Assembly Language and Computer Architecture Lab

Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

Week 6

- Assembly language program control structures.
- How to create logical (or boolean) variables in assembly.
- The basic control structures used in structured programming, and how to translate them into assembly code:

  a) if statements

  b) if-else statements

  c) if-elseif-else statements

  d) sentinel control loops

  e) counter control loops

  f) nested code block

Assembly language program control structures

- **Sequences**: allow programs to execute statements in order one after another

- **Branches**: allow programs to jump to other points in a program.

- **Loops**: allow a program to execute a fragment of code multiple times.

Should follow the structured programming principals and not attempt to develop the programs directly in assembly language.

The structured programming principals

Describe the algorithm

↓

Build in pseudo code

↓

Translate to assembly language

What is pseudo code

- Pseudo code is not a formal language.
- But a very rough set of malleable concepts which can be used to produce an outline of an assembly program.
- The language itself only includes enough detail to allow a programmer to understand what needs to done.
- Can be easily changed as the needs of the programmer change.

What is pseudo code

- For example, consider the following pseudo code program to read two numbers, add them, and print the result back to the user.

```
main
{
    register int i = input("Please enter the first value to add: ");
    register int j = input("Please enter the second value to add: ");
    register int k = i + j;
    print("The result is " + k);
}
```

- Create a program that will contain 3 integer values.
- The use of the **register** modifier on the **int** declaration tells the programmer that they should use a save register (**$s0**...**$s7**) if possible to maintain these values.

Control structures

- **goto** statement
- Simple **if** statements
- **if-else** statements
- **if-elseif-else** statements
- Sentinel control loop
- Counter control loop
- Nested code blocks

Use of **goto** statement

- The simplest branch statement
- Allows unrestricted branching to any point in a program
- Leads to many obfuscated programs before structured computing.
- But it was never the use of **goto** statements that lead to obfuscated programs
- It was programmers penchants for doing the expedient, resulting in unorganized programs.

Simple if statements

- **if** statements that do not have any else conditions.

- Considering the 3 following examples:
  - Example 1: with single logical condition
  - Example 2: with complex logical conditions
  - Example 3: with more complex logical condition

Example 1: With single logical condition

- Considering the following simple program:

```
if (num > 0)
{
    print("Number is positive")
}
```

- From this simple program, there is a lot of complexity hidden in it.

  - The variable **num** will not be directly useable in the program, and will have to be loaded into a register,

  - The subprogram for print must be created.

Example 1: With single logical condition

- Several conditions that are important in understanding the if statement:
  - The statement (**num > 0**) is a statement in which the **>** operator is returning a logical (boolean) value to be evaluated.
  - The value of the boolean variable will have to be calculated before it will be used in assembly language.
  - All boolean values will strictly be 0 (false) and 1 (true).
- Any code between an open brace "{" and close brace "}" is considered part of a code block.

Example 1
With single logical condition

- Translated into assembly language:

```
.data
        num: .word 5
        PositiveNumber: .asciiz "Number is positive"
.text
        # if (num > 0 )
        lw $t0, num
        sgt $t1, $t0, $zero      # $t1 is the boolean (num > 0)
        beqz $t1, end_if         # note: the code block is entered if
                                 # if logical is true, skipped if false.
           #{
           # print ("Number is positive")
           la $a0, PositiveNumber
           jal PrintString
           #}
        end_if:
        jal Exit

.include "utils.asm"
```

sgt: set greater than
if \$t0>0 then set \$t1=1 or 0

- The translation of (**num > 0**) takes 2 assembly instructions.
  - The first loads num into **\$t0** so that the values can be used.
  - The **sgt \$t1, \$t0, \$zero** instruction loads the boolean value into **\$t1** so that it can be compared in the **if** test.

