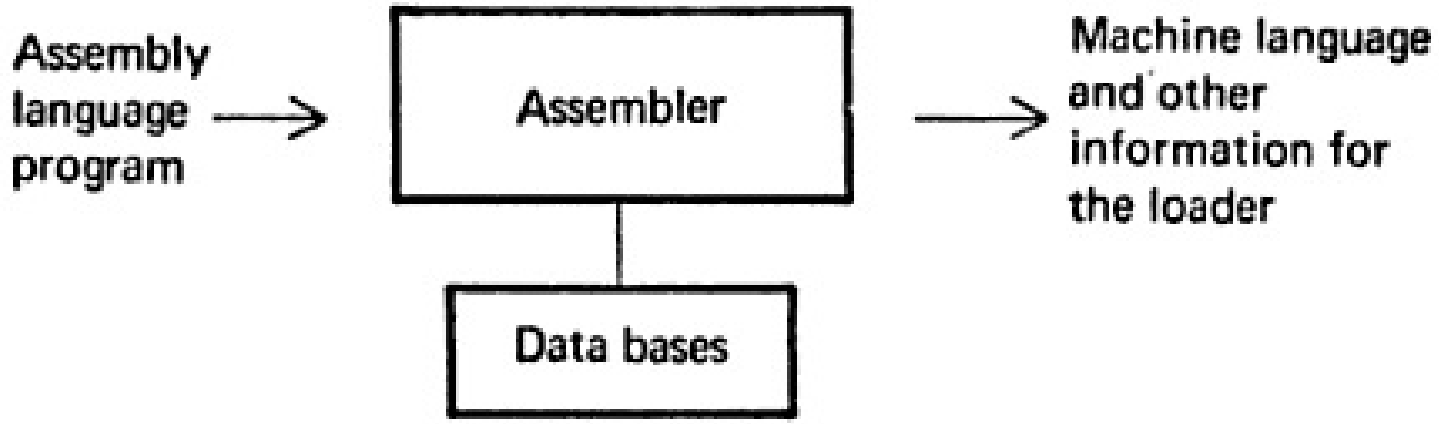
Two-Pass Assembler

Components of System Programming

- Assemblers
- Loaders
- Linker
- Macros
- Compilers
- Formal Systems

Assembler

- An assembler is a program that accept an assembly language program and produces it machine language equivalents



Assembly Language

- An assembly language is a machine dependent, low level programming language which is specific to certain computer system
- An assembly language statement has the format

