## Concepts Introduced

- Introduction
- Number Representation
- Assembly 1
- Assembly 2

## Computer Measurement

- Execution time
- Performance
- Clock period and clock rate
- CPU time, CPI (cycles per instruction)
- Amdahl's Law

## Performance Equations

- Performance has an inverse relationship to execution time.

$$Performance = \frac{1}{Execution\_Time}$$

- Comparing the performance of two machines can be accomplished by comparing execution times.

$$Performance_X > Performance_Y$$

$$\frac{1}{Execution\_Time_X} > \frac{1}{Execution\_Time_Y}$$

$$Execution\_Time_Y > Execution\_Time_X$$

## N Times Faster

- Often people state that a machine X is $n$ times faster than a machine Y. What does this mean?

$$\frac{Performance_X}{Performance_Y} = n = \frac{Execution\_Time_Y}{Execution\_Time_X}$$

- If machine X takes 20 seconds to perform a task and machine Y takes 2 minutes to perform the same task, then machine X is how many times faster than machine Y?

## Measures of Clock Speed

- clock periods
  - millisecond (ms) - $10^{-3}$ of a second
  - microsecond ($\mu$s) - $10^{-6}$ of a second
  - nanosecond (ns) - $10^{-9}$ of a second
  - picosecond (ps) - $10^{-12}$ of a second
  - femtosecond (fs) - $10^{-15}$ of a second
- clock rates
  - kilohertz (KHz) - $10^{3}$ cycles per second
  - megahertz (MHz) - $10^{6}$ cycles per second
  - gigahertz (GHz) - $10^{9}$ cycles per second
  - terahertz (THz) - $10^{12}$ cycles per second
  - petahertz (PHz) - $10^{15}$ cycles per second
- If the clock period for a computer is 2ns, then what is its clock rate?
- Why do computer manufacturers quote clock rates instead of clock periods?

## Measures of Data Size

- bit - Binary digIT
- nibble - four bits
- byte - eight bits
- word - often four bytes (32 bits) on many embedded/mobile processors and eight bytes (64 bits) on many desktops and servers
- kibibyte (Kib) [kilobyte (Kb)] - $2^{10}$ (1,024) bytes
- mebibyte (Mib) [megabyte (Mb)] - $2^{20}$ (1,048,576) bytes
- gibibyte (Gib) [gigabyte (Gb)] - $2^{30}$ (1,073,741,824) bytes
- tebibyte (Tib) [terabyte (Tb)] - $2^{40}$ (1,099,511,627,776) bytes
- pebibyte (Pib) [petabyte (Pb)] - $2^{50}$ (1,125,899,906,842,624) bytes

## CPU Time

- CPU time ignores I/O and the time for executing other processes.
- CPI stands for cycles per instruction.

$$CPU\_time = CPU\_clock\_cycles * clock\_cycle\_time = \frac{CPU\_clock\_cycles}{clock\_rate}$$

$$CPI = \frac{CPU\_clock\_cycles}{instruction\_count}$$

$$CPU\_time = Instruction\_count * CPI * clock\_cycle\_time$$

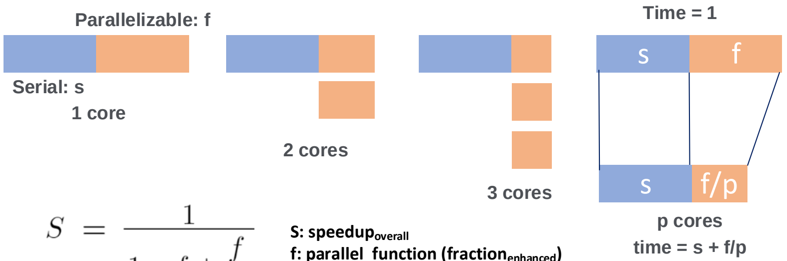
- Suppose two implementations of the same ISA are executing the same program. Computer A has a clock cycle time of 250ps and a CPI of 2.0. Computer B has a clock cycle time of 500ps and a CPI of 1.2. Which computer is faster and by how much?