Example 1
With single logical condition

- Translated into assembly language:

```
.data
        num: .word 5
        PositiveNumber: .asciiz "Number is positive"
.text
        # if (num > 0 )
        lw $t0, num
        sgt $t1, $t0, $zero      # $t1 is the boolean (num > 0)
        beqz $t1, end_if         # note: the code block is entered if
                                 # if logical is true, skipped if false.
           #{
           # print ("Number is positive")
           la $a0, PositiveNumber
           jal PrintString
           #}
        end_if:
        jal Exit

.include "utils.asm"
```

- The **if** test works by asking the question is the **boolean** value true.
  - If the **boolean** value is true, then the code block is entered (the branch is not taken).
  - If the test is false, branch to the end of the code block, and so the code block is not entered.

beqz: branch if equal zero
if true ($t1=1), enter code block
If false ($t1=0), jump to end_if

Example 1
With single logical condition

- Translated into assembly language:

```
.data
        num: .word 5
        PositiveNumber: .asciiz "Number
.text
        # if (num > 0 )
        lw $t0, num
        sgt $t1, $t0, $zero        # $t1 i
        beqz $t1, end_if           # note:
                                   # if lo
           #{
           # print ("Number is positive
           la $a0, PositiveNumber
           jal PrintString
           #}
        end_if:
        jal Exit

.include "utils.asm"
```

- When implementing a code block, the following will always be used. The final **}** for the code block will translate into a label. Thus the simple **if** code fragment above is translated by:

  - Calculating the **boolean** value to control entering the **if** statement code block.

  - Entering the code block if the boolean is true, or branching around it if it is false.

Example 2: With complex logical condition

- Considering the following condition:

```
if ((x > 0 && ((x%2) == 0))
                # is x > 0 and even?
```

- In a HLL, the compiler is going to reduce the complex logical condition into a single equation. So the complex **if** statement above would be translated into the equivalent of the following code fragment:

```
boolean flag = ((x > 0) && ((x%2) == 0))
if (flag)...
```

Example 2
With complex logical condition

- Pseudo code

```
boolean flag = ((x > 0) && ((x%2) == 0))
if (flag)...
```

- Translated into assembly language:

```
lw $t0, x
sgt $t1, $t0, $zero
rem $t2, $t0, 2
and $t1, $t1, $t2
beqz $t1, end_if
```

Example 3
With more complex logical condition

- Considering the following condition:

```
if ((x > 0) && ((x%2) == 0) && (x < 10))
                          # is 0 < x < 10 and even?
```

- The true power of this method of handling logical conditions becomes apparent as the logical conditions become more complex.

```
boolean flag = ((x > 0) && ((x%2) == 0) && (x < 10))
if (flag)...
```

Example 3
With more complex logical condition

- Pseudo code

```
boolean flag = ((x > 0) && ((x%2) == 0) && (x < 10))
if (flag)...
```

- Translated into assembly language:

```
lw $t0, x
sgt $t1, $t0, $zero
li $t5, 10
slt $t2, $t0, $t5
rem $t3, $t0, 2
and $t1, $t1, $t2
and $t1, $t1, $t3
beqz $t1, end_if
```

slt: set less than
if \$t0<\$t5 then set \$t2=1 or 0

if-else statement

- A more useful version of the **if** statement also allows for the false condition, or a **if-else** statement. If the condition is true, the first block is executed, otherwise the second block is executed.

- Considering the following program:

```
if (($s0 > 0) == 0)
{
    print("Number is positive")
}
else

{
    print("Number is negative")

}
```

if-else statement

1. Implement the conditional part of the statement

2. Add two labels to the program:

   - one for the **else**

   - one for the end of the **if**

```
lw $t0, num
sgt $t1, $t0, $zero
beqz $t1, else
#if block
b end_if
#else block
else:
end_if:
```

3. The **beqz** should be inserted after the evaluation of the condition to branch to the **else** label.

4. At the end of the **if** block, branch around the **else** block by using an unconditional branch statement to the **end_if**.

