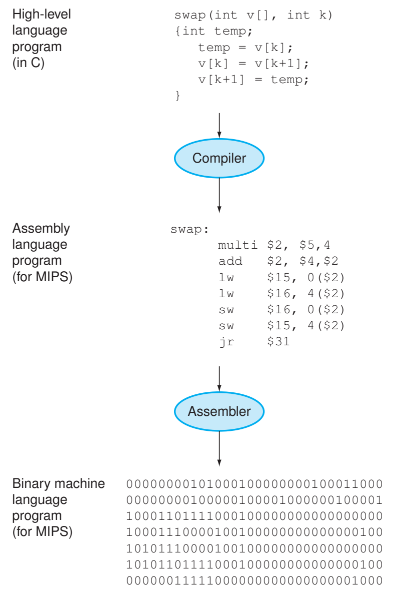
Concepts Introduced

- instruction set design principles
- subset of the MIPS assembly language
- correspondence between high-level language constructs and MIPS assembly code
- stored program concept
- process of transforming a high-level language program into a program that can execute on a computer

Instructions for a Machine

- A high-level language statement is typically translated into multiple assembly instructions.
- An assembly instruction is generally a symbolic representation of a single machine instruction.
- A machine instruction is a set of bits representing a basic operation that a machine can perform.
- An instruction set is the set of possible machine instructions for a specific machine.

Example of Compilation Process


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;
}
```

```
swap:
      multi $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
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Advantages of Using High-Level Languages

- Allows the programmer to think about the problem in a more natural language for the application's intended use.
- Improves programmer productivity.
- Improves program maintainability.
- Allows applications to be independent of the computer on which they are developed.
- Highly optimizing compilers can produce very efficient assembly code optimized for the target machine.

Why Learn Assembly Language?

- Knowledge of assembly language helps to understand concepts in computer organization and to a lesser extent concepts in operating systems, compilers, parallel systems, etc.
- To understand how high-level language constructs are actually implemented so that one can better use them.
  - control constructs (if, while-do, etc.)
  - pointers
  - parameter passing (pass-by-value, pass-by-reference, etc.)
- To help understand performance implications of high-level language features and run-time systems.

MIPS Introduction

- The MIPS is a RISC (Reduced Instruction Set Computer) instruction set, meaning that it has simple and few instructions.
- Introduced in the early 1980's.
- MIPS originally was an acronym for Microprocessor without Interlocked Pipeline Stages.
- The MIPS architecture has been used in SGI, NEC, Nintendo, and other computers.
- Many other instruction sets designed since the introduction of MIPS are similar (e.g. ARM).

RISC Design

- CISC (Complex Instruction Set Computer) ISA
  - Intel x86
- RISC (Reduced Instruction Set Computer) ISAs
  - MIPS, Sun SPARC, IBM PowerPC, ARM
- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited number of addressing modes
  - limited number of operations

Four ISA Design Principles

- Simplicity favors regularity.
- Smaller is faster.
- Make the common case fast.
- Good design demands good compromises.

General Classes of MIPS Assembly Instructions

- arithmetic operations (+, -, *, /)
- logical operations (&, |, ˜ , ˆ , «, »)
- data transfer (loads from memory or stores to memory)
- transfers of control (jumps, branches, calls, returns)

MIPS Instruction Operands

- integer constants
  - character
  - integer
  - address
- registers
  - integer
  - floating-point
  - special
- memory

MIPS Integer Constants

- Generally represented in 16 bits.
- Are extended to 32 bits before used in an operation.
- Constants represent signed values for most operations, though constants represent unsigned values for a few operations.
- Integer constants in MIPS assembly instructions can be represented using decimal, hexadecimal, or ASCII values.
- Why allow a part of an instruction to contain a small constant?
  - Constants are frequently used in programs.
  - Much faster to access a part of an instruction than to load a constant from memory.
  - **Makes the common case fast!**

MIPS Registers

- A limited number of data words reside in a processor and are stored in registers.
- advantages of using registers
  - Registers can be accessed much faster.
    - **Smaller is faster!**
  - Registers use much less power than memory accesses.
  - Registers require only a few bits to address, so their use decreases code size as multiple registers can be accessed in a single instruction.
- MIPS registers
  - 32 integer (*$0 - $31*)
  - *hi* and *lo*
  - 32 floating point (*$f0 - $f31*)

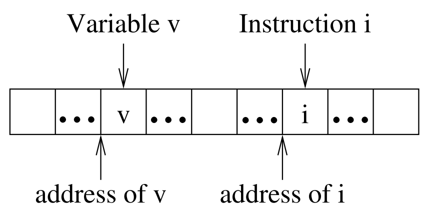
MIPS Integer Registers

- There are only 32 MIPS integer (general-purpose) registers.

