Machine Language Instructions

Introduction

- **Instructions** – Words of a language understood by machine
- **Instruction set** – Vocabulary of the machine
- Current goal: to relate a high level language to instruction set of a machine
- Machine language vs. human language (restricted programming language)
  - Most machines speak similar languages, or dialect of the same language
  - Similar operations in a formal language (arithmetic, comparison)
  - Close to like learning to drive one car
- **MIPS instruction set**
  - Used by NEC, Nintendo, Silicon Graphics, and Sony
  - Designed since the early 1980s

Hardware operations

- **Arithmetic**
  - MIPS assembly language instruction
    
    \[
    \text{add a, b, c}
    \]
    
    Equivalent C statement: \(a = b + c\);
  - Too rigid in nature; to perform \(a = b + c + d + e\); you have to do the following:
    
    \[
    \begin{align*}
    \text{add a, b, c} & \quad \# a = b + c \\
    \text{add a, a, d} & \quad \# a = b + c + d \\
    \text{add a, a, e} & \quad \# a = b + c + d + e
    \end{align*}
    \]
  - Each line can contain only one instruction, and the instruction does not achieve much more than a primitive operation
  - Comments terminate at the end of line
  - Same number of operands for each instruction (three above)
    
    * Keeping number of operands constant keeps hardware simple

**Design Principle 1** *Simplicity favors regularity.*

- Compiling a simple C program into MIPS assembly
  
<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = b + c;)</td>
<td>\text{add a, b, c}</td>
</tr>
<tr>
<td>(d = a - e;)</td>
<td>\text{sub d, a, e}</td>
</tr>
</tbody>
</table>

- Compiling a slightly complex C program into MIPS assembly
  
<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f = (g + h) - (i + j))</td>
<td>\text{add t0, g, h}</td>
</tr>
<tr>
<td></td>
<td>\text{add t1, i, j}</td>
</tr>
<tr>
<td></td>
<td>\text{sub f, t0, t1}</td>
</tr>
</tbody>
</table>

Somehow, the compiler needs to know to perform both additions before performing subtraction, and to use temporary variables
Hardware operands

- **Registers**
  - Required operands of arithmetic instructions
  - Replace memory variables
  - 32-bits per register in the MIPS ISA
  - Since 32-bits occur frequently, and is the size of each operand, it is given the name *word*
  - Only a limited number available in a machine
    * Typically 32 in the current machine, including the MIPS ISA
    * MIPS registers are numbered from 0 to 31
    * Contrast this with number of variables in programming languages, or a typical program
    * The three operands in the MIPS arithmetic instruction must be chosen from these 32 registers
  - The reason for a small number of registers is

  **Design Principle 2  Smaller is faster.**
  A large number of registers will slow down the signal, increasing clock cycle time

- **Convention**
  * MIPS registers corresponding to C variables will be called *$s0, $s1, ...*
  * MIPS registers corresponding to temporary variables will be called *$t0, $t1, ...*

- Compiling a C program using registers
  Assume variables *f, g, h, i, and j* correspond to registers *$s0, ..., $s4*, respectively

<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f = ( g + h ) - ( i + j ) )</td>
<td>add  $t0, $s1, $s2</td>
</tr>
<tr>
<td></td>
<td>add  $t1, $s3, $s4</td>
</tr>
<tr>
<td></td>
<td>sub  $s0, $t0, $t1</td>
</tr>
</tbody>
</table>

- **Data transfer instructions**
  - Data structures in programming languages can contain more than one variable (arrays and structures)
    * Complex data structures are kept in memory and brought into registers as needed
  - Memory words are accessed by an *address*
  - Memory can be treated as a 1D array, with the address providing an index into the array, starting at 0
  - Instruction load
    * Used to transfer data from memory to register (load data into register)
    * Format is name of instruction followed by the register to be loaded, then a constant, and then, a register containing memory address
    * Memory address in second register is called *base address* or starting address for the array, and the constant is called an *offset* with respect to base address
    * Actual memory address is formed by taking the sum of offset and base address
    * MIPS name is *lw* for *load word*

