Chapter 2 — Instructions: Language of the Computer

2.1 Introduction

[Ref: Textbook §2.1]

2.1.1 Instruction Set

The **ISA** (Instruction Set Architecture) of a microprocessor is an assembly language programmer's view of the microprocessor. It specifies things such as the set of all instructions that the microprocessor supports (called the **instruction set**), information about the registers (names, widths, uses), instruction timings (timings are generally given in clock cycles rather than seconds as clock cycles makes the time that it takes for an instruction to execute independent of the clock frequency), memory organization and addressing, and other related information.

Instruction sets are unique to each processor architecture but instruction sets among different processors are often very similar because:

1. All processors are constructed using the same underlying hardware principles, and
2. There are several basic instructions that all processors must support (e.g., ADD).

2.1.2 MIPS, RISC, and CISC

The textbook discusses the MIPS architecture, which is an example of a **Reduced Instruction Set Computer** (RISC) processor. RISC processors were created in the early 1980's[1] and became very popular by the end of the decade.

RISC processors are one type of processor organization. The other dominant, and older, type are the **Complex Instruction Set Computers** (CISC), which only began to be called CISC after the development of RISC processors.

Characteristics of CISC processors:

- The historical way in which processors were designed starting in the 1940's.
- Some instructions were very complex (e.g., VAX polynomial evaluation instruction[2]) and rarely used. Intel 8086 had an instruction to move a string (a block of characters) in memory[3].
- Instructions commonly completed in a variable number of clock cycles.
- Numerous addressing modes (addressing modes refers to how the location and destination of data is specified). The VAX architecture went overboard on these.
- Complex instructions resulted in smaller program sizes (reduced static instruction count) which was useful in the 1960's–1970's when memories were small and very expensive.
- Complex instructions made programmers more productive in an era when assembly language was widely used.

Characteristics of RISC processors:

- Generally fewer number of instructions[4].
- Emphasis on reducing the number of clocks for each instruction to 1 or fewer.
- Instructions are generally all same size (in bits)[5].

1 There were some RISC-like designs that preceded what are generally considered the first RISC processor designs. First-to-invent is not always so clear.

2 The POLY instruction: http://www.openvms.compaq.com/doc/73final/4515/4515pro_024.html#4515ch9_134

3 MOVS. It turned out that the MOVS instruction was so poorly implemented that it was slower than just using a loop at the machine language level to move characters one at a time.

4 Although some current RISC designs have more instructions than historical CISC designs.

5 This is not strictly true, some versions of the ARM architecture—also a RISC architecture—for example, support variable-length instructions.

- Relatively large number of registers.
- Simplified addressing modes (often just register direct, displacement, immediate, and one or two others).
- Increased instruction level parallelism with pipelining, superscalar (multiple execution units), out-of-order execution, branch prediction (will the branch be taken?), and branch target prediction (if we branch, where to?).
- Compiler support is crucial for generating efficient machine code.

MIPS was so successful that the architecture is still widely used, having gone through several generations:

- MIPS I - Implemented in the original 32-bit MIPS Inc. microprocessors, the R2000 (1985) and the R3000 (1988).
- MIPS II - R6000 (1990).
- MIPS III - R4000 (1992). Implemented 64-bit registers, 64-bit integer instructions, and a floating point unit (FPU).
- MIPS IV - R8000 (1994) was the first superscaler (multiple execution units) design. R10000 (1995) supported out-of-order execution.
- MIPS V - Never implemented in a design.

When MIPS (actually SGI, which owned MIPS) stopped manufacturing processors and began to focus on the embedded market by licensing its architecture, the ISA was changed to just two versions:

- MIPS32 - based on MIPS II (1999). A 32-bit architecture.
- MIPS64 - based on MIPS V (1999). A 64-bit architecture.

The textbook is oriented toward the MIPS32 (MIPS II) architecture. Incidentally, MIPS is an acronym for **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages. What "interlocked pipeline stages" means will be made clear in Chapter 4.

2.2 Signed and Unsigned Integers

[Ref: Textbook §2.4]

2.2.1 Representing Unsigned Integers

Using n-bits to represent an unsigned integer allows us to represent integers in the range [0, 2^n - 1]. For example, using $n = 8$ bits permits us to represent numbers in the range 0 (0000_0000) to 255 (1111_1111).

2.2.2 Representing Signed Integers

Using n-bits to represent a signed integer in two's complement allows us to represent integers in the range [-2^{n-1}, 2^{n-1} - 1].

Example: Let $n = 3$.

2.2.3 Converting from Decimal to Binary Two's Complement

Example: Let $n = 32$. Convert x = -123,456 to binary.

Method 1:

Write 123,456 as a 32-bit binary number:	0000 0000 0000 0001 1110 0010 0100 0000
Form the one's complement (negation):	1111 1111 1111 1110 0001 1101 1011 1111
Add 1:	1111 1111 1111 1110 0001 1101 1100 0000
Write in hex format for convenience:	0xFFFE1DC0

Method 2:

Calculate $2^{32} + x$:	4,294,967,296 + -123,456 = 4,294,843,840
Convert 4,294,843,840 to binary:	1111 1111 1111 1110 0001 1101 1100 0000
Write in hex format for convenience:	0xFFFE1DC0

2.2.4 Converting from Binary Two's Complement to Decimal

Example: What signed integer does 0xEAAA1234 represent?

Method 1:

Write in binary:	1110 1010 1010 1010 0001 0010 0011 0100
Subtract 1:	1110 1010 1010 1010 0001 0010 0011 0011
Form the one's complement:	0001 0101 0101 0101 1110 1101 1100 1100
Write in decimal as negative:	-357,952,972

Method 2:

Write in decimal as positive (y):	3,937,014,324
Calculate -(2^{32} - y):	-(4,294,967,296 - 3,937,014,324) = -357,952,972

2.3 Operations and Operands of the Computer Hardware

[Ref: Textbook §2.2–§2.3, §2.6, §2.10] An assembly language program consists of one or more **instructions**, with each instruction having zero or more **operands**. The names of the instructions are called **mnemonics**[6], e.g., `add` is the mnemonic for the instruction that performs addition on signed integers. Operands are the data on which the instruction operates. The general format of each line of an assembly language program is,

`[label]`$_{\text{optional}}$ `instr` `[operands]`$_{\text{optional}}$ `[comment]`$_{\text{optional}}$

For example,

```
fact:  addi  $sp, $sp, -12  # Allocate 3 words
       sw    $ra, 8($sp)    # Save $ra
       sw    $fp, 4($sp)    # Save $fp
       addi  $fp, $sp, 8    # Make $fp point to top of stack frame
```

On the first line, `fact` is a label, `addi` is the instruction mnemonic, `$sp, $sp, -12` are the operands of the instruction, and `# Allocate 3 words` is a comment.

A **label** is an identifier, i.e., a name for something, and in particular, assembly language labels are simply names for memory addresses; `fact` is the address in memory of the `addi` instruction.

Assembly language **comments** are just like comments in high level languages; a comment should document the code. In MIPS assembly language, comments start with a **#** character and proceed to the end of the line.

In the remainder of this section, we will discuss various MIPS assembly language instructions, but before we do, we need to learn more about operands. As mentioned above, an operand is data on which an instruction operates. Operands can be located in different places within a computer system. The three locations we shall discuss are **register** operands, **memory** operands, and **immediate** operands.

Before proceeding, in MIPS32 a **word** is a 32-bit value and a **half-word** is a 16-bit value; a **byte** is, of course, an 8-bit value.

2.3.1 Register Operands

A register is storage within the CPU. All processors will have at least one register, with most having several. The MIPS architecture contains thirty-two 32-bit **general purpose**[7] registers, numbered $0 through $31 (the $ indicates to the assembler that this is a register). Even though the 32-bit general purpose registers can be used for any purpose, by convention, the registers have typical uses. Note also that the MARS MIPS assembler permits us to use the register names in the first column rather than the register numbers.

Reg Name	Reg Num	Typical Use
$zero	$0	Read-only. Always reads as 0. Can be written, but will not be modified.
$at	$1	Assembler temporary. Used by the assembler in implementing pseudoinstructions.
$v0–$v1	$2–$3	Return values from functions.
$a0–$a3	$4–$7	Arguments to function calls.
$t0–$t7	$8–$15	Temporary registers. Any function can write without saving.

6 A mnemonic is something which is supposed to help you remember something else, e.g., Roy G. Biv is a mnemonic for the colors of the rainbow: red, orange, yellow, green, blue, indigo, violet.

7 A general purpose register is one that can generally be used for any purpose. Most processors also have special purpose registers which have a specific purpose and cannot be used for anything else.

Reg Name	Reg Num	Typical Use
$s0–$s7	$16–$23	Saved temporary registers. Must be saved by a function before writing.
$t8–$t9	$24–$25	Temporary registers. Any function can write without saving.
$k0–$k1	$26–$27	Reserved for the OS kernel.
$gp	$28	Global pointer. Contains the address of a data segment containing global data.
$sp	$29	Stack pointer. Contains the address of the top of the stack.
$fp	$30	Frame pointer. Used in function calls to create a stack frame.
$ra	$31	Stores the return address from a function.

The MIPS architecture also includes three **special purpose registers**,

Reg Name	Use
PC	Program counter. Contains the address of the instruction to be fetched and executed.
HI	High register. Some instructions that produce 64-bit values will write 32-bits of the value here.
LO	Low register. Some instructions that produce 64-bit values will write 32-bits of the value here.

2.3.2 Memory Operands

Data that are not currently being used—and stored in registers in the CPU—are stored in memory. Every processor must have data transfer instructions to move data from memory to a register and from a register to memory. In MIPS these instructions are `lw` (load word) and `sw` (store word) which we shall discuss in more detail soon.

2.3.2.1 Memory Alignment Restrictions

Since a MIPS32 word is 4-bytes, if we place words in memory starting at address 0x00, then the bytes of the first word will occupy memory locations 0x00, 0x01, 0x02, and 0x03. The second word would occupy addresses 0x04–0x7, the third word addresses 0x08–0x0B, and so on. Even though the third word occupies memory locations 0x08–0x0B, we say that the address of this word is 0x08, i.e., the address of a word is the address of the first byte of that word. Note, then, that the addresses of words in memory are **aligned** at addresses that are divisible by 4 and this is referred to as the **natural alignment**.

Some architectures permit words to be stored at nonnatural alignments, e.g., the bytes of a word could be stored at addresses 0x02–0x05. However, there is generally a performance hit when such a word is accessed. Due to the way memory is organized and the architecture of the memory bus, accessing this word would require two memory accesses: the first would read the word at 0x00–0x03 and the second would read the word at 0x04–0x07. Next, the relevant bytes from those two words would be extracted and combined to form the word at address 0x02.

In MIPS, words must be naturally aligned in memory, i.e., at addresses that are divisible by 4. Note that in binary, a number that is divisible by 4 will have 00 as the two least significant bits.

2.3.2.2 Endianness

The bytes of a multibyte word, such as 0xFFAA5511, can be stored two ways in memory:

- Little endian - Store the least significant byte (LSB) at the lowest-numbered memory address.
- Big endian - Store the most significant byte (MSB) at the lowest-numbered memory address.

MIPS is a big-endian architecture[8]. The Intel x86 and x86_64 architectures are little endian.

8 The original MIPS design was big-endian. Later designs permitted the CPU to operate in either little- or big-endian mode. Such processors are said to be bi-endian.

2.3.3 Constant or Immediate Operands

Integer constants, such as 0, 1, -1, 4, and 3131, are widely used in programming and at the assembly language level are referred to as **immediate** data. The location of an immediate is within the instruction itself.

2.3.4 MIPS Arithmetic Instructions

2.3.4.1 MIPS Add Signed Word Instruction

[Ref: MIPS Vol. II-A p. 47]

```
add $dst, $src1, $src2  # $dst ← $src1 + $src2
```

The `add` instruction treats the integers in $src1 and $src2 as signed integers and will add them placing the sum in $dst (in MIPS, the destination register is always the leftmost operand)[9].

During `add`, if an overflow occurs, the destination register is not modified and an **integer overflow** exception will occur.

