CSEE 3827: Fundamentals of Computer Systems, Spring 2011

7. MIPS Instruction Set Architecture

Prof. Martha Kim (martha@cs.columbia.edu)
Web: http://www.cs.columbia.edu/~martha/courses/3827/sp11/

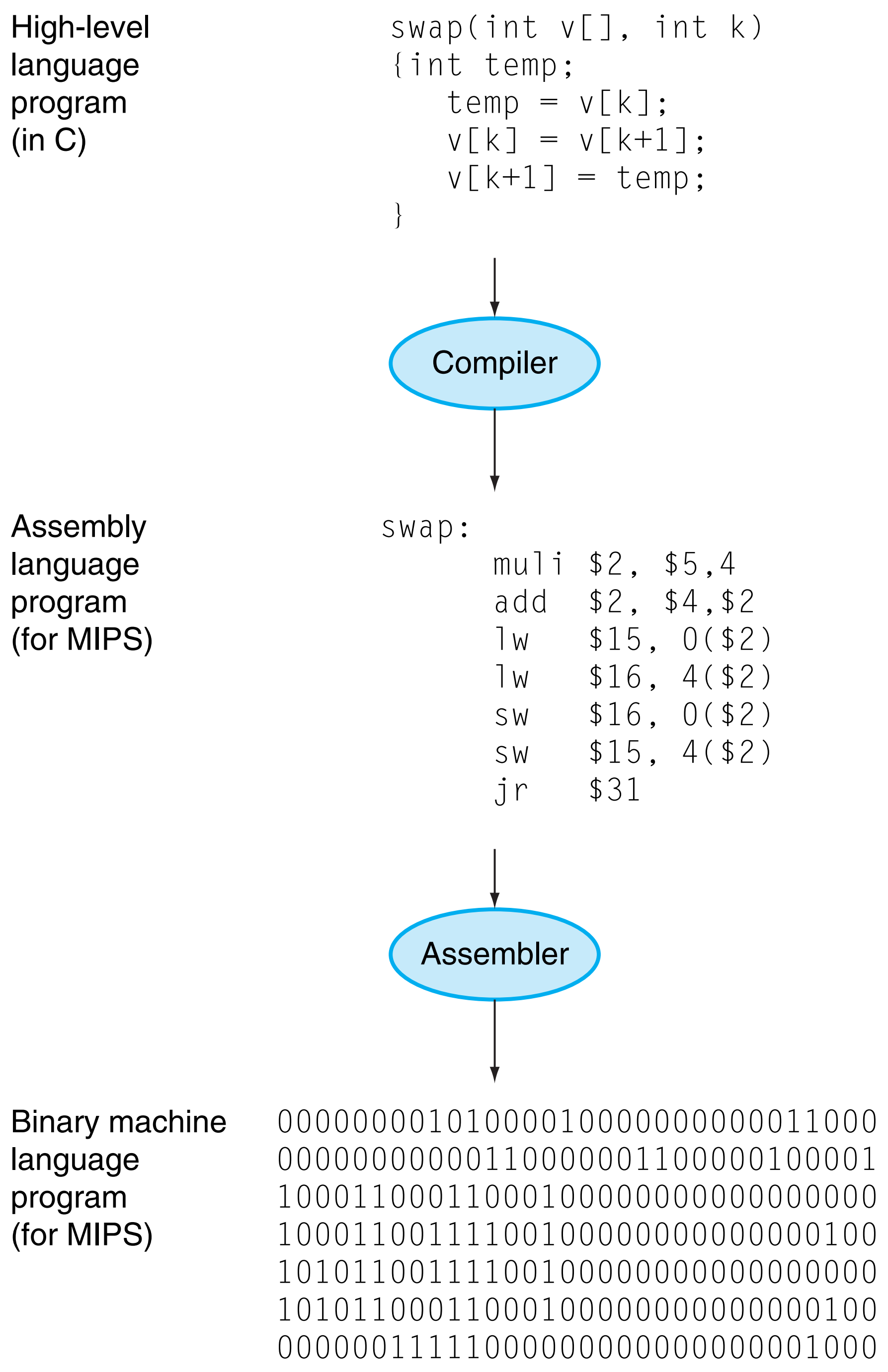
Outline (H&H 6.1-6.7.1)

- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Compiling, Assembling, and Loading
- Odds and Ends

Assembly Language

- To command a computer, you must understand its language.
  - **Instructions**: words in a computer's language
  - **Instruction set**: the vocabulary of a computer's language
- Instructions indicate the operation to perform and the operands to use.
  - **Assembly language**: human-readable format of instructions
  - **Machine language**: computer-readable format (1's and 0's)

Machine v. Assembly Code


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

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

(source code)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

(assembly code)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

(machine code)

What is an ISA?

- An Instruction Set Architecture, or ISA, is an **interface** between the hardware and the software.
- An ISA consists of:
  - a set of operations (instructions)
  - data units (sizes, addressing modes, etc.)
  - processor state (registers)
  - input and output control (memory operations)
  - execution model (program counter)

Why have an ISA?

- An ISA provides binary compatibility across machines that share the ISA
- Any machine that implements the ISA X can execute a program encoded using ISA X.
- You typically see families of machines, all with the same ISA, but with different power, performance and cost characteristics.
  - e.g., the MIPS family: MIPS 2000, 3000, 4400, 10000

MIPS Architecture

- MIPS = Microprocessor without Interlocked Pipeline Stages
- MIPS architecture developed at Stanford in 1984, spun out into MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors had been sold
- Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco
- Once you’ve learned one architecture, it’s easy to learn others.

MIPS is a RISC Architecture

- RISC = Reduced Instruction Set Computer
- RISC is an alternative to CISC (Complex Instruction Set Computer) where operations are significantly more complex.
- Underlying design principles, as articulated by Hennessy and Patterson:
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises
- MIPS (and other RISC architectures) are “load-store” architectures, meaning all operations performed only on operands in registers. (The only instructions that access memory are loads and stores)

What is an ISA?

