Chapter 2:

Basic Computer Operations and Assembly Language Programming

Topics:

Programmer’s view of computer system
Instruction Execution Cycle
Simple Program Translation with MAL
Programmer’s View of Computer System
System bus:

Input/output (IO) devices:

Central Processing Unit (CPU):

Some CPUs: Sun Sparc, PowerPC, Intel x86
For CSc 310, use MIPS CPU (Silicon Graphics)

Main memory:
Units of storage:

bit
nibble
byte

halfword
word
doubleword

MIPS CPU:
* each memory address is 32 bits

Memory is *byte-addressed*:
* each increment of the address refers to the next byte
Memory viewed as array of bytes:

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td></td>
</tr>
<tr>
<td>0x0000 0001</td>
<td></td>
</tr>
<tr>
<td>0x0000 0002</td>
<td></td>
</tr>
<tr>
<td>0x0000 0003</td>
<td></td>
</tr>
<tr>
<td>0x0000 0004</td>
<td></td>
</tr>
<tr>
<td>0x0000 0005</td>
<td></td>
</tr>
<tr>
<td>0x0000 0006</td>
<td></td>
</tr>
<tr>
<td>0x0000 0007</td>
<td></td>
</tr>
<tr>
<td>0x0000 0008</td>
<td></td>
</tr>
<tr>
<td>0x0000 0009</td>
<td></td>
</tr>
<tr>
<td>0x0000 000a</td>
<td></td>
</tr>
<tr>
<td>0x0000 000b</td>
<td></td>
</tr>
</tbody>
</table>
Memory viewed as array of 32-bit words:

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td></td>
</tr>
<tr>
<td>0x0000 0004</td>
<td></td>
</tr>
<tr>
<td>0x0000 0008</td>
<td></td>
</tr>
<tr>
<td>0x0000 000c</td>
<td></td>
</tr>
<tr>
<td>0x0000 0010</td>
<td></td>
</tr>
<tr>
<td>0xffff fffc</td>
<td></td>
</tr>
</tbody>
</table>

Memory byte-order:

MIPS memory is *big-endian*:
The byte with the lowest address is at the most significant end (left-most, end with highest value)

[most commercial CPUs are big-endian, except Intel x86; little-endian:]

page 2.6
Contents of a byte at address 0x0000 0004:

Contents of a byte at address 0x0000 0006:

Contents of a word at address 0x0000 0004 (contents of 4 bytes at 0x0000 0004 to 0x0000 0007):

Aligned versus misaligned:

If a word in memory is aligned, the 4 bytes within the word do not cross word boundaries (words at 0, 4, 8, 0xc, 0x10, 0x14...)

If a word is misaligned, its 4 bytes cross word boundaries. (words at 1, 2, 3, 5, 6, 7 ...)

How to tell if a word address is aligned?
1) look at least significant (rightmost) digit

2) look at least significant 2 bits
Size of memory address (32 bits in MIPS, Pentiums) determines the amount of memory that can be addressed by programs running on a CPU.

32-bit address means a program can “see”

1 KB =
1 MB =
1 GB =

However, this does not mean that your PC has the same amount of physical memory!
Parts of CPU

Control unit:

Arithmetic Logic Unit (ALU):

Registers:

Program counter:
MIPS has 32 registers, each a 32-bit word

Two registers with restricted use:

$0

$1

$30

$31
Registers have alternate names:

$2 - $3 also called $v0 - $v1
$4 - $7 also called $a0 - $a3
$8 - $15 also called $t0 - $t7
$16 - $23 also called $s0 - $s7
$24 - $25 also called $t8 - $t9
$26 - $27 also called $k0 - $k1
$28 also called $gp
$29 also called $sp
$30 also called $s8
$31 also called $ra

For now, use $s0 - $s8 (always safe), or $t0 - $t9.

