Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer

- Different computers have different instruction sets

  - But with many aspects in common

- Early computers had very simple instruction sets

  - Simplified implementation

- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Arithmetic Operations

- Add and subtract, three operands
    - Two sources and one destination

add a, b, c   # a gets b + c

- All arithmetic operations have this form

- *Design Principle 1:* Simplicity favours regularity
    - Regularity makes implementation simpler
    - Simplicity enables higher performance at lower cost

Arithmetic Example

■ C code:

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

■ Compiled 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
```

Register Operands

- Arithmetic instructions use register operands

- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"

- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

  - f, ..., j in $s0, ..., $s4

- Compiled MIPS code:

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

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example 1

- C code:

  g = h + A[8];

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:

  - Index 8 requires offset of 32

    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3

- Compiled MIPS code:

  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)      # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)      # store word
```

Registers vs. Memory

- Registers are faster to access than memory

- Operating on memory data requires loads and stores

  - More instructions to be executed

- Compiler must use registers for variables as much as possible

  - Only spill to memory for less frequently used variables

  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

  `addi  $s3,  $s3,  4`

- No subtract immediate instruction

  - Just use a negative constant

    `addi  $s2,  $s1,  -1`

- *Design Principle 3:* Make the common case fast

  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten

- Useful for common operations
  - E.g., move between registers

  ```
  add $t2, $s1, $zero
  ```

Unsigned Binary Integers

- Given an n-bit number

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

- Range: 0 to $+2^n - 1$

- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1{\times}2^3 + 0{\times}2^2 + 1{\times}2^1 + 1{\times}2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - −1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

$$x + \overline{x} = 1111...111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000\ 0000\ ...\ 0010_2$
  - $-2 = 1111\ 1111\ ...\ 1101_2 + 1$
 $= 1111\ 1111\ ...\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value

- In MIPS instruction set
  - `addi` : extend immediate value
  - `lb, lh`: extend loaded byte/halfword
  - `beq, bne`: extend the displacement

- Replicate the sign bit to the left

  - c.f. unsigned values: extend with 0s

- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - −2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
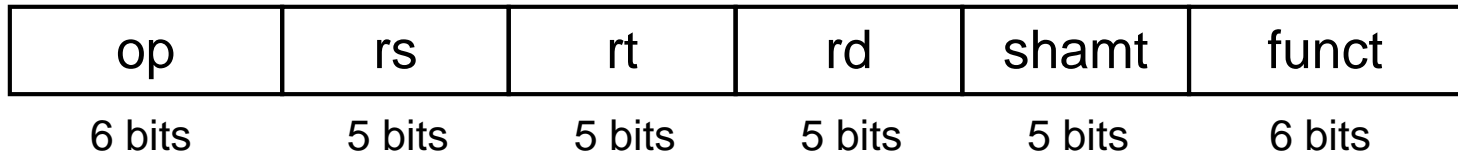
- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

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

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

Hexadecimal

- ## Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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

- ## Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions

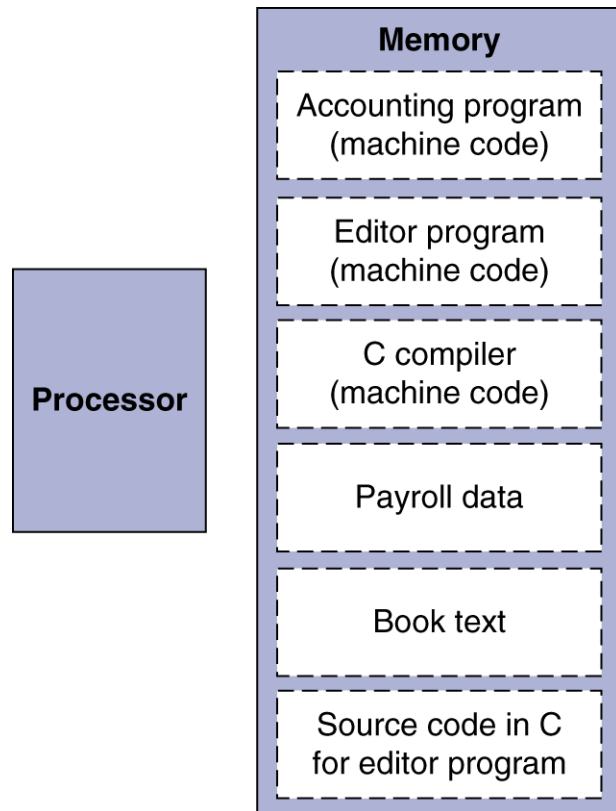| op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers

The BIG Picture

```
            Memory
        ┌─────────────────────┐
        │ Accounting program  │
        │   (machine code)    │
        ├─────────────────────┤
        │   Editor program    │
┌──────┐│   (machine code)    │
│      │├─────────────────────┤
│      ││     C compiler      │
│      ││   (machine code)    │
│Proce-│├─────────────────────┤
│ssor  ││    Payroll data     │
│      │├─────────────────────┤
│      ││     Book text       │
└──────┘├─────────────────────┤
        │ Source code in C    │
        │ for editor program  │
        └─────────────────────┘
```

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

■ **Instructions for bitwise manipulation**

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

■ **Useful for extracting and inserting groups of bits in a word**

Shift Operations

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

- ## shamt: how many positions to shift

- ## Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i

- ## Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

`and $t0, $t1, $t2`

$t2 | 0000 0000 0000 0000 0000 1101 1100 0000

$t1 | 0000 0000 0000 0000 0011 1100 0000 0000

$t0 | 0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

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

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - 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

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
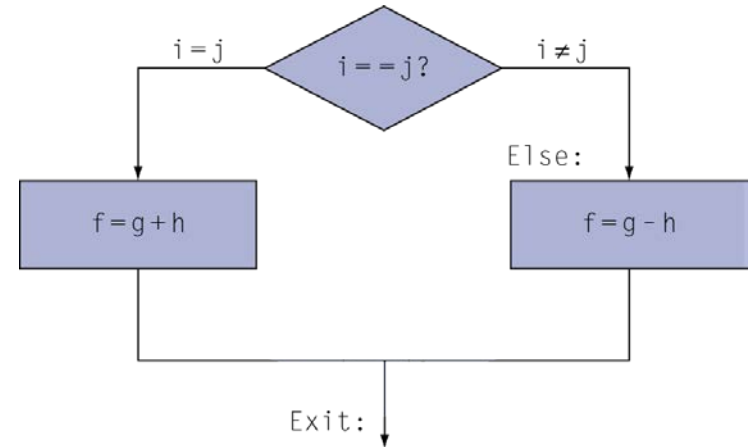 - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

  $$\text{if } (i == j) \; f \; = \; g+h;$$
  $$\text{else } f \; = \; g-h;$$

  - f, g, … in \$s0, \$s1, …

- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

- i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

Note: the code is logically correct but has a simple operational bug!

- add $s3, $zero, $zero # li $s3, 0 also works

```
Loop:   sll    $t1,  $s3,  2 # 0, 4, 8, 12, ... in $t1
        add    $t1,  $t1,  $s6
        lw     $t0,  0($t1)
        bne    $t0,  $s5,  Exit
        addi   $s3,  $s3,  1
        j      Loop
Exit:   …
```

Basic Blocks

- A basic block is a sequence of instructions with

  - No embedded branches (except at end)
  - No branch targets (except at beginning)
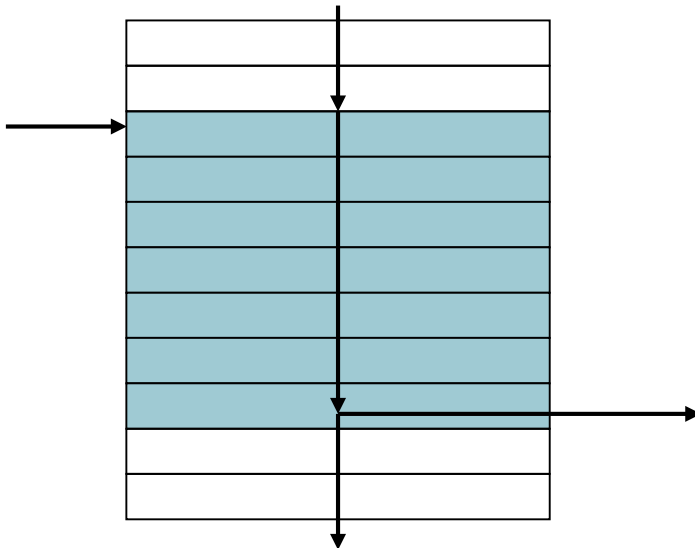
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 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;
- Use in combination with `beq`, `bne`