- Compiling a C program with data in memory

<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g = h + A[8] )</td>
<td>lw $t0, 8($s3)</td>
</tr>
<tr>
<td></td>
<td>add $s1, $s2, $t0</td>
</tr>
</tbody>
</table>

- Instruction store
  * Complementary to load
Machine Language Instructions

- Transfer data from register to memory
- MIPS uses the instruction `sw` (store word)
- Uses the same format as `lw`

**Hardware/software interface**
- Compiler associates variables with registers, and allocates complex data structures to locations in memory
- Most architectures address individual *bytes*, and address of a word matches 4 bytes
- Addresses of sequential words differ by 4
- **Alignment restriction**
  - MIPS words must always start at an address that is a multiple of 4
  - Some machines may use the address of leftmost byte as the word address; called *big endian*
    - MIPS, PowerPC, and SPARC are big endian
  - Data is laid over as
    - Other machines may use the address of rightmost byte as the word address; called *little endian*
    - Pentium is little endian
    - Data is laid over as
    - Addresses are still in terms of bytes
    - Addressing of `A[8]`
      - Offset to be added should be `8 * 4 = 32`
- Compiling C program with load and store (fixing the bug of byte offset)

<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>add $t0, $s2, $t0</code></td>
</tr>
<tr>
<td></td>
<td><code>sw $t0, 48($s3)</code></td>
</tr>
</tbody>
</table>

- Compiling C program with variable array index

<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g = h + A[i]</code></td>
<td><code>add $t1, $s4, $s4        # $t1 = 2 * i</code></td>
</tr>
<tr>
<td></td>
<td><code>add $t1, $t1, $t1        # $t1 = 4 * i</code></td>
</tr>
<tr>
<td></td>
<td><code>add $t1, $t1, $s3        # Address of A</code></td>
</tr>
<tr>
<td></td>
<td><code>lw $t0, 0($t1)</code></td>
</tr>
<tr>
<td></td>
<td><code>add $s1, $s2, $t0</code></td>
</tr>
</tbody>
</table>

- Optimizations
  - Number of variables may be far more than the number of registers
  - Keep frequently used variables in registers
  - **Spilling registers**
    - Using load/store combinations to bring the less frequently used variables into memory and then, putting them back
- **Index register**
  - Register to hold the base address in memory

**Representing instructions**

- Represented as binary numbers inside the computer (*opcode*)
• Registers are part of every instruction
  – Registers $s0 to $s7 map onto registers 16 to 23
  – Registers $t0 to $t7 map onto registers 8 to 15

• Translating a MIPS assembly instruction into machine language
  – Consider the instruction `add $t0, $s1, $s2`
  – Decimal representation

      0 17 18 8 0 32
      + $s1 $s2 $t0 unused +

Six fields, with field 0 and 5 indicating the operation to be performed, field 4 unused, and other fields indicating the registers to be used
  – Binary representation in 32-bits

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

* Layout of instruction is called *instruction format*
  * All MIPS instructions are 32-bit long
  * Numeric version is called *machine instruction*
  * Sequence of machine instructions is called *machine code*

• MIPS fields – Six fields are identified as
  – op Basic operation, or opcode, described in 6 bits
  – rs First register source operand
  – rt Second register source operand
  – rd Register destination
  – shamt Shift amount, used in shift instructions
  – funct Function; specific variant of operation in op field, also called *function code*

  – Problem
  * `lw` specifies two registers and a constant
  * Limiting constant to 5 bits limits the offset to just 32; too small to be useful for accessing large arrays or structures
  * Conflict to keep all instructions the same length and desire for a single instruction format

**Design Principle 3** *Good design demands good compromises.*

* MIPS compromise is to have the same length for all instructions but different format
  – R-type format (register type)
  – I-type format (data transfer instructions)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

* 16-bit address allows access to an address in a range of $\pm 2^{13}$ from the base address ($\pm 32768$ bytes, $\pm 2^{13}$ or $\pm 8192$ words), base address is contained in register rs
* Consider the instruction: `lw $t0, 32($s3) # $t0 = A[8]`

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>10011</td>
<td>01000</td>
<td>0000000000010000</td>
</tr>
<tr>
<td>0x23</td>
<td>0x13</td>
<td>0x08</td>
<td>0x0020</td>
</tr>
</tbody>
</table>
The \texttt{rt} field specifies the destination for this instruction, to receive the result of load (it was second source in \textit{R}-type instruction)

