

What you will learn.

In this chapter you will learn:

- 1) Why goto statements exist in languages.
- 2) How to create logical (or boolean) variables in assembly.
- 3) The basic control structures used in structured programming, and how to translate them into assembly code. The basic control structures covered are:
 - a) if statements
 - b) if-else statements
 - c) if-elseif-else statements
 - d) sentinel control loops
 - e) counter control loops.
- 4) How to calculate branch offsets.

Chapter 7 Assembly language program control structures

The structured programming paradigm is built on the concept that all programs can be built using just **3 types of program control structures**. These structures are:

- **Sequences** that allow programs to execute statements in order one after another.
- **Branches** that allow programs to jump to other points in a program.
- **Loops** that allow a program to execute a fragment of code multiple times.

Most modern HLLs are implemented based on these 3 program control structures with some notable extensions (Java exception handling, continue, and break statements). But as was pointed out in Chapter 5 on simple subprogram execution, the only **way to control program execution sequence in assembly language is through the `$pc` register**. Therefore, in assembly there are no native structured program constructs. This does not mean that an assembly language programmer should abandon the principals of structured programming. What the lack of language based structured programming constructs means is that the assembler programmer is responsible for writing code which aligns with these principals. Not following structured programming principals in assembly is a sure way to create *spaghetti* code, or code where control is passed uncontrolled around the program, much like spaghetti noodles intertwine and following individual strands becomes difficult.

This chapter will introduce pseudocode structure programming control structures like those in Java/C/C++/C#. Programmers familiar with those languages should be able follow the programs with no problems. The text will then show how to translate each control structure from pseudocode into assembly.

All programs in this chapter will be preceded by a pseudocode implementation of the algorithm. Translation from the pseudocode into MIPS assembly will be shown with the program always

following the translations. No programs will be developed directly into assembly. The reason for this is though the direct implementation of the programs in assembly allows more flexibility, programmers in assembly language programming often implement these programs in very unstructured fashions which often result in poor programs and can develop into poor understanding and practices.

To reemphasize the point, it has been the experience of the author that although many new assembly language programmers often try to avoid the structured programming paradigm and reason through an assembly language program, the results are seldom satisfactory. The reader is strongly advised to follow the principals outlined in this chapter, and not attempt to develop the programs directly in assembly language.

Chapter 7.1 Use of goto statements

Many readers of this text will quickly recognize the main mechanism for program control in assembly, the branch statement, as simply a goto statement. These readers have often been told since they started programming that goto statements are evil and should never be used. The reasoning behind this rule is seldom explained and an almost religious adherence has developed to the principal that goto statements are always suspect and should never be used. Like most unexamined principals, this simply misses the larger point and is incorrect except for limited and defined circumstances.

The problem with goto statements is that they allow unrestricted branching to any point in a program. Indeed, this type of unrestricted branching led to many obfuscated programs before structured computing. However, with the advent of structured programming languages, the use of the term *spaghetti code* has even gone out of the normal programmer's vernacular. 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. The unrestricted goto statement never was the problem, it simply was the mechanism that allowed the programmers to create problems.

In assembly language, the only method of program control is through the `$pc`, and most often using branch statements. The branch statements themselves will not lead to unorganized programs, but the unorganized thoughts of the programmers. So, this chapter will not teach how to reason about assembly language programs. All programs will be first structured in pseudocode, and then translated into assembly language. Readers who follow the methodology presented in this text will never encounter an *unrestricted goto*. All branch statements will be explicitly defined, and all branches will be within blocks of code, just as in structured programming languages. So, the branch statements in this text are not evil, and the idea that somehow branching is wrong needs to be modified in the reader's mind.

Note that there is also a practical reason why branching is used here. There is no other available mechanism to use to implement program control structures such as branches and loops. So regardless of the readers qualms about this topic, the reality of the situation is that branch statements are the only valid mechanism to implement programs in assembly language.

Chapter 7.2 Simple if statements

This section will deal with simple if statement, e.g., if statements that do not have any else conditions. This section will give two examples. The first shows how to translate a pseudocode program with a single logical condition in the if statement. The second shows how to handle complex logical conditions.

Chapter 7.2.1 Simple if statements in pseudo code

This section will begin with a small example of an if statement.