- An Instruction Set Architecture, or ISA, is an **interface** between the hardware and the software.

- An ISA consists of: (for MIPS)
  - a set of operations (instructions) $\leftarrow$ arithmetic, logical, conditional, branch, etc.
  - data units (sized, addressing modes, etc.) $\leftarrow$ 32-bit data word
  - processor state (registers) $\leftarrow$ 32, 32-bit registers
  - input and output control (memory operations) $\leftarrow$ load and store
  - execution model (program counter) $\leftarrow$ 32-bit program counter

An example Program in MIPS: Factorial(n)

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

C 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
```

MIPS code

An Program in MIPS

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

C 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
```

MIPS code

Instructions

(description of operation to be performed during a cycle)

An Program in MIPS

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

C code

Registers

```
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
```

MIPS code

An Program in MIPS

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

C code

Constants

```
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
```

MIPS code

An Program in MIPS

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

C code

Access to main memory

```
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
```

MIPS code

An Program in MIPS

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

C code

Control Labels and “Jump” Instructions

```
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
```

MIPS code

Instruction Classes

- **Memory Access**: Move data to/from memory from/to registers
- **Arithmetic/Logic**: Perform (via functional unit) computation on data in registers (store result in a register)
- **Jump/Jump Subroutine**: direct control to a different part of the program (not next word in memory)
- **Conditional branch**: test values in registers. If test returns true, move control to different part of program. Otherwise, proceed to next word

NB: *These are functional classes. Later we will classify the instructions according to their formats (R-type, I-type, etc.)*

Arithmetic Instructions

- Addition and subtraction
- Three operands: two source, one destination
- `add a, b, c    # a gets b + c`
- All arithmetic operations (and many others) have this form

Design principle:

Regularity makes implementation simpler

Simplicity enables higher performance at lower cost

Arithmetic Example 1

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

C code

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

MIPS assembly

Arithmetic Example 1 w. Registers

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

MIPS assembly w.o registers

store: f in $s0, g in $s1, h in $s2, i in $s3, and j in $s4

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

MIPS assembly w. registers

Memory Operands

- Main memory used for composite data (e.g., arrays, structures, dynamic data)
- To apply arithmetic operations
  - Load values from memory into registers (load instruction = mem read)
  - Store result from registers to memory (store instruction = mem write)
- Memory is byte-addressed (each address identifies an 8-bit byte)
- Words (32-bits) are aligned in memory (meaning each address must be a multiple of 4)
- MIPS is big-endian (i.e., most significant byte stored at least address of the word)

Memory Operands

- Main memory used for composite data (e.g., arrays, structures, dynamic data)
- To apply arithmetic operations
  - Load values from memory into registers (load instruction = mem read)
  - Store result from registers to memory (store instruction = mem write)
- Memory is byte-addressed (each address identifies an 8-bit byte)
- Words (32-bits) are aligned in memory (meaning each address must be a multiple of 4)
- MIPS is big-endian (i.e., most significant byte stored at least address of the word)

Memory Operand Example 1

```
g = h + A[8]
```

C code

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

offset *base register*

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

MIPS assembly

Memory Operand Example 2

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

C code

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)
index = 12 requires offset of 48 (12 items x 4 bytes per word)

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

MIPS assembly

Registers v. Memory

- Registers are faster to access than memory
- Operating on data in memory requires loads and stores
    - (More instructions to be executed)
- Compiler should use registers for variables as much as possible
    - Only spill to memory for less frequently used variables
    - Register optimization is important for performance

Immediate Operands

- Constant data encoded in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction, just use the negative constant

```
addi $s2, $s1, -1
```

Design principle: *make the common case fast*

Small constants are common

Immediate operands avoid a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
- $zero cannot be overwritten
- Useful for many operations, for example, a move between two registers

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

Register Numbers

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Note: *Register 1 ($at) is reserved for the assembler, and 26-27 ($k0-$k1) are reserved for the OS.*

MIPS instructions to date

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

NB: *reg = register number between 0 and 31;*
address = 16-bit address

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: register destination 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

MIPS I-format Instructions

op	rs	rt	constant
6 bits	*5 bits*	*5 bits*	*16 bits*

- Includes immediate arithmetic and load/store operations
  - op: operation code (opcode)
  - rs: first source register number
  - rt: destination register number
  - constant: offset added to base address in rs, or immediate operand

MIPS Logical Operations

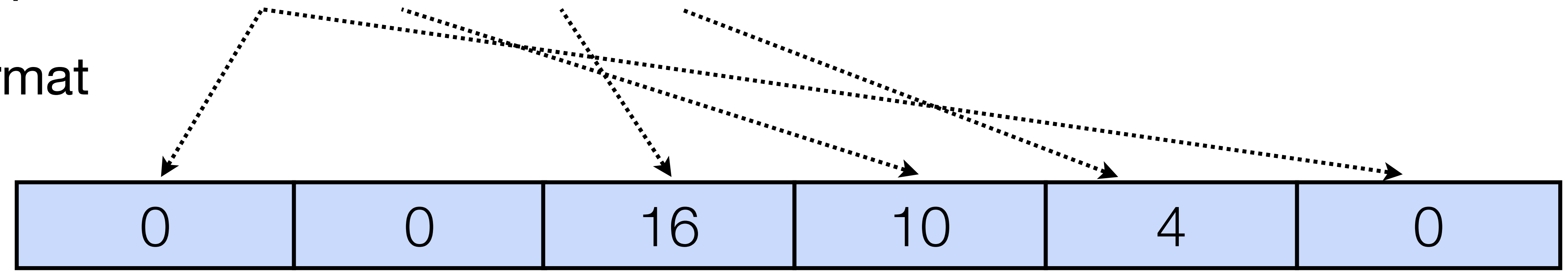
- Instructions for bitwise manipulation

Logical operations	C operators	Java operators	MIPS instructions
Shift left	`<<`	`<<`	`sll`
Shift right	`>>`	`>>>`	`srl`
Bit-by-bit AND	`&`	`&`	`and, andi`
Bit-by-bit OR	`|`	`|`	`or, ori`
Bit-by-bit NOT	`~`	`~`	`nor`

