```

- Giả sử các biến *f* đến *k* tương ứng với *$s0* đến *$s5*, thanh ghi *$t2* mang giá trị 4

Ví dụ: Case/Switch

```
slt $t3 , $s5 , $zero
bne $t3 , $zero , Exit
slt $t3 , $s5 , $t2
beq $t3 , $zero , Exit
add $t1 , $s5 , $s5
add $t1 , $t1 , $t1
add $t1 , $t1 , $t4
lw $t0, 0($t1)
jr $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: ..
```

Gọi thủ tục

- Các bước thực hiện gọi thủ tục
  1. Chuyển thông số vào vị trí (thanh ghi)
  2. Chuyển quyền điều khiển cho thủ tục
  3. Nhận tài nguyên lưu trữ cho thủ tục
  4. Thực hiện công việc của thủ tục
  5. Chuyển kết quả vào vị trí (thanh ghi) để trả về cho chương trình gọi
  6. Trở về chương trình gọi

Ý đồ sử dụng các thanh ghi

- $a0 – $a3: chứa thông số (reg's 4 – 7)
- $v0, $v1: giá trị trả về (reg's 2 and 3)
- $t0 – $t9: chứa giá trị tạm
  - Có thể thay đổi nội dung khi thực hiện thủ tục
- $s0 – $s7: bảo vệ
  - Cất/khôi phục bởi thủ tục
- $gp: Con trỏ toàn cục dữ liệu tĩnh (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: Địa chỉ trở về (reg 31)

Lệnh gọi thủ tục

- Gọi thủ tục: jump and link (jal)

  `jal ProcedureLabel`

  - Địa chỉ lệnh kế chứa trong thanh ghi $ra
  - Nhảy đến địa chỉ đích
- Trở về chương trình gọi: jump register

  `jr $ra`

  - Sao giá trị của $ra vào PC
  - Có thể dùng nhảy theo điều kiện
    - Ví dụ: phát biểu case/switch

Ví dụ: gọi thủ tục (leaf)

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Thông số g, …, j chứa trong $a0, …, $a3
  - f trong $s0 (vì vậy, $s0 cất trong stack)
  - Kết quả trả về trong $v0

Ví dụ: gọi thủ tục (tt.)

- Sau khi biên dịch thành MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)      Save $s0 on stack
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3    Procedure body
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero  Result
    lw   $s0, 0($sp)
    addi $sp, $sp, 4      Restore $s0
    jr   $ra              Return
```

Gọi thủ tục (Non-Leaf)

- Thủ tục gọi thủ tục khác
- Gọi đệ quy, thủ tục gọi phải cất vào stack thông tin:
  - Địa chỉ trở về của nó trong thủ tục "cha"
  - Tất cả các thông số và giá trị tạm thời
- Phục hồi từ stack sau khi thủ tục kết thúc

Ví dụ: gọi thủ tục (Non-Leaf)