if-else statement

5. Once the structure of the **if-else** statement is in place, the code should be put for the block into the structure. This completes the **if-else** statement translation.

```
.data
        num: .word -5
        PositiveNumber: .asciiz "Number is positive"
        NegativeNumber: .asciiz "Number is negative"
.include "utils.asm"

.text
        lw $t0, num
        sgt $t1, $t0, $zero
        beqz $t1, else
                #if block
                la $a0, PositiveNumber
                li $v0, 4
                syscall
                b end_if
                #else block
        else:
                la $a0, NegativeNumber
                jal PrintString
        end_if:
        jal Exit
```

if-elseif-else statement

- Consider the following program:

```
if (grade > 100) || grade < 0)
{
    print("Grade must be between 0..100")
}
elseif (grade >= 90)
{
    print("Grade is A")
}
elseif (grade >= 80)
{
     print("Grade is B")
}
elseif (grade >= 70)
{
     print("Grade is C")
}
elseif (grade >= 60)
{
     print("Grade is D")
}
else{
     print("Grade is F")
}
```

if-elseif-else statement

1. Implement the beginning of the statement with a comment, and place a label in the code for:

   - Each **elseif** condition
   - Final **else**
   - **end_if** conditions

At the end of each code block place a branch to the **end-if** label. Once any block is executed, the entire **if-elseif-else** statement will be exited.

if-elseif-else statement

The code block would look as follows :

```
#if block
    # first if check, invalid input block
    b end_if
grade_A:
    b end_if
grade_B:
    b end_if
grade_C:
    b end_if
grade_D:
    b end_if
else:
    b end_if
end_if:
```

if-elseif-else statement

2. Next put the logic conditions in the beginning of each **if** and **elseif** block. In these **if** and **elseif** statements the code will branch to the next label.

slti: set less than immediate
if $s0<0 then set $t1=1 or 0

sge: set greater or equal
if $s0≧90 then set $t1=1 or 0

```
#if block
    lw $s0, num
    slti $t1, $s0, 0
    sgt $t2, $s0, 100
    or $t1, $t1, $t2
    beqz $t1, grade_A
    #invalid input block
    b end_if
grade_A:
    sge $t1, $s0, 90
    beqz $t1, grade_B
    b end_if
grade_B:
    sge $t1, $s0, 80
    beqz $t1, grade_C
    b end_if
```

```
grade_C:
    sge $t1, $s0, 70
    beqz $t1, grade_D
    b end_if
grade_D:
    sge $t1, $s0, 60
    beqz $t1, else
    b end_if
else:
    b end_if
end_if:
```

if-elseif-else statement

3. The last step is to fill in the code blocks with the appropriate logic.

```
.data
        num: .word 70
        InvalidInput: .asciiz "Number must be > 0 and < 100"
        OutputA: .asciiz "Grade is A"
        OutputB: .asciiz "Grade is B"
        OutputC: .asciiz "Grade is C"
        OutputD: .asciiz "Grade is D"
        OutputF: .asciiz "Grade is F"
.include "utils.asm"

.text
    #if block
        lw $s0, num
        slti $t1, $s0, 0
        sgt $t2, $s0, 100
        or $t1, $t1, $t2
        beqz $t1, grade_A
        #invalid input block
        la $a0, InvalidInput
        jal PrintString
        b end_if
    grade_A:
        sge $t1, $s0, 90
        beqz $t1, grade_B
        la $a0, OutputA
        jal PrintString
        b end_if
    grade_B:
        sge $t1, $s0, 80
        beqz $t1, grade_C
        la $a0, OutputB
        jal PrintString
        b end_if
    grade_C:
        sge $t1, $s0, 70
        beqz $t1, grade_D
        la $a0, OutputC
        jal PrintString
        b end_if
    grade_D:
        sge $t1, $s0, 60
        beqz $t1, else
        la $a0, OutputD
        jal PrintString
        b end_if
    else:
        la $a0, OutputF
        jal PrintString
        b end_if
    end_if:

jal Exit
```

Sentinel control loop

- The concept of a sentinel control loop is a loop with a guard statement that controls whether or not the loop is executed.
- The major use of sentinel control loops is to process input until some condition (a sentinel value) is met.
- For example, a sentinel control loop could be used to process user input until the user enters a specific value.

