What you will learn

In this chapter you will learn:

1. 3-Address machines.
2. the difference between real (or native) MIPS operators, and pseudo-operators.
3. introductory pseudo code.
4. addition and subtraction operators.
5. logical (or bit-wise Boolean) operations in MIPS assembly language.
6. pseudo-operators.
7. multiplication and division operators, and how to use them.
8. shifting data in registers, and the different types of shifts in MIPS.

Chapter 3 MIPS arithmetic and Logical Operators

Chapter 3. 1 3-Address machines

The MIPS CPU organization is called a 3-address machine. This is because most operations allow the specification of 3 registers as part of the instruction. To understand this better, consider the following figure, which is like Figure 2.2 except that now the ALU logic portion of the diagram is emphasized.

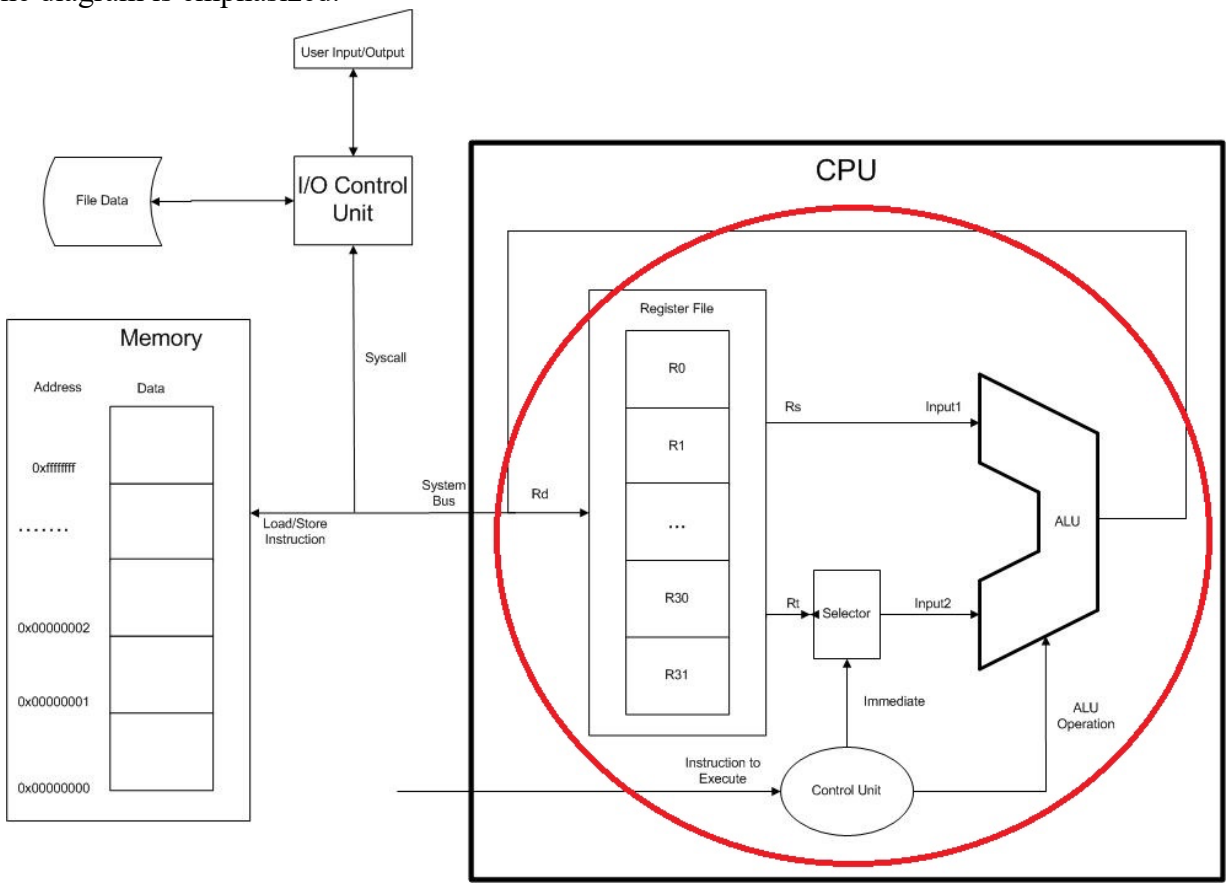


Figure 3-1: MIPS computer architecture

Most MIPS operators take 3 registers as parameters, which is the reason it is called a 3-address machine. They are: 1. the first input to the ALU, always R_s (the first source register); 2. the second input to the ALU, either R_t (the second source register) or an immediate value; and 3. the register to write the result to, R_d (the destination register). All operators, except for the shift operators, in this chapter will follow this format. Even the shift operators will appear to follow this convention, and it will be only when encoding them into machine code in Chapter 4 that they will appear different.

The final thing to note about this diagram is that the ALU takes two inputs. The first is always the R_s register. The second input is determined by the type of operator, of which there will be two types covered in this chapter. The first type of operator is a Register (R) operator. The register operator will always take the second input to the ALU from the R_t register. The format of all R operators will be as follows:

[operator] R_d, R_s, R_t

This syntax means the following:

R_d <- R_s [operator] R_t

The second type of operator is an Immediate (I) operator. There are a limited number of operations for which an "I" operator is defined. When they are defined, the operation they perform will be specified by appending an "i" to the end of the operator's name, and the format will replace the R_t parameter in the R operator with the I (immediate) value which comes from the instruction. The format of all I operators is:

[operator]i R_t, R_s, Immediate value

This syntax means the following:

R_t <- R_s [operator]i Immediate value

This helps explain the difference between a constant and an immediate value. A constant is a value which exists in memory and must be loaded into a register before it is used. An immediate value is defined as part of the instruction, and so is loaded as part of the instruction. No load of a value from memory to a register is necessary, making the processing of an immediate value faster than processing a constant.

There are few real (native) operators in MIPS that do not follow this 3-address format. The Jump (J) operator is the most obvious, but the branch operators and the store operators are also different in that they drop the R_d operator. These will be covered later in the text, and the reason for the format will be obvious.

Other operations in MIPS will appear to not follow these formats. However, these operations are pseudo-operations, and do not exist in the real MIPS instruction set. Some of these pseudo-operations, such as `move` and `li`, have already been seen in Chapter 2. These operators do not require 3 addresses, but they are translated into real operations which do. This will be covered in this chapter.