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

This program fragment looks simple but, as we have already seen, there is a lot of complexity hidden in it. First, the variable `num` will not be directly useable in the program and will have to be loaded into a register and the `subprogram for print must be created`. But hidden in this program fragment are several conditions that are important in understanding the if statement.

- 1) The statement `(num > 0)` is a statement in which the `>` operator is returning a logical (boolean) value to be evaluated. It might be easier if the statement was written as follows, which has the same meaning.

```
boolean flag = num > 0;
if (flag) ...
```

There is no mechanism for putting an expression in a branch statement, so the value of the boolean `variable will have to be calculated before it will be used in` assembly language, just as in this example.

One peculiarity in assembly (as in C/C++) is that logical values are 0 (false) and anything else (true). This can lead to all sorts of strange consequences, so this behavior will not be allowed and all boolean values will strictly be 0 (false) and 1 (true).

- 2) Code blocks are the central organizing unit in this pseudocode. Any code between an open brace `"{"` and close brace `"}"` is considered part of a code block. Subprograms consist of a code block and all control structures must use code blocks to implement the code within a condition.

Chapter 7.2. 2 Simple if statement translated to assembly

The assembly language program for the code fragment above is shown below.

check flag

```

.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

.data
num: .word 5
PositiveNumber: .asciiz "Number is positive"

.include "utils.asm"

```

Program 7-1: A simple program to show an if statement

The code fragment comments are inserted to show how the pseudocode is translated into assembly. The following explanation helps to explain how this program works.

- 1) 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.
- 2) 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.
 This might appear backward as the branch happens if the condition is false, but it is the easiest way to implement the logic (the exercises ask how to implement this if the branch is taken if the condition is true). The reader will quickly grow accustomed to this if used consistently and doing things in a consistent manner is the best defense against annoying bugs.
- 3) There are over 20 formats for the branch instruction in MARS. This text will only use 2. If the branch is based on a condition, the `beqz` will always be chosen. The `seq, slt, sle, sgt, sge` operators will be used to set the correct condition for the `beqz` operator. The only other branch used will be the unconditional branch instruction, `b`.

- 4) When implementing a code block, the following will always be used. The final “}” for the code block will translate into a label. This label will be accessible from the branch statement which is determining whether to enter the code block. Thus, the simple `if` code fragment above is translated by:
 - a) calculating the `boolean` value to control entering the `if` statement code block.
 - b) entering the code block if the `boolean` is true or branching around it if it is false.

Chapter 7.2.3 Simple if statement with complex logical conditions

While the example above shows how to translate a single logical condition, the question of how to translate complex logical conditions is more complex. Programmers might think that to translate a condition such as the one that follows requires complex programming logic.

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

In fact, one of the reasons programs became complex before structured programming became prevalent is that programmers would try to solve this type of complex logical condition using programming logic.

The easy way to solve this problem is to realize that 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)...
```

This code fragment is easily translated into assembly language as follows:

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

Program 7-2: Assembly logic for `((x > 0) && ((x%2) == 0))`

Once again it is left as an exercise for the programmer to convince themselves that this is much easier than implementing the logical condition using logic and various branch statements. The true power of this method of handling logical conditions becomes apparent as the logical conditions become more complex. Consider the following logical condition:

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

This can be translated into assembly language exactly as it appears in pseudocode.

```
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
```

Program 7-3: Assembly language logic for $((x > 0) \&\& ((x\%2) == 0) \&\& (x < 10))$

Chapter 7.3 if-else statements

A more useful version of the if statement also allows for the false condition, or an if-else statement. If the condition is true, the first block is executed, otherwise the second block is executed. A simple code fragment that illustrates this point is shown below.

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

This is a modification to logic in Program 7.1. This code will print if the number is positive or negative. The purpose here is to show how to translate an if-else statement from pseudocode to assembly language. The translation of the if-else statement occurs using the following steps.

- 1) Implement the conditional part of the statement as in Program 7.1.
- 2) Add two labels to the program, one for the else and one for the end of the if. The `beqz` should be inserted after the evaluation of the condition to branch to the else label.
- 3) At the end of the if block, branch around the else block by using an unconditional branch statement to the `endif`. You now have the basic structure of the if statement, and your code should look like the following assembly code fragment.

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

Program 7-4: Assembly code fragment for an if-else statement

- 4) Once the structure of the if-else statement is in place, you should put the code for the block into the structure. This completes the if-else statement translation. This is the following program.

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

.data
num: .word -5
PositiveNumber: .asciiz "Number is positive"
NegativeNumber: .asciiz "Number is negative"

.include "utils.asm"
```

Program 7-5: if-else program example

Chapter 7.4 if-elseif-else statements

The final type of branch to be introduced in this text allows the programmer to choose one of several options. It is implemented as an if-elseif-else statement. The if and elseif statements will contain a conditional to decide if they will be executed or not. The else will be automatically chosen if no condition is true.