Sentinel control loop

- The following pseudo code fragment uses a **while** statement to implement a sentinel control loop which prompts for an integer and prints that integer back until the user enters a “-1” value.

```
int i = prompt("Enter an integer, or -1 to exit")
while (i != -1)
{
    print("You entered " + i);
    i = prompt("Enter an integer, or -1 to exit");
}
```

Sentinel control loop

1. Set the sentinel to be checked before entering the loop.

2. Create a label for the start of the loop, so at the end of the loop the program control can branch back to the start of the loop.

3. Create a label for the end of the loop, so the loop can branch out when the sentinel returns false.

4. Put the check code in place to check the sentinel. If the sentinel check is true, branch to the end of the loop.

5. Set the sentinel to be checked as the last statement in the code block for the loop, and unconditionally branch back to the start of the loop. This completes the loop structure.

Sentinel control loop

- The code that appears similar to the following:

```
.data
        prompt: .asciiz "Enter an integer, -1 to stop: "

.text
        #set sentinel value (prompt the user for input)
        la $a0, prompt
        jal PromptInt
        move $s0, $v0
        start_loop:
                sne $t1, $s0, -1
                beqz $t1, end_loop
                # code block
                la $a0, prompt
                jal PromptInt
                move $s0, $v0
                b start_loop
        end_loop:
```

sne: set not equal
if $\$s0 \neq -1$ then set $\$t1=1$ or 0

Sentinel control loop

6. The structure needed for the sentinel control loop is now in place.

   The logic to be executed in the code block can be included, and any other code that is needed to complete the program.

Sentinel control loop

- The final result of this program follows:

```
.data
        prompt: .asciiz "Enter an integer, -1 to stop: "
        output: .asciiz "\nYou entered: "
.include "utils.asm"

.text
        #set sentinel value (prompt the user for input)
        la $a0, prompt
        jal PromptInt
        move $s0, $v0
        start_loop:
                sne $t1, $s0, -1
                beqz $t1, end_loop

                # code block
                la $a0, output
                move $a1, $s0
                jal PrintInt

                la $a0, prompt
                jal PromptInt
                move $s0, $v0
                b start_loop
        end_loop:
        jal Exit
```

Counter control loop

- A counter controlled loop is a loop which is intended to be executed some number of times.

- The general format is to specify a **starting value** for a counter, the ending condition (normally when the counter reaches a predetermined value), and the increment operation on the counter.

Counter control loop

- Considering the following example which sums the values from 0 to n-1.

```
n = prompt("enter the value to calculate the sum up to: ")
total = 0; # Initial the total variable for sum
for (i = 0; i < n; i++)
{
   total = total + i
}
print("Total = " + total);
```

- The for statement itself has 3 parts:
  - The first is the initialization that occurs before the loop is executed (here it is "i=0").
  - The second is the condition for continuing to enter the loop (here it is "i < size").
  - The final condition specifies how to increment the counter (here it is "i++", or add 1 to i).

Counter control loop

1. Implement the initialization step to initialize the counter and the ending condition variables.

2. Create labels for the start and end of the loop.

3. Implement the check to enter the loop block, or stop the loop when the condition is met.

4. Implement the counter increment, and branch back to the start of the loop.

Counter control loop

- When the above steps are completed, the basic structure of the counter control loop has been implemented, and the code should look similar to the following:

sle: set less or equal
if \$s0≦\$s1 then set \$t1=1 or 0

```
.data
        n: .word 5

.text
        li $s0, 0
        lw $s1, n
        start_loop:
                sle $t1, $s0, $s1
                beqz $t1, end_loop
                # code block
                addi $s0, $s0, 1
                b start_loop
        end_loop:
```

Counter control loop

5. Implement the code block for the for statement. Implement any other code necessary to complete the program. The final assembly code for this program should look similar to the following.

```
.data
        prompt: .asciiz "Enter the value to calculate the sum up to: "
        output: .asciiz "The final result is: "

.include "utils.asm"

.text
        la $a0, prompt
        jal PromptInt
        move $s1,$v0
        li $s0,0
        #Initialize the total
        li $s2,0

        start_loop:
                sle $t1, $s0, $s1
                beqz $t1, end_loop

                # code block
                add $s2, $s2, $s0

                addi $s0, $s0, 1
                b start_loop
        end_loop:

        la $a0, output
        move $a1, $s2
        jal PrintInt

        jal Exit
```

Nested code blocks

- It is common in most algorithms to have nested code blocks.