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

Branch Instruction Design

- Why not **blt**, **bge**, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- **beq** and **bne** are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
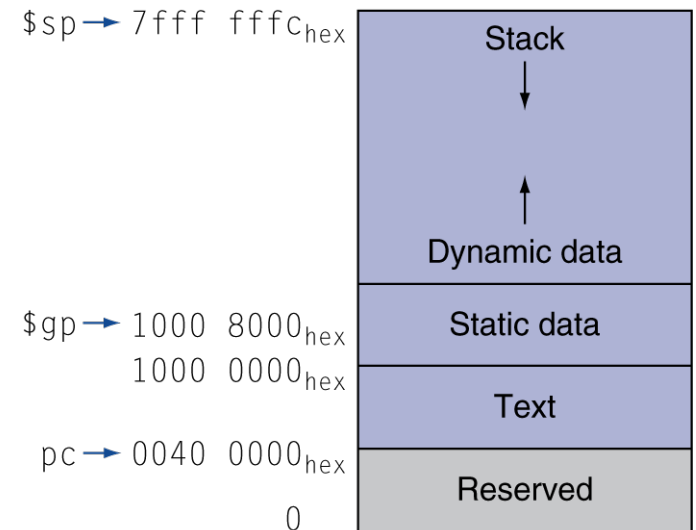 - `slt  $t0, $s0, $s1   # signed`
    - $-1 < +1 \Rightarrow$ $t0 = 1
  - `sltu $t0, $s0, $s1   # unsigned`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow$ $t0 = 0
  - **sltu useful in arrays bound checking (see book)**

Memory Layout Conventions

- Usually followed by assemblers
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage

$sp → 7fff fffc_hex
$gp → 1000 8000_hex
      1000 0000_hex
pc → 0040 0000_hex
     0

Stack
↓
↑
Dynamic data
Static data
Text
Reserved

Procedure Calling

- Steps required

  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Register Usage Conventions

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
    - Can be overwritten by callee
- $s0 – $s7: saved
    - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address

- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- MIPS code: jal leaf_example # call from main()

```
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
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
    - Its return address
    - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code: jal fact # top-level call in main()

```
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      # if not, branch to L1
    addi  $v0, $zero, 1       # if n < 1, result is 1
    addi  $sp, $sp, 8         # pop 2 items from stack
    jr    $ra                 # and return, result in $v0
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, result in $v0
```

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, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
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)
```

  - Store just rightmost byte/halfword

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: jal strcpy # call from main()

```
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, byte offset
    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
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

`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

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump ($\mathbf{j}$ and $\mathbf{jal}$) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits
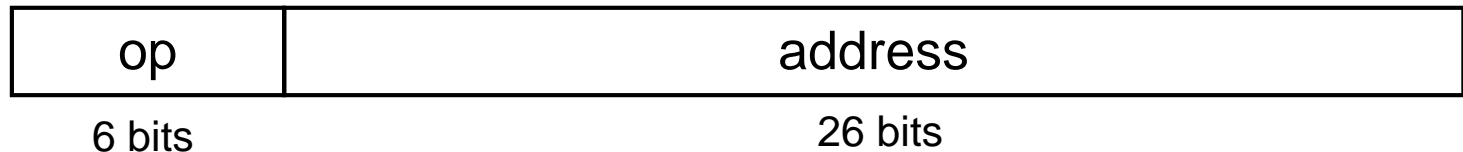
- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28} : (\text{address} \times 4)$

Target Addressing Example

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

Loop: sll $t1, $s3, 2	80000	0	0	19	9	4	0
add $t1, $t1, $s6	80004	0	9	22	9	0	32
lw $t0, 0($t1)	80008	35	9	8	0
bne $t0, $s5, Exit	80012	5	8	21	2
addi $s3, $s3, 1	80016	8	19	19	1
j Loop	80020	2	20000
Exit: ...	80024	

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:   …
```
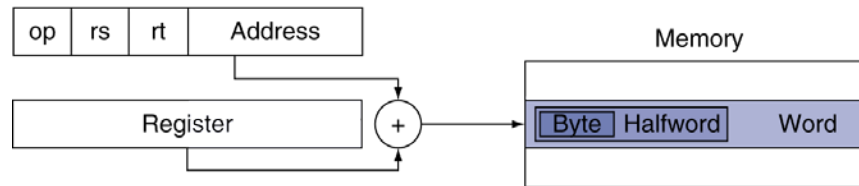
Addressing Mode Summary