## CPU Time (cont.)

- CPI cannot be looked up in a manual as it can be affected by many external events.
  - Pipelines can be flushed.
  - Branch information can be replaced in various buffers (BPB, BTB).
  - Page translation information can be evicted (ITLB, DTLB).
  - Blocks of data or instructions can be evicted from cache or memory.
- CPU time really needs to be measured and it can vary somewhat on each execution.

## Amdahl's Law

- Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.



$$S = \frac{1}{1 - f + \frac{f}{p}}$$

**S: speedup$_{\text{overall}}$**
**f: parallel_function (fraction$_{\text{enhanced}}$)**
**p: process_cores (speedup$_{\text{enhanced}}$)**

## Amdahl's Law

- Amdahl's Law depends on two factors:
  - The fraction of the time the enhancement can be exploited.
  - The improvement gained by the enhancement while it is exploited.

$$speedup_{overall} = \frac{execution\_time_{old}}{execution\_time_{new}} = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{speedup_{enhanced}}}$$

$$execution\_time_{new} = execution\_time_{old} * (1 - fraction_{enhanced} + \frac{fraction_{enhanced}}{speedup_{enhanced}})$$

- If the speed of a CPU is improved by a factor of 5 and the CPU requires 40% of the machines execution time, then what is the overall speedup?

$$\frac{1}{1 - 0.4 + \frac{0.4}{5}}$$

## Commonly Used Bases

- Which base is commonly used for computers and why?

| Base | Name | Digits | Example |
|---|---|---|---|
| 10 | Decimal | 0-9 | $5023_{10}$ |
| 2 | Binary | 0-1 | $1001110011111_2$ |
| 8 | Octal | 0-7 | $11637_8$ |
| 16 | Hexadecimal | 0-9,a-f | $139f_{16}$ |

## Binary Number Representation

- The binary representation contains two symbols: { 0, 1 }
- Position of each symbol represents a power of two

$$d_n d_{n-1} \ldots d_1 d_0 . d_{-1} \ldots d_{-m} = d_n \times 2^n + d_{n-1} \times 2^{n-1} + \ldots + d_1 \times 2^1 + d_0 \times 2^0$$
$$+ d_{-1} \times 2^{-1} + d_{-2} \times 2^{-2} + \ldots + d_{-m} \times 2^{-m}$$
$$= \sum_{i=-m}^{n} d_i \times 2^i$$

- What is the value of the binary representation 111?

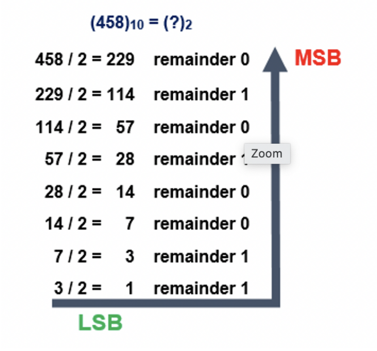
$$\begin{array}{llll} & 1 & 1 & 1 \\ & \uparrow & \uparrow & \uparrow \\ \text{position:} & 2 & 1 & 0 \end{array}$$

$$\begin{aligned} 111 &= 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 4 + 1 \times 2 + 1 \times 1 \\ &= 4 + 2 + 1 = 7 \end{aligned}$$

- It is also the way to convert binary number to decimal number.

## Conversion from Decimal to Binary Number

- Converting from decimal to binary (before the decimal point):
  - Repeatedly divide it by 2, until the quotient is 0.
  - Write down the remainder, the last remainder first.
- Example:

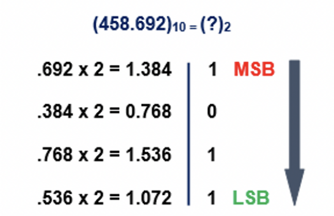


$(458)_{10} = (?)_2$

| | |
|---|---|
| 458 / 2 = 229 | remainder 0 |
| 229 / 2 = 114 | remainder 1 |
| 114 / 2 = 57 | remainder 0 |
| 57 / 2 = 28 | remainder 1 |
| 28 / 2 = 14 | remainder 0 |
| 14 / 2 = 7 | remainder 0 |
| 7 / 2 = 3 | remainder 1 |
| 3 / 2 = 1 | remainder 1 |

