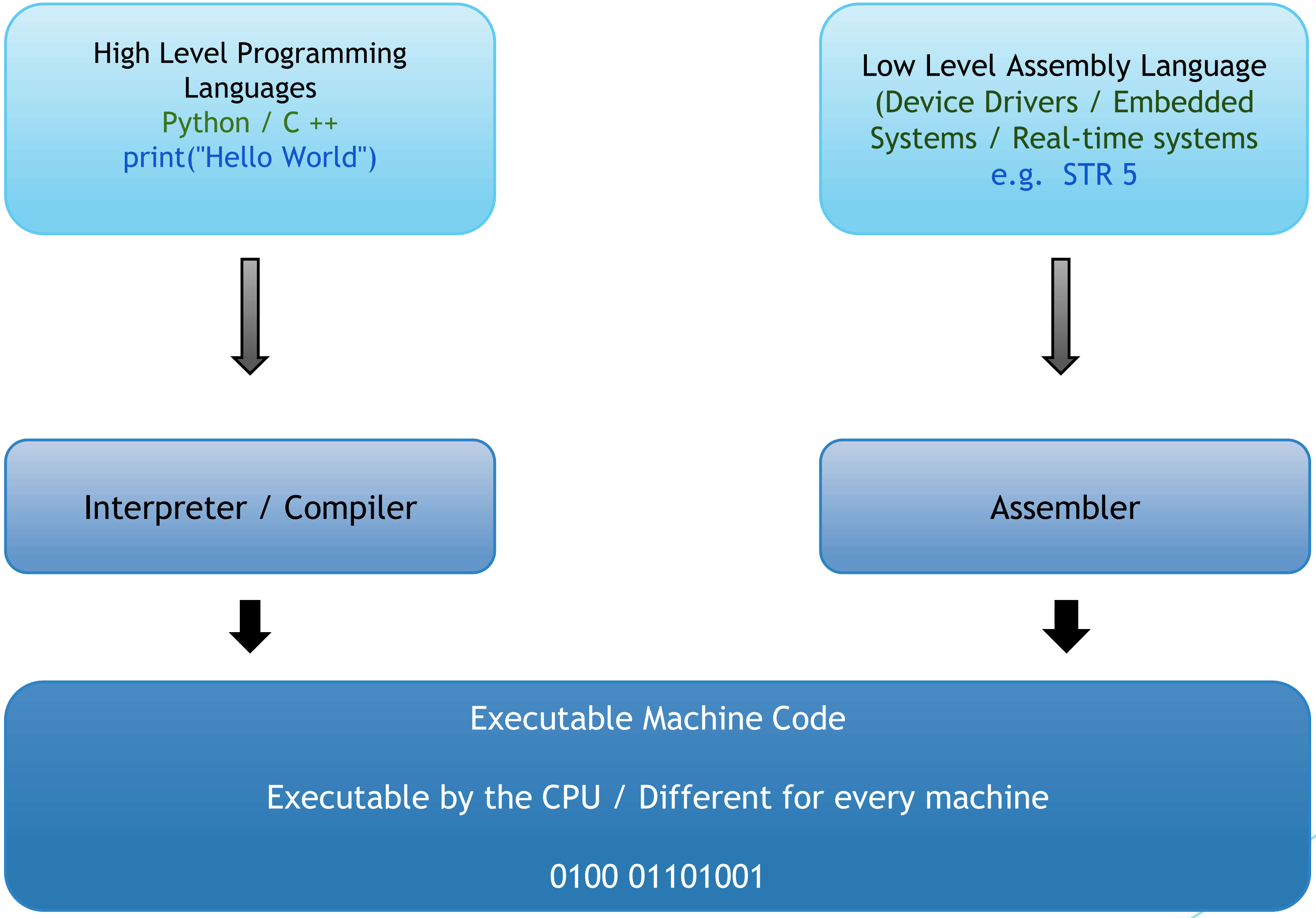
2 Pass Assemblers

CS 348
Implementation of Programming Languages Lab
Department of CSE
IIT Guwahati



High Level Programming Languages
Python / C ++
print("Hello World")

Low Level Assembly Language
(Device Drivers / Embedded Systems / Real-time systems
e.g. STR 5