- Consider the following example:

```
int n = prompt("Enter a value for the summation n, -1 to stop");
while (n != -1)
{
    if (n < -1)
    {
        print("Negative input is invalid");
    }
    else
    {
        int total = 0
        for (int i = 0; i < n; i++)
        {
             total = total + i;
        }
        print("The summation is " + total);
    }
}
```

Nested code blocks

- This program consists of:
  - a sentinel control loop, to get the user input
  - an **if** statement, to check that the input is greater than 0
  - a counter control loop
- The **if** statement is nested inside of the sentinel control block, and the counter loop is nested inside of the **if-else** statement.

Nested code blocks

1. Begin by implementing the outer most block, the sentinel control block. The code should look similar to the following:

```
#Sentinel Control Loop
.data
        prompt: .asciiz "Enter an integer, -1 to stop: "
.text
        la $a0,prompt
        jal PromptInt
        move $s0,$v0
        start_outer_loop:
                sne $t1,$s0,-1
                beqz $t1,end_outer_loop

                #code block

                la $a0,prompt
                jal PromptInt
                move $s0,$v0
                b start_outer_loop
        end_outer_loop:
```

Nested code blocks

- The code block in the sentinel loop in the above fragment is now replaced by the **if-else** statement to check for valid input.

- When completed, the code should look similar to the following:

```
#code block
#If test for valid input
slti $t1,$s0,-1
beqz $t1,else
        #if block
        b end_if
else:
        #else block
end_if:
```

Nested code blocks

3. The **if** block in the above code fragment is replaced by the error message, and the else block is replaced by the sentinel control loop.

```
    #if block
    la $a0,error
    jal PrintInt
    b end_if
else:
    #else block
    #summation loop
    li $s1,0
    li $s2,0 #initialize loop
start_inner_loop:
        sle $t1,$s1,$s0
        beqz $t1,end_inner_loop

        add $s2,$s2,$s1

        addi $s1,$s1,1
        b start_inner_loop
end_inner_loop:
la $s0,output
move $a1,$s2
jal PrintInt
end_if:
```

Nested code blocks

- The completed program:

```
#Sentinel Control Loop
.data
        prompt: .asciiz "\nEnter an integer, -1 to stop: "
        error: .asciiz "\nValues for n must be > 0"
        output: .asciiz "\nThe total is: "
.include "utils.asm"

.text
        la $a0,prompt
        jal PromptInt
        move $s0,$v0
        start_outer_loop:
                sne $t1,$s0,-1
                beqz $t1,end_outer_loop

                #code block
                #If test for valid input
                slti $t1,$s0,-1
                beqz $t1,else
                        #if block
                        la $a0,error
                        jal PrintInt
                        b end_if
                else:
                        #else block
                        #summation loop
                        li $s1,0
                        li $s2,0 #initialize loop

                        start_inner_loop:
                                sle $t1,$s1,$s0
                                beqz $t1,end_inner_loop

                                add $s2,$s2,$s1

                                addi $s1,$s1,1
                                b start_inner_loop
                        end_inner_loop:
                        la $s0,output
                        move $a1,$s2
                        jal PrintInt
                end_if:

                la $a0,prompt
                jal PromptInt
                move $s0,$v0
                b start_outer_loop
        end_outer_loop:
```

A full assembly language program

Average Grade Program

- Implement a program which reads numeric grades from a user and calculate an average.
- The average and corresponding letter grade will be printed to the console.