## Conversion from Decimal to Binary Number (cont.)

- Converting from decimal to binary (After the decimal point):
  - Multiply the fractional part including decimal point by 2 until the first digit after decimal point becomes 0.
  - Write down the integral part, the first remainder first.
- Example



$(458.692)_{10} = (?)_2$

.692 x 2 = 1.384 | 1 MSB
.384 x 2 = 0.768 | 0
.768 x 2 = 1.536 | 1
.536 x 2 = 1.072 | 1 LSB

- So: $(458.692)_{10} = (111001010.1011)_2$

## Conversion between Binary and Hexadecimal Number

- Extremely easy.
  - From base 2 to base 16: divide the digits into groups of 4, then apply the table.
  - From base 16 to base 2: replace every digit by a 4-bit string according to the table.
- Because 16 is 2 to the power of 4.
- Examples:

base 2 to base 16

0001010111011001
$\rightarrow 0001_2 0101_2 1101_2 1001_2$
$\rightarrow 1_{16} 5_{16} D_{16} 9_{16}$
$\rightarrow 15D9_{16}$

base 16 to base 2

$A2D4_{16}$
$\rightarrow A_{16} 2_{16} D_{16} 4_{16}$
$\rightarrow 1010_2 0010_2 1101_2 0100_2$
$\rightarrow 1010001011010100$

## Unsigned and Signed Integers

- Unsigned integers: non-negative numbers
  Binary representation: $d_{n-1}d_{n-2}\cdots d_1d_0 =$
  $d_{n-1} * 2^{n-1} + d_{n-2} * 2^{n-2} + \cdots + d_1 * 2^1 + d_0 * 2^0$
- Signed integers:
  - Non-negative integers: MSB is 0
  - Negative integers: MSB is 1

  Two's complement representation: $d_{n-1}d_{n-2}\cdots d_1d_0 =$
  $-d_{n-1} * 2^{n-1} + d_{n-2} * 2^{n-2} + \cdots + d_1 * 2^1 + d_0 * 2^0$

## Two's Complement Negation

- Negation of a two's compliment number is accomplished by inverting the bits and adding 1.

$$\begin{aligned} x + \overline{x} &= 111...111 = -1 \\ x + \overline{x} + 1 &= 0 \\ \overline{x} + 1 &= -x \end{aligned}$$

- Example 1:

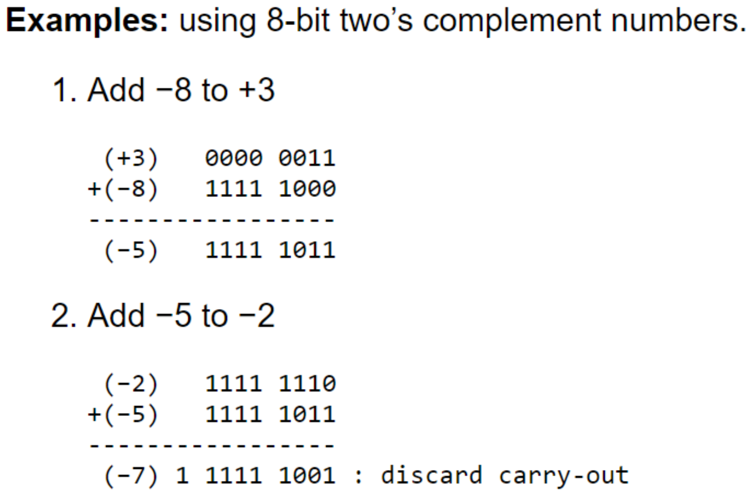
$$\begin{aligned} 3_{10} &= 00000000000000000000000000000011_2 \\ -3_{10} &= 11111111111111111111111111111100_2 + 1_2 \\ &= 11111111111111111111111111111101_2 \\ &= -3_{10} \end{aligned}$$

- Example 2:

$$\begin{aligned} -3_{10} &= 11111111111111111111111111111101_2 \\ 3_{10} &= 00000000000000000000000000000010_2 + 1_2 \\ &= 00000000000000000000000000000011_2 \\ &= 3_{10} \end{aligned}$$