Name	Number	Usage	Callee Must Preserve?
$zero	**$0**	**hardwired constant value zero**	**N/A**
$at	$1	reserved for use by assembler	no
$v0-$v1	**$2-$3**	**values for function results and expression evaluation**	**no**
$a0-$a3	**$4-$7**	**function arguments**	**no**
$t0-$t7	**$8-$15**	**temporaries**	**no**
$s0-$s7	**$16-$23**	**saved temporaries**	**yes**
$t8-$t9	**$24-$25**	**more temporaries**	**no**
$k0-$k1	$26-$27	reserved for use by OS kernel	N/A
$gp	$28	global pointer	yes
$sp	$29	stack pointer	yes
$fp	$30	frame pointer	yes
$ra	$31	return address	yes

Memory

- Memory contains both data and instructions.
- Memory can be viewed as a large array (vector) of bytes.
- The beginning of a variable or an instruction is associated with specific elements of this array.
- The address of a variable or an instruction is its offset from the beginning of memory.



MIPS Assembly File

- A MIPS assembly file consists of a set of lines.
- Each line can be:
  - directive
  - instruction
- Each directive or instruction can start with a label, which provides a symbolic name for a data or instruction location.
- Each line can include a comment, which start with a # character and continues to the end of the line.

General Form of a MIPS Assembly Language Program

- All directives and instructions are placed on separate lines.

```
        .data
        <declarations of variables>
        .text
        .globl main
main:
        <instructions>
        jr $ra    # instruction indicating a return
```

MIPS Directives

directive	meaning
`.align` *n*	Align next datum on 2^n boundary.
`.asciiz` *str*	Place the null-terminated string *str* in memory.
`.byte` *b1,...,bn*	Place the *n* byte values in memory.
`.data`	**Switch to the data segment.**
`.double` *d1,...,dn*	Place the *n* double precision values in memory.
`.float` *f1,...,fn*	Place the *n* single precision values in memory.
`.globl` *sym*	The label *sym* can be referenced in other files.
`.half` *h1,...,hn*	Place the *n* halfword values in memory.
`.space` ***n***	**Allocates *n* bytes of space.**
`.text`	**Switch to the text segment.**
`.word` ***w1,...,wn***	**Place the *n* word values in memory.**

MIPS Instructions

- general form

  <optional label> <operation> <operands>

- example assembly lines

```
loop:   addu $t2,$t3,$t4    # instruction with a label
        subu $t2,$t3,$t4    # instruction without a label
L2:     # A label can appear on a line by itself.
# A comment can appear on a line by itself.
```

MIPS Instruction Formats

- R format is used for shifts and instructions that reference only registers.
- I format is used for loads, stores, branches, and immediate instructions.
- J format is used for jump and call instructions.

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct
I format	op	rs	rt	immed		
J format	op	targaddr				

op – instruction opcode	shamt – shift amount
rs – first register source operand	funct – additional opcodes
rt – second register source operand	immed – offsets/constants
rd – register destination operand	targaddr – jump/call target

MIPS Instruction Size

- All MIPS instructions are 32 bits.
  - **Simplicity favors regularity!**
- Make simple instructions that are commonly needed fast and accomplish other operations as a series of simple instructions.
  - **Make the common case fast!**

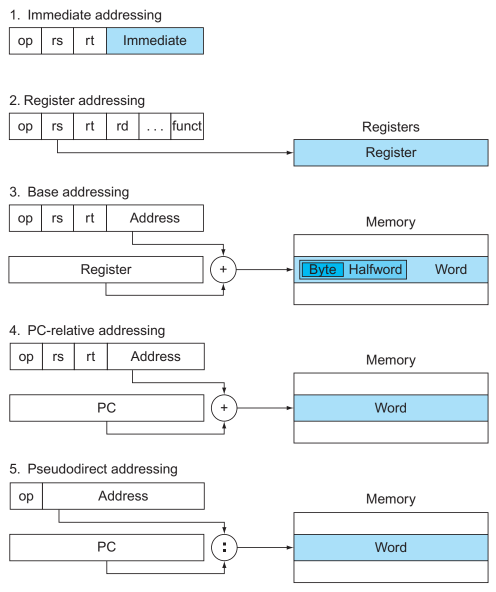
Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct
I format	op	rs	rt	immed		
J format	op	targaddr				

op – instruction opcode	shamt – shift amount
rs – first register source operand	funct – additional opcodes
rt – second register source operand	immed – offsets/constants
rd – register destination operand	targaddr – jump/call target

MIPS Addressing Modes

- addressing mode - a method for evaluating an operand
- MIPS addressing modes
  - **immediate** - operand contains an integer constant that is sign or zero extended
  - **register** - operand contains a register number that is used to access the integer or FP register file
  - **base displacement** - operand represents a data memory value whose address is the sum of the register value referenced by the register number and a sign-extended constant
  - **PC relative** - operand represents an instruction address that is the sum of the PC and a sign-extended constant
  - pseudodirect - operand represents an instruction address that is the field concatenated with the upper bits of the PC

MIPS Addressing Modes (cont.)



