BIRSA INSTITUTE OF TECHNOLOGY (TRUST)

NH-33, GETLATU, RANCHI

Department: - Electronics and Communication Engineering
Lecture notes
Semester: - 4th
Subject: - Digital Technologies and Microprocessor
Lecturer: - Alok Kumar Singh

ASSEMBLY LANGUAGE PROGRAMS

A. About ALP:

Assembly languages are a family of low-level languages for programming computers, microprocessors, microcontrollers, and other (usually) integrated circuits. They implement a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called mnemonics) that help the programmer remember individual instructions, registers, etc. An assembly language is thus specific to certain physical or virtual computer architecture (as opposed to most high-level languages, which are usually portable).

A utility program called an assembler is used to translate assembly language statements into the target computer's machine code. The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data. This is in contrast with high-level languages, in which a single statement generally results in many machine instructions.

Many sophisticated assemblers offer additional mechanisms to facilitate program development, control the assembly process, and aid debugging. In particular, most modern assemblers include a macro facility (described below), and are called macro assemblers.

Assemblers are generally simpler to write than compilers for high-level languages, and have been available since the 1950s. Modern assemblers, especially for RISC based architectures, such as MIPS, Sun SPARC, HP PA-RISC and x86(-64), optimize instruction scheduling to exploit the CPU pipeline efficiently.

There are two types of assemblers based on how many passes through the source are needed to produce the executable program. One pass assemblers go through the source code once and assumes that all symbols will be defined before any instruction that

references them. Two pass assemblers (and multi-pass assemblers) create a table with all unresolved symbols in the first pass, then use the 2nd pass to resolve these addresses. The advantage in one pass assemblers is speed - which is not as important as it once was with advances in computer speed and capabilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in the program source. As a result, the program can be defined in a more logical and meaningful way. This makes two-pass assembler programs easier to read and maintain.

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing
- Object-Oriented features such as encapsulation, polymorphism, inheritance, interfaces.

B. BASIC ELEMENTS

Any Assembly language consists of 3 types of instruction statements which are used to define the program operations:

1. Opcode mnemonics
2. Data sections
3. Assembly directives

1. Opcode mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level languages. Generally, an opcode is a symbolic name for a single executable machine language instruction, and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or

opcode plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be either immediate (typically one byte values, coded in the instruction itself) or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works.

2. Data sections

There are instructions used to define data elements to hold data and variables. They define what type of data, length and alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined.

3. Assembly directives / pseudo-ops

Assembly Directives are instructions that are executed by the Assembler at assembly time, not by the CPU at run time. They can make the assembly of the program dependent on parameters input by the programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of the program to make it easier for the programmer to read and maintain. The names of pseudo-ops often start with a dot to distinguish them from machine instructions.

Some assemblers also support *pseudo-instructions*, which generate two or more machine instructions. Most assemblers provide flexible symbol management, allowing programmers to manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with replaceable addresses.

Assembly languages, like most other computer languages, allow comments to be added to assembly source code that are ignored by the assembler. Good use of comments is even

more important with assembly code than with higher-level languages, as the meaning and purpose of a sequence of instructions is harder to decipher from the code itself.

Wise use of these facilities can greatly simplify the problems of coding and maintaining low-level code. *Raw* assembly source code as generated by compilers or dis assemblers — code without any comments, meaningful symbols, or data definitions — is quite difficult to read when changes must be made.

PROGRAM CONTROL INSTRUCTIONS

Program control instructions change or modify the flow of a program. The most basic kind of program control is the **unconditional branch** or **unconditional jump**. **Branch** is usually an indication of a short change relative to the current program counter. **Jump** is usually an indication of a change in program counter that is not directly related to the current program counter (such as a jump to an absolute memory location or a jump using a dynamic or static table), and is often free of distance limits from the current program counter.

The penultimate kind of program control is the **conditional branch** or **conditional jump**. This gives computers their ability to make decisions and implement both loops and algorithms beyond simple formulas.

Most computers have some kind of instructions for **subroutine call** and **return** from subroutines.

There are often instructions for **saving** and **restoring** part or all of the processor state before and after subroutine calls. Some kinds of subroutine or return instructions will include some kinds of save and restore of the processor state.

Even if there are no explicit hardware instructions for subroutine calls and returns, subroutines can be implemented using jumps (saving the return address in a register or memory location for the return jump). Even if there is no hardware support for saving the processor state as a group, most (if not all) of the processor state can be saved and restored one item at a time.

NOP, or no operation, takes up the space of the smallest possible instruction and causes no change in the processor state other than an advancement of the program counter and any time related changes. It can be used to synchronize timing (at least crudely). It is often used during development cycles to temporarily or permanently wipe out a series of instructions without having to reassemble the surrounding code.

Stop or **halt** instructions bring the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset, or external action.

Reset instructions reset the processor. This may include any or all of: setting registers to an initial value, setting the program counter to a standard starting location (restarting the computer), clearing or setting interrupts, and sending a reset signal to external devices.

- **JMP** Jump; Intel 80x86; unconditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed])
- **JMP** Jump; MIX; unconditional jump to location M; J-register loaded with the address of the instruction which would have been next if the jump had not been taken
- **Jcc** Jump Conditionally; Intel 80x86; conditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed]) based on a tested condition: JA/JNBE, JAE/JNB, JB/JNAE, JBE/JNA, JC, JE/JZ, JNC, JNE/JNZ, JNP/JPO, JP/JPE, JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JNO, JNS, JO, JS
- **Jcc** Jump on Condition; MIX; conditional jump to location M based on comparison indicator; if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; JL (less), JE (equal), JG (greater), JGE (greater-or-equal), JNE (unequal), JLE (less-or-equal)
- **LOOP** Loop While ECX Not Zero; Intel 80x86; used to implement DO loops, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero then the program continues to the next instruction (exiting the loop), otherwise the program makes a byte branch to continue the loop; does not modify flags
- **LOOPE** Loop While Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit

the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPZ; does not modify flags

- **LOOPNE** Loop While Not Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNZ; does not modify flags
- **LOOPNZ** Loop While Not Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNE; does not modify flags
- **LOOPZ** Loop While Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPE; does not modify flags
- **JCXZ** Jump if Count Register Zero; Intel 80x86; conditional jump if CX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **JECXZ** Jump if Extended Count Register Zero; Intel 80x86; conditional jump if ECX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **CALL** Call Procedure; Intel 80x86; pushes the address of the next instruction following the subroutine call onto the system stack, decrements the system stack pointer, and changes program flow to the address specified (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register or memory location])

- **RET** Return From Procedure; Intel 80x86; fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **IRET** Return From Interrupt; Intel 80x86; transfers the value at the top of the system stack into the flags register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **PUSHA** Push All Registers; Intel 80x86; move contents all 16-bit general purpose registers to memory pointed to by stack pointer (in the order AX, CX, DX, BX, original SP, BP, SI, and DI); does not affect flags
- **POPA** Pop All Registers; Intel 80x86; move memory pointed to by stack pointer to all 16-bit general purpose registers (except for SP); does not affect flags
- **NOP** No Operation; no change in processor state other than an advance of the program counter
- **HLT** Halt; stop machine, computer restarts on next instruction