More on these later, when we talk about function calls.
High Level Language Program:

statement 1
statement 2
statement 3

Assembly Language Program:

instruction 1
instruction 2
instruction 3

Von Neumann (or Eckert/Mauchly) architecture:

1) programs and data are stored in the same memory
2) instructions are executed in sequential order, except for branches/jumps
What happens when a user runs a program?

1) user types `a.out`
2) CPU “finds” a.out file
3) CPU loads memory with program code and data from file
Typical assembly language format:

operation operand1 operand2 operand3 ...

Operation: what the instruction does (+, -)
Operands: variables

Variables are allocated to locations in memory, or to registers

For example ($s0-$s3 contain int variables):

add $s0,$s1,$s2 means $s0 = $s1 + $s2
sub $s3,$s1,$s2 means $s3 = $s1 - $s2

For our simple example, assume:

each instruction is 32 bits
the program starts at PC = 0x400020

(These are typical addresses for MIPS/SPIM)
CPU goes through 5 main steps to process each instruction.

Instruction Fetch (IF):

Instruction Decode (ID):

Execute (EX):

Memory Access (MEM):

Write Result (WR):
Levels of Assembly Language in MIPS

MAL (MIPS assembly language)

TAL (True assembly language)
MIPS (like PowerPC, Sun Sparc, Intel IA64) is a *load/store* architecture. This means:

Given a program, where to allocate its variables?

**Option 1: in memory**
- Advantage: more general
- Disadvantages: slower access, more code

**Option 2: in registers**
- Advantages: fast access, less code
- Disadvantages:
  a) may not have enough registers for all vars
  b) must keep track of where each var is

A good compiler will try to allocate frequently used variables to registers.
First MAL Program (variables in memory)

int a=0,b=1,c=2,d=3,e=4;

int main(void)
{
    b = e;
    a = b + c;
    d = b - c;
    e = e - 1;
}

To move an int x from memory to register $s0:
[load word]

To move an int x from register $s0 to memory:
[store word]

More steps for each statement:
1) move variables from memory into registers
2) perform computation
3) move result from register into memory
[use registers as *temporaries*]
load word (from memory to register):

lw R, label
[R is any register]

contents of R
= contents of aligned word at label (in memory)

Example: lw $13, x

\[
\begin{array}{c}
x: \begin{array}{c}
0x10010000 \\
0x10010004 \\
13
\end{array}
\end{array}
\begin{array}{c}
0x1234 abcd
\end{array}
\]
store word (from register to memory):

sw R, label
[R is any register]

contents of aligned word at label (in memory)
= contents of R

Example: sw $13, x
.data
a:     .word      0

.text
main:
[see Example 2.1]
Labels:
* mark locations in memory
* each label associated with a memory address

Data allocation section:
* like variable declaration section
* global variables only
* allocates some size of memory
* no (actual) type (!)

Assembler directives:
(directions for assembler; not program code)

.data

label: .word ??

label: .word ??:N

label: .byte ??:N

label: .float ??:N
.text

label: .space N

label: .asciiz “abc2”

label: .ascii “xyz”
Same MAL Program (variables in registers)

```c
int a=0,b=1,c=2,d=3,e=4;

int main(void)
{
    b = e;
    a = b + c;
    d = b - c;
    e = e - 1;
}

Compiler decides:

    a     $s0
    b     $s1
    c     $s2
    d     $s3
    e     $s4

[see Example 2.2]
```
.data
[left blank for this example...]
.text
main:
MAL arithmetic/logic instructions

int x,y,z;

Compiler decides:

x  $s0
y  $s1
z  $s2

MAL: li $s0,4 [or any constant]
contents of $s0 = 4
C/C++: x = 4

MAL: move $s0,$s1
contents of $s0 = contents of $s1
C/C++: x = y

MAL: add $s0,$s1,$s2
contents of $s0
= contents of $s1 + contents of $s2
C/C++: x = y+z
MAL: sub $s0,$s1,$s2
contents of $s0
= contents of $s1 - contents of $s2
C/C++: \( x = y-z \)

MAL: mul $s0,$s1,$s2
contents of $s0
= contents of $s1 \ast \) contents of $s2
C/C++: \( x = y \ast z \)

MAL: div $s0,$s1,$s2 [integer division]
contents of $s0
= contents of $s1 / contents of $s2
C/C++: \( x = y/z \)

MAL: rem $s0,$s1,$s2
contents of $s0
= contents of $s1 \% \) contents of $s2
C/C++: \( x = y \% z \)
Compound arithmetic statements

Example:
D = b*b - 4*a*c;

Use temporary variables, break into simple statements.