Chapter 3. 2 Addition in MIPS assembly

Chapter 3.2. 1 Addition operators

There are 4 real addition operators in MIPS assembly. They are:

- `add`, which takes the value of the R_s and R_t registers containing integer numbers, adds the numbers, and stores the value back to the R_d register. The format and meaning are:

  format: `add` R_d, R_s, R_t
  meaning: R_d <- R_s + R_t

- `addi`, which takes the value of R_s and adds the 16-bit immediate value in the instruction, and stores the result back in R_t. The format and meaning are:

  format: `addi` R_t, R_s, Immediate
  meaning: R_t <- R_s + Immediate

- `addu`, which is the same as the `add` operator except that the values in the registers are assumed to be unsigned, or whole, binary numbers. There are no negative values, so the values run from $0...2^{32}-1$. The format and the meaning are the same as the `add` operator above:

  format: `addu` R_d, R_s, R_t
  meaning: R_d <- R_s + R_t

- `addiu`, which is the same as the `addi` operator, but again the numbers are assumed to be unsigned[1]:

  format: `addiu` R_t, R_s, Immediate
  meaning: R_t <- R_s + Immediate

In addition to the real operators, there are a number of pseudo add operators, which are:

- `add` using a 16-bit immediate value. This is shorthand for the add operator to implement an `addi` operator. The same is principal applies for the `addu` if an immediate value is used, and the operator is converted into an `addiu`. The format, meaning, and translation of this instruction is:

  format: `add` R_t, R_s, Immediate
  meaning: R_t <- R_s + Immediate
  translation: `addi` R_t, R_s, Immediate

[1] Note that the instruction `addiu $t1, $t2, -100` is perfectly valid, but as a later example program will show, the results are not what you would expect. The value of -100 is converted into a binary number. When negative numbers are used, the behavior, while defined, is not intuitive. Unsigned numbers means only whole numbers, and when using unsigned numbers it is best to only use unsigned whole numbers.

- `add`, `addi`, `addu`, or `addiu` with a 32-bit immediate value. When considering these operators, it is important to remember that in the real I format instruction the immediate value can only contain 16 bits. So, if an immediate instruction contains a number needing more than 16, the number is loaded in two steps. First load the upper 16 bits into a register using the Load Upper Immediate (`lui`) operator[2], then load the lower 16 bits using the Or Immediate (`ori`) operator. The addition is then done using the R instruction add operator. Thus, the instruction:

  ```
  addi Rt, Rs, (32-bit) Immediate
  ```

  would be translated to:

  ```
  lui $at[3], (upper 16 bits) Immediate        #load upper 16 bits into $at
  ori $at, $at, (lower 16 bits) Immediate   #load lower 16 bits into $at
  add Rt, Rs, $at[4]
  ```

Chapter 3.2. 2 Addition Examples

This section will implement examples of different formats of the add operator and show screenshots from MARS.

```
# File:     Program3-1.asm
# Author:   Charles Kann
# Purpose:  To illustrate some addition operators

# illustrate R format add operator
li $t1, 100
li $t2, 50
add $t0, $t1, $t2

# illustrate add with an immediate.  Note that
# an add with a pseudo instruction translated
# into an addi instruction
addi $t0, $t0, 50
add $t0, $t0, 50

# using an unsign number.  Note that the
# result is not what is expected
# for negative numbers.
addiu $t0, $t2, -100

# addition using a 32 immediate. Note that 5647123
# base 10 is 0x562b13
addi $t1, $t2, 5647123
```

Program 3-1: Addition Examples

[2] The `lui` operator loads the upper 16 bits of a register with the high 16 bits in the immediate value.

[3] The `$at` is the assembler reserved register. The programmer cannot access it directly, it is reserved for the assembler to use as scratch space, as is done here.

[4] Be careful when using immediate values, as they can be either numbers or bit strings. This can create confusion if you mix hex values with decimals (e.g., 0xFFFF and -1). So, the rule will be given that when using arithmetic (add, sub, multi, and div), always use decimal numbers. When using logical operations (and, or, not, xor, etc.), always use hex values. The problems with intermingling these will be explored in the problems at the end of the chapter.

To begin exploring this program, assemble the program and bring up the execute screen, as illustrated in the follow screen capture. One important detail in this screen capture is the representation of the program found in two of the columns. One column is titled *Source*, and this contains the program exactly as you entered it. Another is titled *Basic*, and this contains the source code as it is given to the assembler. There are several changes between the original source code and the Basic code. The Basic code has had all the register mnemonics changed to the register numbers. But the bigger change is that all the instructions using pseudo-operators have been changed into instructions using one or more real operators. For example, in line 14 which has the statement `add $t0, $t0, 50`, the `add` operator has been changed into the `addi` operator, as was discussed in the previous section. The same is true of line 22, where the `addi $t1, $t2, 5647123` instruction has been covered to a `lui`, `ori`, and R format `add` instruction.

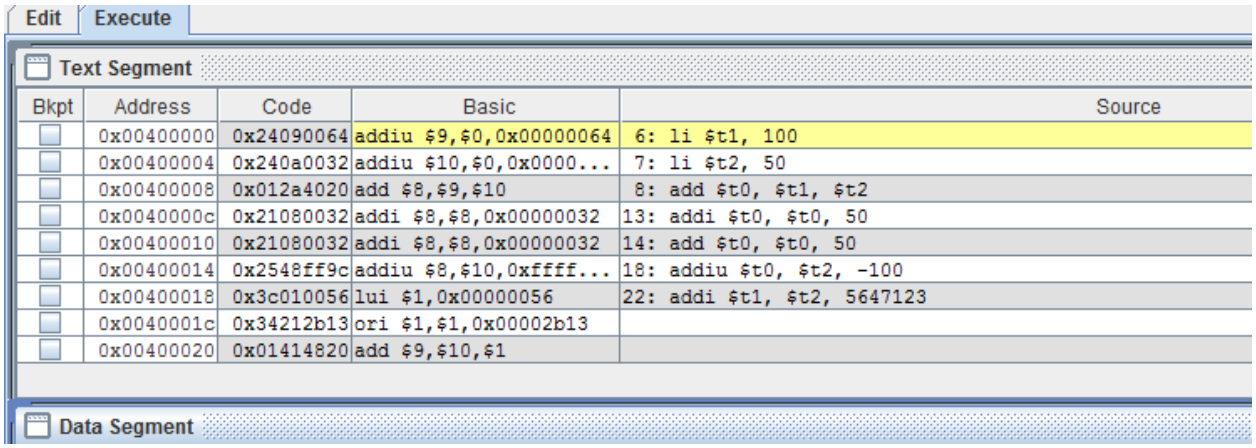


Figure 3-2: Assembled addition example

This output shows that some operators that were used in the previous chapter are pseudo-operators. The `li` operator was suspect because it only had 2 address parameters, a source, and a destination, and indeed it turns out not to be a real operator.