Example: Register $t0 contains the value of variable a, $t1 contains the value of variable b, and we have associated register $t2 with the variable c. Write the instruction to perform $c = a + b$:

2.3.4.2 MIPS Subtract Signed Word Instruction

[Ref: MIPS Vol. II-A p. 277]

```
sub $dst, $src1, $src2  # $dst ← $src1 - $src2
```

During `sub`, if an overflow occurs, the destination register is not modified and an **integer overflow** exception will occur.

2.3.4.3 MIPS Multiply Word to GPR Instruction

[Ref: MIPS Vol. II-A p. 214]

```
mul $dst, $src1, $src2  # $dst ← $src1 × $src2
```

The two 32-bit words in $src1 and $src2 are treated as signed integers and the least significant 32-bits of the product is written to $dst; the most significant 32-bits of the product are lost.

Example: Register $t0 contains the value of variable a, $t1 contains the value of variable b, and we have associated register $t2 with the variable c. Write the instruction to perform $c = a \times b$,

```
mul $t2, $t0, $t1  # $dst ← $src1 × $src2
```

2.3.4.4 MIPS Divide Word Instruction

[Ref: MIPS Vol. II-A p. 127]

```
div $src1, $src2  # LO ← quotient($src1 / $src2); HI ← remainder($src1 / $src2)
```

The two 32-bit words in $src1 and $src2 are treated as signed integers. The 32-bit quotient is written to special purpose register LO and the 32-bit remainder is written to special purpose register HI. These values can be moved to general purpose registers using the `mfhi` (move from HI) and `mflo` (move from LO) instructions.

9 When an operand is a register, the **addressing mode** being used to access the operand is referred to as **register direct addressing mode**.

Example: Register $t0 contains the value of variable a, $t1 contains the value of variable b, we have associated register $t2 with the variable c, and register $t3 is associated with variable d. Write the instructions to perform $c = a \,/\, b$ and $d = a \bmod b$,

```
div   $t0, $t1  # LO ← a / b; HI ← a mod b
mflo  $t2       # c ← a / b
mfhi  $t3       # d ← a mod b
```

Example:

```
int a, b, c, d, e;
e = ((a + b) * (c - d)) % 7;
```

Assume $s0 is associated with e, $s1 contains the value of a, $s2 contains the value of b, $s3 contains the value of c, $s4 contains the value of d, and $s5 contains 7,

2.3.4.5 MIPS Add Immediate Word Instruction

[Ref: MIPS Vol. II-A p. 49]

`addi $dst, $src,` $\text{imm}_{15:0}$ `  # $dst ← $src +` ***sign-ext***`(`$\text{imm}_{15:0}$`)`

The immediate $imm_{15:0}$ is a 16-bit signed integer which is **sign-extended**[10] to form a 32-bit signed integer before adding it to the contents of $\$src$[11]. If an overflow occurs during addition, the destination register will not be modified and an integer overflow exception will occur.

Example:

```
int x;
...
++x;
```

Assume the value of x is in $s0,

```
addi  $s0, $s0, 1  # $s0 ← x + 1
```

Example:

```
x += 5;
```

Assume the value of x is in $t8,

```
addi   $t8, $t8, 5  # $t8 ← x + 5
```

10 To sign-extend a 16-bit signed integer to a 32-bit integer means to copy bit 15 (which will be 0 for integers ≥ 0 and 1 for integers < 0) to bit positions 31:16 of the resulting 32-bit integer. Signed 16-bit integer bits 15:0 are copied to 32-bit integer bits 15:0.

11 When an operand is an immediate, the addressing mode being used to access the operand is referred to as **immediate addressing mode**. Every processor supports immediate mode.

Note that MIPS does not have a subtract immediate instruction because the immediate is a signed integer. Example:

```
x -= 10;
```

Assume the value of x is in $t2,

```
addi  $t2, $t2, -10  # $t2 ← x + -10
```

2.3.5 MIPS Memory Access Instructions

2.3.5.1 MIPS Load Word Instruction

[Ref: MIPS Vol. II-A p. 170]

`lw $dst,` $\texttt{off}_{15:0}$`($base)  # $dst ← MEM[$base+`***`sign-ext`***`(`$\texttt{off}_{15:0}$`)]:MEM[$base+`***`sign-ext`***`(`$\texttt{off}_{15:0}$`)+3]`

The offset $off_{15:0}$ is a 16-bit signed integer which is sign-extended to form a 32-bit signed integer before adding it to $\$base$[12]. The sum is the address in memory of a word which will be loaded into $dst.

Example:

```
int A[100], g, h;
...
g = h + A[8];
```

Assume $s1 is associated with variable g, $s2 contains h, and $s3 contains the address of A, i.e., `&A` (where `&` is the address operator in C). Since an `int` is 4-bytes, the address of A[8] will be &A + 4 × 8 = &A + 32,

2.3.5.2 MIPS Store Word Instruction

[Ref: MIPS Vol. II-A p. 281]

`sw $src,` $\texttt{off}_{15:0}$`($base)  # MEM[$base+`***`sign-ext`***`(`$\texttt{off}_{15:0}$`)]:MEM[$base+`***`sign-ext`***`(`$\texttt{off}_{15:0}$`)+3] ← $src`

The offset $off_{15:0}$ is a 16-bit signed (two's complement) integer which is sign-extended to form a 32-bit signed integer before adding it to $\$base$. The sum is the address in memory where the word in $src will be written.

Example:

```
int A[100], h;
...
A[12] = h + A[15];
```

Assume $s1 contains h and $s2 contains the address of A. Note, the address of A[12] is &A + 4 × 12 = &A + 48 and the address of A[15] is &A + 4 × 15 = &A + 60,

```
lw   $t0, 60($s2)   # $t0 ← A[15]
add  $t0, $s1, $t0  # $t0 ← h + A[15]
sw   $t0, 48($s2)   # A[12] ← h + A[15]
```

2.3.5.3 MIPS Add Load Byte Unsigned Instruction

[Ref: MIPS Vol. II-A p. 153]

`lbu $dst,` $\texttt{off}_{15:0}$`($base)  # $dst ← `***`zero-ext`***`(MEM[$base+`***`sign-ext`***`(`$\texttt{off}_{15:0}$`)])`

12 This form of addressing is referred to as **base addressing mode** or **displacement addressing mode**. Most processors support base addressing mode.

The offset $off_{15:0}$ is a 16-bit signed integer which is sign-extended to form a 32-bit signed integer before adding it to $\$base$. The sum is the address in memory of a byte which is **zero-extended**[13] and loaded into $dst. `lbu` is typically used when dealing with character data.

Example:

```
char name[100];
...
char ch = name[0];
```

Assume $s1 contains the address of *name* and $t0 is associated with variable *ch*,

```
lbu  $t0, 0($s1)  # $t0 ← name[0]
```

2.3.5.4 MIPS Store Byte Instruction

[Ref: MIPS Vol. II-A p. 249]

`sb $src, off`$_{15:0}$`($base)  # MEM[$base+`***sign-ext***`(off`$_{15:0}$`)] ← $src`

The offset $off_{15:0}$ is a 16-bit signed integer which is sign-extended to form a 32-bit signed integer before adding it to $\$base$. The sum is the address in memory where the byte in $src will be written.

Example:

```
char name[100];
...
char ch = name[0] + name[5];
```

Assume $s1 contains the address of *name* and $s2 contains the address of *ch*,

```
lbu  $t0, 0($s1)    # $t0 ← name[0]
lbu  $t1, 5($s1)    # $t1 ← name[5]
add  $t0, $t0, $t1  # $t0 ← name[0] + name[5]
sb   $t0, 0($s2)    # ch ← name[0] + name[5]
```

2.3.6 Logical Instructions

2.3.6.1 MIPS Logical AND Instruction

```
and $dst, $src1, $src2  # $dst ← $src1 & $src2
```

Corresponding bits of $src1 and $src are AND-ed and the result is written to $dst.

Example: Suppose $s0 contains 0x3C and $s1 contains 0x5B. Then `and $s2, $s0, $s1` will write 0x18 to $s2.

2.3.6.2 MIPS Logical OR Instruction

```
or $dst, $src1, $src2  # $dst ← $src1 | $src2
```

Corresponding bits of $src1 and $src are OR-ed and the result is written to $dst.

2.3.6.3 MIPS Logical NOR Instruction

```
nor $dst, $src1, $src2  # $dst ← ~($src1 | $src2)
```

Corresponding bits of $src1 and $src are NOR-ed and the result is written to $dst.

13 Bits 31:8 of the 32-bit word will contain 0's and bits 7:0 of the word will contain the byte read from memory.

2.3.6.4 MIPS Logical XOR Instruction (Exclusive OR)

```
xor $dst, $src1, $src2  # $dst ← $src1 ^ $src2
```

Corresponding bits of $src1 and $src are XOR-ed and the result is written to $dst.

2.3.6.5 MIPS Logical NOT Instruction

MIPS does not have a logical NOT instruction, but the operation can be performed using NOR.

a	a NOR 0

```
nor $dst, $src1, $zero  # $dst ← ~$src1
```

2.3.6.6 MIPS Logical AND Immediate Instruction

```
andi $dst, $src1, imm15:0  # $dst ← $src1 & zero-ext(imm15:0)
```

2.3.6.7 MIPS Logical OR Immediate Instruction

```
ori $dst, $src1, imm15:0  # $dst ← $src1 | zero-ext(imm15:0)
```

2.3.6.8 MIPS Logical XOR Immediate Instruction

```
xori $dst, $src1, imm15:0  # $dst ← $src1 ^ zero-ext(imm15:0)
```

2.3.7 Shifting Instructions

Shifting instructions are used to move bits of a word left or right.

2.3.7.1 MIPS Shift Word Logical Left Instruction

```
sll $dst, $src1, shamt4:0  # $dst ← $src1 << shamt4:0
```

Example:

```
unsigned int x = 0x45;
unsigned int y = x << 4;
```

Assume the value of x is in $t0 and y is associated with $t1,

```
sll $t1, $t0, 4
```

Shifting an integer left n times is equivalent to multiplying by 2^n, although one has to be careful not to trash the sign bit of a signed integer.

```
int x = 1073741824;  // 0x4000_0000
int y = x << 1;      // One might think y = 2,147,483,648
```

Assume the value of x is in $t0 and y is associated with $t1.

```
sll $t1, $t0, 1  # $t1 = 0x8000_0000 = -2,147,483,648
```

2.3.7.2 MIPS Shift Word Logical Right Instruction

```
srl $dst, $src1, shamt4:0  # $dst ← $src1 >> shamt4:0
```

Shifting an unsigned integer right n times is equivalent to dividing by 2^n, but this is not necessarily true for signed integers.

```
int x = 2000;
int y = -2000;
int a = x >> 4;  // 2000 / 16 = 125
    0000 0000 0000 0000 0000 0111 1101 0000
        \    \    \    \    \    \    \
==> 0000 0000 0000 0000 0000 0000 0111 1101 = 125

int b = y >> 4;  // -2000 / 16 = -125
    1111 1111 1111 1111 1111 1000 0011 0000
        \    \    \    \    \    \    \
==> 0000 1111 1111 1111 1111 1111 1000 0011 = 268,435,331
```

2.3.7.3 MIPS Shift Word Right Arithmetic Instruction

`sra $dst, $src1, shamt`$_{4:0}$ `  # $dst ← $src1 >> shamt`$_{4:0}$ `(preserves sign)`

```
int y = -2000;
int b = y >> 4;
    1111 1111 1111 1111 1111 1000 0011 0000
        \    \    \    \    \    \    \
==> 1111 1111 1111 1111 1111 1111 1000 0011 = -125
```

2.3.8 Pseudoinstructions

Pseudoinstructions are used to form small sequences of common and reusable assembly language instructions. A **pseudoinstruction** is a virtual instruction implemented by the assembler and is just a sequence of one or more physical (hardware-implemented) instructions.

