MIPS Assembly Language Programming

Bob Britton

Lesson #3

Benefits of Studying Assembly Language Programming

Obtain Insights into writing more efficient code

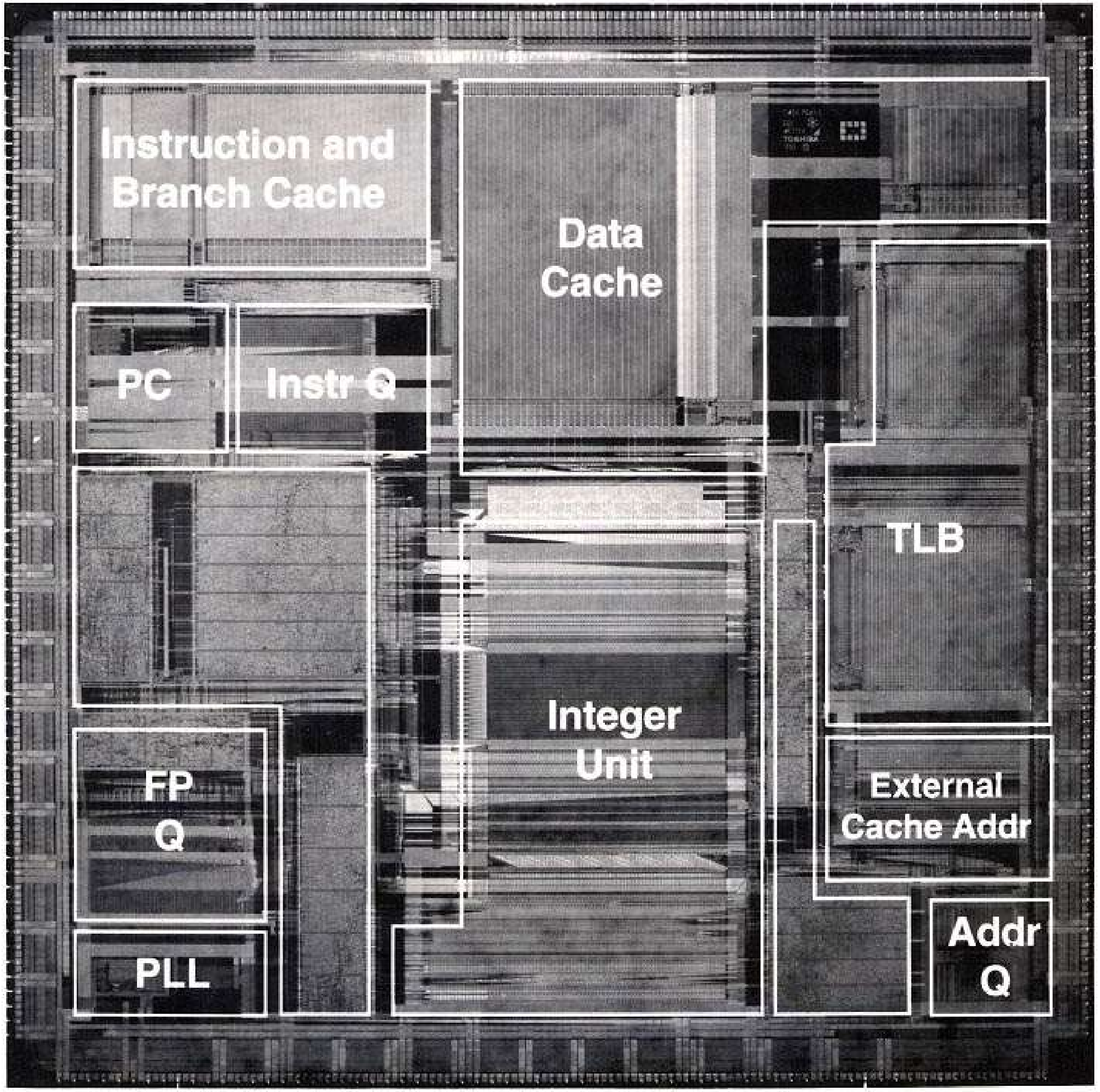
Will become familiar with what compilers do

Acquire an understanding of how computers are built

Open new opportunities in the field of embedded processors

MIPS

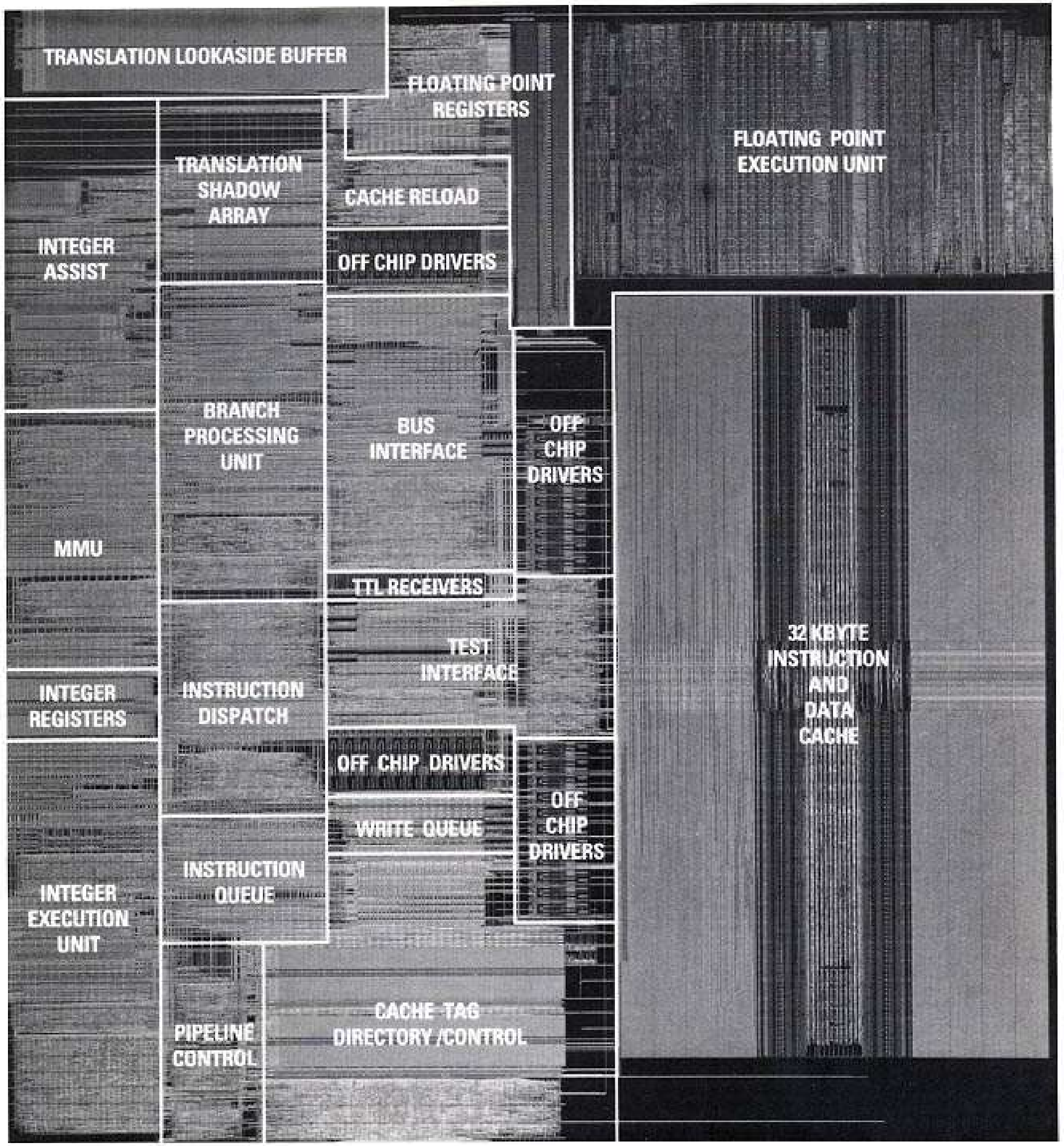
MIPS R8000 (TFP IU)