It is often useful to look at the Basic column in MARS to see how your source is presented to the assembler.

Next notice that in Figure 3-2 the first line of the program is highlighted in yellow. This means that the program is ready to execute at the first line in the program. Clicking on the green arrow as shown in Figure 3-3 the program has executed the first line of the program and is waiting to run the second line. As a result of running the first line, the register `$t1` (`$9`) has been updated to contain the value 100 (0x64), which is shown in Figure 3-3.

Continue running the program and you will note that the next line to execute will always be highlighted in yellow and the last register to be changed will be highlighted in green. When line 7 is run, line 8 is yellow and `$t2` (`$10`) contains the value 0x32 (50_{10}) and is highlighted in green. After the addition at line 8, line 13 is highlighted in yellow, and register `$t0` (`$8`) contains 0x96 (or 150_{10}). Continue to step through the program until line 18 highlighted and ready to run. At this point, register `$t0` has the value 0xfa (250_{10}). Once this statement is executed, the value in `$t0` changes from 0xfa changes to 0xfffffce (-50_{10}), not 0x96 (150_{10}) as you might expect.

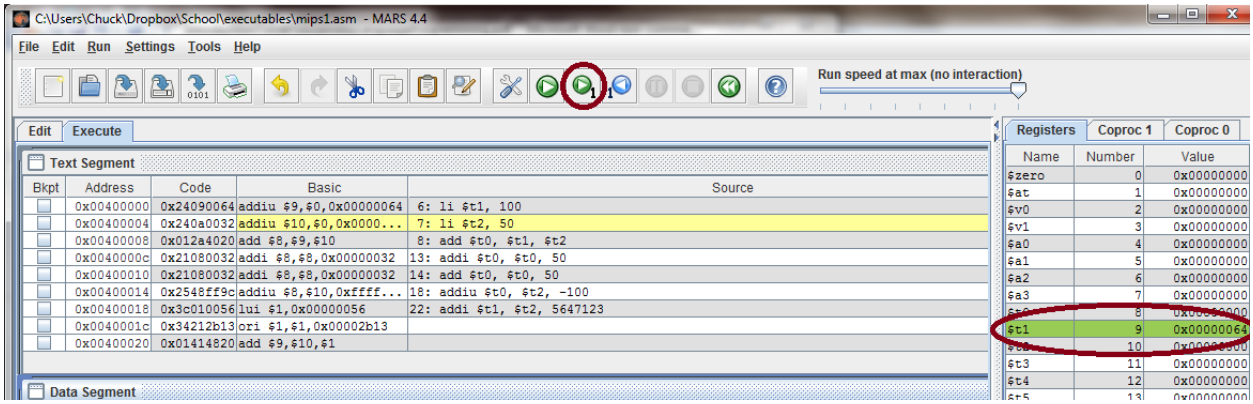


Figure 3-3: Addition Example after running 1 step.

Chapter 3.2. 3 Introduction to pseudocode

Writing a program in assembly language results in very large, complex programs which become hard to write, understand, and debug. HLL were designed to abstract away a large portion of the complexity introduced in writing assembly code. Even though all programs eventually require this level of complexity, a compiler is introduced to translate the relatively simpler structure of a HLL to the more complex assembly language code.

One way to write assembly code would be to first write the code in a HLL and perform the translation to assembly in a similar manner to a compiler. However, HLL must be formally defined to allow a compiler to work properly and some of the translations implemented by a compiler are less than straight forward.

What is really needed is a language which can be used to specify the salient points of code in a higher-level form but with less formality so the translation to assembly is relatively straight forward and simple. The language which is usually used for this is called *pseudocode*. As its name implies, pseudocode is not a formal language, instead it is 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 be done. How the program is implemented is left up to the programmer.

Pseudocode is also useful in that new concepts can be added to the language as needed, so long as the meaning of those constructs is clear to a programmer. This means that the language can be easily changed as the needs of the programmer change.

While there is no one formal definition of pseudocode, this text will give some conventions it will use. Consider the following pseudocode 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);
}
```

This program tells the assembly language programmer to 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. If the `register` modifier is not used, the programmer has a choice to use memory or registers to store the values. In addition, there will be a `volatile` modifier used later, which will mean that the programmer must use a memory variable. Note that the `register` modifier is only a suggestion. Since the number of save registers is limited to 8, there is a possibility that one will not be available to the programmer, and the variable might need to be written to memory.

The next construct from this pseudocode that will be discussed is the `input` and `print`. These are purposefully made to look like HLL methods to be more easily understood by programmers. But the programmer can implement them in any manner they choose, as macros, as subprograms, or directly in the code. They tell the programmer that for the input, a prompt should be written to the console, and a value read from the user and stored. The `print` tells the program to write out a string and append the result of the addition.

Chapter 3.2. 4 Assembly language addition program

The following assembly language program implements the previous pseudocode program.

```
# File name:       Program3-2.psc
# Author:          Charles Kann
# Purpose:         To illustrate how to translate a pseudocode
#                  program into assembly
#
# Pseudo Code
#  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);
#  }

.text
.globl main
main:
        # Register conventions
        # i is $s0
        # j is $s1
        # k is $s2
        #  register int i =
        #   input("Please enter the first value to add: ");
        addi $v0, $zero, 4
        la $a0, prompt1
        syscall
        addi $v0, $zero, 5
        syscall
        move $s0, $v0

        # register int j =
        #      input("Please enter the second value to add: ");
        addi $v0, $zero, 4
```

```
        la $a0, prompt2
        syscall
        addi $v0, $zero, 5
        syscall
        move $s1, $v0

        # register int k = i + j;
        add  $s2, $s1, $s0

        # print("The result is " + k);
        addi $v0, $zero, 4
        la $a0, result
        syscall
        addi $v0, $zero, 1
        move $a0, $s2
        syscall