Interpreter / Compiler

Assembler

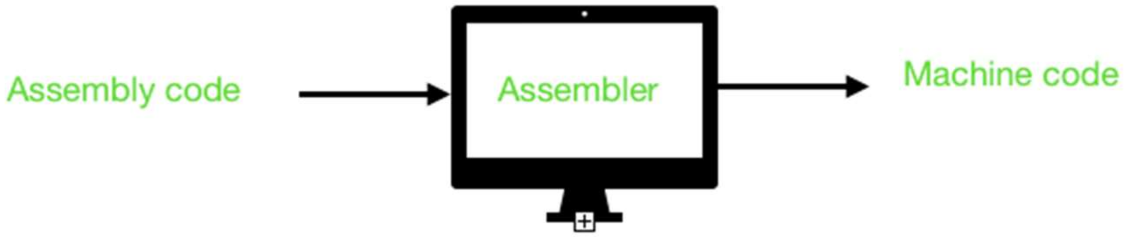
Executable Machine Code

Executable by the CPU / Different for every machine

0100 01101001

Introduction

- **Assembler** is a program for converting instructions written in low-level assembly code into relocatable object code and generating information for the loader.
- It generates machine-level instructions by evaluating the mnemonics (symbols) in the operation field and finds the value of symbols and literals to produce object code.



Assembler Directives

- Assembler Directives are Pseudo-Instructions. They are not translated into machine instructions. They provide instructions to the assembler.
- Basic assembler directives:
  - START
  - END
  - BYTE
  - WORD
  - RESB
  - RESW

Symbols, Literals, Opcodes and Operands

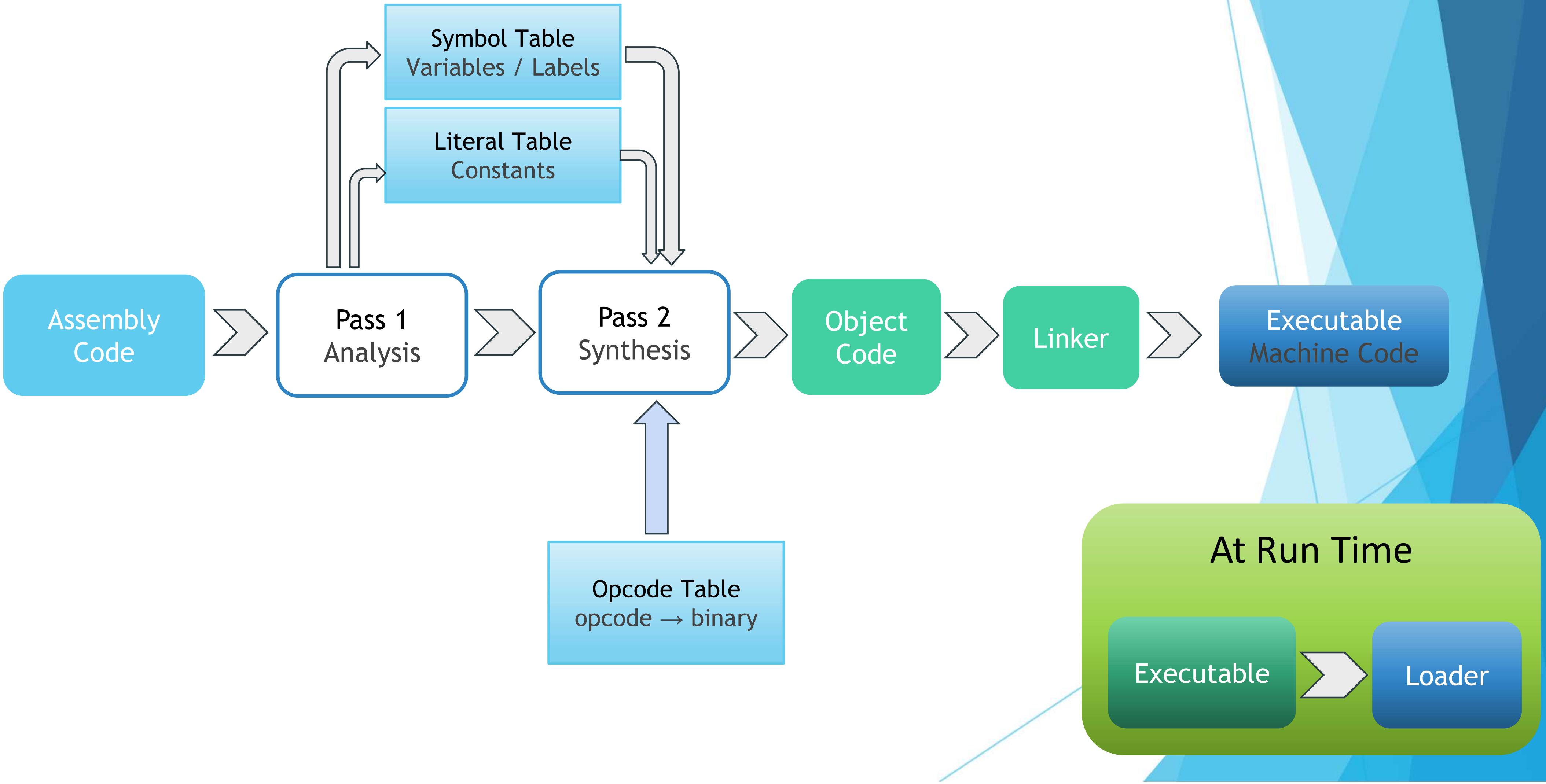
- **Symbols:** A symbol is a single character or combination of characters used as a label or operand.