Compiler decides:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$s0</td>
</tr>
<tr>
<td>b</td>
<td>$s1</td>
</tr>
<tr>
<td>c</td>
<td>$s2</td>
</tr>
<tr>
<td>D</td>
<td>$s3</td>
</tr>
<tr>
<td>temp</td>
<td>$t0</td>
</tr>
<tr>
<td>temp1</td>
<td>$t1</td>
</tr>
</tbody>
</table>
MAL Input/output

To perform input/output (I/O), the syscall instruction is used to call system routines.

Before syscall is executed,
1) $v0 contains a code number to indicate the type of operation (read an int, print an int etc)
2) $a0 contains additional arguments (e.g. integer to be printed)
3) result from user input appear in $v0

C++:
int n=9;

    cout << n;

    cin >> n;
C++:
char str[] = "abc";

cout << str;

MAL:
str: .asciiz "abc"
Selection (if, if-else)

C++:

    if (x > y)
        sum = sum + x;

Suppose x is $s0, y is $s1, sum is $s2.

MAL conditional branches:

if (condition true) goto label;
else execute next instruction

beq op1, op2, label [branch if equal]

if (contents of op1 == contents of op2) goto label
else execute next instruction

op1 is any register
op2 is any register or constant

Example: beq $s0, $s1, label (test $s0 == $s1)
beq $s0,4,label (test $s0 == 4)
Other conditional branches:

\[
\begin{align*}
\text{bne op1,op2, label [test op1 \neq op2]} \\
\text{bgt op1,op2, label [test op1 > op2]} \\
\text{bge op1,op2, label [test op1 \geq op2]} \\
\text{blt op1,op2, label [test op1 < op2]} \\
\text{ble op1,op2, label [test op1 \leq op2]}
\end{align*}
\]

Conditional branches that test zero:

\[
\begin{align*}
\text{beqz op1, label [test op1 \neq 0]} \\
\text{bnez op1, label [test op1 \neq 0]} \\
\text{bgtz op1, label [test op1 > 0]} \\
\text{bgez op1, label [test op1 \geq 0]} \\
\text{bltz op1, label [test op1 < 0]} \\
\text{blez op1, label [test op1 \leq 0]}
\end{align*}
\]
Original:

if (x > y)  
    sum = sum + x;

Rewrite:

if (x <= y) goto skip;
    sum = sum + x;
skip: [next statement]

Compiler decides: x is $s0, y is $s1, sum is $s2

MAL:

ble $s0,$s1, skip
    add $s2, $s2, $s0
skip: [next instruction]

[see Example 2.3]
Same example, variables in memory:

```c
if (x > y)
    sum = sum + x;
```

MAL:

```asm
.data
.x: .word 0
.y: .word 0
.sum: .word 0

.text
main:
[some code not shown...]
```
[see Example 2.4]
Making structured flowcharts:

1) each box is either an arithmetic statement or a test
2) arithmetic statements have only one exit
3) tests have two exits, yes and no
4) boxes should be arranged in a straight line
5) no conditions are in main line of flow
6) yes conditions are side branches

To translate from C++ code to flowchart:

1) write down each statement or test, in order.
2) change test conditions if necessary

Flow chart for if (x>y) sum = sum+x; :
Unconditional branches (goto):

b label [branch; goto label]
j label [jump; goto label]

If-else statements:

Example (suppose num is $s0, flag is $s1)

if (num > 0)
    flag = 1;
else if (num < 0)
    flag = -1;
else flag = 0;
Flow charts:

MAL version (num is $s0, flag is $s1):

[see Example 2.5]
If statements with “and” conditions:

```c
if ((x > y) && (x < 0))
    flag = 23;
```

(Suppose x is $s0, y is $s1, flag is $s2)

Flow chart:

MAL code:
If statements with “or” conditions:

```java
if ((a != 0) || (b < 5))
    grade = -1;
```

(Suppose a is $s0, b is $s1, grade is $s2)
Flow chart:

MAL code:
Repetition: loops

C++:

    sum = 0;
    for (i=1; i<= limit; i++)
        sum = sum + i;

Suppose sum is $s0, i is $s1, limit is $s2. Flow chart and MAL code (version 1):
Flow chart and MAL code (version 2):

Version 2 is more efficient. All your loops should follow version 2, with test condition at the bottom of the loop. [see Example 2.6]
do-while loop:

while loop:
Truth table for simple logical operations

<table>
<thead>
<tr>
<th>y</th>
<th>z</th>
<th>y and z</th>
<th>y or z</th>
<th>y nand z</th>
<th>y nor z</th>
<th>y xor z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bitwise logical operations: logical operation applied to each bit of operand (Suppose x is $s0, y is $s1, z is $s2)

*C/C++: x = ~y (different from x = !y)*
MAL: not $s0,$s1
contents of $s0 = not (contents of $s1)

Example:
int x, y = 0x12345678;

x = ~y;

x becomes:
C/C++: \( x = y \& z \)
MAL: and $$s0,$$s1,$$s2
contents of $$s0$$
= contents of $$s1$$ and contents of $$s2$$

C/C++: \( x = y \mid z \)
MAL: or $$s0,$$s1,$$s2$$
contents of $$s0$$
= contents of $$s1$$ or contents of $$s2$$

C/C++: no nor operator
MAL: nor $$s0,$$s1,$$s2$$
contents of $$s0$$
= contents of $$s1$$ nor contents of $$s2$$

C/C++: \( x = y ^ z \)
MAL: xor $$s0,$$s1,$$s2$$
contents of $$s0$$
= contents of $$s1$$ xor contents of $$s2$$
Shifts and rotates

\textit{C/C++: } \textit{x = y \ll AMT (x in $s0, y in $s1)}
MAL shift left logical: sll $s0, $s1, AMT
contents of \textit{x} = contents of \textit{y} shifted left by AMT bits, “empty” bits filled with zeros

Example:
y = 0x89abcdef // in $s1

After sll $s0,$s1,3
$s0$ becomes:

\textit{C/C++: } \textit{x = y \gg AMT (x,y declared unsigned)}
MAL shift right logical: srl $s0,$s1, AMT
contents of \textit{x} = contents of \textit{y} shifted right by AMT bits, “empty” bits filled with zeros

Example:
y = 0x89abcdef // in $s1
After `srl $s0,$s1,3`
$s0$ becomes:

\[ C/C++: x = y >> AMT \quad (x,y \text{ declared int}) \]
MAL shift right arithmetic: `sra $s0,$s1, AMT`
contents of \(x = \) contents of \(y\) shifted right by AMT bits, “empty” bits filled with sign bit

Example:
y = 0x89abcdef \quad // in \$s1

After `sra $s0,$s1,3`,
$s0$ becomes:
Relationship between shifts and arithmetic operations:

shift left n bits equivalent to

shift right n bits equivalent to

*must watch for overflow!*
C/C++: no rotate operator
MAL rotate left: rol $s0,$s1, AMT
contents of $s0 = contents of $s1 rotated left by AMT bits

Example:
$s1 = 0x89abcdef
After rol $s0,$s1,3
$s0 becomes:

C/C++: no rotate operator
MAL rotate right: ror $s0,$s1, AMT
contents of $s0 = contents of $s1 rotated right by AMT bits
Summary of MAL arithmetic/logic instructions
[op1, op2, op3 are operands, usually registers or constants]

li op1, CONSTANT
move op1, op2
add op1, op2, op3
sub op1, op2, op3
mul op1, op2, op3
div op1, op2, op3
rem op1, op2, op3
not op1, op2
and op1, op2, op3
or op1, op2, op3
nand op1, op2, op3
nor op1, op2, op3
xor op1, op2, op3
sll op1, op2, AMT
srl op1, op2, AMT
sra op1, op2, AMT
rol op1, op2, AMT
ror op1, op2, AMT
Summary of MAL branch instructions
[op1, op2, op3 are operands, usually registers or constants]

b label
j label
beq op1, op2, label
bne op1, op2, label
bgt op1, op2, label
bge op1, op2, label
blt op1, op2, label
ble op1, op2, label
beqz op1, label
bnez op1, label
bgtz op1, label
bgez op1, label
bltz op1, label
blez op1, label

Instruction for requesting system services:

syscall
Some useful Unix commands

Useful Unix commands

- `ls` list files in current directory
- `cat <filename>` display contents of file called `<filename>` on screen
- `more <filename>` like "cat", but page by page
- `cd <dir-name>` change to directory with name `<dir-name>`
- `cp <oldfile> <newfile>` make a copy of `<oldfile>`; the new copy is called `<newfile>`
- `mv <oldfile> <newfile>` change the name of `<oldfile>` to `<newfile>`

Some useful spim commands

- `exit` -- Exit from the simulator
- `read "FILE"` -- Read FILE of assembly code into memory
- `load "FILE"` -- Same as read
- `run <ADDR>` -- Start the program at optional ADDRESS
- `step` -- execute the next TAL instruction in the program
- `step <N>` -- Step the program for N instructions
- `continue` -- Continue program execution without stepping
- `print $N` -- Print register N
- `print $fN` -- Print floating point register N
- `print ADDR` -- Print contents of memory at ADDRESS
- `print PC` -- Print address of current instruction (not yet executed)
- `reinitialize` -- Clear the memory and registers
- `breakpoint <ADDR>` -- Set a breakpoint at address
- `delete <ADDR>` -- Delete all breakpoints at address
- `list` -- List all breakpoints
- `.` -- Rest of line is assembly instruction to put in memory
- `<cr>` -- Newline reexecutes previous command
- `?` -- Print this message

Most commands can be abbreviated to their unique prefix
e.g., `ex`, `re`, `l`, `ru`, `s`, `p`
Programming Style Guidelines

Your projects will be graded on both correctness and clarity. Since you are not the only person who has to read your programs, you must comment them adequately so other people can understand them easily.

1) Program header

All your projects must start with a program header, something like this (fill in the information):

```#
#       CSc 310 Project ??
#       Name: ????
#       Date: ????
#       Description: This program does ????
#
```

The Description should be a brief statement of what the program does, what the user has to enter (if anything), and what results the program prints out.

2) Indentation

Your MIPS assembly language program should be neatly divided into four columns which line up properly. (Hint: using "tabs" is a good way to line things up.) The first column should contain labels only. The second column should contain instruction operations only. The third column should contain the list of operands for that instruction. The fourth column should contain comments (if any).

Make sure that each line of your program code is less than 80 characters. Tabs should be set to 8 characters.

3) Commenting style

You should write your comments so that even someone who is not familiar with MIPS assembly language can follow what is happening in the program.

Describe all important variables used in the .data section. Identify which registers are used as which variables, where appropriate.