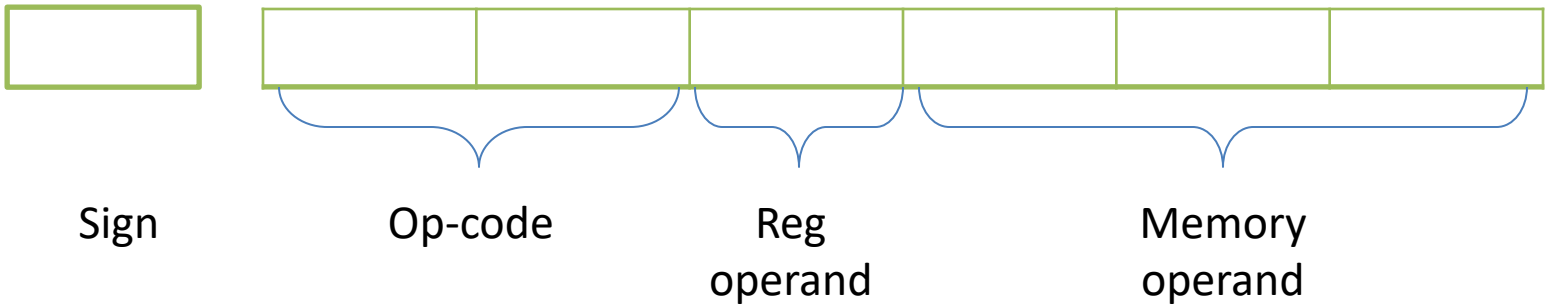
[label] <Opcode> <operand spec>

Mnemonic Operations

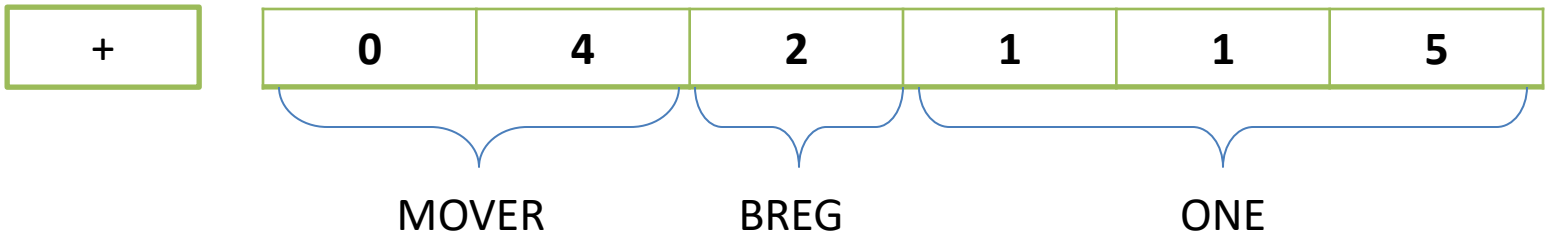
Instruction Op-code	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	
02	SUB	
03	MULT	
04	MOVER	Move memory to register
05	MOVEM	Move register to memory
06	COMP	Set condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	
10	PRINT	

