4

Instruction Set Architecture and ALU Design

Computers can execute binary programs. These programs consist of data and instructions of a particular CPU (central processing unit). A binary program is usually generated by a compiler that compiles a program written in high-level programming languages such as C, or by an assembler that translates a program written in the assembly programming language of a particular CPU, such as x86 or MIPS (microprocessor without interlocked pipeline stages).

This chapter introduces the instruction set architecture (ISA), some MIPS instructions, an assembler and simulator of MIPS integer instructions, and the design of an arithmetic logic unit (ALU) which calculates the operation results of some MIPS integer instructions.

4.1 Instruction Set Architecture

ISA is an important issue in hardware/software codesign. An ISA tells compiler developers "what a CPU can do," and tells CPU designers "what a CPU should do." Compiler developers use the ISA to develop compilers, and CPU designers design a CPU to implement the ISA. That is, an ISA is an interface between software and hardware, as shown in Figure 4.1.

An ISA defines the formats of instructions, the operations of instructions, the types of operands, the memory and registers the instructions can access, the byte ordering, and the addressing modes. Some popular ISAs include the Intel's x86, SGI/MIPS's MIPS32/MIPS64, IBM's PowerPC, SUN Microsystems' SPARC, HP's HP-PA, and ARM's ARM.

4.1.1 Operand Types

The main task of instructions is calculations on operands of different types. Table 4.1 lists some common types of operands. The corresponding keywords in C are also given in the table.

A byte is always 8 bits but the length of a word depends on the bit-processing ability of the CPU: for example, it has 16 bits in the 8086 CPU. Here we use 32 bits as the length of a word.

If an operand on which an instruction of a 32-bit CPU calculates is not 32 bits in length, a byte or a half word for example, the operand will be extended to 32 bits. If the operand is an unsigned type, zeroes will be appended on the left side of the operand. We call this a zero extension. If the operand is a signed type,

Computer Principles and Design in Verilog HDL, First Edition. Yamin Li.

Companion Website: www.wiley.com/go/li/verilog

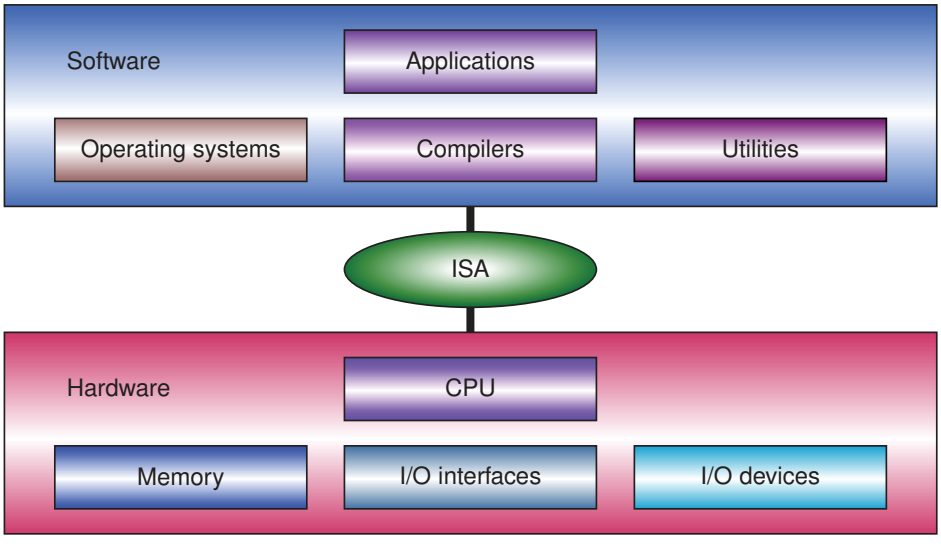

Figure 4.1 ISA as an interface between software and hardware

Table 4.1 Operand types

Operand type	Bits	Value range	Corresponding to C
Byte	8	−128 to +127	signed char
Unsigned byte	8	0 to 255	unsigned char
Half word	16	−32,768 to +32,767	short int
Unsigned half word	16	0 to 65,535	unsigned short int
Word	32	−2,147,483,648 to +2,147,483,647	int
Unsigned word	32	0 to 4,294,967,295	unsigned int
Single-precision FP	32		float
Double-precision FP	64		double

the sign extension will be performed by filling each extra bit in the left side with the most significant bit (the sign bit) of the operand.

4.1.2 Little Endian and Big Endian

Little endian and big endian define the order in which a sequence of bytes of a word is stored in memory. Little endian is an order in which the least significant byte (little end) is stored first (the lowest address). Big endian is an order in which the most significant byte (big end) is stored first.

Figure 4.2 shows an example. For a 4-byte word 0x76543210, the least significant byte 0x10 is stored in address xxxxxxxx0 for a little endian machine, while in a big endian machine the most significant byte 0x76 is stored in address xxxxxxx0.

We can write a program to detect which endian is being used in the machine. The following C code uses a pointer to detect the endian. A pointer is a variable that stores the address of another variable.

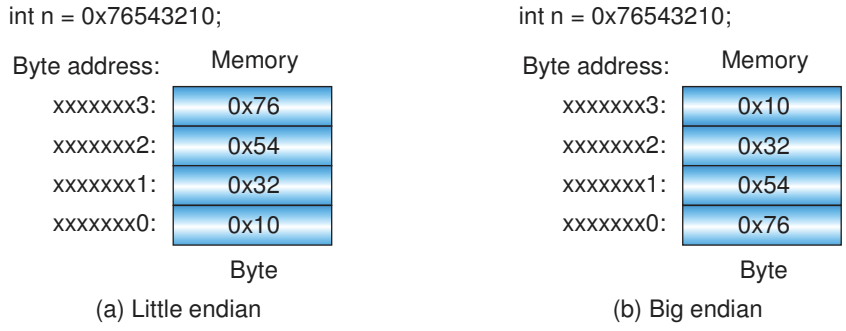

Figure 4.2 Little endian and big endian

```
main () {                                        // endian_pointer.c
    int n = 0x76543210;                          // a word
    if (*(unsigned char *)&n == 0x10)            // char starting address
         printf("little endian\n");              // stored 0x10
    else printf("big endian\n");                 // stored 0x76
}
```

Also we can use a union structure to do it: see the following C code. Just as a person has different names, all members of a union define different manifestations of the same variable.

```
main() {                                         // endian_union.c
    union {                                      // union:
        int intword;                             // to assess a word
        unsigned char characters[sizeof (int)]; // with different names
    } u;                                         // union u
    u.intword = 0x76543210;                      // access as an integer
    if (u.characters[sizeof (int) - 1] == 0x76) // access as array of byte
         printf("little endian\n");              // high byte in high address
    else printf("big endian\n");                 // high byte in low address
}
```

A byte is the smallest unit of memory access. It can be stored at any location of the memory. For other data types, there is a problem of data alignment. The data alignment refers to where the data is located in the memory. Referring to Figure 4.3, a memory address a is said to be n-byte aligned when n is a power of 2 and a is a multiple of n bytes.

Aligned data can be accessed faster. For example, the 4 bytes of an aligned word can be written to or read from memory with one access by using the word address (all the 4 bytes have a same word address). On the other hand, we must access to memory two times if the word is unaligned. Some ISAs allow the unaligned data placement. If it is not allowed and there is an unaligned access, an alignment fault exception will occur.

4.1.3 Instruction Types

An instruction type refers to what kind of operation is done by an instruction. The instruction types differ between ISAs. Some instruction types that commonly appear in the modern ISAs are described below.

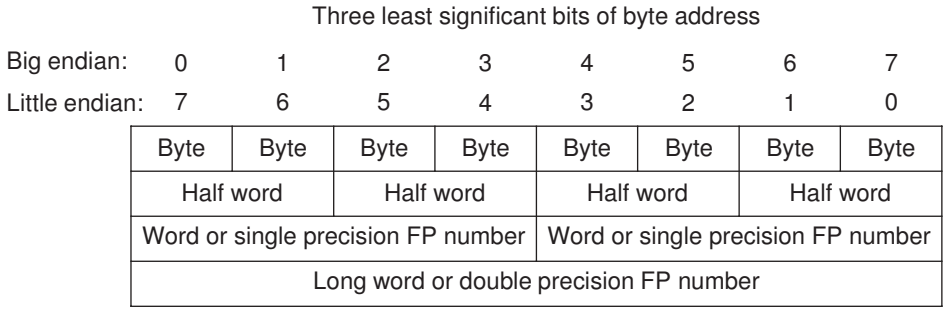

Figure 4.3 Data alignment in memory locations

4.1.3.1 Arithmetic Operation Type

Instructions of this type perform arithmetic operations on integers, such as addition, subtraction, multiplication, division, and square root.

4.1.3.2 Logic Operation Type

Logic operations include bitwise logical AND, OR, and NOT. Most ISAs provide NOR (NOT OR) or XOR (exclusive OR) instead of NOT.

4.1.3.3 Shift Operation Type

There are mainly three types of shift instructions: shift left, logical shift right, and arithmetic shift right. Other shift instructions include arithmetic shift left, rotate shift, and rotate shift with carry.

4.1.3.4 Memory Access Type

Memory access instructions transfer data between the memory and registers inside the CPU. A load instruction loads memory data to the register. A store instruction writes register data to the memory. All RISC (reduced instruction set computer) type ISAs have these two instructions but CISC (complex instruction set computer) type ISAs may combine them into computational instructions.

4.1.3.5 Input/Output Access Type

Input/output instructions transfer data between I/O and CPU registers. In most RISC type ISAs, some addresses of the virtual memory are assigned to the I/O space so that the load and store instructions can be used to access I/O. This is called a memory-mapped I/O. CISC type ISAs may have a dedicated I/O space, therefore the input and output instructions must be prepared for accessing the I/O.

4.1.3.6 Control Transfer Type

Control transfer instructions alter the order of the instruction execution. Conditional branch instructions confirm a condition to determine whether to branch to a target address. A jump instruction jumps to a target address unconditionally. A subroutine call instruction jumps to the entry of the subroutine and saves the return address to somewhere. And a return instruction returns from the subroutine.

4.1.3.7 Floating-Point Calculation Type

All the computational instructions described above operate on integers. Floating-point instructions perform arithmetic operations on floating-point numbers. There are also some instructions that convert data formats between integers and floating-point numbers.

4.1.3.8 System Control Type

System control instructions include system calls, return from exceptions, and instructions that read CPU state registers or write CPU control registers.

4.1.4 Instruction Architecture

The instruction architectures decide how the CPU will store data. Figure 4.4 shows four instruction architectures that are used in modern CPUs.

In a stack architecture, the operands are implicitly on top of the stack. Two source operands are popped from the top of the stack, and the result is pushed onto the stack. The stack has a feature of first-in last-out or last-in first-out; it cannot be accessed randomly. JVM (Java virtual machine) Bytecode uses this architecture.

In an accumulator architecture, one operand is implicitly in the accumulator and the other operand is in the memory or register. The operation result is stored in accumulator implicitly. Z80 and 6502 ISAs use this architecture.