2.6 million transistors
17.2 × 17.3 mm
First silicon: May 1994

PowerPC

PowerPC 601-100

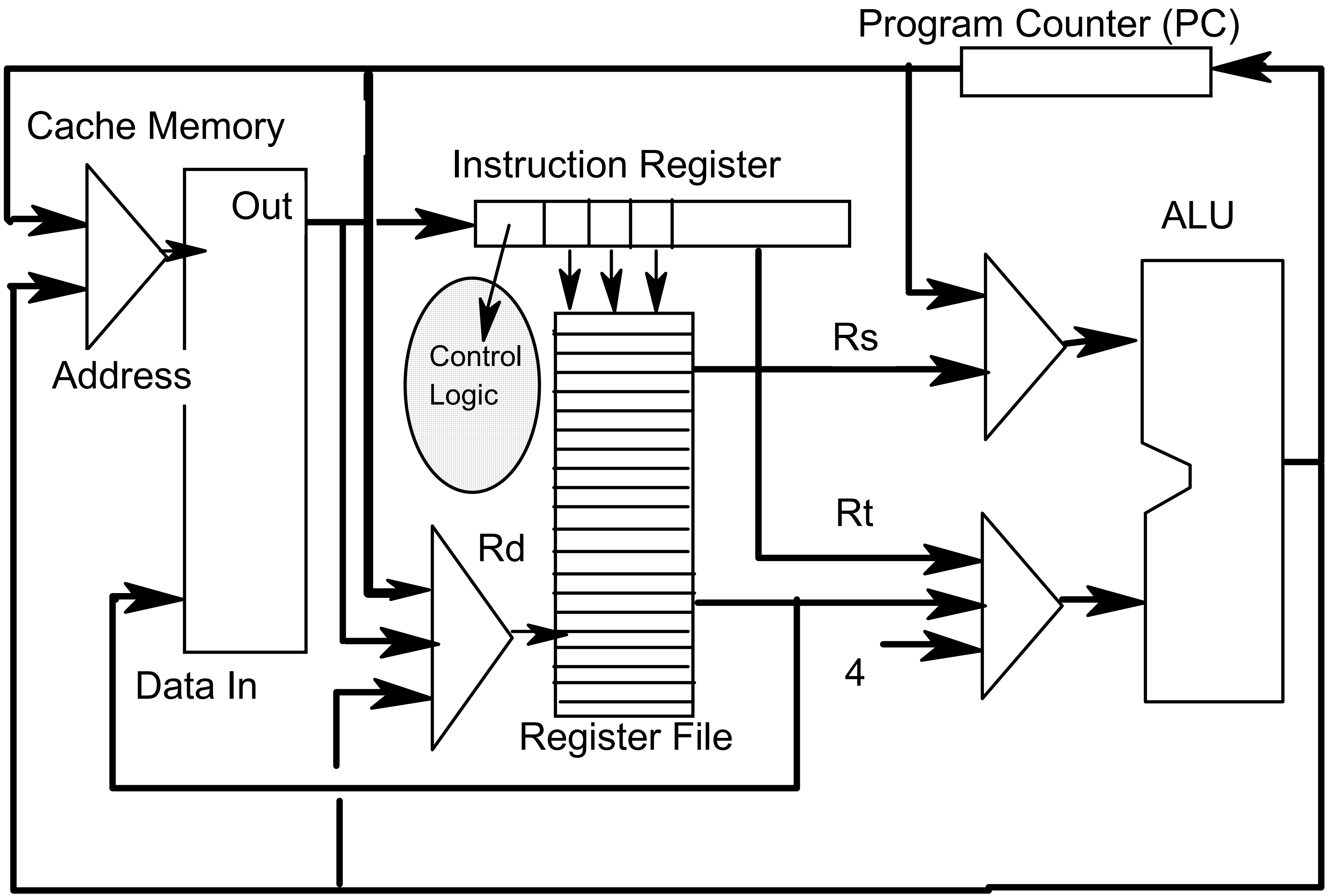


2.8 million transistors
8.6 × 8.6 mm
First silicon: March 1994

MIPS Computer Organization

- Datapath Diagram
- Control Logic

DataPath Diagram



Register File

Number	Value	Name
0		$zero
1		$at
2		$v0
3		$v1
4		$a0
5		$a1
6		$a2
7		$a3
8		$t0
9		$t1
10		$t2
11		$t3
12		$t4
13		$t5
14		$t6
15		$t7
16		$s0
17		$s1
18		$s2
19		$s3
20		$s4
21		$s5
22		$s6
23		$s7
24		$t8

Return values from functions

Pass parameters to functions

Caller Saved Registers –
Use these registers in functions

Callee-Saved Registers –
Use these registers for values that must be maintained across function calls.

An Example MIPS Assembly Language Program

```
Label     Op-Code Dest. S1,  S2     Comments
          move    $a0, $0           # $a0 = 0
          li      $t0, 99           # $t0 = 99
loop:
          add     $a0, $a0, $t0     # $a0 = $a0 + $t0
          addi    $t0, $t0, -1      # $t0 = $t0 - 1
          bnez    $t0, loop         # if ($t0  != zero) branch to loop
          li      $v0, 1            # Print the value in $a0
          syscall
          li      $v0, 10           # Terminate Program Run
          syscall
```

