Lecture 1

ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

An assembly language is a **machine dependent**, **low level** programming language which is **specific** to a certain computer system

Three basic features

- Mnemonic operation codes
- Symbolic operands
- Data declarations

Statement format

[Label] <Opcode> <operand spec>[,<operand spec>..]

If a label is specified in a statement, it is associated as a symbolic name with memory word (s) generated for the statement

<operand spec> has following syntax

<symbolic name> [+<displacement>][(<index register>)]

Each statement has **two operands**

First operand is always a register which can be any one of AREG, BREG, CREG and DREG

Second operand refers to a memory word using a symbolic name and an optional displacement

Example

MOVER AREG, A

This statement moves the contents from A to AREG

MOVER AREG, A+5

This statement moves the contents from A+5 (i.e. address of A with displacement 5) to AREG

Mnemonic operation codes

Instruction opcode	Assembly mnemonic	Remarks
00	STOP	Stop execution
01	ADD	First operand is modified. Condition code is set
02	SUB	
03	MULT	
04	MOVER	Register← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	First operand is not used
10	PRINT	

The assembly statement for BC (Branch condition)

BC <condition code spec>, <memory address>

It transfers control to memory word with the address <memory address> if current value of condition code matches <condition code spec>

<condition code spec> is a character string whose value can be GT, LT, EQ, NE, LE, LT

BC statement with condition code spec **<u>ANY implies unconditional transfer</u>** of control

The opcode, register operand and memory operand occupy 2,1 and 3 digits respectively

Sign is not a part of instruction

Condition code specified in a specified in BC statement is encoded into first operand position using codes 1-6 for specifications **LT, LE, EQ, GT, GE and ANY**

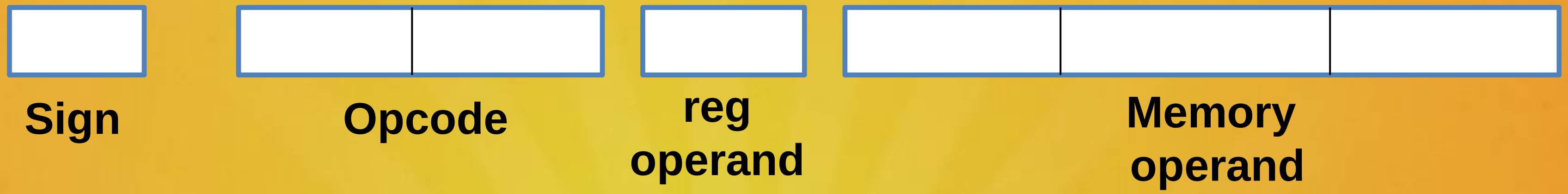


Fig : Instruction Format

	START	101		
	READ	N	101)	+ 09 0 113
	MOVER	BREG, ONE	102)	+ 04 2 115
	MOVEM	BREG, TERM	103)	+ 05 2 116
AGAIN	MULT	BREG, TERM	104)	+ 03 2 116
	MOVER	CREG, TERM	105)	+ 04 3 116
	ADD	CREG, ONE	106)	+ 01 3 115
	MOVEM	CREG, TERM	107)	+ 05 3 116
	COMP	CREG, N	108)	+ 06 3 113
	BC	LE, AGAIN	109)	+ 07 2 104
	MOVEM	BREG, RESULT	110)	+ 05 2 114
	PRINT	RESULT	111)	+ 10 0 114
	STOP		112)	+ 00 0 000
N	DS	1	113)	
RESULT	DS	1	114)	
ONE	DC	‘1’	115)	+ 00 0 000
TERM	DS	1	116)	
	END			

Assembly language statements

Imperative statements

Declaration statements

Assembler directives

Imperative statement

Indicates an **<u>action</u>** to be performed during execution of assembled program

Each imperative statement translates into one machine instruction

Example

```
MOVER     AREG, A
ADD       AREG, B
```

Declaration statement

DS (declare storage) statement reserves areas of memory and associates names with them

```
    [Label]   DS     <constant>
```

Example

```
        A     DS     1
        G     DS     200
```

DC (declare constant) constructs memory words containing constants

```
[Label]   DC    '<value>'
```

Example :

```
ONE       DC    '1'
```

The statement associates name ONE with memory word containing value ‘1’

Use of constants

DC statement does not really implement constants

It initializes memory words to given values

Values may be changed by moving a new value into the memory word

```
        MOVEM       BREG, ONE
```

Usage of constants in two ways

1. Immediate operands
2. Literals

Immediate operands can be used in assembly statement only if the architecture of target machine includes the necessary features

```
ADD        AREG, 5
```