- Useful for inserting and extracting groups of bits in a word

Shift Operations

- Shift left logical (op = `sll`)
  - Shift left and fill with 0s
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical (op = `srl`)
  - Shift right and fill with 0s
  - `srl` by *i* bits divides by 2^i *(for unsigned values only)*
- shamt indicates how many positions to shift
- example: `sll $t2, $s0, 4  # $t2 = $s0 << 4 bits`
- R-format



Full Complement of Shift Instructions

- sll: shift left logical (`sll $t0, $t1, 5  # $t0 <= $t1 << 5`)
- srl: shift right logical (`srl $t0, $t1, 5  # $t0 <= $t1 >> 5`)
- sra: shift right arithmetic (`sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`)

- Variable shift instructions:
  - sllv: shift left logical variable

    (`sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`)
  - srlv: shift right logical variable

    (`srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`)
  - srav: shift right arithmetic variable

    (`srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`)

Generating Constants

- 16-bit constants using `addi`:

```
// int is a 32-bit signed word
int a = 0x4f3c
```

C code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

MIPS assembly

- 32-bit constants using load upper immediate (`lui`*) and `ori` *`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)

```
int a = 0xFEDC8765;
```

C code

```
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

MIPS assembly

*`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.

AND Operations

- example: `and $t0, $t1, $t2  # $t0 = $t1 & $t2`

- Useful for masking bits in a word (selecting some bits, clearing others to 0)

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

OR Operations

- example: `or $t0, $t1, $t2  # $t0 = $t1 | $t2`

- Useful to include bits in a word (set some bits to 1, leaving others unchanged)

```
$t1:  0000 0000 0000 0000 0000 1101 1100 0000
$t2:  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

- MIPS has 3 operand NOR instruction, used to compute NOT
- example: `nor $t0, $t1, $zero  # $t0 = ~$t1`

```
$t1: 0000 0000 0000 0000 0000 1101 1100 0000
$t0: 1111 1111 1111 1111 1111 0010 0011 1111
```

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- Instruction labeled with colon e.g. `L1: add $t0, $t1, $t2`

```
• beq rs, rt, L1 # if (rs == rt) branch to instr labeled L1

• bne rs, rt, L1 # if (rs != rt) branch to instr labeled L1

• j L1           # unconditional jump to instr labeled L1
```

Compiling an If Statement

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

C code

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

MIPS assembly

- Where, f is in $s0, g is in $s1, and h is in $s2
- The assembler calculates the addresses corresponding to the labels

Compiling a Loop Statement

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

C code

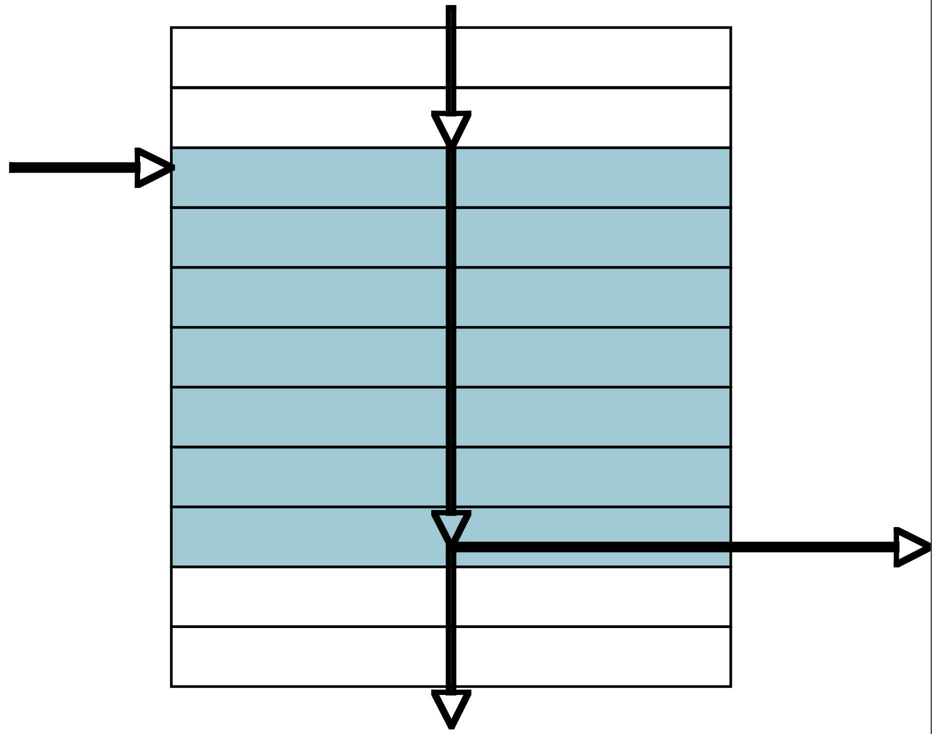
```
Loop:
  sll $t1, $s3, 2
  add $t1, $t1, $s5
  lw $t0, 0($t1)
  bne $t0, $s4, Exit
  addi $s3, $s3, 1
  j Loop
Exit:
```

MIPS assembly

- Where, i is in $s3, k is in $s4, address of save in $s5

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches except at the end
  - No branch targets except at the beginning



- A compiler identifies basic blocks for optimization
- Advanced processors can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
- `slt rd, rs, rt              # (rs < rt) ? rd=1 : rd=0`
- `slti rd, rs, constant       # (rs < constant) ? rd=1 : rd=0`
- Use in combination with `beq` or `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 <, >= etc. is slower than for = and !=
    - Combining with a branch involves more work per instruction, requiring a slower clock
    - All instructions penalized because of this
- As beq and bne are the common case, this is a good compromise