2.3.8.1 Move Pseudoinstruction

```
move $dst, $src  # $dst ← $src
```

Moves the value from the $src register to the $dst register. `move` can be implemented as,

```
add $dst, $src, $zero
```

2.3.8.2 Logical Not Pseudoinstruction

```
not $dst, $src  # $dst ← ~$src
```

Writes the one's complement of $src into $dst. `not` can be implemented as,

```
nor $dst, $src, $zero
```

2.3.8.3 Load Address Pseudoinstruction

```
la $dst, label  # $dst ← &label (where & is the C address-of operator)
```

A label is a name for a memory address. The `la` pseudoinstruction simply loads the address of *label* into $dst and can be implemented as,

```
lui $dst, label31:16          # $dst31:16 ← label31:16; $dst15:0 ← 0
ori $dst, $dst, label15:0     # $dst15:0 ← label15:0
```

where `lui` (load upper immediate) loads the 16 most significant bits of *label* into the 16 most significant bits of $dst (while the low 16 bits are cleared).

2.3.8.4 Load Immediate Pseudoinstruction

`li $dst, imm31:0  # $dst ← imm31:0`

Loads the 32-bit signed immediate $imm_{31:0}$ into $dst. `li` can be implemented as,

```
lui $dst, imm31:16    # $dst31:16 ← imm31:16; $dst15:0 ← 0
ori $dst, imm15:0     # $dst15:0 ← imm15:0
```

Note that it is faster to load a 16-bit signed immediate using the `addi` instruction,

```
addi $t0, $zero, 12576  # $t0 ←12,576
```

because `li` will take two clock cycles as opposed to only one for `addi`.

2.3.8.5 Negate Pseudoinstruction

`neg $dst, $src  # dst ← -$src`

$src is a 32-bit signed integer, the negation of which is written into $dst. `neg` is implemented as,

```
sub $dst, $zero, $src
```

2.3.8.6 No Operation Pseudoinstruction

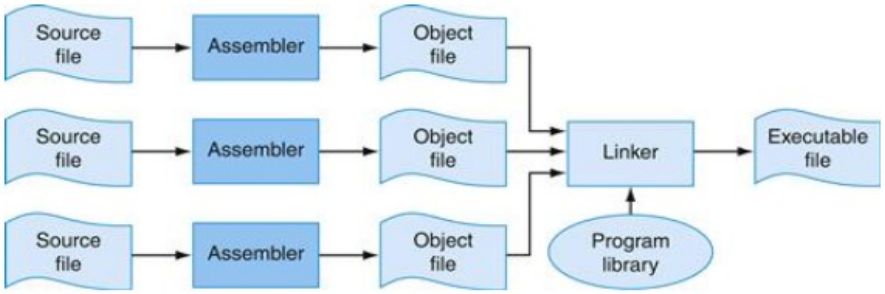
`nop  # does nothing`

A `nop` instruction does not do anything, i.e., it does not change the state of the system and although it may seem like a silly instruction, since it does not do anything, nop's have their purposes and probably all processor architectures implement this instruction. nop can be implemented as,

```
sll $zero, $zero, 0  # $zero (cannot be written) ← 0 << 0
```

2.4 MIPS Assembly Language Programming and MARS

[Ref: Textbook Appendix A; MIPS Assembly Language Programmer's Guide] A software tool called an **assembler** reads an assembly language source code file and writes an object code file containing the equivalent machine language instructions. The **linker** is a software tool that combines multiple object code files (possibly including object code from system libraries) to create a single **executable file**, Fig. A.1.1,



MARS contains an assembler and linkers that will build our executable files.

2.4.1 Labels

A **label** is a name associated with a memory location. In MARS, labels may be written using lower and upper case letters, digits, and the underscore character, and cannot start with a digit. Labels are case-sensitive, so *one*, *One*, and *ONE* are three separate labels.

2.4.2 Directives

Most assembly languages include **directives** which are not assembly language code, but rather are are like "messages" which control the assembler. In the MIPS documentation, these are referred to as *pseudo op-codes* but I will call them directives so as to not get you confused regarding the difference between *pseudoinstruction* and *pseudo op-code*.

2.4.2.1 .text Directive

Syntax: `.text`

Creates a section called the text section, which is the section where instructions are written, e.g.,

```
.text
main:
    addi    $v0, $zero, 4   # $v0 ← SysPrintStr service code
    la      $a0, hello_str  # $a0 ← addr of "Hello world.\n"
    syscall                 # Call SysPrintStr($a0)
```

When a source code file is assembled, the instructions which are in the *.text* section of the source code file are output to the *.text* section of the object code file (at least in Unix and Unix-like OS's). The linker, which combines all of the *.o* files together, knows that bytes in the *.text* section represent instructions and it will place all of those instructions in a section named *.text* in the binary executable file. The *loader* (the program in your OS that loads a program from secondary storage into RAM for execution) will placed the instructions in the *.text* section of the binary in a memory region allocated just for instructions.

2.4.2.2 .asciiz Directive

Syntax: `.asciiz` ***`string`***

Allocates space in memory for the characters of *string*. A null character[14] will be placed in memory following the last character of *string*. For example,

```
hello_str: .asciiz "Hello world.\n"  # Allocates 14 bytes of memory
```

2.4.2.3 .data Directive

Syntax: `.data`
Creates a section called the data section, which is the section where string literals and global data are allocated, e.g.,

```
.data
hello_str: .asciiz "Hello world.\n"  # Allocates 14 bytes in the data section
```

2.4.2.4 .byte Directive

Syntax: `.byte` ***value***$_1$ `[,` ***value***$_2$`...]` or `.byte` ***value*** `:` ***n***
The first form allocates one or more bytes in the data section, the number depending on the number of values that are listed. Each byte will be initialized to the corresponding value. The second form allocates n bytes in the data section with each byte initialized to (the same) *value*. For example,

```
.data
space_char: .byte  32          # Allocates 1 byte init'd to ' '
array1:     .byte  0, 1, 2, 3  # Allocates 4 bytes init'd to 0, 1, 2, and 3
array2:     .byte  -1 : 50     # Allocates 50 bytes, all init'd to -1
```

space_ char is allocated one byte and is initialized to 32, which is the ASCII value of the space character. *array1* is allocated as an array of four bytes, with *array1*[0] being initialized to 0, *array1*[1] to 2, ..., and *array1*[3] to 3. *array2* is allocated as an array of fifty bytes, with each array element initialized to -1.

2.4.2.5 .word Directive

Syntax: `.word` ***value***$_1$ `[,` ***value***$_2$`...]` or `.word` ***value*** `:` ***n***
Allocates word values in the data section, e.g.,

```
.data
m:      .word 100                 # Allocates 1 word (4 bytes) init'd to 100
array3: .word 10, 20, 30, 40, 50  # Allocates 5 words (20 bytes)
array4: .word 18 : 22             # Allocates 22 words (88 bytes) all init'd to 18
```

m is allocated one word and is initialized to 100. *array3* is an array of five words, with the elements being initialized to 10 (*array3*[0]), 20 (*array3*[1]), 30, 40, and 50 (*array3*[4]). *array4* is allocated as an array of 22 words, each initialized to 18.

2.4.2.6 .space Directive

Syntax: `.space` ***n***
Allocates n bytes of memory—each byte is initialized to 0—in the data section, e.g.,

```
.data
array5: .space 400  # Equivalent to: int array5[100] = { 0 }
```

2.4.2.7 .eqv Directive

Syntax: `.eqv` ***symbol expression***

14 The character with ASCII value 0. Another term is *null byte*.

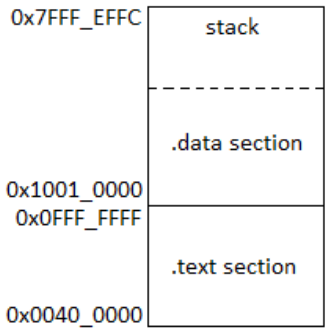
The assembler will replace occurrences of *symbol* in the source code file with *expression*. Can be used to define named constants to improve the readability of the code, e.g.,

```
.eqv MAX_TIME      100
.eqv SYS_PRINT_STR   4
```

If you know C, this is equivalent to `#define MAX_TIME 100` and `#define SYS_PRINT_STR 4`.

2.4.3 MIPS Memory Usage in the MARS Simulator

MARS supports three different memory configurations, which specify the memory addresses of things, including the .data and .text sections. To see the memory configurations, click **Main Menu | Settings | Memory Configuration**. We will use the **default** memory configuration, with this layout,



The text section (containing instructions) begins at 0x0040_0000, which is the address of the first instruction of the program. The data section (storeing string literals and global variables) starts at 0x1001_0000 with subsequent bytes being stored at increasing memory addresses. The runtime stack starts at 0x7FFF_EFFC which will be the initial value of the $sp stack pointer register. We will discuss the stack in more detail later.

2.4.4 MARS System Services

The MARS "operating system" provides several useful services. These are invoked by the MIPS `syscall` instruction with a **service code** which must be in $v0. Refer to the MARS Help system for a complete list of services. In general, the procedure to invoke a system service is:

1. Load the service code in $v0.
2. Load any required arguments in the argument registers $a0, $a1, $a2, and $a3.
3. Issue the `syscall` instruction.
4. Retrieve any return values from the specified registers, often $v0 and $v1.

Service	Code	Arguments	Result
Print integer	1	$a0 = integer to print	Prints contents of $a0 to console
Print string	4	$a0 = addr of null-terminated string	Prints string to console
Read integer	5		$v0 contains integer read from keyboard
Read string	8	$a0 = addr of string buffer $a1 = max num of chars to read	See MARS Help for more info
Exit	10		Terminates the program
Print char	11	$a0$_{7:0}$ = ASCII value of char	Prints ASCII char to console
Read char	12		$v0 contains the char read from keyboard

2.4.5 Example Program: Hello World

```
#-------------------------------------------------------------------------------------------------------
# FILE: HelloWorld.s
#
# DESCRIPTION
# Displays "Hello world." on the console.
#
# AUTHOR
# Kevin Burger (burgerk@asu.edu)
# Computer Science & Engineering
# Arizona State University
#-------------------------------------------------------------------------------------------------------

#=======================================================================================================
# MARS Service Codes
#=======================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_STR   4

#=======================================================================================================
# DATA
#=======================================================================================================
.data
s_hello: .asciiz "Hello world.\n"

#=======================================================================================================
# TEXT
#=======================================================================================================
.text
main:
# SysPrintStr("Hello world.\n")
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr() sercice code
    la      $a0, s_hello                # $a0 = addr of "Hello world.\n"
    syscall                             # Call SysPrintStr()

# SysExit()
    addi    $v0, $zero, SYS_EXIT        # $v0 = SysExit() service code
    syscall                             # Call SysExit()
```

2.4.6 Example Program: Color and Age

```
#-------------------------------------------------------------------------------------------------------
# FILE: ColorAge.s
#
# DESCRIPTION
# Prompts the user to enter their favorite color and age.
#
# AUTHOR
# Kevin Burger (burgerk@asu.edu)
# Computer Science & Engineering
# Arizona State University
#-------------------------------------------------------------------------------------------------------

#=======================================================================================================
# System Call Equivalents
#=======================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_STR   4
.eqv SYS_READ_INT    5
.eqv SYS_READ_STR    8
```

```
#=============================================================================================================
# DATA SECTION
#=============================================================================================================
.data
age:         .word    0                                 # int age = 0
s_age:       .asciiz  "How old are you? "                # char *s_age = "How old are you? "
s_color:     .asciiz  "What is your favorite color? "    # char *s_color = "What is your favorite color? "
s_response:  .space   50                                 # char s_response[50] = { '\0' };

#=============================================================================================================
# TEXT
#=============================================================================================================
.text
main:
# SysPrintStr("What is your favorite color? ")
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr service code
    la      $a0, s_color                # $a0 = addr of string
    syscall                             # Call SysPrintStr()

# SysReadString(s_response, 49)
    addi    $v0, $zero, SYS_READ_STR    # $v0 = SysReadString() service code
    la      $a0, s_response             # $a0 = addr of string buffer
    addi    $a1, $zero, 49              # $a1 = max num of chars to read
    syscall                             # Call SysReadString()

# SysPrintStr("How old are you? ")
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr() service code
    la      $a0, s_age                  # $a0 = addr of string
    syscall                             # Call SysPrintStr()

# age = ReadInt()
    addi    $v0, $zero, SYS_READ_INT    # $v0 = SysReadInt() code
    syscall                             # Call SysReadInt()
    la      $t0, age                    # $t0 = addr of age
    sw      $v0, 0($t0)                 # age = SysReadInt()

# SysExit()
exit:
    addi    $v0, $zero, SYS_EXIT        # $v0 = SysExit() service code
    syscall                             # Call SysExit()
```

2.5 Representing Instructions in the Computer

At the machine language level, all instructions and data are represented in binary. **Encoding** is the process of converting an assembly language instruction to the equivalent binary machine language instruction and encoding is the primary job of the assembler.