-Symbols may consist of numeric digits, underscores, periods, uppercase or lowercase letters, or any combination of these.

-The symbol cannot contain any blanks or special characters, and cannot begin with a digit. Uppercase and lowercase letters are distinct.

- **Literals:** Constants. Assembly language source code can contain numeric, string, Boolean, and single character literals.
- **Opcodes and Operands:** The opcode is the instruction that is executed by the CPU and the operand is the data or memory location used to execute that instruction.

2 Pass Assembler Process



Pass 1 - Analysis Phase

This part scans the program looking for symbols, labels, variables, etc, and organises them in tables

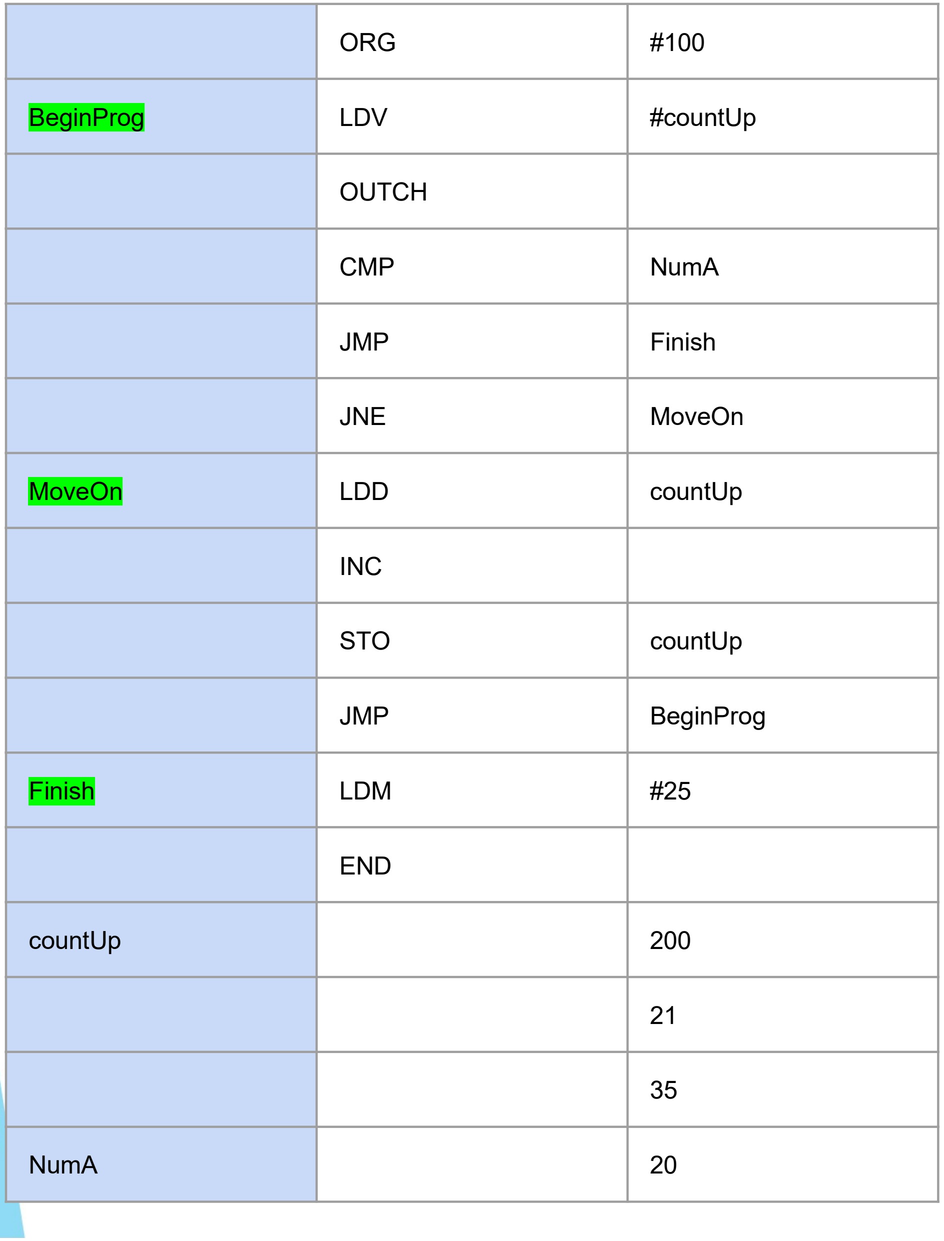
- Passes through the instructions in sequence, looking for symbol addresses
- Create a symbol and literal table
- Keep track of the location counter
- Process Pseudo operations (macros / directives)
- Error Checking