Signed v. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example:

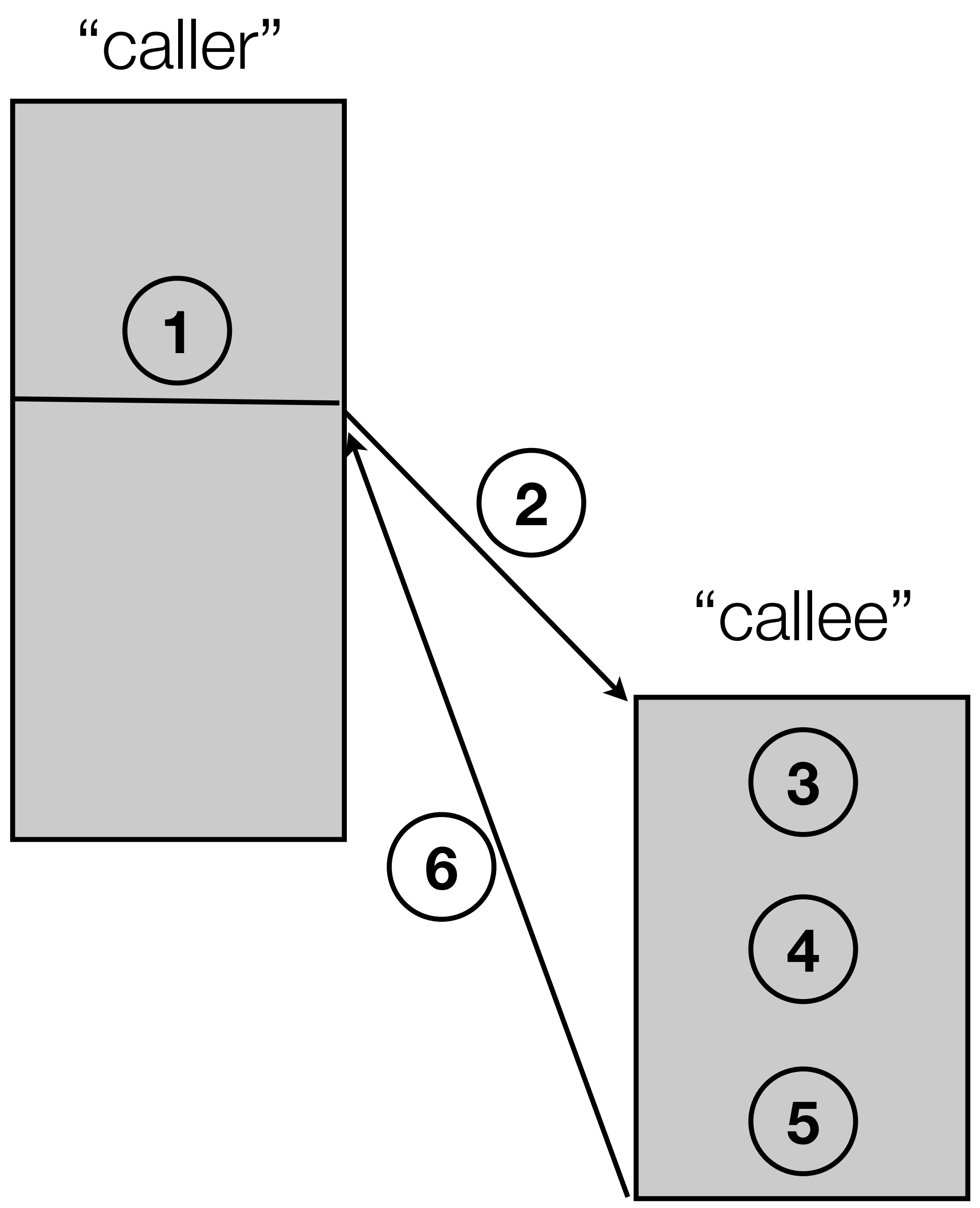
```
$s0: 1111 1111 1111 1111 1111 1111 1111 1111

$s1: 0000 0000 0000 0000 0000 0000 0000 0001
```

```
slt $t0, $s0, $s1  # signed: -1 < 1 thus $t0=1
sltu $t0, $s0, $s1 # unsigned: 4,294,967,295 > 1 thus $t0=0
```

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

- $a0-$a3: arguments
- $v0, $v1: result values
- $t0-$t9: temporaries, can be overwritten by callee
- $s0-$s7: contents saved