Three Instruction Word Formats

- Register Format

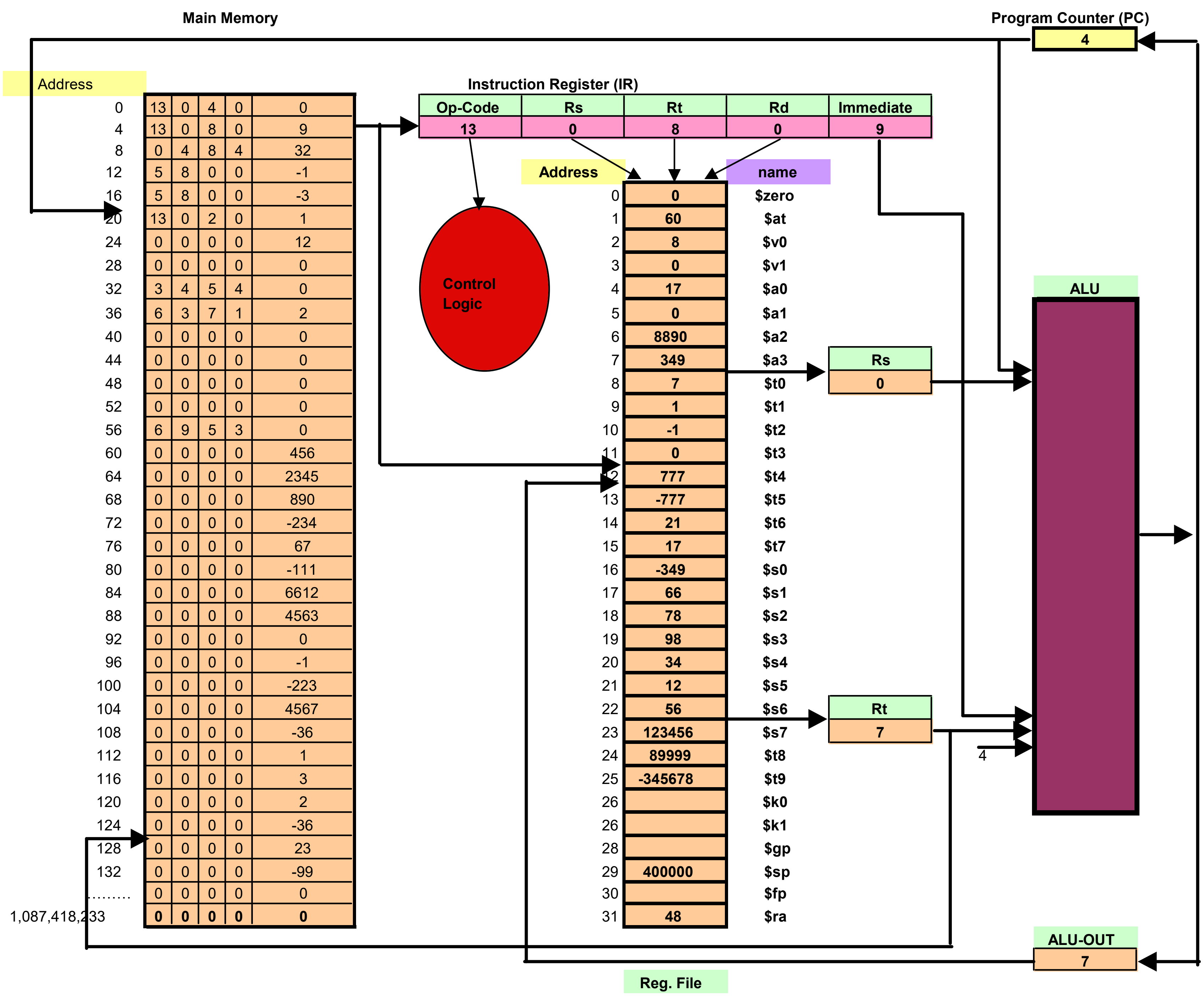
Op-Code	Rs	Rt	Rd	Code
6	5	5	5	6

- Immediate Format

Op-Code	Rs	Rt	16 - Bit Immediate Value
6	5	5	16

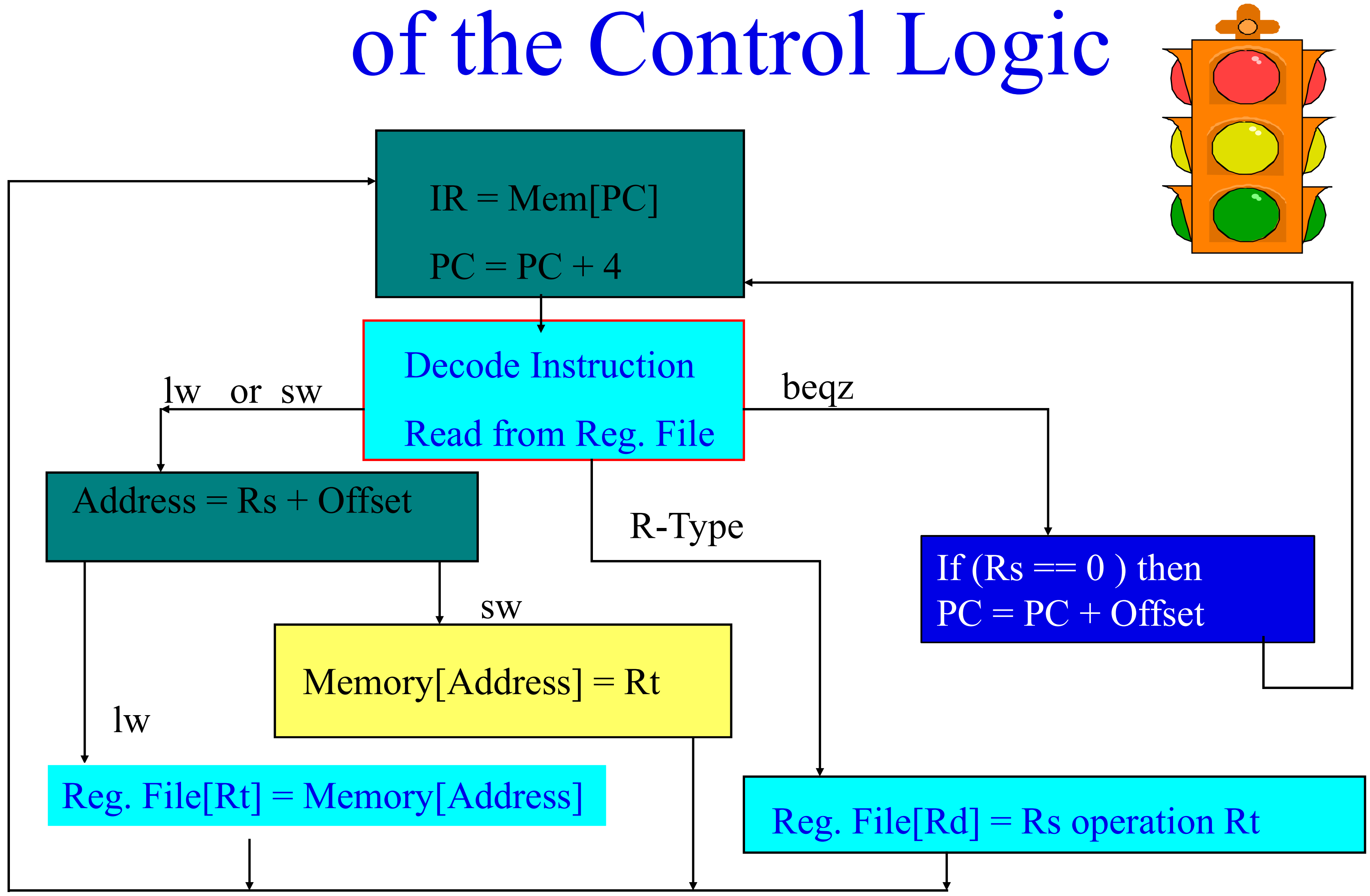
- Jump Format

Op-Code	26 Bit Current Segment Address
6	26



Main Memory
Program Counter (PC)
4
Address
Instruction Register (IR)
Op-Code
Rs
Rt
Rd
Immediate
13
0
8
0
9
0 | 13 | 0 | 4 | 0 | 0
4 | 13 | 0 | 8 | 0 | 9
8 | 0 | 4 | 8 | 4 | 32
12 | 5 | 8 | 0 | 0 | -1
16 | 5 | 8 | 0 | 0 | -3
20 | 13 | 0 | 2 | 0 | 1
24 | 0 | 0 | 0 | 0 | 12
28 | 0 | 0 | 0 | 0 | 0
32 | 3 | 4 | 5 | 4 | 0
36 | 6 | 3 | 7 | 1 | 2
40 | 0 | 0 | 0 | 0 | 0
44 | 0 | 0 | 0 | 0 | 0
48 | 0 | 0 | 0 | 0 | 0
52 | 0 | 0 | 0 | 0 | 0
56 | 6 | 9 | 5 | 3 | 0
60 | 0 | 0 | 0 | 0 | 456
64 | 0 | 0 | 0 | 0 | 2345
68 | 0 | 0 | 0 | 0 | 890
72 | 0 | 0 | 0 | 0 | -234
76 | 0 | 0 | 0 | 0 | 67
80 | 0 | 0 | 0 | 0 | -111
84 | 0 | 0 | 0 | 0 | 6612
88 | 0 | 0 | 0 | 0 | 4563
92 | 0 | 0 | 0 | 0 | 0
96 | 0 | 0 | 0 | 0 | -1
100 | 0 | 0 | 0 | 0 | -223
104 | 0 | 0 | 0 | 0 | 4567
108 | 0 | 0 | 0 | 0 | -36
112 | 0 | 0 | 0 | 0 | 1
116 | 0 | 0 | 0 | 0 | 3
120 | 0 | 0 | 0 | 0 | 2
124 | 0 | 0 | 0 | 0 | -36
128 | 0 | 0 | 0 | 0 | 23
132 | 0 | 0 | 0 | 0 | -99
……… | 0 | 0 | 0 | 0 | 0
1,087,418,233 | 0 | 0 | 0 | 0 | 0
Control Logic
Address
name
0 | 0 | $zero
1 | 60 | $at
2 | 8 | $v0
3 | 0 | $v1
4 | 17 | $a0
5 | 0 | $a1
6 | 8890 | $a2
7 | 349 | $a3
8 | 7 | $t0
9 | 1 | $t1
10 | -1 | $t2
11 | 0 | $t3
12 | 777 | $t4
13 | -777 | $t5
14 | 21 | $t6
15 | 17 | $t7
16 | -349 | $s0
17 | 66 | $s1
18 | 78 | $s2
19 | 98 | $s3
20 | 34 | $s4
21 | 12 | $s5
22 | 56 | $s6
23 | 123456 | $s7
24 | 89999 | $t8
25 | -345678 | $t9
26 | | $k0
26 | | $k1
28 | | $gp
29 | 400000 | $sp
30 | | $fp
31 | 48 | $ra
Rs
0
Rt
7
4
ALU
ALU-OUT
7
Reg. File

A Register Transfer Description of the Control Logic



MIPS Instruction Set

See Appendix C in the textbook for a detailed description of every instruction.

- Arithmetic, Logic, and Shifting Instructions
- Conditional Branch Instructions
- Load and Store Instructions
- Function Call Instructions

Pseudo Instructions †

• Load Address	la	$s0, table
• Load Immediate	li	$v0, 10
• Move	move	$t8, $sp
• Multiply	mul	$t2, $a0, $a1
• Divide	div	$s1, $v1, $t7
• Remainder	rem	$s2, $v1, $t7
• Negate	neg	$s0, $s0

Negate Unsigned	negu	Rd, Rs
Nop	nop	
Not	not	Rd, Rs
Remainder Unsigned	remu	Rd, Rs, Rt
Rotate Left Variable	rol	Rd, Rs, Rt
Rotate Right Variable	ror	Rd, Rs, Rt
Remainder	rem	Rd, Rs, Rt
Rotate Left Constant	rol	Rd, Rs, sa
Rotate Right Constant	ror	Rd, Rs, sa
Set if Equal	seq	Rd, Rs, Rt
Set if Greater Than or Equal	sge	Rd, Rs, Rt
Set if Greater Than or Equal Unsigned	sgeu	Rd, Rs, Rt
Set if Greater Than	sgt	Rd, Rs, Rt
Set if Greater Than Unsigned	sgtu	Rd, Rs, Rt
Set if Less Than or Equal	sle	Rd, Rs, Rt
Set if Less Than or Equal Unsigned	sleu	Rd, Rs, Rt
Set if Not Equal	sne	Rd, Rs, Rt
Unaligned Load Halfword Unsigned	ulh	Rd, n(Rs)
Unaligned Load Halfword	ulhu	Rd, n(Rs)
Unaligned Load Word	ulw	Rd, n(Rs)
Unaligned Store Halfword	ush	Rd, n(Rs)
Unaligned Store Word	usw	Rd, n(Rs)

MIPS Register File

Register Naming Convention

(See Figure 1.2 on Page 4)

- \$0 : Constant Zero