MIPS supports three basic instruction formats, each of which is 32-bits wide:

- R-format **R**egister. All operands for the instruction are in registers.
- J-format **J**ump. Encodes the `j` (jump) instruction.
- I-format **I**mmediate. The instruction involves an immediate (constant).

Since there are 32 general purpose registers, register numbers are encoded in instructions as 5-bit binary values representing the register number, e.g., register $t0 is $8 so it would be encoded in an instruction as 01000_2.

2.5.1 R-Format Instructions

The format of a R-format instruction is,

Field	Width	Instr Bits	Description
op	6	31:26	Opcode.
rs	5	25:21	First source register operand.
rt	5	20:16	Second source register operand.
rd	5	15:11	Destination register operand.
shamt	5	10:6	Shift amount (only used in shifting instruction; 0 otherwise).
funct	6	5:0	Function code; combined with opcode to uniquely identify instructions.

2.5.2 I-Format Instructions

The format of a I-format instruction is,

Field	Width	Instr Bits	Description
op	6	31:26	Opcode.
rs	5	25:21	First source register operand.
rt	5	20:16	Source or destination register operand.
$imm_{15:0}$	16	15:0	A 16-bit signed immediate.

2.5.3 J-Format Instructions

The format of a J-format instruction is,

Field	Width	Instr Bits	Description
op	6	31:26	Opcode = 000010 for J.
addr	26	25:0	Jump address.

2.5.4 Example Instruction Encodings

2.5.4.1 add $t0, $s1, $s2

[Ref: MIPS32 Vol II-A, p. 47]

```
Syntax:          _____________________________
add op field:    _____________________________
add funct field: _____________________________
rd               _____________________________
rs:              _____________________________
rt:              _____________________________
Encoding:        ____________________________________________________ = 0x________________
                 op        rs        rt        rd        shamt funct
```

2.5.4.2 addi $s4, $s3, 57

[Ref: MIPS32 Vol II-A, p. 49]

```
Syntax:          addi rt, rs, imm
addi op field:   001000
rs:              $s3 = $19 = 10011
rt:              $s4 = $20 = 10100
imm:             0000_0000_0011_1001
Encoding:        001000 10011 10100 0000000000111001 = 0x2274_0039
                 op     rs    rt    imm
```

2.5.4.3 lw $t3, 32($s3)

[Ref: MIPS32 Vol II-A, p. 170]

```
Syntax:          _____________________________
lw op field:     _____________________________
rs:              _____________________________
rt:              _____________________________
imm:             _____________________________
Encoding:        ____________________________________________________ = 0x________________
                 op        rs       rt       imm
```

2.5.4.4 sw $v1, -48($sp)

[Ref: MIPS32 Vol II-A, p. 281]

```
Syntax:          sw rt, offset(rs)
sw op field:     101011
rs:              $sp = $29 = 11101
rt:              $v1 = $3 = 00011
imm:             1111_1111_1101_0000
Encoding:        101011 11101 00011 1111111111010000 = 0xAFA3FFD0
                 op     rs    rt    imm
```

2.5.5 Example Instruction Decodings

[Ref: MIPS Vol II-A, Appendix A, Tables A.2 and A.3] **Decoding** is the process of converting a binary machine language instruction to the equivalent assembly language instruction. A **disassembler** is a program that will perform decoding.

2.5.5.1 0x0232_4027

Write 0x0232_4027 in binary: `000000 10 0011 0010 0100 0000 0010 0111`. Since the op field is 000000 we know this is an R-format instruction. Refer to *SPECIAL* table A.3 for the funct field, which is in bits 5:0:

`funct: 100111 = NOR`

The encoding of the NOR instruction can be found on p. 223.

```
Syntax:          nor rd, rs, rt
Encoding:        000000  10001  10010  01000  00000  100111
                 op      rs     rt     rd     shamt  funct
rs:              10001 = $17 = $s1
rt:              10010 = $18 = $s2
rd:              01000 = $8 = $t0
Instruction:     nor $t0, $s1, $s2
```

2.5.5.2 0x001D_58C0

Write 0x001D_58C0 in binary: `000000 00 0001 1101 0101 1000 1100 0000`. Since the op field is 000000 we know this is an R-format instruction. Refer to *SPECIAL* table A.3 for the funct field, which is in bits 5:0:

`funct: 000000 = SLL`

The encoding of the SLL instruction can be found on p. 265.

```
Syntax:          sll rd, rt, shamt
Encoding:        000000  00000  11101  01011  00011  000000
                 op      rs     rt     rd     shamt  funct
rs:              unused
rt:              11101 = $29 = $sp
rd:              01011 = $11 = $t3
shamt:           00011 = 3
Instruction:     sll $t3, $sp, 3
```

2.6 Instructions for Making Decisions

[Ref: Textbook §2.6] To implement HLL **if** statements and **loops** requires instructions which perform the relational operations (<, >, ≤, ≥, ==, and !=) and based on the result of the comparison, execute one sequence of instructions or a different sequence.

2.6.1 MIPS Jump Instruction

```
j label  # PC ← label
```

A `j` instruction is a form of **unconditional branch**, i.e., we *always* jump to *label* and start executing the instructions there. This is equivalent to a HLL *goto statement*.

2.6.2 MIPS Branch Equal Instruction

```
beq $src1, $src2, label  # If $src1 == $src2 then PC ← label else PC ← PC + 4
```

A `beq` instruction is a form of **conditional branch**, i.e., we only start executing the instructions at *label* if the values in $src1 and $src2 are the same. In this case, we say that we "take" the branch.

2.6.3 MIPS Branch Not Equal Instruction

```
bne $src1, $src2, label  # If $src1 != $src2 then PC ← label else PC ← PC + 4
```

We take the branch if $src1 does not equal $src2.

2.6.4 MIPS Branching Pseudoinstructions

[Ref: MIPS32 Vol II-A p. 267] Several of the MIPS branching instructions are actually implemented as pseudoinstructions using `beq`, `bne`, and the `slt` instruction:

```
slt $dst, $src1, $src2  # If $src1 < $src2 then $dst ← 1 else $dst ← 0
```

Branch if Greater Than:	`bgt $src1, $src2, label`
Operation:	`if $src1 > $src2 PC ← label else PC ← PC + 4`
Implemented as:	`slt $at, $src2, $src1` `bne $at, $zero, label`
Branch if Less Than:	`blt $src1, $src2, label`
Operation:	`if $src1 < $src2 PC ← label else PC ← PC + 4`
Implemented as:	`slt $at, $src1, $src2` `bne $at, $zero, label`
Branch if Greater Than or Equal:	`bge $src1, $src2, label`
Operation:	`if $src1 ≥ $src2 PC ← label else PC ← PC + 4`
Implemented as:	`slt $at, $src1, $src2` `beq $at, $zero, label`
Branch if Less Than or Equal:	`ble $src1, $src2, label`
Operation:	`if $src1 ≤ $src2 PC ← label else PC ← PC + 4`
Implemented as:	`slt $at, $src2, $src1` `beq $at, $zero, label`

2.6.5 Implementing HLL If Statements

Consider a C **if** statement, which has this syntax,

```
if (cond) {
   true-clause-stmts
}
```

This statement can be implemented in assembly language using this pseudocode,

```
   branch if cond is false to end_if
   true-clause-stmts
end_if:
```

Example:

```
if (x != y) {
   a = 1;
}
```

Suppose the value of x is in $t0, the value of y is in $t1, and a is associated with $s0,

```
   beq    $t0, $t1, end_if  # if x == y goto end_if
   addi   $s0, $zero, 1     # a = 1
end_if:
```

Notice that the C relational operator was `!=` but in the assembly language code we implemented ==. In general, if we have a C **if** statement of the form,

```
if (var1 op var2) { ... }
```

Then the assembly language translation will be of the form (assume *var1* is in $t0 and *var2* is in $t1),

```
   bop    $t0, $t1, end_if  # if (var1 op var2) is false goto end_if
   ...
end_if:
```

where $\overline{\text{op}}$ means the opposite of op, e.g., if op is $<=$ then $\overline{\text{op}}$ is $>$.

2.6.6 Implementing If-Else Statements

Consider a C **if-else** statement,

```
if (cond) {
   true-clause-stmts
} else {
   false-clause-stmts
}
```

which can be implemented in assembly language using this pseudocode,

```
   branch if cond is false to false_clause
   true-clause-stmts
   j end_if
false_clause:
   false-clause-stmts
end_if:
```

An alternative way of implementing the **if else** statement is to write the instructions for the false clause above those of the true clause,

```
      branch if cond is true to true_clause
      false-clause-stmts
      j end_if
   true_clause:
      true-clause-stmts
   end_if:
```

Example:

```
if (i == j) {
   f = g + h;
} else {
   f = g - h;
}
```

Suppose variable f is associated with $s0, the value of g is in $s1, the value of h is in $s2, the value of i is in $s3, and the value of j is in $s4,

```
      bne   $s3, $s4, false_clause  # if i != j goto false_clause
      add   $s0, $s1, $s2           # f = g + h
      j     end_if                  # Jump over false clause
   false_clause:                    # Come here if i != j
      sub   $s0, $s1, $s2           # f = g - h
   end_if:                          # True clause jumps here
```

Alternatively, writing the instructions for the false clause first,

```
      beq  $s3, $s4, true_clause    # If i == j goto true_clause
      sub  $s0, $s1, $s2            # f = g - h
      j    end_if                   # Jump over true clause
   true_clause:                     # Come here if i == j
      add  $s0, $s1, $s2            # f = g + h
   end_if:                          # False clause jumps here
```

2.6.7 Implementing a While Loop

Consider a C **while** loop,

```
while (cond) {
   loop-body
}
```

To write a **while** loop in assembly language, it is helpful to recognize that we can rewrite a C **while** loop as an **if** statement and a **goto**,

```
while (i < 10) {              loop_begin:
   a += 2 * i;                   if (i >= 10) goto end_loop;
   ++i;              ➜           a += 2 * i;
}                                ++i;
                                 goto loop_begin;
                              end_loop:
```

A **goto** statement is implemented in MIPS assembler as a `j` instruction and we just discussed how to implement an **if** statement in assembler. Therefore, the assembly language translation of a C **while** loop would be,

```
loop_begin:
   branch if cond is false to end_loop
   loop-body
   j loop_begin
end_loop:
```

Alternatively, we can write the code that checks the loop condition at the end of the loop,

```
   j check_cond
loop_begin:
   loop-body
check_cond:
   branch if cond is true to loop_begin
```

Example:

```
int i = 1;
while (i != 10) {
   ...
   ++i;
}
```

We will store the value of i in $t0 and 10 in $s0,

```
   addi  $t0, $zero, 1         # i ($t0)  = 1
   addi  $s0, $zero, 10        # $s0 = 10
loop_begin:
   beq   $t0, $s0, end_loop    # if i == 10 then drop out of loop
   ...
   addi  $t0, $t0, 1           # Increment i
   j     loop_begin            # Continue looping
end_loop:                      # Come here when i == 10
```

Checking the loop condition at the bottom of the loop,

```
   addi  $t0, $zero, 1         # i ($t0)  = 1
   addi  $s0, $zero, 10        # $s0 = 10
   j     check_cond            # Go check the loop condition
loop_begin:                    # Come here when i != 10
   ...
   addi  $t0, $t0, 1           # Increment i
check_cond:                    # Check the loop condition
   bne   $t0, $s0, loop_begin  # If i != 10 execute the loop body
```

2.6.8 Implementing a For Loop

Consider a C **for** loop which has this syntax,

```
for (initialization-expression; conditional-expression; update-expression) {
   loop-body
}
```

A C **for** loop can always be rewritten as a **while** loop,

```
initialization-expression
while (conditional-expression) {
   loop-body
   update-expression
}
```

and since we know how to write an assembly language **while** loop, we now know how to write a **for** loop.