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

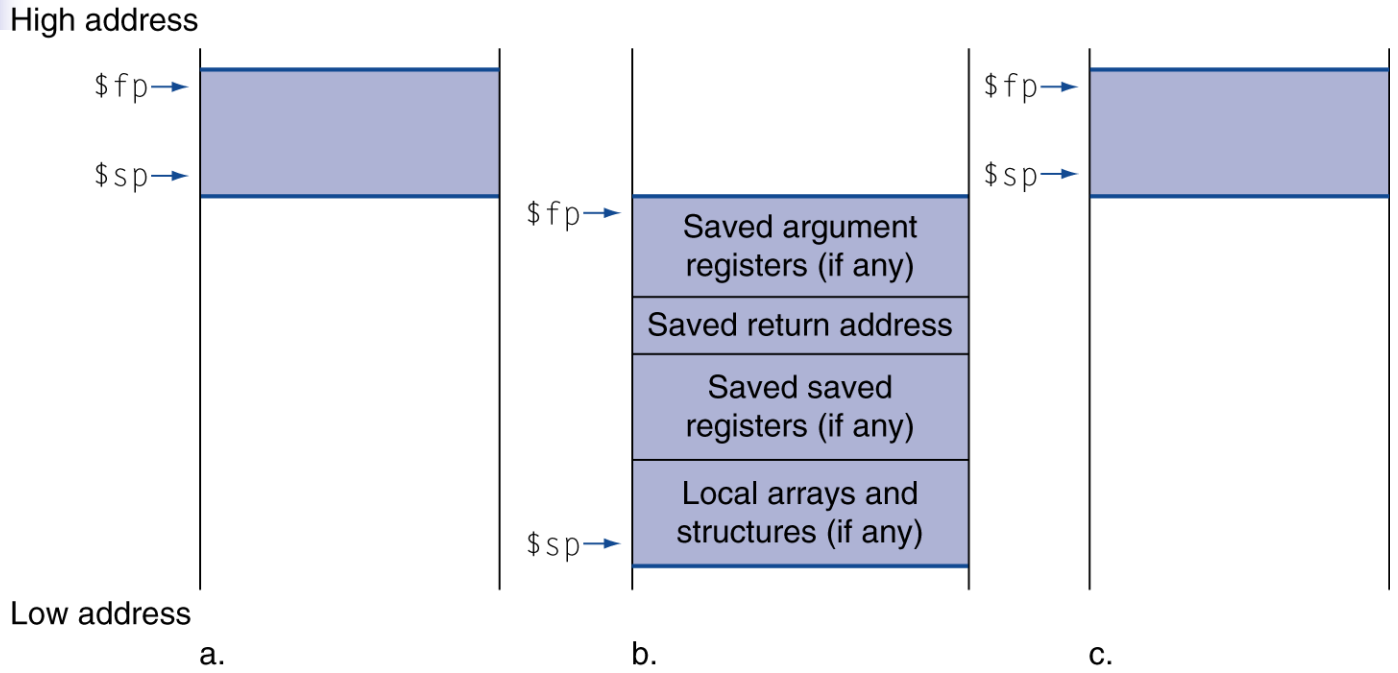
  - Thông số n chứa trong $a0
  - Kết quả trả về chứa trong $v0

Ví dụ: gọi thủ tục (Non-Leaf) tt.

- Sau khi biên dịch thành MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

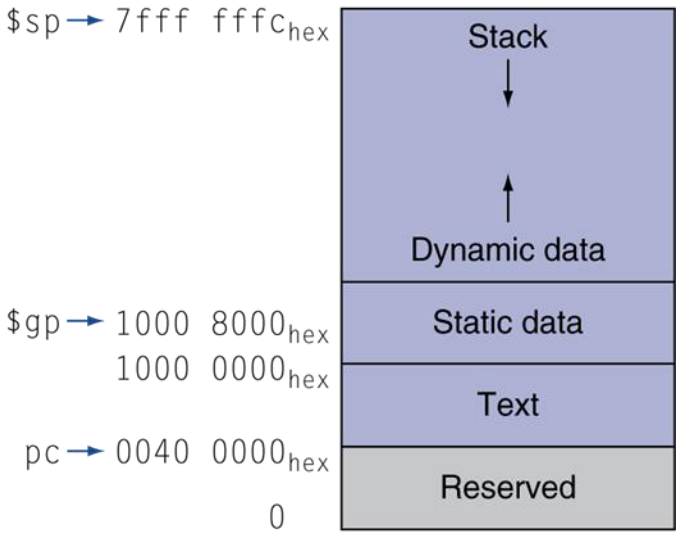
Cách lưu trữ trong Stack



- Dữ liệu cục bộ được cấp phát tại thủ tục
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Compiler sử dụng để quản lý lưu trữ trong stack

Bố cục chứa trong bộ nhớ

- Text: mã lệnh chương trình
- Dữ liệu tĩnh: biến toàn cục
  - Ví dụ: static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dữ liệu động: heap
  - E.g., malloc in C, new in Java
- Stack: lưu trữ tự động



Dữ liệu ký tự

- Tập ký tự dạng Byte-encoded
  - ASCII: 128 Ký tự
    - 95 graphic, 33 điều khiển
  - Latin-1: 256 Ký tự
    - ASCII, +96 ký tự graphics
- Tập ký tự 32-bit dạng Unicode:
  - Sử dụng trong Java, C++ wide characters, …
  - Chứa toàn bộ mã ký tự thế giới, cùng với symbols
  - UTF-8, UTF-16: variable-length encodings

Nhóm các lệnh Byte/Halfword

- Dùng cho các tác vụ xử lý theo bit
- MIPS byte/halfword load/store
  - Xử lý chuỗi khá phổ biến

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Chỉ ghi phần giá trị thấp byte/halfword

Ví dụ: Sao chuỗi (String Copy)

- C code (naïve):
  - Ký tự Null- đánh dấu kết thúc string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Địa chỉ của x, y chứa trong $a0, $a1
  - i chứa trong $s0

Ví dụ: String Copy (tt.)