- \$v0 : Returned values from functions
- \$a0 : Arguments passed to functions
- \$t0 : Temporary registers (functions)
- \$s0 : Saved registers (main program)
- \$sp : Stack Pointer
- \$ra : Return address

TABLE 1.1 The Register File

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for the assembler
v0	2	Used for return values from function calls
v1	3	
a0	4	Used to pass arguments to functions
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called functions)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called function must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called function)
t9	25	
k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls

Exercises – Chapter 1

1.1 Explain the difference between a register and the ALU.

1.2 Explain the difference between Assembly Language and Machine Language.

1.3 Explain the difference between Cache Memory and the Register File.

1.4 Explain the difference between the Instruction Register and the Program Counter.

1.5 Explain the difference between a Buss and a control line.

1.6 Identify a kitchen appliance that contains a finite state machine.

1.7 If a 500 MHz machine takes one-clock cycle to fetch and execute an instruction, then what is the instruction execution rate of the machine?

1.8 How many instructions could the above machine execute in one minute?

1.9 Let's suppose we have a 40-year-old computer that has an instruction execution rate of one thousand instructions per second. How long would it take in days, hours, and minutes, to execute the same number of instructions that you derived for the 500 MHz machine?

1.10 What is an algorithm?

Pseudocode

Using Pseudocode to Document a MIPS Assembly Language Program

When documenting an algorithm in a language such as Pyhon or C, programmers use descriptive variable names such as: speed, volume, size, count, amount, etc. After the program is compiled, these variable names correspond to memory locations. To efficiently execute code, a compiler will attempt to keep the variables that are referenced most often in processor registers because access to a variable in a processor register is much faster than access to memory. MIPS has 32 processor registers. The names used to reference these registers are defined in Figure 2.1 on page 4 in the textbook.

As an assembly language programmer you must take maximum advantage of the processor registers. For example, you may have a value in register $s0 corresponding to speed, a value in register $s1 corresponding to volume, a value in register $s2 corresponding to size, and a value in register $t3 corresponding to count.

When using pseudocode to document an assembly language program, you will be expected to use the names of the registers you intend to use in the assembly language code. It is advisable to create a cross reference table between the processor register name and what it is being used for in the program .

We use register names in pseudocode because the purpose of the pseudocode is to document an assembly language program.

Unless you identify the registers being used, the pseudocode is quite limited in terms of having any correspondence to the assembly language code.