MIPS R Format

- The MIPS R format is used for instructions that only reference registers and for shift operations.
- The *op* field must have the value of zero for the R format to be used.
- The *funct* field indicates the type of operation to be performed for R format instructions.
- The *shamt* field is only used for the `sll`, `sra`, and `srl` instructions since the shift amount for words cannot exceed the unsigned value 31 (5 bits).

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct

op – instruction opcode
rs – first register source operand
rt – second register source operand
rd – register destination operand
shamt – shift amount
funct – additional opcodes

R Format Instruction Encoding Example

- R-format example: `addu $t2,$t3,$t4`

fields	op	rs	rt	rd	shamt	funct
size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
decimal	0	11	12	10	0	33
binary	000000	01011	01100	01010	00000	100001
hexadecimal	0x016c5021					

MIPS I Format

- The MIPS I format is used for arithmetic/logical immediate instructions, loads and stores, and conditional branches.
- The *op* field is used to identify the type of instruction.
- The *rs* field is used as a source register.
- The *rt* field is used as a source or destination register, depending on the instruction.
- The *immed* field is zero extended if it is a logical operation. Otherwise it is sign extended.

Name	Fields			
Field Size	6 bits	5 bits	5 bits	16 bits
I format	op	rs	rt	immed

op – instruction opcode
rs – first register source operand
rt – second register source operand
immed – offsets/constants

I Format Instruction Encoding Examples

- I-format example 1: `addiu $t0,$t0,1`

fields	op	rs	rt	immed
size	6 bits	5 bits	5 bits	16 bits
decimal	9	8	8	1
binary	001001	01000	01000	0000000000000001
hexadecimal	0x25080001			

- I-format example 2: `lw $s1,100($s2)`

fields	op	rs	rt	immed
size	6 bits	5 bits	5 bits	16 bits
decimal	35	18	17	100
binary	100011	10010	10001	0000000001100100
hexadecimal	0x8e510064			

I Format Instruction Encoding Examples (cont.)

- Conditional branches are also encoded using the I format.
- The branch displacement is a signed value in instructions (not bytes) from the point of the branch.
- branch example:

```
L2: instruction
    instruction
    instruction
    beq $t6,$t7,L2
```

fields	op	rs	rt	immed
size	6 bits	5 bits	5 bits	16 bits
decimal	4	14	15	-3
binary	000100	01110	01111	1111111111111101
hexadecimal	0x11cffffd			

MIPS J Format

- The MIPS J format is used for unconditional jumps and function calls.
- The op field is used to identify the type of instruction.
- The `targaddr` field is used to indicate an absolute target instruction address.

Name	Fields	
Field Size	6 bits	26 bits
J format	op	targaddr

op – instruction opcode
targaddr – jump/call target

J Format Instruction Encoding Example

- J-format jump example: `j L1`
- Assume L1 is at the address 4194340 in decimal, which is 400024 in hexadecimal. We fill the target field as an address in instructions (0x100009) rather than bytes (0x400024).

fields	op	target address
size	6 bits	26 bits
decimal	2	0x100009
binary	000010	000001 0000 0000 0000 0000 1001
hexadecimal	0x08100009	

QtSpim

- QtSpim is a self-contained simulator that allows you to load, assemble, execute, and interactively debug your MIPS assembly program.
- QtSpim is accessible on the Linux computers in the Majors lab.
- QtSpim can also be downloaded for free from:
  - http://sourceforge.net/projects/spimsimulator/files

QtSpim Syscalls

- Syscalls provide operating system services.
- QtSpim input/output (I/O) and exit occurs through syscalls.

Service	Call Code Arg	Other Arguments	Result
print_int	**$v0 = 1**	**$a0 = integer**	
print_float	$v0 = 2	$f12 = float	
print_double	$v0 = 3	$f12 = double	
print_string	$v0 = 4	$a0 = string address	
read_int	**$v0 = 5**		**integer in $v0**
read_float	$v0 = 6		float in $f0
read_double	$v0 = 7		double in $f0
read_string	$v0 = 8	$a0 = string address $a1 = max length	
exit	**$v0 = 10**		
print_char	$v0 = 11	$a0 = char	
read_char	$v0 = 12		char in $v0

Pseudoinstructions Used for System Calls

- A pseudoinstruction is an assembly instruction that is not directly implemented as a machine instruction.
- The following pseudoinstructions are useful for assigning arguments for system calls.