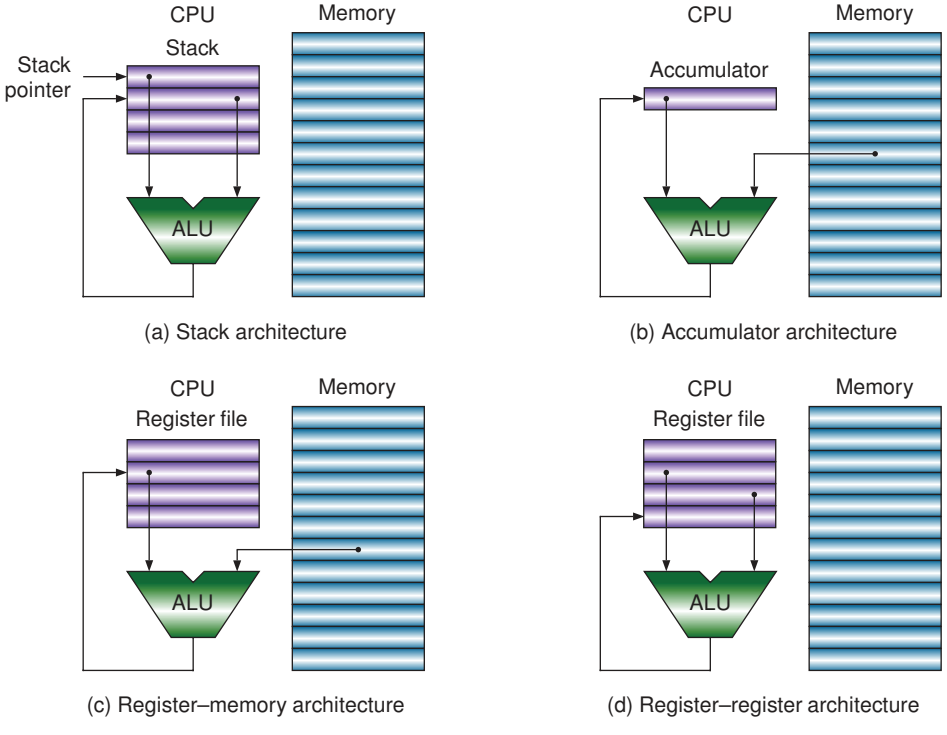

Figure 4.4 Instruction architecture

Table 4.2 Code examples of instruction architectures

Stack		Accumulator		General-purpose register						
				Register–memory			Register–register			
push	x	load	x	load	r1,	x	load	r1,	x	
push	y	**add**	**y**	**add**	**r1,**	**y**	load	r2,	y	
add		store	z	store	z,	r1	**add**	**r3,**	**r1,**	**r2**
pop	z						store	z,	r3	

In a register–memory architecture, one operand is explicitly in a general-purpose register file and the other operand is in the memory. The operation result is stored in the register. The Intel x86 ISA uses this architecture.

In a register–register architecture, all operands are explicitly in a general-purpose register file. The operation result is also stored in the register. Almost all RISC type ISAs use this architecture. The data transfer between the register file and memory is performed by load and store instructions.

Table 4.2 gives the instruction code examples that implement $z = x + y$ in the architectures described above, where x, y, and z are in memory, and r1, r2, and r3 are registers. The **add** instruction performs the addition.

4.1.5 Addressing Modes

The addressing modes define how the instructions get operands. An operand can be in a register, in the memory, or in the instruction (immediate). Some simple addressing modes are given below.

4.1.5.1 Register Operand Addressing

The operand is in a register of the register file. In the following example, the two source operands are in register 1 (`r1`) and register 2 (`r2`), respectively. The result of the addition is stored in register 3 (`r3`).

```
add  r3, r1, r2          ; r3 <-- r1 + r2
```

4.1.5.2 Immediate Addressing

The operand is in instruction. In the following example, `-1` (constant) is given within the instruction.

```
addi r1, r1, -1          ; r1 <-- r1 - 1
```

4.1.5.3 Direct Addressing

The operand is in the memory whose address is given in the instruction. In the following example, the second source operand is in the memory location of `0x1234`.

```
add r3, r1, [0x1234]     ; r3 <-- r1 + Memory[0x1234]
```

4.1.5.4 Register Indirect Addressing

The operand is in the memory whose address is given by a register. In the following example, the second source operand is in the memory, and the content in register `r2` is the memory address.

```
add r3, r1, (r2)         ; r3 <-- r1 + Memory[r2]
```

4.1.5.5 Offset Addressing

The operand is in the memory whose address is the sum of an offset and the content of a register. In the following example, the second source operand is in the memory and the memory address is the sum of `0x1234` and the content in register `r2`.

```
add r3, r1, 0x1234(r2)  ; r3 <-- r1 + Memory[0x1234+r2]
```

4.2 MIPS Instruction Format and Registers

This section gives an example of the MIPS ISAs. MIPS has two ISA versions—MIPS32 and MIPS64. This book focuses on MIPS32.

4.2.1 MIPS Instruction Format

All the MIPS instructions are 32 bits in length. Figure 4.5 gives three formats of the MIPS instructions. `op` is the opcode (operation code) of the instruction. `func` stands for function.

The `op` of the R (register) format instructions is 0, and the operation of an instruction is specified by `func`. Each of `rs`, `rt`, and `rd` is a 5-bit register number. Shift instructions use `sa` to specify the shift amount (a constant). The content in register `rt` will be shifted, and the result of the shift will be written to register `rd`. Other R format instructions read two source operands from registers `rs` and `rt`, respectively, operate on the two operands, and write the result to the register `rd`.

The computational instructions of the I (immediate) format use immediate as the second source operand. The 16-bit `immediate` must be sign- or zero-extended into 32 bits. The first source operand is in register `rs`, and the result is written to register `rt`.

The conditional branch instructions of the I format compare the two operands located in registers `rs` and `rt`, respectively, and determine whether to branch to the target address. `immediate` is a signed word-offset and is used to calculate the branch target address.

The load instructions of the I format load memory data to the register `rt`. The store instructions store the register `rt` data to the memory. For both load and store instructions, the memory address is the sum of the sign-extended `immediate` and the operand in the register `rs`.

The `address` in the J (jump) format instructions will be shifted to the left by 2 bits to form the low 28 bits of the jump target address.

4.2.2 MIPS General-Purpose Registers

A register number of MIPS instructions has 5 bits. Therefore, there are 32 registers in the integer register file. Table 4.3 lists the names and usages of these registers.

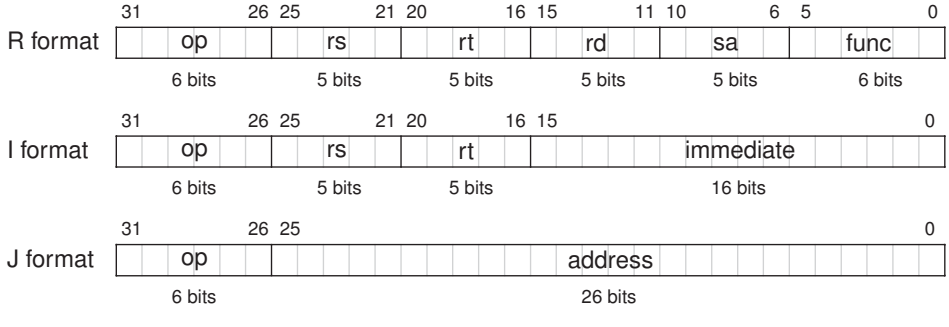

Figure 4.5 MIPS instruction formats

Table 4.3 MIPS general-purpose registers

Register name	Register number	Use
$zero	0	Constant 0
$at	1	Assembler temporary
$v0 to $v1	2 to 3	Function return value
$a0 to $a3	4 to 7	Function parameters
$t0 to $t7	8 to 15	Temporaries
$s0 to $s7	16 to 23	Saved temporaries
$t8 to $t9	24 to 25	Temporaries
$k0 to $k1	26 to 27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Note that the register 0 is not a register; its content is always a constant 0. Although Table 4.3 gives the convention on the usage of the registers, there are no differences among the remaining 31 registers from the hardware point of view. This usage gives the rules for preparing MIPS assembly language programs. `gcc`, a GNU C compiler, for MIPS machines follows these rules very well.

4.3 MIPS Instructions and AsmSim Tool

This section describes some MIPS instructions in detail and introduces a MIPS assembler and simulator, called AsmSim.

4.3.1 Some Typical MIPS Instructions

Table 4.4 lists some MIPS instructions that will be implemented in our CPU. We selected 20 typical and essential MIPS integer instructions that cover the three instruction formats. Each of these instructions is explained below.

```
add/sub/and/or/xor rd, rs, rt   # rd <-- rs op rt
```

These five instructions have the same format and perform addition, subtraction, AND, OR, and XOR, respectively. `rs` and `rt` are the two source register numbers, and `rd` is the destination register number.

```
sll/srl/sra rd, rt, sa          # rd <-- rt shift sa
```

These are three shift instructions: shift left logical (`sll`), shift right logical (`srl`), and shift right arithmetic (`sra`). The shift amount is given by the 5-bit `sa`.

```
addi rt, rs, immediate          # rt <-- rs + (sign)immediate
```

The `addi` (add immediate) instruction adds a 16-bit signed `immediate` to the 32-bit value in register `rs`. The result of the addition is saved to register `rt`. The 16-bit `immediate` is sign-extended into 32 bits.

```
andi/ori/xori rt, rs, immediate # rt <-- rs op (zero)immediate
```

Table 4.4 Twenty MIPS integer instructions

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
add	000000	rs	rt	rd	00000	100000	Register add
sub	000000	rs	rt	rd	00000	100010	Register subtract
and	000000	rs	rt	rd	00000	100100	Register AND
or	000000	rs	rt	rd	00000	100101	Register OR
xor	000000	rs	rt	rd	00000	100110	Register XOR
sll	000000	00000	rt	rd	sa	000000	Shift left
srl	000000	00000	rt	rd	sa	000010	Logical shift right
sra	000000	00000	rt	rd	sa	000011	Arithmetic shift right
jr	000000	rs	00000	00000	00000	001000	Register jump
addi	001000	rs	rt		Immediate		Immediate add
andi	001100	rs	rt		Immediate		Immediate AND
ori	001101	rs	rt		Immediate		Immediate OR
xori	001110	rs	rt		Immediate		Immediate XOR
lw	100011	rs	rt		offset		Load memory word
sw	101011	rs	rt		offset		Store memory word
beq	000100	rs	rt		offset		Branch on equal
bne	000101	rs	rt		offset		Branch on not equal
lui	001111	00000	rt		immediate		Load upper immediate
j	000010			address			Jump
jal	000011			address			Call

The `andi` (and immediate), `ori` (or immediate), and `xori` (exclusive or immediate) instructions perform bitwise logical AND, OR, and XOR, respectively, on the value in register `rs` and a 16-bit unsigned `immediate`. The result is written to register `rt`. The 16-bit `immediate` is zero-extended into 32 bits.

```
lui rt, immediate                # rt <-- immediate << 16
```

The `lui` (load upper immediate) instruction shifts `immediate` to the left by 16 bit. By using `lui` and `ori` instructions, we can set a register to an any 32-bit constant (`lui` sets high 16 bits and `ori` sets low 16 bits).

```
lw rt, offset(rs)                # rt <-- memory[rs + offset]
```

The `lw` (load word) instruction loads a 32-bit word from memory. The memory address is calculated by adding a 16-bit signed immediate (`offset`) to the 32-bit value in register `rs`. The loaded word is saved to register `rt`.

```
sw rt, offset(rs)                # memory[rs + offset] <-- rt
```