You will also find that as soon as you are able to develop the pseudocode in this format it is a very simple process to translate pseudocode into assembly language code.

Pseudocode for assembly language programs will have the appearance of Pyhon or C in terms of control structures and arithmetic expressions, but descriptive variable names will usually only appear in the LOAD ADDRESS (la) instruction where there is a reference to a symbolic memory address. In assembly language you define and allocate space for variables in the data segment of memory using assembler directives such as *.word* and *.space*. You will find that all of the MIPS instructions require the use of processor registers. When writing pseudocode you should specify the processor registers you are planning to use to accomplish the task.

Now for an example, let us suppose that we want to write an assembly language program to find the sum of the integers from 1 to N.
In other words do the following: 1 + 2 + 3 + 4 + 5 + 6 + 7 + ...+ N, where “N” is an input value.

On the next slide you will see a pseudocode description of the algorithm and following that the corresponding assembly language program, where processor register $t0 is used to accumulate the sum, and processor register $v0 is used as a loop counter.

Use a word processor to create the following program file.
Be sure to save as text only.

Next, load the program into SPIM. Run the program and experiment with the different features of the MIPS simulator. (For example: Single Step)

Read the help file for a description of how to use the simulator.

An Example MIPS Program

```
#  Program #1 : (descriptive name)          Programmer: YOUR NAME
#  Due Date  : Feb. 8, 2023                      Course: CPE 223
#  Last Modified: Feb. 1, 2023                 Section: 99
##################################################################
# Functional Description:  Find the sum of the integers from 1 to N where
#  N is a value input from the keyboard.
##################################################################
# Algorithmic Description in Pseudocode:
# main:    v0 <<  value read from the keyboard (syscall 4)
#            if (v0 < = 0 ) stop
#           t0 = 0;                  # t0 is used to accumulate the sum
#           While (v0 >= 0)  { t0 = t0 + v0;  v0 = v0 - 1}
#           Output to monitor syscall(1) << t0;     goto main
###################################################################
# Register Usage: $t0 is used to accumulate the sum
#                        $v0 the loop counter, counts down to zero
###################################################################
```

```
                .data
prompt:         .asciiz   "\n\n   Please Input a value for N = "
result:         .asciiz    " The sum of the integers from 1 to N is "
bye:            .asciiz    "\n  **** Adios Amigo - Have a good day **** "
                .globl    main
                .text
main:           li        $v0, 4             # system call code for print_str
                la        $a0, prompt        # load address of prompt into a0
                syscall                      # print the prompt message
                li        $v0, 5             # system call code for read_int
                syscall                      # reads a value of N into v0
                blez      $v0,  done         # if ( v0  < = 0 ) go to done
                li        $t0, 0             # clear $t0 to zero
loop:           add       $t0, $t0, $v0      # sum of integers in register $t0
                addi      $v0, $v0, -1       # summing  in reverse order
                bnez      $v0, loop                    # branch to loop if $v0 is !=
zero
                li        $v0, 4             # system call code for print_str
                la        $a0, result        # load address of message into $a0
                syscall                      # print the string
                li        $v0, 1             # system call code for print_int
                move      $a0, $t0           # a0 = $t0
                syscall                      # prints the value  in register $a0
                b         main
```

```
done:       li      $v0, 4          # system call code for print_str
            la      $a0, bye        # load address of msg. into $a0
            syscall                 # print the string

            li      $v0, 10         # terminate program
            syscall                         # return control to
system
```

MUST HAVE A BLANK LINE AT THE END OF THE TEXT FILE

Input/Output System Calls

See Appendix A

Service Call	$v0 Code	Arguments	Results
Print_integer	1	$a0 = integer	
Print_ string	4	$a0 = &string	
Read_integer	5		$v0= integer
Read_string	8	$a0 = &buffer $a1 = Length of buffer	
Exit	10		

SYSTEM I/0 SERVICES

Service	Code in $v0	Argument(s)	Result(s)
Print Integer	1	$a0 = number to be printed	
Print Float	2	$f12 = number to be printed	
Print Double	3	$f12 = number to be printed	
Print String	4	$a0 = address of string in memory	
Read Integer	5		number returned in $v0
Read Float	6		number returned in $f0
Read Double	7		number returned in $f0
Read String	8	$a0 = address of input buffer in memory $a1 = length of buffer (n)	
Sbrk	9	$a0 = amount	address in $v0
Exit	10		

The system call Read Integer reads an entire line of input from the keyboard up to and including the newline. Characters following the last digit in the decimal number are ignored. Read String has the same semantics as the Unix library routine fgets. It reads up to $n - 1$ characters into a buffer and terminates the string with a null byte. If fewer than $n - 1$ characters are on the current line, Read String reads up to and including the newline and again null terminates the string. Print String will display on the terminal the string of characters found in memory starting with the location pointed to by the address stored in `$a0`. Printing will stop when a null character is located in the string. Sbrk returns a pointer to a block of memory containing n additional bytes. Exit terminates the user program execution and returns control to the operating system.

Translation of “if – then -- else”

if ($t8 < 0) **then**
{$s0 = 0 - $t8;
$t1 = $t1 +1}
else
{$s0 = $t8;
$t2 = $t2 + 1}

Translation of pseudocode to MIPS assembly language. In MIPS assembly language, anything on a line following the number sign (#) is a comment. Notice how the comments in the code below help to make the connection back to the original pseudocode.

```
        bgez    $t8, else           # if ($t8 is > or = zero) branch to else
        sub     $s0, $zero, $t8     # $s0 gets the negative of $t8
        addi    $t1, $t1, 1         # increment $t1 by 1
        b       next                # branch around the else code
else:
        ori     $s0, $t8, 0         # $s0 gets a copy of $t8
        addi    $t2, $t2, 1         # increment $t2 by 1
next:
```

Translation of a “While” statement

```
         $v0 = 1
         While ($a1 < $a2) do
         {$t1 = mem[$a1];
         $t2 = mem[$a2];
         If ($t1 != $t2) go to break;
         $a1 = $a1 +1;
         $a2 = $a2 –1;}
         return
break:   $v0 = 0
         return
```

Here is a translation of the above “while” pseudocode into MIPS assembly language code.

```
        li      $v0, 1              # Load $v0 with the value 1
loop:
        bgeu    $a1, $a2, done      # If( $a1 >= $a2) Branch to done
        lb      $t1, 0($a1)         # Load a Byte: $t1 = mem[$a1 + 0]
        lb      $t2, 0($a2)         # Load a Byte: $t2 = mem[$a2 + 0]
        bne     $t1, $t2, break     # If ($t1 != $t2) Branch to break
        addi    $a1, $a1, 1         # $a1 = $a1 + 1
        addi    $a2, $a2, -1        # $a2 = $a2 - 1
        b       loop                # Branch to loop
break:
        li      $v0, 0              # Load $v0 with the value 0
done:
```

Translation of a "for loop"

$a0 = 0;

For ($t0 =10; $t0 > 0; $t0 = $t0 -1) **do** {$a0 = $a0 + $t0}

The following is a translation of the above "for-loop" pseudocode to MIPS assembly language code.

```
        li      $a0, 0          #  $a0 = 0
        li      $t0, 10         # Initialize loop counter to 10
loop:
        add     $a0, $a0, $t0
        addi    $t0, $t0, -1    # Decrement loop counter
        bgtz    $t0, loop       # If ($t0 >0) Branch to loop
```