Example: Write a loop that executes 10 times.

```
for (int i = 1; i <= 10; ++i) {
   ...
}
```

Rewriting the **for** loop as a **while** loop,

```
int i = 1;
while (i <= 10) {
   ...
   ++i;
}
```

Translating to assembly language:

```
   addi  $t0, $zero, 1        # i ($t0)  = 1
   addi  $s0, $zero, 10       # $s0 = 10
loop_begin:
   bgt   $t0, $s0, end_loop   # If i > 10 drop out of loop
   ...
   addi  $t0, $t0, 1          # ++i
   j     loop_begin           # Go check loop condition again
end_loop:
```

2.7 Example MIPS Assembly Language Programs

2.7.1 EvenOdd.s

```
#*********************************************************************************************************
# FILE: EvenOdd.s
#
# DESCRIPTION
# Implements a HLL if-else statement to print a message telling the user whether an integer is even or
# odd.
#
# PSEUDOCODE
# Function main()
#     SysPrintStr("Enter an integer? ")
#     n = SysReadInt()
#     SysPrintInt(n)
#     If (n % 2 == 0) {
#         SysPrintStr(" is even.\n")
#     Else
#         SysPrintStr(" is odd.\n")
#     EndIf
#     SysExit()
# End Function main
#
# NOTES
# A binary integer that is even will have bit 0 cleared to 0; if bit 0 is 1, the binary integer is odd.
# Therefore, we can determine if n is even or odd by AND-ing n with 1 and checking to see if the result
# is 0 (n is even) or 1 (n is odd). This is much faster than executing a DIV instruction.
#*********************************************************************************************************

#=========================================================================================================
# System Call Equivalents
#=========================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_INT   1
.eqv SYS_PRINT_STR   4
.eqv SYS_READ_INT    5

#=========================================================================================================
# DATA SECTION
#=========================================================================================================
.data
s_even:      .asciiz  " is even.\n"         # char *s_even = " is even.\n"
s_odd:       .asciiz  " is odd.\n"          # char *s_odd = " is odd.\n"
s_prompt:    .asciiz  "Enter an integer? "  # char *s_prompt = "Enter an integer? "

#=========================================================================================================
# TEXT
#=========================================================================================================
.text
main:
# SysPrintStr("Enter an integer? ")
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr service code
    la      $a0, s_prompt               # $a0 = addr of string to print
    syscall                             # Call SysPrintStr()

# n = SysReadInt()
    addi    $v0, $zero, SYS_READ_INT    # $v0 = SysReadInt service code
    syscall                             # $v0 = SysReadInt()
    move    $a0, $v0                    # n ($a0) = SysReadInt()
```

```
# SysPrintInt(n)
    addi    $v0, $zero, SYS_PRINT_INT   # $v0 = SysPrintInt service code
    syscall                             # SysPrintInt(n)

# if (n % 2 == 0) ...
    andi    $a0, $a0, 1                 # bit 0 of $a0 will be 0 if n is even or 1 if n is odd
    bne     $a0, $zero, false_clause    # if n is odd goto false_clause

# SysPrintStr(" is even.\n")
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr service code
    la      $a0, s_even                 # $a0 = addr of string to print
    syscall                             # Call SysPrintStr()
    j       end_if                      # Skip over false clause

# else SysPrintStr(" is odd.\n")
false_clause:
    addi    $v0, $zero, SYS_PRINT_STR   # $v0 = SysPrintStr service code
    la      $a0, s_odd                  # $a0 = addr of string to print
    syscall                             # Call SysPrintStr()

# SysExit()
end_if:
    addi    $v0, $zero, SYS_EXIT        # $v0 = SysExit service code
    syscall                             # Call SysExit()
```

2.7.2 10to1.s

```
#*********************************************************************************************************
# FILE: 10to1.s
#
# DESCRIPTION
# Implements a HLL for statement to print the numbers 10, 9, 8, ..., 0 on the console.
#
# PSEUDOCODE
# Function main()
#     For (i = 10; i >= 0; --i) Do
#         SysPrintInt(i)
#         SysPrintChar(' ')
#     EndFor
#     SysExit()
# End Function main
#
# Rewriting the for loop as a while loop:
#
# Function main()
#     i = 10
#     While (i >= 0) Do
#         SysPrintInt(i)
#         SysPrintChar(' ')
#         --i
#     EndWhile
#     SysExit()
# End Function main
#
# And rewriting the while loop as an if statement and a goto:
#
# Function main()
#     i = 10
```

```
#   loop_begin:
#     If (i < 0) Then Goto end_loop
#         SysPrintInt(i)
#         SysPrintChar(' ')
#         --i
#         Goto loop_begin
#   end_loop:
#     SysExit()
# End Function main
#*********************************************************************************************************
#=========================================================================================================
# System Call Equivalents
#=========================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_CHAR 11
.eqv SYS_PRINT_INT   1
#=========================================================================================================
# TEXT
#=========================================================================================================
.text
main:
    addi    $t0, $zero, 10               # i = 10
loop_begin:
    blt     $t0, $zero, end_loop         # If (i < 10) Then Goto end_loop
    addi    $v0, $zero, SYS_PRINT_INT    # $v0 = SysPrintInt service code
    move    $a0, $t0                     # $a0 = i
    syscall                              # SysPrintInt(i)
    addi    $v0, $zero, SYS_PRINT_CHAR   # $v0 = SysPrintChar service code
    addi    $a0, $zero, 32               # $a0 = ASCII value of space char
    syscall                              # SysPrintChar(' ')
    addi    $t0, $t0, -1                 # --i
    j       loop_begin                   # Goto loop_begin
end_loop:
    addi    $v0, $zero, SYS_EXIT         # $v0 = SysExit service code
    syscall                              # SysExit()
```

2.7.3 Prime1.s

```
#*********************************************************************************************************
# FILE: Prime1.s
#
# DESCRIPTION
# Prompts the user to enter an integer and prints a message telling the user if the integer is prime or
# composite.
#
# NOTE
# This is version 1 and is not optimized for speed.
#
# PSEUDOCODE (STRUCTURED)
# int div, is_prime, n
# Function main()
#     SysPrintStr("Enter an integer (>= 2)? ")
#     n = SysReadInt()
#     If (n == 2) Then
#         is_prime = true
#     ElseIf (n % 2 == 0) Then
#         is_prime = false
```

```
#     Else
#         is_prime = true
#         div = 3
#         While (div < n && is_prime == true) Do
#             If (n % div == 0) Then
#                 is_prime = false
#             Else
#                 div += 2
#             EndIf
#         EndWhile
#     EndIf
#     If (is_prime == true) Then
#         SysPrintInt(n)
#         SysPrintStr(" is prime.\n")
#     Else
#         SysPrintInt(n)
#         SysPrintStr(" is composite.\n")
#     EndIf
#     SysExit()
# End Function main
#
# It is helpful when converting the pseudocode into assembly language to first convert the "structured"
# pseudocode (using if statements, if-else statements, and while loops) into "unstructured" pseudocode
# using only if statements and gotos. The reason is that the unstructured pseudocode maps more directly
# into assembly language (assembly language is a highly unstructured programming language).
#
# PSEUDOCODE (UNSTRUCTURED)
# int div, is_prime, n
# Function main()
#     SysPrintStr("Enter an integer (>= 2)? ")
#     n = SysReadInt()
#     If (n != 2) Then Goto false1
#     is_prime = true
#     Goto end_if1
#   false1:
#     If (n % 2 != 0) Then Goto false2
#     is_prime = false
#     Goto end_if1
#   false2:
#     is_prime = true
#     div = 3
#   loop_begin:
#     If (div >= n) Then Goto end_loop
#     If (is_prime == false) Then Goto end_loop
#     If (n % div != 0) Then Goto false3
#     is_prime = false
#     Goto end_if2
#   false3:
#     div += 2
#   end_if2:
#     goto loop_begin
#   end_loop:
#   end_if1:
#     If (is_prime != true) Then Goto false4
#     SysPrintInt(n)
#     SysPrintStr(" is prime.\n")
#     Goto end_if3
```

```
#   false4:
#     SysPrintInt(n)
#     SysPrintStr(" is composite.\n")
#   end_if3:
#     SysExit()
# End Function main
#*****************************************************************************************************

#=====================================================================================================
# System Call Equivalents
#=====================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_INT   1
.eqv SYS_PRINT_STR   4
.eqv SYS_READ_INT    5
.eqv SYS_READ_STR    8

#=====================================================================================================
# Other Equivalents
#=====================================================================================================
.eqv FALSE 0
.eqv TRUE  1

#=====================================================================================================
# DATA SECTION
#=====================================================================================================
.data
div:       .space   4  # int div = 0
is_prime:  .space   4  # int is_prime = FALSE
n:         .space   4  # int n = 0

s_prompt:  .asciiz  "Enter an integer (>= 2)? "
s_prime:   .asciiz  " is prime.\n"
s_comp:    .asciiz  " is composite.\n"

#=====================================================================================================
# TEXT SECTION
#=====================================================================================================
.text
main:
# Load $s0 with the address of global variable div. Globals div, is_prime and n will be at 0($s0), 4($s0),
# and 8($s0), respectively. When we use a register this way, it is referred to as a "base" register.
    la      $s0, div

# SysPrintStr("Enter an integer (>= 2)? ");
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_prompt              # $a0 = addr of s_prompt
    syscall                            # Call SysPrintStr()

# n = SysReadInt();
    addi    $v0, $zero, SYS_READ_INT   # $v0 = SysReadInt code
    syscall                            # Call SysReadInt()
    sw      $v0, 8($s0)                # n = SysReadInt()

# if (n != 2) Then Goto false1
    lw     $t0, 8($s0)                 # $t0 = n
    addi   $t1, $zero, 2               # $t1 = 2
    bne    $t0, $t1, false1            # if n != 2 goto false1
    addi   $t0, $zero, TRUE            # $t0 = TRUE
    sw     $t0, 4($s0)                 # is_prime = TRUE
    j      end_if1                     # goto end_if1
```

```
false1:
# if (n % 2 != 0) Then Goto false2
    lw     $t0, 8($s0)                # $t0 = n
    addi   $t1, $zero, 2              # $t1 = 2
    div    $t0, $t1                   # HI = n % div
    mfhi   $t0                        # $t0 = n % div
    bne    $t0, $zero, false2         # if n % div != 0 goto false2
    addi   $t0, $zero, FALSE          # $t0 = FALSE
    sw     $t0, 4($s0)                # is_prime = FALSE
    j      end_if1                    # goto end_if1

false2:
# is_prime = TRUE
    addi    $t0, $zero, TRUE          # $t0 = TRUE
    sw      $t0, 4($s0)               # is_prime = TRUE

# div = 3
    addi    $t0, $zero, 3             # $t0 = 3

loop_begin:
    sw      $t0, 0($s0)               # Write $t0 to div

# If (div >= n) Then Goto end_loop
    lw      $t1, 8($s0)               # $t1 = n
    lw      $t0, 0($s0)               # $t0 = div
    bge     $t0, $t1, end_loop        # if div >= n goto end_loop

# If (is_prime == false) Then Goto end_loop
    lw      $t1, 4($s0)               # $t1 = is_prime
    beq     $t1, $zero, end_loop      # if is_prime == FALSE goto end_loop

# Calculate n % div
    lw      $t0, 8($s0)               # $t0 = n
    lw      $t1, 0($s0)               # $t1 = div
    div     $t0, $t1                  # HI = n % div
    mfhi    $t0                       # $t0 = n % div

# if n % div != 0 Then Goto false3
    bne     $t0, $zero, false3        # if n % div != 0 goto false3
    sw      $zero, 4($s0)             # is_prime = FALSE
    j       end_if2                   # Goto endif_2

false3:
# div += 2
    lw      $t0, 0($s0)               # $t0 = div
    addi    $t0, $t0, 2               # $t0 = div + 2
    sw      $t0, 0($s0)               # div += 2

# Goto loop_begin
end_if2:
    j       loop_begin

end_loop:
end_if1:
# If (is_prime == false true) Then Goto false4
    lw      $t0, 4($s0)               # $t0 = is_prime
    beq     $t0, $zero, false4        # if is_prime == false goto false4

# SysPrintInt(n)
    addi    $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    lw      $a0, 8($s0)               # $a0 = n
    syscall                           # SysPrintInt(n)
```

```
# SysPrintStr(" is prime.\n")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_prime               # $a0 = addr of s_prime
    syscall                            # Call SysPrintStr()
    j       end_if3                    # Skip over false clause

false4:
# SysPrintInt(n)
    addi    $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    lw      $a0, 8($s0)                # $a0 = n
    syscall                            # SysPrintInt(n)

# SysPrintStr(" is composite.\n")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_comp                # $a0 = addr of s_comp
    syscall                            # Call PrintString()

end_if3:
# Terminate the program.
    addi    $v0, $zero, SYS_EXIT       # $v0 = SysExit service code
    syscall                            # Call SysExit()
```