Sau khi biên dịch thành MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Hằng 32-bit

- Phần lớn các hằng hạn chế trong 16-bit
  - Đáp ứng đủ cho các toán hạng trực tiếp 16-bit
- Với các Hằng lớn hơn (32-bit)

  `lui rt, constant`
  - Sao 16-bit của hằng vào 16 bits bên trái của rt
  - Xóa 16 bits bên phải của rt về 0

```
lui $s0, 61
```

0000 0000 0011 1101	0000 0000 0000 0000

```
ori $s0, $s0, 2304
```

0000 0000 0111 1101	0000 1001 0000 0000

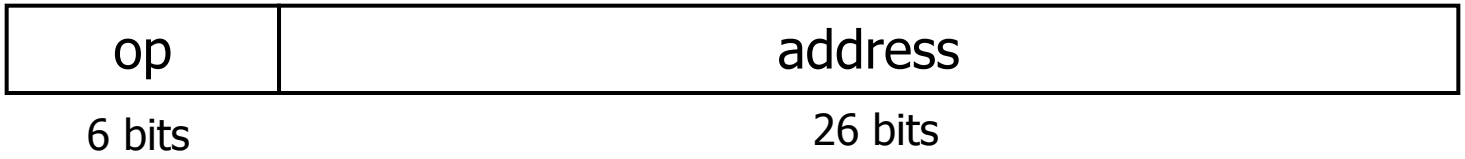
Xác định địa chỉ rẽ nhánh

- Dạng lệnh rẽ nhánh gồm:
  - Opcode, 2 thanh ghi, target address
- Vị trí nhảy đến địa chỉ rẽ nhánh thường gần lệnh rẽ nhánh: nhảy tới hoặc lui

- Tương đối với giá trị PC
  - Địa chỉ đích = PC + offset × 4
  - PC đã tăng lên 4, khi lệnh thực hiện

Địa chỉ nhảy trực tiếp

- Đích của lệnh Jump (`j` and `jal`) bất cứ đâu trong đoạn lệnh chương trình



- (Pseudo) Địa chỉ đích
  - = $PC_{31...28}$: (address × 4)

Ví dụ: Xác định địa chỉ đích

- Sử dụng lại đoạn code vòng lặp trước đây
  - Giả sử Loop bắt đầu từ địa chỉ 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

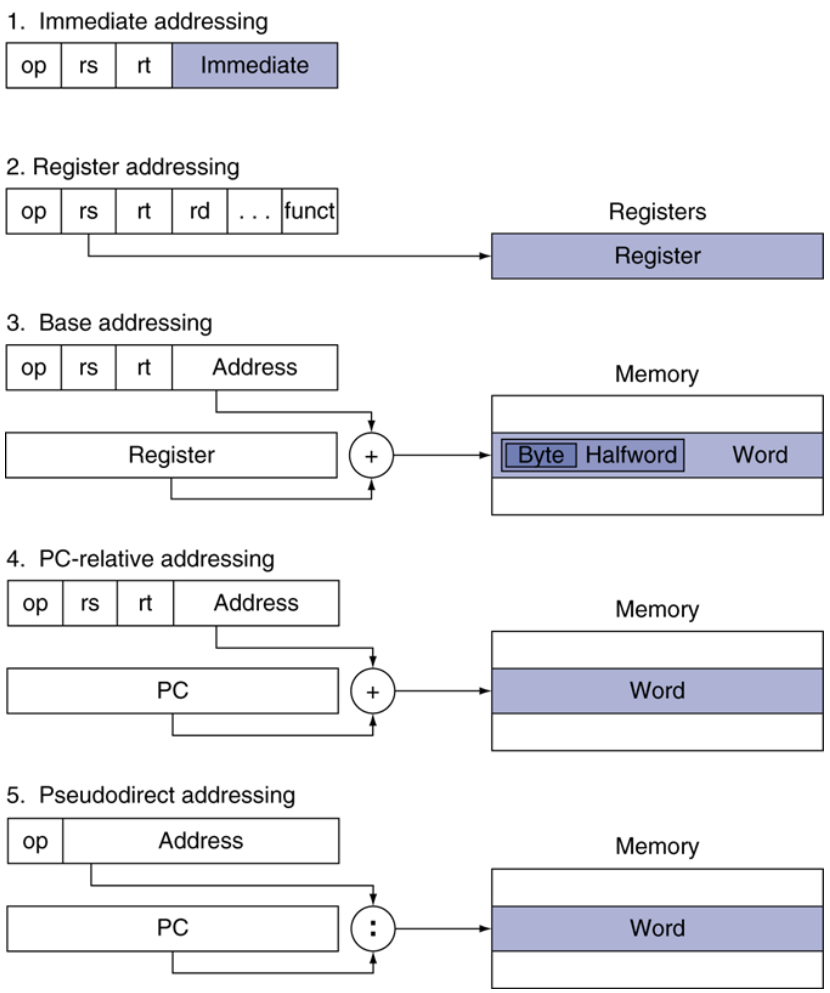
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Rẽ nhánh xa