To introduce the if-elseif-else statement, the following program which translates a number grade into a letter grade is implemented. The following pseudocode fragment shows the logic for this if-elseif-else statement.

```
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")
    }

```

To translate the `if-elseif-else` statement, once again the overall structure for the statement will be generated and then the code blocks will be filled in. Students and programmers are strongly encouraged to implement algorithmic logic in this manner. Students who want to implement the code in a completely forward fashion, where statements are generated as they exist in the program, will find themselves completely overwhelmed and will miss internal and stopping conditions, especially when nest blocks containing nested logic is used late in this chapter.

The steps in the translation of the `if-elseif-else` statement are as follows.

- 1) Implement the beginning of the statement with a comment and place a label in the code for each `elseif` condition, and for the final `else` and `end_if` conditions. At the end of each code block place a branch to the `end-if` label (once any block is executed you will exit the entire `if-elseif-else` statement). Your code 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:

```

- 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. When this step is completed, you should now have code that looks like the following:

```

#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:

```

- 3) The last step is to fill in the code blocks with the appropriate logic. The following program implements this completed if-elseif-else statement.

```

.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

.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"

```

Program 7-6: Program to implement if-elseif-else statement in assembly

Chapter 7.5 Loops

There are two basic loop structures found in most introduction to programming books. These two loop structures are **sentinel-controlled loops** and **counter controlled loops**. These loops are like while loops and for loops in most programming languages, so in this text the while loop will be used to implement sentinel-control loops and the for loop to implement counter-control loops. How to translate each of these looping structures from pseudocode into assembly language will be covered in the next two sections.

Chapter 7.5.1 Sentinel-control loop

The definition of a sentinel is a guard, so the concept of a **sentinel-control loop** is a loop with a **guard statement that controls whether 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, for example -1. The following pseudocode 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.

```

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");
}

```

The following defines the steps involved in translating a sentinel-control loop from pseudocode into assembly.

- 1) Set the sentinel to be checked before entering the loop.
- 2) Create a label for the start of the loop. This is so that 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. This is 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, and you should have code that appears similar to the following:

```
.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:

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

- 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. The result of this program follows.

```
.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

.data
prompt: .asciiz "\nEnter an integer, -1 to stop: "
output: .asciiz "\nYou entered: "

.include "utils.asm"

```

Program 7-7: Sentinel control loop program

Chapter 7.5.2 Counter-control loop

A counter-controlled loop is a loop which is intended to be executed some number of times. Normally this is associated with a for loop in most HLL. 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. An example is the following pseudocode for loop 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). These 3 parts are included in the translation of the structure.

The following defines the steps involved in translating a counter-control loop from pseudocode into assembly.

- 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. When you have completed these steps, the basic structure of the counter control loop has been implemented, and your code should look like the following:

```

.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:

.data
    n: .word 5

```

- 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 like the following.

```

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

start_loop:
    slt $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

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

.include "utils.asm"

```

Program 7-8: Counter control loop program

Chapter 7.6 Nested code blocks

It is common in most algorithms to have nested code blocks. A simple example would be a program which calculates the sum of all values from 0 to n , where the user enters values for n until $n-1$ if entered. In addition, there is a constraint on the input that only positive values of n be considered, and any negative values of n will produce an error.

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), and 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. Now the importance of being able to structure this program using pseudocode and to translate the pseudocode into assembly becomes apparent.

The pseudocode for this algorithm follows.

```
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);
    }
}
```

The program to implement this pseudocode is much larger and more complex. Implementing the program without first producing the pseudocode and translating it to assembly, even for a relatively simple algorithm such as this, is difficult and often yields unfavorable results

However, the translation of this pseudocode into assembly is a relatively straight forward process, as will be illustrated here.