Example	Meaning	Comment
`li $v0,4`	`$v0 = 4`	assign constant to register
`la $a0,_a`	`$a0 = _a`	assign address to register
`move $a0,$t0`	`$a0 = $t0`	assign register to register

General Form of MIPS Arithmetic and Logical Instructions

- Most MIPS arithmetic/logical instructions require 3 operands.
  - Simplicity favors regularity.
- form 1: <operation> <dstreg>,<src1reg>,<src2reg>

Example	Meaning	Comment
`addu $t0,$t1,$t2`	`$t0 = $t1 + $t2`	**addition (without overflow)**
`subu $t1,$t2,$t3`	`$t1 = $t2 - $t3`	**subtraction (without overflow)**

- form 2: <operation> <dstreg>,<srcreg>,<constant>

Example	Meaning	Comment
`addiu $t1,$t2,1`	`$t1 = $t2 + 1`	**addition immediate (without overflow)**

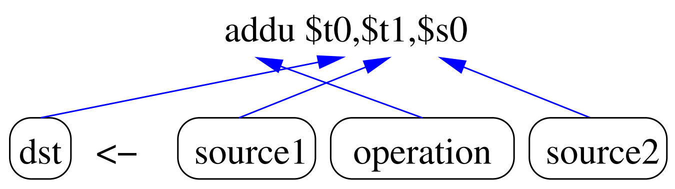
- Why is there no `subiu` instruction?

Order of MIPS Arithmetic and Logical Instruction Operands

addu $t0,$t1,$s0

dst <– source1 operation source2

Order of MIPS Arithmetic and Logical Instruction Operands



Using MIPS Arithmetic Instructions

- Consider the following C++ source code fragment.

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

- Assume the values of `f`, `g`, `h`, `i`, and `j` are associated with the registers `$t2`, `$t3`, `$t4`, `$t5`, and `$t6`, respectively. Write MIPS assembly code to perform this assignment assuming `$t7` is available.

```
addu $t2,$t3,$t4     # tmp1 = g + h
addu $t7,$t5,$t6     # tmp2 = i + j
subu $t2,$t2,$t7     # f = tmp1 - tmp2
```

MIPS Integer Multiply, Divide, and Modulus Instructions

- Integer multiplication, division, and modulus operations can also be performed.
- MIPS provides two extra registers, `hi` and `lo`, to support division and modulus operations.

Example	Meaning	Comment
`mult $t1,$t2`	**`$lo = $t1 * $t2`**	**multiplication**
`divu $t2,$t3`	**`$lo = $t2 / $t3`** **`$hi = $t2 % $t3`**	**division and modulus**
`mflo $t1`	**`$t1 = $lo`**	**move from $lo register**
`mfhi $t1`	**`$t1 = $hi`**	**move from $hi register**

Example of Calculating a Quotient and a Remainder

- Given values `$t1` and `$t2`, the following sequence of MIPS instructions assigns the quotient (`$t1 / $t2`) to `$s0` and the remainder (`$t1 % $t2`) to `$s1`.

```
divu  $t1,$t2      # perform both division
                   # and modulus operations
mflo  $s0          # move quotient into $s0
mfhi  $s1          # move remainder into $s1
```

Boolean Operations

- The following table shows the basic boolean operations.

X	Y	not X	X and Y	X or Y	X nand Y	X nor Y	X xor Y
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

Logical Operations

- Logical operations consist of bitwise boolean operations and shifting operations.
- These operations can be used to extract or insert fields of bits within a word.

General Form of MIPS Bitwise Instructions

- Bitwise instructions apply boolean operations on each of the corresponding pairs of bits of two values.

Example	Meaning	Comment
`and $t2,$t3,$t4`	**`$t2 = $t3 & $t4`**	**bitwise and**
`or $t3,$t4,$t5`	**`$t3 = $t4 | $t5`**	**bitwise or**
`nor $t4,$t3,$t6`	`$t4 = ~($t3 | $t6)`	bitwise nor
`xor $t7,$t2,$t4`	`$t7 = $t2 ^ $t4`	bitwise xor
`andi $t2,$t3,7`	`$t2 = $t3 & 7`	bitwise and with immediate
`ori $t3,$t4,5`	`$t3 = $t4 | 5`	bitwise or with immediate
`xori $t7,$t2,6`	`$t7 = $t2 ^ 6`	bitwise xor with immediate

General Form of MIPS Shift Instructions

- Shift instructions move the bits in a word to the left or right by a specified amount.
- Shifting left (right) by i is the same as multiplying (dividing) by 2^i.
- An arithmetic right shift replicates the most signficiant bit to fill in the vacant bits.
- A logical right shift fills in the vacant bits with zero.