Tokenization:

Read the input one ASCII char at a time; i.e. as a stream of char. The first step is to group characters into meaningful tokens.

Macros
Names, subroutines can can be used more than once. Designed to make programming easier and more module

Directives
Configuration instructions for assembler (such as memory allocation)
Not a program instruction itself.



	ORG	#100
BeginProg	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Labels

The programmer uses these to refer to specific lines in the code rather than to refer to them by a line number.

This makes the program easier to read for humans, allowing the code to be broken down into sections.

	ORG	#100
BeginProg	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Labels

The programmer uses these to refer to specific lines in the code rather than to refer to them by a line number.

This makes the program easier to read for humans, allowing the code to be broken down into sections.

Pass 1 Analysis walkthrough

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	
NumA	

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	
NumA	
Finish	

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	
NumA	
Finish	
MoveOn	

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	
NumA	
Finish	
MoveOn	105

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	
NumA	
Finish	109
MoveOn	105

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	111
NumA	
Finish	109
MoveOn	105

	ORG	#100
BeginProg:	LDV	#countUp
	OUTCH	
	CMP	NumA
	JMP	Finish
	JNE	MoveOn
MoveOn:	LDD	countUp
	INC	
	STO	countUp
	JMP	BeginProg
Finish:	LDM	#25
	END	
countUp		200
		21
		35
NumA		20

Symbol Table	
BeginProg	100
countUp	111
NumA	114
Finish	109
MoveOn	105

SYMTAB

- **Content:**

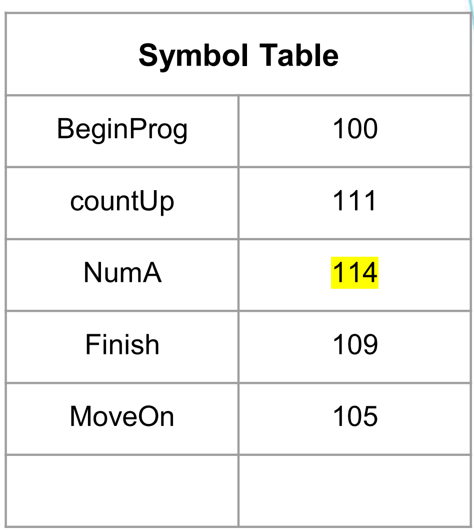
  Label name, value, flag, (type, length) etc.

- **Characteristic:**

  Dynamic table (insert, delete, search)

- **Implementation:**

  Hash table



Symbol Table	
BeginProg	100
countUp	111
NumA	114
Finish	109
MoveOn	105

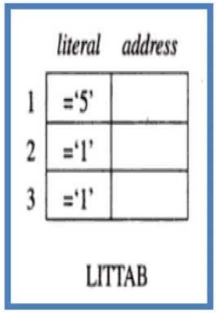
LITTAB

Contents: Literal name, the operand value, and length, the address assigned to the operand

- **How to build?**

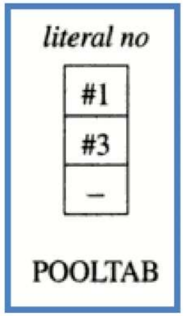
Build LITTAB with literal name, operand value, and length, leaving the address unassigned