        #End the program
        addi $v0, $zero, 10
        syscall
.data
prompt1: .asciiz "Please enter the first value to add: "
prompt2: .asciiz "Please enter the second value to add: "
result:  .asciiz "The result is "
```

Chapter 3.2. 5 Assembly language addition program commentary

Since much of this program uses operators and concepts from previous sections, it does not require as many comments. However, the following points are worth noting.

- Preamble comments are still important in any program. It is a very good idea to include the pseudocode as part of the preamble to show how the program was developed.

- The `move` operator is strange since it has only two parameters. This would lead one to believe that it is probably a pseudo-operator and it is:
  `move $t0, $t1 -> add $t0, $t1, $zero`

Chapter 3. 3 Subtraction in MIPS assembly

Subtraction in MIPS assembly is like addition with one exception. The `sub`, `subu, and subui` behave like the `add`, `addu,` and `addui` operators. The only major difference with subtraction is that the `subi` is not a real instruction. It is implemented as a pseudo instruction with the value to subtract loaded into the `$at` register, and then the R instruction `sub` operator is used. This is the only difference between addition and subtraction.

- `sub`, which takes the value of the R_s and R_t registers containing integer numbers, subtracts the numbers, and stores the value back to the R_d register. The format and meaning are:

  format: `sub` R_d, R_s, R_t
  meaning: R_d <- R_s - R_t

- `sub` pseudo-operator, which takes the value of R_s, subtracts the 16-bit immediate value in the instruction, and stores the result back in R_t. The format, meaning, and translation are:

  format: `sub Rt, Rs, Immediate`
  meaning: `Rt <- Rs - Immediate`
  translation:
  ```
  addi $at, $zero, Immediate
  sub Rt, Rs, $at
  ```

- `subi` pseudo-operator, which takes the value of R_s, subtracts the 16-bit immediate value in the instruction, and stores the result back in R_t. The format, meaning, and translation are:

  format: `subi Rt, Rs, Immediate`
  meaning: `Rt <- Rs - Immediate`
  translation:
  ```
  addi $at, $zero, Immediate
  sub Rt, Rs, $at
  ```

- `subu`, which is the same as the `add` operator, except that the values in the registers are assumed to be unsigned, or whole, binary numbers. There are no negative values, so the values run from $0 \ldots 2^{32}-1$. The format and the meaning are the same as the `add` operator above:

  format: `subu Rd, Rs, Rt`
  meaning: `Rd <- Rs + Rt`

- `subiu` pseudo-operator, which is the same as the `addi` operator, but again the numbers are assumed to be unsigned:

  format: `subiu Rt, Rs, Immediate`
  meaning: `Rt <- Rs + Immediate`
  translation:
  ```
  addi $at, $zero, Immediate
  subu Rt, Rs, $at
  ```

In addition to the real operators, there are several pseudo sub operators, which use 32-bit immediate values. The 32-bit values are handled exactly as with the `add` instructions, with a sign extension out to 32 bits.

Chapter 3. 4 Multiplication in MIPS assembly

Multiplication and division are more complicated than addition and subtraction and require the use of two new, special purpose registers, the `hi` and `lo` registers. The `hi` and `lo` registers are not included in the 32 general purpose registers which have been used up to this point, and so are not directly under programmer control. These sections on multiplication and division will look at the requirements of the multiplication and division operations that make them necessary.

Multiplication is more complicated than addition because the result of multiplication can require up to twice as many digits as the input values. To see this, consider multiplication in base 10, 9x9=81 (2 one-digit numbers yield a two-digit number), and 99x99=9801 (2 two-digit numbers yield a 4-digit number). As this illustrates, the results of a multiplication require up to twice as

many digits as in the original numbers being multiplied. This same principal applies in binary. When two 32-bit numbers are multiplied, the result requires a 64-bit space to store the results.

Since multiplication of two 32-bit numbers requires 64-bits, two 32-bit registers are required. All computers require two registers to store the result of a multiplication, though the actual implementation of those two registers is different. In MIPS, the `hi` and `lo` registers are used, with the `hi` register being used to store the 32-bit larger/most significant part of the multiplication, and the `lo` register being used to the store the 32-bit smaller/least significant part.

In MIPS, all integer values must be 32 bits, so if there is a valid answer, it must be contained in the lower 32 bits of the answer. To implement multiplication in MIPS, the two numbers must be multiplied using the `mult` operator and the valid result moved from the `lo` register. This is shown in the following code fragment which multiplies the value in `$t1` by the value in `$t2` and stores the result in `$t0`.

```
mult $t1, $t2
mflo $t0
```

However, what happens if the result of the multiplication is too big to be stored in a single 32-bit register? Again, consider base 10 arithmetic. 3*2=06, and the larger part of the answer is 0. However, 3*6=18, and the larger part of the answer is non-zero. This is true of MIPS multiplication as well. When two positive numbers are multiplied, if the `hi` register contains nothing but 0's then there is no overflow as the multiplication did not result in any value in the larger part of the result. If the `hi` register contains any values of 1, then the result of the multiplication did have an overflow, as part of the result is contained in the larger part of the result. This is shown in the two examples, 3*2=06, and 3*6=18, below.

```
  0011          0011
* 0010        * 0110
------        ------
0000 0110     0001 0010
```

A simple check for overflow is needed when two positive numbers are multiplied to see if the `hi` register is all 0's, if so, the result did not overflow, otherwise there was overflow. This is fine for two positive or two negative numbers, but what if the input values are mixed? For example, 2*(-3) = -6, and 2*(-8) = -16. To understand what would happen, these problems will be implemented using 4-bit registers.

Remember that 4-bit registers can contain integer values from -8...7. So, the multiplication of 2*(-3) and 2*(-6) in 4-bits with an 8-bit result is shown below:

```
  0010          0010
* 1101        * 1010
------        ------
1111 1010     1110 1110
```

In the first example, the high 4-bits are 1111, which is the extension of the sign for -6. This says that the example did not overflow. In the second example, the high 4-bits are 1110. Since all 4 bits are not 1, they cannot be the sign extension of a negative number, and the answer did overflow. So, an overly simplistic view might say that if the high order bits are all 0's or all 1's,

there is no overflow. While this is a necessary condition to check for overflow, it is not sufficient. To see this, consider the result of 2*(-6).