Translation of a “switch statement”

Pseudocode Description:

```
        $s0 = 32;
top:    cout << “Input a value from 1 to 3”
        cin >> $v0
              switch($v0)
                    {
                    case(1): { $s0 = $s0 << 1; break}
                    case(2): { $s0 = $s0 << 2; break}
                    case(3): { $s0 = $s0 << 3; break}
                    default:   goto top ;
                    }
         cout <<$s0
```

“switch statement” continued

```
                .data
                .align   2
jumptable:      .word    top, case1, case2, case3
prompt:         .asciiz  "\n\n Input a value N from 1 to 3:  "
result:         .asciiz  " The value 32 shifted left by N bits is now = "
                .text
main:
                li       $s0, 32
top:
                li       $v0, 4              # Code to print a string
                la       $a0, prompt
                syscall
                li       $v0, 5              # Code to read an integer
                syscall
                bltz     $v0, exit
                beqz     $v0, top            # Default for less than one
                li       $t3, 3
                bgt      $v0, $t3, top       # Default for greater than 3
                la       $a1, jumptable
                sll      $t0, $v0, 2         # Create a word offset
                add      $t1, $a1, $t0       # Form a pointer into jumptable
                lw       $t2, 0($t1)         # Load an address from jumptable
                jr       $t2                 # Go to specific case
```

“switch statement” continued

```
case1:
            sll     $s0, $s0, 1
            b       done
case2:
            sll     $s0, $s0, 2
            b       done
case3:
            sll     $s0, $s0, 3
done:
            li      $v0, 4                  # Code to print a string
            la      $a0, result
            syscall
            li      $v0, 1                  # Code to print a value
            move    $a0, $s0
            syscall
            bgez    $s1, main
exit:
            li      $v0, 10
            syscall
```

Exercises – Chapter 2

2.1 Using Appendix A, translate each of the following pseudocode expressions into MIPS assembly language:

(a) t3 = t4 + t5 – t6;

(b) s3 = t2 / (s1 – 54321);

(c) sp = sp – 16;

(d) cout << t3;

(e) cin >> t0;

(f) a0 = &array;

(g) t8 = Mem(a0);

(h) Mem(a0+ 16) = 32768;

(i) cout << “Hello World”;

(j) If (t0 < 0) then t7 = 0 – t0 else t7 = t0;

(k) while (t0 != 0) { s1 = s1 + t0; t2 = t2 + 4; t0 = Mem(t2) };

(l) for (t1 = 99; t1 > 0; t1=t1 -1) v0 = v0 + t1;

Exercises – Chapter 2

(m) t0 = 2147483647 - 2147483648;

(n) s0 = -1 * s0;

(o) s1 = s1 * a0;

(p) s2 = srt($s0^2$ + 56) / a3;

(q) s3 = s1 - s2 / s3;

(r) s4 = s4 * 8;

(s) s5 = π * s5;

2.2 Analyze the assembly language code that you developed for each of the above pseudocode expressions and calculate the number of clock cycles required to fetch and execute the code corresponding to each expression. (Assume it takes one clock cycle to fetch and execute every instruction except multiply and divide, which require 32 clock cycles and 38 clock cycles respectively.)

2.3 Show how the following expression can be evaluated in MIPS assembly language, without modifying the contents of the "s" registers:

$t0 = ($s1 - $s0 / $s2) * $s4 ;

Exercises Continued

2.2 **Analyze the assembly language code that you developed for each of the above pseudocode expressions and calculate the number of clock cycles required to fetch and execute the code corresponding to each expression. (Assume it takes one clock cycle to fetch and execute every instruction except multiply and divide, which require 32 clock cycles and 38 clock cycles respectively.)**

2.3 **Show how the following expression can be evaluated in MIPS assembly language, without modifying the contents of the "s" registers:**

$t0 = ($s1 - $s0 / $s2) * $s4 ;

Number Systems

- Introduction
- Polynomial Expansion
- Binary Numbers
- Hexadecimal Numbers
- Two's Complement Number System
- Arithmetic & Overflow Detection
- American Standard Code for Information Interchange (ASCII)

Polynomial Expansion of a Decimal Number (Base 10)

$$\underline{496}_{10} = 4 \times 10^{2} + 9 \times 10^{1} + 6 \times 10^{0}$$

Polynomial Expansion of a Binary Number (Base 2)

$$\underline{00101101}_{2} = 1 \times 2^{5} + 0 \times 2^{4} + 1 \times 2^{3} + 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0}$$

A Faster Method ------ Double and Add

$\underline{00101101}$ = 45 (1, 2, 5, 11, 22, 45)

Conversion of Decimal Numbers to Binary

Divide by 2 and record the remainder

45	Remainder	1 0 1 1 0 1
22	1	
110		
5	1	
2	1	
1	0	
0	1	

Practice - Convert 25 to Binary

Divide by 2 and record the remainder

25	Remainder
12	

To represent binary values in the positive and negative domains we use the

Two's Complement Number System

Here is the polynomial expansion of a two's complement number 8-bit binary number N:

$$N = - d7 \times 2^{7} + d6 \times 2^{6} + d5 \times 2^{5} + d4 \times 2^{4} + d3 \times 2^{3} + d2 \times 2^{2} + d1 \times 2^{1} + d0 \times 2^{0}$$

Notice the Minus sign

*** You need to memorize powers of 2 ***

The Two's Complement Operation

When we take the two's complement of a binary number, the result will be the negative of the value we started with.

For example, the binary value **`00011010`** is 26 in decimal.

To find the value minus 26 in binary we perform the two's complement operation on **`00011010`**.

Scan the binary number from right to left leaving all least significant zeros (0) and the first one (1) unchanged, and then complementing the remaining digits to the left: **`11100110`**

The result is the value minus 26 in binary.