*** must be restored by callee

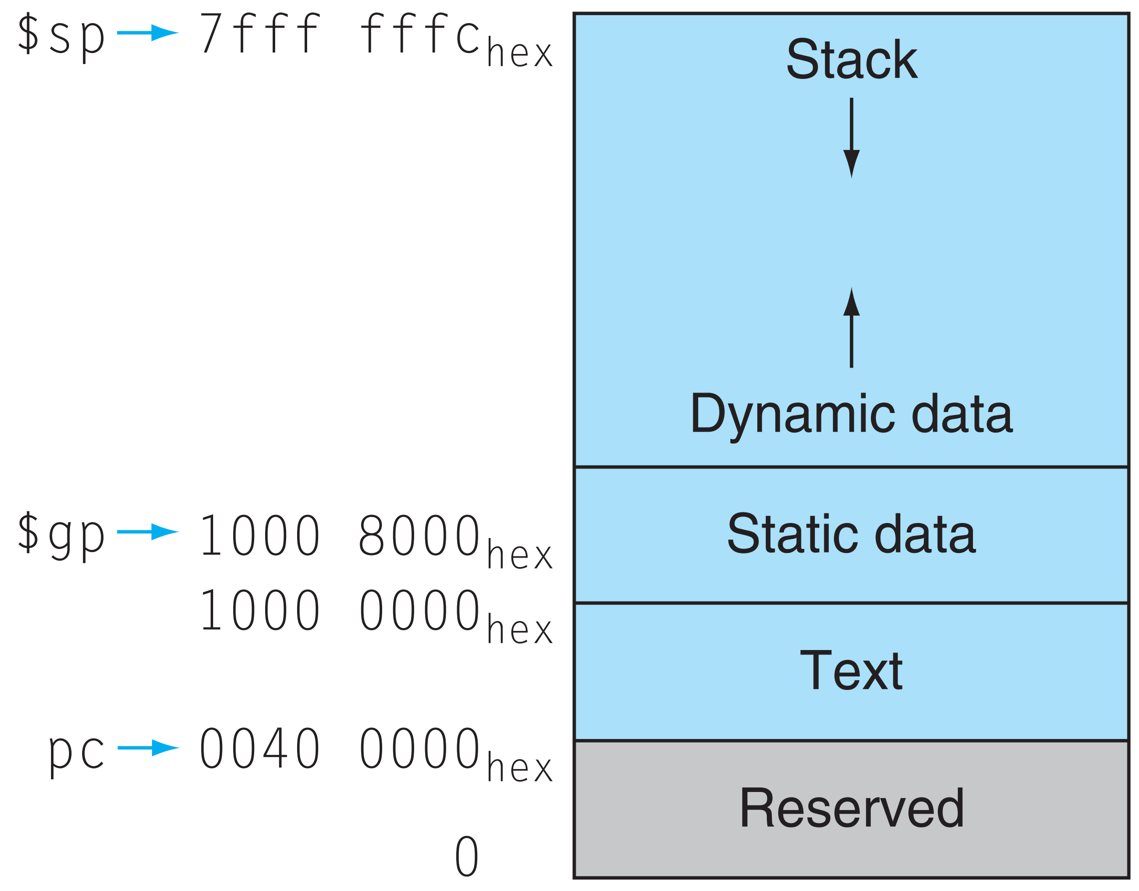
- $gp: global pointer for static data
- $sp: stack pointer
- $fp: frame pointer
- $ra: return address

Note: *There is nothing special about these registers' design, only their implied use!!!*

e.g., could store return value in $sp if calling and callee program both agreed to do this - just beware of messing up the stack for all other programs if not properly restored

Memory Layout

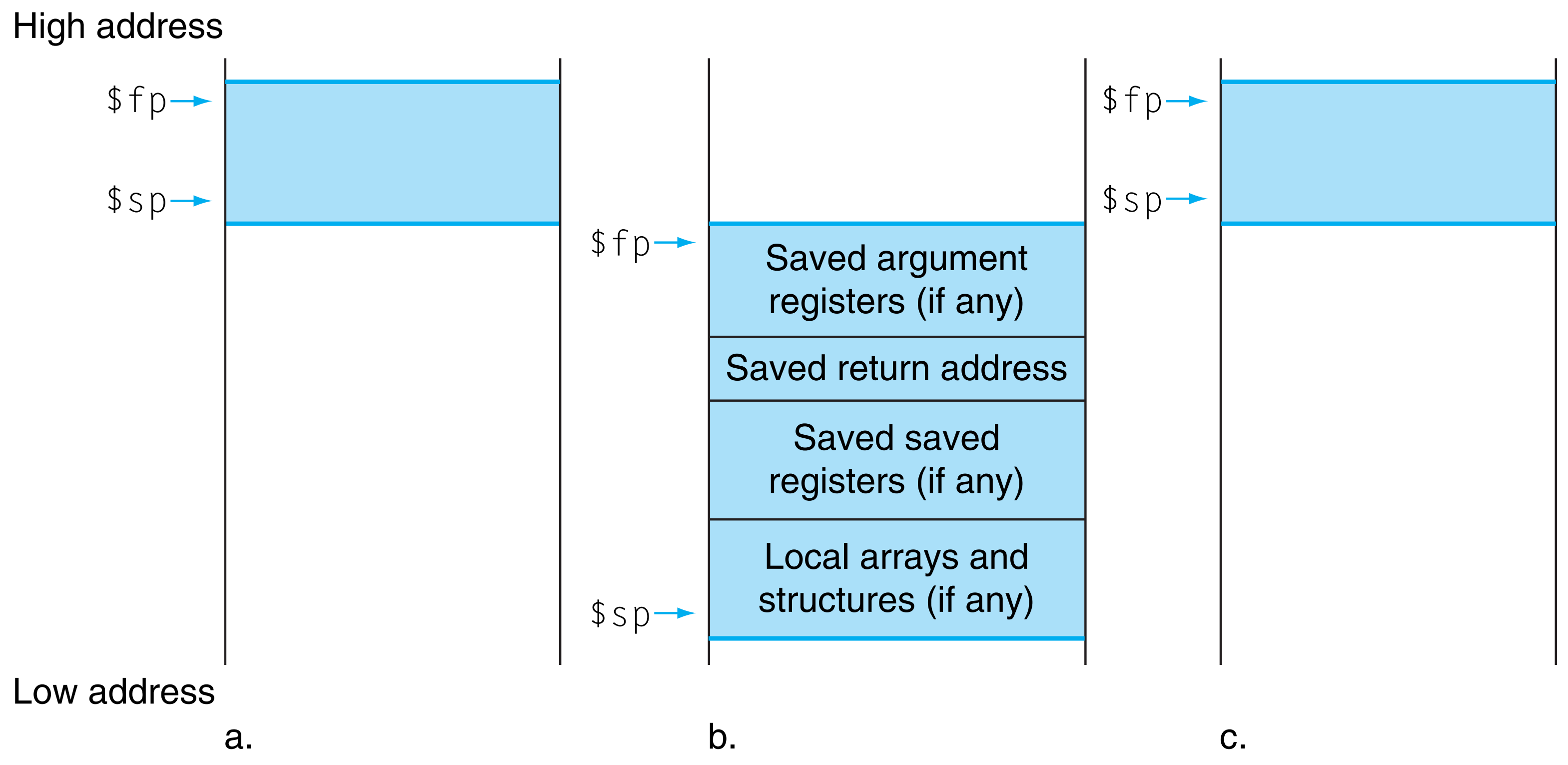
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to an address allowing +/- offsets in this segment
- Dynamic data: heap
  - e.g., malloc in C, new in Java
- Stack: automatic storage



