

Machine & Assembly Language Programming

Considering the next problem in design

- Now that we know how instruction sets may be formulated, and how individual instructions are executed
 - How are programs expressed?
 - Logical sequences of machine coded instructions
 - How is the notion of “high level” programming supported?
 - Assembly language
 - How are “high level” languages translated to machine code?
 - Assembler translator

Goals

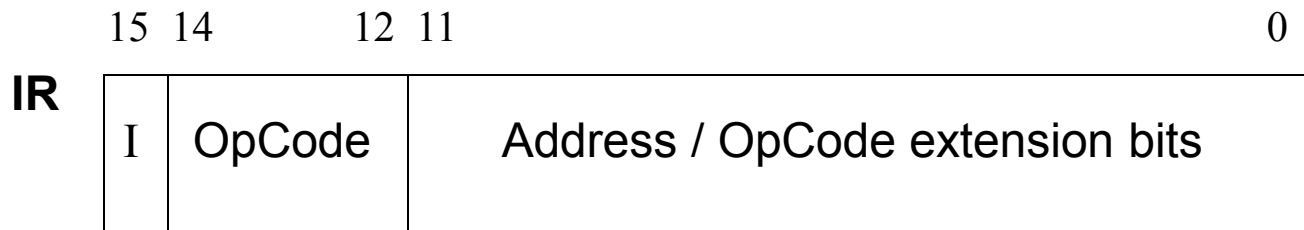
- We conclude our lecture series by considering computer organization
 - Combining the CPU, Memory and Bus architectures together with logic programming
- In this final sequence of lectures slides we discuss some issues related to programming (Mano – Chapter 6)
 - Assembly language
 - Assembler translators
 - Programming examples

Stored programs

- A stored program is a set of instructions and data expressed in binary language, stored in non-volatile (ie. disk storage) memory
- Programs can be executed only from Memory.
 - Thus, a program must be loaded from disk to RAM in order to execute it.
 - Loaded programs are called *processes*.
 - Individual *instructions* must be transferred to the CPU where they are executed, using data that must be obtained from either CPU registers or RAM
- A process is executed by executing each individual instruction that, collectively, fulfill the intentions of the programmer.

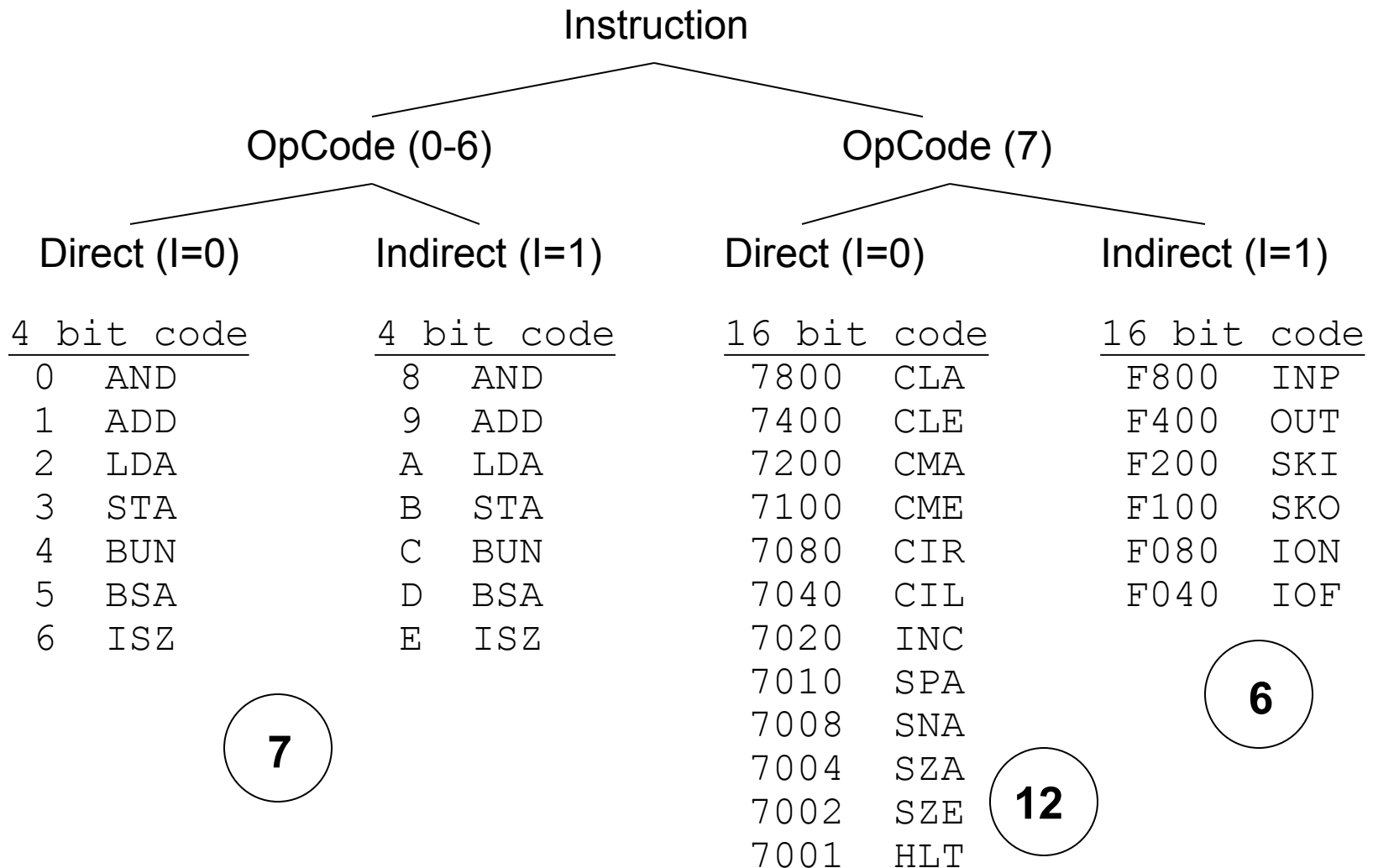
Instruction Hierarchy

- The computer is engineered to support a reasonable number of independent instructions
 - Uses a hierarchical scheme (1+3+12 scheme)
 - OpCode – 3 bits – supports 8 unique codes
 - 0-6 (instructions)
 - 7 (toggle)
 - Mode – 1 bit – supports 2 addressing modes
 - I = 0 Direct addressing
 - I = 1 Indirect addressing
 - Special Case :: OpCode = 7 (toggle) PLUS Mode bit (I)
 - Use 12 bits of Address field to specify **additional** instructions



Instruction Hierarchy

- Mano's instruction set consists of 25 instructions:



Programming the basic computer

- Remind yourself that the only language that the computer logic understands is Machine Language
 - Expressed in binary (voltage values in gates and on wires)
- From the perspective of the machine we have previously constructed, consider the example program:

<u>Location</u>	<u>Instruction Code</u>
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

Although the computer may understand this program encoding, it is very difficult for humans to understand what is intended, or how it is to be carried out.