The `sw` (store word) instruction stores a 32-bit word to memory. The memory address is calculated by adding a 16-bit signed `offset` to the 32-bit value in register `rs`. The word to be stored is in register `rt`.

```
beq rs, rt, label                # if (rs == rt) PC <-- label
```

The `beq` (branch on equal) instruction transfers control to a PC-relative target address (`label`) if the values in registers `rs` and `rt` are equal. The branch target address is calculated by adding an 18-bit

signed constant (the 16-bit `offset` shifted to the left by 2 bits) to the address of the instruction following `beq`.

```
bne rs, rt, label                # if (rs != rt) PC <-- label
```

The `bne` (branch on not equal) instruction transfers control to a PC-relative target address (`label`) if the values in registers `rs` and `rt` are not equal. The branch target address is calculated by adding an 18-bit signed constant (the 16-bit `offset` shifted to the left by 2 bits) to the address of the instruction following `bne`.

```
j target                         # PC <-- target
```

The `j` (jump) instruction transfers control to a target address whose low 28 bits are the 26-bit `target` shifted to the left by 2 bits; the remaining upper bits are the corresponding bits of the address of the instruction following the `j` instruction.

```
jal target                       # $31 <-- PC + 8; PC <-- target
```

The `jal` (jump and link) is a subroutine call instruction that does the same job as the `j` instruction and meanwhile saves the return address to register `$31`. The return address is the location at which execution continues after returning from the subroutine. MIPS uses the delayed branch technique; the return address is `PC + 8`.

```
jr rs                            # PC <-- rs
```

The `jr` (jump register) instruction transfers control to a target address given in register `rs`. The return from a subroutine can be done by letting `rs` to be 31.[1]

In MIPS ISA, there is no `subi` (subtract immediate) instruction. Such instruction can be implemented by `addi` because the immediate is represented with a signed number: $x - i = x + (-i)$.

MIPS assembler supports several pseudo-instructions. A pseudo-instruction will be translated into an actual instruction or a sequence of instructions by the assembler. For example, a `nop` (no operation) pseudo-instruction is translated into `sll $0, $0, 0`, which shifts the content of register `$0` to the left by 0 bit and writes the result to register `$0`. The 32-bit encoding of this instruction is 0.

Another pseudo-instruction is `li` (load immediate), which can load a 32-bit immediate to a register. For example, `li $4, 0x1234abcd` loads the 32-bit immediate `0x1234abcd` to register `$4`. Because an MIPS instruction is 32 bits in length and its opcode field takes 6 bits, loading a 32-bit immediate into a register cannot be done by only one instruction. The assembler translates `li $4, 0x1234abcd` into a sequence of two instructions: `lui $4, 0x1234` and `ori $4, $4, 0xabcd`. The first instruction writes `0x12340000` to register `$4`, and the second instruction writes the result of `0x12340000` OR `0x0000abcd`, or `0x1234abcd`, to register `$4`.

Table 4.5 lists some MIPS pseudo-instructions and the corresponding actual instructions. A pseudo-instruction can be translated in several ways. For example, the `move $4, $2` (the content of

[1] The behavior of the subroutine call and return is likely similar to that a person on a business trip. When the author teaches students in Japan about the MIPS `jal` and `jr` instructions, the following example is used: when the person goes to the destination city on a business trip, he or she takes a fight of JAL (Japan Airline) because he or she must go there hurriedly. When returning to his or her home city, he or she can take a train of JR (Japan Railway) without haste. One more thing is that he or she must know where the home is. This information is in the person's brain memory. MIPS CPU remembers it by recording the return address into the $31 register.

Table 4.5 Some MIPS pseudo-instruction examples

Pseudo-instruction	Actual instruction(s)	Meaning
`nop`	`sll $0, $0, 0`	No operation
`clear $4`	`sll $4, $0, 0`	Clear
`move $4, $5`	`sll $4, $5, 0`	Copy content of `$5` to `$4`
`not $4, $5`	`nor $4, $5, $0`	Write inverse of `$5` to `$4`
`subi $4, $5, 1`	`addi $4, $5, -1`	Subtract a 16-bit immediate
`li $4, 0x1234abcd`	`lui $4, 0x1234` `ori $4, $4, 0xabcd`	Load a 32-bit immediate
`la $4, label`	`lui $4, %hi(label)` `ori $4, $4, %lo(label)`	Load a label address
`b label`	`beq $0, $0, label`	Unconditional branch
`bz $4, label`	`beq $4, $0, label`	Branch on zero
`bnz $4, label`	`bne $4, $0, label`	Branch on not zero

Table 4.6 Code sequences for unimplemented instructions

Unimplemented instruction	Code sequence	Meaning
`bltz $4, label`	`srl $at, $4, 31` `bne $at, $0, label`	Branch on negative
`bgez $4, label`	`srl $at, $4, 31` `beq $at, $0, label`	Branch on not negative

`$2` is copied to `$4`) pseudo-instruction can be translated into `sll/srl/sra $4, $2, 0`; it can be also translated into `add/sub/or/xor $4, $2, $0`, or `addi/ori/xori $4, $2, 0`.

The `not` pseudo-instruction can be translated into `nor` instruction, but our CPU will not implement the `nor`. To perform `not $4, $5`, we can use a sequence of two instructions: `addi $4, $0, -1` and `xor $4, $5, $4`. The first instruction loads `0xffffffff` to register `$4`; the second instruction performs an XOR operation on the content of `$5` and `0xffffffff` and writes the result to register `$4`. By using only the 20 instructions listed in Table 4.4, we can implement other instructions. Table 4.6 lists two examples. The register `$at` (`$1`) is used in the sequences.

4.3.2 Supporting Subroutine Call and Pointer

Subroutine call and pointer are two important concepts in high-level programming languages. We use examples to show how MIPS implements these concepts.

4.3.2.1 Subroutine Call

The MIPS CPU uses `jal` and `jr` instructions to implement a subroutine call and return, respectively. The `jal` instruction saves the return address (PC + 8 in pipelined CPU; here we assume PC + 4) into register `$ra` (`$31`) and jumps to the entry of the subroutine. `jr $ra` writes the content of `$ra` (return address) into the PC. Let's see the following C code.

```
int sum(int *array, int n) {                        // subroutine_call.c
    int i;
    int total = 0;
    for (i = 0; i < n; i++) {
        total += array[i];                          // sum of n array elements
    }
    return(total);                                  // return sum
}
int main() {
    int a[] = {1, 2, 3};                            // initialize the array
    printf ("The sum is %d\n", sum(a, 3));          // call subroutine
    return(0);
}
```

In the `main` program, the integer array `a` holds three elements: `a[0]` = 1, `a[1]` = 2, and `a[2]` = 3. The `main` program calls the function `sum` (a subroutine is named as a function in C and we use the both names in the text). It passes two parameters to `sum`: the start address of `a` and the number of elements. The function `sum` calculates the sum of the elements with a `for` loop and returns the result `total` to the `main` program. Actually, the `main` program can be also considered as a subroutine. The operating system (OS) calls it, and after the execution is finished, the control will be returned to the OS.

The C code can be compiled to the following assembly program. Because register `$ra` contains the return address to the OS initially, before calling function `sum` the content of `$ra` must be saved somewhere; otherwise, it will be overwritten by the `jal sum` instruction. MIPS saves the content of `$ra` to a memory stack. Stack uses the main memory, and it has a stack pointer (register `$sp`) which points to the top of the stack. The local variables, the array `a` for instance, are also saved in the stack. If some registers are required to have the same values before and after the function call, their contents must be also saved in the stack.

```
 1: .data                          # data segment
 2: $LC0:                          # address of array a
 3:     .word  1, 2, 3             # elements of array a
 4: $LC1:                          # address of string
 5:     .ascii "The sum is %d\n"   # string
 6: .text                          # code segment
 7: sum:                           # entry of subroutine sum
 8:     subi   $sp, $sp, 8         # reserve stack space
 9:     move   $3,  $0             # i = 0
10:     move   $6,  $0             # total = 0
11:     blez   $5,  $L3            # goto $L3 if n <= 0
12: $L5:                          # for loop
13:     sll    $2,  $3,  2         # i * 4 (4 bytes per word)
14:     add    $2,  $2,  $4        # base address + i * 4
15:     lw     $2,  0($2)          # load a[i]
16:     addi   $3,  $3,  1         # i++
17:     add    $6,  $6,  $2        # total += a[i]
18:     bne    $3,  $5,  $L5       # goto $L5 if i != n
19: $L3:                          # end of loop
20:     move   $2,  $6             # move total to $2
```

```
21:     addi   $sp, $sp, 8       # release stack space
22:     jr     $ra               # return from subroutine
23: main:                        # program entry
24:     subi   $sp, $sp, 40      # reserve stack space
25:     sw     $ra, 32($sp)      # save return address
26:     la     $5,  $LC0         # address of array a
27:     lw     $2,  0($5)        # load a[0]
28:     lw     $3,  4($5)        # load a[1]
29:     lw     $4,  8($5)        # load a[2]
30:     sw     $2,  16($sp)      # store a[0] to stack
31:     sw     $3,  20($sp)      # store a[1] to stack
32:     sw     $4,  24($sp)      # store a[2] to stack
33:     addi   $4,  $sp, 16      # $4: address of a[0]
34:     li     $5,  3            # $5: n = 3
35:     jal    sum               # call subroutine sum
36:     la     $4,  $LC1         # $4: address of string
37:     move   $5,  $2           # $5: total
38:     jal    printf            # call printf
39:     move   $2,  $0           # return value 0
40:     lw     $ra, 32($sp)      # restore return address
41:     addi   $sp, $sp, 40      # release stack space
42:     jr     $ra               # return to operating system
43: .end                         # end of program
```

Referring to Figure 4.6(a), in MIPS the stack grows downward and the register `$sp` (`$29`) points to the top location of the stack. In line 24 of the assembly program (the entry of the `main` program), a 40-byte (or 10-word) stack space is reserved. The instruction in line 25 saves the return address in register `$ra` to the stack. The instructions in lines 26–32 push the local variables (three array elements) onto the stack. Before calling the subroutine, the parameters for the subroutine are putted to registers `$4` and `$5`. Register `$4` holds the starting address of the array `a` (line 33) and register `$5` holds a 3, which is the number of array elements (line 34). Then the subroutine `sum` is called (line 35). The result calculated by the subroutine is in register `$2`; it is copied to `$5` (line 37). The string address of a string that will be printed out is written to register `$4` (line 36). Then the program calls a system function `printf` to display the result (line 38). The user's work is done. Before returning to the OS (line 42), the return value in `$2` is cleared (line 39) for `return(0)` in the C code, the original return address is restored to register `$ra` from the stack (line 40), and the stack space is released (line 41).

Referring to Figure 4.6(b), in the subroutine of the assembly program an 8-byte stack space is reserved (line 8). Because there is no further subroutine call, the return address need not be saved. The local variables within the subroutine should use the stack, but there are only two local variables: `i` and `total`, that directly use registers `$3` (line 9) and register `$6` (line 10), respectively. Therefore, the reserved stack was not used at all. Both variables are cleared to 0 initially. The instructions in lines 11–18 do the `for` loop. Then the result is moved to the register `$2` (line 20). After releasing the stack (line 21), the subroutine is completed by returning to the `main` program (line 22).

