CMPE 220

Class 4 – Assembly Language & Assemblers

RISC versus CISC

Reduced Instruction Set Computer

- One clock-cycle per instruction
- Effective *pipelining*
- Fewer addressing modes
- Requires more instructions per program (more RAM)
- Lower gate count
- Lower energy use

Complex Instruction Set Computer

- Multiple / variable clock-cycles per instruction
- More addressing modes
- Requires fewer instructions per program (less RAM)
- Higher gate count – more chip real estate
- Higher energy use

CISC Computers are Often Microprogrammed

- Machine instruction set: CISC
- Micro-machine instruction set: RISC
- From the standpoint of a compiler, and assembler, or a programmer, I would call the a CISC machine

Week 3: What is an Assembler?

- An *assembler* is a program that converts “assembly language” source code into binary instructions (aka *machine code* or *machine instructions*)

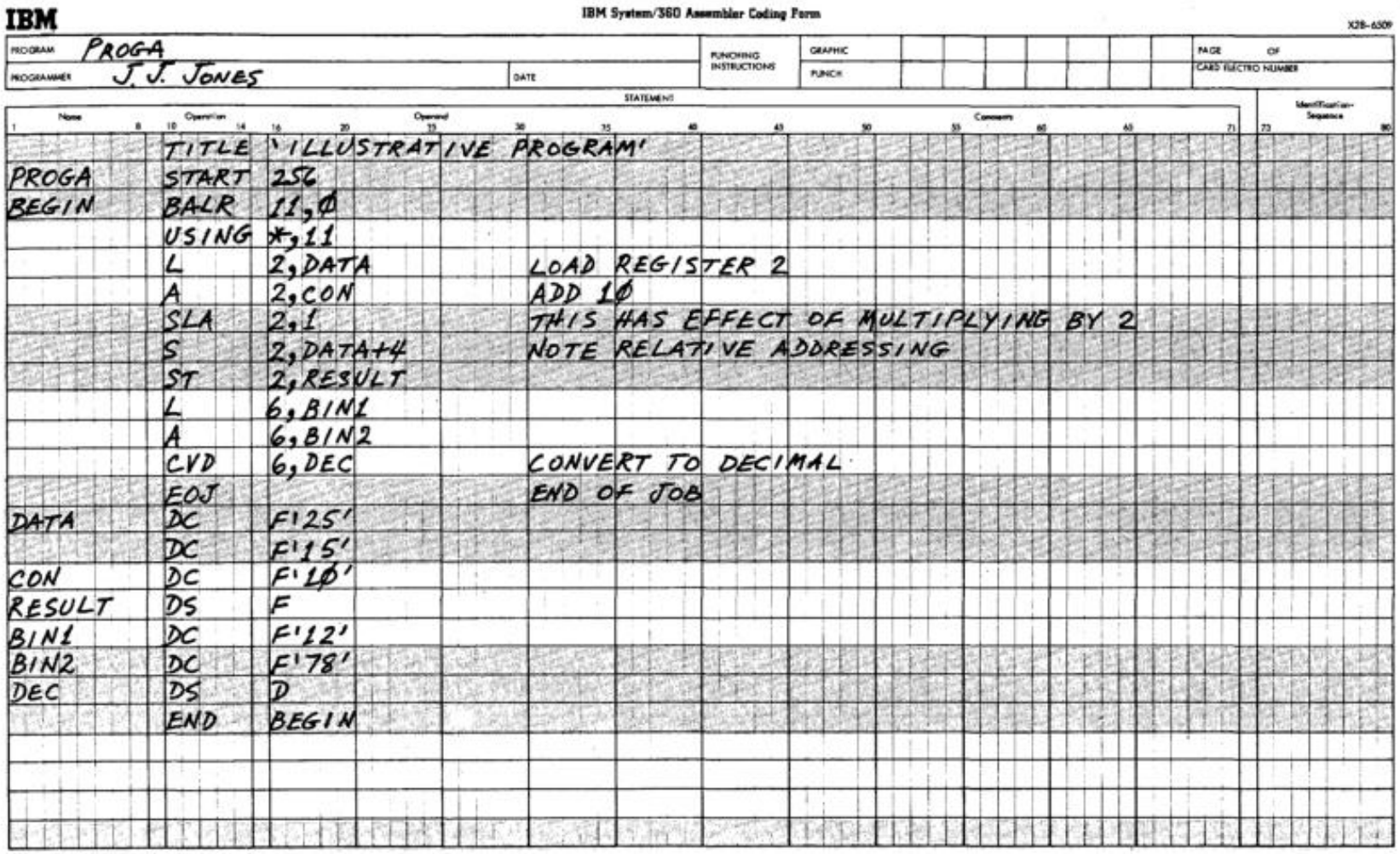
Machine Code – Common in 1940s

Instruction	Action
0101 1111 1111 0001	Load the value from the following address into the A register; advance the program counter by 2
0011 1110 1000 0101	(data address)
0111 1111 1111 0001	Subtract the following value from the A register; advance the program counter by 2
0000 0000 0000 1100	(value = 12)
0110 1111 1111 0001	Store the value from the A register into the following address; advance the program counter by 2
0011 1110 1000 0101	(data address)
0110 1010 1111 0001	Compare the following value to the A register; if A is less than or equal to the value, jump to address
0000 0000 0001 0100	(value = 20)
0100 1000 1000 0110	(program address)

Assembly Language

Instruction	Action
LDA inventory	Load the value from the specified location into the A register
SBA 12	Subtract a value (12) from the A register
STA inventory	Store the value from the A register into the specified location
CMPA 20, low_inventory	Compare the A register to a value (20); if A <= 20, go to the address “low_inventory”
• • •	
low_inventory:	

Assembly Language Coding Sheet



IBM

IBM System/360 Assembler Coding Form

X28-6509

PROGRAM: PROGA

PROGRAMMER: J.J. JONES

DATE:

PUNCHING INSTRUCTIONS:

GRAPHIC:

PUNCH:

PAGE OF

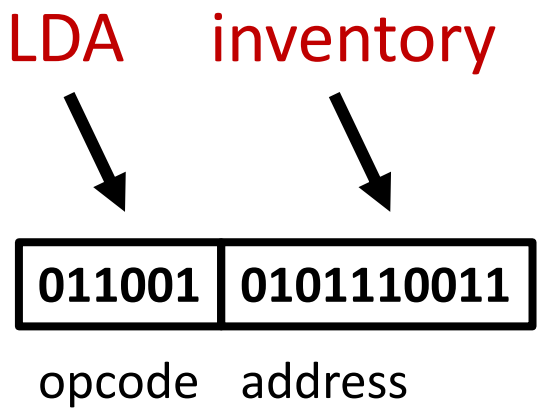
CARD ELECTRO NUMBER

STATEMENT

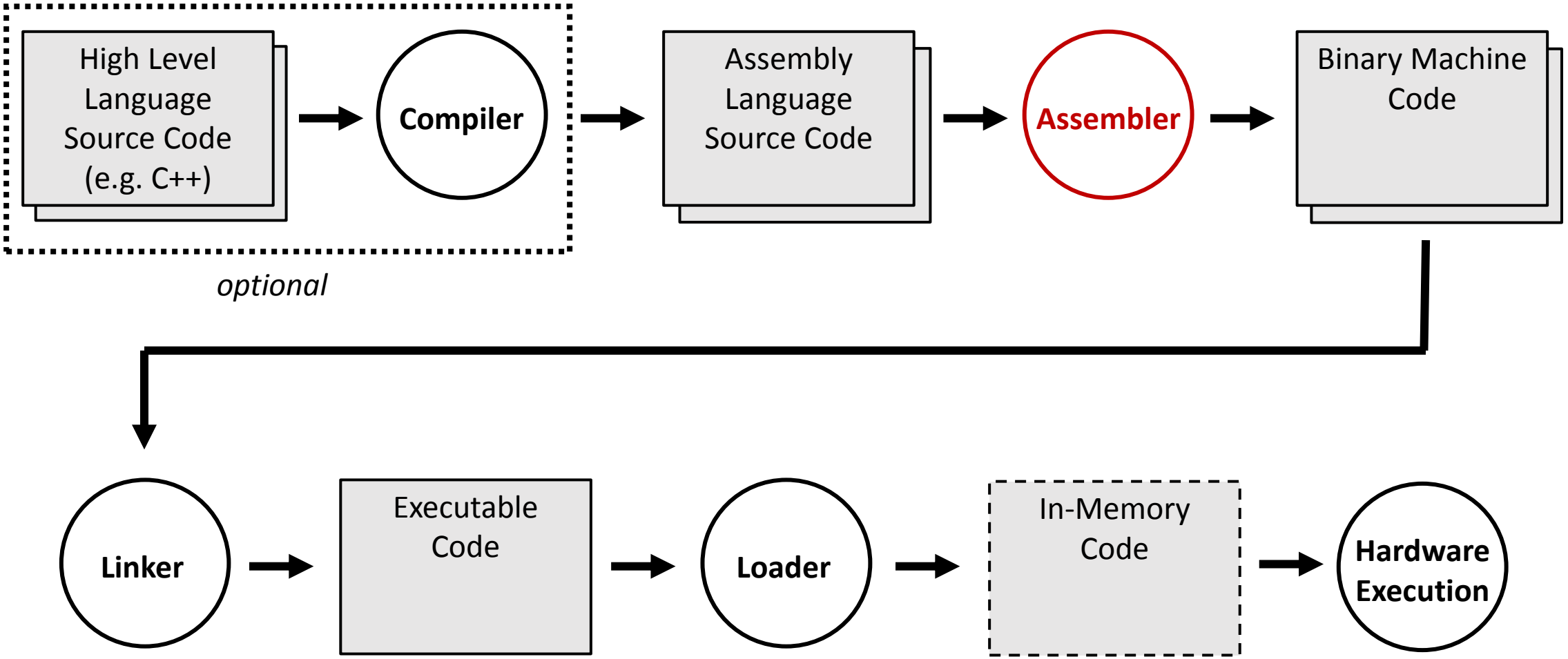
Name	Operation	Operand	Comments	Identification-Sequence
	TITLE	'ILLUSTRATIVE PROGRAM'		
PROGA	START	256		
BEGIN	BALR	11,0		
	USING	*,11		
	L	2,DATA	LOAD REGISTER 2	
	A	2,CON	ADD 10	
	SLA	2,1	THIS HAS EFFECT OF MULTIPLYING BY 2	
	S	2,DATA+4	NOTE RELATIVE ADDRESSING	
	ST	2,RESULT		
	L	6,BIN1		
	A	6,BIN2		
	CVD	6,DEC	CONVERT TO DECIMAL	
	EOJ		END OF JOB	
DATA	DC	F'25'		
	DC	F'15'		
CON	DC	F'10'		
RESULT	DS	F		
BIN1	DC	F'12'		
BIN2	DC	F'78'		
DEC	DS	D		
