Character Data

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- MIPS byte / halfword load/store
  - Sign (1) extend to 32 bits in rt

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

  - Zero (0) extend to 32 bits in rt

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

  - Store just rightmost byte/halfword

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

String Copy Example

- C code (naïve):
  - Null-terminated string

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

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

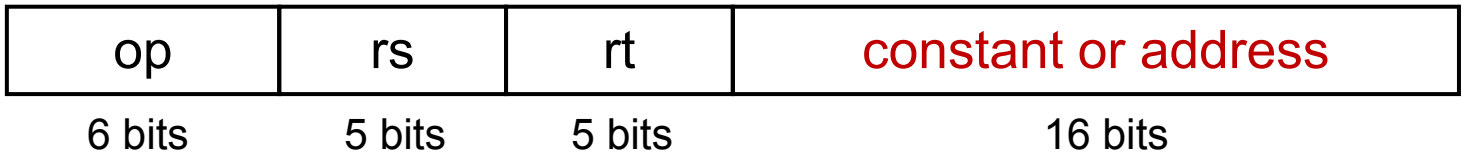
```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0 (i)
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # 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
```

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

MIPS Addressing

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient



- 32-bit constant or 32-bit address is needed some times.

- How to load a 32-bit constant into register $s0?

32-bit Constants

0000 0000 0011 1101 0000 1001 0000 0000

61_{10} 2304_{10}

`lui: load upper immediate`

```
lui rt, constant
```

- Copies 16-bit constant to left 16 bits of rt
- Clears right 16 bits of rt to 0

```
lui $s0, 61
```