## Binary Addition

- Rules: add the values and discard any carry-out bit



**Examples:** using 8-bit two's complement numbers.

1. Add −8 to +3

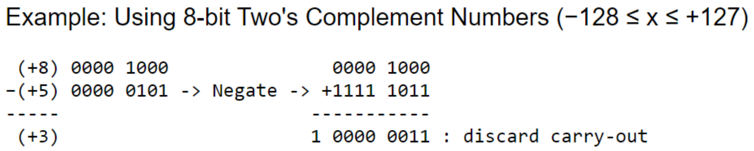
```
 (+3)   0000 0011
+(-8)   1111 1000
-----------------
 (-5)   1111 1011
```

2. Add −5 to −2

```
 (-2)   1111 1110
+(-5)   1111 1011
-----------------
 (-7) 1 1111 1001 : discard carry-out
```

## Binary Subtraction

- Rules: if we consider $A - B$, we first negate $B$ and add it to $A$ and discard any carry-out bit



Example: Using 8-bit Two's Complement Numbers (−128 ≤ x ≤ +127)

```
 (+8) 0000 1000                 0000 1000
-(+5) 0000 0101 -> Negate -> +1111 1011
-----                          ----------
 (+3)                         1 0000 0011 : discard carry-out
```

## Detecting Overflow

- Overflow occurs when the result of performing an arithmetic operation is not within the range of representable values.
- Overflow on addition only occurs when the MSBs of the operands are the same and the MSB of the result differs.

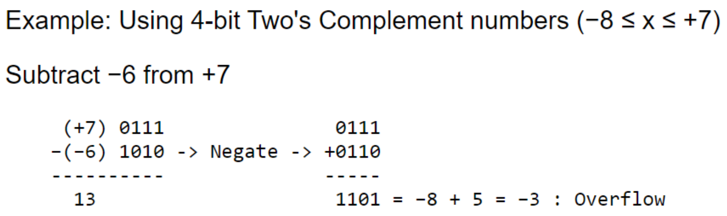
| A + B | | | |
|---|---|---|---|
| Operand A | Operand B | Result | Overflow? |
| $\geq 0$<br>$< 0$ | $\geq 0$<br>$< 0$ | $< 0$<br>$\geq 0$ | Yes |
| $\geq 0$<br>$< 0$ | $\geq 0$<br>$< 0$ | $\geq 0$<br>$< 0$ | No |
| $\geq 0$<br>$\geq 0$<br>$< 0$<br>$< 0$ | $< 0$<br>$< 0$<br>$\geq 0$<br>$\geq 0$ | $< 0$<br>$\geq 0$<br>$< 0$<br>$\geq 0$ | No<br>No<br>No<br>No |

## Overflow for Addition

Example: Using 4-bit Two's Complement numbers ($-8 \le x \le +7$)

```
 (-7)   1001
+(-6)   1010
------------
(-13) 1 0011 = 3 : Overflow (largest -ve number is -8)
```

## Overflow for Subtraction



Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)

Subtract −6 from +7

```
 (+7) 0111                 0111
-(-6) 1010 -> Negate -> +0110
----------              -----
  13                       1101 = -8 + 5 = -3 : Overflow
```

## Zero Extension – Extending to a Larger Unsigned Representation

- Converting an unsigned value to a larger representation is called zero extension.
- Zero extension is accomplished by filling in the new bits of the larger representation with zero.

| C Type | Number | Decimal |
|---|---:|---:|
| unsigned char | 10010110 | 150 |
| unsigned short | 0000000010010110 | 150 |
| unsigned int | 00000000000000000000000010010110 | 150 |

## Sign Extension – Extending to a Larger Two's Complement Representation

- Converting a two's complement value to a larger representation is called sign extension.
- Sign extension is accomplished by taking the most significant bit from the value in the smaller representation and replicating it to fill in the new bits of the larger representation.