Describe what the instructions are doing within the context of the program, *not* the definition for each instruction. You don't have to comment every single instruction, but it should be clear
what those lines of your program are doing. For example, this is good commenting style:

```assembly
# i     $s0
# x     $s1
# sum   $s2

main:   li $s0,1           # for (i=1; i<=10; i++)
        move $s1,$v0
        blez $s1,skip        # if (x > 0)
        add $s2,$s2,$s0     # sum=sum+i;

loop:   li $v0,5           # cin >> x;
        syscall
        move $s1,$v0
        blez $s1,skip        # if (x <= 0) goto skip
        add $s2,$s2,$s0     # sum=sum+i;

skip:   add $s0,$s0,1       # }
        ble $s0,10,loop
```

Notice that C/C++ style indentations within the comments helps to make the comments more readable. Just by reading the comments, we can tell that the "sum=sum+i" line is executed if x > 0, and all the statements from "cin..." through "sum = ..." belong inside the for loop.

In contrast, this is much less readable:

```assembly
loop:   li $v0,5           # cin >> x;
        syscall
        move $s1,$v0
        blez $s1,skip        # if (x <= 0) goto skip
        add $s2,$s2,$s0     # sum=sum+i;

skip:   add $s0,$s0,1       # i++;
        ble $s0,10,loop     # if (i <= 10) goto loop
```

Almost every single instruction is commented, but it's harder to read the comments. It's not obvious that we have a loop, and what goes inside the loop body.

However, this doesn't always mean that fewer comments are better! As much as possible, your comments should read like a high-level language program, with extra text to explain what is happening, if necessary. (Sometimes, it's not easy to make your comments look like a high-level language program.) You should always try to describe the algorithm, the "big picture" of what is happening with the code, and not just describe every single instruction and get lost in the low level details.

The programming examples that you'll use in this class, in `~whsu/310/PROGS`, will give you many examples of what the instructor considers good commenting style. (However, the programs provided in CSc 311 are often not properly commented; the difference should be obvious.)
Complete Program Listings for Chapter 2
(also found in ~whsu/310/PROGS on libra)

1) First MAL program (variables in memory)
2) First MAL program (variables in registers)
3) If statement example (variables in memory)
4) If statement example (variables in registers)
5) If-else example (variables in registers)
6) For loop example (variables in registers)
# CSc 310 Example 2.1: First MAL program, variables in memory
# Name: William Hsu
# Date: 7/11/2000
# Description: Demonstrates simple arithmetic instructions
#               with variables in memory
#
# int a=0,b0=1,c=2,d=3,e=4;
#
# int main(void)
# {
#   b0 = e;
#   a = b0 + c;
#   d = b0 - c;
#   e = e - 1;
#
#   cout << a << endl;
#   cout << b0 << endl;
#   cout << c  << endl;
#   cout << d  << endl;
#   cout << e  << endl;
# }

.data
a:    .word   0
b0:   .word   1
c:    .word   2
d:    .word   3
e:    .word   4
endl: .asciiz "\n"

.text
main: lw $s0,e
     sw $s0,b0
     lw $s0,b0
     lw $s1,c
     add $t0,$s0,$s1
     sw $t0,a
     lw $s0,b0
     lw $s1,c
     sub $t0,$s0,$s1
     sw $t0,d
     lw $s0,e

page 2.59
sub     $s0,$s0,1
sw      $s0,e

lw      $a0,a          # cout << a << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

lw      $a0,b0          # cout << b0 << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

lw      $a0,c          # cout << c << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

lw      $a0,d          # cout << d << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

lw      $a0,e          # cout << e << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

li      $v0,10
syscall
# CSc 310 Example 2.2: First MAL program, variables in registers
# Name: William Hsu
# Date: 7/11/2000
# Description: Demonstrates simple MAL arithmetic instructions
#
# int a=0,b0=1,c=2,d=3,e=4;
# int main(void)
# {
#   b0 = e;
#   a = b0 + c;
#   d = b0 - c;
#   e = e - 1;
#   cout << a << endl;
#   cout << b0 << endl;
#   cout << c << endl;
#   cout << d << endl;
#   cout << e << endl;
# }
#
# a     $s0
# b0    $s1
# c     $s2
# d     $s3
# e     $s4

.data
endl:   .asciiz "\n"