Example	Meaning	Comment
`sll $t2,$t3,2`	`$t2 = $t3 << 2`	shift left logical
`sllv $t3,$t4,$t5`	`$t3 = $t4 << $t5`	shift left logical variable
`sra $t4,$t3,1`	`$t4 = $t3 >> 1`	shift right arithmetic (signed)
`srav $t7,$t2,$t4`	`$t7 = $t2 >> $t4`	shift right arithmetic variable (signed)
`srl $t2,$t3,7`	`$t2 = $t3 >> 7`	shift right logical (unsigned)
`srlv $t3,$t4,$t6`	`$t3 = $t4 >> $t6`	shift right logical variable (unsigned)

Constructing Global Addresses and Large Constants

- The `lui` instruction can be used to construct large constants or addresses. It loads a 16-bit value in the 16 most significant bits of a word and clears the 16 least significant bits.

Form	Example	Meaning	Comment
`lui <dreg>,<const>`	`lui $t1,12`	`$t1 = 12 << 16`	load upper immediate

- example: Load `131,071` (or `0x1ffff`) into `$2`.

```
lui $2,1            # put 1 in the upper half of $2
ori $2,$2,0xffff    # set all bits in the lower half
```

- Having all instructions the same size and a reasonable length means having to construct global addresses and some constants in two instructions.
- **Good design demands good compromises!**

General Form of MIPS Data Transfer Instructions

- The MIPS can only access memory with load and store instructions.
- form: <operation> <reg1>,<constant>(<reg2>)

Example	Meaning	Comment
`lw $t2,8($t3)`	$\mathbf{\$t2 =_{32} Mem[\$t3 + 8]}$	**32-bit load**
`lh $t3,0($t4)`	$\$t3 =_{32} (Mem[\$t4]_0)^{16}\ \#\#\ Mem[\$t4]$	signed 16-bit load
`lhu $t8,2($t3)`	$\$t8 =_{32} 0^{16}\ \#\#\ Mem[\$t3 + 2]$	unsigned 16-bit load
`lb $t4,0($t5)`	$\$t4 =_{32} (Mem[\$t5]_0)^{24}\ \#\#\ Mem[\$t5]$	signed 8-bit load
`lbu $t6,1($t9)`	$\$t6 =_{32} 0^{24}\ \#\#\ Mem[\$t9 + 1]$	unsigned 8-bit load
`sw $t5,-4($t2)`	$\mathbf{Mem[\$t2 - 4] =_{32} \$t5}$	**32-bit store**
`sh $t6,12($t3)`	$Mem[\$t3 + 12] =_{16} \$t6_{16..31}$	16-bit store
`sb $t7,1($t3)`	$Mem[\$t3 + 1] =_{8} \$t7_{24..31}$	8-bit store

Using Data Transfer Instructions

- Consider the following source code fragment.

```
int a, b, c, d;
...
a = b + c - d;
```

- Assume the addresses of a, b, c, and d are in the registers $t2, $t3, $t4, and $t5, respectively. The following MIPS assembly code performs this assignment assuming $t6 and $t7 are available.

```
lw   $t6,($t3)     # load b into tmp1
lw   $t7,($t4)     # load c into tmp2
addu $t6,$t6,$t7   # tmp1 = tmp1 + tmp2
lw   $t7,($t5)     # load d into tmp2
subu $t6,$t6,$t7   # tmp1 = tmp1 - tmp2
sw   $t6,($t2)     # store tmp1 into a
```

Indexing Array Elements with a Variable Index

- Assembly code can be written to access array elements using a variable index. Consider the following source code fragment.

```
int a[100], i;
...
a[i] = a[i] + 1;
```

- Assume the value of i is in $t0. The following MIPS code performs this assignment.

```
     .data
_a:  .space 400            # declare space for array _a
     ...
     la    $t1,_a          # load address of _a
     sll   $t2,$t0,2       # determine offset from _a
     addu  $t2,$t2,$t1     # add offset and _a
     lw    $t3,0($t2)      # load the value
     addiu $t3,$t3,1       # add 1 to the value
     sw    $t3,0($t2)      # store the value
```

Memory Alignment Requirements

- The MIPS requires alignment of memory references to be an integer multiple of the size of the data being accessed.
- These alignments are enforced by the compiler.
- The processor checks this alignment requirement by inspecting the least significant bits of the address.

```
Byte:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Half:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX0
Word:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX00
Double: XXXXXXXXXXXXXXXXXXXXXXXXXXXXX000
```

Memory Byte Order

- byte order
  - Big Endian - most significant byte is the word address
  - Little Endian - least significant byte is the word address
- Consider the MIPS directive specifying that four bytes should be allocated with the values 0-3.

```
.align 4
.byte  1,2,3,4
```

- Below are the order and values of the bytes.