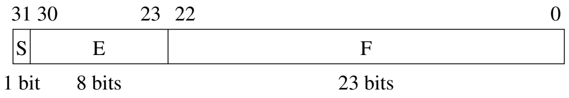
| C Type | Number | Decimal |
|---|---:|---:|
| char | `00000011` | 3 |
| short | `0000000000000011` | 3 |
| int | `00000000000000000000000000000011` | 3 |
| char | `11111101` | −3 |
| short | `1111111111111101` | −3 |
| int | `11111111111111111111111111111101` | −3 |

## Scientific Notation

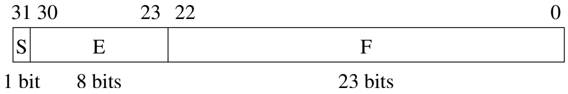
- Scientific notation uses 3 integers to represent values. representation.
  - radix (or base) r
  - significand (or mantissa) s
  - exponent x
- The standard form is:
  - $s * r^x$
- 15.625 can be represented as:
  - $15.625 * 10^0$
  - $1.5625 * 10^1$
  - $15625 * 10^{-3}$

## IEEE 754 Floating-Point Standard

- A standard for floating-point representation called the IEEE 754 Floating-Point Standard (FPS) is now widely used.
  - Programs become more portable since the results of floating-point operations are similar across different machines.
  - Floating-point data can be transferred from one machine to another without performing conversions.
- Below is the format used for representation of single precision floating-point values in the IEEE FPS, where $S$ is the sign bit, $E$ is a biased exponent, and $F$ represents the bits of the significand with the leading bit hidden.



## IEEE 754 Floating-Point Standard (cont.)



- Below is how the IEEE FPS format is interpreted.

  $(-1)^S * (1 + 0.F) * 2^{(E-127)}$
- The 1 is added to 0.F to make the leading bit of normalized binary numbers implicit and save a bit of space. This is referred to as the hidden bit.
- The E is adjusted by a bias of 127. Positive exponents will have larger unsigned biased value than negative exponents. This makes it easier to determine the larger magnitude of two IEEE FPS values (The larger the E, the larger the magnitude).

## Example of Representing a Value in IEEE FPS

- Determine the hexadecimal IEEE FPS pattern that represents the decimal value 9.5.
- IEEE FPS pattern: $(-1)^S * (1 + 0.F) * 2^{(E-127)}$

$$\begin{aligned} 9.5_{10} &= 1001.1_2 \\ &= 1.0011_2 * 2^3 \\ &= (-1)^0 * (1 + 0.0011_2) * 2^{(130-127)} \\ &= (-1)^S * (1 + 0.F) * 2^{(E-127)} \end{aligned}$$

$$\begin{aligned} FPS &= S\ E\ F \\ &= 0\ 130_{10}\ 00110000000000000000000_2 \\ &= 0\ 10000010_2\ 00110000000000000000000_2 \\ &= 0100_2\ 0001_2\ 0001_2\ 1000_2\ 0000_2\ 0000_2\ 0000_2\ 0000_2 \\ &= 41180000_{16} \end{aligned}$$

## Example of Representing a Value in IEEE FPS (cont.)

- Determine the hexadecimal IEEE FPS pattern that represents the decimal value -6.25.
- IEEE FPS pattern: $(-1)^S * (1 + 0.F) * 2^{(E-127)}$

$$\begin{aligned}
-6.25_{10} &= -110.01_2 \\
&= -1.1001_2 * 2^2 \\
&= (-1)^{-1} * (1 + 0.1001_2) * 2^{(129-127)} \\
&= (-1)^S * (1 + 0.F) * 2^{(E-127)}
\end{aligned}$$

$$\begin{aligned}
FPS &= S \ E \ F \\
&= 1 \ 129_{10} \ 10010000000000000000000_2 \\
&= 1 \ 10000001_2 \ 10010000000000000000000_2 \\
&= 1100_2 \ 0000_2 \ 1100_2 \ 1000_2 \ 0000_2 \ 0000_2 \ 0000_2 \ 0000_2 \\
&= c0c80000_{16}
\end{aligned}$$

## Example of Determining an IEEE FPS Pattern Value

- Determine what decimal value that $0xc0900000_{16}$ represents in the IEEE FPS.