Programming the basic computer

- The example is so simple that we can replace binary code with symbolic operations

Symbolic Operations makes it easier

Expressing data in machine language format, especially arithmetic data, requires both conceptual understanding and technical skill in *translating* from one representation (eg. decimal) to another (hexadecimal, octal, binary).

Location	Instruction Code	Mnemonic	Decimal Data
0	2004	LDA 004	
1	1005	ADD 005	
2	3006	STA 006	
3	7001	HALT	
4	0053	First op	83
5	FFE9	Second op	-23
6	0000	Store sum	0 (Initial)

Hex/machine Decimal/human

Programming the basic computer

- The previous program demonstrates several important requirements in designing a non-binary oriented programming language suitable for human programmers.
 - Being able to identify each operation by its mnemonic.
 - Being able to represent address locations symbolically
 - To avoid having to carefully and tediously determine actual values of addresses (which may change many times during program creation)
 - To permit limited degrees of program relocatability in memory
 - Being able to define data storage locations as named variables
 - With limited *data type* attribution
 - With option to **assign initial values** in various formats

<u>Location</u>	<u>Instruction Code</u>
-----------------	-------------------------

	ORG 0 /Origin of program at location 0
--	---

	LDA A /Load data at A into AC
--	------------------------------------

	ADD B /Add data at B to AC
--	---------------------------------

	STA C /Store AC data at C
--	--------------------------------

	HLT /Halt the computer
--	-----------------------------

A,	DEC 83 /Decimal data (operand) initialized
----	---

B,	DEC -23 /Decimal data initialized
----	--

C,	DEC 0 /Decimal data (initial 0)
----	--------------------------------------

	END /End of symbolic program
--	-----------------------------------

1	1005
---	------

2	
---	--

5	
---	--

6	0000
---	------

ADD	005
-----	-----

STA	006
-----	-----

HALT	
------	--

First op	83
----------	----

Second op	-23
-----------	-----

Store sum	0 (Initial)
-----------	-------------

Assembly
Language
Program

Assembly Language

- Assembly language programs are written using statements with three fields
 - **Label** :: A symbolic address - a string delimited by a comma ‘,’
 - **Instruction** :: either a machine instruction or a pseudo-instruction
 - **Comment** :: Programmer comments not relevant to program logic

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Symbolic Addresses

- Symbolic addresses, or **Labels**,
 - Consist of strings delimited by a comma ‘,’ !
 - Consist of 1, 2 or 3 characters only !
 - May be preceded by white spaces.

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD BB	/Machine instruction, symbol reference
	STA CCC	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
BB,	DEC -23	/Pseudo-instruction with label field
CCC,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Comments

- **Comments**

- **Begin with a slash ‘/’ character !**
- **All characters after the slash are part of the comment**
- **Comments are terminated by the new line character(s) !**
- **May be preceded by white spaces.**

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Instructions (0)

- **There are three types of Instructions**
 - **Memory Reference Instructions (MRI)**
 - **Non-Memory Reference Instructions (non-MRI)**
 - Register reference
 - Input/Output reference
 - **Pseudo-Instructions** (*soo-do*, or *syu-do*) [For Assembler]

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Instructions (1)

- **Memory Reference Instructions (MRI)**

- Have an address operand
 - Symbolic address label
 - Decimal number
- May be followed by a second operand 'I' for indirect addressing
- OpMnemonics and operands are separated by one or more white spaces
- Instructions are terminated by either a comment, or a new line char.

AND, ADD, LDA,
STA, BUN, BSA, ISZ

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Instructions (2)

- **Non-Memory Reference Instructions (non-MRI)**

- Do not have any operands
 - Implicitly specify the register operand(s)
- Two kinds of instructions:
 - Register reference
 - Input/Output reference

CLA, CLE, CMA, CME, CIR, CIL,
INC, SPA, SNA, SZA, SZE, HLT

INP, OUT, SKI, SKO, ION, IOF

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Instructions (3)

- **Pseudo-Instructions**

- Provide directions to the Assembler for translating the program or instruction
- Four instructions:
 - **ORG n** – Set the Assembler memory Location Counter ($LC = n$)
 - **END** – Denote the end of the Assembly Language program
 - **DEC** – Specify a value in decimal notation
 - **HEX** – Specify a value in hexadecimal notation

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-instruction
	LDA A	/Machine instruction, symbol reference
	ADD B	/Machine instruction, symbol reference
	STA C	/Machine instruction, symbol reference
	HLT	/Machine instruction
A,	DEC 83	/Pseudo-instruction with label field
B,	DEC -23	/Pseudo-instruction with label field
C,	DEC 0	/Pseudo-instruction with label field
	END	/Pseudo-instruction

Assembly Language – Instructions (4)

- **Pseudo-Instruction Set Extensions**
 - Provide additional directions to the Assembler for translating the program or instruction
 - Handling and reporting of errors
 - Labels in ORG, END
 - Unrecognized OpMnemonics
 - Unrecognized syntax

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
A,	DEC 83,-23	/Declare, initialize two contiguous words
B,	BSS 10	/Declare 10 contiguous words, set to 0
C,	OCT 137	/Declare Octal type with Octal value
D,	BIN 10011	/Declare Binary type with binary value / padded on left with 0's to fill word

Assembly Language – Instructions (5)

- **Machine Instruction Set Extensions**
 - There is still room to specify up to 6 additional machine instructions
 - 12 MRI, 7 non-MRI, **6 I/O (out of 12 possible codes)**
 - **Suggestions:**
 - **SUB :: Subtraction**
 - **MUL :: Multiplication**
 - **DIV :: Integer division**
 - **MOD :: Modular division (remainder)**
 - **LAD :: Load specified address location to AC**
 - **LDT :: Load Temporary Register (TR) from AC**
 - **CPS :: Copy string from address in TR to specified address location, number of words to move in AC. Both AR and TR are incremented by one for each word copied.**
 - **CMP :: Compare AC with value at specified address, need to create status flags – Z (zero), P (positive), N (negative)**
 - **SHR :: Shift AC right arithmetically**
 - **SHL :: Shift AC left arithmetically**
 - **How does one make a choice ?**