Here, 5 is an immediate operand

A literal is an operand with the syntax **=‘<value>’**

It differs from a constant because its location cannot be specified in assembly program (but there will be difference of one literal with another based on literal pool, hence memory not allocated for one literal but there is for literal pools)

This helps to ensure that **its value is not changed during execution**

```
        ADD   AREG, ='5'
```

It is equivalent to arrangement using DC statement

```
             ADD     AREG, FIVE
             -- --
     FIVE    DC      '5'
```

When assembler encounters use of literal in operand field of a statement, it handles the literal using arrangement similar to second example above

Value of literal is protected by the fact that the name and address of this word is not known to the assembly language programmer

Assembler directives

They instruct the assembler to perform certain actions during the assembly of a program

START <constant>

This directive indicates that the first word of the target program generated by the assembler should be placed in memory word with address <constant>

END [<operand spec>]

This directive indicates the end of source program

<operand spec> indicates address of instruction where the execution of program should begin

By default, execution begins with the first instruction of the assembled program

Advantage of Assembly Language

Primary advantages of assembly language programming arise from **the use of symbolic operand** specifications

Example

Compute ½ x N!

Font Color highlight changes in program

```
          START    101
          READ     N                  101)   + 09 0 114
          MOVER    BREG, ONE          102)   + 04 2 116
          MOVEM    BREG, TERM         103)   + 05 2 117
AGAIN     MULT     BREG, TERM         104)   + 03 2 117
          MOVER    CREG, TERM         105)   + 04 3 117
          ADD      CREG, ONE          106)   + 01 3 116
          MOVEM    CREG, TERM         107)   + 05 3 117
          COMP     CREG, N            108)   + 06 3 114
          BC       LE, AGAIN          109)   + 07 2 104
          DIV      BREG, TWO          110)   + 08 2 118
          MOVEM    BREG, RESULT       111)   + 05 2 115
          PRINT    RESULT             112)   + 10 0 115
          STOP                        113)   + 00 0 000
N         DS       1                  114)
RESULT    DS       1                  115)
ONE       DC       ‘1’                116)   + 00 0 001
TERM      DS       1                  117)
TWO       DC       ‘2’                118)   + 00 0 001
          END
```

One statement has been inserted before the PRINT statement to implement division by 2

This leads to changes in addresses of constants and reserved memory areas

Addresses used in most instructions of the program had to change

A SIMPLE ASSEMBLY SCHEME
Design specification of an assembler

Four step approach to develop a design specification for an assembler

1.Identify the information necessary to perform a task

2.Design a suitable data structure to record the information

3.Determine the processing necessary to obtain and maintain the information

4.Determine the processing necessary to perform the task

Analysis phase

Primary function is the **building of symbol table**

Must determine the addresses with which symbolic names used in a program are associated

To determine the address of N, we must fix the addresses of all program elements preceding it

This function is called **memory allocation**

To implement memory allocation a data structure called **location counter (LC)** is introduced
LC is always made to contain **address of the next memory word** in target program
It is **initialized to the constant specified in the START** statement

Whenever analysis phase sees a label in assembly statement, it enters the label and contents of LC in a new entry of symbol table

It then finds the number of memory words required by assembly statement and updates LC contents

This ensures that LC points to next memory word in target program even when machine instructions have different lengths and DS/DC statements reserve different amounts of memory

To update contents of LC, analysis phase needs to know the **lengths** of different instructions

Information simply depends on assembly language, hence mnemonics table can be extended to include this information in a new field called length

Refer to processing involved in maintaining the LC as **LC processing**

Synthesis phase

Consider the assembly statement

```
MOVER      BREG, ONE
```

Following information to synthesize the machine instruction corresponding to this statement

1. Address of the memory word with which name ONE is associated
2. Machine operation code corresponding to the mnemonic MOVER