A full assembly language program

1. Before starting the project, it is recommended that the pseudo code be written:
    - Allows the programmer to reason at a higher level of abstraction
    - Makes it easier to implement the code because it is a straight translation from pseudo code to assembly.
    - The pseudo code serves as documentation for how the program works and should be included in a comment at the start of the assembly file, but not kept in a separate file so it does not get lost.

A full assembly language program

2. Include a preamble comment giving information such as:

- Filename
- Author
- Date
- Purpose
- Modification History
- Pseudo Code

```
# Pseudo Code
#global main()
#{
#     // The following variables are to be stored in data segment, and
#     // not simply used from a register.  They must be read each time
#     // they are used, and saved when they are changed.
#     static volatile int numberOfEntries = 0
#     static volatile int total = 0
#
#     // The following variable can be kept in a save register.
#     register int inputGrade  # input grade from the user
#     register int average
#
#     // Sentinel loop to get grades, calculate total.
#     inputGrade = prompt("Enter grade, or -1 when done")
#     while (inputGrade != -1)
#     {
#         numberOfEntries = numberOfEntries + 1
#         total = total + inputGrade
#         inputGrade = prompt("Enter grade, or -1 when done")
#     }
#
#     # Calculate average
#     average = total / numberOfEntries
#
#     // Print average
#     print("Average = " + average)
#
#     //Print grade if average is between 0 and 100, otherwise an error
#     if ((grade >= 0) & (grade <= 100))
#     {
```

```
#           if (grade >= 90)
#           {
#               print("Grade is A")
#           }
#           if (grade >= 80)
#           {
#               print("Grade is B")
#           }
#           if (grade >= 70)
#           {
#               print("Grade is C")
#           }
#           else
#           {
#               print("Grade is F")
#           }
#       }
#       else
#       {
#           print("The average is invalid")
#       }
#}
```

Assigment 6.1

- This assignment implements "if-then-else" statement using some fundamental instructions, such as **slt**, **addi**, **jump** and **branch**.

if (i<=j)
x=x+1;
z=1;
else
y=y-1;
z=2*z;

- The assembly code follows:

```
start:
       slt    $t0,$s2,$s1         # j<i
       bne    $t0,$zero,else      # branch to else if j<i
       addi   $t1,$t1,1           #     then part: x=x+1
       addi   $t3,$zero,1         # z=1
       j      endif               # skip "else" part
else:  addi   $t2,$t2,-1          # begin else part: y=y-1
       add    $t3,$t3,$t3         # z=2*z
endif:
```

Assigment 6.1

- Create a new project to implement the above code.
    - Initialize for i and j variable.
    - Compile and upload to the simulator.
    - Run this program step by step, observe the changing of memory and the content of registers at each step.

Assigment 6.2

Modify the Assignment 6.1, so that the condition tested is:

- $i<j$
- $i>=j$
- $i+j<=0$
- $i+j > m+n$

Assigment 6.3

- The following pseudo code demonstrates how to implement loop statement. This program computes the sum of elements of array A.

```
loop:  i=i+step;
       Sum=sum+A[i];
       If(i !=n) goto loop;
```

- Assuming that the index i, the starting address of A, the comparison constant n, step and sum are found in registers $s1, $s2, $s3, $s4 and $s5, respectively.

```
.text
loop: add   $s1,$s1,$s4      #i=i+step
      add   $t1,$s1,$s1      #t1=2*s1
      add   $t1,$t1,$t1      #t1=4*s1
      add   $t1,$t1,$s2      #t1 store the address of A[i]
      lw    $t0,0($t1)       #load value of A[i] in $t0
      add   $s5,$s5,$t0      #sum=sum+A[i]
      bne   $s1,$s3,loop     #if i != n, goto loop
```

Assigment 6.3

Create a new project implementing the above code.

- Initialize for i, n, step, sum variables and array A.
- Compile and upload to the simulator.
- Run this program step by step, observe the changing of memory and the content of registers by each step.
- Try to test with some more cases (change the value of variables).

Assigment 6.4

Modify the Assignment 6.3, so that the condition tested at the end of the loop is

- i<n
- i<=n
- sum>=0
- A[i] == 0

Assigment 6.5

- A switch/case statement allows multiway branching based on the value of an integer variable.