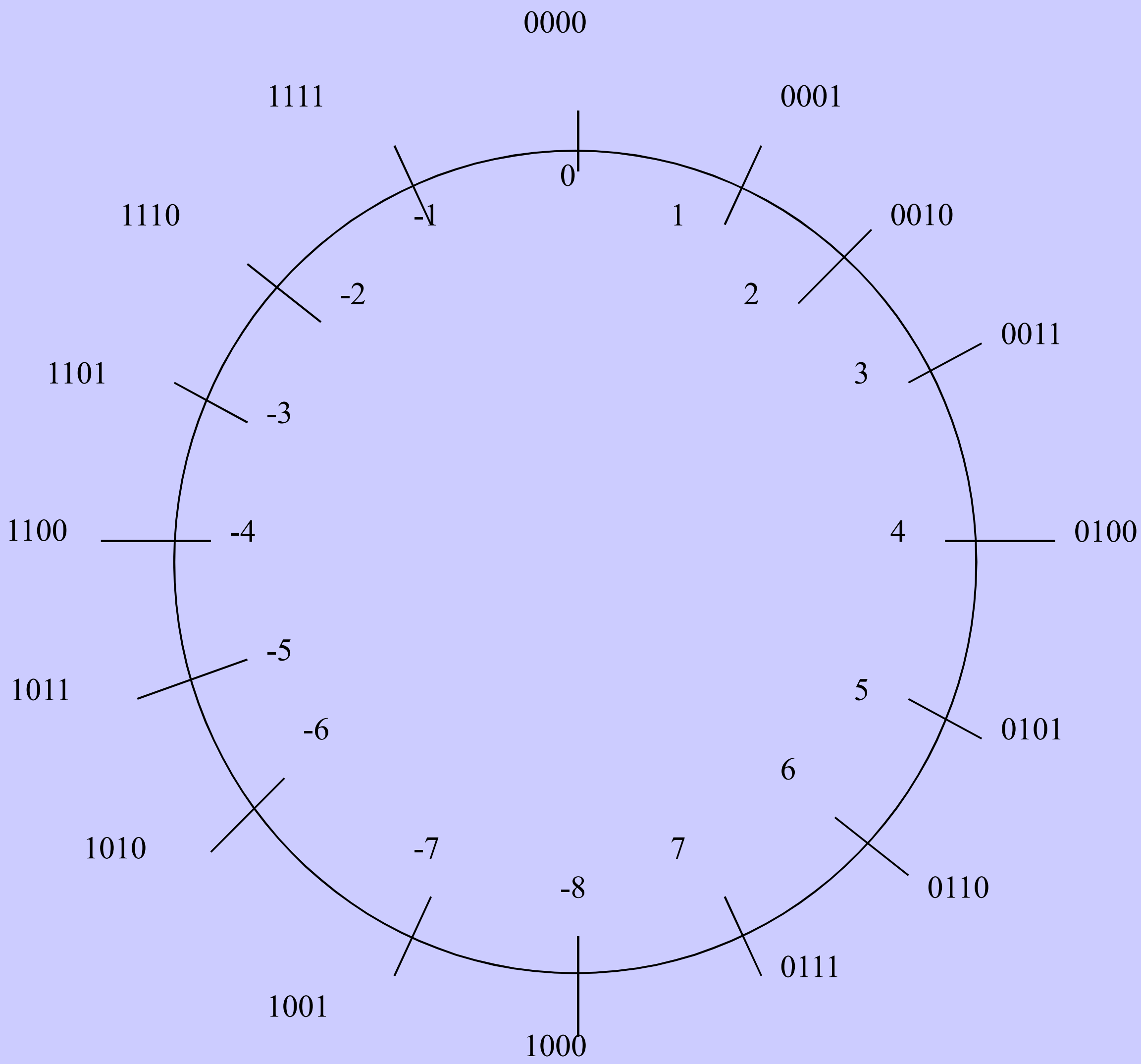
Binary Arithmetic & Overflow Detection
in the Two's Complement Number System

Here is an addition example where we assume we are limited to 8 binary digits.

$$
\begin{array}{rlr}
 & 01000100 = & 68 \\
+ & \underline{00111100} = & \underline{60} \\
 & 10000000 = & -128
\end{array}
$$

Overflow Occurred

Overflow


0000
1111
0001
0
-1
1
1110
0010
-2
2
0011
1101
3
-3
1100
-4
4
0100
-5
1011
5
0101
-6
6
1010
-7
7
0110
-8
0111
1001
1000

Binary Arithmetic in the Two's Complement Number System

Here is a subtraction example where we assume we are limited to 8 binary digits. To subtract in binary we always add the two's complement of the subtrahend.

01000100	=	01000100		68
-00111100	=	+11000100		60
00001000	=	00001000	=	8

The Rule for Detection of Overflow

##

Adding numbers of opposite signs, overflow is impossible.

When adding numbers of the same sign, if the result is not the same as the operands then overflow occurred.

##

Here is an example:

You are given the following two numbers in two's complement representation. Perform the binary subtraction and indicate if there is signed overflow. ______

Explain Why:

```
  11101000          11101000          = -24
 -00010011         +11101101          = -19
 ---------         ---------            ---
                    11010101 Correct
                             Result = -43
```

Sign Extension

The value –43 as an 8-bit binary number is: **`11010101`**

The value –43 as an 32-bit binary number is:

`11111111111111111111111111010101`

In Hexadecimal –43 appears as: 0xFFFFFFD5

###

The value 68 as an 8-bit binary number is: **`01000100`**

The value 68 as an 32-bit binary number is:

`00000000000000000000000001000100`

In Hexadecimal 68 appears as: 0x00000044

The Hexadecimal Number System

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Here is an example of how we compactly represent binary numbers in hexadecimal:

| | | | |

001111001000111101111110

0x 3 C 8 F 7 E

Chapter 2 Exercises Continued

2.3 Show how the following expression can be evaluated in MIPS assembly language, without modifying the contents of the "s" registers:

$t0 = ($s1 - $s0 / $s2) * $s4 ;

2.4 The datapath diagram for the MIPS architecture shown in figure 1.1 with only one memory module is referred to as a von Neumann architecture. Most implementations of the MIPS architecture use a Harvard Architecture, where there are two separate memory modules, one for instructions and the the other for data. Draw such a datapath diagram.

2.5 Show how the following pseudocode expression can be efficiently evaluated in MIPS assembly language, without modifying the contents of the "s" registers:

$t0 = ($s0 / 8 - 2 * $s1 + $s2 ;

Exercises

3.1 Convert the decimal number 35 to an 8-bit binary number.

3.2 Convert the decimal number 32 to an 8-bit binary number.

3.3 Using the double and add method convert 00010101 to a decimal number.

3.4 Using the double and add method convert 00011001 to a decimal number.

3.5 Explain why the Least Significant digit of a binary number indicates if the number is odd or even.

3.6 Convert the binary number 00010101 to a hexadecimal number.

3.7 Convert the binary number 00011001 to a hexadecimal number.

3.8 Convert the hexadecimal number 0x15 to a decimal number.

3.9 Convert the hexadecimal number 0x19 to a decimal number.

3.10 Convert the decimal number -35 to an 8-bit two's complement binary number.

Exercises

3.11 Convert the decimal number -32 to an 8-bit two's complement binary number.

3.12 Assuming the use of the two's complement number system find the equivalent decimal values for the following 8-bit binary numbers:

(a) 10000001
(b) 11111111
(c) 01010000
(d) 11100000
(e) 10000011

3.13 Convert the base 8 number 204 to decimal

3.14 Convert the base 7 number 204 to decimal

3.15 Convert the base 6 number 204 to decimal

3.16 Convert the base 5 number 204 to decimal

3.17 Convert the base 10 number 81 to a base 9 number.

Exercises

3.18 **For each row of the table below convert the given 16 bit number to each of the other two bases, assuming the two's complement number system is used.**

16 Bit Binary	Hexadecimal	Decimal
1111111100111100		
	0xFF88	
		-128
1111111111111010		
	0x0011	
		-25

3.19 **You are given the following two numbers in two's complement representation. Perform the binary addition and indicate if there is signed overflow. __**

Explain Why:

01101110
<u>00011010</u>

Exercises

3.20 You are given the following two numbers in two's complement representation. Perform the binary subtraction and indicate if there is signed overflow. ______

Explain Why:

```
 11101000
-00010011
```

3.21 Sign extend the 8 bit hex number 0x88 to a 16 bit number. 0x_________

3.22 The following subtract instruction is located at address 0x00012344. What are the two possible values for the contents of the PC after the branch instruction executed? 0x_____________ 0x_____________

This branch instruction is described in Appendix C.

```
loop:   addi   $t4, $t4, -8
        sub    $t2, $t2, $t0
        bne    $t4, $t2,loop
```

Exercises

3.23 **You are given the following two 8-bit binary numbers in the two's complement number system. What values do they represent in decimal?**

X = $\underline{10010100}_2$ = $\underline{\hspace{3cm}}_{10}$ Y = $\underline{00101100}_2$ = $\underline{\hspace{3cm}}_{10}$

Perform the following arithmetic operations on X and Y. Show your answers as 8-bit binary numbers in the two's complement number system.
To subtract Y from X, find the two's complement of Y and add it to X.
Indicate if overflow occurs in performing any of these operations.