$$
\begin{aligned}
FPS &= c0900000_{16} \\
&= 1100_2\ 0000_2\ 1001_2\ 0000_2\ 0000_2\ 0000_2\ 0000_2\ 0000_2 \\
&= 1\ 10000001_2\ 00100000000000000000000_2 \\
&= S\ \ E\ \ F
\end{aligned}
$$

$$
\begin{aligned}
value &= (-1)^S * (1 + 0.F) * 2^{(E-127)} \\
&= (-1)^1 * (1 + 0.001_2) * 2^{(129-127)} \\
&= (-1) * (1.001_2) * 2^2 \\
&= -100.1_2 \\
&= -4.5_{10}
\end{aligned}
$$

## Example of Determining an IEEE FPS Pattern Value (cont.)

- Determine what decimal value that $0x40580000_{16}$ represents in the IEEE FPS.

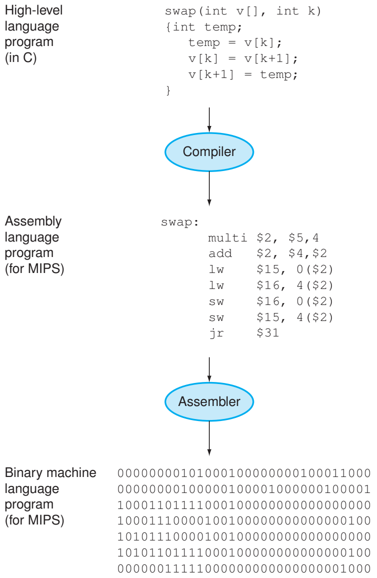
$$\begin{aligned}
FPS &= 40580000_{16} \\
&= 0100_2\ 0000_2\ 0101_2\ 1000_2\ 0000_2\ 0000_2\ 0000_2\ 0000_2 \\
&= 0\ 10000000_2\ 10110000000000000000000_2 \\
&= S\ \ E\ \ F
\end{aligned}$$

$$\begin{aligned}
value &= (-1)^S * (1 + 0.F) * 2^{(E-127)} \\
&= (-1)^0 * (1 + 0.1011_2) * 2^{(128-127)} \\
&= 1 * (1.1011_2) * 2^1 \\
&= 11.011_2 \\
&= 3.375_{10}
\end{aligned}$$

## Instructions for a Machine

- A high-level language statement is typically represented by several assembly instructions.
- An assembly instruction is generally a symbolic representation of a 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
```

## 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 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 |

## 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. |
| `.extern` *sym size* | Declare that the datum stored at *sym* is *size* bytes and is a global label. |
| `.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 at the current location in the current segment. |
| `.text` | Switch to the text segment. |
| `.word` *w1,...,wn* | Place the *n* word values in memory. |

## 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<br>\$a1 = max length | |
| exit | \$v0 = 10 | | |
| print_char | \$v0 = 11 | \$a0 = char | |
| read_char | \$v0 = 12 | | char in \$v0 |

## 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)

## Logical operations: 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 $2^i$.
- A logical left (right) shift fills in the vacant bits with zero.
- An arithmetic right shift replicates the most signficiant bit to fill in the vacant bits.

| 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) |

## Exercise: Using Shift Instructions

- Write a single MIPS assembly instruction to multiply \$t2 by 4 ($2^2$) and put the result in \$t3.
- Answer: sll \$t3 \$t2 2.

- Assume \$t1 has the value 2. What are the values assigned to \$t2 if we perform `sll $t2,$t1,3`?
- Answer: shift left by 3 bits - > multiply value by $2^3$. The result is $2 \times 2^3 = 16$.

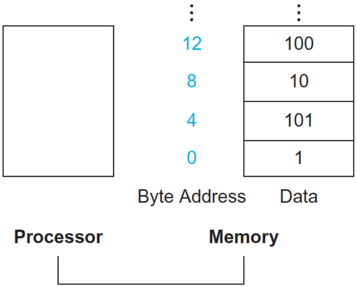
- Assume \$t2 has the value -4. What are the values assigned to \$t3 and \$t4 if we perform the following instructions?
  ```
  sra $t3,$t2,2
  srl $t4,$t2,2
  ```
- Answer: $(-4)_{10} = (11111111111111111111111111111100)_2$
  \$t3: $(11111111111111111111111111111111)_2 = (-1)_{10}$
  \$t4: $(00111111111111111111111111111111)_2 = (2^{31} - 1)_{10}$

## Data Transfer Instructions

- The processors keep only a small amount of data in registers, but memory contains billions of data elements
- Data transfer instructions are used to transfer data between register and memory
- The MIPS can only access memory with load and store instructions.

## Specifying Memory Address

- Memory is organized as an array of bytes (8 bits)
- In memory, each element is kept as a word (4 bytes), which must start at address that are multiples of 4.



- How to get the address of a element in the array?
- Base address + offset (multiples of 4)

## General Form of MIPS Data Transfer Instructions

- form: <operation>   <reg1>,<constant>(<reg2>)
- reg2: keep base address; constant: keep offset
- load instruction: copy data from memory to a register
- store instruction: copy data from a register to memory

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

- ##: contatenation; $=_{\#}$: # are assigned; $0^{\#}$: # bits of zero; #1..#2: bit range where the LSB is labeled bit 31.

## Indexing Array Elements with a Constant Index

- Use MIPS assembly directives to declare a four element integer array named A.
- Write MIPS assembly instructions to accomplish the following C statement. Assume $t5, $t6, and $t7 are available.