- 1) Begin by implementing the outer most block, the sentinel-control block. Your code should look like the following:

```
# Sentinel Control Loop
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:

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

```

- 2) 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, your code should look like the following:

```

# Sentinel Control Loop
la $a0, prompt
jal PromptInt
move $s0, $v0
start_outer_loop:
    sne $t1, $s0, -1
    beqz $t1, end_outer_loop

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

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

```

- 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. This entire code fragment is then placed in the program resulting in the following complete program.

```

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

    # If test for valid input
    slti $t1, $s0, -1
    beqz $t1, else
        la $a0, error
    else:
        # Sentinel Control Loop
        la $a0, prompt
        jal PromptInt
        move $s0, $v0
        b start_outer_loop
    end_outer_loop:

```

```

        jal PrintString
        b end_if
    else:
        # summation loop
        li $s1, 0
        li $s2, 0 # initialize total

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

            add $s2, $s2, $s1

            addi $s1, $s1, 1
            b start_inner_loop
        end_inner_loop:
        la $a0, output
        move $a1, $s2
        jal PrintInt

    end_if:

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

.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"

```

Program 7-9: Program illustrating nested blocks

Though somewhat long, this assembly code is straight forward to produce and relatively easy to follow, particularly if the documentation at the start of the program includes pseudocode for the algorithm that was used.

Chapter 7.7 A full assembly language program

Now that the basics have been covered, a real assembly language program will be implemented. This program will read numeric grades from a user and calculate an average. The average and corresponding letter grade will be printed to the console.

Before starting, it is recommended that the pseudocode be written. This serves two purposes. First it allows the programmer to reason at a higher level of abstraction, and it makes it easier to implement the code because it is a straight translation from pseudocode to assembly. Second, the pseudocode serves as documentation for how the program works. The pseudocode should be included in a comment at the start of the assembly file

Once the pseudocode is written, the assembly code can be implemented. This is done below. Note that this is a program and not a code fragment that is used to illustrate a MIPS assembly feature. Therefore, this has a preamble comment giving information such as Filename, Author, Date, Purpose, Modification History, and the Pseudocode. Most professors and companies have their own standard for preamble comments, and they should be followed when documenting the code.

Finally realize that it is a myth that assembly code is not readable. If care is taken when writing it and documenting it, it can be just as readable as code in a higher-level language. However, the code is much more verbose and the ability to use abstraction is greatly reduced. Just as a high-level language is no substitute for assembly when it is needed, assembly is no substitute for a high-level language when it is appropriate.

```
# Filename: AverageGrade.asm
# Author:   Charles Kann
# Date:     12/29/2013
# Purpose:  Illustration of program to calculate a student grade
# Modification Log:
#           12/29/2014 - Initial release
#
# 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")
#     }
#}

.text
.globl main
main:
    # Register Conventions:
    #     $s0 - current inputGrade
    #     $s1 - average
    la $a0, prompt
    jal PromptInt
    move $s0, $v0

BeginInputLoop:
    addi $t0, $zero, -1        # set condition $s0 != -1
    sne $t0, $t0, $s0
    beqz $t0, EndInputLoop    # check condition to end loop

    la $t0, numberOfEntries # increment # of entries
    lw $t1, 0($t0)
    addi $t1, $t1, 1
    sw $t1, 0($t0)

    la $t0, total              # accumulate total
    lw $t1, 0($t0)
    add $t1, $t1, $s0
    sw $t1, 0($t0)

    la $a0, prompt            # prompt for next input
    jal PromptInt
    move $s0, $v0
    b BeginInputLoop
EndInputLoop:

    la $t0, numberOfEntries    #Calculate Average
    lw $t1, 0($t0)
    la $t0, total
    lw $t2, 0($t0)
    div $s1, $t2, $t1