```
    0010
  * 1010
1111 0100
```

Once again, the high 4-bits are 1111, so it looks like there is not an overflow. But the difficulty here is that the low 4 bits show a positive number, so 1111 indicates that the lowest 1 (the one underlined), is really part of the multiplication result, and not an extension of the sign. This result does show overflow. So, to show overflow the result contained in the `hi` register must match all 0's or all 1's and must match the high order (sign) bit of the `lo` register.

Now that the fundamentals of integer multiplication have been covered, there are five MIPS multiplication operators which will be looked at. They are:

- `mult`, which multiplies the values of R_s and R_t and saves it in the `lo` and `hi` registers. The format and meaning of this operator are:

  format: `mult` R_s, R_t
  meaning: `[hi,lo] <-` R_s `*` R_t

- `mflo`, which moves the value from the `lo` register into the R_d register. The format and meaning of this operator are:

  format: `mflo` R_d
  meaning: R_d `<- lo`

- `mfhi`, which moves the value from the `hi` register into the R_d register. The format and meaning of this operator are:

  format: `mfhi` R_d
  meaning: R_d `<- hi`

- `mul`, which multiples the values in R_s and R_t, and stores them in R_d. This differs from the `mult` above because it has three arguments. The format and meaning of this operator are:

  format: `mul` R_d, R_s, R_t
  meaning: R_d `<-` R_s `*` R_t

- `mulo` pseudo-operator, which multiples the values in R_s and R_t, and stores them in R_d, checking for overflow. If overflow occurs an exception is raised, and the program is halted with an error. The format and meaning of this operator are:

  format: `mulo` R_d, R_s, R_t
  meaning: R_d `<-` R_s `*` R_t

- Note that both the `mul` and `mulo` operators have an immediate pseudo-operator implementation. The format, meaning, and translation of the pseudo-operators is as follows:

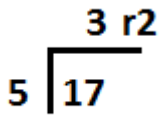
format: `mul` R_d, R_s, Immediate
meaning: R_d <- R_s * Immediate
translation:
```
addi $Rt, $zero, Immediate
mul Rd, Rs, Rt
```

format: `mulo` R_d, R_s, Immediate
meaning: R_d <- R_s * Immediate
translation:
```
addi $Rt, $zero, Immediate
mulo Rd, Rs, Rt
```

Chapter 3. 5 Division in MIPS Assembly

Division, like multiplication, requires two registers to produce an answer. The reason for this has to do with how the hardware calculates the result and is harder to explain without considering the hardware used, which is beyond the scope of this textbook. The reader is thus asked to just believe that two registers are needed, and that MIPS will again use the registers `hi` and `lo`.

To understand division, we will again begin with a base 10 example. Remember how division was done when it was introduced to you in elementary school. The result of 17 divided by 5 would be the following:



In this equation, the value 5 is called the divisor, the 17 is the dividend, 3 is the quotient, and 2 is the remainder.

In MIPS, when integer division is done, the `lo` register will contain the quotient, and the `hi` register will contain the remainder. This means that in MIPS integer arithmetic when the quotient is taken from the low register the results will be truncated. If the programmer wants to round the number, this can be implemented using the remainder.

Now that the fundamentals of integer division have been covered, there are two MIPS division operators that will be looked at. They are:

- `div`, which has 3 formats. The first format is the only real format of this operator. The operator divides R_s by R_t and stores the result in the [`hi`,`lo`] register pair with the quotient in the `lo` and the remainder in the `hi`.

The format and meaning of this operator are:

```
format:      div Rs, Rt
meaning:     [hi,lo] <- Rs / Rt
```

The second format of `div` is a pseudo instruction. It is a 3-address format but is still a pseudo instruction. In addition to a division instruction, it also checks for a zero divide. The specifics of the check will not be covered now as it involves `bne` and `break` instructions. The format, meaning, and translation of the pseudo-operator is as follows:

```
format:      div Rd, Rs, Rt
meaning:     [if Rt != 0] Rd <- Rs / Rt
             else break
translation: bne Rt, $zero, 0x00000001
             break
             div Rs, Rt
             mflo Rd
```

The third format of the div operator is a pseudo instruction. The format, meaning, and translation of the pseudo-operator is as follows:

```
format:      div Rd, Rs, Immediate
meaning:     Rd <- Rs / Immediate
translation: addi $Rt, $zero, Immediate
             div Rs, Rt
             mflo Rd
```

- `rem` (remainder), which has 2 formats. There are only pseudo formats for this instruction. The first format of the `rem` operator is a pseudo instruction. The format, meaning, and translation of the pseudo-operator is as follows:

```
format:      rem Rd, Rs, Rt
meaning:     [if Rt != 0] Rd <- Rs % Rt
             else break
translation: bne Rt, $zero, 0x00000001
             break
             div Rs, Rt
             mfhi Rd
```

  The second format of the `rem` operator is also a pseudo instruction. The format, meaning, and translation of the pseudo-operator is as follows:

```
format:      rem Rd, Rs, Immediate
meaning:     Rd <- Rs / Immediate
translation: addi $Rt, $zero, Immediate
             div Rs, Rt
             mfhi Rd
```

Chapter 3.5. 1 Remainder operator, even/odd number checker

Several interesting algorithms are based on the remainder operator. For example, this section presents a short program which checks if a number is odd or even. Note that since branching has not yet been covered this program will print a 0 if the number is even, and 1 if the number is odd.

The algorithm checks the remainder of a division by 2. If the number is evenly divisible by 2, the remainder will be 0 and the number is even. The pseudocode for this algorithm uses the "%" or modulus operator to obtain the remainder.

```
main
{
    int i = prompt("Enter your number");
    int j = i % 2;
    print("A result of 0 is even, a result of 1 is odd: result = " + j;
}
```

Chapter 3.5. 2 Remainder operator, even/odd number checker

The following is the MIPS implementation of the even/odd checker. To find the remainder the `div` operator is used to divide by 2 and the remainder retrieved from the `hi` register.

```
# File:       Program3-3.asm
# Author:     Charles W. Kann
# Purpose:    A user enters a number and prints 0 if even, 1 if odd
.text
.globl main
main:
   # Get input value
   addi $v0, $zero, 4      # Write Prompt
   la $a0, prompt
   syscall
   addi $v0, $zero, 5      # Retrieve input
   syscall
   move $s0, $v0

   # Check if odd or even
   addi $t0, $zero, 2      # Store 2 in $t0
   div $t0, $s0, $t0       # Divide input by 2
   mfhi $s1                # Save remainder in $s1

   # Print output
   addi $v0, $zero, 4      # Print result string
   la $a0, result
   syscall
   addi $v0, $zero, 1      # Print result
   move $a0, $s1
   syscall

   #Exit program
   addi $v0, $zero, 10
   syscall

.data
prompt: .asciiz "Enter your number: "
result: .asciiz "A result of 0 is even, 1 is odd: result =  "
```

Program 3-2: Even/odd number checking program

Chapter 3. 6 Solving arithmetic expressions in MIPS assembly

Using the MIPS arithmetic operations covered so far, a program can be created to solve equations. For example, the following pseudocode program, where the user is prompted for a value of x and the program prints out the results of the equation $5x^2 + 2x + 3$.