2.7.4 Prime2.s

```
#*********************************************************************************************************************
# FILE: Prime2.s
#
# DESCRIPTION
# Prompts the user to enter an integer and determine if the integer is prime or composite.
#
# See Prime1.s.
#
# Prime1.s is not terribly optimized. An optimizing compiler would be able to significant reduce the
# instruction count of this program. We will optimize the code this way:
#
# 1. We do not actually store the values of the global variables in memory but rather keep them in
#    registers: $s0 is div, $s1 is is_prime, and $s2 is n.
# 2. We only write the code to print n once.
# 3. Performing a DIV instruction to determine if n % 2 == 0 is slow. If n is even then bit 0 will be 1
#    and if n is odd, then bit 0 will be 1. We can determine this using ANDI.
#
# REGISTER USAGE
# $s0 - variable div
# $s1 - variable is_prime
# $s2 - variable n
#*********************************************************************************************************************

#=====================================================================================================================
# System Call Equivalents
#=====================================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_INT   1
.eqv SYS_PRINT_STR   4
.eqv SYS_READ_INT    5
.eqv SYS_READ_STR    8

#=====================================================================================================================
# Other Equivalents
#=====================================================================================================================
.eqv FALSE 0
.eqv TRUE  1
```

```
#=============================================================================================================
# DATA SECTION
#=============================================================================================================
.data
s_prompt:  .asciiz  "Enter an integer (>= 2)? "
s_prime:   .asciiz  " is prime.\n"
s_comp:    .asciiz  " is composite.\n"

#=============================================================================================================
# TEXT SECTION
#=============================================================================================================
.text
main:
# SysPrintStr("Enter an integer (>= 2)? ");
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_prompt              # $a0 = addr of s_prompt
    syscall                            # Call SysPrintStr()

# n = SysReadInt();
    addi    $v0, $zero, SYS_READ_INT   # $v0 = SysReadInt code
    syscall                            # Call SysReadInt()
    move    $s2, $v0                   # div = SysReadInt()

# if (n != 2) Then Goto false1
    addi   $t1, $zero, 2               # $t1 = 2
    bne    $s2, $t1, false1            # if n != 2 goto false1
    addi   $s1, $zero, TRUE            # is_prime = TRUE
    j      end_if1                     # goto end_if1

false1:
# if (n % 2 != 0) Then Goto false2
    andi   $t0, $s2, 1                 # $t0 = n % 2
    bne    $t0, $zero, false2          # if n % div != 0 goto false2
    add    $s1, $zero, $zero           # is_prime = FALSE
    j      end_if1                     # goto end_if1

false2:
# is_prime = TRUE
    addi    $s1, $zero, TRUE           # is_prime = TRUE

# div = 3
    addi    $s0, $zero, 3              # div = 3

loop_begin:
# If (div >= n) Then Goto end_loop
    bge     $s0, $s2, end_loop         # if div >= n goto end_loop

# If (is_prime == false) Then Goto end_loop
    beq     $s1, $zero, end_loop       # if is_prime == FALSE goto end_loop

# Calculate n % div
    div     $s2, $s0                   # HI = n % div
    mfhi    $t0                        # $t0 = n % div

# if n % div != 0 Then Goto false3
    bne     $t0, $zero, false3         # if n % div != 0 goto false3
    add     $s1, $zero, $zero          # is_prime = FALSE
    j       end_if2                    # Goto endif_2

false3:
# div += 2
    addi    $s0, $s0, 2                # $t0 = div + 2
```

```
# Goto loop_begin
end_if2:
    j       loop_begin

end_loop:
end_if1:
# SysPrintInt(n)
    addi    $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    move    $a0, $s2                   # $a0 = n
    syscall                            # SysPrintInt(n)

# If (is_prime == false true) Then Goto false4
    beq     $s1, $zero, false4         # if is_prime == false goto false4

# SysPrintStr(" is prime.\n")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_prime               # $a0 = addr of s_prime
    syscall                            # Call SysPrintStr()
    j       end_if3                    # Skip over false clause

false4:
# SysPrintStr(" is composite.\n")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_comp                # $a0 = addr of s_comp
    syscall                            # Call PrintString()

end_if3:
# Terminate the program.
    addi    $v0, $zero, SYS_EXIT       # $v0 = SysExit service code
    syscall                            # Call SysExit()
```

Comparing the instruction statistics for the unoptimized version and the optimized version (with $n = 97$) we can see that the optimized code resulted in 49% fewer instructions being executed. More importantly, we eliminated 385 very slow `lw` and `sw` instructions.

	Prime1.s	*Prime2.s*
Total number of instructions executed	789	402
ALU instructions	158 (20%)	154 (38%)
Jump instructions	48 (6%)	48 (12%)
Branch instructions	145 (18%)	145 (37%)
Memory access instructions	385 (49%)	0 (0%)
Other instructions	53 (7%)	55 (14%)

2.8 Supporting Procedures in Computer Hardware

[Ref: Textbook §2.8] A **procedure** is the same thing as a **function**, **subroutine**, or **method**.

2.8.1 MIPS Jump and Link Instruction

To call a procedure in MIPS assembly language, we use the `jal` instruction, which places the return address (PC + 4) in $ra.

```
jal label  # $ra ← PC + 4; PC ← label
```

2.8.2 MIPS Jump Register Instruction

The `jr` instruction will cause control to begin executing instructions at the address in a register.

```
jr $reg    # PC ← $reg
```

`jr` is commonly used with $ra to return from a procedure.

2.8.3 Calling Procedures

When calling a procedure we generally have to perform these six steps (we may skip steps 1, 3, or 5 depending on the situation):

1. Place the arguments somewhere the callee can access them (skip if no input arguments).
2. Save the return address and change PC to begin executing the procedure (`jal` instruction).
3. The procedure executes code to allocate any local variables (skip if no defined local variables).
4. The procedure executes instructions to performs the desired task.
5. The procedure places the return value in a location where the caller can access it (skip if there is not a return value).
6. Change PC to go back to the return address (the `jr $ra` instruction).

By convention, certain MIPS registers are used for specific purposes during procedure calls:

$a0 - $a3	Arguments to the procedure
$v0 - $v1	Return values from the procedure
$ra	Return address (the address of the instruction following the `jal` instruction).

In this and subsequent sections we will discuss how to translate this C code to MIPS assembly language.

```
void foo() {
   int a = 4, b = 7, c;
   c = bar(a, b);
   printf("%d", c);
}

int bar(int x, int y) {
   int a = x + x;
   int b = y + y;
   return a + b;
}

void main() {
   foo();
}
```

In this program there are no global variables, and as we have seen, in MIPS assembly language we allocate the global variables in the `.data` section. Local variables—those defined in the procedure or as procedure parameters—are not allocated in the `.data` section as this would: (1) make them global, and (2) allocate the variables for the entire duration of the program. Remember, in HLL's, a local variable is not allocated until the procedure in which the variable is defined is called and is deallocated when the procedure returns.

2.8.3.1 Stack Frames

The standard way of handling local variables at the assembly language level is to use a **stack**. A stack is a LIFO (last in first out) data structure. The standard operations are push, pop, and peek:

```
push 2
push 4
push 7
pop
pop
push 8
peek
```

In MIPS, $sp is the stack pointer register and always contains the address in memory of the top item on the stack (in MARS, $sp is initialized to 0x7FFF_EFFC before your program begins execution). In MIPS—and this is true of most architectures—the stack grows **downward** in memory, so if $sp contains 0x7FFF_EFB0 then the top word on the stack is at address 0x7FFF_EFB0 and the word below the top word is at address 0x7FFF_EFB4. If a new word is pushed onto the stack, $sp would be changed to 0x7FFF-EFB0 - 4 = 0x7FFF_EFAC and the word being pushed would be written to that memory location.

MIPS does not have hardware instructions for executing push, pop, and peek, but they are easily written.

Push the contents of $reg onto the stack

```
addi  $sp, $sp, -4  # $sp ← $sp - 4
sw    $reg, 0($sp)  # MEM[$sp] ← $reg
```

Pop the top item on the stack into $reg

```
lw    $reg, 0($sp)  # $reg ← MEM[$sp]
addi  $sp, $sp, 4   # $sp ← $sp + 4
```

Peek the top item on the stack into $reg

```
lw    $reg, 0($sp)  # $reg ← MEM[$sp]
```

In assembly language, a called procedure will create a **stack frame** (also called an **activation frame** or **activation record**) at the beginning of its execution and will destroy the stack frame before returning. The stack frame will contain two things, each of which is optional:

1. The return address to the calling function (only necessary if the procedure does not call another[15]).
2. Storage locations for defined local variables (if there are any).

15 A procedure that does not call another procedure is called a leaf procedure.

Consider the translation of *foo*():

```
#-----------------------------------------------------------------------------------------------------------------
# We create the stack frame so it will look like this:
#
# +--------------+
# |  saved $ra   |  12($sp)
# +--------------+
# |   local c    |  8($sp)
# +--------------+
# |   local b    |  4($sp)
# +--------------+
# |   local a    |  $sp
# +--------------+
#-----------------------------------------------------------------------------------------------------------------
foo:
# Create stack frame.
    addi      $sp, $sp, -16              # Allocate room for 4 words: $ra and local vars a, b, and c
    sw        $ra, 12($sp)               # Save $ra

# int a = 4, b = 7, c;
    addi      $t0, $zero, 4              # $t0 = 4
    sw        $t0, 0($sp)                # a = 4
    addi      $t0, $zero, 7              # $t0 = 7
    sw        $t0, 4($sp)                # b = 7

# c = bar(a, b);
    lw        $a0, 0($sp)                # $a0 = a
    lw        $a1, 4($sp)                # $a1 = b
    jal       bar                        # Call bar(a, b)
    sw        $v0, 8($sp)                # c = bar(a, b)

# printf("%d", c);
    addi      $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    lw        $a0, 8($sp)                # $a0 = c
    syscall                              # Call SysPrintInt

# Destroy stack frame.
    lw        $ra, 12($sp)               # Restore $sp
    addi      $sp, $sp, 16               # Destroy stack frame
    jr        $ra                        # Return
```

Notice that local variables *a*, *b*, and *c* come into existence (they are **allocated** on the stack) when *foo*() begins executing and they go out of existence (they are are **deallocated**) when *foo*() returns.

2.8.3.2 Saving Registers

Suppose *foo*() was storing values in registers $t0 and $s3 before calling *bar*(). Suppose *bar*() writes to $t0 and $s3. When *bar*() returns, the values that *foo*() was storing in $t0 and $s3 are now gone because *bar*() **clobbered** them. How do we avoid this clobbering? In MIPS the convention is:

1. A callee can write to $t0-$t9 without saving them (these are temporary registers).
2. A callee cannot write to $s0-$s7 saving and restoring them (these are save temporary registers).

To put this in the perspective of the caller:

1. The caller must save any $t0-$t9 registers being used before calling a function (they are temporary).
2. The caller does not need to save $s0-$s7 before calling a function (the callee will save these).

What about other registers, e.g., the $v and $a registers? The standard MIPS convention is that $v and $a registers are treated like $t registers in that the caller must save them and the callee can freely use them. Summarizing:

Caller Must Save	**Callee Must Save**
$a0 - $a3	$s0 - $s7
$t0 - $t9	$ra
$v0 - $v1	$sp and $fp

Note: We do not need to save registers when performing a `syscall` because all registers are automatically saved and restored during a system call.