0000 0000 0111 1101 0000 0000 0000 0000

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

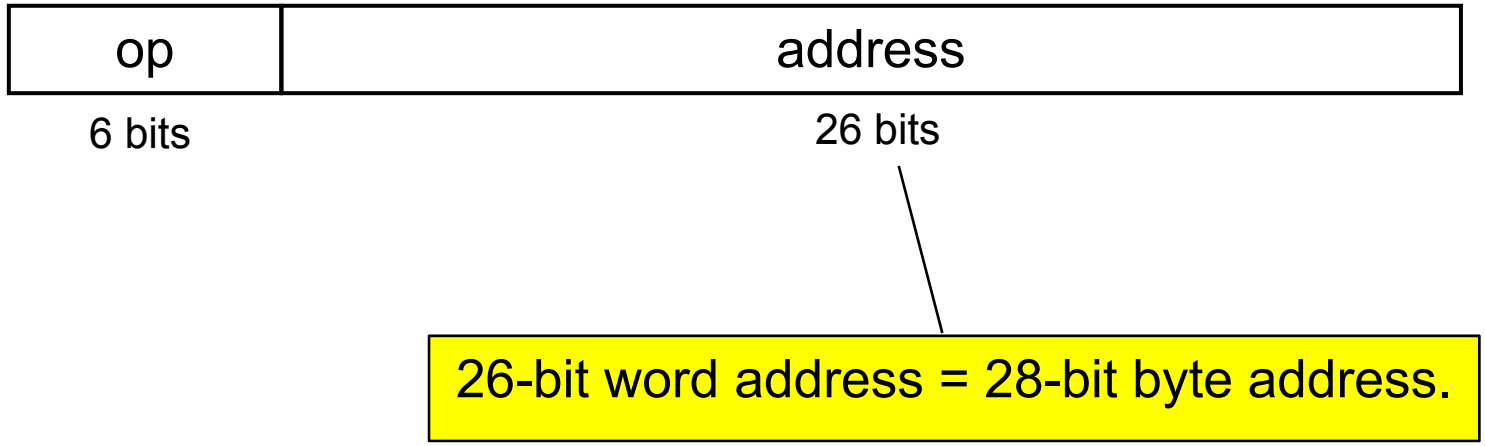
0000 0000 0111 1101 0000 1001 0000 0000

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

J-format

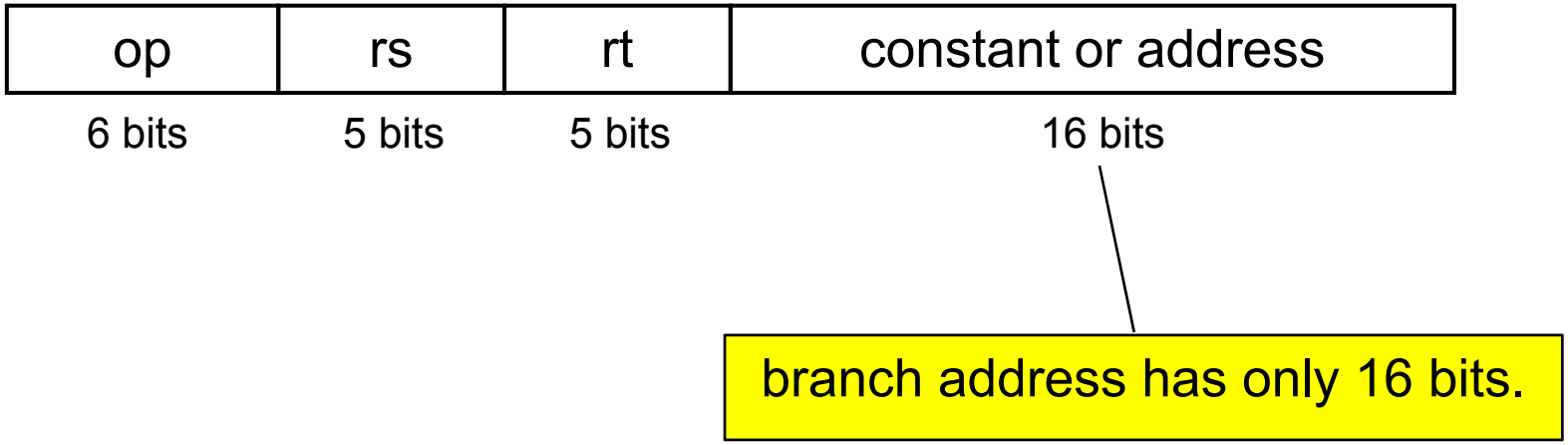
```
j 10000  #go to location 10000
```



Conditional Branch

- Branch instructions specify
  - Opcode, two registers, target address

```
bne $s0, $s1, Exit  #go to Exit if $s0≠$s1
```

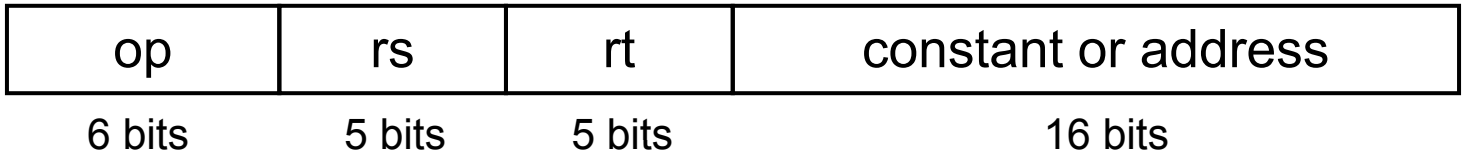


Problem: *No program can be bigger than* 2^{16} *!*

PC-relative Addressing

- PC (Program Counter) : register that contains the address of current instruction
  - Most branch targets are near branch

- PC-relative addressing
  - Target address = PC + offset address



  - MIPS uses PC-relative addressing for all conditional branches

PC-relative Addressing

- In MIPS implementation, PC often early points to the next instruction.

- Actually,
  - MIPS address = PC + 4 + offset
  - 16-bit offset → word address.