When an LTORG statement is encountered, assign an address to each literal not yet assigned an address.



	literal	address
1	=‘5’	
2	=‘1’	
3	=‘1’	

LITTAB



literal no
#1
#3
–

POOLTAB

OPTAB

- **Content:**

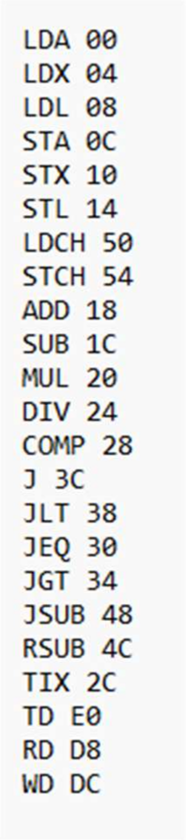
Mnemonic, machine code (instruction format, length) etc.

- **Characteristic:**

Static Table

- **Implementation:**

Array or hash table, easy for search



```
LDA 00
LDX 04
LDL 08
STA 0C
STX 10
STL 14
LDCH 50
STCH 54
ADD 18
SUB 1C
MUL 20
DIV 24
COMP 28
J 3C
JLT 38
JEQ 30
JGT 34
JSUB 48
RSUB 4C
TIX 2C
TD E0
RD D8
WD DC
```

Possible Errors in Pass 1

- Duplicate label
- Invalid operand
- Unrecognized entry in opcode field etc.

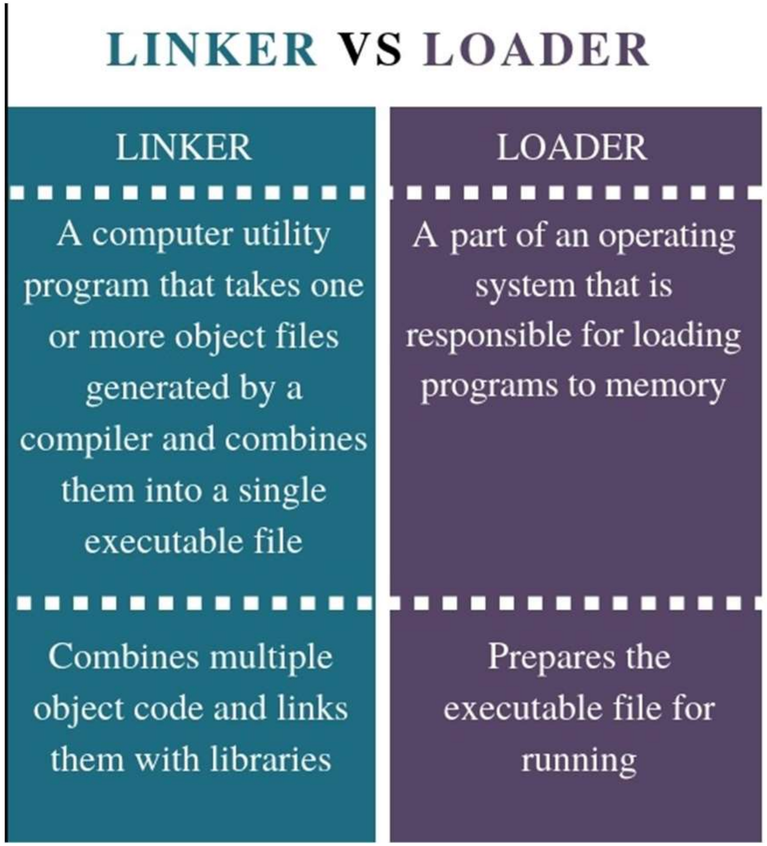
Pass 2 - Synthesis Phase

If no errors are found in pass one, then the second pass assembles the code into object code.