Local Data on the Stack

- Local data allocated by the callee

- Procedure frame (activation record) used by compiler to manage stack storage


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- Cross-call register preservation

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

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

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

C code

- Arguments g, h, i, j in \$a0 - \$a3
- f will go in \$s0 (so will have to save existing contents of \$s0 to stack)
- result in \$v0

Leaf Procedure Example 2

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

C code

```
leaf_example:
    addi $sp, $sp, -4       save $s0 on stack
    sw $s0, 0($sp)
    add $t0, $a0, $a1
    add $t1, $a2, $a2       procedure body
    sub $s0, $t0, $t1
    add $v0, $s0, $zero     result
    lw $s0, 0($sp)          restore $s0
    addi $sp, $sp, 4
    jr $ra                  return
```

MIPS assembly

Non-Leaf Procedures

- A non-leaf procedure is a procedure that calls another procedure
- For a nested call, the caller needs to save to the stack
  - Its return address
  - Any arguments and temporaries needed after the call
- After the call, the caller must restore these values from the stack

Non-Leaf Procedure Example

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

C code

Non-Leaf Procedure Example 2

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

C 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
```

MIPS assembly

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 are variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS has byte/halfword load/store
    - `lb rt, offset(rs)  # sign extend byte to 32 bits in rt`
    - `lh rt, offset(rs)  # sign extend halfword to 32 bits in rt`
    - `lbu rt, offset(rs) # zero extend byte to 32 bits in rt`
    - `lhu rt, offset(rs) # zero extend halfword to 32 bits in rt`
    - `sb rt, offset(rs)  # store rightmost byte`
    - `sh rt, offset(rs)  # store rightmost halfword`

String Copy Example

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

C code (naive)

- Null-terminated string
- Addresses of x and y in $a0 and $a1 respectively
- i in $s0

String Copy Example 2

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

C code (naive)

```
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
```

MIPS assembly

32-bit constants

- Most constants are small, 16 bits usually sufficient

- For occasional, 32-bit constant:

```
lui rt, constant
```

  - copies 16-bit constant to the left (upper) bits of rt

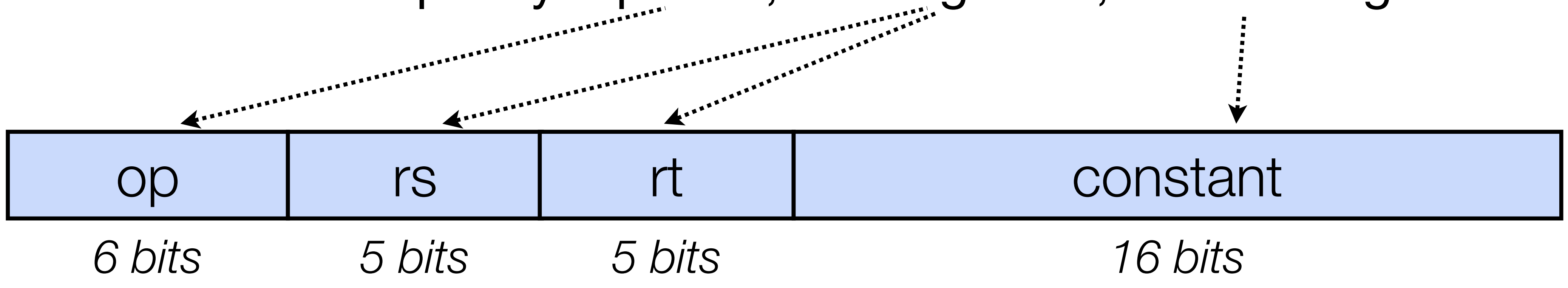
  - clears right (lower) 16 bits of rt to 0

- example usage:

```
    lui $s0, 61  $s0: 0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304  $s0: 0000 0000 0111 1101 0000 1001 0000 0000
```

Branch Addressing

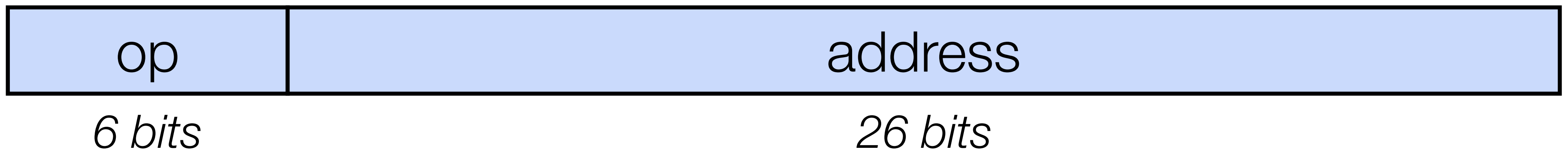
- Branch instructions specify: opcode, two registers, branch target



- Most branch targets are near branch (either forwards or backwards)
- PC-relative addressing
    - target address = PC + (offset * 4)
    - PC already incremented by four when the target address is calculated

Jump Addressing

- Jump (j and jal) targets could be anywhere in a text segment, so, encode the full address in the instruction



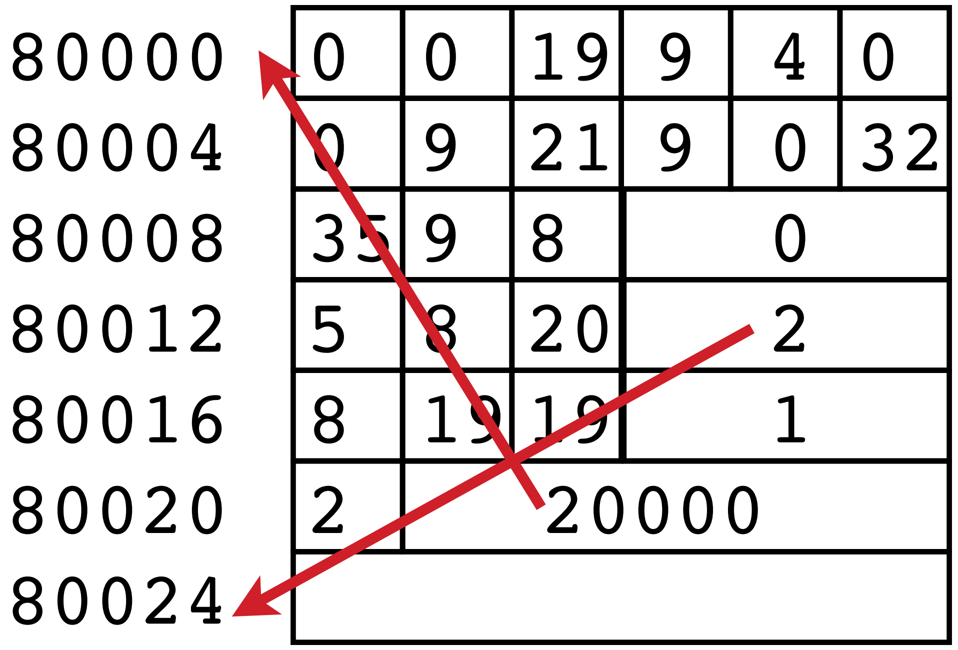
- target address = PC[31:28] : (address * 4)

Target Addressing Example

- Loop code from earlier example

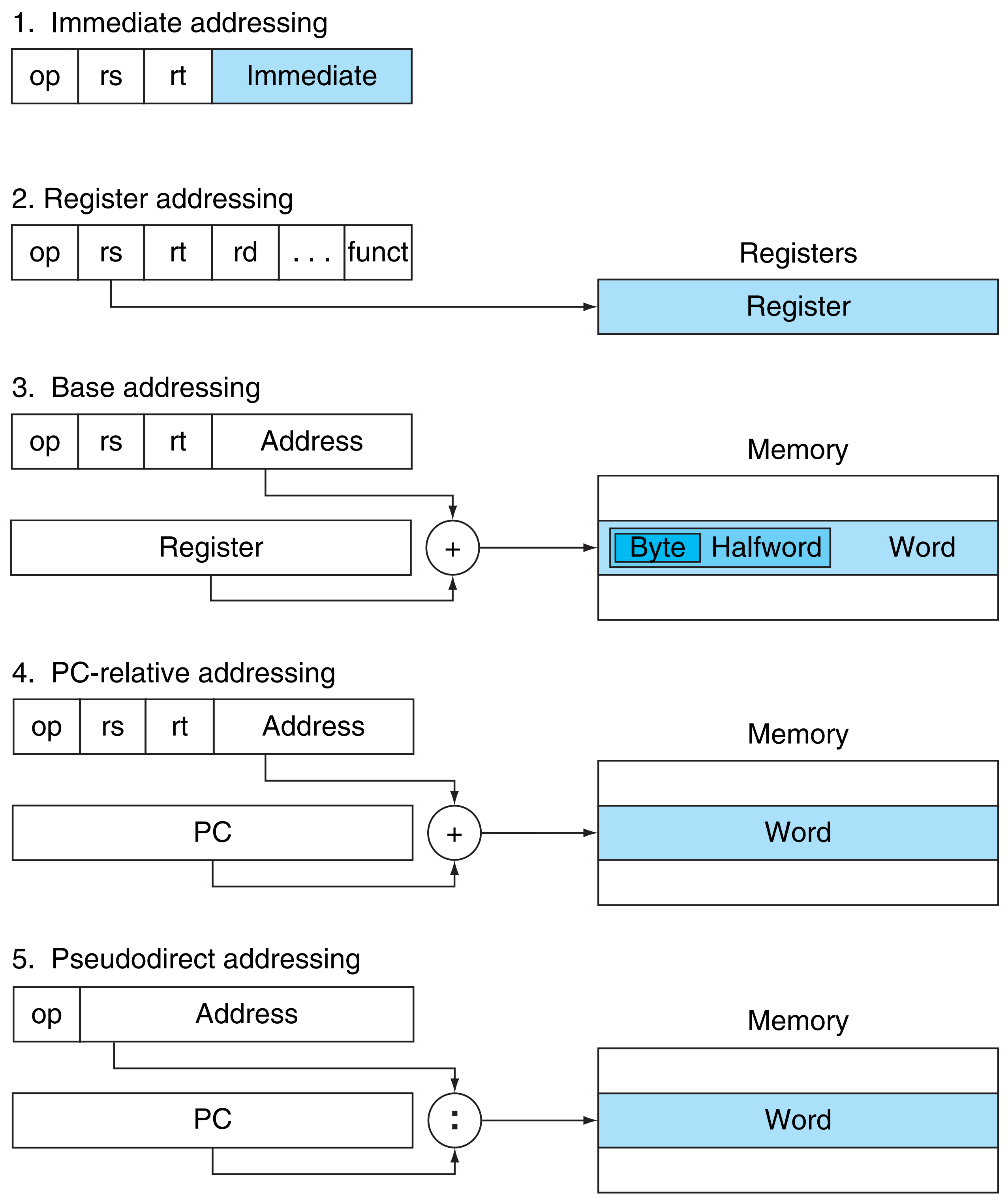
- Assume loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s5
      lw   $t0, 0($t1)
      bne  $t0, $s4, Exit
      addi $s3, $s3, 1
      j    Loop
Exit:
```



80000	0	0	19	9	4	0
80004	0	9	21	9	0	32
80008	35	9	8	0		
80012	5	8	20	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Addressing Mode Summary



Branching Far Away

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

- Example:

```
beq $s0,$s1, L1
```

$\xrightarrow{\textit{becomes}}$

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

Assembler Pseudoinstructions

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

- Pseudoinstructions are shorthand. They are recognized by the assembler but translated into small bundles of machine instructions.