Assembler Translator for Assembly Language

- A program that translates an assembly language program (source code) into a structured set of binary strings representing executable machine code is called an *Assembler*.
 - Assemblers are simplified versions of Compilers, due to the simplified syntax and grammar rules used to define an assembly language
 - All assembly language statements must be well-formed, regular expressions
 - All instruction mnemonics must be built into the Assembler so that it may recognize them and provide the correct translation responses
 - Direct Assembler actions for translation
 - Translate Mnemonics to binary OpCodes

Assembler Translator ... (Continued)

- **Grammar must be specified within Assembler logic**
 - **Unique definitions of Symbolic Address Labels**
 - **Recognition and correct address translation of Symbolic Address Labels used as memory operands**
 - **Recognition and correct data translation of data declarations, both type and value of operand**
 - **Recognition and correct translation of Indirect address mode (I)**
- **Note that correct resolution (ie. translation) of symbolic address label operands may require two sequential passes through the assembly language program**
- **Assemblers are not responsible for**
 - **Specifying the load point address of the executable binary image of the program**
 - **Specifying the initial value of the PC when the program is loaded into memory.**

Symbolic Address Resolution

- **Memory reference instructions usually reference memory locations symbolically (eg. variable name)**
- **Programmers may define**
 - Earlier than the reference in
 - Later than the reference in

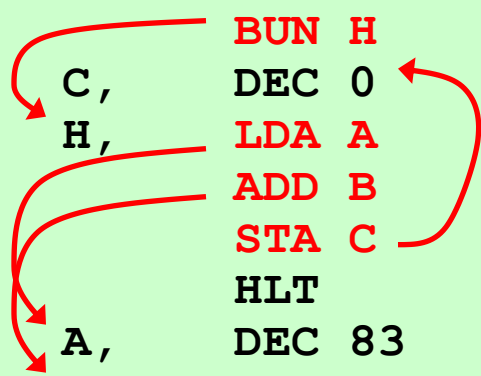
PROBLEM:

SOLUTION:

Use a **Symbol Table** to record symbol definitions and References.

Need to pass through program twice in order to resolve all symbolic addresses.

<u>Label</u>	<u>Instruction</u>	<u>Comment</u>
	ORG 0	/Pseudo-in
	BUN H	/Machine i
C,	DEC 0	/Pseudo-in
H,	LDA A	/Machine i
	ADD B	/Machine i
	STA C	/Machine i
	HLT	/Machine in
A,	DEC 83	/Pseudo-instr
B,	DEC -23	/Pseudo-instr
	END	/Pseudo-instr



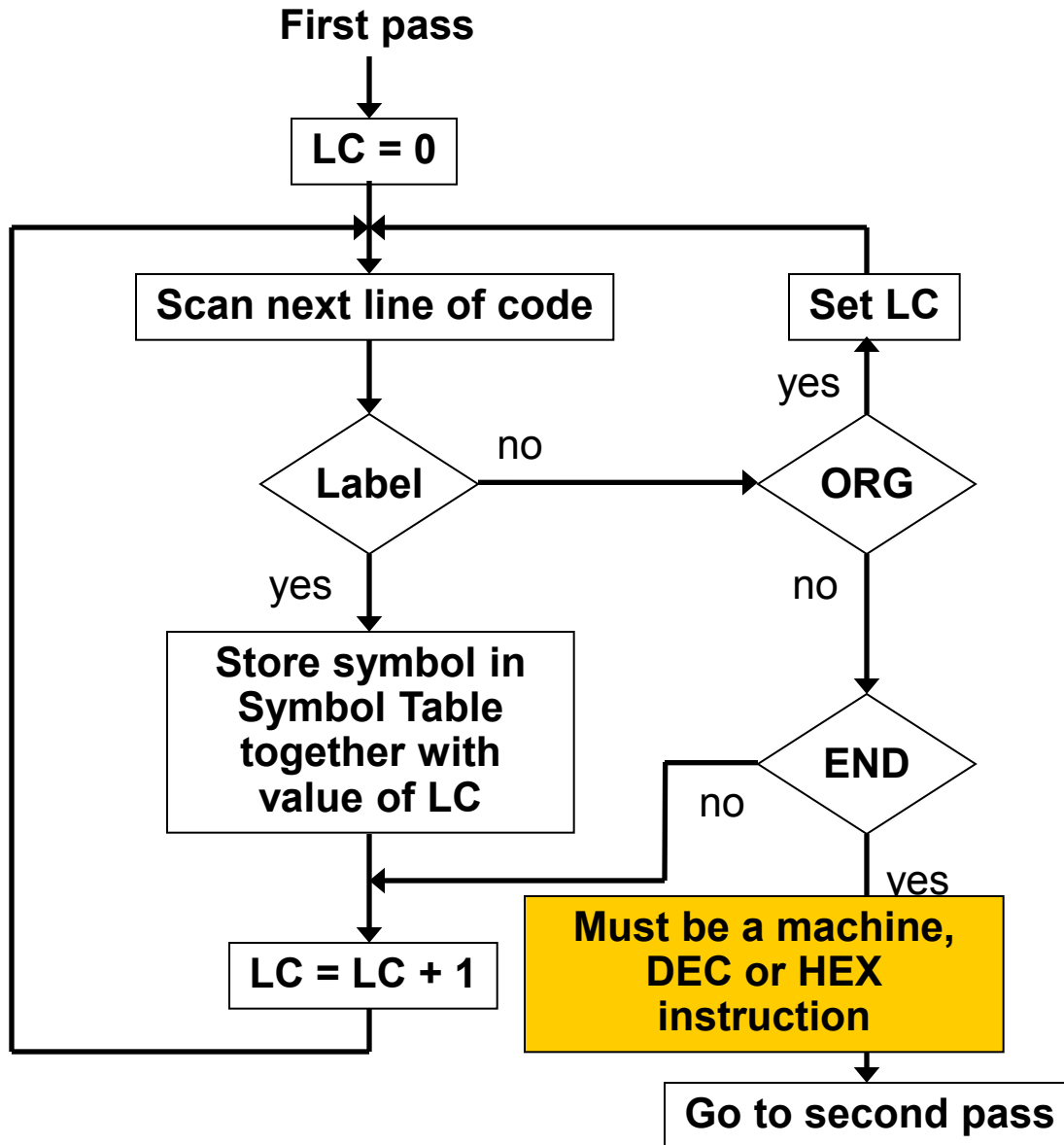
Location Referencing

- **In order to correctly translate a symbolic address reference, it is necessary to know where the symbol is defined within the program**
- **A special memory location counter (LC, or MLC) is defined within the Assembler program.**
 - **Default initial value :: $LC = 0$**
 - **Set value to specified location :: $ORG\ n\ (LC = n)$**
 - **Default Modification**
 - **For each machine instruction and DEC/HEX pseudo-instruction, $LC = LC + 1$**