- Complexity is reduced by keeping the instruction format similar
- First three fields in both \textit{R}-type and \textit{I}-type instructions are the same in length and name
- Formats are distinguished by the \texttt{op} field
- The \texttt{funct} field is recognized based on the bit pattern in the \texttt{op} field

Example: Assume that base of A is in \$t1 and h corresponds to \$s2

<table>
<thead>
<tr>
<th>C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lw $t0, 1200($t1)} # $t0 = A[300]</td>
<td>\texttt{0x23 0x09 0x08 0x04B0}</td>
</tr>
<tr>
<td>\texttt{add $t0, $s2, $t0} # $t0 = h + A[300]</td>
<td>\texttt{0x00 0x12 0x08 0x00 0x20}</td>
</tr>
<tr>
<td>\texttt{sw $t0, 1200($t1)} # A[300] = h + A[300]</td>
<td>\texttt{0x2B 0x09 0x08 0x04B0}</td>
</tr>
</tbody>
</table>

Equivalent machine code is:

- For \textit{I}-type instructions, base register is specified in second field (\texttt{rs}), destination (or source) is specified in third field (\texttt{rt}), and the offset in final field
- For \textit{R}-type instructions, we need \texttt{funct} in sixth field, two source operands in second and third fields, and destination in fourth field
- The \texttt{op} for \texttt{lw} and \texttt{sw} differs in just one bit, with no difference in the rest of the fields

\textit{Stored program concept}
- Both instructions and data are kept in memory as bit patterns (or binary numbers)
  - MIPS has 32 general purpose registers, each of length 32 bits
  - MIPS can address $2^{32}$ bytes ($2^{30}$ words) of memory

Making decisions

- Conditional branches
  - Decision making is implemented in a high level language by using an \texttt{if} statement
  - Decision making in MIPS assembly language
    - Start with two instructions
      \begin{Verbatim}
      beq register1, register2, label \quad \texttt{if ( register1 == register2 ) goto label}
      bne register1, register2, label \quad \texttt{if ( register1 \neq register2 ) goto label}
      \end{Verbatim}
    - Mnemonics are equivalent to \textit{branch if equal} and \textit{branch if not equal}, and are known as \textit{conditional branches}
  - Compiling C code into MIPS assembly; assume g, h, i, j, k correspond to registers \$s0 through \$s4
    \begin{Verbatim}
    if ( i == j ) go to L1;
    \quad beq \$s2, \$s3, L1
    f = g + h;
    \quad add \$s0, \$s1, \$s2
    L1: f = f - i;
    \quad \texttt{sub} \$s0, \$s0, \$s2
    \end{Verbatim}
  - Label L1 corresponds to the address of the subtract instruction
  - Modern programming languages almost never have any goto-statement, or discourage their use (see \url{http://www.acm.org/classics/oct95/})
  - Compiling if-then-else into conditional branches
* Use an unconditional branch or jump, specified by \( j \)

<table>
<thead>
<tr>
<th>C Program</th>
<th>Modified C Program</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( i == j )</td>
<td>if ( i != j ) go to ELSE;</td>
<td>bne $s3, $s4, ELSE</td>
</tr>
<tr>
<td>( f = g + h; )</td>
<td>( f = g + h; )</td>
<td>add $s0, $s1, $s2</td>
</tr>
<tr>
<td>else</td>
<td>go to EXIT;</td>
<td>j EXIT</td>
</tr>
<tr>
<td>( f = g - h; )</td>
<td>ELSE: ( f = g - h; )</td>
<td>ELSE: sub $s0, $s1, $s2</td>
</tr>
<tr>
<td>EXIT:</td>
<td></td>
<td>EXIT:</td>
</tr>
</tbody>
</table>

− Conditional branch instructions are \( I \)-type instructions, and instruction such as \( \text{bne } s1, s2, 100 \) translates as follows:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>0x04</td>
<td>0x11</td>
<td>0x12 0x0019</td>
</tr>
<tr>
<td>beq</td>
<td>0x05</td>
<td>0x11</td>
<td>0x12 0x0019</td>
</tr>
</tbody>
</table>