X+Y	X-Y	Y-X
10010100	10010100	00101100
$\underline{00101100}$		

Exercise 3.24

The following code segment is stored in memory starting at memory location 0x00012344.
What are the two possible values for the contents of the PC after the branch instruction has executed? 0x__________ 0x__________
Add in line pseudocode to describe each instruction.

```
loop:  lw    $t0, 0($a0)     #
       addi  $a0, $a0, 4     #
       andi  $t1, $t0, 1     #
       beqz  $t1,  loop      #
```

SPIM - The MIPS Simulator,

Register Window

Text Window

Data Window

Message Window

Console

Text Segment

```
[0x00400020]    0x34020004  ori $2, $0, 4                  ; 34: li  $v0, 4
[0x00400024]    0x3c041001  lui $4, 4097 [prompt]   ; 35: la $a0, prompt
[0x00400028]    0x0000000c  syscall                         ; 36: syscall
[0x0040002c]    0x34020005  ori $2, $0, 5                  ; 38: li  $v0, 5
[0x00400030]    0x0000000c  syscall                          ; 39: syscall
[0x00400034]    0x1840000d  blez $2 52 [end-0x00400034]     ; 41: blez   $v0,  end
[0x00400038]    0x34080000  ori $8, $0, 0                  ; 42: li  $t0, 0
[0x0040003c]    0x01024020  add $8, $8, $2                 ; 44: add  $t0, $t0, $v0
[0x00400040]    0x2042ffff  addi $2, $2, -1                 ; 45: addi  $v0, $v0, -1
[0x00400044]    0x14403ffe  bne $2, $0, -8 [loop-0x00400044]; 46: bnez  $v0,  loop
[0x00400048]    0x34020004  ori $2, $0, 4                   ; 47: li  $v0, 4
[0x0040004c]    0x3c011001  lui $1, 4097 [result]         ; 48: la $a0, result
[0x00400050]    0x34240022  ori $4, $1, 34 [result]
[0x00400054]    0x0000000c  syscall                           ; 49: syscall
```

Analyzing the Data Segment

```
             .data
prompt:  .asciiz    "\n   Please Input a value for N = "
result:    .asciiz    "   The sum of the integers from 1 to N is "
bye:        .asciiz     " **** Adios Amigo – Have a good day ****"
```

		a e l P	**e s**	
[0x10010000]	0x2020200a	0x**61656c50**	0x4920**6573**	0x7475706e
[0x10010010]	0x76206120	0x65756c61	0x726f6620	0x3d204e20
[0x10010020]	0x20200020	0x65685420	0x6d757320	0x20666f20
[0x10010030]	0x20656874	0x65746e69	0x73726567	0x6f726620
[0x10010040]	0x2031206d	0x4e206f74	0x20736920	0x20200a00

This is an example of an addressing structure called Little Indian where the right most byte in a word has the smaller address.

Translating Assembly Language to Machine language

Use the information in Appendix C to verify that 0x3402000A is the correct machine language encoding of the instruction

ori $2, $0, 10   li $v0, 10

In Appendix C we are shown how this instruction is encoded in binary

ori Rt, Rs, Imm   **# RF[Rt] = RF[Rs] OR Imm**

Op-Code	Rs	Rt	Imm
001101	sssss	ttttt	iiiiiiiiiiiiiiii

0011	0100	0000	0010	0000	0000	0000	1010

0x 3 4 0 2 0 0 0 A

Translating Assembly Language to Machine language
R-Type Instruction

Use the information in Appendix C to verify that 0x01024020 is the correct machine language encoding of the instruction

add $8, $8, $2　　　　　add $t0, $t0, $v0

In Appendix C we are shown how this instruction is encoded in binary

add　Rd, Rs, Rt　# RF[Rd] = RF[Rs] + RF[Rt]

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000100000

0000	0001	0000	0010	0100	0000	0010	0000
0x 0	1	0	2	4	0	2	0

Exercise 4.1

Translate the following assembly language instructions to their corresponding machine language codes as they would be represented in hexadecimal. (Hint – Refer to Appendix C and Appendix D.)

```
loop:   addu    $a0, $0, $t0          #
        ori     $v0, $0, 4            #
        syscall                       #
        addi    $t0, $t0, -1          #
        bnez    $t0, loop             #
        andi    $s0, $s7, 0xffc0      #
        or      $a0, $t7, $s0         #
        sb      $a0, 4($s6)           #
        srl     $s7, $s7, 4           #
```

Translating Assembly Language Store Byte to Machine Language

In Appendix C we are shown how Store Byte is encoded in binary

sb Rt, offset(Rs) # Mem[RF[Rs] + Offset] = RF[Rt]

Op-Code	Rs	Rt	Offset
101000	sssss	ttttt	iiiiiiiiiiiiiiii

sb $4, 4($22)   sb $a0, 4($s6)

1010	0010	1100	0100	0000	0000	0000	0100
0xA	2	C	4	0	0	0	4

Translating Assembly Language Shift Instruction to Machine Language

In Appendix C we are shown how Shift Right Logical is encoded in binary

srl Rd, Rs, sa # Rs = Rt << sa

Op-Code		Rt	Rd	sa	
000000	00000	ttttt	ddddd	vvvvv	000010

srl $23, $23, 4 srl $s7, $s7, 4

0000	0000	0001	0111	1011	1001	0000	0010

0x0	0	1	7	B	9	0	2

PCSpim Translation

```
[0x00082021     addu $4, $0, $8                        ; 3: addu    $a0, $0, $t0
[0x34020004     ori $2, $0, 4                          ; 4: ori      $v0, $0, 4
[0x0000000c     syscall                                ; 5: syscall
[0x2108ffff     addi $8, $8, -1                        ; 6: addi   $t0, $t0, -1
[0x1500fffc      bne $8, $0, -16 [main-0x00400030]; 7: bnez    $t0, main
[0x32f0ffc0     andi $16, $23, -64                     ; 8: andi    $s0, $s7, 0xffc0
[0x01f02025     or $4, $15, $16                        ; 9: or       $a0, $t7, $s0
[0xa2c40004     sb $4, 4($22)                          ; 10: sb      $a0, 4($s6)
[0x0017b902     srl $23, $23, 4                        ; 11: srl  $S7, $s7, 4
```

Exercises 4.2

What is the character string corresponding to the following ASCII codes?
Remember that for simulations running on Intel–based platforms, the characters are stored in reverse order within each word.)

**** Adios Amigo – Have