	END	BEGIN		

Why is it Called an “Assembler?”

- Because it “assembles” machine code instructions!



Building Software



Wait a Minute!

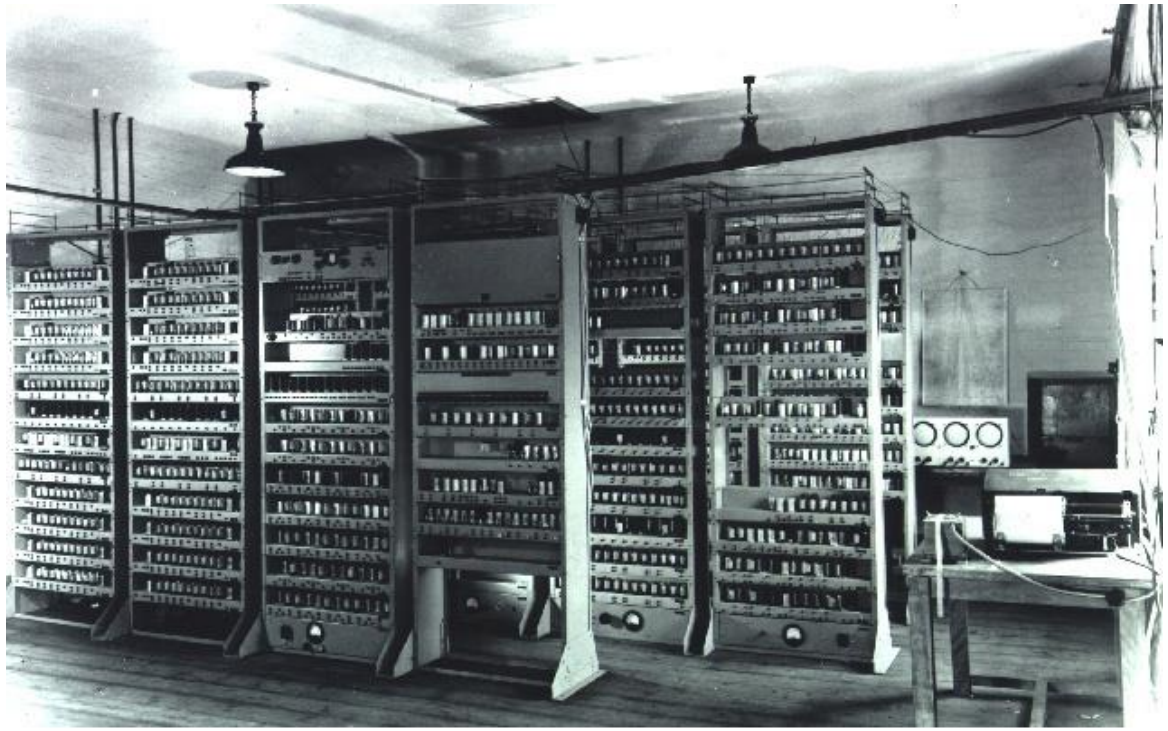
- When you build a program, you don't go through all those steps!
  gcc –o program program_source.c
  ./program
- Modern compiler commands "hide" many of the steps… but you still have the option of breaking out the steps, as we saw in the *makefile* examples:
  %.o: %.c $(DEPS)
  $(CC) -c -o $@ $< $(CFLAGS)
- To compile a C/C++ to assembly language:
  gcc -S -o my_asm_output.s helloworld.c