```

```

    la $a0, avgOutput          # Print the average
    move $a1, $s1
    jal PrintInt
    jal PrintNewLine

    sge $t0, $s1, 0             # Set the condition
                                # (average >= 0) & (average <= 100)

    addi $t1, $zero, 100
    sle $t1, $s1, $t1
    and $t0, $t0, $t1
    beqz $t0, AverageError      # if Not AverageError
                                # PrintGrades
    sge $t0, $s1, 90
    beqz $t0, NotA
        la $a0, gradeA
        jal PrintString
        b EndPrintGrades
    NotA:
        sge $t0, $s1, 80
        beqz $t0, NotB
        la $a0, gradeB
        jal PrintString
        b EndPrintGrades
    NotB:
        sge $t0, $s1, 70
        beqz $t0, NotC
        la $a0, gradeC
        jal PrintString
        b EndPrintGrades
    NotC:
        la $a0, gradeF
        jal PrintString
        b EndPrintGrades
    EndPrintGrades:
        b EndAverageError
    AverageError:                #else AverageError
        la $a0, invalidAvg
        jal PrintString
    EndAverageError:

    jal Exit

.data
    numberOfEntries: .word 0
    total:           .word 0
    average:         .word
    prompt:          .asciiz "Enter grade, or -1 when done: "
    avgOutput:       .asciiz "The average is "
    gradeA:          .asciiz "The grades is an A"
    gradeB:          .asciiz "The grade is a B"
    gradeC:          .asciiz "The grade is a C"
    gradeF:          .asciiz "The grade is a F"
    invalidAvg:      .asciiz "The average is invalid"

.include "utils.asm"

```

Chapter 7.8 How to calculate branch amounts in machine code

This chapter has shown how to use the branch statements to implement structure programming logic. However, how a branch statement manipulates the `$pc` register to control the execution has yet to be discussed. This section will cover the details of how the branch statement is implemented in machine code.

Chapter 7.8.1 Instruction Addresses

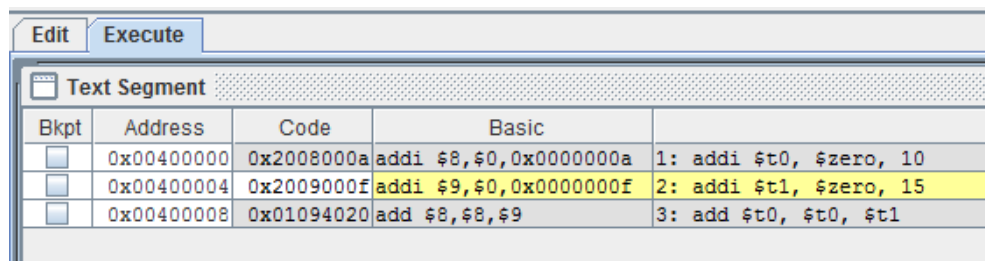
When the memory for the MIPS computer was shown in section 3.2, a segment labeled *program text* (or simply *text*) was shown as starting at address 0x00400000. This section of the memory contains the machine code translation of the instructions from the `.text` segment of your program. Thus, the text segment of memory is where all the machine code instructions for the program are stored.

When a program is assembled, the first instruction is inserted at address 0x00400000. The instructions for the program each take 4 bytes, so the assembler keeps an internal counter and for each instruction it adds 4 to that counter and uses that number for the address of the next instruction. The new instruction is then placed at that address in the memory and the process is continued so as to allow each subsequent assembly instruction inserted at the next available word boundary. Thus, the first instruction in a program is at address 0x00400000, the second instruction is at address 0x00400004, etc. Note that machine instructions must always start on a word boundary.

A simple example of an assembled program is the following.

```
addi $t0, $zero, 10
addi $t1, $zero, 15
add $t0, $t0, $t1
```

This program is shown below in a MARS screen image. The *Address* column of the grid shows the address of the instruction. In this example, the first instruction is stored at 0x00400000, the second at 0x00400004, and the third at 0x00400008.



Bkpt	Address	Code	Basic
<input type="checkbox"/>	0x00400000	0x2008000a	addi \$8,\$0,0x0000000a 1: addi \$t0, \$zero, 10
<input type="checkbox"/>	0x00400004	0x2009000f	addi \$9,\$0,0x0000000f 2: addi \$t1, \$zero, 15
<input type="checkbox"/>	0x00400008	0x01094020	add \$8,\$8,\$9 3: add \$t0, \$t0, \$t1

Figure 7-1: Instruction addresses for a simple program

So, if all the instructions are real instructions, placing the instructions at the correct address is as simple as adding 4 to each previous instruction. There is a problem with pseudo-operators however, as one pseudo instruction can map to more than one real instruction. For example, a `la`

pseudo instruction maps to 2 instructions and thus takes up 8 bytes. Some instructions, such as immediate instructions which can have either 16-bit or 32-bit arguments, can be different lengths depending on the arguments. Thus, it is important to be able to translate pseudo-operators into real instructions. This is shown in the following example program. Note how the Source column (far right) is translated in real instructions in the Basic column.