Here is the complete translation of the C code. Note that this program has no `.data` section because there are no global variables or string literals.

```
#=============================================================================================================
# System Call Equivalents
#=============================================================================================================
.eqv SYS_EXIT       10
.eqv SYS_PRINT_INT   1

.text
#-------------------------------------------------------------------------------------------------------------
# PROCEDURE: foo()
# We create the stack frame so it will look like this:
#
# +--------------+
# |  saved $ra   |  12($sp)
# +--------------+
# |   local c    |  8($sp)
# +--------------+
# |   local b    |  4($sp)
# +--------------+
# |   local a    |  $sp
# +--------------+
#-------------------------------------------------------------------------------------------------------------
foo:
# Create stack frame.
    addi      $sp, $sp, -16              # Allocate room for 4 words: $ra and local vars a, b, and c
    sw        $ra, 12($sp)               # Save $ra

# int a = 4, b = 7, c;
    addi      $t0, $zero, 4              # $t0 = 4
    sw        $t0, 0($sp)                # a = 4
    addi      $t0, $zero, 7              # $t0 = 7
    sw        $t0, 4($sp)                # b = 7

# c = bar(a, b);
    lw        $a0, 0($sp)                # $a0 = a
    lw        $a1, 4($sp)                # $a1 = b
    jal       bar                        # Call bar(a, b)
    sw        $v0, 8($sp)                # c = bar(a, b)
# printf("%d", c);
    addi      $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    lw        $a0, 8($sp)                # $a0 = c
    syscall                              # Call SysPrintInt
```

```
# Destroy stack frame and return.
    lw        $ra, 12($sp)              # Restore $sp
    addi      $sp, $sp, 16              # Destroy stack frame
    jr        $ra                       # Return

#-----------------------------------------------------------------------------------------------------------------
# PROCEDURE: bar()
# We create the stack frame so it will look like this:
#
# +--------------+
# |   local b    |  4($sp)
# +--------------+
# |   local a    |  $sp
# +--------------+
#
# Note that since bar() is a leaf procedure, there is no need to save and restore $ra.
#-----------------------------------------------------------------------------------------------------------------
bar:
# Create stack frame.
    addi      $sp, $sp, -8              # Allocate room for 2 words: local vars a and b

# a = x + x;
    add       $t0, $a0, $a0             # $t0 = x + x
    sw        $t0, 0($sp)               # a = x + x

# b = y + y;
    add       $t0, $a1, $a1             # $t0 = y + y
    sw        $t0, 4($sp)               # b = y + y

# Destroy stack frame and return a + b
    lw        $t0, 0($sp)               # $t0 = a
    lw        $t1, 4($sp)               # $t1 = b
    add       $v0, $t0, $t1             # $v0 = a + b
    addi      $sp, $sp, 8               # Destroy stack frame
    jr        $ra                       # Return

#-----------------------------------------------------------------------------------------------------------------
# PROCEDURE: main()
# In MARS (using the default memory configuration) on entry to main(), $sp will be 0x7FFF_EFFC. Since
# main() does not allocate any local variables and does not return, we do not need to create a stack
# frame.
#-----------------------------------------------------------------------------------------------------------------
main:
    jal       foo                       # Call foo()
    addi      $v0, $zero, SYS_EXIT      # $v0 = SysExit service code
    syscall                             # SysExit()
```

2.9 Example MIPS Assembly Language Program: Procedures

In this section, we shall examine a complete MIPS program that involves multiple procedures. What the program does is quite simple, although the assembly language code is a bit lengthy. The program prompts the user to enter the numerators and denominators of two fractions and then prints the quotient after dividing the first fraction by the second fraction. For example, here is a sample run,

```
Enter numerator? 2
Enter denominator? 4
Enter numerator? -3
Enter denominator? 7
2/4 / -3/7 = -14/12
```

The program consists of a main procedure and five other procedures: *div_fraction*(), *invert_fraction*(), *mult_fraction*(), *print_fraction*(), and *read_fraction*(). Study the pseudocode in the program header and then study each procedure to see how it is implemented at the assembly language level. Pay particular attention to how arguments are passed (in the $a register), how values are returned (in the $v registers), how the stack frame is allocated and deallocated the beginning and end of each procedure, and how local variables are allocated and accessed within the stack frame.

```
#*********************************************************************************************************
# FILE: Fraction.s
#
# DESCRIPTION
# Prompts the user to enter two fractions, computes the quotient of dividing the two fractions, and prints
# the results.
#
# PSEUDOCODE
# Function main() Returns Nothing
#     int num1, den1, num2, den2, quot_num, quot_den
#     num1, den1 = read_fraction()
#     num2, den2 = read_fraction()
#     quot_num, quot_den = div_fraction(num1, den1, num2, den2)
#     print_fraction(num1, den1)
#     SysPrintStr(" / ")
#     print_fraction(num2, den2)
#     SysPrintStr(" = ");
#     print_fraction(quot_num, quot_den)
#     SysPrintChar('\n')
#     SysExit
# End Function main
#
# Function div_fraction(num1, den1, num2, den2) Returns quot_num, quot_den
#     int inv_num, inv_den, quot_num, quot_den
#     inv_num, inv_den = invert_fraction(num2, den2)
#     quot_num, quot_den = mult_fraction(num1, den1, inv_num, inv_den)
#     Return quot_num, quot_den
# End Function div_fraction
#
# Function invert_fraction(num, den) Returns inv_num, inv_den
#     int inv_num, inv_den
#     inv_num = den
#     inv_den = num
```

```
#     If (inv_den < 0) Then
#         inv_num = -inv_num
#         inv_den = -inv_den
#     End If
#     Return inv_num, inv_den
# End Function invert_fraction
#
# Function mult_fraction(num1, den1, num2, den2) Returns prod_num, prod_den
#     int prod_num, prod_den
#     prod_num = num1 * num2
#     prod_den = den1 * den2
#     Return prod_num, prod_den
# End Function mult_fraction
#
# Function print_fraction(num, den)
#     SysPrintInt(num)
#     SysPrintChar('/')
#     SysPrintInt(den)
# End Function print_fraction
#
# Function read_fraction() Returns num, den
#     int num, den
#     SysPrintStr("Enter numerator? ")
#     num = SysReadInt()
#     SysPrintStr("Enter denomerator? ")
#     den = SysReadInt()
#     Return num, den
# End Function read_fraction
#*****************************************************************************************************************

#=================================================================================================================
# System Call Equivalents
#=================================================================================================================
.eqv SYS_EXIT        10
.eqv SYS_PRINT_CHAR  11
.eqv SYS_PRINT_INT    1
.eqv SYS_PRINT_STR    4
.eqv SYS_READ_INT     5
.eqv SYS_READ_STR     8

#=================================================================================================================
# DATA SECTION
#=================================================================================================================
.data
s_num_prompt:  .asciiz  "Enter numerator? "
s_den_prompt:  .asciiz  "Enter denominator? "
s_slash:       .asciiz  " / "
s_equal:       .asciiz  " = "

#=================================================================================================================
# TEXT SECTION
#=================================================================================================================
.text
```

```
#-----------------------------------------------------------------------------------------------------------
# PROCEDURE: main()
#
# STACK
# We allocate 6 words: num1, den1, num2, den2, quot_num, quot_den.
#
# Note: we do not need to save $ra because main() does not return.
#
# +------------------+
# | local num1       | $sp + 20
# +------------------+
# | local den1       | $sp + 16
# +------------------+
# | local num2       | $sp + 12
# +------------------+
# | local den2       | $sp + 8
# +------------------+
# | local quot_num   | $sp + 4
# +------------------+
# | local quot_den   | $sp
# +------------------+
#-----------------------------------------------------------------------------------------------------------
main:
# Create stack frame and allocate 6 words for locals num1, den1, num2, den2, quot_prod, quot_den.
    addi    $sp, $sp, -24              # Allocate 6 words in stack frame

# num1, den1 = read_fraction()
    jal     read_fraction              # Call read_fraction()
    sw      $v0, 20($sp)               # Save num1
    sw      $v1, 16($sp)               # Save den1

# num2, den2 = read_fraction()
    jal     read_fraction              # Call read_fraction()
    sw      $v0, 12($sp)               # Save num2
    sw      $v1, 8($sp)                # Save den2

# quot_num, quot_den = div_fraction(num1, den1, num2, den2)
    lw      $a0, 20($sp)               # $a0 = num1
    lw      $a1, 16($sp)               # $a1 = den1
    lw      $a2, 12($sp)               # $a2 = num2
    lw      $a3, 8($sp)                # $a3 = den2
    jal     div_fraction               # Call div_fraction(num1, den1, num2, den2)
    sw      $v0, 4($sp)                # Save quot_num
    sw      $v1, 0($sp)                # Save quot_den

# print_fraction(num1, den1)
    lw      $a0, 20($sp)               # $a0 = num1
    lw      $a1, 16($sp)               # $a1 = den1
    jal     print_fraction             # Call print_fraction(num1, den1)

# SysPrintStr(" / ")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_slash               # $a0 = addr-of " / "
    syscall                            # SysPrintStr(" / ")

# print_fraction(num2, den2)
    lw      $a0, 12($sp)               # $a0 = num2
    lw      $a1, 8($sp)                # $a1 = den2
    jal     print_fraction             # Call print_fraction(num2, den2)
```

```
# SysPrintStr(" = ");
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_equal               # $a0 = addr-of " = "
    syscall                            # SysPrintStr(" = ")

# print_fraction(quot_num, quot_den)
    lw      $a0, 4($sp)                # $a0 = quot_num
    lw      $a1, 0($sp)                # $a1 = quot_den
    jal     print_fraction             # Call print_fraction(quot_num, quot_den)

# SysPrintChar('\n')
    addi    $v0, $zero, SYS_PRINT_CHAR # $v0 = SysPrintChar service code
    addi    $a0, $zero, 10             # $a0 = ASCII value of linefeed character '\n'
    syscall                            # SysPrintChar('\n')

# SysExit()
    add     $sp, $sp, 24               # Deallocate 6 words
    addi    $v0, $zero, SYS_EXIT       # $v0 = SysExit service code
    syscall                            # Call SysExit()

#-----------------------------------------------------------------------------------------------------------
# PROCEDURE: div_fraction()
#
# PARAMETERS
# $a0 - num1
# $a1 - den1
# $a2 - num2
# $a3 - den2
#
# STACK
# We allocate 7 words: $ra, $a0-$a3, inv_num, inv_den.
#
# Note we have to save $a0-$a3 (containing the input parameters) because when we call invert_fraction()
# we do not know if that function will alter those registers. It is the responsiblity of the caller to
# save $t registers, $a registers, and $v registers.
#
# +------------------+
# |   saved $ra      | $sp + 24
# +------------------+
# | saved $a0 (num1) | $sp + 20
# +------------------+
# | saved $a1 (den1) | $sp + 16
# +------------------+
# | saved $a2 (num2) | $sp + 12
# +------------------+
# | saved $a3 (den2) | $sp + 8
# +------------------+
# | local inv_num    | $sp + 4
# +------------------+
# | local inv_den    | $sp
# +------------------+
#
# RETURNS
# $v0 - quot_num
# $v1 - quot_den
#-----------------------------------------------------------------------------------------------------------
```

```
div_fraction:
# Create stack frame and allocate 7 words. Save $ra and $a0-$a3.
    addi    $sp, $sp, -28              # Alloc 7 words
    sw      $ra, 24($sp)               # Save $ra
    sw      $a0, 20($sp)               # Save 1st arg in param num1
    sw      $a1, 16($sp)               # Save 2nd arg in param den1
    sw      $a2, 12($sp)               # Save 3rd arg in param num2
    sw      $a3, 8($sp)                # Save 4th arg in param den2

# inv_num, inv_den = invert_fraction(num2, den2)
    lw      $a0, 12($sp)               # $a0 = num2
    lw      $a1, 8($sp)                # $a1 = den2
    jal     invert_fraction            # Call invert_fraction(num2, den2)
    sw      $v0, 4($sp)                # Save returned numerator in inv_num
    sw      $v1, 0($sp)                # Save returned denominator in inv_den

# quot_num, quot_den = mult_fraction(num1, den1, inv_num, inv_den)
    lw      $a0, 20($sp)               # $a0 =  num1
    lw      $a1, 16($sp)               # $a1 =  den1
    lw      $a2, 4($sp)                # $a2 =  inv_num
    lw      $a3, 0($sp)                # $a3 =  inv_den
    jal     mult_fraction              # Call mult_fraction(num1, den1, inv_num, inv_den)

# Note that mult_fraction returns quot_num in $v0 and quot_den in $v1. This procedure simply returns
# those values in $v0 and $v1 as well.

# Return quot_num, quot_den
    lw      $ra, 24($sp)               # Restore $ra
    add     $sp, $sp, 28               # Deallocate 7 words
    jr      $ra                        # Return quot_num in $v0, quot_den in $v1

#-----------------------------------------------------------------------------------------------------------
# PROCEDURE: invert_fraction()
#
# PARAMETERS
# $a0 - num
# $a1 - den
#
# STACK
# We allocate 2 words: inv_num, and inv_den.
#
# +------------------+
# | local inv_num    | $sp + 4
# +------------------+
# | local inv_den    | $sp
# +------------------+
#
# RETURNS
# $v0 - inv_num
# $v1 - inv_den
#-----------------------------------------------------------------------------------------------------------
invert_fraction:
# Create stack frame and allocate 2 words.
    addi    $sp, $sp, -8               # Allocate 2 words in stack frame

# inv_num = den
    sw      $a1, 4($sp)                # inv_num = den

# inv_den = num
    sw      $a0, 0($sp)                # inv_den = num
```

```
# If (inv_den < 0) Then ...
    lw      $t0, 0($sp)               # $t0 = inv_den
    bge     $t0, $zero, end_if        # If inv_den >= 0 skip over true clause
    lw      $t1, 4($sp)               # $t1 = inv_num
    neg     $t1, $t1                  # $t1 = -inv_num
    sw      $t1, 4($sp)               # inv_num = -inv_num
    neg     $t0, $t0                  # $t0 = -inv_den
    sw      $t0, 0($sp)               # inv_den = -inv_den
end_if:

# Return inv_num, inv_den
    lw      $v0, 4($sp)               # $v0 = inv_num
    lw      $v1, 0($sp)               # $v1 = inv_den
    add     $sp, $sp, 8               # Deallocate 2 words
    jr      $ra                       # Return inv_num in $v0 and inv_den in $v1

#-----------------------------------------------------------------------------------------------------------
# PROCEDURE: mult_fraction()
#
# PARAMETERS
# $a0 - num1
# $a1 - den1
# $a2 - num2
# $a3 - den2
#
# STACK
# We allocate 2 words: prod_num, prod_den.
#
# +------------------+
# | local prod_num   | $sp + 4
# +------------------+
# | local prod_den   | $sp
# +------------------+
#
# RETURNS
# $v0 - prod_num
# $v1 - prod_den
#-----------------------------------------------------------------------------------------------------------
mult_fraction:
# Create stack frame and allocate 2 words.
    addi    $sp, $sp, -8              # Allocate 2 words in stack frame

# prod_num = num1 * num2
    mul     $t0, $a0, $a2             # $t0 = num1 * num2
    sw      $t0, 4($sp)               # prod_num = num1 * num2

# prod_den = den1 * den2
    mul     $t0, $a1, $a3             # $t0 = den1 * den2
    sw      $t0, 0($sp)               # prod_den = den1 * den2

# Return prod_num, prod_den
    lw      $v0, 4($sp)               # $v0 = prod_num
    lw      $v1, 0($sp)               # $v1 = prod_den
    add     $sp, $sp, 8               # Deallocate 2 words
    jr      $ra                       # Return prod_num in $v0 and prod_den in $v1
```

```
#-------------------------------------------------------------------------------------------------------------
# PROCEDURE: print_fraction()
#
# PARAMETERS
# $a0 - num1
# $a1 - den1
#
# STACK
# print_fraction() is a leaf procedure and does not allocate any local variables so there is no need to
# create a stack frame.
#
# RETURNS
# Nothing
#-------------------------------------------------------------------------------------------------------------
print_fraction:
# SysPrintInt(num)
    addi    $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    syscall                            # SysPrintInt(num)

# SysPrintChar('/')
    addi    $v0, $zero, SYS_PRINT_CHAR # $v0 = SysPrintChar service code
    addi    $a0, $zero, 47             # $a0 = ASCII value of '/'
    syscall                            # SysPrintChar('/')

# SysPrintInt(den)
    addi    $v0, $zero, SYS_PRINT_INT  # $v0 = SysPrintInt service code
    move    $a0, $a1                   # $a0 = den
    syscall                            # SysPrintInt(den)

# Return
    jr      $ra                        # Return nothing

#-------------------------------------------------------------------------------------------------------------
# PROCEDURE: read_fraction()
#
# PARAMETERS
# None
#
# STACK
# We allocate 3 words: $ra, num, den.
#
# +------------------+
# |   saved $ra      | $sp + 8
# +------------------+
# |   local num      | $sp + 4
# +------------------+
# |   local den      | $sp
# +------------------+
#
# RETURNS
# $v0 - num
# $v1 - den
#-------------------------------------------------------------------------------------------------------------
read_fraction:
# Create stack frame and allocate 3 words.
    addi    $sp, $sp, -12              # Allocate 3 words in stack frame
    sw      $ra, 8($sp)                # Save $ra
```

```
# SysPrintStr("Enter numerator? ")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_num_prompt          # $a0 = addr-of "Enter numerator? "
    syscall                            # SysPrintStr("Enter numerator? ")

# num = SysReadInt()
    addi    $v0, $zero, SYS_READ_INT   # $v0 = SysReadInt service code
    syscall                            # $v0 = SysReadInt()
    sw      $v0, 4($sp)                # num = SysReadInt()

# SysPrintStr("Enter denomerator? ")
    addi    $v0, $zero, SYS_PRINT_STR  # $v0 = SysPrintStr service code
    la      $a0, s_den_prompt          # $a0 = addr-of "Enter denominator? "
    syscall                            # SysPrintStr("Enter denominator? ")

# den = SysReadInt()
    addi    $v0, $zero, SYS_READ_INT   # $v0 = SysReadInt service code
    syscall                            # $v0 = SysReadInt()
    sw      $v0, 0($sp)                # den = SysReadInt()

# Return num, den
    lw      $ra, 8($sp)                # Restore $ra
    lw      $v0, 4($sp)                # $v0 = num
    lw      $v1, 0($sp)                # $v1 = den
    add     $sp, $sp, 12               # Deallocate 3 words
    jr      $ra                        # Return num in $v0 and den in $v1
```