MSB to LSB bytes	1	2	3	4
Big Endian byte values	1	2	3	4
Little Endian byte values	4	3	2	1

- The order of the bytes in a word in QtSpim depends on the byte order of the underlying machine where the simulator is executing.

Transfer of Control Instructions

- Transfers of control instructions can cause the next instruction to be executed that is not the next sequential instruction.
- Transfers of control are used to implement control statements in high-level languages.
  - unconditional (goto, break, continue, call, return)
  - conditional (if-then, if-then-else, switch)
  - iterative (while, do, for)

General Form of MIPS Jump and Branch Instructions

- MIPS provides direct jumps to support unconditional transfers of control to a specified location.
- MIPS provides indirect jumps to support returns and switch statements.
- MIPS provides conditional branch instructions to support decision making. MIPS conditional branches test if the values of two registers are equal or not equal.

General Form	Example	Meaning	Comments
j <label>	**j L1**	**goto L1;**	**direct jump**
jr <sreg>	jr $ra	goto $ra;	indirect jump
beq <s1reg>,<s2reg>,<label>	**beq $t2,$t3,L1**	**if ($t2 == $t3) goto L1;**	**branch equal**
bne <s1reg>,<s2reg>,<label>	**bne $t2,$t3,L1**	**if ($t2 != $t3) goto L1;**	**branch not equal**

Example of Translating an If Statement

- example source statement:

```
if (i == j)
   k = k+i;
```

- Translate into MIPS instructions assuming `i`, `j`, and `k`, are in the registers `$t2`, `$t3`, and `$t4`, respectively.

```
   bne  $t2,$t3,L1      # if ($t2 != $t3) goto L1
   addu $t4,$t4,$t2     # k = k + i
L1:
```

General Form of MIPS Comparison Instructions

- MIPS provides *set less than* instructions that set a register to 1 if the first source register is less than the value of the second operand. Otherwise it sets it to 0.
- There are versions to perform unsigned comparisons as well.

General Form	Example	Meaning	Comments
slt <dreg>,<s1reg>,<s2reg>	**slt $t2,$t3,$t4**	**if ($t3 < $t4) $t2 = 1; else $t2 = 0;**	**compare less than**
sltu <dreg>,<s1reg>,<s2reg>	sltu $t2,$t3,$t4	if ($t3 < $t4) $t2 = 1; else $t2 = 0;	compare less than unsigned
slti <dreg>,<sreg>,<const>	slti $t2,$t3,100	if ($t3 < 100) $t2 = 1; else $t2 = 0;	compare less than constant
sltiu <dreg>,<s1reg>,<const>	sltiu $t2,$t3,100	if ($t3 < 100) $t2 = 1; else $t2 = 0;	compare less than constant unsigned

Example of Translating an If-Then-Else Statement

- example source statement:

```
if (a < b)
   c = a;
else
   c = b;
```

- Translate into MIPS instructions assuming a, b, and c, are in the registers $t2, $t3, and $t4, respectively. Assume $t5 is available.

```
   slt  $t5,$t2,$t3      # a < b
   beq  $t5,$zero,L1     # if ($t5 == 0) goto L1
   move $t4,$t2          # c = a
   j    L2               # goto L2
L1:
   move $t4,$t3          # c = b
L2:
```

Translating an If-Statement with a Different Condition

- example source statement:

```
if (a > b)
   c = a;
```

- Translate into MIPS instructions assuming a, b, and c, are in the registers $t2, $t3, and $t4, respectively. Assume $t5 is available.

```
   slt  $t5,$t3,$t2      # b < a
   beq  $t5,$zero,L1     # if ($t5 == 0) goto L1
   or   $t4,$t2,$zero    # c = a
L1:
```

Translating Other High-Level Control Statements

- How can we translate other high-level control statements (while, do, for)?
- We can first express the C statement using C `if` and `goto` statements.
- After that we can translate using MIPS unconditional jumps (j), comparisons (slt, slti), and conditional branches (beq, bne).

Example of Translating a For Statement

- example source statement:

```
sum = 0;
for (i = 0; i < 100; i++)
   sum += a[i];
```

- First, we replace the `for` statement using an `if` and `goto` statements.

```
      sum = 0;
      i = 0;
      goto test;
loop: sum += a[i];
      i++;
test: if (i < 100) goto loop;
```

Example of Translating a For Statement (cont.)

