**Assembly language lecture note (lecturer 8)**

**INSTRUCTION FORMATS**

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities. They list all hardware-implemented instructions, specify their binary code format, and provide a precise definition of each instruction.

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction. The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1 An operation code field that specifies the operation to be performed.

2. An address field that designates a memory address or a processor registers.

3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction. The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields is an instruction. Operations specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5. Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1 Single accumulator organization.

2 General register organization.

3 Stack organization.

An example of an accumulator-type organization is the basic computer. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD. Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

An example of a general register type of organization is that.The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 To denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction ADD R1, R2 Would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction. Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 Denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer).

Thus transfer-type instructions need two address fields to specify the source and the destination. General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation R1 ← R + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction PUSH X Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction ADD In a stack computer consists of an operation code only with no address field.

The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU. To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

X = (A + B) ∗ (C + D). . Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

THREE-ADDRESS INSTRUCTIONS

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) ∗ (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD R1, A, B R1 ← M [A] + M [B]

ADD R2, C, D R2 ← M [C] + M [D]

MUL X, R1, R2 M [X] ← R1 ∗ R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A. The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

TWO-ADDRESS INSTRUCTIONS Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate

X = (A + B) ∗ (C + D) is as follows:

MOV R1, A R1 ← M [A]

ADD R1, B R1 ← R1 + M [B]

MOV R2, C R2 ← M [C]

ADD R2, D R2 ← R2 + M [D]

MUL R1, R2 R1 ← R1∗R2

MOV X, R1 M [X] ← R1

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

ONE-ADDRESS INSTRUCTIONS One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second and assume that the AC contains the result of tall operations.

The program to evaluate X = (A + B) ∗ (C + D) is

LOAD A AC ← M [A]

ADD B AC ← A [C] + M [B]

STORE T M [T] ← AC

LOAD C AC ← M [C]

ADD D AC ← AC + M [D]

MUL T AC ← AC ∗ M [T]

STORE X M [X] ← AC

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

ZERO-ADDRESS INSTRUCTIONS A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) ∗ (C + D) will be written for a stack organized computer. (TOS stands for top of stack)

PUSH A TOS ← A

PUSH B TOS ← B

ADD TOS ← (A + B)

PUSH C TOS ← C

PUSH D TOS ← D

ADD TOS ← (C + D)

MUL TOS ← (C + D) ∗ (A + B)

POP X M [X] ← TOS

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions.

ADDRESSING MODE The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution in dependent on the addressing mode of the instruction. The addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is.

An assembly program can be divided into three sections −

* The **data** section,
* The **bss** section, and
* The **text** section.

## The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

The syntax for declaring data section is −

section.data

## The *bss* Section

The **bss** section is used for declaring variables. The syntax for declaring bss section is −

section.bss

## The *text* section

The **text** section is used for keeping the actual code. This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins.

Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like −

; This program displays a message on screen

or, on the same line along with an instruction, like −

add eax, ebx ; adds ebx to eax

## Assembly Language Statements

Assembly language programs consist of three types of statements −

* Executable instructions or instructions,
* Assembler directives or pseudo-ops, and
* Macros.

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.

**Macros** are basically a text substitution mechanism.