`move $t0,$t1` $\xrightarrow{\textit{becomes}}$ `add $t0,$zero,$t1`

`blt $t0,$t1,L` $\xrightarrow{\textit{becomes}}$
```
slt $at,$t0,$t1
bne $at,$zero,L
```

- `$at` (register 1) is an “assembler temporary”

Programming Pitfalls

- Sequential words are not at sequential addresses -- increment by 4 not by 1!
- Keeping a pointer to an automatic variable (on the stack) after procedure returns

Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
    - R-type instruction
    - Function bits tell what instruction it is
- Otherwise, opcode tells what instruction it is

Machine Code

	op	rs	rt	imm
(0x2237FFF1)	001000	10001	10111	1111 1111 1111 0001
	2 2	3	7	F F F 1

	op	rs	rt	rd	shamt	funct
(0x02F34022)	000000	10111	10011	01000	00000	100010
	0 2	F	3	4	0	2 2

Field Values

op	rs	rt	imm
8	17	23	-15

op	rs	rt	rd	shamt	funct
0	23	19	8	0	34

Assembly Code

```
addi $s7, $s1, -15
sub $t0, $s7, $s3
```

In conclusion: Fallacies

1. Powerful (complex) instructions lead to higher performance
    - Fewer instructions are required
    - ***But*** complex instructions are hard to implement. As a result implementation may slow down all instructions including simple ones.
    - Compilers are good at making fast code from simple instructions.
2. Use assembly code for high performance
    - Modern compilers are better than predecessors at generating good assembly
    - More lines of code (in assembly) means more errors and lower productivity

In conclusion: More Fallacies

3. Backwards compatibility means instruction set doesn't change

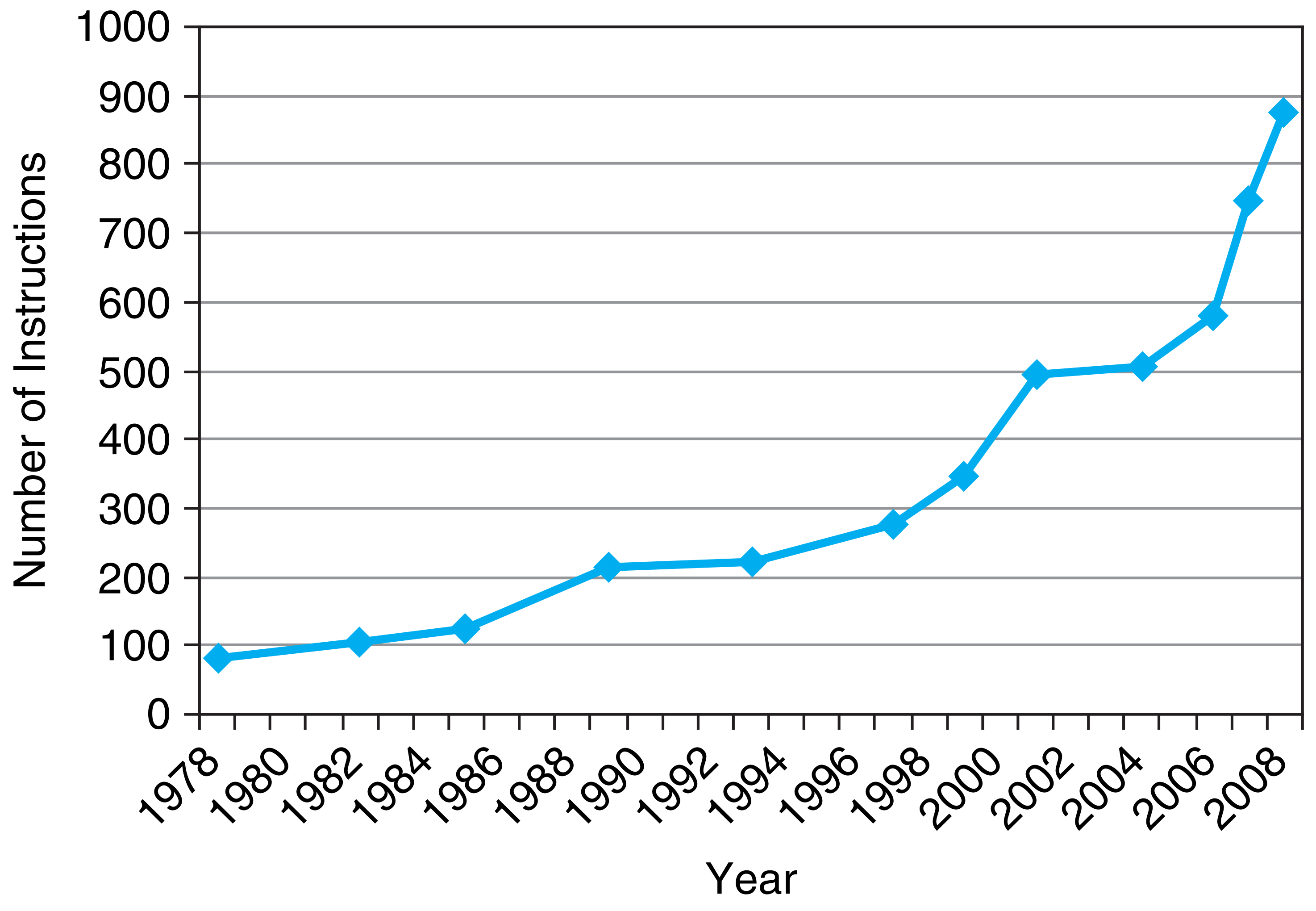


FIGURE 2.43 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors. Copyright © 2009 Elsevier, Inc. All rights reserved.