.text
main:   li      $s0,0
li      $s1,1
li      $s2,2
li      $s3,3
li      $s4,4

move    $s1,$s4      # b = e;
add     $s0,$s1,$s2   # a = b + c;
sub     $s3,$s1,$s2   # d = b - c;
sub     $s4,$s4,1     # e = e - 1;

move    $a0,$s0      # cout << a << endl;
li      $v0,1
syscall
la      $a0,endl
li      $v0,4
syscall

move   $a0,$s1          # cout << b0 << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

move   $a0,$s2          # cout << c << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

move   $a0,$s3          # cout << d << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

move   $a0,$s4          # cout << e << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall

li      $v0,10
syscall
# CSc 310 Example 2.3: If statement, variables in memory
# Name: William Hsu
# Date: 7/11/2000
# Description: Demonstrates if statement translation
#   variables in memory
#
#include <iostream.h>
int x=4,y=-1,sum=0;
#
int main()
{
  if (x > y)
    sum = sum + x;
#
  cout << sum << endl;
#
.data
x:     .word   4
y:     .word   -1
sum:   .word   0
endl:  .asciiz "\n"

.text
main:  lw    $s0,x
      lw    $s1,y
      lw    $s2,sum
      ble   $s0,$s1,skip    # if (x > y)
      add   $s2,$s2,$s0     #   sum = sum + x;

skip:  sw    $s2,sum
      move  $a0,$s2         # cout << sum << endl;
      li     $v0,1
      syscall

      la     $a0,endl
      li     $v0,4
      syscall

      li     $v0,10
      syscall
# CSc 310 Example 2.4: If statement, variables in registers
# Name: William Hsu
# Date: 7/11/2000
# Description: Demonstrates if statement translation
#
#
# #include <iostream.h>
# int x=4,y=-1,sum=0;
# int main()
# {
#   if (x > y)
#     sum = sum + x;
#   cout << sum << endl;
# }
#
# x     $s0
# y     $s1
# sum   $s2

.data
endl:   .asciiz "\n"

.text
main:   li      $s0,4
li      $s1,-1
li      $s2,0

ble     $s0,$s1,skip    # if (x > y)
add     $s2,$s2,$s0     # sum = sum + x;

skip:   move    $a0,$s2         # cout << sum << endl;
li      $v0,1
syscall

la      $a0,endl
li      $v0,4
syscall
li      $v0,10
syscall
# CSc 310 Example 2.5: If-else statement, variables in registers
# Name: William Hsu
# Date: 7/11/2000
# Description: Demonstrates nested if-else statement
#
#
# int main(void)
# {
#   int num, flag;
#   cin >> num;
#   if (num > 0)
#     flag = 1;
#   else if (num < 0)
#     flag = -1;
#   else flag = 0;
#   cout << flag << endl;
# }
#
# num  $s0
# flag $s1

.data
endl: .asciiz "\n"

.text
main:   li      $v0,5           #   cin >> num;
syscall
move    $s0,$v0
blez    $s0,next1       #   if (num > 0)
li      $s1,1           #     flag = 1;
b       cont
next1:  bgez    $s0,next2       #   else if (num < 0)
li      $s1,-1          #     flag = -1;
b       cont
next2:  li      $s1,0           #   else flag = 0;
cont:   move    $a0,$s1         #   cout << flag << endl;
li      $v0,1
syscall
la      $a0,endl
li      $v0, 4
syscall

li      $v0, 10
syscall
# CSc 310 Example 2.6: For loop
# Name: William Hsu
# Date: 7/11/2000
# Description: User enters limit. Program computes sum of integers from 1 to limit, and prints the sum.
#
#include <iostream.h>
int main(void)
{
int sum=0,i,limit;

    cin >> limit;
    for (i=1; i<=limit; i++)
        sum = sum + i;
    cout << sum << endl;
}

sum   $s0
i     $s1
limit $s2

.data
endl:   .asciiz "\n"