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
while (save[i] == k) i += 1;
```

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

80000 = 20000*4

80024 = 80012 + 4 + 2*4

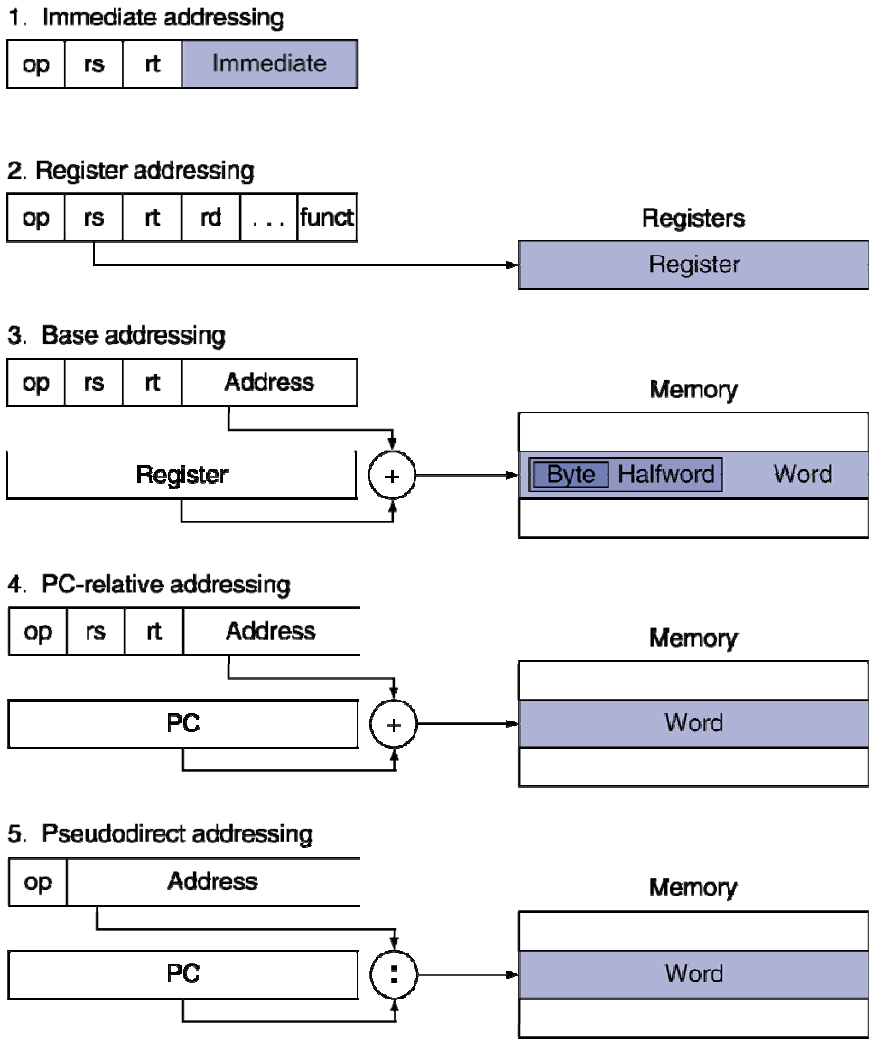
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

- Example

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

Addressing Mode Summary



Target address = $PC_{31...28}$: (address x 4)

Synchronization

Synchronization

- Two processors sharing an area of memory
  - Data race if P1 and P2 don’t synchronize, result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
- Could be a single instruction, Or an atomic pair of instructions

Synchronization in MIPS

- Load linked:

```
ll rt, offset(rs)
```

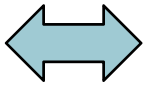
- Store conditional:

```
sc rt, offset(rs)
```

- Used in sequence: if contents in the location specified by `ll` are changed before `sc`, sc fails.
  - sc succeeds if content is not changed since the ll, returns 1 in rt
  - sc fails if content is changed, returns 0 in rt

Synchronization in MIPS

- Example: atomic swap (to test/set lock variable)

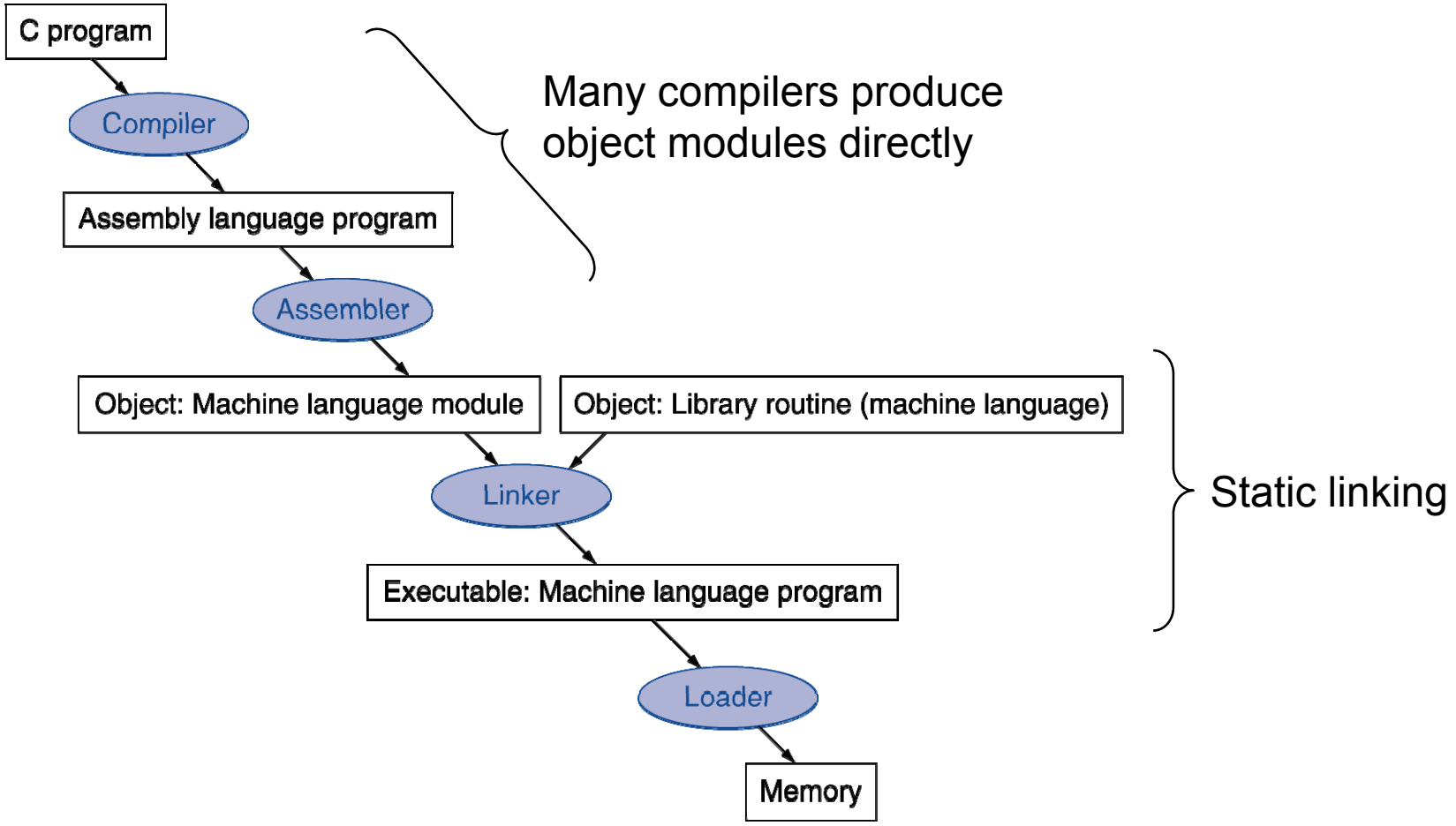



```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

If the value in memory is modified between `ll` and `sc` instructions, `sc` returns 0 in $t0, causing the code to try again.

Translation & Startup

Translation and Startup



Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: simplify programming

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

$at (register 1): assembler temporary

Object File

- Object file contains:
  - Header: size and position of pieces in object file
  - Text segment: machine language code
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external references
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image

1. Merges segments
2. Determine address of labels
3. Patch internal and external references

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create address space large enough for the text/data
  3. Copy instructions and data into memory
  4. Copy parameters to the main program on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine that
     - Copies parameters into argument registers
     - Calls main routine of the program
     - When main routine returns, do `exit` system call

Static Linking

- Static Linking: link libraries before the program is run.
  - Still use older version of libraries when new version is available
  - Load all routines that are called anywhere in the program even if they are not executed.

Dynamic Linking

- Dynamic Linking: not link/load library procedures until the program is run.
  - Automatically picks up new library versions
  - Problem: still link all routines of the library that might be called. (initial version of DLLs)

- Lazy Linkage: each routine is linked only after it is called.

Starting Java Applications