COMPILER DESIGN LABORATORY

TARGET MACHINE ARCHITECTURE

Brief description of the machine Architecture

SIM or ***Simple Integer Machine*** is a hypothetical machine with an elementary instruction set that supports integer arithmetic. The machine has eight **General Purpose Registers R_0..R_7**, each of which can hold an integer. The **Memory words** are numbered 0,1,2… and each one can hold an integer. The arithmetic operations supported by the ALU are addition, multiplication, subtraction, division and modulo operation on integers. The logical operations support comparison between values in two registers. The branching instructions allow control transfer based on the result of a comparison. Each instruction in the instruction set of SIM fits into one memory word. The machine has three special registers – **Stack Pointer (SP)**, **Base Pointer (BP)** and **Instruction Pointer (IP)**. The stack pointer is generally used to point to the last element of the stack and is normally initialised immediately below the global data of the program. When data is pushed on to the stack (using the push instruction) the stack pointer gets automatically *incremented*. Thus, the stack grows towards higher memory locations. The instruction pointer carries the address of the current instruction under execution and is automatically incremented to point to the next instruction to be executed after the completion of the current instruction. The base pointer is generally used to store the base address of an activation record for procedure evocations. Although any other register can act as the base pointer, availability of an explicit base pointer gives better structure and clarity to the run-time environment generation phase of program compilation.

SIM INSTRUCTION SET

SIM has eight instruction classes. All SIM arithmetic and logical instructions act on **integer operands only**.

1. Data transfer : MOV
2. Arithmetic : ADD, SUB, MUL, DIV, MOD, INR, DCR
3. Logical : LT, GT, EQ, NE, GE, LE
4. Branching : JZ, JNZ, JMP
5. Stack : PUSH, POP
6. Subroutine : CALL, RET
7. Input/Output : IN, OUT
8. Start/Halt : START,HALT

Instruction Syntax and Semantics

Comments

Comments are specified after "//" following an instruction on the same line.

Example: MOV R0, R1 // This is a comment.

Data Transfer

***Immediate Addressing*:**

MOV Ri, NUM // The value NUM is transferred to the register Ri.

Example: MOV R0, -9 // Register R0 now contains –9

***Register Addressing*:**

MOV Ri, Rj // Copy contents of Rj to Ri

Example:

MOV R0, -9

MOV R1, 8

MOV R0, R1 //R0 now contains 8

***Register Indirect Addressing*:**

MOV Ri, [Rj] // Copy contents of memory location pointed by Rj to Ri

MOV [Ri], Rj // Contents of Rj are copied to the location whose address is in Ri

Example: Let the memory location 1005 have value 1237.

MOV R0, 1005

MOV R1, [R0] // Now R1 contains 1237

***Direct Addressing*:**

MOV [LOC], Rj // Contents of Rj are transferred to the address LOC

MOV Rj, [LOC] // Contents of the memory location LOC are transferred to Rj

Example: Let the memory location 1005 have value 1237

MOV R0, [1005] // Now R0 has value 1237.

Note: Ri, Rj can be SP or BP along with other registers. *No instruction can take IP as an argument.*

Arithmetic

ADD, SUB, MUL, DIV and MOD have the following general format.

OP Ri, Rj // The result of Ri op Rj is stored in Ri

INR and DCR are used to increment/decrement the value of a register by one.

INR Rj // Similar syntax for DCR

Here Ri, Rj may be any registers except SP, BP and IP.

Example:

MOV R0, 3

MOV R1, 5

MOD R1, R0 // Now R1 stores value 2

Logical

For all logical operators the operands may be any two registers except SP, BP and IP.

a. LT Ri, Rj // Stores 1 in Ri if the value stored in Ri is less than that in Rj. Ri is set to 0 otherwise.

b. GT Ri, Rj // Stores 1 in Ri if the value stored in Ri is greater than that in Rj. Ri set to 0 otherwise.

c. EQ Ri, Rj // Stores 1 in Ri if the value stored in Ri is equal to that in Rj. Set to 0 otherwise.

d. NE Ri, Rj // Stores 1 in Ri if the value stored in Ri is not equal to that in Rj. Set to 0 otherwise.

e. GE Ri, Rj // Stores 1 in Ri if the value stored in Ri is greater than or equal to that in Rj. Set to 0 otherwise.

f. LE Ri, Rj // Stores 1 in Ri if the value stored in Ri is less than or equal to that in Rj. Set to 0 otherwise.

Branching

Branching is achieved by changing the value of the IP to the address of a specified LABEL. However, this is an implicit process and transparent to the programmer.

a) JZ Ri, LABEL // Jumps to LABEL if the contents of Ri is zero.

b)JNZ Ri, LABEL // Jumps to LABEL if the contents of Ri is not zero.

c) JMP LABEL // Unconditional Jump to instruction specified at LABEL

Here Ri can be any register except SP, BP and IP.

Example:

MOV R0, 4

MOV R1, 5

L1:NE R0, R1 // If R0 and R1 contain different values, set R0 to 1, else set R0 to 0.

JZ R0, L2 // If R0 is 0, jump to Label L2

MOV R2, 1

ADD R0, R2 // This increments the value of R0 by 1

JMP L1 // Unconditional jump to Label L1

L2: OUT R0 // This outputs the value of R0. (See discussion that follows).

Stack

SP is normally set to the address of the last element of the stack. It is the programmer's responsibility to suitably initialise SP. Stack has to be allocated in the memory of SIM.

a) PUSH Ri // Increment SP by 1and copy contents of Ri to the location pointed to by SP.

b) POP Ri // Copy contents of the location pointed to by SP into Ri and decrement SP by 1.

For both these instructions Ri may be any register except IP.

Example:

```
MOV SP, 1000 // Initialise SP to 1000

MOV R0, 7

PUSH R0 // Now the memory location 1001 contains value 7. SP takes value 1001

POP BP // Now BP contains value 7. SP has value 1000.
```

Subroutine

The CALL instruction copies the address of the next instruction to be fetched (IP + 1) on to the stack, and transfers control to the label specified. The RET instruction restores the IP value stored in the stack and continues execution fetching the next instruction pointed to by IP. The subroutine instructions provide a neat mechanism for procedure evocations.

a. CALL LABEL // Increment SP by 1, transfers IP+1 to location pointed to by SP and jumps to LABEL

b. RET // Sets IP to the value pointed to by SP and decrements SP.

Example:

```
MOV SP, 2000 // SP is initialised

MOV R0, 3

CALL L1 // SP takes value 2001 and the address of L2 is stored in that location

L2: HLT // Machine halts (See discussion below).

L1: MOV R0, 00

RET // The stack top is transferred to IP, SP is decrement to 2000.
```

Input/Output

a. IN Ri // Transfers the contents of the standard input to Ri

b. OUT Ri // Transfers the contents of Ri to the standard output

Ri can be any register except IP, BP and SP.

Example:

```
MOV R0, 6

OUT R0 // 6 is printed by the standard output
```

Start/Halt.

START // IP will be initialised to this instruction automatically when a program is taken for execution and

//execution starts from the next instruction after START

HALT // This instruction halts the machine.

Please send Error reports/Comments/Queries about this document to kmurali@nitc.ac.in