The address is specified in terms of word address

• Loops

− Used for iteration, can be implemented with conditional branch and jump

− A simple loop in C
d o
g += A[i];
i += j;
while ( i != h );
− Equivalent loop using conditional branch

LOOP:
g = g + A[i];
i = i + j;
if ( i != h ) goto LOOP;
− Making the same code in MIPS assembly

# Load A[i] into temporary register

LOOP:  add $t1, $s3, $s3  # $t1 = 2 * i
       add $t1, $t1, $t1  # $t1 = 4 * i
       add $t1, $t1, $s5  # $t1 = A + 4 * i -- The address of A[i]
       lw $t0, 0($t1)  # $t0 = A[i]
       add $s1, $s1, $t0  # g = g + A[i]
       add $s3, $s3, $s4  # i = i + j

       bne $s3, $s2, LOOP  # if ( i != h ) goto LOOP

− Basic block

* A sequence of instructions without branches, or branches only at the end, and without labels, or labels only at the beginning
* No entry points in the middle of code
* No exit points in the middle of code

− Compiling a while loop

* A traditional C loop
while ( save[i] == k )
i = i + j;

* Modified C loop
LOOP: if ( save[i] != k )
go to EXIT;
i = i + j;
go to LOOP;
EXIT:

* MIPS assembly code
  # Load save[i] into temporary register
LOOP: add $t1, $s3, $s3 # $t1 = 2 * i
   add $t1, $t1, $t1 # $t1 = 4 * i
   add $t1, $t1, $s6 # $t1 = save + 4 * i -- &save[i]
lw $t0, 0($t1) # $t0 = save[i]
bne $t0, $s5, EXIT # if ( save[i] != k ) go to EXIT
   add $s3, $s3, $s4 # i = i + j
   j LOOP # go to LOOP
EXIT:

– Comparing two numbers to find out which is larger

* Achieved by the instruction slt, (set on less than)
* The instruction
  slt $t0, $s1, $s2
  sets register $t0 to 1 if $s1 is less than $s2; otherwise, it is reset (or set to 0)
* slt is R-type instruction, and instruction such as slt $s1, $s2, $s3 translates as follows:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>0x00</td>
<td>0x12</td>
<td>0x13</td>
<td>0x17</td>
<td>0x00</td>
</tr>
</tbody>
</table>

– The $zero register
* MIPS compiler uses the register $zero to read the result of all conditional expressions
* Register $zero is mapped to register 0

• Conditions
  – Compiling a less than test
   * The C statement
     if ( a < b ) go to LESS;
   * MIPS assembly code
     slt $t0, $s0, $s1 # $t0 = ( $s0 < $s1 ) ? 1 : 0;
     bne $t0, $zero, LESS # if ( $t0 != 0 ) go to LESS;
  – MIPS ISA does not have an instruction to compare and branch in a single instruction but uses two faster and simpler instructions to achieve the effect
  – slt is an R-type instruction

• Case/switch statement
  – Consider the C code
switch ( k )
{
    case 0:
        f = i + j;  break;
    case 1:
        f = g + h;  break;
    case 2:
        f = g - h;  break;
    case 3:
        f = i - j;  break;
}

- Simplest way to code it is using a sequence of if-then-else
- A more efficient way is to use a jump address table
  * Jump address table is simply a vector of addresses corresponding to labels in the code
  - MIPS has a jump register instruction (jr) to provide an unconditional jump to the address specified in the register
  - MIPS assembly code for the C code given above
    * Variables f through k correspond to registers $s0 through $s5
    * Register $t2 contains constant 4
    * Register $t4 contains the address of vector table (jumptable)
    * An unconditional jump is provided by the instruction j
      <table>
        | op  | rs  | rt  | rd  | shamt | funct |
        |-----|-----|-----|-----|-------|-------|
        | jr  | 0x00| 0x09| 0x00| 0x00  | 0x08  |
      </table>
      slt  $t3, $s5, $zero  # k < 0 ?
      bne  $t3, $zero, EXIT
      slt  $t3, $s5, $t2    # k < 4 ?
      beq  $t3, $zero, EXIT
      # At this point, we know that 0 <= k < 4
      add  $t1, $s5, $s5     # k *= 2
      add  $t1, $t1, $t1     # k *= 2
      add  $t1, $t1, $t4     # $t1 = jumptable + k
      lw   $t0, 0($t1)       # $t0 = jumptable[k]
      jr   $t0               # Jump to jumptable[k]
      L0:  add  $s0, $s3, $s4 # f = i + j
          j       EXIT
      L1:  add  $s0, $s1, $s2 # f = g + h
          j       EXIT
      L2:  sub  $s0, $s1, $s2 # f = g - h
          j       EXIT
      L3:  sub  $s0, $s3, $s4 # f = i - j
      EXIT:
    * jr is an R-type instruction, and instruction such as jr $t1 translates as follows:
Procedures