4.3.2.2 Pointer

Now, we show how to use MIPS instructions to implement the pointer operations in C programs. Simply we can define a variable of a certain type (`int`, `float`, or `double`) with a name. The value of a variable

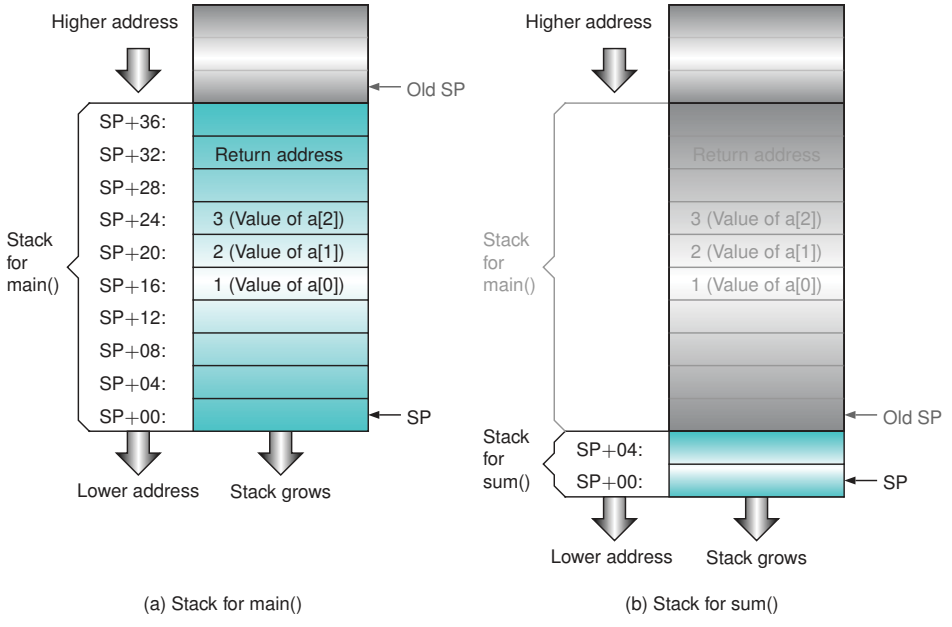

Figure 4.6 Stack for function call

can vary. The value is stored in the memory. A pointer can be considered as the memory address; in that location, the value of a variable is stored.

In the following C code example, we first define a variable `num` whose type is `int` (integer) and a variable `ptr` whose type is a pointer. `ptr` points to a location where an integer (`*ptr`) is stored. Then, we let the value of the variable `num` to be a 1, and let the value of the variable `ptr` to be the address of `num` (`&num`). Finally, we display the value of the `ptr` with a pointer format (`%p`).

```
main() {                          // pointer.c
    int  num;                     // an integer variable num
    int *ptr;                     // a pointer variable ptr pointing to an int
    num =  1;                     // let the value of num be a 1
    ptr = &num;                   // let the value of ptr be the address of num
    printf ("The address of num is %p\n", ptr);             // print ptr out
}
```

Before running this program, we knew that the value of the variable `num` is a 1, but we didn't know the value of the variable `ptr`. That is, we didn't know where the value of `num` was stored. The following assembly codes implement the C codes above.

```
 1: .data                                          # data segment
 2: $LC0:                                          # address of string
 3:     .ascii "The address of num is %p\n"        # string
 4: .text                                          # code segment
 5: main:                                          # program entry
```

```
 6:      subi   $sp, $sp, 32                    # reserve stack space
 7:      sw     $ra, 24($sp)                    # save return address
 8:      li     $2,  0x00000001                 # num = 1
 9:      sw     $2,  16($sp)                    # store num into stack
10:      la     $4,  $LC0                       # $4: address of string
11:      addi   $5,  $sp, 16                    # $5: ptr = &num
12:      jal    printf                          # call printf
13:      lw     $ra, 24($sp)                    # restore return address
14:      addi   $sp, $sp, 32                    # release stack space
15:      jr     $ra                             # return to operating system
16: .end                                       # end of program
```

The ASCII string in line 3 will be printed out. `%p` is the pointer format directive used for displaying the value of `ptr`. The label address of the string is `$LC0` (line 2). In line 6, a 32-byte (or 8-word) stack space is reserved. The instruction in line 7 saves the return address to the stack. The value 1 is loaded into register `$2` (line 8). It is then stored in a location of the stack memory; the memory location is the content of register `$sp` plus 16 (line 9), which is the value of `ptr`, as shown in Figure 4.7.

For printing the value of `ptr` out (line 12), the address of the string is putted into register `$4` (line 10), and the value of the pointer `ptr` is putted into register `$5` (line 11), which is the location where the value of `num` is stored. Before finishing the program, the original return address is restored to register `$ra` from the stack (line 13), and the stack space is released (line 14).

4.3.3 AsmSim—A MIPS Assembler and Simulator

A better way to understand the operation of each instruction is to see its execution results. This section introduces a Web-based graphical tool, named AsmSim, that contains a MIPS assembler and an execution simulator for the MIPS integer instructions.

The main features of the tool include the supports of the common function calls that are usually used in C programs, the capability of showing images on a virtual video graphics array (VGA) display, and the

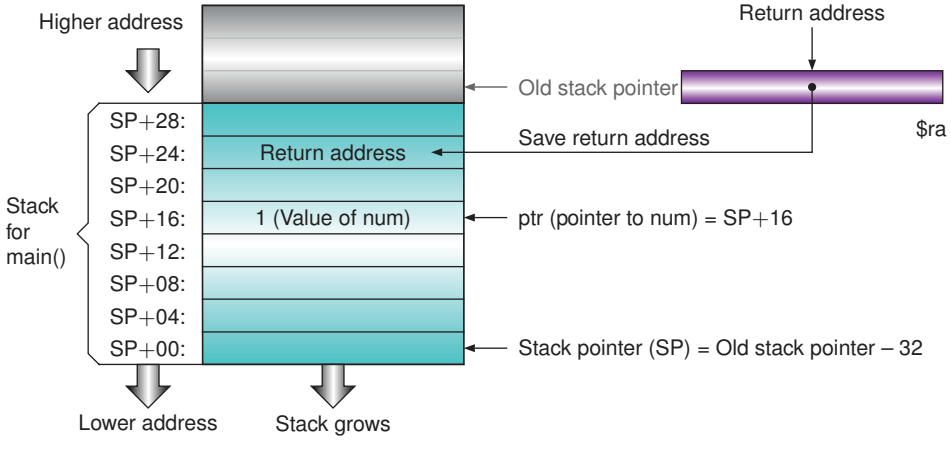

Figure 4.7 Pointer of a variable

auto-generation of Xilinx COE files and Altera MIF files. In detail, the AsmSim provides the following functions:

1. The basic MIPS assembly programs' assembling and simulation;
2. The basic input and output system function calls, such as scanf() and printf();
3. Interrupt mechanism, like addKeyListener() and removeKeyListener() in Java, for handling the keyboard interrupt request;
4. Manipulating a standard VGA (640×480 pixels) window. The direct video random access memory (VRAM) access function allows the user program to read/write a pixel (24-bit RGB pattern) from/to the VRAM. It is also possible for the user program to define the starting address of the VRAM;
5. Graphics object draw and fill. The currently supported objects include line, oval, rectangle, and string. The generated image is stored in an off-screen buffer, and the `paint` function call will display the image on the graphics display by writing the image to the VRAM;
6. Automatically generating Xilinx COE and Altera MIF memory initialization files for the purpose of using Xilinx and Altera FPGA boards;
7. Some other useful function calls, like "get a random number," "get the timer," "sleep," and "get calendar." AsmSim also supports the data structure of the linked list for implementing graph algorithms.

The current version of the AsmSim implements almost all the MIPS integer instructions. It accepts the assembly program. The three windows, namely the main window, the program edit window, and the graphics console window, are launched.

The main window (Figure 4.8) shows the instructions, the values of the registers, the data of the data memory, and the control buttons. The instructions are displayed with memory addresses, instruction encodings, and the user source codes. The values of the registers are displayed in hexadecimal format. In addition to the register names, the register numbers are also shown in the window. There are 11 buttons in the main window, whose functions are described below (the last three buttons are not shown in the figure because of space limitation).

1. **[edit]**: opens the program edit window (in case it was closed);
2. **[step]**: executes one instruction that is highlighted currently;
3. **[goto]**: executes instructions until a break-point is reached.
4. **[ascii]**: displays the ASCII of the selected contents in the data memory;
5. **[restart]**: reloads the user program;
6. **[run]**: executes user program;
7. **[stop]**: stops the execution;
8. **[quit]**: quits from debug mode and enters command line mode;
9. **[inst]**: displays the encodings and formats of the MIPS instructions;
10. **[xilinx]**: generates Xilinx COE file;
11. **[altera]**: generates Altera MIF file.

Referring to Figure 4.8, the highlighted instruction is the one that will be executed next. The figure shows the image after the first two instructions were executed. The `subi $sp, $sp, 24` instruction reserves stack space of 24 bytes for the `main` function. The content of the `$sp` register is highlighted. The `sw $ra, 20($sp)` instruction saves the return address (the content of `$31` register) into the stack. After the execution of this instruction, the memory location of 0x0000ff3c (0x0000ff28 + 0x14) contains 0x03ffff00. This content is also highlighted but is not shown in the figure.

The program edit window (Figure 4.9) is used to edit the user program. We followed the rule of the `gcc` for the MIPS register usages. The assembler supports two kinds formats of the comments: `/**/` and `#`. We can use `control_s` for searching text, just like `emacs`. The figure shows the case of searching "`print`." The MIPS assembly program shown in the figure prints out "`Hello, World!`" on the console window. The address of the hello message is loaded into the `$4` register. Then, the `jal printf` instruction invokes a system call to display the message.

The keyword `.text` indicates the start of the code segment. We can also use `.code`. `.data` indicates the start of data segment. Within the data segment, we can use `.ascii` or `.asciiz` to declare the strings.