- Trong trường hợp địa chỉ đích rẽ nhánh quá xa (vượt giá trị độ dời 16-bit), Hợp ngữ sẽ điều chỉnh lại code.
- Ví dụ:

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

Tóm tắt Addressing Mode



Đồng bộ

- Hai bộ xử lý dùng chung 1 vùng bộ nhớ
  - P1 ghi thông tin, sau đó P2 đọc
  - Có sự tranh chấp truy cập, nếu P1 & P2 không đòng bộ với nhau → Kết quả không xác định được
- Hỗ trợ phần cứng yêu cầu
  - Tác vụ Atomic đọc/ghi bộ nhớ
  - Không cho phép truy cập nào khác, khi xảy ra tác vụ đọc hoặc ghi
- Các tác vụ thực hiện chỉ với 1 lệnh
  - Ví dụ: hoán vị register ↔ memory
  - Hoặc 1 cặp atomic lệnh

Đồng bộ trong MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Ví dụ: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

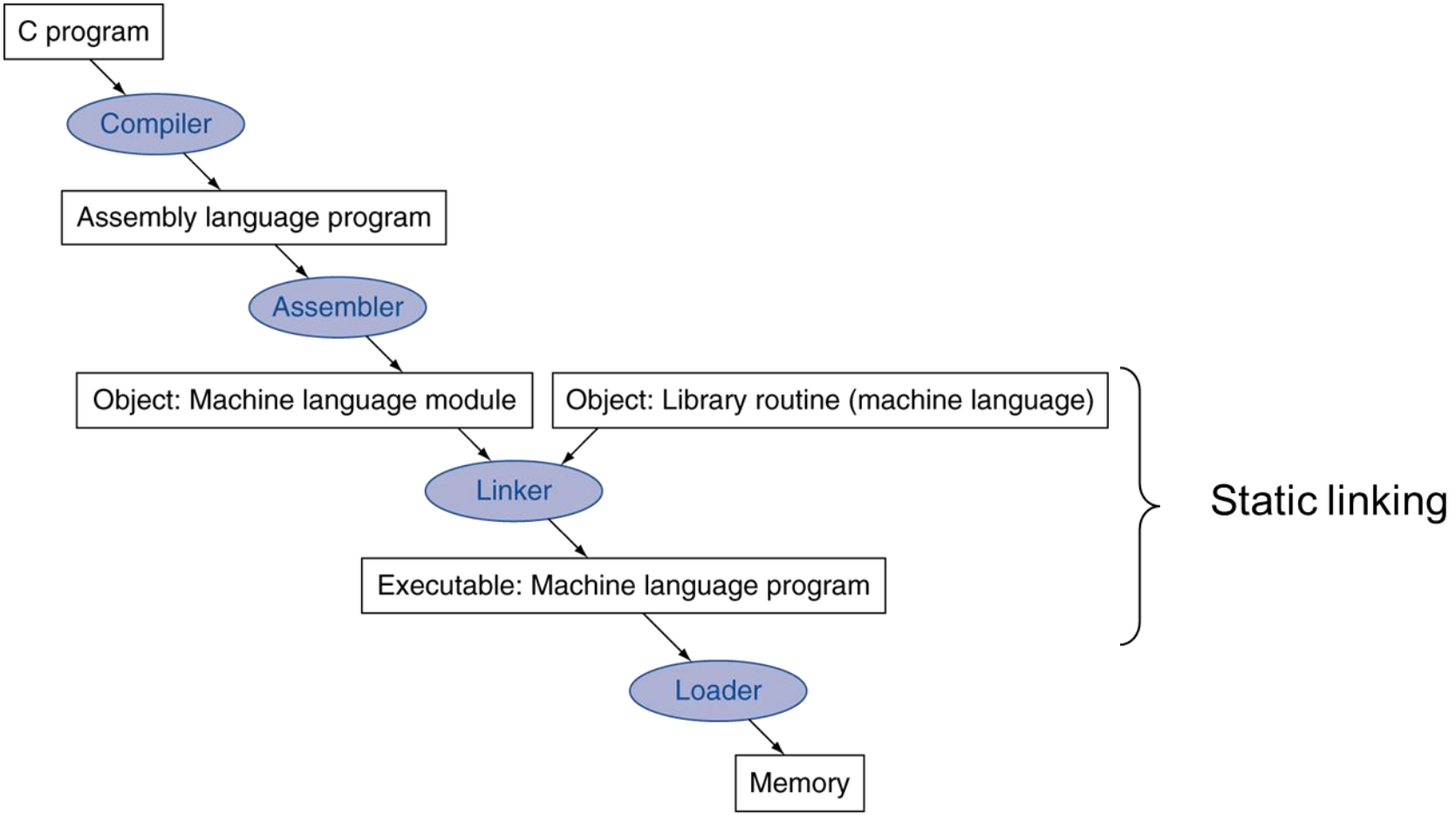
Lệnh giả trong hợp ngữ

- Phần lớn lệnh trong hợp ngữ tương đồng 1-1 với lệnh mã máy
- Lệnh giả (Pseudo): dễ nhớ, ví dụ

```
move $t0, $t1     → add $t0, $zero, $t1
blt $t0, $t1, L   → slt $at, $t0, $t1
                    bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Biên dịch và thực hiện



Tạo Object Module

- Assembler (hoặc compiler) biên dịch chương trình ra lệnh máy
- Thiết lập các thông tin để xây dựng 1 chương trình để có thể thực thi, bao gồm
  - Header: đặc tả nội dung của object module
  - Text segment: các lệnh đã được biên dịch
  - Static data segment: dữ liệu được cấp phát cho chương trình trong suốt quá trình tực thi
  - Relocation info: định vị tuyệt đối của chương trình được nạp vào bộ nhớ
  - Symbol table: global definitions and external refs
  - Debug info: liên quan đến gỡ rối chương trình

Liên kết các Object Modules