This process often includes the following:

- Symbolic addresses are replaced with relative addresses
- Symbolic opcodes are replaced with binary opcodes

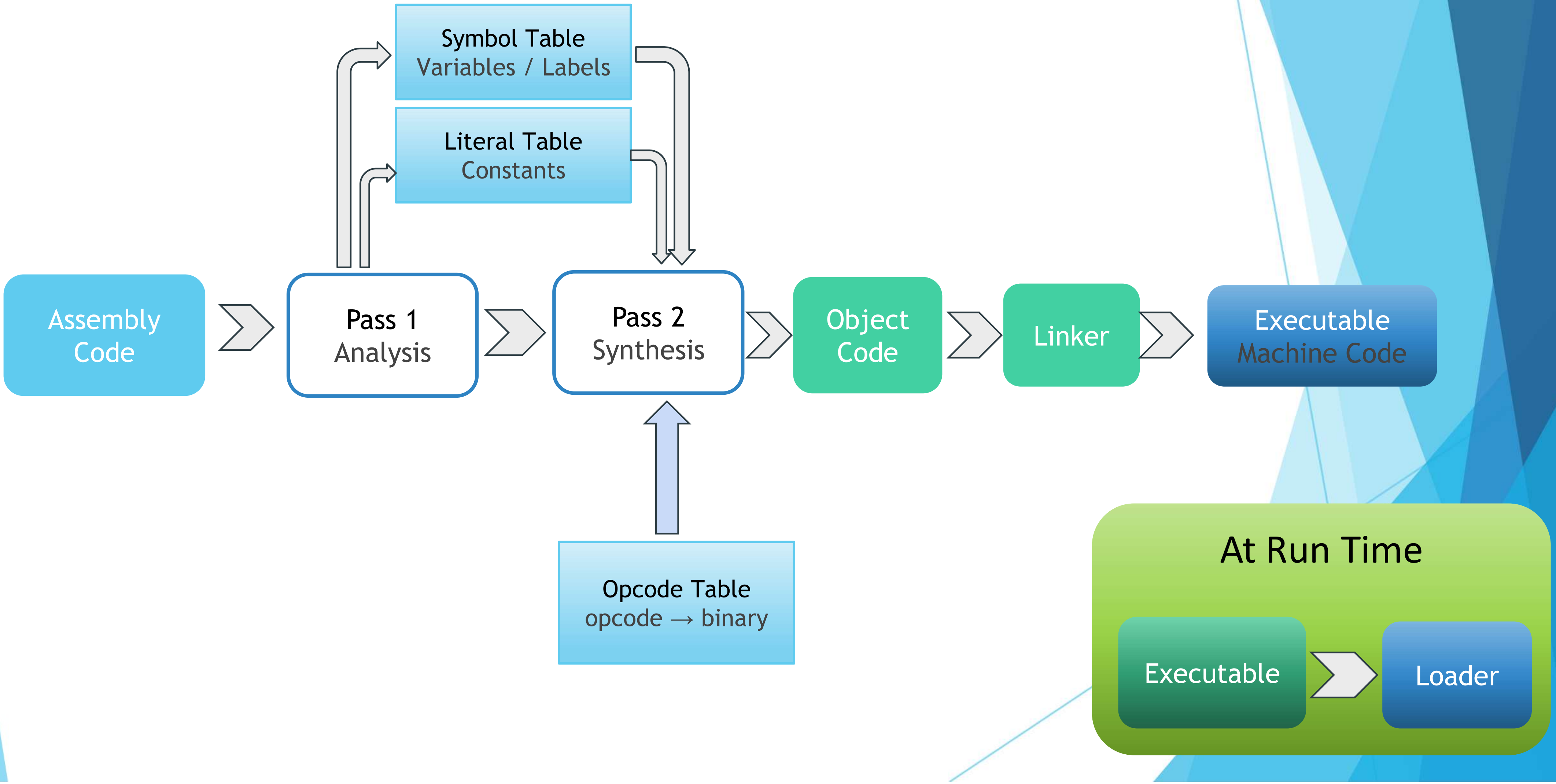
Linker vs Loader



LINKER VS LOADER

LINKER	LOADER
A computer utility program that takes one or more object files generated by a compiler and combines them into a single executable file	A part of an operating system that is responsible for loading programs to memory
Combines multiple object code and links them with libraries	Prepares the executable file for running