1. Immediate addressing

| op | rs | rt | Immediate |

2. Register addressing

| op | rs | rt | rd | ... | funct |

Registers
Register

3. Base addressing

| op | rs | rt | Address |

Register

Memory
Byte | Halfword | Word

4. PC-relative addressing

| op | rs | rt | Address |

PC

Memory
Word
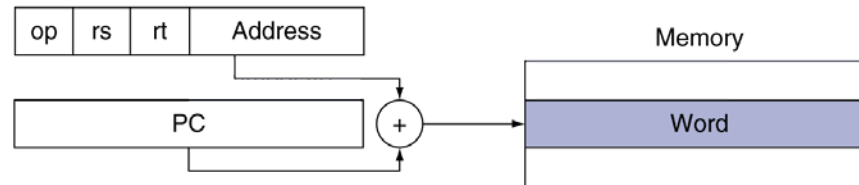
5. Pseudodirect addressing

| op | Address |

PC

Memory
Word
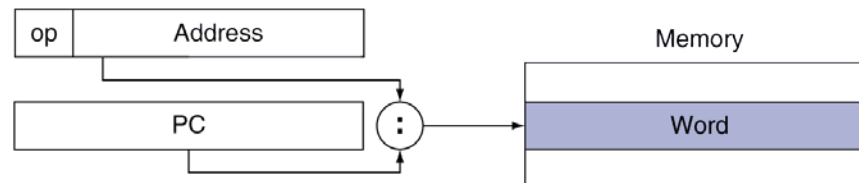
C Bubble Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k]  = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap:   sll $t1, $a1, 2    # $t1 = k * 4
        add $t1, $a0, $t1   # $t1 = v+(k*4)
                            #   (address of v[k])
        lw $t0, 0($t1)      # $t0 (temp) = v[k]
        lw $t2, 4($t1)      # $t2 = v[k+1]
        sw $t2, 0($t1)      # v[k] = $t2 (v[k+1])
        sw $t0, 4($t1)      # v[k+1] = $t0 (temp)
        jr $ra              # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)
  ```
  void sort (int v[], int n)
  {
    int i, j;
    for (i = 0; i < n; i += 1) {
      for (j = i - 1;
              j >= 0 && v[j] > v[j + 1];
              j -= 1) {
        swap(v,j);
      }
    }
  }
  ```
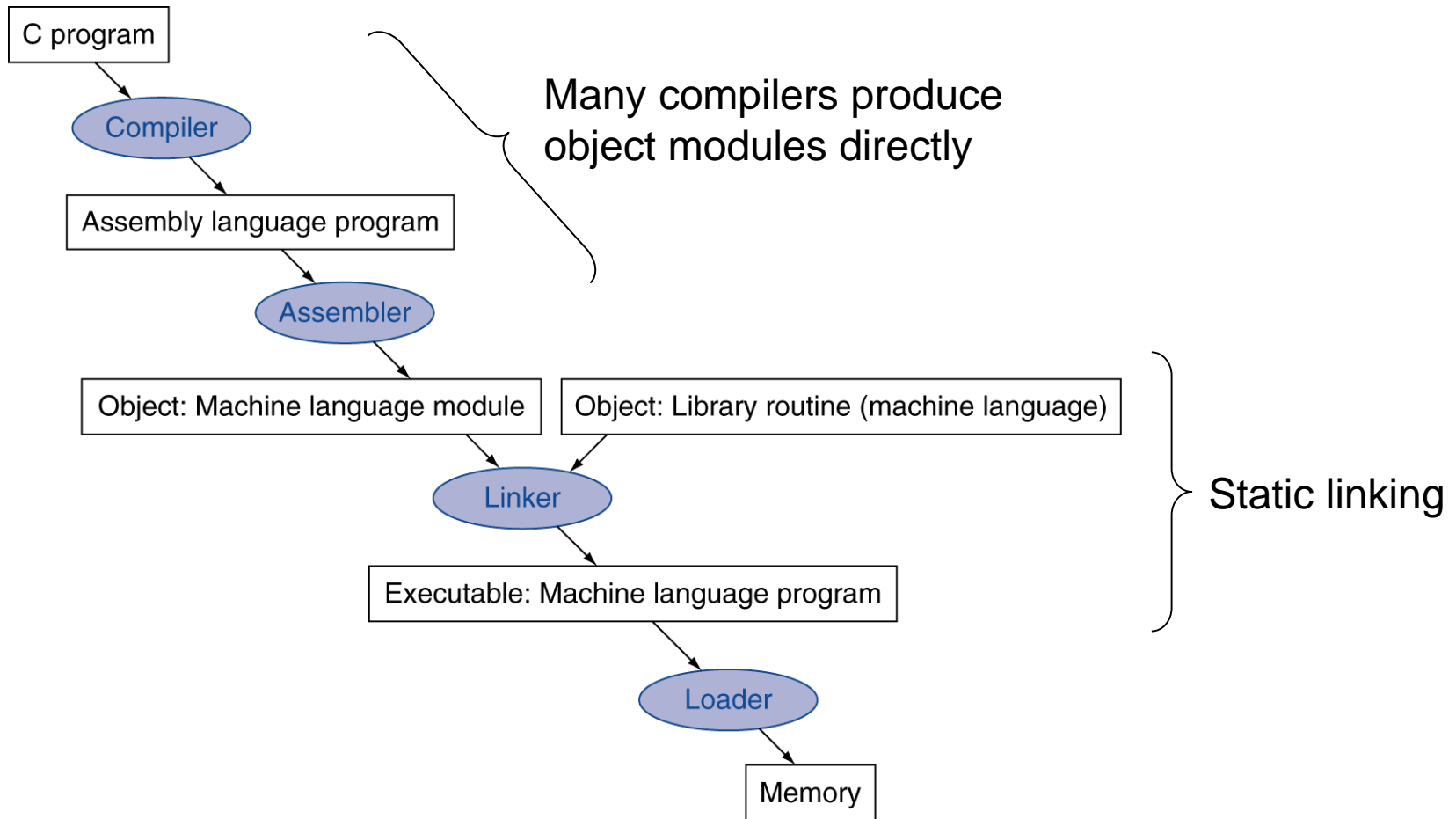 - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3    # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, −1      # j = i − 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2       # $t1 = j * 4
        add  $t2, $s2, $t1     # $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # $t3 = v[j]
        lw   $t4, 4($t2)       # $t4 = v[j + 1]
        slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        move $a0, $s2          # 1st param of swap is v (old $a0)
        move $a1, $s1          # 2nd param of swap is j
        jal  swap              # call swap procedure
        addi $s1, $s1, −1      # j −= 1
        j    for2tst           # jump to test of inner loop
exit2:  addi $s0, $s0, 1       # i += 1
        j    for1tst           # jump to test of outer loop
```

Annotation
Move params
Outer loop
Inner loop
Pass params & call
Inner loop
Outer loop

The Full Procedure