```
main
{
     int x = prompt("Enter a value for x: ");
     int y = 5 * x * x + 2 * x + 3;
     print("The result is: " + y);
}
```

This program is implemented in MIPS below.

```
# File:         Program3-3.asm
# Author:       Charles Kann
# Purpose:      To calculate the result of 5*x^ + 2*x + 3

.text
.globl main
main:
   # Get input value, x
   addi $v0, $zero, 4
   la $a0, prompt
   syscall
   addi $v0, $zero, 5
   syscall
   move $s0, $v0

   # Calculate the result of 5*x*x + 2* x + 3 and store it in $s1.
   mul $t0, $s0, $s0
   mul $t0, $t0, 5
   mul $t1, $s0, 2
   add $t0, $t0, $t1
   addi $s1, $t0, 3

   # Print output
   addi $v0, $zero, 4      # Print result string
   la $a0, result
   syscall
   addi $v0, $zero, 1      # Print result
   move $a0, $s1
   syscall

   #Exit program
   addi $v0, $zero, 10
   syscall

.data
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
```

Program 3-3: Program to calculate 5*x^2 + 2*x + 3

Chapter 3. 7 Division and accuracy of an equation

One thing to always keep in mind when using division with integers in any language (including Java, C/C++, etc.) is that the results are truncated. This can lead to errors and different answers depending on the order of evaluation of the terms in the equation. For example, most 5th graders know that "(10/3) * 3 = 10", as the 3's should cancel. However, in integer arithmetic the result of "10/3 = 3", and so "(10 / 3) * 3 = 9" (not 10). If you reverse the order of the operations, you will find that "(10 * 3) / 3 = 10". This is shown in the following program.

```
# File:     Program3-4.asm
# Author:   Charles Kann
# Purpose:  To show the difference in result if
#           ordering of multiplication and division
#           are reversed.

.text
.globl main
main:
    addi $s0, $zero, 10 # Store 10 and 3 in registers $s0 and $s1
    addi $s1, $zero, 3

    div $s2, $s0, $s1   # Write out (10/3) * 3
    mul $s2, $s2, $s1
    addi $v0, $zero, 4
    la $a0, result1
    syscall
    addi $v0, $zero, 1
    move $a0, $s2
    syscall

    mul $s2, $s0, $s1 # Write out (10*3)/3
    div $s2, $s2, $s1
    addi $v0, $zero, 4
    la $a0, result2
    syscall
    addi $v0, $zero, 1
    move $a0, $s2
    syscall

    addi $v0, $zero, 10 #Exit program
    syscall

.data
result1: .asciiz "\n(10/3)*3 = "
result2: .asciiz "\n(10*3)/3 = "
```

Program 3-4: Program to show order of operations matters

There are times (for example, when calculating a parent node in a Complete Binary Tree, which is covered in most textbooks on Data Structures) that division using truncation is desired. However, in general the simple rule to follow is when there is a mix of operations including division with integer numbers, code the expression to do the multiplication, addition, and subtraction before doing any division. This preserves the greatest accuracy of the result. This principal is true of integer arithmetic in any language, including any HLL.

Chapter 3. 8 Logical operators

Most programmers have seen logical operators in HLL for Boolean data types. As was covered earlier in this text, there are often two forms of these operators, the logical (or short-circuiting) operators and the bitwise operators. MIPS only implements bitwise operators, but they are called *logical* operators. Since all Boolean operations can be implemented with only 3 operations, the AND, OR, and NOT operations, this section will present these 3 operations. In addition, the XOR operation will be included for convenience because it is often easier to state a logical expression using XOR than to do so using AND, OR, and NOT.

The logic operators covered are:

- `and` operator, which has three formats. The first is the only real format of this operator. The operator does a bitwise AND of R_s and R_t and stores the result R_d register. The format and meaning of this operator are:

  format: `and Rd, Rs, Rt`
  meaning: `Rd <- Rs AND Rt`

  The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is immediate value, and so this is just a shorthand for implementing the `andi` operator. The format, meaning, and translation of the pseudo-operators is as follows:

  format: `and Rt, Rs, Immediate`
  meaning: `Rt <- Rs AND Immediate`
  translation: `andi Rt, Rs, Immediate`

  The third format of the `and` operator is also a pseudo instruction and strange in that only logical operators have this format. In this instruction, R_s and R_t are assumed to be the same register, and the 3rd operator is immediate value The format, meaning, and translation of the pseudo-operators is as follows:

  format: `and Rs, Immediate`
  meaning: `Rs <- Rs AND Immediate`
  translation: `andi Rs, Rs, Immediate`

- `andi` operator. The operator does a bitwise AND of R_s and an immediate value and stores the result R_t register. The format and meaning of this operator are:

  format: `andi Rt, Rs, Immediate`
  meaning: `Rt <- Rs AND Immediate`

  The shorthand with a single register also applies to the `andi` instruction.

  format: `andi Rs, Immediate`
  meaning: `Rs <- Rs AND Immediate`
  translation: `andi Rs, Rs, Immediate`

- `or` operator, which has three formats. The first is the only real format of this operator. The operator does a bitwise OR of R_s and R_t and stores the result R_d register. The format and meaning of this operator are:

  format: `or` R_d, R_s, R_t
  meaning: R_d <- R_s OR R_t

  The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is an immediate value, and so this is just a shorthand for implementing the `ori` operator. The format, meaning, and translation of the pseudo-operators is as follows:

  format: `or` R_t, R_s, Immediate
  meaning: R_t <- R_s OR Immediate
  translation: `ori` R_t, R_s, Immediate

  The shorthand with a single register also applies to the `or` instruction.

  format: `or` R_s, Immediate
  meaning: R_s <- R_s OR Immediate
  translation: `ori` R_s, R_s, Immediate

- `ori` operator. The operator does a bitwise OR of R_s and an immediate value and stores the result R_t register. The format and meaning of this operator are:

  format: `ori` R_t, R_s, Immediate
  meaning: R_t <- R_s OR Immediate

  The shorthand with a single register also applies to the `ori` instruction.

  format: `ori` R_s, Immediate
  meaning: R_s <- R_s OR Immediate
  translation: `ori` R_s, R_s, Immediate

- `xor` operator, which has three formats. The first format is the only real format of this operator. The operator does a bitwise OR of R_s and R_t and stores the result R_d register. The format and meaning of this operator are:

  format: `xor` R_d, R_s, R_t
  meaning: R_d <- R_s XOR R_t

  The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is an immediate value, and so this is just a shorthand for implementing the `ori` operator. The format, meaning, and translation of the pseudo-operators is as follows:

  format: `xor` R_t, R_s, Immediate
  meaning: R_t <- R_s XOR Immediate
  translation: `xori` R_t, R_s, Immediate

The shorthand with a single register also applies to the `xor` instruction.

format: `xor Rs, Immediate`
meaning: `Rs <- Rs AND Immediate`
translation: `xori Rs, Rs, Immediate`

- `xori` operator. The operator does a bitwise OR of `Rs` and an immediate value and stores the result `Rt` register. The format and meaning of this operator are:

  format: `xori Rt, Rs, Immediate`
  meaning: `Rt <- Rs XOR Immediate`

  The shorthand with a single register also applies to the `xori` instruction.

  format: `xori Rs, Immediate`
  meaning: `Rs <- Rs XOR Immediate`
  translation: `xori Rs, Rs, Immediate`

- `not` operator. The operator does a bitwise NOT (bit inversion) of `Rs` and stores the result `Rt` register. The format and meaning of this operator are:

  format: `not Rs, Rt`
  meaning: `Rs <- NOT(Rt)`
  translation: `nor Rs, Rt, $zero`

In addition to the real operators, there are several pseudo sub operators which use 32-bit immediate values. The 32-bit values are handled exactly as with the add instructions with a sign extension out to 32 bits.

Chapter 3. 9 Using logical operators

Most students have only seen logical operators used as branch conditions, such as `if`, `while`, and `for` statements, and so they cannot see why these operators are useful. This section will show some uses of these operators.

Chapter 3.9. 1 Storing immediate values in registers

In MARS, the `li` instruction is translated into an "`addui Rd, $zero, Immediate`" instruction. This is perfectly valid, however when doing addition there always has to be propagation of a carry-bit, and so addition can take a relatively long time (O(log n)…O(n)). However the OR operation is a bitwise operation and does not have any carry bits and can be executed in O(1) time. In MARS the OR is not faster than addition but in some architectures the OR is in fact faster.

If the OR is faster than addition, the `li` instruction can be implemented as "`ori Rd, $zero, Immediate`" with some advantage.

Chapter 3.9. 2 Converting a character from upper case to lower case

Ch 1 showed that in ASCII the difference between upper- and lower-case letters is the 0x20 bit. An upper-case letter can be converted to lower-case by setting this bit to 1 and vice versa. Many novice programmers may implement a solution by adding 0x20. But the letter is already upper-case, the result is no longer a letter.

To do this conversion correctly, the letters should be OR'ed with 0x20. If this is done, letters which are already lower-case are not changed. This is illustrated in the following program.