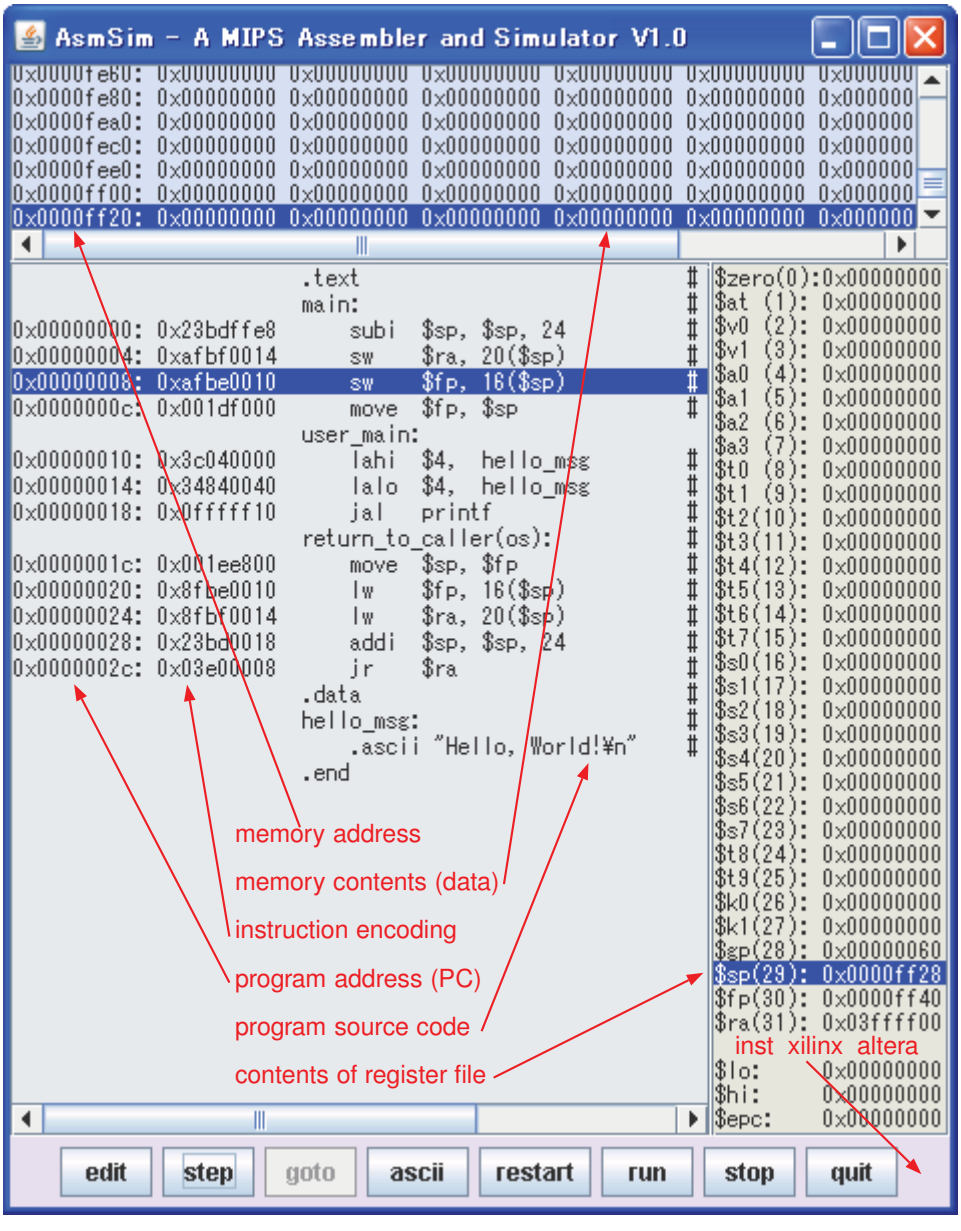

Figure 4.8 AsmSim main window

Other directives include `.word` and `.comm`. `.word` supports the declaration of multiple words by separating each word with a comma. `.comm` reserves memory space. It has the format `.comm array_name, bytes`, where `array_name` is a variable name of an array and `bytes` defines the memory space length in bytes. Asmsim also supports `.comm array_name, bytes, align`, where `align` is the alignment size in bytes.

The assembler supports the use of some pseudo-instructions, for example, the `la` (load address) instruction in the figure. The text in the editor window is highlighted with different colors. The `Assemble`

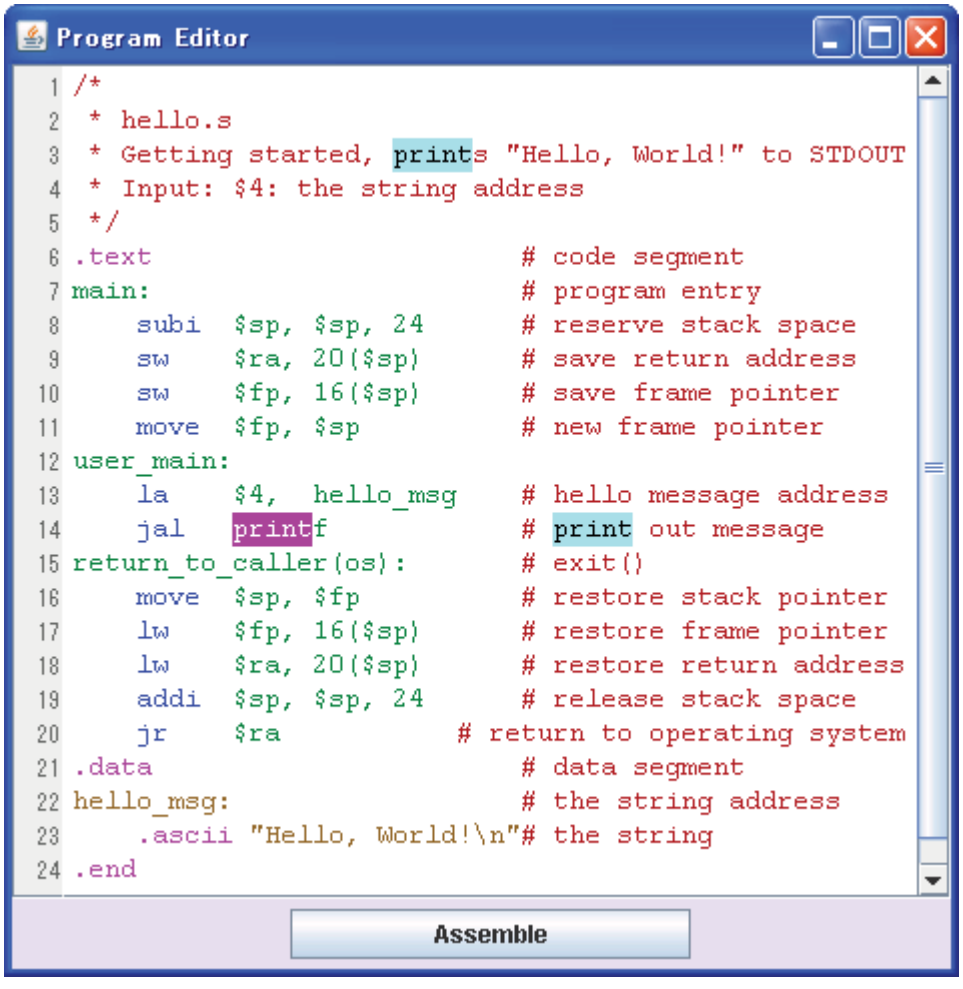

Figure 4.9 AsmSim program editor window

button assembles the program, and then the assembled program appears in the main window. If there are grammar errors in the assembly program, the errors will be pointed out in the main window.

The graphics console window (Figure 4.10) has two parts: the upper part is a standard VGA (640 × 480 pixels) virtual window and the lower part is a console window. Because of space limitations, we have shown only the partial image of the window. The console window is used for inputting commands/data and outputting messages. The VGA window is used for showing image in a VRAM (video random access memory). The user program can write true-color pixels into the VRAM directly with `sw` instruction. We also prepared some function calls to draw graphics objects and character string, which will be described later in detail.

Once a program is loaded or the program in the edit window is assembled, AsmSim enters the debug mode. In the debug mode, we can input the following debug commands in the console: `step`, `run`, `restart`, `ascii`, and `quit`. The `quit` command lets AsmSim to quit from the debug mode and enter the command line mode in which the `asmsim` command can be executed. AsmSim is provided with some MIPS assembly program examples, as listed below.

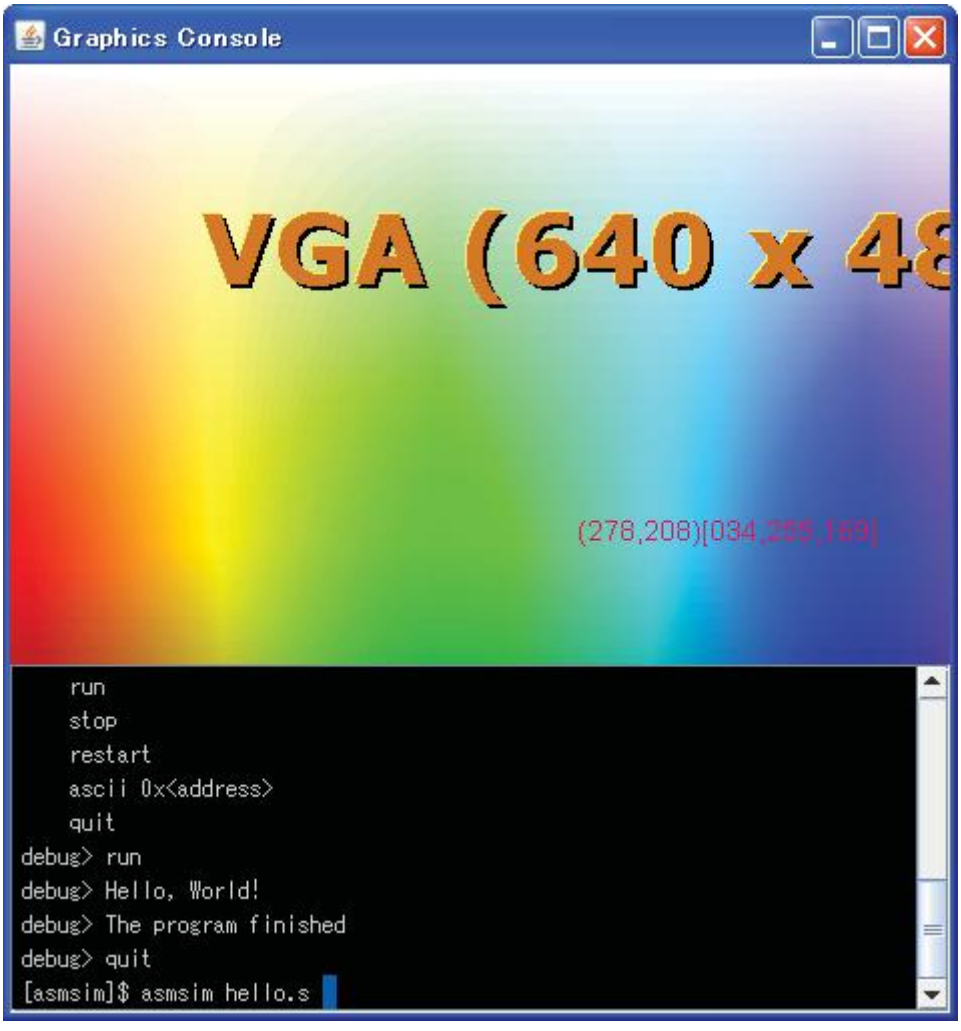

Figure 4.10 AsmSim graphics console window

```
[asmsim]$ asmsim
Usage: asmsim file
  ex., asmsim hello.s
    file:
      hello.s printf.s scanf.s getchar.s putchar.s sprintf.s getrandom.s
      gettimer.s getarrow.s getcal.s getcals.s linked_list.s sleep.s
      recursive.s malloc.s dfs.s bfs.s graphics.s key_event.s vram.s fonts.s
      kanji.s picture.s subroutine_call.s pointer.s root_nonrestoring.s
You can edit your own program in the Program Editor window, and click
Assembly button.
[asmsim]$ asmsim hello.s
```

The following describes briefly the basic operations of these examples.