```
sort:       addi  $sp, $sp,  −20        # make room on stack for 5 registers
            sw $ra,  16($sp)            # save $ra on stack
            sw $s3, 12($sp)             # save $s3 on stack
            sw $s2,  8($sp)             # save $s2 on stack
            sw $s1,  4($sp)             # save $s1 on stack
            sw $s0,  0($sp)             # save $s0 on stack
            ...                         # procedure body
            ...
            exit1: lw $s0,  0($sp)      # restore $s0 from stack
            lw $s1,  4($sp)             # restore $s1 from stack
            lw $s2,  8($sp)             # restore $s2 from stack
            lw $s3, 12($sp)             # restore $s3 from stack
            lw $ra, 16($sp)             # restore $ra from stack
            addi  $sp, $sp,  20         # restore stack pointer
            jr $ra                      # return to calling routine
```

Translation and Startup

C program

Compiler

Assembly language program

Assembler

Many compilers produce object modules directly

Object: Machine language module   Object: Library routine (machine language)

Linker

Executable: Machine language program

Static linking

Loader

Memory

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one

- Pseudoinstructions: figments of the assembler's imagination that helps assembly language programmers

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

  - $at (register 1): assembler temporary

- Other such pseudoinstructions are listed in the manual that has been uploaded

Loading a Program

- Load from image file on disk into memory
    1. Read header to determine segment sizes
    2. Create virtual address space
    3. Copy text and initialized data into memory
        - Or set page table entries so they can be faulted in
    4. Set up arguments on stack
    5. Initialize registers (including $sp, $fp, $gp)
    6. Jump to startup routine
        - Copies arguments to $a0, … and calls main
        - When main returns, do exit syscall

Fallacies

- Powerful instruction $\Rightarrow$ higher performance
    - Fewer instructions required
    - But complex instructions are hard to implement
        - May slow down all instructions, including simple ones
    - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
    - But modern compilers are better at dealing with modern processors
    - More lines of code $\Rightarrow$ more errors and less productivity