.text
main:   li      $v0,5           #   cin >> limit;
        syscall
move    $s2,$v0
li      $s0,0
li      $s1,1           #   for (i=1; i<=limit; i++)
        bgt     $s1,$s2,cont

loop:   add     $s0,$s0,$s1     #     sum = sum + i;
        add     $s1,$s1,1
        ble     $s1,$s2,loop

cont:   move    $a0,$s0         #   cout << sum << endl;
        li      $v0,1
        syscall
        la      $a0,endl
        li      $v0,4
        syscall
li $v0, 10
syscall
## ASCII Codes

<table>
<thead>
<tr>
<th>ASCII code (hex)</th>
<th>char</th>
<th>ASCII code (hex)</th>
<th>char</th>
<th>ASCII code (hex)</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nul</td>
<td>7</td>
<td>bel</td>
<td>a</td>
<td>nl</td>
</tr>
<tr>
<td>20</td>
<td>sp</td>
<td>40</td>
<td>@</td>
<td>60</td>
<td>\</td>
</tr>
<tr>
<td>21</td>
<td>!</td>
<td>41</td>
<td>A</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>22</td>
<td>&quot;</td>
<td>42</td>
<td>B</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>23</td>
<td>#</td>
<td>43</td>
<td>C</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>24</td>
<td>$</td>
<td>44</td>
<td>D</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>45</td>
<td>E</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>26</td>
<td>&amp;</td>
<td>46</td>
<td>F</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>27</td>
<td>'</td>
<td>47</td>
<td>G</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>28</td>
<td>(</td>
<td>48</td>
<td>H</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>29</td>
<td>)</td>
<td>49</td>
<td>I</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>2a</td>
<td>*</td>
<td>4a</td>
<td>J</td>
<td>6a</td>
<td>j</td>
</tr>
<tr>
<td>2b</td>
<td>+</td>
<td>4b</td>
<td>K</td>
<td>6b</td>
<td>k</td>
</tr>
<tr>
<td>2c</td>
<td>,</td>
<td>4c</td>
<td>L</td>
<td>6c</td>
<td>l</td>
</tr>
<tr>
<td>2d</td>
<td>-</td>
<td>4d</td>
<td>M</td>
<td>6d</td>
<td>m</td>
</tr>
<tr>
<td>2e</td>
<td>.</td>
<td>4e</td>
<td>N</td>
<td>6e</td>
<td>n</td>
</tr>
<tr>
<td>2f</td>
<td>/</td>
<td>4f</td>
<td>O</td>
<td>6f</td>
<td>o</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>50</td>
<td>P</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>51</td>
<td>Q</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>52</td>
<td>R</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>33</td>
<td>3</td>
<td>53</td>
<td>S</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>34</td>
<td>4</td>
<td>54</td>
<td>T</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>35</td>
<td>5</td>
<td>55</td>
<td>U</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>56</td>
<td>V</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>37</td>
<td>7</td>
<td>57</td>
<td>W</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
<td>58</td>
<td>X</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>59</td>
<td>Y</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>3a</td>
<td>:</td>
<td>5a</td>
<td>Z</td>
<td>7a</td>
<td>z</td>
</tr>
<tr>
<td>3b</td>
<td>;</td>
<td>5b</td>
<td>[</td>
<td>7b</td>
<td>{</td>
</tr>
<tr>
<td>3c</td>
<td>&lt;</td>
<td>5c</td>
<td>\</td>
<td>7c</td>
<td></td>
</tr>
<tr>
<td>3d</td>
<td>=</td>
<td>5d</td>
<td>]</td>
<td>7d</td>
<td>}</td>
</tr>
<tr>
<td>3e</td>
<td>&gt;</td>
<td>5e</td>
<td>^</td>
<td>7e</td>
<td>~</td>
</tr>
<tr>
<td>3f</td>
<td>?</td>
<td>5f</td>
<td>_</td>
<td>7f</td>
<td>del</td>
</tr>
</tbody>
</table>