Types of Assemblers (nomenclature)

- **Assembler:** converts assembly language to binary machine code
- **Macro Assembler:** allows the programmer to define new instructions that the assembler "expands" into the actual instruction set
  This does not create new machine instructions
- **High Level Assembler:** an assembler that includes certain high-level statements – such as IF/THEN/ELSE statements or loops – that don't correspond direction to machine instructions
- **Cross Assembler:** an assembler that runs on one machine, but generates binary machine code for a different machine
- **Micro Assembler:** converts microassembly source code into *microcode*… the low level code that implements the machine instruction set

A (small) Bit of History

- *Kathleen Booth* is credited with creating the first assembler in 1947, while working on the ARC2 (Automatic Relay Calculator) computer at the University of London.
- *David Wheeler* independently developed an assembler for the EDSAC (Electronic Delay Storage Automatic Calculator) in 1948.
  - The IEEE credits Wheeler with creating the first assembler.



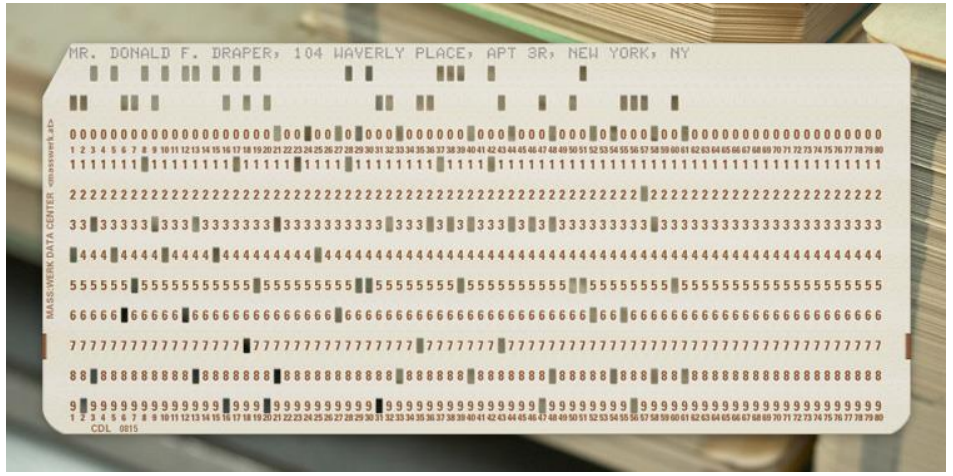
Electronic Delay Storage Automatic Calculator

Requirements for the first Assemblers

- They needed a way to enter mnemonic instructions, so that programmers didn't need to remember binary opcodes and instruction formats.
- They needed a way to associate mnemonic labels with memory addresses.
- But: they needed to be very simple! The first assemblers were written in binary machine code, and ran on computers with *tiny* amounts of memory.