1. **hello.s** displays hello message on console window as described before.
2. **printf.s** prints out a word in three formats: "`%d`" (decimal), "`%x`" (hexadecimal), and "`%s`" (string).
3. **scanf.s** reads a decimal integer, a hexadecimal integer, and a string from keyboard (need pressing Enter key).
4. **getchar.s** reads a character from keyboard (no need pressing Enter key).
5. **putchar.s** prints a character out on console window.
6. **sprintf.s** is similar to printf.s but it writes contents to a buffer.
7. **getrandom.s** gets a random number between 0 and 99.
8. **gettimer.s** gets the value of the system timer.
9. **getarrow.s** waits for pressing an arrow key.
10. **getcal.s** gets the calendar information in integer format.
11. **getcals.s** gets the calendar information in string format.
12. **linked_list.s** shows the data structure of the linked list.
13. **sleep.s** demonstrates sleeping for a moment.
14. **recursive.s** calculates Fibonacci numbers recursively. Generally, recursive programs are difficult to understand even if they are written in high-level programming languages, such as C and Java. By a single-step instruction execution, we can track the stack and see how the recursive program is implemented.
15. **malloc.s** allocates memory space.
16. **dfs.s** implements the depth-first search (DFS) algorithm. The graph is represented with the linked list.
17. **bfs.s** implements the breadth-first search (BFS) algorithm. The graph is represented with the linked list.
18. **graphics.s** draws graphics objects and string on graphics window.
19. **key_event.s** is a simple game program that demonstrates the keyboard interrupt. The game program uses four arrow keys to keep a randomly moved object to the center of graphics window.
20. **vram.s** writes true-color pixels into VRAM directly with `sw` instruction.
21. **fonts.s** shows ASCII fonts using a user-defined font table.
22. **kanji.s** displays some Japanese characters using a user-defined font table.
23. **picture.s** shows a bit-mapped picture image on graphics window.
24. **subroutine_call.s** shows how the function call in C is implemented.
25. **pointer.s** shows how the pointer in C is implemented.
26. **root_nonrestoring.s** implements a nonrestoring square root algorithm.

AsmSim supports a set of system function calls that can be invoked by the `jal` instruction. For example, the `jal printf` in the following program

```
.data                                          # data segment
print_dec:                                     # string address
    .ascii "0x%x in decimal: %d\n"             # string
a_word:                                        # data address
    .word 0x476f6f64                           # data
.text                                          # code segment
    la    $4,  print_dec                       # prepare to print
    la    $6,  a_word                          # address of data
    lw    $5,  0($6)                           # load the word
    move  $6,  $5                              # the same
    jal   printf                               # print out sum
.end
```

will print out the following message on the console:

```
debug> 0x476f6f64 in decimal: 1198485348
```

`la` (load address) is a pseudo-instruction that loads a 32-bit address to a register. It is translated into `lui` (loading upper 16 bits of the address) and `ori` (loading lower 16 bits of the address) instructions. We follow the gcc's agreements to arrange the registers for the arguments. In the example above, register `$4` holds the address of the string. In the string, the two format directives `%x` and `%d` indicate to print data in hexadecimal format and in decimal format, respectively. The corresponding data are putted into registers `$5` and `$6`. If we use `%s` (printing in string) in the string, it will print out "`good`," that is, the word `0x476f6f64` is explained as ASCIIs in the big endian format.

AsmSim supports several system calls that are commonly used in C or Java programs. We give some system calls and the register arrangements for the function parameters below.

1. **getchar()** gets a character from STDIN. The ASCII of the character will be in register `$2` after the invocation.
2. **putchar()** puts a character to STDOUT. Register `$4` holds the ASCII of the character which will be outputted.
3. **scanf()** reads an input from STDIN. Register `$4` holds the address of a string which can be "`%i`," "`%d`," "`%u`," "`%x`," or "`%s`," indicating the format of the input data. Register `$5` holds the memory address where the inputted value will be stored.
4. **printf()** writes output(s) to STDOUT. Register `$4` holds the address of a string which may contain multiple directives of "`%i`," "`%d`," "`%u`," "`%x`," "`%X`," "`%p`," and/or "`%s`." The multiple output data corresponding to the directives are putted into registers `$5`, `$6`, `$7`, memory[reg[`$sp`]+16], memory[reg[`$sp`]+20], … , in sequence.
5. **sprintf()** writes output(s) to a buffer. Register `$4` holds the address of the buffer. Register `$5` holds the address of a string which may contain multiple directives of "`%i`," "`%d`," "`%u`," "`%x`," "`%X`," "`%p`," and/or "`%s`." The multiple output data corresponding to the directives are putted into registers `$6`, `$7`, memory[reg[`$sp`]+16], memory[reg[`$sp`]+20], … , in sequence.
6. **malloc()** allocates a block of memory. Register `$4` holds the block size in bytes. The starting address of the allocated memory block will be in register `$2` after the invocation.
7. **gettimer()** gets the system timer in ms (milliseconds). The timer value will be in register `$2` after the invocation.
8. **getrandom()** gets a random integer number r with $0 \leq r < n$, where n is given in register `$4`. The random number will be in register `$2` after the invocation.
9. **getarrow()** gets an arrow key's information. The information of the pressed arrow key will be in register `$2` after the invocation: 0x8000 for the Up Arrow key; 0x8100 for the Left Arrow key; 0x8200 for the Down Arrow key; and 0x8300 for the Right Arrow key. We also defined 0xff00 for the Escape key.
10. **getcal()** gets the calendar in integers. Register `$4` holds the starting address of an integer array where the month, date, year, day, hour, minute, and second will be stored in sequence.
11. **getcals()** gets the calendar in string. Register `$4` holds the address of the string. The format of the string is "Month Date Year Day Hour:Minute:Second."
12. **sleep()** suspends execution for an interval of time. Register `$4` holds the interval in milliseconds.
13. **key_event_ena()** enables keyboard interrupt. Pressing a key will generate an interrupt and cause a transfer of control to a predefined exception handler.
14. **key_event_dis()** disables keyboard interrupt.
15. **drawline()** draws a color line on an off-screen buffer. Register `$4` holds the starting address of an integer buffer where the line color, $x1$, $y1$, $x2$, and $y2$ are stored in sequence. The line will be drawn from $(x1, y1)$ to $(x2, y2)$. The line color is defined by a 24-bit integer: 8 bits for each of red, green, and blue, from MSB to LSB.

16. **drawoval()** draws a color oval which fits into a specified rectangle on an off-screen buffer. Register `$4` holds the starting address of an integer buffer where the line color, *x*, *y*, *w*, and *h* are stored in sequence. (*x*, *y*) is the up-left coordinate of the rectangle; *w* and *h* are the width and height of the rectangle, respectively. The color is specified by a 24-bit integer.
17. **drawrect()** draws a color rectangle on an off-screen buffer. Register `$4` holds the starting address of an integer buffer where the line color, *x*, *y*, *w*, and *h* are stored in sequence. (*x*, *y*) is the up-left coordinate of the rectangle; *w* and *h* are the width and height of the rectangle, respectively. The color is specified by a 24-bit integer.
18. **drawrect3d()** is similar to drawrect() but the rectangle appears to be raised above the surface (3D). The parameters are the same as for drawrect().
19. **drawstring()** draws specified text at specified location on an off-screen buffer. Register `$4` holds the starting address of a buffer where the text color, *x*, *y*, font information, and the text are stored in sequence. (*x*, *y*) is the coordinate where the text will be drawn; the font information is given by a 24-bit integer that specifies the font name (8 bits), font type (8 bits), and font size (8 bits). The supported font names in this simulator include "Times New Roman" (0), "Arial" (1), "Courier New" (2), and "Colonna MT" (3). The supported font types are "Plain" (0), "Bold" (1), and "Italic" (2).
20. **filloval()** fills a color oval on an off-screen buffer. The parameters are the same as for drawoval().
21. **fillrect()** fills a color rectangle on an off-screen buffer. The parameters are the same as for drawrect().
22. **fillrect3d()** fills a color 3D rectangle on an off-screen buffer. The parameters are the same as for drawrect3d().
23. **refresh_vga_auto()** enables the automatic VRAM display. In this mode, the VGA image is updated whenever there is any change in the off-screen buffer.
24. **refresh_vga_manu()** means refreshing the VGA manually. It inhibits the automatic VRAM display. The VGA image is updated only when the user program executes the system call "paint()."
25. **paint()** shows the image in the off-screen buffer on the VGA. The drawing of objects and filling of objects described above are done in the off-screen buffer.

Using key_event_ena(), gettimer(), getrandom(), sleep(), and graphics draw and fill function calls, we can develop some interesting games, Tetris and Othello for example. A code fragment listed below illustrates how to use the fillrect3d() and drawstring() system calls to fill a 3D rectangle and draw a string, respectively.

```
.text                                          # code segment
main:                                          # program entry
    subi  $sp, $sp, 64                         # reserve stack space
    sw    $ra, 60($sp)                         # save return address
    sw    $fp, 56($sp)                         # save frame pointer
    move  $fp, $sp                             # new frame pointer
fill_back_ground:
    la    $4,  g_bgcolor                       # back ground color
    jal   fillrect                             # fill rectangle
fill_3d_rect:
    la    $4,  g_fillrect3d                    # fill a rect parameters
    jal   fillrect3d                           # fill a rect
draw_my_canvas_string:
    la    $4,  g_drawstring                    # draw the string parameters
    jal   drawstring                           # draw the string
paint_on_canvas:
    jal   paint                                # paint off-screen buffer
return_to_caller(os):                          # exit();
```

```
    move  $sp, $fp                          # restore stack pointer
    lw    $fp, 56($sp)                      # restore frame pointer
    lw    $ra, 60($sp)                      # restore return address
    addi  $sp, $sp,  64                     # release stack space
    jr    $ra                               # return to operating system
.data                                       # data segment
g_bgcolor:
    .word    0xffffff                       # color
    .word      0,   0                       # x, y
    .word    640, 480                       # width, height
g_drawstring:
    .word    0x000000                       # color
    .word    230, 180                       # x, y
    .word    0x020128                       # font name_type_size
    .ascii   "myCanvas"                     # string
g_fillrect3d:
    .word    0x007f00                       # color
    .word    350, 220                       # x, y
    .word    80, 50                         # width, height
.end
```

We implemented the interrupt mechanism in the AsmSim that performs functions such as addKeyListener() and removeKeyListener() in Java, for handling the keyboard interrupt. The key_event_ena() system call enables keyboard interrupt. Once the interrupt is enabled, pressing a key will cause a transfer of control to an exception handler. The system call of key_event_dis() disables keyboard interrupt.

Many applications, especially game programs, require the use of arrow keys. Because there are no ASCII codes for arrow keys, we defined 16-bit codes for four arrow keys and Escape key as described above. Once the interrupt is enabled, the 16-bit code of the pressed key is putted in register `$2` (the ASCII of an ordinary key is putted in the lower byte and the upper byte is reset to zero). The following code fragment illustrates the structure of the interrupt handler.

```
.text                                       # code segment
__Key_Event:                                # key event entry
    subi  $sp, $sp, 40                      # reserve stack space
    sw    $ra, 36($sp)                      # save return address
    sw    $fp, 32($sp)                      # save frame pointer
    sw    $3,  28($sp)                      # save $3
    sw    $4,  24($sp)                      # save $4
    sw    $5,  20($sp)                      # save $5
    move  $fp, $sp                          # new frame pointer
key_code:
    move  $5,  $2                           # key code in $2
    la    $4,  x                            # address of x_axis
test_escape:
    li    $3,  0xff00                       # escape?
    bne   $3,  $5, not_esc                  # no, check next
    sw    $3,  0($4)                        # 0xff00: outside x
    j     return                            # return
```

```
not_esc:
    li    $3,  0x8100                          # left arrow?
    bne   $3,  $5, not_left
    ...
not_left:
    li    $3,  0x8300                          # right arrow?
    bne   $3,  $5,  not_right
    ...
    move  $sp, $fp                             # restore stack pointer
    lw    $5,  20($sp)                         # restore $5
    lw    $4,  24($sp)                         # restore $4
    lw    $3,  28($sp)                         # restore $3
    lw    $fp, 32($sp)                         # restore frame pointer
    lw    $ra, 36($sp)                         # restore return address
    addu  $sp, $sp, 40                         # release stack space
    eret                                       # return from exception
main:
    ...
    jal   key_event_ena                        # enable key event
    ...
    jr    $ra                                  # return to OS
.data                                          # data segment
    ...
.end
```

In the main program, the keyboard interrupt is enabled with the instruction `jal key_event_ena`. The interrupt handler entry is `__Key_Event`, which is a predefined label. Once entered into the handler, further interrupt is disabled automatically. In the beginning of the handler, some important registers are saved into the stack. Then we can determine which key is pressed by checking the value in register `$2`, and do something according to the pressed key. Before returning from the interrupt handler, the contents of the saved registers must be restored. The instruction for returning from interrupt is `eret` (exception return).

