Ananda Gunawardena

Lecture 26
Assembler Fundamentals

All programs written in a high-level language like C are converted into machine language so they can be executed by the underlying hardware. However, the process of converting high level source code to machine language goes through several intermediate steps. One of them is the conversion of source code into **assembly language** instructions native to the hardware. Converted assembly code is optimized by the C compiler so they can be executed more efficiently by the hardware. Assembly code from high level code is then **assembled** using a program called **"assembler"** into object code which then in turn linked up with supporting library code to generate the **executable code**. We know that it is difficult to program in machine language, and therefore assembly language provides an intermediate step where programs can be written using English like instructions using direct instructions that can manipulate hardware. These assembly instructions closely mimic how they are carried out by the hardware. A typical assembly code instruction contains an operation code (opcode), an English description of what operation a hardware is supposed to perform and operands to support the operation(if any). For example, an assembly instruction to add two "registers" and place the answer in another register may look like

```
add r1,r2,r3
```

where add is the operations code (opcode) for the assembly instruction. We note that assembly instructions may vary from architecture to architecture.

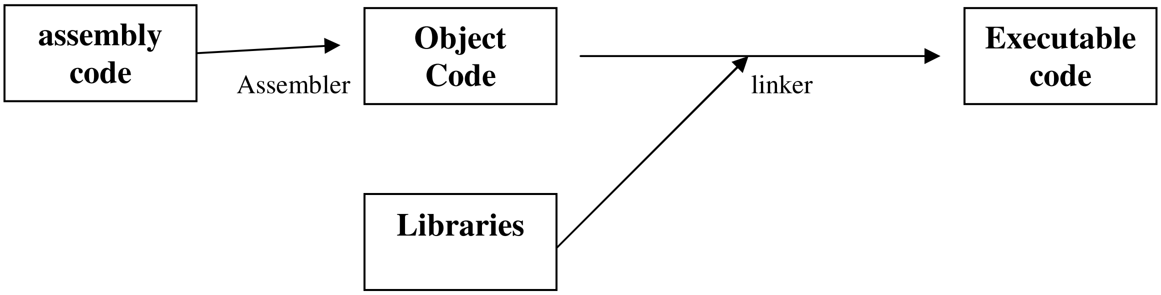
A collection of assembly language instructions with other operating systems directives, form a complete assembly language program. An example of an assembly program in i686 is given by (man uname)

```
#listing 1
.global main
main:
      movl $20, %eax
      ret
```

The above program is a simple program that moves the content 20 to a register eax. The $ identifies immediate values and % identify a register. Assuming that above program is in a file name first.s, the program can be compiled with gcc compiler and generate a binary executable file called a.out by using

- gcc first.s
- ./a.out

The generic process of generating the executable from assembly code is shown below.



High level language instructions and assembly instructions are written using readable text. Assembly language instructions are architecture dependent. For example, Intel family of processors may understand one type of assembly language instructions. The opcode of an assembly language instruction may change from architecture to architecture. However, opcodes like **add, sub, mul** instructions that represent addition, subtraction, and multiply are common to most assembly language instructions.

Instruction Set Architecture (ISA)

Instruction set architecture (ISA) provides a perspective of the processor from assembly language or machine language programmer's point of view. In simple terms, ISA describes the instructions that processor understands, including register set and how the memory is organized etc. A real world processor ISA would include few additional items such as data types; interrupt handlers, exception handling etc. ISA is part of the computer architecture specific to a particular hardware.

Registers

Registers are special purpose memory locations built into the processor that are on the top of the memory hierarchy. Most assembly instructions directly operate on registers, loading values into registers from memory, performing operations on them and storing answers back in the memory. The registers are named like **eax, ebx, ecx** etc and registers **ebp and esp** are used for manipulating the base pointer and stack pointer, which we will visit later. The size of a register (say 32-bit) and number of registers (say 8) depends on particular computer architecture. A typical instruction written in GNU assembly that operates on a register looks like

movl $10, %eax

instructs moving the value 10 immediately to register eax. There are address registers, data registers, constant value registers, conditional registers etc used for purposes like storing the address, storing data, storing a value used to initialize(eg:zero), or values used for holding the truth value of a condition. There is also an instruction register(IR) that can hold the instruction currently being executed.