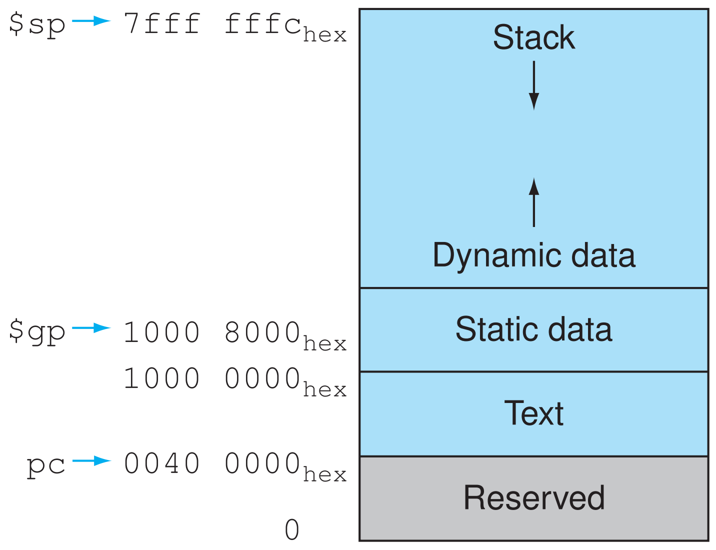
- We can next translate into MIPS instructions.
- Assume sum, i, and the starting address of a, are in $t2, $t3, and $t4, respectively and that $t5 is available.

```
    li    $t2,0            # sum = 0
    move  $t3,$zero        # i = 0
    j     test             # goto test
 loop:
    sll   $t5,$t3,2        # tmp = i*4
    addu  $t5,$t5,$t4      # tmp = tmp + &a
    lw    $t5,0($t5)       # load a[i] into tmp
    addu  $t2,$t2,$t5      # sum += tmp
    addiu $t3,$t3,1        # i++
 test:
    slti  $t5,$t3,100      # test i < 100
    bne   $t5,$zero,loop   # if true goto loop
```

Types of Memory in a Process

- There are four general areas of memory in a process.
  - The *text* area contains the instructions for the application and is fixed in size.
  - The *static data* area is also fixed in size and contains:
    - global variables
    - static local variables
    - string and sometimes floating-point constants
  - The *run-time stack* contains activation records, each containing information associated with a function invocation.
    - saved values of callee-save registers
    - local variables and arguments not allocated to registers
    - space for the maximum words of arguments passed on stack to other functions
  - The *heap* contains dynamically allocated data (*new* operator in C++ or *malloc* function call in C).

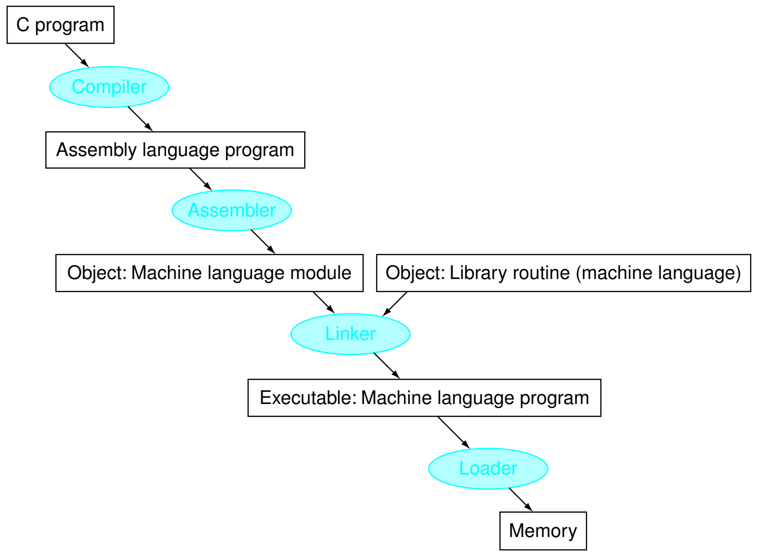
Memory Organization of a Process



Translation Process Steps

- Preprocessing
- Compiling
- Assembling
- Linking
- Loading

Translating C Code to Be Executed


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Preprocessor

- Some preliminary processing is performed on a C or C++ file.
  - definitions and macros
  - file inclusion
  - conditional compilation
  - line numbering

Compiler

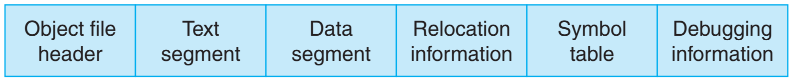
- Compiling is referred to as both the entire translation process from source file to executable or the step that translates a source file in a high-level language (sometimes already preprocessed) and produces an assembly file.
- Compilers are also responsible for checking for correct syntax, making semantic checks, and performing optimizations to improve performance, code size, and energy usage.