Machine Instruction Format

- The machine instruction format is



- For example



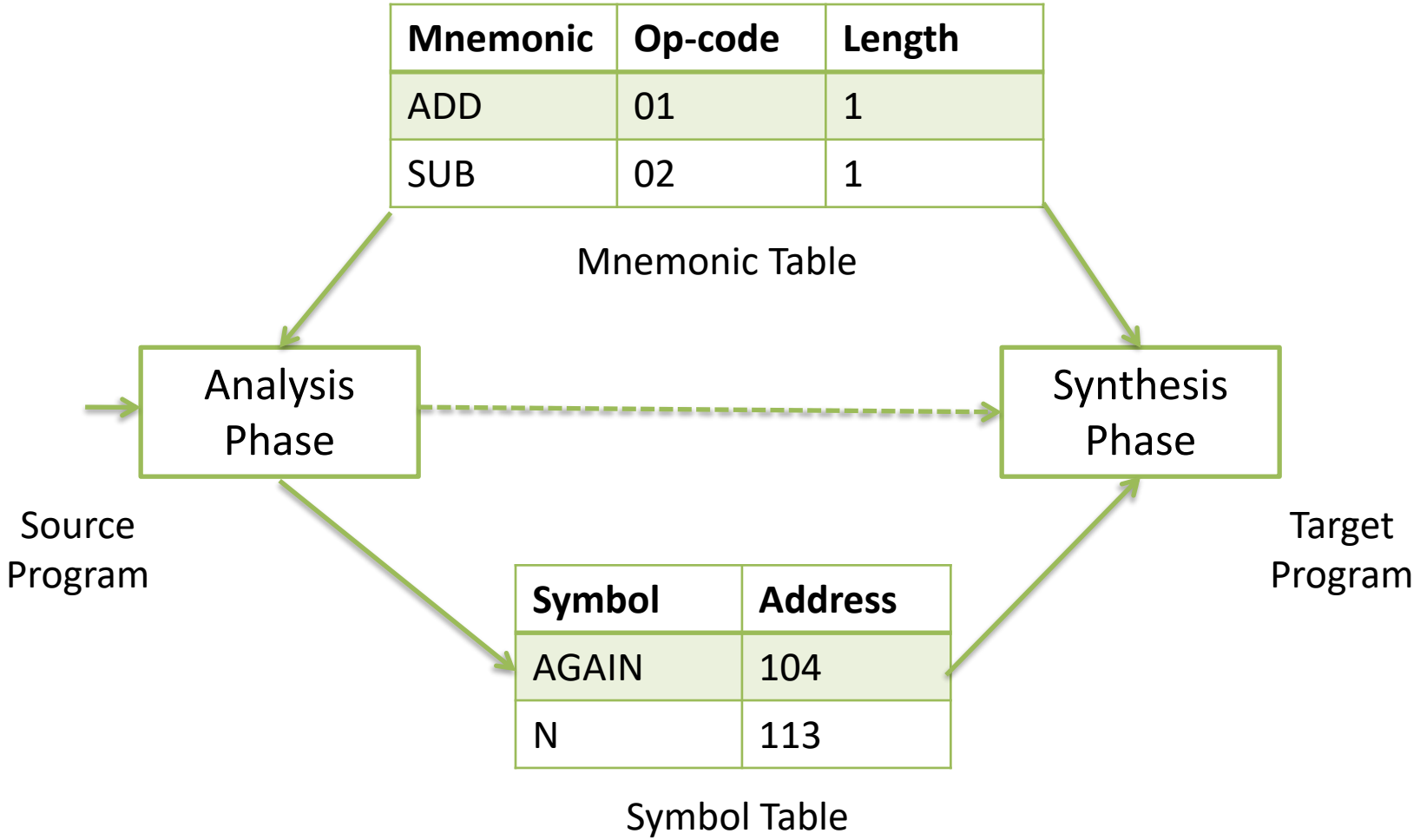
Register Operand

Instruction Op-code	Register
1	AREG
2	BREG
3	CREG
4	DREG

Simple Program

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

Design Structure of an Assembler



Pass Structure of Assembler

- Generally, there are two types of assemblers
  - Single pass assembler
  - Two pass assembler

Design of A Two-Pass Assembler

- Task performed by the two pass assembler are
  - Pass I:
    - Separate the symbol, mnemonic and operand fields
    - Build the symbol table
    - Generate the Literal Table
  - Pass II:
    - Synthesize the target program

Databases for Pass-I

Instruction Op-code	Assembly Mnemonic	Length

Mnemonic Table

Symbol	Address	Length

Symbol Table

Literal	address

Literal Table

Separation of Mnemonic

- Structure for store the mnemonic

```
struct mnemonic
{
    char label[10];
    char opcode[10];
    char operand[10];
};
```

Cont…

- Consider the assembly program

-	**START**	**101**
-	READ	N
-	MOVER	BREG, ONE
AGAIN	MULT	BREG, TERM
-	ADD	CREG, ONE
-	BC	LE, AGAIN
-	MOVEM	BREG, RESULT
-	PRINT	RESULT
N	DS	1
RESULT	DS	1
ONE	DC	‘1’
TERM	DS	1
-	END	-

Cont…

```
f1=fopen("Assembly_Program.txt","r");
fscanf(f1,"%s%s%s", label, opcode, operand);
while(strcmp(opcode, "END") != 0)
{
 printf("\n%s \t%s \t%s", label, opcode, operand);
 fscanf(f1,"%s%s%s", label, opcode, operand);
 }
```

Symbol Table Generation

- Consider the simple program

-	START	101	
-	READ	N	101
-	MOVER	BREG, ONE	102
-	MOVEM	BREG, TERM	103
AGAIN	MULT	BREG, TERM	104
-	MOVER	CREG, TERM	105
-	ADD	CREG, ONE	106
-	COMP	CREG, N	107
-	BC	LE, AGAIN	108
-	MOVEM	BREG, RESULT	109
-	PRINT	RESULT	110
-	STOP	-	111
N	DS	1	112
RESULT	DS	1	113
ONE	DC	'1'	114
TERM	DS	1	115
-	END	-	

Symbol	Address
AGAIN	104
N	113
RESULT	114
ONE	115
TERM	116

Cont…

- To determine the address of symbols, first fix the address of all program element. This is called as memory allocation.
- The memory allocation is done by the help of LC
- LC always contains the address of the next memory word
- LC is initialize with the START statement
- LC is updated by the length of the instruction
- The Length of the instruction is obtained from the mnemonic table.

Cont…

- The structure for symbol table is

Symbol	Address

```
struct Symbol_Table
{
    char symbol[10];
    int address;
};
```

Cont…

```
while(strcmp(opcode, "END") != 0)
{
    if(strcmp(opcode, "START") ==0)
    {
        if(strcmp(operand, “- ") != 0)
            LC = atoi(ASS[k].operand);      //Initialization of LC
        else
            LC = 0;             //Default initialization of LC
    }
    fscanf(f1,"%s%s%s", label, opcode, operand);
}
```

Cont…

```
while(strcmp(opcode, "END") != 0)
{
    …
    if(strcmp(label, “-") !=0)  //if Label present
    {
        strcpy(ST[j].symbol, label);//Copy label into ST
        ST[j].val = LC;                    //Copy the address into ST
        j++;
    }
    fscanf(f1,"%s%s%s", label, opcode, operand);
}
```

Cont…

- Generally, LC set by some assembler directives
- ORIGIN
  - Syntax: ORIGIN <address spec>
  - Set LC to the address given by <address spec>
- EQU
  - Syntax: <symbol> EQU <address spec>
  - Define the symbol to represent address given by <address spec>
- LTORG
  - It permit a programmer to specify where literals should be placed

Cont…

- Consider the program

	START	**200**	
	MOVER	AREG, =‘5’	200
LOOP	MOVER	AREG, A	201
	…		
	ORIGIN	LOOP+2	
	MULT	CREG,B	203
	…		
A	DS	1	217
BACK	EQU	LOOP+1	
B	DS	1	218
	END		