Assembler Flowchart

- **Assembler requirements:**
 - **Location Counter (LC)**
 - **Symbol Table**
 - **Recognition of OpCodes**
 - **Recognition of Pseudo-Instructions**
 - **Recognition of Comments**
 - **Detection and Definition of User-Defined Symbolic Addresses**
 - **Detection and Resolution of User-Defined Symbolic Address References**
 - **Multiple passes through program source code**
 - **Production of Object Code (ie. *machine* code)**

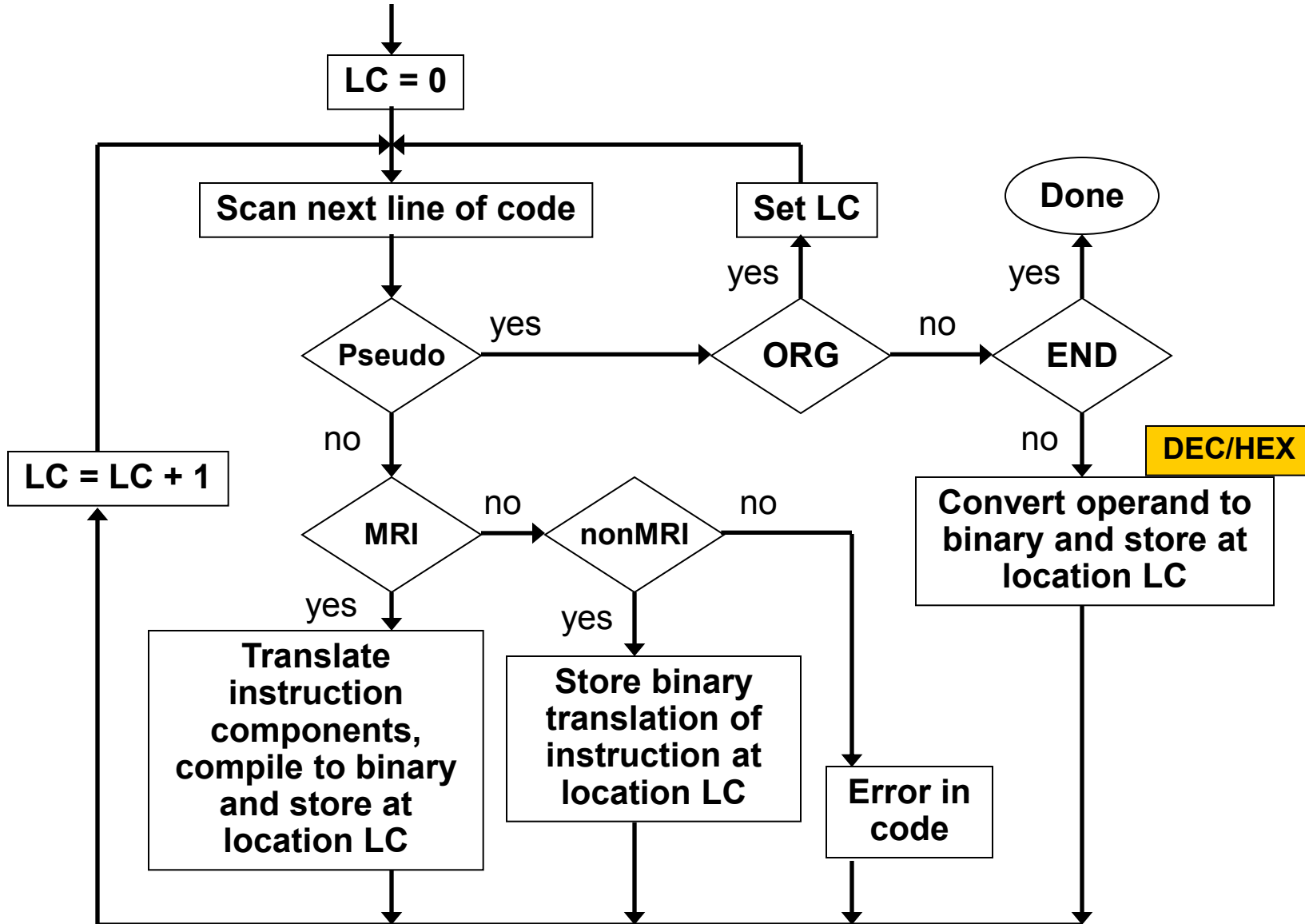
Assembler Flowchart – Pass 1



Symbol Table	
LABEL1	LC1
■ ■ ■	
LABELn	LCn

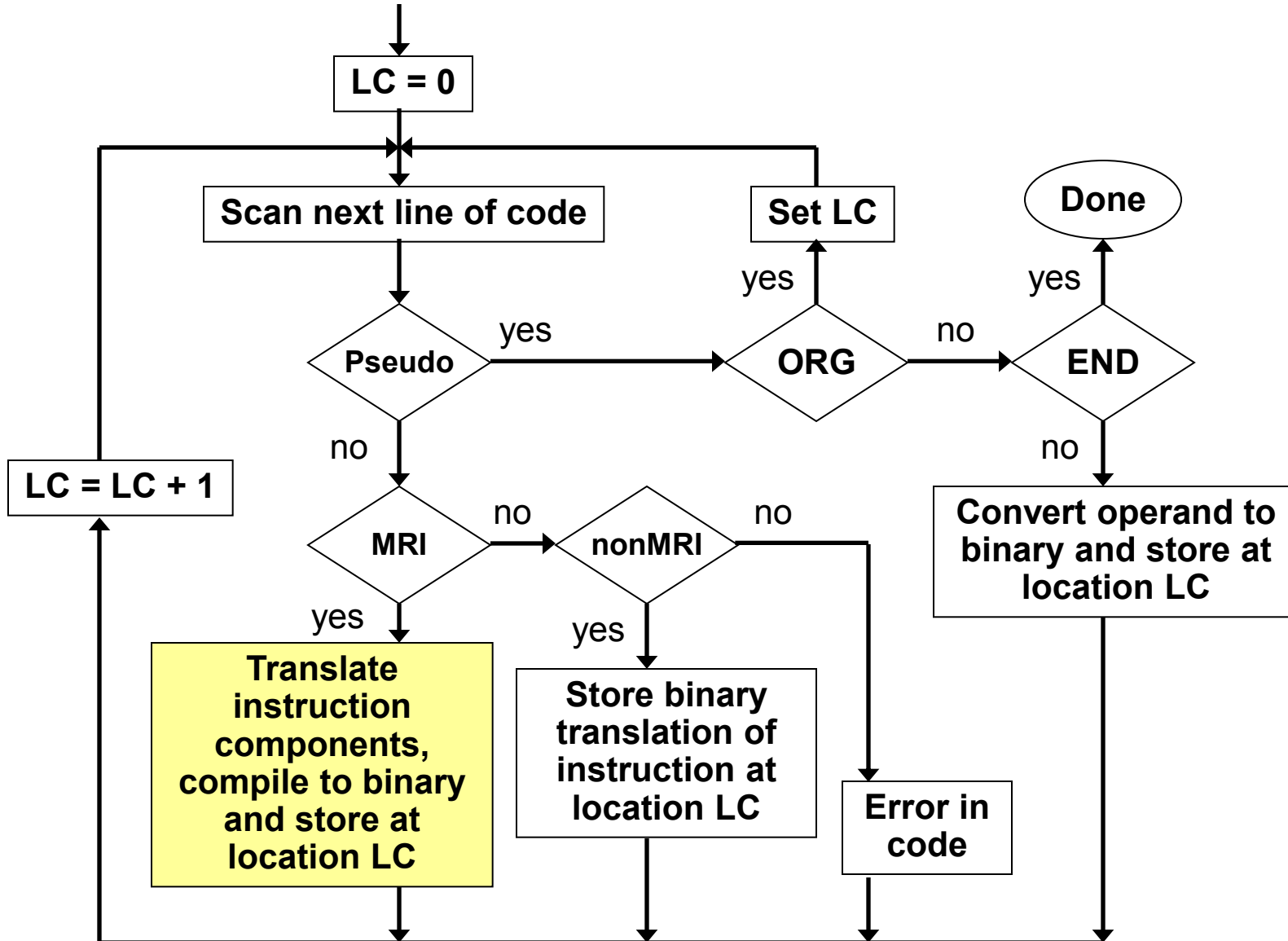
Assembler Flowchart – Pass 2

Second pass

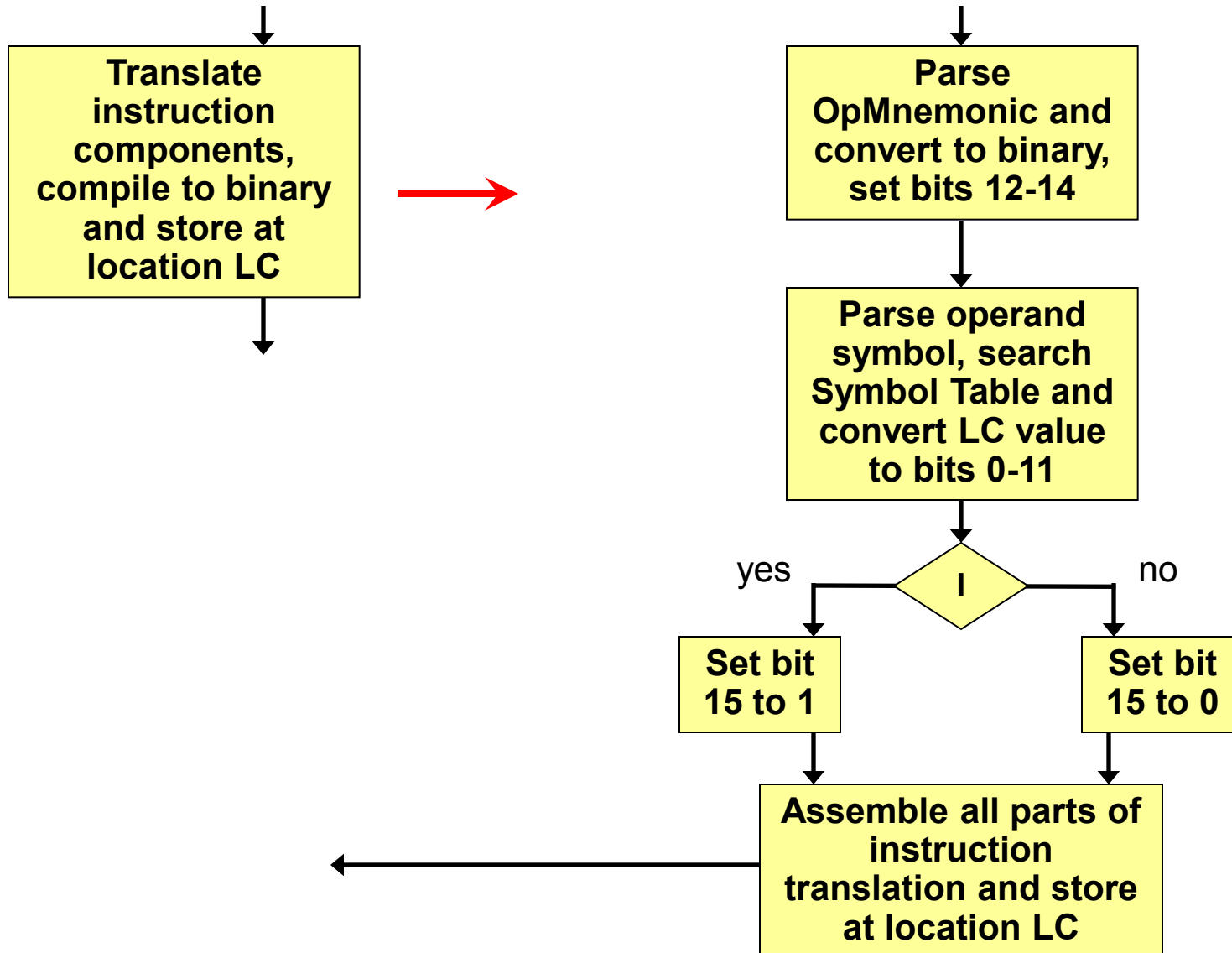


Assembler Flowchart – Pass 2

Second pass



Assembler Flowchart – Pass 2 (MRI)