```
# File:       Program3-5.asm
# Author:     Charles Kann
# Purpose:    This shows that adding 0x20 to a character can result in an
#             error when converting case, but or'ing 0x20 always works.
.text
.globl main
main:
    # Show adding 0x20 only works if the character is upper case.
    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero, 0x41   # Load the character "A"
    addi $a0, $t0, 0x20    # Convert to "a" by adding
    ori $v0, $zero, 11     # Print the character
    syscall

    ori $v0, $zero, 4
    la $a0, output2
    syscall
    ori $t0, $zero,0x61    # Load the character "a"
    addi $a0, $t0, 0x20    # Attempt to convert to lower case
    ori $v0, $zero, 11     # Print the character, does not work
    syscall

    # Show or'ing 0x20 works if the character is upper or lower case.
    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero, 0x41   # Load the character "A"
    ori $a0, $t0, 0x20     # Convert to "a" by adding
    ori $v0, $zero, 11     # Print the character
    syscall

    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero,0x61    # Load the character "a"
    ori $a0, $t0, 0x20     # Attempt to convert to lower case
    ori $v0, $zero, 11     # Print the character, does not work
    syscall
    ori $v0, $zero, 10     # Exit program
    syscall

.data
output1: .asciiz "\nValid conversion: "
output2: .asciiz "\nInvalid conversion, nothing is printed: "
```

Program 3-5: Letter case conversion in assembly

Chapter 3.9. 3 Reversible operations with XOR

Often it is nice to be able to invert the bits in a value in a way that they are easily translated back to the original value. Inverting bits and restoring them is shown in the program below. One use of this is to swap two values without using a temporary storage space.

```
# File:     Program3-6.asm
# Author:   Charles Kann
#Purpose:   To show the XOR operation is reversible
.text
.globl main
main:
    ori $s0, $zero, 0x01234567 # the hex numbers

    # Write out the XOR'ed value
    la $a0, output1
    li $v0, 4
    syscall
    xori $s0, $s0,  0xffffffff # the results in $s0 will be fedcba98
    move $a0, $s0
    li $v0, 34
    syscall

    # Show the original value has been restored.
    la $a0, output2
    li $v0, 4
    syscall
    xori $s0, $s0,  0xffffffff # the results in $s0 will be 01234567
    move $a0, $s0
    li $v0, 34
    syscall

    ori $v0, $zero, 10   # Exit program
    syscall
.data
    output1: .asciiz "\nAfter first xor:  "
    output2: .asciiz "\nAfter second xor: "
```

Chapter 3. 10 Shift Operations

The final topic covered in this chapter are shift operations. Shift allow bits to be moved around inside of a register. There are many reasons to do this, particularly when working with low level programs such as device drivers. The major reason this might be done in a HLL is multiplication and division, but as was stated earlier, multiplication and division using constants should not be implemented by the programmer as the compiler will automatically generate the best code for the situation. So, these operations are difficult to justify in terms of higher-level languages. These operations will simply be presented here with an example of multiplication and division by a constant. The reader will have to trust that these operators are useful in assembly and will be used in future chapters.

There are 2 directions of shifts, a right and left shift. The right shift moves all bits in a register some specified number of spaces to the right and the left shift moves bits to the left.

The bits that are used to fill in the spaces as the shift occurs are determined by the type of shift. A logic shift uses zeros (0) to replace the spaces. The arithmetic shift replaces the spaces with the high order (left most) bits. In an integer, the high order bit determines the sign of the number, so shifting the high order keeps the correct sign for the number[5]. Note that the sign only matters when doing a right shift, so only the right shift will have an arithmetic shift.

Finally, there is a circular shift (called a rotate in MIPS) that shifts in the bit that was shifted out on the opposite side. For example, when the 4-bit value 0011 is rotated right, a 1 is returned is shifted into the left most bit, giving 1001. Likewise, when the 4-bit value 0011 is rotated left, a 0 is returned and is shifted into the right most bit, giving 0110.

The main use of a rotate command is to allow the programmer to look at each of the 32 bits in a value one at a time and at the end the register contains the original value. The `rol` and `ror` operators (rotate left/right) are pseudo-operators and a circular shift can be implemented using combinations of left and right logical shifts and an OR operation.

The following examples show how each of these shifts work. The numbers here are shown in hex, but are converted to binary to do the shifts.