Symbol	**Address**
LOOP	201
A	217
BACK	202
B	218

Cont…

```
while(strcmp(opcode, "END") != 0)
{
    …
    if(strcmp(opcode, "ORIGIN") == 0)
    {
        LC = atoi(operand);       //set the LC
    }
    if(strcmp(opcode, "EQU") == 0)
    {
        LC = atoi(operand);       //set the LC
    }
    fscanf(f1,"%s%s%s", label, opcode, operand);
}
```

Generation of Literal Table

- Consider the example

	START	**200**	
	MOVER	AREG, =‘5’	200
	MOVEM	AREG, A	201
LOOP	MOVER	AREG, A	202
	MOVER	CREG, B	203
	ADD	CREG,=‘1’	204
	LTORG		
	…		
A	DS	1	217
B	DS	1	218
	END		

Literal	**Address**
=‘5’	205
=‘1’	206

Cont…

- The structure for the Literal Table is

Literal	address

```
struct Literal_Table
{
    char literal[10];
    int address;
};
```

Cont…

- Initialize the literal table

```
for(i = 0; i<10; i++)
        LT[i].val = -1;
```

Cont…

```
while(strcmp(opcode, "END") != 0)
{
        …
        while(operand[i] != '\0')
        {
                if(operand[i] == '=')
                {
                        j = 0;
                        while(operand[i] != '\0')
                                LT[ltc].literal[j++] = operand[i++]; //copy the literals
                        LT[ltc].literal[j] = '\0';
                        ltc++;
                }
                else
                        i++;
        }
        fscanf(f1,"%s%s%s", label, opcode, operand);
}
```

Cont…

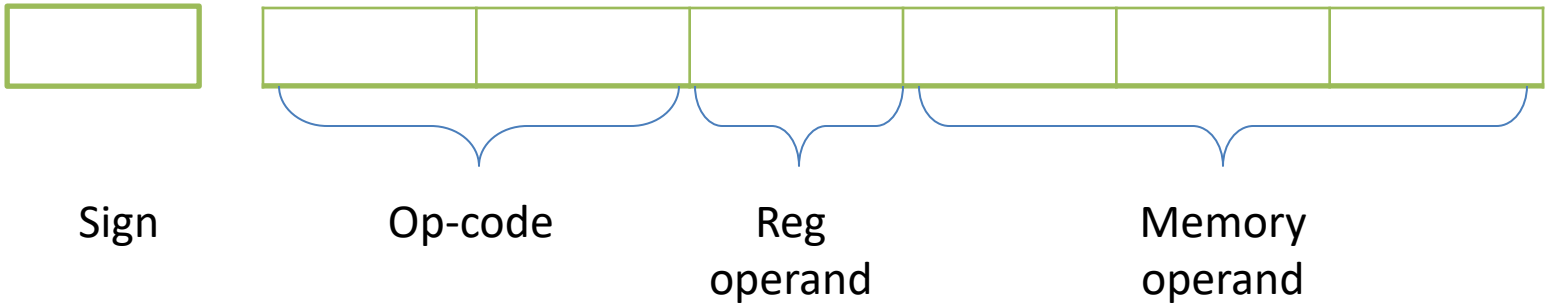
```
while(strcmp(opcode, "END") != 0)
{
    …
    if(strcmp(opcode, "LTORG") == 0)
    {
        for(i = 0; i<ltc; i++)
        {
            LT[i].val = LC; //copy the literal address
            LC++;
        }
    }
    fscanf(f1,"%s%s%s", label, opcode, operand);
}
```

Cont…

```
while(strcmp(opcode, "END") != 0)
{
    …
    fscanf(f1,"%s%s%s", label, opcode, operand);
}
for(i = 0; i<ltc; i++)
{
    if(LT[i].val == -1)
    {
        LT[i].val = LC; //copy the literal address
        LC++;
    }
}
```

Pass II of an Assembler

- Pass II of an assembler generates machine code in the format



Cont…

- Pass II of an assembler uses
  - Machine operation table
  - Symbol table
  - Literal table

Algorithm

- Read symbol table, Literal table and machine operation table.
- Read assembly language program
- Check whether opcode is not equal to end.
  - Search machine operation table for opcode and use its machine code
  - Check whether the operand is literal, if true, search literal table and use corresponding address.
  - Check the operand is symbol, if true, search symbol table and use corresponding address.
- stop the process.