2.11 MIPS Addressing

2.11.1 Addressing in Jumps

Recall, the format of the MIPS `j` (jump) instruction is:

If memory addresses are 32-bits then where do the additional 6-bits of the jump target address come from? First, in MIPS, the jump target address must be word-aligned, i.e., at an address that is divisible by four. Remember, that an address that is divisible by four—when written in binary—will have the two least significant bits cleared to 00. Consequently, the first step in determining the jump target address is to stick those two 0-bits onto the right end of $addr_{25:0}$ forming $jump\text{-}target\text{-}address_{27:0}$; this is equivalent to shifting $addr_{25:0}$ left two times. Now, we have only have four missing bits and in MIPS, those four bits come from $(\text{PC}+4)_{31:28}$[16]

$$jump\text{-}target\text{-}address_{31:0} = (\text{PC}+4)_{31:28} \,||\, (addr_{25:0} << 2)$$

where we are using || to represent **bit concatenation**. Since the instruction only encodes 26-bits of the jump target address, the jump target address cannot be *any* memory address, but rather must be in the range $[(\text{PC}+4)_{31:28} \,||\, 0\text{x}000\_0000$ to $(\text{PC}+4)_{31:28} \,||\, 0\text{xFFF}\_\text{FFFC}]$. For example, suppose PC = 0x5000_41A0 and a `j` instruction encoded as 0x0B04_080C is encountered. What is the jump target address?

2.11.2 Branches and PC-Relative Addressing

Recall that a `beq` instruction is encoded as an I-format instruction:

Most processor architectures form the branch target address by using a form of addressing which is known as **PC-relative addressing** where a *branch offset* is added to PC to form the branch target address. If we form the 32-bit branch target address by adding PC and the 16-bit immediate field of the branch instruction (which forms a two's complement offset in the range [-32768, 32767]) then we could branch to any address which is in the range [PC - 32768, PC + 32767]. To extend this range, MIPS treats the offset as being words rather than bytes, i.e., $offset = imm_{15:0} << 2$. Furthermore, due to the design of the MIPS datapath (discussed in Ch. 4) the value of PC is actually PC + 4 when the branch target address is computed. Consequently,

16 At the time the jump target address is computed in the hardware during the execution of a J instruction, PC has already had 4 added to it.

Example: Consider this code. What would be the encoding of the beq, j, and bne instructions assuming that the address of the add instruction is 0x0040_4000?

```
loop:
   add   $t0, $t0, $t1          # 0x0040_4000
   sll   $t0, $t0, 2            # 0x0040_4004
   slt   $t1, $t0, $zero        # 0x0040_4008
   beq   $t1, $zero, false      # 0x0040_400C
   li    $t2, 13                # 0x0040_4010
   j     end_if                 # 0x0040_4014
false:
   li    $t2, -13               # 0x0040_4018
end_if:
   bne   $t0, $zero, loop       # 0x0040_401C
end_loop:
   nop                          # 0x0040_4020
```

When the `j end_if` instruction is fetched from memory to be executed, PC is 0x0040_4014 and PC+4 is 0x0040_4018. The jump target address is 0x0040_401C,

$$\textit{jump-target-address}_{31:0} = (\text{PC+4})_{31:28} \,||\, (\textit{addr}_{25:0} << 2)$$

Solving for $\textit{addr}_{25:0}$ we have,

$$\textit{addr}_{25:0} = (\textit{jump-target-address}_{31:0} - (\text{PC+4})_{31:28}000\_0000) >> 2$$

Consequently,

$\textit{addr}_{25:0} = (0x0040\_401C - 0x0000\_0000) >> 2$
$\textit{addr}_{25:0} = 0x0040\_401C >> 2$
$\textit{addr}_{25:0} = 0000\ 0000\ 0100\ 0000\ 0100\ 0000\ 0001\ 1100 >> 2$
$\textit{addr}_{25:0} = 00\ 0001\ 0000\ 0001\ 0000\ 0000\ 0111$ (discarding the four msb's after shifting right)

The j opcode is 000010, so the instruction encoding will be:

```
000010 00000100000001000000000111 = 0x08101007
```

When the beq instruction is fetched from memory to be executed, PC is 0x0040_400C and PC+4 is 0x0040_4010. The branch target address is 0x0040_4018,

$$\textit{branch-target-address}_{31:0} = (\text{PC} + 4) + (\textit{sign-ext}(\textit{imm}_{15:0}) << 2)$$

Solving for $\textit{imm}_{15:0}$ we have,

$$\textit{imm}_{15:0} = (\textit{branch-target-address}_{31:0} - (\text{PC} + 4)) >> 2$$

Consequently,

$\textit{imm}_{15:0} = (0x0040\_4018 - 0x0040\_4010) >> 2$
$\textit{imm}_{15:0} = 0x08 >> 2$
$\textit{imm}_{15:0} = 1000 >> 2$
$\textit{imm}_{15:0} = 0000\ 0000\ 0000\ 0010$

The encoding for `beq` will be:

```
000100 01001 00000 0000000000000010 = 0x11200002
```

When the `bne` instruction is fetched from memory to be executed, PC is 0x0040_401C and PC+4 is 0x0040_4020. The branch target address is 0x0040_4000. Consequently,

$imm_{15:0}$ = (0x0040_4000 - 0x0040_4020) >> 2
$imm_{15:0}$ = -0x20 >> 2 (note: -0x20 in 32-bit two's complement is 0xFFFFFFE0)
$imm_{15:0}$ = 1111 1111 1111 1111 1111 1111 1110 0000 >> 2
$imm_{15:0}$ = 1111 1111 1111 1000 (which is -8 in decimal)

The encoding for `bne` will be:

```
000101 01000 00000 1111111111111000 = 0x1500FFF8
```