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	1: la \$t1, label1
<input type="checkbox"/>	0x00400004	0x34290000	ori \$9,\$1,0x00000000	
<input type="checkbox"/>	0x00400008	0x8d2a0000	lw \$10,0x00000000(\$9)	2: lw \$t2, 0(\$t1)
<input type="checkbox"/>	0x0040000c	0x3c011001	lui \$1,0x00001001	3: lw \$t3, label2
<input type="checkbox"/>	0x00400010	0x8c2b0004	lw \$11,0x00000004(\$1)	
<input type="checkbox"/>	0x00400014	0x014b4020	add \$8,\$10,\$11	4: add \$t0, \$t2, \$t3

Figure 7-2: Instruction addresses for program with pseudo-operators

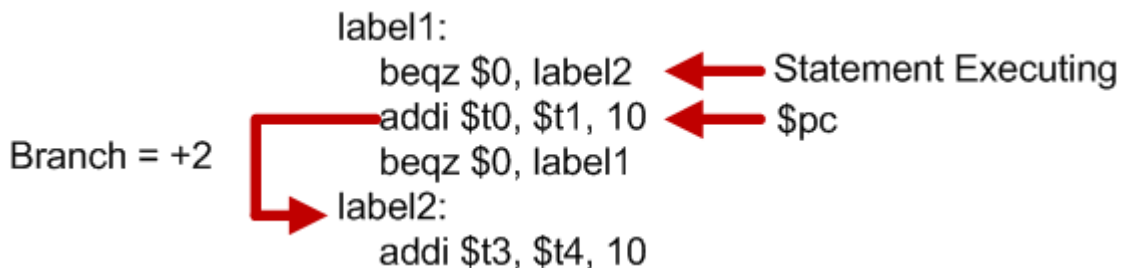
It is important to be able to number the instructions correctly to calculate branch offsets.

Chapter 7.8.2 Value in the \$pc register

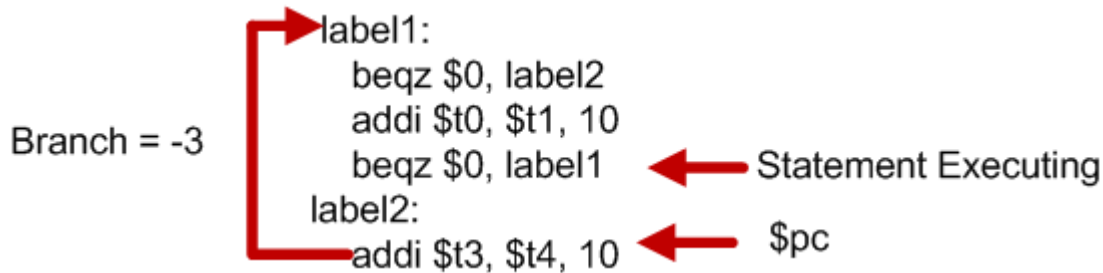
Branches in MIPS assembly work by adding or subtracting the value of the immediate part of the instruction from the \$pc register. So, if a branch is taken, \$pc is updated by the immediate value, and control of the program continues at the labeled instruction.

Earlier in Chapter 3 when it was explained how the \$pc register is used to control the flow of the program, it was apparent that at the start of each instruction the \$pc points to the instruction to execute. So, a reader could think that the value to be incremented by the immediate part of the branch is the address of the current instruction. However, when an instruction executes, the first thing that is done is the \$pc is incremented by 4 to point to the next instruction. This makes sense since usually the program is processed sequentially. However, this means that when a branch is executed the amount which must be added or subtracted will be from the next sequential instruction, not the current instruction.

The following example shows how this works. In the first branch instruction, the branch is to label2. The distance between this instruction and the label consists of 3 real instructions, which is 3 words or 12 bytes, from the current instruction. However, since the \$pc was already incremented to point to the next instruction, the branch will be incremented by 8 bytes, not 12.



The second branch instruction branches backward to label1. In this case, the distance between the instruction and the label is -2 instructions, which is 2 words or 8 bytes, back from the current instruction. However, because the `$pc` is incremented to point to the next instruction, -3 words, or -12 bytes, must be subtracted from the `$pc` in the branch instruction.



The following MARS screen shot shows that this is indeed the branch offsets for each of the branch instructions.