- Legacy name for function
- Performs a task given to it and returns control to the point where the task was originally assigned
  - Place parameters in a place accessible to procedure
  - Transfer control to procedure
  - Acquire storage resources needed for procedure
  - Perform the task
  - Place the result in a place accessible to calling module
  - Return control to point of origin

- More register allocation
  - $a0 – $a3: Argument registers to pass parameters
  - $v0 – $v1: Value registers to return values
  - $ra: Return address register to return to point of origin

- **jump and link** instruction
  - jal  proc_addr
  - Jumps to proc_addr and saves the address of the next instruction (after jal) in register $ra (return address)
  - Return address ensures that the return is to the place from where the procedure was called as it could be called from multiple locations

- **Program counter**
  - Register to hold the address of current instruction being executed
  - Abbreviated to pc in MIPS
  - The address saved in $ra by jal is pc + 4
  - Return jump is now easily accomplished by jr $ra

- Execution of procedures
  - Caller puts the parameter values in registers $a0–$a3
  - Issues the instruction jal foo to jump to procedure foo
  - foo does the work, and puts the result in registers $v0–$v1
  - foo returns control to caller by issuing jr $ra

- Using more registers
  - Four arguments and two return value registers
  - All registers must be restored to their original values after return from procedure
  - *Spill* registers using a stack in memory
  - Achieved by a special register called stack pointer or sp
    - Stacks grow from higher address to lower address

- Example of procedure call
int foobar ( const int g, const int h, const int i, const int j )
{
    int f = ( g + h ) - ( i + j );
    return ( f );
}

• In assembly language:

foobar:
    sub $sp, $sp, 12            # Make room for three words
    sw $t1, 8($sp)             # Save registers $s0, $t0, and $t1
    sw $t0, 4($sp)
    sw $s0, 0($sp)

# Variables g, h, i, j correspond to $a0, $a1, $a2, $a3

    add $t0, $a0, $a1          # ( g + h )
    add $t1, $a2, $a3          # ( i + j )
    sub $v0, $t0, $t1          # ( g + h ) - ( i + j ); into return value

    lw $s0, 0($sp)             # Restore registers
    lw $t0, 4($sp)
    lw $t1, 8($sp)

    add $sp, $sp, 12           # Adjust stack pointer
    jr $ra                     # Return to caller

• Convention to reduce register spilling
  – Saved registers ($s0 – $s7) must be saved by the called procedure and restored before return
  – Temporary registers ($t0 – $t9) need not be preserved by the called procedure
  – The convention implies that we could have reduced the saving and restoration steps for temporary registers
    in the above code

• Nested procedures
  – Procedures that do not call other procedures are known as leaf procedures
  – All the registers, including $ra and temporaries may need to be preserved, using the stack
  – Recursive procedures may make it even more tough
  – Compiling a recursive procedure to compute factorial

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

– Parameter n is the argument or $a0
– Assembly code
Machine Language Instructions

```assembly
fact: sub $sp, $sp, 8 # Make room for 2 items on stack
sw $ra, 4( $sp ) # Return address
sw $a0, 0( $sp ) # Input argument, saved for multiplication

slt $t0, $a0, 1 # n < 1?
beq $t0, $zero, L1 # Prepare to call function again

add $v0, $zero, 1 # Return 1
add $sp, $sp, 8 # Pop two items; you don’t restore them (why?)

jr $ra # Return to caller

L1: sub $a0, $a0, 1 # Perform recursion
jal fact

# Return point from recursion
lw $a0, 0 ( $sp ) # Restore n to value before recursive call
lw $ra, 4 ( $sp ) # Return address before recursive call
add $sp, $sp, 8 # Restore stack pointer
mul $v0, $a0, $v0 # return ( n * fact ( n - 1 ) )
jr $ra # Return to caller
```