```
A[3] = A[0] + A[1] + A[2];
```

```
.data
_A:  .word 1,2,3,0       # declare space for array A
.text
...
la    $t5,_A          # load address of A
lw    $t6,0($t5)      # load A[0]
lw    $t7,4($t5)      # load A[1]
addu  $t6,$t6,$t7   # add A[0] and A[1]
lw    $t7,8($t5)      # load A[2]
addu  $t6,$t6,$t7   # add A[2]
sw    $t6,12($t5)     # store into A[3]
```

## 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
```

## 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 |

- For beq and bne instructions, nothing happens if the condition is not true.

## 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:
```

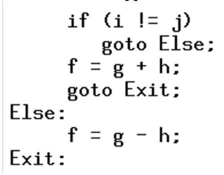
- Note that: the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if*.

## Example of Translating an If-Then-Else Statement

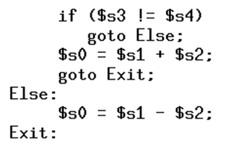
- example source statement:

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

- Translate into MIPS instructions assuming `f`, `g`, `h`, `i`, and `j` are in registers $s0 through $s4 respectively.

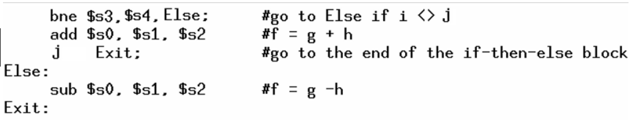


```
     if (i != j)
        goto Else;
     f = g + h;
     goto Exit;
Else:
     f = g - h;
Exit:
```



```
     if ($s3 != $s4)
        goto Else;
     $s0 = $s1 + $s2;
     goto Exit;
Else:
     $s0 = $s1 - $s2;
Exit:
```

$\boxed{Finalresults:}$



```
     bne $s3,$s4,Else;       #go to Else if i <> j
     add $s0, $s1, $s2       #f = g + h
     j    Exit;              #go to the end of the if-then-else block
Else:
     sub $s0, $s1, $s2       #f = g -h
Exit:
```

## 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:
```

- How about the translation for the following statement?

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

## Translating an If-Statement with a Different Condition

- example source statement:

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

- Key point: $a >= b$ is the same as $!(a < b)$.
- 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
bne  $t5,$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
```

## Optimizing the Translation of the For Statement

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

- How can we make the code on the previous two slides more efficient?
  - We don't need to test the exit condition the first time.
  - We can step through the array since each element is 4 bytes after the last element.
  - We can efficiently check if we have stepped past the last element to be processed.

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

➡

```
      sum = 0;
      p = &a;
      exit = &a[100];
loop: sum += *p;
      p++;
      if (p != exit) goto loop;
```

## Optimizing the Translation of the For Statement (cont.)

- Again assume `sum` and the starting address of `a` are in `$t2` and `$t4`, respectively. Also assume that `$t3` and `$t5` are available.
- We can reduce the body of the loop from 7 instructions to 4 instructions.