Edit		Execute			
Text Segment					
Bkpt	Address	Code	Basic		
<input type="checkbox"/>	0x00400000	0x10000002	beq \$0,\$0,0x00000002	2: +2	beqz \$0, label2
<input type="checkbox"/>	0x00400004	0x2128000a	addi \$8,\$9,0x0000000a	3: -3	addi \$t0, \$t1, 10
<input type="checkbox"/>	0x00400008	0x1000ffffd	beq \$0,\$0,0xffffffffd	4: -1	beqz \$0, label1
<input type="checkbox"/>	0x0040000c	0x218b000a	addi \$t1,\$t2,0x0000...	6:	addi \$t3, \$t4, 10

Figure 7-3: Branch offset program example

Chapter 7.8.3 How the word boundary effects branching

Remember that the I format instruction uses a 16-bit immediate value. If this was the end of the story, then branches could be up to 64K bytes from the current `$pc`. In terms of instructions, this means that a branch can access instructions that are -8191...8192 real instructions from the current instruction. This may be sufficient for most cases, but there is a way to allow the size of the branch offset to be increased to 2^{18} bits. Remember that all instructions must fall on a word boundary, so the address will always be divisible by 4. This means that the lowest 2 bits in every address must always be "00". Since we know the lowest two bits must always be "00", there is no reason to keep them, and they are dropped. Thus, the branch forward in the previous instruction is 2 ($1000_2 \gg 2 = 0010_2$, or more simply $8/4 = 2$). The branch backward is likewise -3 ($110100_2 \gg 2 = 11101_2$, or more simply $-12/4 = -3$).

Remember that the branch offsets are calculated in bytes and that the two lowest order 00 bits have been truncated and must be reinserted when the branch address is calculated. The reason this caution is given is that the size of the offset in the branch instruction is the number of real instructions the current `$pc` needs to be incremented/decremented. This is just a happy coincidence. It makes calculating the offsets easier as all that needs to be done is count the

number of real instructions between the `$pc` and the label, but that in no way reflects the true meaning of the offset.

Chapter 7.8. 4 Translating branch instructions to machine code

Now that the method of calculating the branch offsets for the branch instructions has been explained, the following program shows an example of calculating the branch offsets in a program. Note that in this example, the trick of dropping the last two bits of the address will be used, so the branch offsets can be used simply by adding/subtracting line numbers. Therefore, the text will read “the `$pc` points to line”, which is correct, as opposed to “the `$pc` contains the address of line”, which would be incorrect.

- 1) Start with the program as written by the programmer. Note that there are 3 branch statements. Only these 3 branch statements will be translated to machine code. In this case the entire program, including comments, is included so that the reader understands the program. However, comments are not kept when a translation to machine code is made, so the subsequent presentations of these programs will drop the comments.

```
# Filename: PrintEven.asm
# Author: Charles Kann
# Date: 12/29/2013
# Purpose: Print even numbers from 1 to 10
# Modification Log:
# 12/29/2013 - Initial release
#
# Pseudo Code
#global main()
#{
# // The following variable can be kept in a save register.
# register int i
#
# // Counter loop from 1 to 10
# for (i = 1; i < 11; i++)
# {
#     if ((i %2) == 0)
#     {
#         print("Even number: " + i)
#     }
# }
#}

.text
.globl main
main:
    # Register Conventions:
    # $s0 - i
    addi $s0, $zero, 1

    BeginForLoop:
    addi $t0, $zero, 11
    slt $t0, $s0, $t0
    beqz $t0, EndForLoop
    addi $t0, $zero, 2
```

```

        div $s0,$t0
        mfhi $t0
        seq $t0, $t0, 0
        beqz $t0, Odd

        la $a0, result
        move $a1, $s0
        jal PrintInt
        jal PrintNewLine

Odd:
        addi $s0, $s0, 1
        b BeginForLoop
EndForLoop:

        jal Exit

.data
        result: .asciiz "Even number: "
        .include "utils.asm"

```

- 2) The next step is to translate all pseudo instructions in the program into real instructions and then number each instruction.