Requirements for an Assembler (continued)

- They could use *characters* to represent assembly language instructions.
- Instructions could be entered with punched cards.
  - Punched cards had been in use since the 1890s.
  - Businesses were comfortable working with punched cards and keypunches.
  - Card readers were easily adapted for use with computers



Assembly Language Instruction Formats

- The requirements led to a very simply format, that is still in use today.

updateinventory	*LDA*	*inventory*
Label (optional)	opcode	Address or register labels (optional)

Non-Instruction Statements (SIC)

- So far, we've talked about statements that correspond one-to-one to machine code instructions… but there is another statement type required: the *memory declaration*.
- Reserve some memory, and assign a label:

```
inventory       RESW    5     (Reserve 5 words)
partnumbers     RESB    100   (Reserve 100 bytes)
```

- Reserve some memory, and assign a label *and starting value*:

```
inventory       WORD    100   (Reserve 1 word; value=100)
partname        BYTE    C'widget'  (Reserve 6 bytes; value='widget')
Iochannel       BYTE    X'05'      (Reserve 1 byte; value=x05)
```

Other Housekeeping Statements

- Indicated starting address of program:
  programname START 1000
- Indicate end of program, and location of first statement:
  END starthere

What an Assembler Does

- Convert mnemonic opcodes to machine language code
- Convert symbolic references to memory addresses
- Assemble machine code instructions
- Write a binary machine code file

Two-Pass Assembler

- 1st Pass
  - Identify statements
  - Determine memory layout
  - Assign addresses to symbolic references and build “symbol table”
- 2nd Pass
  - Assemble instructions
  - Output binary machine code
  - Print program listing & address assignments (Symbol Table)

Assembly Example: SIC/XE

Line #	Label	Instruction	Argument	*Address*	*Instruction Size**
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*3*
3		LDT	Sales	*1003*	*3*
4		SUBR	T, A	*1006*	*2*
5		J	Somewhereelse	*1008*	*3*
6	Partnumber	BYTE	C'005740'	*1011*	*6*
7	Inventory	WORD	500	*1017*	*3*
8	Sales	WORD	27	*1020*	*3*

- Note that the SIC/XE has variable length instructions. This is often true of CISC machines. RISC machines have uniform length instructions.

1st Pass: Build Symbol Table

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*3*
3		LDT	Sales	*1003*	*3*
4		SUBR	T, A	*1006*	*2*
5		J	Somewhereelse	*1008*	*3*
6	Partnumber	BYTE	C'005740'	*1011*	*6*
7	Inventory	WORD	500	*1017*	*3*
8	Sales	WORD	27	*1020*	*3*

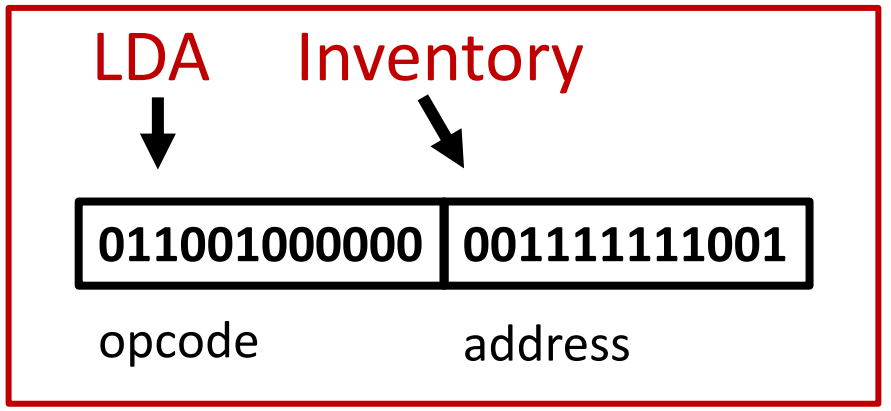
- The Symbol Table contains the *addresses* of the symbols, not the values stored at those locations.

Symbol	Address
Program	1000
Partnumber	1011
Inventory	1017
Sales	1020

2nd Pass: Assemble Machine Instructions

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*3*
3		LDT	Sales	*1003*	*3*
4		SUBR	T, A	*1006*	*2*
5		J	Somewhereelse	*1008*	*3*
6	Partnumber	BYTE	C'005740'	*1011*	*6*
7	Inventory	WORD	500	*1017*	*3*
8	Sales	WORD	27	*1020*	*3*

Symbol	Address
Program	1000
Partnumber	1011
Inventory	1017
Sales	1020



Single Pass Assembler

- Uses two tables: a Symbol Table and a Reference Table
- When an undefined symbol is encountered, it’s added to the Reference Table
- When the symbol is defined, its address is placed in Symbol Table, and all locations that *reference* the symbol are updated.