AsmSim provides a standard VGA window which has 640 pixels per horizontal line and 480 lines. In addition to the graphics draw and fill function calls, AsmSim allows the user program to write pixels to VRAM directly. The VRAM is organized as a one-dimensional array. Suppose we want to write a pixel at the (x, y) position of the VGA, where x and y are the horizontal and vertical coordinates, respectively; then we can calculate the address of the VRAM as $a = 640 \times y + x = (512 + 128) \times y + x = (y \ll 9) + (y \ll 7) + x$.

AsmSim allows the user program to define the starting address of the VRAM. The following code illustrates how to write a pixel to VRAM. The stating address of the VRAM is defined as 0xd0000000 (to 0xd007ffff). x is in `$7`, y is in `$6`, and the pixel is in `$5`. The pixel is written to VRAM with the `sw` instruction.

```
‘define _Video_RAM 0xd0000000                  # - 0xd007ffff
.text                                          # code segment
    ...
    la    $4,  _Video_RAM                      # vram start address
    sll   $8,  $6,  9                          # y << 9
    add   $9,  $8,  $4                         # + base of vram
```

```
    sll   $8,  $6,  7                    # y << 7
    add   $8,  $8,  $7                   # + x
    add   $9,  $8,  $9                   # vram address
    sw    $5,  0($9)                     # write pixel
```

We can use `lw` and `sw` instructions to read from and write to the VRAM, respectively. The default mode is that, once a pixel is written into VRAM, it appears on the VGA immediately. But this mode will take a long time when a large number of pixels need to be written into the VRAM continuously. Therefore, we provide a system call, namely refresh_vga_manu(), to inhibit automatic display. Under this mode, the VGA image is updated only when the user program executes the system call paint(). Another system call, refresh_vga_auto(), resumes the default mode.

AsmSim also supports the user-defined font tables. For example, the following data define the 8×8 fonts of character "A."

```
ascii_font_table:
    .word 0x3c66667e, 0x66666600         # 'A', ASCII: 0x41
```

The VGA can display $640/8 \times 480/8 = 80 \times 60$ characters. The font table defines the shape of characters, and we can use different colors to show the background and foreground of the characters. The user program may maintain a character position (x, y) in VGA for $0 \le x < 80$ and $0 \le y < 60$, and calculate the VRAM addresses according to (x, y) to put the font pixels. Figure 4.11 shows the execution result of the fonts.s program, which displays the ASCII characters on graphics window.

Graph searching algorithms are subjects in the course of discrete systems, graph theory, or data structure and algorithms. There are many methods to represent a graph in computers, such as the incidence matrix, adjacency matrix, and linked list. The incidence matrix representation will lose the cycle edge information, and the adjacency matrix representation may lose the parallel edges information. AsmSim supports the data structure of the linked list. Figure 4.12 gives a graph example in which there is a cycle edge at node v_1.

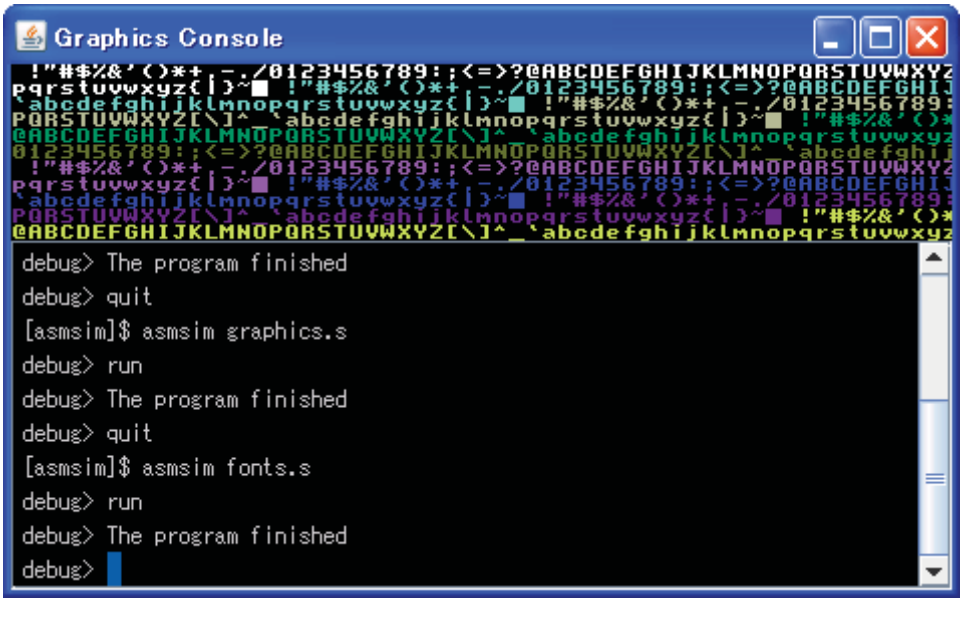

Figure 4.11 AsmSim graphics window output when executing fonts.s

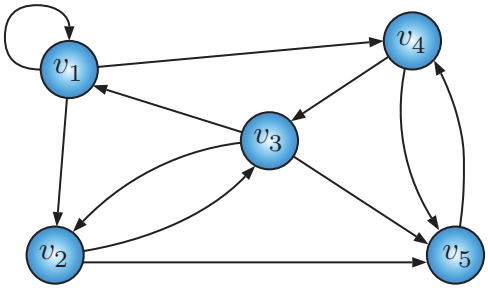

Figure 4.12 A graph that will be represented with linked lists

The graph can be represented with the following linked-list data structure. AsmSim supports the labels (pointers) in the fields of the word value.

```
`define NULL 0x0
.data
v1:  .word   1, p1                          # value, pointer
p1:  .word  v1, p2                          # value, pointer
p2:  .word  v2, p3                          # value, pointer
p3:  .word  v4, NULL                        # value, pointer
v2:  .word   2, p4                          # value, pointer
p4:  .word  v5, p5                          # value, pointer
p5:  .word  v3, NULL                        # value, pointer
v3:  .word   3, p6                          # value, pointer
p6:  .word  v1, p7                          # value, pointer
p7:  .word  v2, p8                          # value, pointer
p8:  .word  v5, NULL                        # value, pointer
v4:  .word   4, p9                          # value, pointer
p9:  .word  v5, p10                         # value, pointer
p10: .word  v3, NULL                        # value, pointer
v5:  .word   5, p11                         # value, pointer
p11: .word  v4, NULL                        # value, pointer
```

Then, we can use the assembly programming language to implement some search algorithms, such as DFS and BFS, which are generally implemented in C or Java. Suppose that the search starts from node v_1 of the graph shown in Figure 4.12. DFS algorithm visits the nodes of the graph in the order v_1, v_4, v_3, v_5, and v_2; and the BFS algorithm visits the nodes of the graph in the order v_1, v_4, v_2, v_3, and v_5.

With the editor window, you can write your own program, then assemble and execute it in the main window as well as the graphics console window. AsmSim can be executed entirely online with a web browser. There is also a `jar` (Java archive) version, which can be downloaded from Wiley's page.

4.4 ALU Design

This section describes how to design an ALU that performs the calculations of the instructions in Table 4.4. By checking the calculations of the instructions, we know that the ALU should be able to perform the following operations:

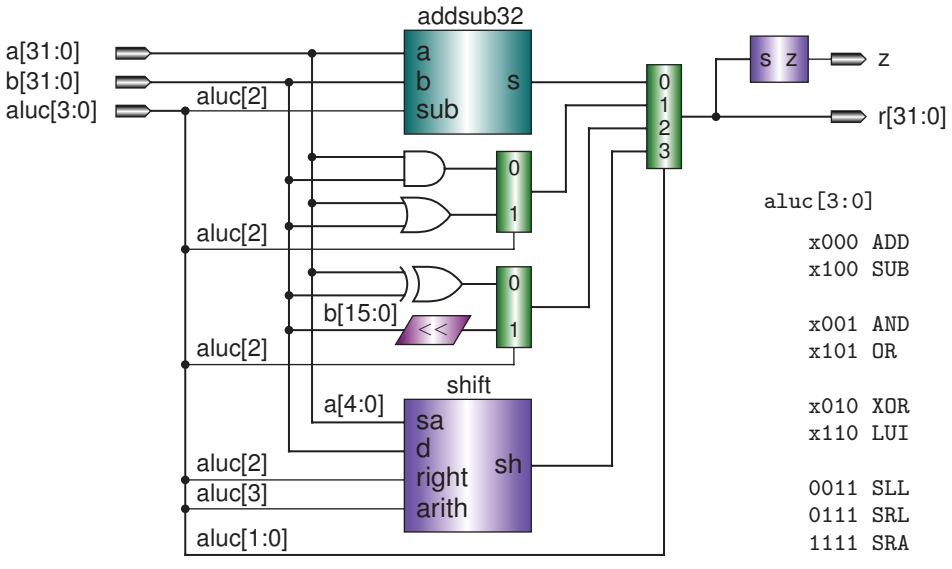

Figure 4.13 Schematic diagram of ALU

1. ADD (addition) for instructions of `add`, `addi`, `lw`, and `sw`;
2. SUB (subtraction) for instructions of `sub`, `beq`, and `bne`;
3. AND (bitwise logical and) for instructions of `and` and `andi`;
4. OR (bitwise logical or) for instructions of `or` and `ori`;
5. XOR (bitwise logical exclusive or) for instructions of `xor` and `xori`;
6. LUI (load upper immediate) for `lui` instruction;
7. SLL (shift left logical) for `sll` instruction;
8. SRL (shift right logical) for `srl` instruction;
9. SRA (shift right arithmetic) for `sra` instruction;

`jr`, `j`, and `jal` instructions do not require ALU to perform any operations. We have given almost all the components that can perform the above operations in the previous two chapters. The idea of designing ALU is to prepare these components and use a multiplexer to select an output from all the outputs of the components based on an arithmetic logic unit control (ALUC) signal which specifies what operation should be done. Figure 4.13 shows a possible circuit of ALU.

The inputs of ALU include two 32-bit data `a` and `b`, and a 4-bit `aluc` (ALUC). Three multiplexers are used: two 2-to-1 multiplexers and a 4-to-1 multiplexer. Their selection signals use some bit(s) of `aluc`. The component of `addsub32` performs ADD or SUB based on `aluc[2]`. The component of `shift` performs SLL, SRL, or SRA, based on `aluc[3:2]`. Three kinds of logic gates perform AND, OR, and XOR, respectively. LUI can be done by wiring low 16 bits to high 16 bits. The outputs of the ALU are a 32-bit `r` (result) and a 1-bit `z` (zero) flag. If the ALU result is 0, the zero flag will be a 1; otherwise it outputs a 0.