```
and   $t2,$t2,$zero     # sum = 0
move  $t3,$t4           # p = &a
addiu $t5,$t3,400       # exit = &a[100]
loop:
lw    $t6,0($t3)        # load a[i] into tmp
addu  $t2,$t2,$t6       # sum += tmp
addiu $t3,$t3,4         # p++
bne   $t3,$t5,loop      # if (p != exit) goto loop
```

## Steps for Executing a Function

- caller actions for the function call
  - Place arguments where the callee can access them: $a0 - $a3
  - Place return address where the callee can access it.
  - Transfer control to the callee: jal LI (Li is the label of the callee)
- callee actions when entering function
  - Allocating storage needed for the callee: $s0 - $s7, $t0 - $t9
  - Preserve values of callee-save registers used in the function: $s0 - $s7
    - addi $sp, $sp, -4
    - sw $0, 0($sp)
- callee performs the task associated with the function

## Steps for Executing a Function (cont.)

- callee actions when exiting function:
  - Assign the result value in a place where caller can access it: $v0- $v1,
  - Restore the values of callee-save registers used in the function.
    - lw $s0, 0($sp)
    - addi $sp, $sp, 4
  - Deallocate storage needed for the callee.
  - Transfers control back to the point after the call: jr $ra
- caller accesses the result value

## 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 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) and there were more available opcode values in the `funct` field than the op field.

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

## R Format Instruction Encoding Examples

- R-format example 1: `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 | | | | | |

- R-format example 2: `sll $t5,$t6,7`

| fields | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| decimal | 0 | 0 | 14 | 13 | 7 | 0 |
| binary | 000000 | 00000 | 01110 | 01101 | 00111 | 000000 |
| hexadecimal | 0x000e69c0 | | | | | |

## 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 sign extended if it is an arithmetic operation. It is zero extended if it is a logical operation.

| 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 | | | |

## Branch Example of MIPS I Format (cont.)

| address | instruction |
|---|---|
| 40000008 | addi $5, $5, 1 |
| 4000000C | beq $0, $5, label |
| 40000010 | addi $5, $5, 1 |
| 40000014 | addi $5, $5, 1 |
| 40000018 | label addi $5, $5, 1 |
| 4000001C | addi $5, $5, 1 |
| 40000020 | etc... |

- Binary code to beq $0,$5, label is 0x10050002, which means 2 instructions from the next instruction.
- Before executing beq: PC = 0x4000000C
- After executing beq: PC+4 = 0x40000010
- Relative address 4*2 = 0x00000008
- Effective Address = 0x40000018

| op | rs | rt | Immediate value |
|---|---|---|---|
| 00010 | 00000 | 00101 | 0000000000000010 |

## MIPS J Format

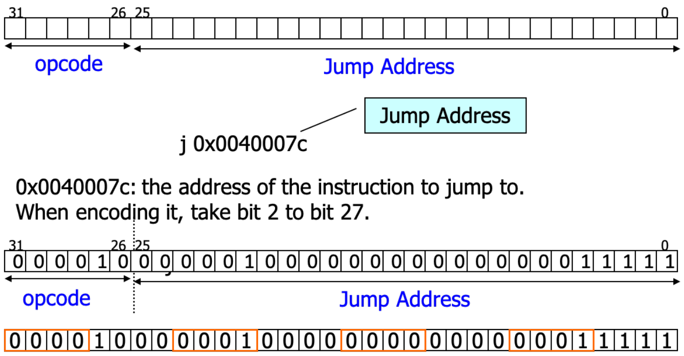
- The MIPS J format is used for unconditional jumps and function calls.
- The op field is used to identify the type of instruction.
- The `targaddr` field is used to indicate an absolute target instruction address divided by 4.
    - An absolute target instruction address divided by 4 means shift its binary representations 2 bits to the right.

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

op – instruction opcode    targaddr – jump/call target

## Example of MIPS J Format

- j:



Encoding = 0x0810001f