Program Examples

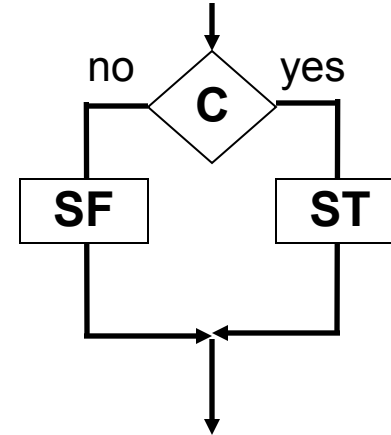
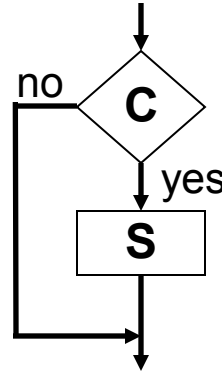
- **We consider several examples in order to illustrate how programs may be written using Assembly Language**
- **More importantly, we concentrate on how such programs are translated into machine code**
 - **Where the “code hits the circuitry”**



Program Example 1: Branching

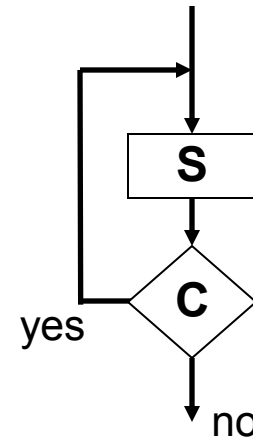
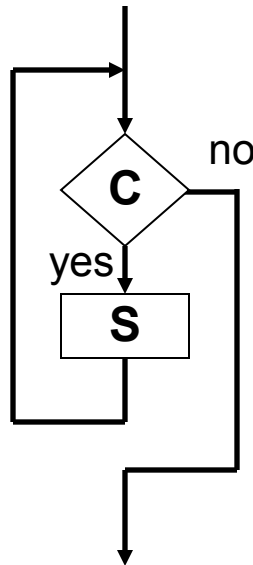
- **Decision**

- **If-then**
- **If-then-else**



- **Repetition (loops)**

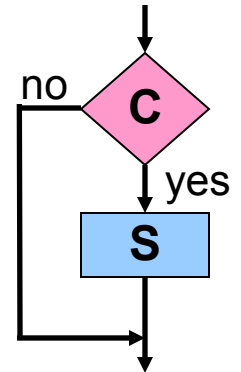
- **While-do**
(For-until-do)
- **Do-while**



Program Example 1A: Decision

- Program a simple <if-then> branching structure
- Assume a flag is stored at FLG in memory. If $FLG > 0$ then we add integers N1 and N2 and store the result; otherwise we do nothing.

```
ORG 0
LDA FLG      /PREPARE CONDITION
SPA         /IF TRUE PERFORM THEN-clause
BUN CON      /OTHERWISE, BYPASS clause
THN, LDA N1  /BEGIN THEN-clause
ADD N2
STA SUM      /END THEN-clause
CON, . . .   /CONTINUE PROGRAM
HLT
FLG, DEC 1
N1,  DEC 20
N2,  DEC 15
SUM, DEC 0
END
```



Program Example 1A: Decision

- Pr
- As
- ad
- no

ASSEMBLER - PASS 1

LC	SYM	INSTR
---	---	-----
		ORG 0
000		LDA FLG
001		SPA
002		BUN CON
003	THN,	LDA N1
004		ADD N2
005		STA SUM
006	CON,	HLT
007	FLG,	DEC 1
008	N1,	DEC 20
009	N2,	DEC 15
00A	SUM,	DEC 0
00B		END

SYMB
CON
FLG
N1
N2
SUM
THN

OBJECT CODE FILE: (BINARY FILE) (LD PT 000)

20077010400620081009300A700100010014000F0000

ASSEMBLER - PASS 2

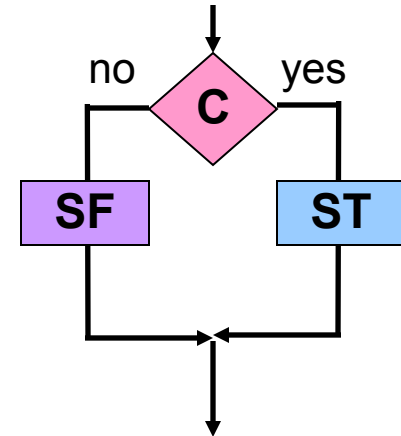
LC	OBJ	SYM	INSTR
---	----	---	-----
			ORG 0
000	2007		LDA FLG
001	7010		SPA
002	4006		BUN CON
003	2008	THN,	LDA N1
004	1009		ADD N2
005	300A		STA SUM
006	7001	CON,	HLT
007	0001	FLG,	DEC 1
008	0014	N1,	DEC 20
009	000F	N2,	DEC 15
00A	0000	SUM,	DEC 0
00B			END

N1
N2
SUM
THN

RDS

Program Example 1B: Decision

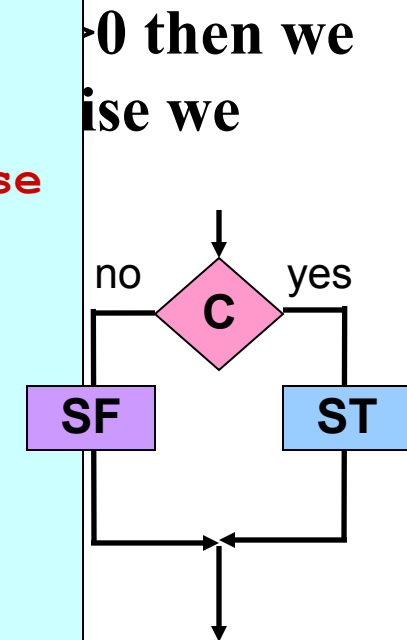
- Program an <if-then-else> branching structure
- Assume a flag is stored at FLG in memory. If $FLG > 0$ then we add integers N1 and N2 and store the result; otherwise we subtract N2 from N1 and store the result.