```
Shift left logical 2 spaces: 0x00000004 -> 0x00000010
Shift right logical 3 spaces: 0x0000001f -> 0x0000003
Shift right arithmetic 3 spaces: 0x0000001f -> 0x00000003
Shift right arithmetic 2 spaces: 0xfffffffe1 -> 0xfffffff8
Rotate right 2 spaces: 0xfffffffe1 -> 0x7fffffff8
Rotate left 2 space: 0xfffffffe1 -> 0xffffff87
```

The following are the shift operations provided in MIPS.

- `sll` (shift left logical) operator. The operator shifts the value in R_t shift amount (shamt) bits to the left, replacing the shifted bits with 0's, and storing the results in R_d. Note that the registers R_d and R_t are used. The numeric value in this instruction is not an immediate value, but a shift amount. Shift values are limited to the range 0…31 in all the following instructions.

  format: `sll` R_d, R_t, shamt
  meaning: R_d <- R_t << shamt

- `sllv` (shift left logical variable) operator. The operator shifts the value in R_t bits to the left by the number in R_s, replacing the shifted bits with 0's. The value in R_s should be limited to the range 0…31, but the instruction will run with any value.

  format: `sllv` R_d, R_t, R_s
  meaning: R_d <- R_t << R_s

[5] This is why it is in some sense incorrect to call the high order bit of an integer a "sign bit". To do division the high order (left-most) bit is replicated to the right. If this high order bit was just a sign bit, it would not be replicated, as the one sign bit only specifies the sign.

- `srl` (shift right logical) operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with 0's, and storing the results in R_d.

  format: `srl` R_d, R_t, shamt
  meaning: R_d <- R_t >> shamt

- `srlv` (shift right logical variable) operator. The operator shifts the value in R_t bits to the right by the number in R_s, replacing the shifted bits with 0's. The value in R_s should be limited to the range 0…31, but the instruction will run with any value.

  format: `srlv` R_d, R_t, R_s
  meaning: R_d <- R_t >> R_s

- `sra` (shift right arithmetic) operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with sign bit for the number, and storing the results in R_d.

  format: `sra` R_d, R_t, shamt
  meaning: R_d <- R_t >> shamt

- `srav` (shift right arithmetic variable) operator. The operator shifts the value in R_t to the right by the number in R_s, replacing the shifted bits the sign bit for the number. The value in R_s should be limited to the range 0…31, but the instruction will run with any value.

  format: `srav` R_d, R_t, R_s
  meaning: R_d <- R_t >> R_s

- `rol` (rotate left) pseudo-operator. The operator shifts the value in R_t the shift amount (shamt) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in R_d.

  format: `rol` R_d, R_t, shamt
  meaning: R_d[shamt..0] <- R_t[31...31-shamt+1],
  R_d[31...shamt] <- R_t[31-shamt..0],
  translation:
  ```
  srl $at, $Rt, shamt
  sll $Rd, $Rt, shamt
  or $Rd, $Rd,  $at
  ```

- `ror` (rotate right) pseudo-operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in R_d.

  format: `ror` R_d, R_t, shamt
  meaning: R_d[31-shamt...shamt] <- R_t[31..shamt],
  R_d[31..31-shamt+1] <- R_t[shamt-1..0],
  translation:
  ```
  srl $at, $Rt, shamt
  sll $Rd, $Rt, shamt
  or $Rd, $Rd, $at
  ```

Chapter 3.10. 1 Program illustrating shift operations

The following program illustrates the shift operations from the previous section.

```
# File:     Program3-6.asm
# Author:   Charles Kann
# Purpose:  To illustrate various shift operations.

.text
.globl main
main:
                        #SLL example
   addi $t0, $zero, 4
   sll  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result1
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

                        #SRL example
   addi $t0, $zero, 16
   srl  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result2
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

                        #SRA example
   addi $t0, $zero, 34
   sra  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result3
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

                        #SRA example
   addi $t0, $zero, -34
   sra  $s0, $t0, 2     # sra 2 bits, which is division by 4
   addi $v0, $zero, 4   # Output the result
   la $a0, result4
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

                        #rol example
   ori $t0, $zero, 0xfffffffe1
   ror $s0, $t0, 2
   li $v0, 4
   la $a0, result6
```

```
    syscall
    li $v0, 34
    move $a0, $s0
    syscall

                        #ror example
    ori $t0, $zero, 0xfffffffe1
    rol $s0, $t0, 2
    li $v0, 4
    la $a0, result6
    syscall
    li $v0, 34
    move $a0, $s0
    syscall

    addi $v0, $zero, 10 # Exit program
    syscall

.data
result1: .asciiz "\nshift left logical 4 by 2 bits is "
result2: .asciiz "\nshift right logical 16 by 2 bits is "
result3: .asciiz "\nshift right arithmetic 34 by 2 bits is "
result4: .asciiz "\nshift right arithmetic -34 by 2 bits is "
result5: .asciiz "\nrotate right 0xfffffffe1 by 2 bits is "
result6: .asciiz "\nrotate left 0xfffffffe1 by 2 bits is "
```

Program 3-6: Program illustrating shift operations

Chapter 3. 11 Summary

This chapter was covered MIPS operators and how to use these operators in instructions. Some simple programs were given to illustrate the operators, but they were very simplistic in that they only used sequences, as branches and loops are not yet covered. In addition, only register memory was used, which greatly limited the types of programs that could be written. These limitations will be dealt with in subsequent chapters.

Chapter 3. 12 Exercises

1) Implement a program to do a bitwise complement (NOT) of an integer number entered by the user. You should use the `xori` operator to implement the not, do not use the `not` operator. Your program should include a proper and useful prompt for input and print the results in a meaningful manner.

2) Implement a program to calculate the 2's complement of a number entered by the user. The program should only use the `xori` and `addi` operators. Your program should include a proper and useful prompt for input and print the results in a meaningful manner.