Single Pass Example

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*3*
3		LDT	Sales	*1003*	*3*
4		SUBR	T, A	*1006*	*2*
5		J	Somewhereelse	*1008*	*3*
6	Partnumber	BYTE	C’005740’	*1011*	*6*
7	Inventory	WORD	500	*1017*	*3*
8	Sales	WORD	27	*1020*	*3*

Symbol	Reference
Inventory	1000
Sales	1003
Somewhereelse	1008

Symbol	Address
Program	1000
Partnumber	1011
Inventory	1017
Sales	1020

SIC/XE – Special Hardware Cases

- Some Instructions (e.g. LDA) may be *3-byte* or *4-byte*

- 24-bit
- **disp:** 12-bit address displacement

- 32-bit
- **address:** 20-bit address

2nd Pass Must *Update* Instructions & Tables

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*3*
3		LDT	Sales	*1003*	*3*
4		SUBR	T, A	*1006*	*2*
• • •		• • •			
6	Partnumber	BYTE	C'005740'	*27011*	*6*
7	Inventory	WORD	500	*27017*	*3*
8	Sales	WORD	27	*27020*	*3*

Symbol	Reference
Inventory	1000
Sales	1003
Somewhereelse	1008

Symbol	Address
Program	1000
Partnumber	27011
Inventory	27017
Sales	27020

2nd Pass Must *Update* Instructions & Tables

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*4*
3		LDT	Sales	*1004*	*3*
4		SUBR	T, A	*1007*	*2*
• • •		• • •			
6	Partnumber	BYTE	C'005740'	*27012*	*6*
7	Inventory	WORD	500	*27018*	*3*
8	Sales	WORD	27	*27021*	*3*

Symbol	Reference
Inventory	1000
Sales	1004
Somewhereelse	1009

Symbol	Address
Program	1000
Partnumber	27012
Inventory	27018
Sales	27021

2nd Pass Must *Update* Instructions & Tables

Line #	Label	Instruction	Argument	*Address*	*Instruction Size*
1	Program	START	1000	*1000*	*0*
2		LDA	Inventory	*1000*	*4*
3		LDT	Sales	*1004*	*4*
4		SUBR	T, A	*1008*	*2*
• • •		• • •			
6	Partnumber	BYTE	C'005740'	*27013*	*6*
7	Inventory	WORD	500	*27019*	*3*
8	Sales	WORD	27	*27022*	*3*

Symbol	Reference
Inventory	1000
Sales	1004
Somewhereelse	1010

Symbol	Address
Program	1000
Partnumber	27013
Inventory	27019
Sales	27022

Programming an Assembler

- **Symbol Table:** built by 1st Pass; matches symbols to addresses
- **Reference Table:** built be 1st pass; tracks references to symbols
- **Scanner / Tokenizer:** scans each line looking for tokens delimited by space characters:
  label (optional) – instruction – address or register (optional) – comment?
- **Instruction lookup:** searches a pre-defined table for the matching instruction string. The table contains:
  - Opcode
  - Instruction length
  - Instruction type
- **Assembly routines:** a subroutine to assemble each instruction format.

A Bit More History: The First Assemblers

- In the 40s and 50s, memory was *very* limited. It wasn't possible to store a *two-pass assembler*, the source code of the program being "assembled", and the various data structures in memory.

  Consequently, computer operators would:
  - Load the binary object code for the assembler 1st pass from a card reader.
  - Load the source code of the program being assembled from a card reader.
  - Load the binary object code for the assembler 2nd pass from a card reader.
  - Load the source code of the program being assembled from a card reader.

History (continued)

- With the advent of magnetic disks in the late 50s, operators didn't need to load multiple card decks… the two "passes" of the assembler were stored on disk, along with the source code of the program being assembled.
- Memory was still very limited, and the same sequence of "overlay" steps was still performed… although much faster!