2 Pass Assembler Process



Symbol Table
Variables / Labels

Literal Table
Constants

Assembly Code

Pass 1
Analysis

Pass 2
Synthesis

Object Code

Linker

Executable
Machine Code

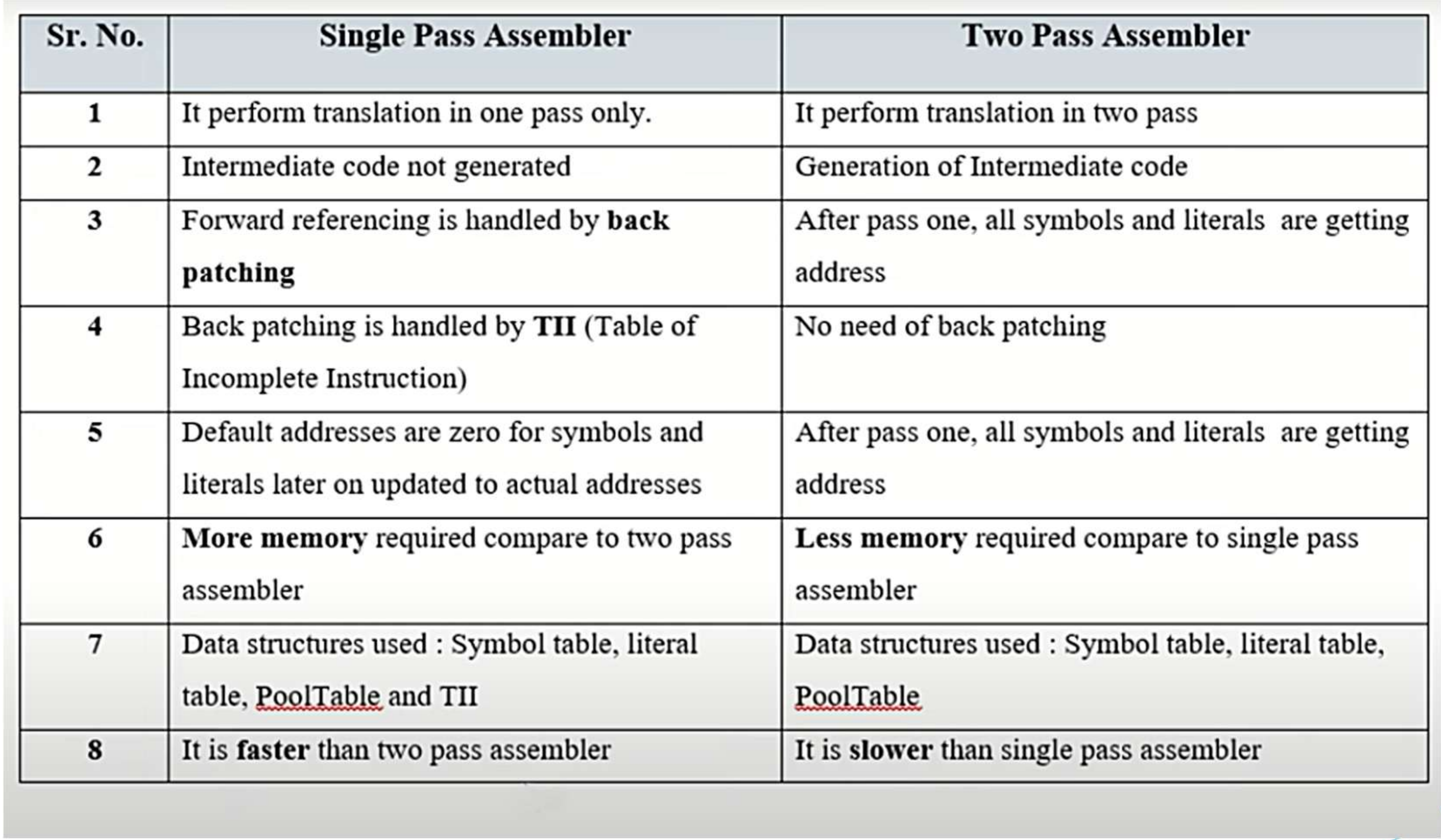
Opcode Table
opcode → binary

At Run Time

Executable

Loader

One Pass vs Two Pass Assembler



Sr. No.	Single Pass Assembler	Two Pass Assembler
1	It perform translation in one pass only.	It perform translation in two pass
2	Intermediate code not generated	Generation of Intermediate code
3	Forward referencing is handled by **back patching**	After pass one, all symbols and literals are getting address
4	Back patching is handled by **TII** (Table of Incomplete Instruction)	No need of back patching
5	Default addresses are zero for symbols and literals later on updated to actual addresses	After pass one, all symbols and literals are getting address
6	**More memory** required compare to two pass assembler	**Less memory** required compare to single pass assembler
7	Data structures used : Symbol table, literal table, PoolTable and TII	Data structures used : Symbol table, literal table, PoolTable
8	It is **faster** than two pass assembler	It is **slower** than single pass assembler

Thank you!