A Hypothetical Machine (SAM)

To understand how computers are organized and how they carry out program instructions, let us assume a hypothetical machine with seven general purpose registers and an additional register that contains the value zero for initializing other registers. Our computer also contains a special purpose register called **program counter** (PC) that keeps track of the current address in the memory. After executing an instruction, PC moves forward by 4 bytes to load and execute the next instruction. During the execution of the program, an instruction is loaded from memory into the **instruction register (IR).** Instruction register help decode and carry out the instruction. Let us take a look at the registers and instruction set of our hypothetical machine.

Registers

Register	Number	Notes
Z	000	Constant: Always zero (0)
A	001	
B	010	
C	011	
D	100	
E	101	
F	110	
G	111	
PC		Program Counter. 24 bits wide. Not addressable
IR		Instruction Register. 32 bits wide. Not addressable.

We assume that the PC is 24 bits wide and therefore our simulated memory has $2^{24} - 1$ addressable bytes. Now we can define a 4-bit instruction set for our hypothetical architecture. The basic instructions for a computer are quite simple consisting of *branch instructions, I/O instructions, Arithmetic instructions, device instructions and comparison instructions*. Using 4-bits we can define 16 different instructions as listed below.

Instructions for SAM

CONTROL INSTRUCTIONS

Instruction	Op	Address	Function
HLT	0000	XXXX XXXX XXXX XXXX XXXX XXXX XXXX	Stop simulation
JMP	0001	0000 AAAA AAAA AAAA AAAA AAAA AAAA	Jump (line number)
CJMP	0010	0000 AAAA AAAA AAAA AAAA AAAA AAAA	Jump if true
OJMP	0011	0000 AAAA AAAA AAAA AAAA AAAA AAAA	Jump if overflow

LOAD-STORE INSTRUCTIONS

Instruction	Op	Register	Value	Function
LOAD	0100	0RRR	AAAA AAAA AAAA AAAA AAAA AAAA	Load (hex address)
STORE	0101	0RRR	AAAA AAAA AAAA AAAA AAAA AAAA	Store (hex address)
LOADI	0110	0RRR	0000 0000 IIII IIII IIII IIII	Load Immediate
NOP	0111	0000	0000 0000 0000 0000 0000 0000	no operation

MATH INSTRUCTIONS

Instruction	Op	Reg0	Reg1	Reg2		Function
ADD	1000	0RRR	0RRR	0RRR	0000 0000 0000 0000	Reg0 = (Reg1 + Reg2)
SUB	1001	0RRR	0RRR	0RRR	0000 0000 0000 0000	Reg0 = (Reg1 - Reg2)

DEVICE I/O

Instruction	-Op-	Reg0	0000 0000 0000 0000	Port	Function
IN	1010	0RRR	0000 0000 0000 0000	PPPP PPPP	Read Port into Reg0
OUT	1011	0RRR	0000 0000 0000 0000	PPPP PPPP	Write Reg0 out to Port

COMPARISON

Instruction	-Op-	Reg0	Reg1		Function
EQU	1100	0RRR	0RRR	0000 0000 0000 0000 0000	Cflg = (Reg0 == Reg1)
LT	1101	0RRR	0RRR	0000 0000 0000 0000 0000	Cflg = (Reg0 < Reg1)
LTE	1110	0RRR	0RRR	0000 0000 0000 0000 0000	Cflg = (Reg0 <= Reg1)
NOT	1111	0000	0000	0000 0000 0000 0000 0000	Cflg = (!Cflg)

Writing an Assembly Program

Now let us write a simple program in the assembly language defined for our hypothetical machine. Writing an assembly program requires the understanding of the language as well as how underlying hardware will carry out your instructions. We will start with some simple programs.