Use of two data structures during synthesis phase

- Symbol table
- Mnemonics table

Each entry of symbol table has two primary fields

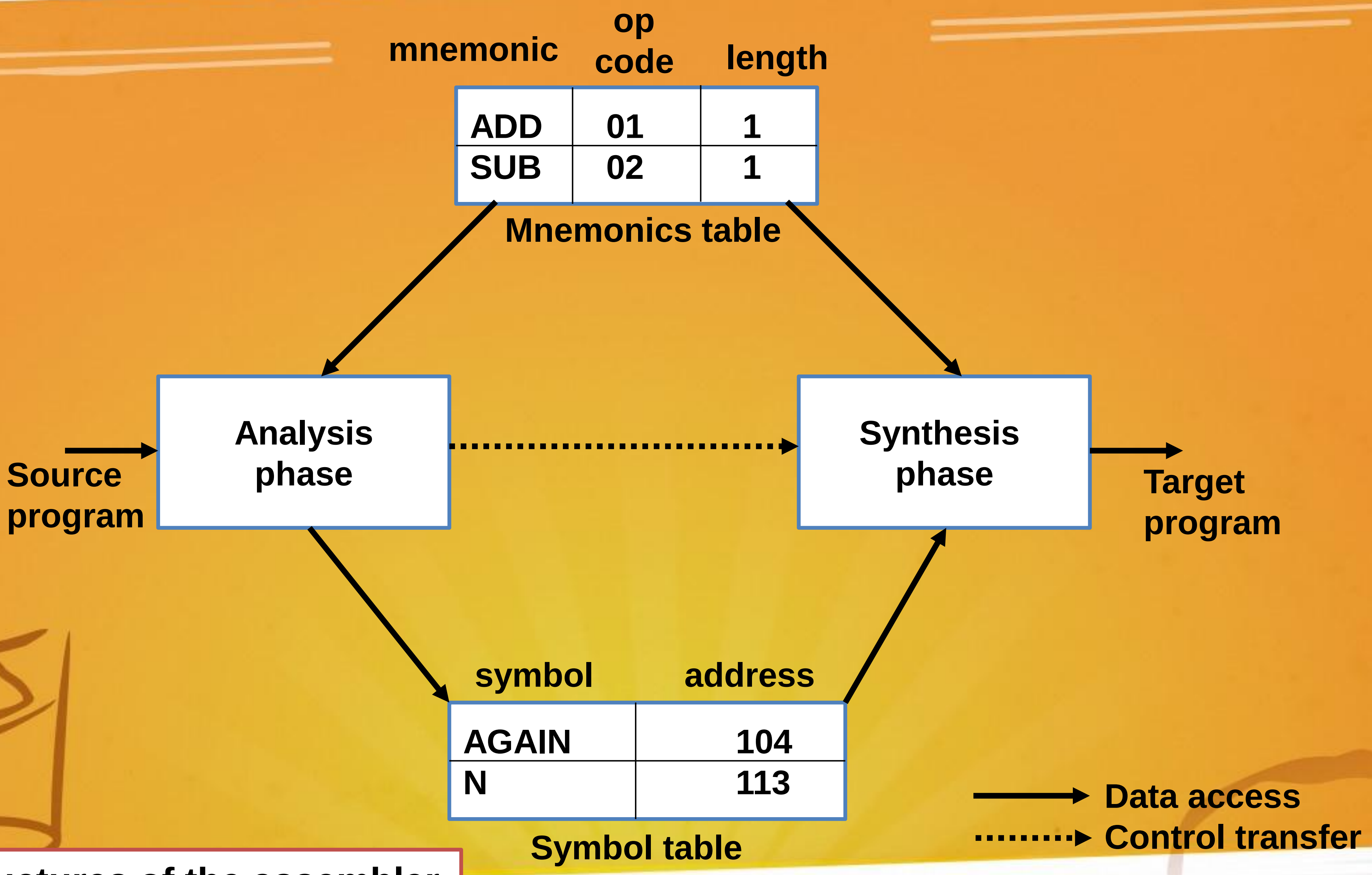
- Name
- Address

Table is built by analysis phase

An entry in mnemonics table has two primary fields

- Mnemonic
- Opcode

Synthesis phase uses these tables to obtain machine address with which a name is associated, and machine opcode corresponding to mnemonic
Hence tables have to be searched with symbol names and the mnemonics as keys



mnemonic	op code	length
ADD	01	1
SUB	02	1

symbol	address
AGAIN	104
N	113

Fig : Data structures of the assembler

Mnemonic table is a fixed table which is merely accessed by analysis and synthesis phases

Symbol table is constructed during analysis and used during synthesis

Tasks performed by each phase

Analysis phase

1. Isolate the label, mnemonic opcode and operand fields of a statement
2. If label is present, enter the pair (symbol,<LC contents>) in a new entry of symbol table
3. Check validity of mnemonic opcode through a look-up in mnemonics table
4. Perform LC processing, i.e. update the value contained in LC by considering the opcode and operands of the statement

Synthesis phase

1. Obtain the machine opcode corresponding to the mnemonic from the Mnemonics table
2. Obtain address of memory operand from the Symbol table
3. Synthesize a machine instruction or machine form of a constant, as the case may be