Program Example 1B: Decision

- Program an <if-then-else> branching structure

```
ORG 0
LDA FLG      /PREPARE CONDITION
SPA          /IF TRUE PERFORM THEN-clause
BUN ELS      / OTHERWISE, BRANCH TO ELSE-clause
THN, LDA N1  /BEGIN THEN-clause
ADD N2
STA VAL      /END THEN-clause
BUN CON      /BRANCH TO END OF IF-structure
ELS, LDA N2  /BEGIN ELSE-clause
CMA
INC
ADD N1
STA VAL      /END ELSE-clause
CON, HLT     /CONTINUE PROGRAM
FLG, DEC 1
N1, DEC 20
N2, DEC 15
VAL, DEC 0
END
```



Program Example 1B: Decision

ASSEMBLER - PASS 1

LC	SYM	INSTR
----	-----	-------

		ORG 0
000		LDA FLG
001		SPA
002		BUN ELS
003	THN,	LDA N1
004		ADD N2
005		STA VAL
006		BUN CON
007	ELS,	LDA N2
008		CMA
009		INC
00A		ADD N1
00B		STA VAL
00C	CON,	HLT
00D	FLG,	DEC 1
00E	N1,	DEC 20
00F	N2,	DEC 15
010	VAL,	DEC 0
011		END

SYMBOL	LC
CON	00C
ELS	007
FLG	00D
N1	00E
N2	00F
THN	003
VAL	010

ANCH TO END OF IF-st

IN ELSE-clause

ELSE-clause

CONTINUE PROGRAM

VAL, DEC 0
END

ASSEMBLER - PASS 2

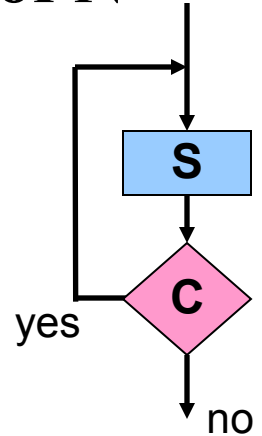
LC	OBJ	SYM	INSTR
----	-----	-----	-------

			ORG 0
000	200D		LDA FLG
001	7010		SPA
002	4007		BUN ELS
003	200E	THN,	LDA N1
004	100F		ADD N2
005	3010		STA VAL
006	400C		BUN CON
007	200F	ELS,	LDA N2
008	7200		CMA
009	7020		INC
00A	100E		ADD N1
00B	3010		STA VAL
00C	7001	CON,	HLT
00D	0001	FLG,	DEC 1
00E	0014	N1,	DEC 20
00F	000F	N2,	DEC 15
010	0000	VAL,	DEC 0
011			END

LENGTH=17 WORDS

Program Example 1C: Loops

- **A simple counting loop to find the sum of a list of N integers.**



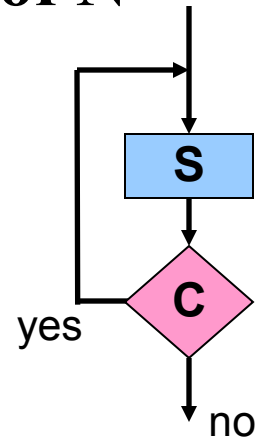
- A simple integer

```

        ORG 0
        LDA N          /PREPARE COUNT VARIABLE
        CMA
        INC
        STA N
        SPA             /TEST IF N>0
        BUN CON         / OTHERWISE, BYPASS LOOP
LOP,    LDA SUM         /BEGIN LOOP
        ADD VP I        /Note use of indirect mode
        STA SUM
        ISZ VP          /Increment V-pointer
        ISZ N           /N=N+1, EXIT LOOP IF N=0
        BUN LOP         /BRANCH TO LOOP BEGIN
CON,    HLT             /CONTINUE PROGRAM
N,      DEC 5
SUM     DEC 0
VP,     HEX 20          /ADDRESS OF V
        ORG 20
V,      DEC 5           /VECTOR OF INTEGERS
        DEC 15
        DEC 10
        DEC 8
        DEC 17
        END

```

st of N



ASSEMBLER - PASS 1

LC SYM INSTR
 --- ---

ORG 0
 000 LDA N
 001 CMA
 002 INC
 003 STA N
 004 SPA
 005 BUN CON
 006 LOP, LDA SUM
 007 ADD VP I
 008 STA SUM

/PREPARE COUNT

SYMBOL LC
 CON 00C
 LOP 006
 N 00D
 V 020
 VP 00F

OBJECT FILE: (LD PT 000) LENGTH 37 WORDS (HEX 25)

LOC HEX CODE

--- -----

00D N, 000 200D 7200 7020 300D 7010 000C 200E 9009
 00E 008 300E 600F 600D 4006 7001 0005 0000 0020
 00F VE 010 0000 0000 0000 0000 0000 0000 0000 0000
 018 0000 0000 0000 0000 0000 0000 0000 0000
 020 V, 020 0005 000F 000A 0008 0011

DEC 8

DEC 17

END

ASSEMBLER - PASS 2

LC OBJ SYM INSTR
 --- ----

ORG 0
 000 200D LDA N
 001 7200 CMA
 002 7020 INC
 003 300D STA N
 004 7010 SPA
 005 400C BUN CON
 006 200E LOP, LDA SUM
 007 9009 ADD VP I
 008 300E STA SUM
 009 600F ISZ VP

N
 LOP

5
 DEC 0

20

20

5

15

10

8

024 0011

DEC 17

025

END

LENGTH=37 WORDS

Program Example 2: Multiplication (0)

- We consider a typical problem of multiplying two integer values.
- When considering any algorithm, one must consider the computer architecture in terms of the instruction set.
 - The **fundamental building blocks** of programming
 - Constraints on the **expression of logical intention**
- Multiplication of two positive integers may be accomplished by repeated addition
 - Ex. $4 \times 8 = 8 + 8 + 8 + 8$
- Must also take into account the very real possibility that the product will overflow the accumulator register
 - Ex. 4-bit registers $0101 \times 0100 = 1\ 0100$

But, integers may be either positive, negative or zero.

Program Example 2: Multiplication (1)

- In general, $M \times N$ requires checking the sign of

CONCEPTUAL

This requires :

- Checking the sign of each operand,
- Storing the overall sign,
- Obtaining the 2's complement (negation) of the operands.
- Using repeated addition to obtain multiplication

- M positive, N positive :: $M \times N$
- M positive, N negative :: $-(M \times (-N))$
- M negative, N positive :: $-((-M) \times N)$
- M negative, N negative :: $(-M) \times (-N)$

PRACTICAL CONSTRAINTS

- Only the AC register can be used to accumulate addition
- Since the AC must also be used for other tasks, it must be repeatedly stored and recovered from memory
- A counter will need to be set for loop control. Various types of branching instructions will be needed for decisions and loops.