– Some registers must be preserved while others are not so important
– As a rule of thumb, preserve a register’s value if you need it when you return from procedure call

• Allocating space for new data
  – Stack is also used to store local variables for the procedure
  – This segment of stack, including both local variables and saved registers, is called procedure frame or activation record
  – Frame pointer ($fp)
    * Used to keep track of first word of the frame of a procedure
    * Specified by register 30 (s8) in SPIM
    * Stack pointer may change during the procedure execution (why?)
    * Frame pointer offers a stable base register for local memory references, and is useful when the stack changes during procedure execution
    * Frame pointer is initialized using the address in $sp on a call and $sp is restored using fp
  – jal actually saves the address of the instruction that follows jal into $ra so that return can be accomplished by a simple jr $ra
  – C variables
    * Automatic variables (local scope)
    * Static variables (local scope; non-local lifetime)
    * Global variables
    * Static data is accessed in MIPS by using a register called $gp, or global pointer

Beyond numbers

• Need to access data in bytes (such as characters)
• Load byte and store byte instructions (least significant 8 bits in register)
Machine Language Instructions

```plaintext
lb $t0, 0($sp)     # Read byte from source
sb $t0, 0($gp)    # Store byte in destination
```

- Characters are normally combined into strings, using three choices
  - First byte gives the length of string
  - A separate variable contains the length of string (C++)
  - Last position in string is indicated by end of string mark (C)

- Example: String copy in C
  ```c
  void strcpy ( char * x, char * y )
  {
      while ( *x++ = *y++ );
  }
  ```

- MIPS assembly, with x and y pointers in $a0 and $a1
  ```plaintext
  strcpy:
  add $t1, $a1, $zero  # Copy addresses into temporary
  add $t0, $a0, $zero
  L1:  lb $t2, 0($t1)  
       sb $t2, 0($t0)  
       beq $t2, $zero, L2  # Copied last byte? Yes, go to L2
  add $t0, $t0, 1
  add $t1, $t1, 1
  j  L1
  L2:  jr  $ra
  ```

- Unicode
  - Preferred encoding of characters
  - Useable for more than just plain English characters
  - Used in Java
  - Another reason to use `sizeof(char)` rather than 1 when allocating space in code
  - Word alignment in stack and strings, packing four bytes per word in MIPS

Other styles of addressing in MIPS

- Constant or immediate operands
  - You can keep a constant in an instruction itself, instead of having to load it from memory
  - Same format as branch and data transfer instructions (I-type)
    - *I stands for immediate*
  - Constant can be up to 16 bits
  - Translating assembly constant to machine language with the instruction
    - `addi $sp, $sp, 4`
      ```plaintext
      | op | rs | rt | imm |
      |----|----|----|-----|
      | 6  | 5  | 5  | 16  |
      | 0x08 | 0x29 | 0x29 | 0x0004 |
      ```
  - Comparisons can be tested using `slti`
Design Principle 4 *Make the common case fast.*

- Constants being part of instruction get loaded faster than loading them from memory
- Constants longer than 16 bits can be loaded using the instruction *load upper immediate*, or *lui*
- Consider the following instruction

\[
\text{\texttt{lui }$t0, 0xFF}
\]

The register $t0$ contains: 0x00FF 0000 (the space is added for readability)

- Loading a 32-bit constant 0x003D 0900 into $s0$

\[
\begin{align*}
\text{\texttt{lui }$s0, 0x003D} \quad & \text{addi }$s0, $s0, 0x0900
\end{align*}
\]

- SPIM compiler automatically breaks large constants into smaller pieces to be handled by assembly language instructions
- Assembler also can do this job, using a temporary register $at$
- Also, it is preferable to use instruction *ori* in place of *addi* for copying large constants