The Verilog HDL code implementing the ALU circuit shown in Figure 4.13 is given below. The corresponding operations of the `aluc` are listed in the right side of the code where an `x` denotes a don't care input.

```
module alu (a,b,aluc,r,z);              // 32-bit alu with a zero flag
    input  [31:0] a, b;                 // inputs: a, b
    input   [3:0] aluc;                 // input:  alu control: // aluc[3:0]:
```

```
    output [31:0] r;                 // output: alu result   // x 0 0 0  ADD
    output        z;                 // output: zero flag    // x 1 0 0  SUB
    wire   [31:0] d_and = a & b;                             // x 0 0 1  AND
    wire   [31:0] d_or  = a | b;                             // x 1 0 1  OR
    wire   [31:0] d_xor = a ^ b;                             // x 0 1 0  XOR
    wire   [31:0] d_lui = {b[15:0],16'h0};                   // x 1 1 0  LUI
    wire   [31:0] d_and_or  = aluc[2]? d_or  : d_and;        // 0 0 1 1  SLL
    wire   [31:0] d_xor_lui = aluc[2]? d_lui : d_xor;        // 0 1 1 1  SRL
    wire   [31:0] d_as, d_sh;                                // 1 1 1 1  SRA
    // addsub32   (a,b,sub,    s);
    addsub32 as32 (a,b,aluc[2],d_as);                        // add/sub
    // shift      (d,sa,    right,  arith,  sh);
    shift shifter (b,a[4:0],aluc[2],aluc[3],d_sh);           // shift
    // mux4x32  (a0,  a1,      a2,       a3,  s,        y);
    mux4x32 res (d_as,d_and_or,d_xor_lui,d_sh,aluc[1:0],r); // alu result
    assign z = ~|r;                                          // z = (r == 0)
endmodule
```

The modules of `mux4x32` (a 32-bit 4-to-1 multiplexer) and `shift` were already given in Chapter 2. The following Verilog HDL code implements the module of `addsub32` that performs an addition or a subtraction by invoking the module of `cla32`, a 32-bit carry-lookahead adder, which can be found in the Chapter 3.

```
module addsub32 (a,b,sub,s);                        // 32-bit adder/subtracter
    input  [31:0] a, b;                             // inputs: a, b
    input         sub;                              // sub == 1: s = a - b
                                                    // sub == 0: s = a + b
    output [31:0] s;                                // output sum s
    // sub == 1: a - b = a + (-b) = a + not(b) + 1 = a + (b xor sub) + sub
    // sub == 0: a + b = a +   b  = a +     b  + 0 = a + (b xor sub) + sub
    wire   [31:0] b_xor_sub = b ^ {32{sub}};      // (b xor sub)
    // cla32   (a, b,          ci,  s);
    cla32 as32 (a, b_xor_sub, sub, s);              // b: (b xor sub); ci: sub
endmodule
```

The simulation waveform of the ALU is shown in Figure 4.14. In sequence, we simulated ADD (`aluc` = 0), SUB (`aluc` = 4), AND (`aluc` = 1), OR (`aluc` = 5), XOR (`aluc` = 2), LUI (`aluc` = 6), SLL (`aluc` = 3), SRL (`aluc` = 7), SRA (`aluc` = f), SUB, and ADD. The zero flag `z` outputs 1 when `r` is 0.

In addition to the 20 instructions listed in Table 4.4, the instructions of exception handling, TLB management, and calculations on floating-point numbers, as listed in Table 4.7, will be also implemented in our CPU designs. These instructions will be described in the following chapters.

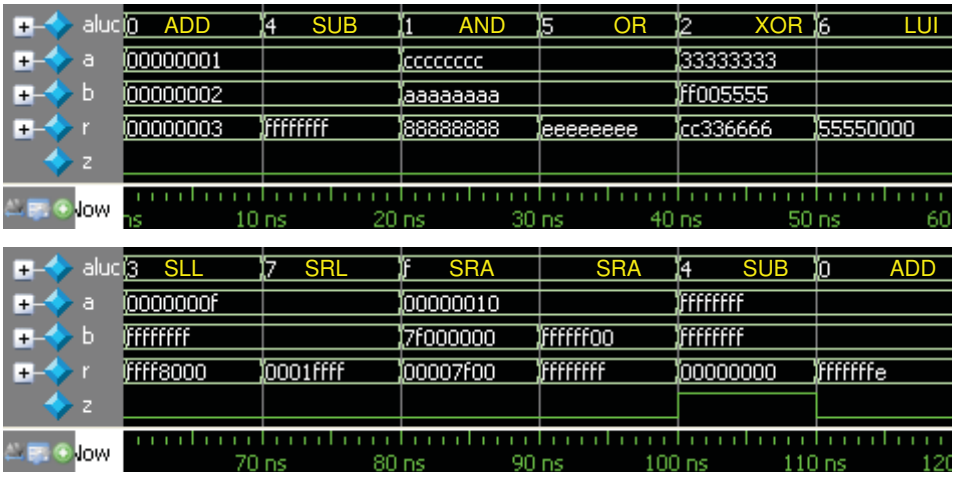

Figure 4.14 Waveform of ALU

Table 4.7 MIPS instructions related to interrupt/exception, TLB, and FP

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
syscall	000000	00000	00000	00000	00000	001100	System call
eret	010000	10000	00000	00000	00000	011000	Return from exception
tlbwi	010000	10000	00000	00000	00000	000010	Write indexed TLB entry
tlbwr	010000	10000	00000	00000	00000	000110	Write random TLB entry
mfc0	010000	00000	rt	rd	00000	000000	Load control word
mtc0	010000	00100	rt	rd	00000	000000	Store control word
lwc1	110001	rs	ft		offset		Load FP word
swc1	111001	rs	ft		offset		Store FP word
add.s	010001	10000	ft	fs	fd	000000	FP add
sub.s	010001	10000	ft	fs	fd	000001	FP subtract
mul.s	010001	10000	ft	fs	fd	000010	FP multiply
div.s	010001	10000	ft	fs	fd	000011	FP division
sqrt.s	010001	10000	00000	fs	fd	000100	FP square root

Exercises

4.1 Using only the instructions listed in Table 4.4, write code sequences to implement the following instructions.

```
bgez  rs,  label            # branch on greater than or equal to zero
bgtz  rs,  label            # branch on greater than zero
blez  rs,  label            # branch on less than or equal to zero
bltz  rs,  label            # branch on less than zero
```

4.2 Redesign the ALU in the behavioral-style Verilog HDL (you can use `casex` statement).

4.3 Add an output `v` (a flag of overflow) to ALU. When the result of addition or subtraction on the 2's complement numbers overflows, `v` outputs a 1.

4.4 Design an ALU that can perform a `sla` (shift left arithmetic) instruction (keeping the sign bit unchanged).

4.5 Try to understand the following MIPS assembly program, add comments after `#`, and execute it with AsmSim.

```
.data
$LC0:   .ascii  "%02d: q=%04x "
$LC1:   .ascii  "r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=%08x "
$LC2:   .ascii  "r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=%08x "
$LC3:   .ascii  "q=q*2+1=%04x\n"
$LC4:   .ascii  "q=q*2+0=%04x\n"
$LC5:   .ascii  "Input an unsigned integer in hex (i.e. c0000000): d = "
$LC6:   .ascii  "%x"
$LC7:   .ascii  "hex: q = %08x, r = %08x, d = q*q + r = %08x\n"
$LC8:   .ascii  "dec: q = %08d, r = %08d, d = q*q + r = %u\n"
.text
squart: subi  $sp,  $sp,  40       #
        sw    $19,  28($sp)        #
        move  $19,  $4             #
        sw    $20,  32($sp)        #
        move  $20,  $5             #
        sw    $17,  20($sp)        #
        move  $17,  $0             #
        sw    $16,  16($sp)        #
        move  $16,  $0             #
        sw    $18,  24($sp)        #
        li    $18,  0x0000000f     #
        sw    $31,  36($sp)        #
$L5:    la    $4,   $LC0           #
        move  $5,   $18            #
        move  $6,   $17            #
        jal   printf               #
```

```
        bltz  $16,  $L6          #
        sll   $2,   $16,  2      #
        sll   $3,   $18,  1      #
        srl   $3,   $19,  $3     #
        andi  $3,   $3,   0x0003 #
        or    $2,   $2,   $3     #
        sll   $3,   $17,  2      #
        ori   $3,   $3,   0x0001 #
        subu  $16,  $2,   $3     #
        la    $4,   $LC1         #
        j     $L12               #
$L6:    sll   $2,   $16,  2      #
        sll   $3,   $18,  1      #
        srl   $3,   $19,  $3     #
        andi  $3,   $3,   0x0003 #
        or    $2,   $2,   $3     #
        sll   $3,   $17,  2      #
        ori   $3,   $3,   0x0003 #
        addu  $16,  $2,   $3     #
        la    $4,   $LC2         #
$L12:   move  $5,   $16          #
        jal   printf             #
        bltz  $16,  $L8          #
        sll   $2,   $17,  1      #
        ori   $17,  $2,   0x0001 #
        la    $4,   $LC3         #
        j     $L13               #
$L8:    sll   $17,  $17,  1      #
        la    $4,   $LC4         #
$L13:   move  $5,   $17          #
        jal   printf             #
        subi  $18,  $18,  1      #
        bgez  $18,  $L5          #
        bgez  $16,  $L11         #
        sll   $2,   $17,  1      #
        ori   $2,   $2,   0x0001 #
        addu  $16,  $16,  $2     #
$L11:   sw    $16,  0($20)       #
        move  $2,   $17          #
        lw    $31,  36($sp)      #
        lw    $20,  32($sp)      #
        lw    $19,  28($sp)      #
        lw    $18,  24($sp)      #
        lw    $17,  20($sp)      #
        lw    $16,  16($sp)      #
        addi  $sp,  $sp,  40     #
        jr    $31                #
```

```
main:   subi  $sp,  $sp,  40       #
        sw    $31,  32($sp)        #
        sw    $17,  28($sp)        #
        sw    $16,  24($sp)        #
        la    $4,   $LC5           #
        jal   printf               #
        la    $4,   $LC6           #
        addi  $5,   $sp,  16       #
        jal   scanf                #
        lw    $4,   16($sp)        #
        addi  $5,   $sp,  20       #
        jal   squart               #
        move  $16,  $2             #
        mult  $16,  $16            #
        mflo  $17                  #
        lw    $7,   20($sp)        #
        la    $4,   $LC7           #
        move  $5,   $16            #
        move  $6,   $7             #
        addu  $7,   $17,  $7       #
        jal   printf               #
        lw    $7,   20($sp)        #
        la    $4,   $LC8           #
        move  $5,   $16            #
        move  $6,   $7             #
        addu  $7,   $17,  $7       #
        jal   printf               #
        lw    $31,  32($sp)        #
        lw    $17,  28($sp)        #
        lw    $16,  24($sp)        #
        addi  $sp,  $sp,  40       #
        jr    $31                  #
.end
```

4.6 Write an MIPS assembly program to show your photograph on VGA and execute it with AsmSim. You may write a program in C or Java to extract data words from the photograph file.

4.7 Design an ALU that can perform all the calculations of the MIPS integer instructions.