- Linker: Còn gọi là *link editor*, cho phép ghép các *object file* riêng lẻ lại với nhau thành một chương trình thống nhất có thể thực thi được gọi là *executable file*
- Quá trình ghép diễn ra theo 3 bước
  - Xếp mã chương trình và dữ liệu lại với nhau
  - Xác định địa chỉ cho các nhãn chương trình và dữ liệu So trùng các tham cứu nội và ngoại (*internal/external reference*)
- Một *executable file* có các thành phần gần giống với *object file* trừ các phần: *relocation information*, *symbol table* và *debugging information*
- Các *object file*, ngoài các chương trình do người dùng (*user*) viết, còn có các trình con viết sẵn trong thư viện (*library*) do *compiler* cung cấp, do người dùng tạo lập hay từ các nguồn chuyên biệt

Nạp một chương trình

- Nạp tập tin thực thi trên đĩa vào bộ nhớ
  1. Đọc header để xác định dung lượng các đoạn
  2. Tạo không gian địa chỉ ảo
  3. Khởi động dữ liệu trong bộ nhớ
  4. Set up arguments on stack
  5. Khởi động các thanh ghi (gồm $sp, $fp, $gp)
  6. Nhảy tới đầu chương trình
     - Sao các thông số vào $a0, … và gọi main
     - Khi kết thúc trở về từ main, do exit syscall

Liên kết động

- Chỉ liên kết/nạp khi thủ tục được gọi
  - Yêu cầu phần code của thủ tục được cấp phát bộ nhớ
  - Tránh việc phát sinh cấp phát sinh ra bởi kết nối với thư viện
  - Tự động cập nhật phiên bản mới của thư viện

Kết luận

- Các nguyên tắc thiết kế
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Các lớp phần mềm/cứng
  - Biên dịch, Hợp ngữ, Phần cứng
- MIPS: là mô hình đặc thù kiến trúc tập lệnh RISC

Kết luận (tt.)

- Đo đạc thực hiện tập lệnh của MIP với chương trình đánh giá

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%