- **Addressing in branches and jumps**
  - *J*-type instructions
    - Contain a 6-bit opcode and the rest of the bits for address
    - Translating jump instruction to machine language
  
  \[
  \text{\texttt{j 0x400018}}
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
<tr>
<td>j 0x02</td>
<td>0x400018</td>
</tr>
</tbody>
</table>

- Conditional branch instructions are *I*-type, leaving only 16 bits for the address
- Addresses larger than 16 bits can be accommodated by using a register whose contents will be added to the address specified in branch instruction
- Since most of the branches are close to the current instruction, we can use $pc$ as the register to be added, leading to a range of $\pm 2^{15}$ from the current value in $pc$, leading to PC-relative addressing
  - Distance or range of branch can be stretched by using the fact that all MIPS instructions are 4 bytes long
  - Jump-and-link instructions may go anywhere in the process and hence, they are performed with *J*-type instructions
    - The 26-bit field also uses word addressing, allowing for a jump that is 2^8 bytes
    - The full 32-bit addressing can be achieved by using a jump register instruction
  - Branching far away (done automatically by assembler by inverting the condition)

\[
\text{\texttt{beq }$s0, $s1, L1} \quad \text{gets translated to}
\]

\[
\begin{align*}
\text{\texttt{bne }$s0, $s1, L2} \\
\text{\texttt{j L1}}
\end{align*}
\]

**L2:**

Starting a program

- Editor
• Compiler

• Assembler
  – Pseudoinstructions
  – Object file
    * Machine language instructions
    * Data allocation
    * Information to put the instructions in memory (format)
  – Symbol table
  – Object file format
    * Header
      - Size and position of other pieces of file
    * Text segment
      - Actual machine language code
    * Data segment
      - Static data
      - Dynamic data
    * Relocation information
      - Instructions and data words that depend on absolute addresses
    * Symbol table
      - Undefined symbols such as `extern` variables
    * Debugging information
      - Associating machine instructions with source language statements

• Linker, or linkage editor
  – Putting together independently assembled modules
  – Place code and data modules symbolically in memory
  – Determine the addresses of data and instruction labels
  – Patch both the internal and external references
  – Executable file

• Loader
  – Read executable file header to determine the size of text and data segments
  – Allocate memory to hold text and data
  – Copy text (instructions) and data from executable file into memory
  – Copy parameters for `main()` to stack
  – Initialize machine registers and stack pointer
  – Start the program

Examples on swap and sort

• Reading assignment

Arrays versus pointers
Machine Language Instructions

- Refer to earlier example on strcpy function, comparing with the one in the book
- C compiler treats arrays as pointers

```cpp
#include <iostream>

int main()
{
    using namespace std;

    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    cout << "6[a] = " << 6[a] << endl;

    return ( 0 );
}
```

- The above code works because `a[i]` gets translated internally to `*( a + i )`

**PowerPC and 80x86 ISA**

- Instruction complexity, clock speed, and number of instructions executed by a given program
- IBM/Motorola PowerPC
  - Similar to MIPS in most respects
    * 32 integer registers
    * Each instruction is 32-bits long
    * Data transfer possible only through load and store
  - Two more addressing modes and a few operations
  - Indexed addressing
    * Allows two registers to be added together during instruction execution
    * MIPS code
      ```
      add $t0, $a0, $s3  # $a0 is array base; $s3 is index
      lw  $t1, 0($t0)    # $t1 = $a0[$s3]
      ```
    * PowerPC code
      ```
      lw  $t1, $a0 + $s3  # $t1 = $a0[$s3]
      ```
    - Update addressing
      * Look at our pointer arithmetic version of strcpy
      * Automatically increment the base pointer
      * MIPS code
        ```
        lw  $t0, 4($s3)    # $t0 = *($s3 + 4 )
        addi $s3, $s3, 4   # $s3 += 4
        ```
      * PowerPC code
        ```
        lwu $t0, 4($s3)    # $t0 = *($s3 + 4 ); $s3 += 4
        ```
    - New instructions
      * Load multiple and store multiple
      * Transfer up to 32 words of data in one instruction; useful for copying large data
      * Special counter register to speed up loops