Line #	Label	Statement
1		addi \$16, \$0, 0x00000001
2	BeginForLoop	addi \$8, \$0, 0x0000000b
3		slt \$8, \$16, \$8
4		beq \$8, \$0, ????? (label EndForLoop)
5		addi \$8, \$0, 0x00000002
6		div \$16,\$8
7		mfhi \$8
---		#seq \$t0, \$t0, 0 is 4 real instructions
8		addi \$1, \$0, 0x00000000
9		subu \$8, \$8, \$1
10		ori \$1, \$0, 0x00000001
11		sltu \$8, \$8, \$1
12		beq \$8, \$0, ???? (label Odd)
----		# la \$a0, result is 2 real instructions
13		lui \$1, 0x00001001
14		ori \$4, \$1, 0x00000000
15		addu \$5, \$0, \$16
16		jal ----- (doesn't matter at this point)
17		jal ----- (doesn't matter at this point)
18	Odd	addi \$16, \$16, 0x00000001
19		beq \$0, \$0, ???? (label BeginForLoop)
20	EndForLoop	jal ---- (doesn't matter at this point)

- 3) Calculate the offsets. The first branch instruction, "`beq $t0, EndForLoop`" is at line 4, so the `$pc` when it is executing would point to line 5. The label is at line 20, so the branch offset would be 15. The `beq` instruction is an I type instruction with an opcode of 0x4, so the machine code translation of this instruction is 0x1100000f.

The next branch instruction, "`beq $8, $0, Odd`" is at line 12 and the label `Odd` is at line 18. This means we can subtract 18-13 (as the `$pc` has been updated) and the branch offset is 5. The translation to machine code of this instruction is 0x11000005.

The final branch instruction, "`beq $0, $0, BeginForLoop`" is at line 19 and the label `BeginForLoop` is at line 2. This means that we can subtract 2-20, which gives a branch offset of -18. Note that this branch is negative, so -18 must be a negative 2's complement, or 0xfffffee. The translation to machine code of this instruction is 0x0100ffee.

Chapter 7.8.5 PC-relative addressing

Branch statements use PC-relative addressing since all branch addresses are calculated as an offset from the PC. This is contrasted with Jump (J) instructions which branch to absolute addresses. So, while a branch address must be calculated, a jump address is whatever is in the jump instruction. Both implement branches to different parts of the program, so why are there the two different formats? The first reason is that a J instruction can access the entire `.text` segment of memory. To access the entire `.text` segment requires 26 bits to store the address. This leaves no room for registers which need to be compared, as in the I instruction. The branch instructions can do operations like compare registers but is limited in that the address it contains only has 16 bits. This means that the branch instruction is limited in that it can only access addresses relatively local to the current `$pc`. So, the basic difference is that the jump instruction can access any point in the text memory but cannot be conditional. The branch instruction can be conditional but cannot access all the `.text` memory.

PC-relative addressing has another advantage. The compiler can generate the code for the branch at compile time, as it does not need to know the absolute addresses of the statements, only how far they are from the current `$pc`. This means that the code can easily be moved (or relocated) in the `.text` area and still work correctly. In the example of generating machine code for the branch instruction above, note that it doesn't matter if the code fragment for printing odd/even numbers is at address 0x10010000, 0x10054560, or any other address. The branch is always relative to the `$pc`, so where the code exists is irrelevant to its correct execution. However, because the J instructions all branch to a fixed address, that address must be defined before the program begins to execute and the absolute address cannot be changed (the code cannot be relocated).

So, the difference between branches and jumps comes down to how they are used. Normally when compiling a program, any program control transfer inside of a file (if statements, loops, etc.) is implemented using branch statements. Any program control transfer to a point outside of a file, which means a call to a subprogram, is normally implemented with a jump¹.

¹ Note that to implement jumps is complex and requires a program called a *linkage editor*, or more simply a *linker*. How a linker works is beyond the scope of this text.

Chapter 7.9 Exercises

- 1) When using a branch instruction, the lowest two bits in the offset to add to the `$pc` can be dropped.
 - a) Why can these two bits be dropped?
 - b) What must happen when the branch address is later calculated?
 - c) Do you think the lowest two bits can be dropped in the absolute address for a *jump* instruction? Why or why not?
- 2) In section 7.8.3, it was said that a branch could access addresses that were -8191...8192 distance from the current `$pc`. However, the 2's complement integer has values from -8192...8191.
Why the discrepancy between the value of the 2's complement integer and the size of the branch?
- 3) Explain why program code that uses PC-relative addresses is relocatable, but program code that uses absolute addresses is not relocatable.
- 4) What is the maximum distance from the PC which can be addressed using a branch statement? Give your answer as an address distance.