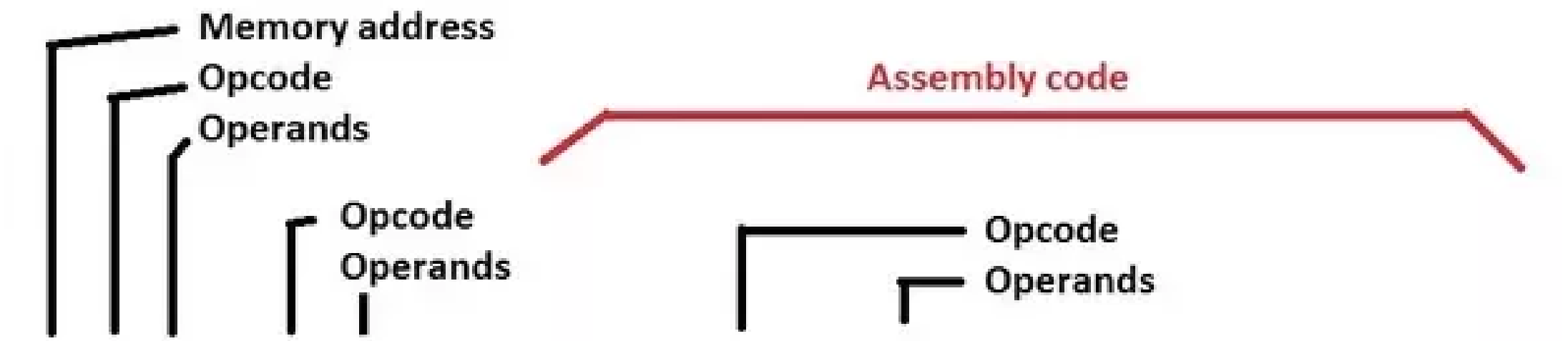
Assembler Output: the Object File

- Not just a binary executable file!
- Specifics vary from system to system.
- Common elements:
  - Header record:
    - Start address
    - Length
  - Reference Table (for linking object files)
  - Symbol Table (for linking object files)

Assembler Output: the Listing

- Line numbers and source code (including comments)
- Binary (octal, hex) instruct codes
- Symbol Table
- Error messages!

Assembler Output: Sample Listing




```
                        ;
AB4D A5 11     LDA $11  TRMNOK: LDA     INPFLG
AB4F F0 11     BEQ $AB62        BEQ     TRMNO1          ;IF INPUT TRY AGAIN.
AB51 30 04     BMI $AB57        BMI     GETDTL
AB53 A0 FF     LDY #$FF         LDYI    255             ;MAKE IT LOOK DIRECT.
AB55 D0 04     BNE $AB5B        BNEA    STCURL          ;ALWAYS GOES.
AB57 A5 3F     LDA $3F  GETDTL: LDWD    DATLIN          ;GET DATA LINE NUMBER.
AB59 A4 40     LDY $40
AB5B 85 39     STA $39  STCURL: STWD    CURLIN          ;MAKE IT CURRENT LINE.
AB5D 84 3A     STY $3A
AB5F 4C 08 AF  JMP $AF08  SNERR4: JMP    SNERR
```

Assembler Error Conditions

- Unrecognized Opcode:
  LDQ
- Missing address:
  LDA
- Missing register(s):
  ADDR
  ADDR S
- Invalid register(s):
  ADDR S, G

Assembler Error Conditions (continued)

- Unknown Address: location referenced but not defined:
  LDA inventroy
- Duplicate label definition:
  inventory WORD 500
  - •
  - •
  - •

  inventory WORD 500
- Location defined but never referenced (warning):
  inventofy WORD 500

What is Pseudocode?

- An informal high-level description of an algorithm
- Typically includes programming language constructs such as IF/THEN/ELSE, DO/WHILE, FOR, etc.
- Typically includes use of variables
- Augmented by natural language (Descriptions)
- May be more or less “formal”
- May resemble a particular programming language

Pseudocode: Scan a Line

- *label opcode argument ;comment*

- Get next line from input file
- $current-char = first character
- IF ($current-char is not a space) THEN
  - Copy characters until blank or end-of-line into $label;
- Skip blanks
- IF ($current-char == end-of-line) THEN
  - GOTO error
- Copy characters until blank or end-of-line into $opcode
- Skip blanks
- IF (($current-char != end-of-line) and ($current-char != ';')) THEN
  - Copy characters until blank or end-of-line into $argument
  - Skip blanks
- IF ($current-char != end-of-line) THEN
  - Copy characters until end-of-line into $comment

Pseudocode: Scan a Line (cont)

- Get next line from input file
- IF (first character is not a space) THEN
  - Copy characters until blank into $label;
- Skip blanks
- IF (end-of-line) THEN
  - GOTO error
- Copy characters until blank into $opcode
- Skip blanks
- IF (not end-of-line) THEN
  - Copy characters until blank into $argument