Assembler

- Assemblers take an assembly language file as input and produce an object file as output.
- Assembling is typically accomplished in two passes.
  - Stores all of the identifiers representing addresses or values in a table as there can be forward references.
  - Translates the instructions and data into bits for the object file.

Object File

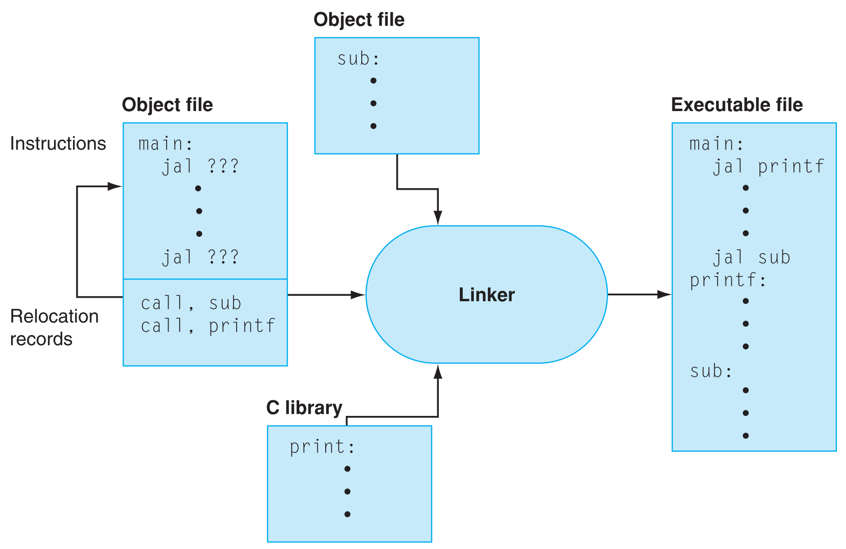
- The object file contains:
  - an object file header describing the size and position of the other portions of the object file
  - the text segment containing the machine instructions
  - the data segment containing the data values
  - relocation information identifying the list of instructions and data words that depend on absolute addresses
  - a symbol table containing global labels and associated addresses in object file and the list of unresolved references
  - debugging information to allow a symbolic debugger to associate machine instruction addresses with source line statements and data addresses with variable names



Linker

- Linkers take object files and object libraries as input and produce an executable file as output.
- Linkers also resolve external references by either finding the symbols in another object file or in a library. The linker aborts if any external references cannot be resolved.
- The linker determines the addresses of absolute references using the relocation information in the object files.
- The executable has a similar format as object files with no unresolved references and no relocation information.

Linking Example

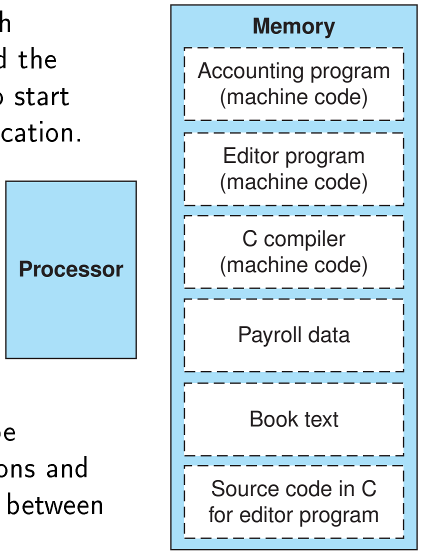


Loader

- The loader copies the executable file (or a portion of it) from disk into memory so it can start executing.
  - Reads the executable file's header to determine the size of the text and data segments.
  - Allocates the address space for the process (text, static data, heap, and stack segments).
  - Copies the instructions into the text segment and data into the static data segment.
  - Copies arguments passed to the program onto the stack.
  - Initializes the machine registers.
  - Jumps to a start-up routine that will call the `main` function.

Stored Program Concept

- Memory can contain both instructions and data and the computer is instructed to start executing at a specific location.
- Different programs can be loaded in different locations and the processor can switch between processes very quickly.


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Goals When Designing an Instruction Set

- Make the processor easy to build.
- Make the compiler and the assembler easy to construct.
- Maximize performance and energy efficiency.
- Minimize cost.

Instruction Set Design Principles

- Simplicity favors regularity.
  - Keeping all instructions a single size.
  - Most arithmetic instructions have three operands.
- Smaller is faster.
  - There are only 32 registers in the register set.
- Good design demands good compromises.
  - Having all instructions the same size and a reasonable length means having to construct addresses and some constants in two instructions.
- Make the common case fast.
  - Make simple instructions that are more commonly needed fast and accomplish operations as a series of simple instructions.

Fallacies and Pitfalls

- Fallacy: More powerful instructions mean higher performance.
- Fallacy: One should write in assembly language to obtain the highest performance.