- In the following example, the switch variable test can assume one of the three values in [0, 2] and a different action is specified for each case.

```
switch(test) {
    case 0:
        a=a+1; break;
    case 1:
        a=a-1; break;
    case 2:
        b=2*b; break;
}
```

Assigment 6.5

- Assuming that a and b are stored in registers $s2 and $s3.

```
.data
test: .word 1
.text
            la    $s0, test   #load the address of test variable
            lw    $s1, 0($s0) #load the value of test to register $t1
            li    $t0, 0      #load value for test case
            li    $t1, 1
            li    $t2, 2
            beq   $s1, $t0, case_0
            beq   $s1, $t1, case_1
            beq   $s1, $t2, case_2
            j     default
case_0:     addi  $s2, $s2, 1       #a=a+1
            j     continue
case_1:     sub   $s2, $s2, $t1     #a=a-1
            j     continue
case_2:     add   $s3, $s3, $s3     #b=2*b
            j     continue
default:
continue:
```

Assigment 6.5

Create a new project implementing the above code.

- Compile and upload to the simulator.
- Run this program step by step; observe the changing of memory and the content of registers by each step.
- Change the value of test variable and run this program some times to check all cases.

Assigment 6.6

- Create a new project to implement this function: find the element with the largest absolute value in a list of integers.

- Assuming that this list is store in an integer array and we know the number of elements in it.

Assignment 6.7

- The sum of two 32-bit integers may not be representable in 32 bits. In this case, we say that an overflow has occurred. Overflow is possible only with operands of the same sign.

- For two nonnegative (negative) operands, if the sum obtained is less (greater) than eitheir operand, overflow has occurred.

- The following program dectects overflow based on this rule. Two operands are stored in register **$s1** and **$s2**, the sum is stored in register **$s3**. If overflow occur, **$t0** register is set to 1 and clear to 0 in otherwise.

Assignment 6.7

```
.text
start:
        li      $t0,0                  #No Overflow is default status
        addu    $s3,$s1,$s2            # s3 = s1 + s2
        xor     $t1,$s1,$s2            #Test if $s1 and $s2 have the same sign

        bltz    $t1,EXIT               #If not, exit
        slt     $t2,$s3,$s1
        bltz    $s1,NEGATIVE#Test if $s1 and $s2 is negative?
        beq     $t2,$zero,EXIT         #s1 and $s2 are positive
               # if $s3 > $s1 then the result is not overflow
        j       OVERFLOW
NEGATIVE:
        bne     $t2,$zero,EXIT         #s1 and $s2 are negative
               # if $s3 < $s1 then the result is not overflow
OVERFLOW:
        li      $t0,1                  #the result is overflow
EXIT:
```

Assigment 6.8

- Write a program to find prime numbers from 3 to n in a loop by dividing the number n by all numbers from 2...n/2 in an inner loop.
  - Using the remainder (rem) operation, determine if n is divisible by any number.
  - If n is divisible, leave the inner loop.
  - If the limit of n/2 is reached and the inner loop has not been exited, the number is prime and you should output the number.
  - So if the user were to enter 25, your program would print out "2, 3, 5, 7, 11, 13, 17, 19, 23".

Assigment 6.9

- Write a program to prompt the user for a number, and determine if that number is prime.
    - Your program should print out "Number *n* is prime" if the number is prime, and "Number *n* is not prime if the number is not prime.
    - The user should be able to enter input a "-1" is entered.
    - It should print an error if 0, 1, 2 or any negative number other than -1 are entered.

Assigment 6.10

- Write a program to allow a user to guess a random number generated by the computer from 1 to *maximum* (the user should enter the maximum value to guess).
  - In this program the user will enter the value of *maximum*, and the syscall service 42 will be used to generate a random number from 1 to maximum.
  - The user will then enter guesses and the program should print out if the guess is too high or too low until the user guesses the correct number.
  - The program should print out the number of guesses the user took.

End of week 6