Program 1: Write a program to add the numbers 10 and 15 and output to port #15 (output port)

```
0 LOADI    A        10         # Load number 10 into register A
1 LOADI    B        15         # Load number 15 into register B
2 ADD      C        A   B      # Add registers A and B, store in C
3 OUT      C        15         # Print out the number to output port
```

Program 2: Write a program that reads a single digit integer from keyboard and output

```
0 LOADI    A        48         # Load number ‘0’ into register A
1 IN       B        0          # read a character from Port 0 (in)
2 SUB      C        B   A      # convert character to int and store in C
3 OUT      C        15         # print out the number to output port
```

Program 3: Write a program that reads a single digit integer from keyboard and output the number if the number is greater or equal to 5.

```
0 LOADI     A       48         # Load number ‘0’ into register A
1 IN        B       0          # read a character from Port 0 (in)
2 SUB       C       B   A      # convert character to int and store in C
3 LOADI     B    5             # load number 5 into register B
4 LTE     C    B               # C < B
5 CJMP    {line 7}              # if true jump to end of program
6 OUT       C       15         # print out the number to output port
7 HLT                          # terminate the program
```

Program 4: Write a program that reads a single digit integer from keyboard and output all numbers between 1 and number

```
0 LOADI     A       48         # Load number ‘0’ into register A
1 IN        B       0          # read a character from Port 0 (in)
2 SUB       C       B   A      # convert character to int and store in C
3 LOADI   B    1               # load number 1 into register B
4 LTE     C    B               # is C < 1?
5 CJMP   {line 9}               # if true jump to end of program
6 OUT     C         15         # print out the number to output port
7 SUB     C    C   B           # C = C - 1
8 JMP     {line 4}             # go to line 4
9 HLT                          # terminate the program
```

Exercises:

1. Write an assembly program (using our hypothetical assembly language) to read two non-negative integers greater than 1 from stdin(port #0), multiply and output the answer to stdout(port #15)

2. Write an assembly program that reads two one digit numbers and output the max of the two. Extend the program to read 3 one digit integers and find max.

3. Write an assembly program that can read an integer of any length from the stdin and output the number.

4. Write an assembly program that can read any integer (as in #2) and find the factorial of the number.

5. Convert each of the assembly programs you wrote in (1)-(4) to machine code using the codes defined above. That is each program is displayed as instructions containing 0's and 1's.

Answers:

1. Write an assembly program (using our hypothetical assembly language) to read two non-negative integers from stdin(port #0), multiply and output the answer to stdout(port #15)

```
0     LOADI   A       48          # Load number '0' into register A
1     IN      B       0           # read a character from Port 0 (in)
2     SUB     C       B    A      # convert character to int and store in C
3     IN      B       0           # read a character from Port 0 (in)
4     SUB     D       B    A      # convert character to int and store in D
5     LOADI   G       1           # Load 1 to register D
6     LOADI   E       0           # initialize a sum
7     EQU     C       G           # is C = 1?
8      CJMP   {line 12 }          # if true we are done
9     ADD      E      E   D      # sum = sum + D
10    SUB     C       C   G     # c = c-1
11    JMP     {line 7}\
12    OUT     E       15          #output the product
```

2. Write an assembly program that reads two one digit numbers and output the max of the two. Extend the program to read 3 one digit integers and find max.

```
0     LOADI    A      48          # Load number '0' into register A
1     IN       B      0           # read a character from Port 0 (in)
2     SUB      C      B    A      # convert character to int and store in C
3     IN      B       0
4     SUB     D        B   A
5     LTE     C       D
6     CJMP    {line 9}
7     OUT     C       15
8     JMP     {line 10}
9     OUT     D       15
10    HLT
```

3. Write an assembly program that can read an integer of any length from the stdin and output the number.

4. Write an assembly program that can read any integer (as in #2) and find the factorial of the number.

5. Convert each of the assembly programs you wrote in (1)-(4) to machine code using the codes defined above. That is each program is displayed as instructions containing 0's and 1's.