Program Example 2: Multiplication (2)

```
ORG 0
LDA M      /LOAD M
SPA        /TEST IF M>0
BUN CM     /GO TO MODIFY M
TSN, LDA N  /LOAD N
SPA        /TEST IF N>0
BUN CN     /GO TO MODIFY N
MUL, LDA N  /LOAD COUNTER
CMA        /PREPARE COUNTER
INC
STA C      /STORE COUNTER
LOP, LDA S  /LOAD SUM
ADD M      /ADD M
STA S      /STORE SUM
ISZ C      /TEST IF C<0
BUN LOP    /REPEAT LOOP
BUN STP    / IF C=0 STOP
CM, SNA     /TEST IF M<0
BUN STP    / IF 0 STOP
CMA        /NEGATE M
INC
STA M      /STORE M
BUN TSN    /GO TO TEST N
```

```
CN, SNA     /TEST IF N<0
      BUN STP  / IF C=0 STOP
      CMA      /NEGATE N
      INC
      STA N     /STORE N
      BUN MUL   /GO TO MULTIPLY
STP, HLT     /STOP PROGRAM
M, DEC 20
N, DEC -64
C, DEC 0     /COUNTER
S, DEC 0     /SUM
END
```

Program Example 2: Multiplication (3)

- **An alternative approach**
 - Use simple properties of radix-2 (binary) multiplication
 - More efficient (fixed time complexity)

M	0000	1101	Decimal 13
N	x 0000	1011	Decimal 11
=	=====	=====	
	0000	1101	
	0 0001	101	
	00 0000	00	
	000 0110	1	

	000	1000 1111	Product = 143

Note: In general, multiplying an N digit integer by an M digit integer results in, at most, an N+M digit product.

Program Example 2: Multiplication (4)

```

M      0000 1101      Decimal 13  T=M, P=0 (Initialize) C=4
N  x  0000 1011      Decimal 11  Product = 143
=      =====

```

```

T      0000 1101      N=CIR(N) [0000 0101]  E=1
P  +  0000 0000      (E) : P=P+T
      -----
P      0000 1101      CLE T=CIL(T) C=C-1 [3]  STOP IF C IS ZERO

```

```

T      0001 1010      N=CIR(N) [0000 0010]  E=1
P  +  0000 1101      (E) : P=P+T
      -----
P      0010 0111      CLE T=CIL(T) C=C-1 [2]  STOP IF C IS ZERO

```

```

T      0011 0100      N=CIR(N) [0000 0001]  E=0
                        (E) : P=P+T  Do Nothing
CLE T=CIL(T) C=C-1 [1]  STOP IF C IS ZERO

```

```

T      0110 1000      N=CIR(N) [0000 0000]  E=1
P  +  0010 0111      (E) : P=P+T
      -----
P      1000 1111      CLE T=CIL(T) C=C-1 [0]  STOP IF C IS ZERO
T      1101 0000

```

P = 1+2+4+8+128=143 as expected.

Program Example 2: Multiplication (5)

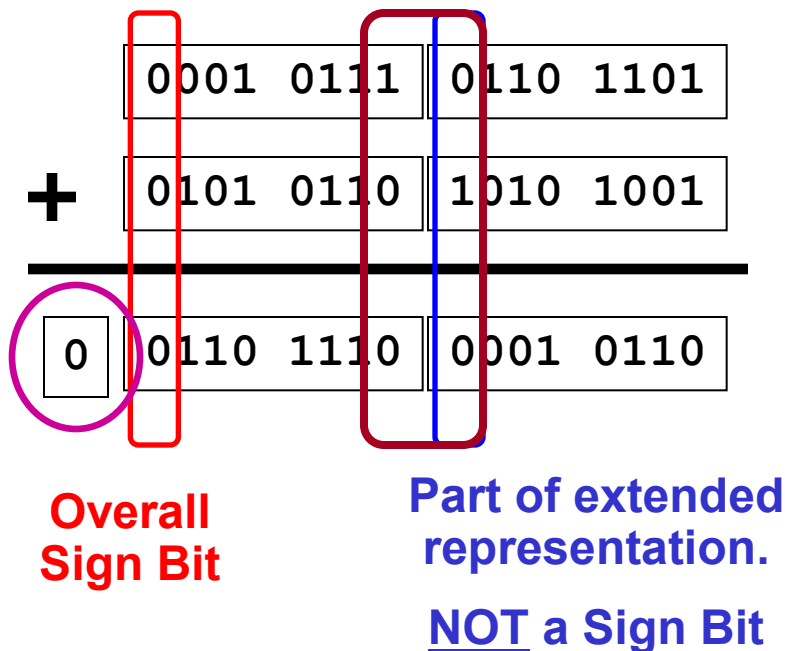
```
      ORG 0
      BUN MUL
C,    DEC -4      /COUNTER
M,    DEC 13      /MULTIPLICAND
N,    DEC 11      /MULTIPLIER
P,    DEC 0
T,    DEC 0
MUL,  LDA M       /INITIALIZE SHIFTER T
      STA T
CHN,  LDA N       /TEST LOW ORDER BIT OF N
      CLE
      CIR
      SZE        /IF BIT IS ZERO DO NOTHING
      BUN AD     / OTHERWISE, ADD SHIFTER
      BUN BY
AD,   LDA P       /ADD SHIFTER TO PRODUCT
      ADD T
      STA P       /STORE PRODUCT
BY,   LDA T       /SHIFT SHIFTER LEFT
      CLE
      CIL
      STA T       /STORE SHIFTER UPDATE
      ISZ C       /INCREMENT COUNTER, IF ZERO STOP
      BUN CHN    / OTHERWISE, REPEAT LOOP AT CHN
      HLT
      END
```

This can be further extended for use as a subroutine, reinitializing the counter and passing new values M and N each time.

Also, sign checking can be incorporated.

Example 3: Double-Precision Addition (0)

- **Extending precision and ADD support is called *emulated* addition. This is a common technique.**
 - Requires understanding of hardware and representations
 - Consider the problem of extending 8-bit to 16-bit:



PROBLEMS:

1. Must handle Hi order bit properly in each part of the extended representation
2. Must handle overall sign
3. Should handle overflow possibility
4. Must handle carry bit from each low order part addition

Example 3: Double-Precision Addition (1)

- **Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).**
 - **First, deal with low order part of operands**

	0001 0111	0110 1101
+	0101 0110	1010 1001
<hr/>		

Example 3: Double-Precision Addition (1a)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - First, deal with low order part of operands

	0001 0111	0110 1101	0000 0110	0000 1101
+	0101 0110	1010 1001	1111 0000	0000 1111
<hr/>				

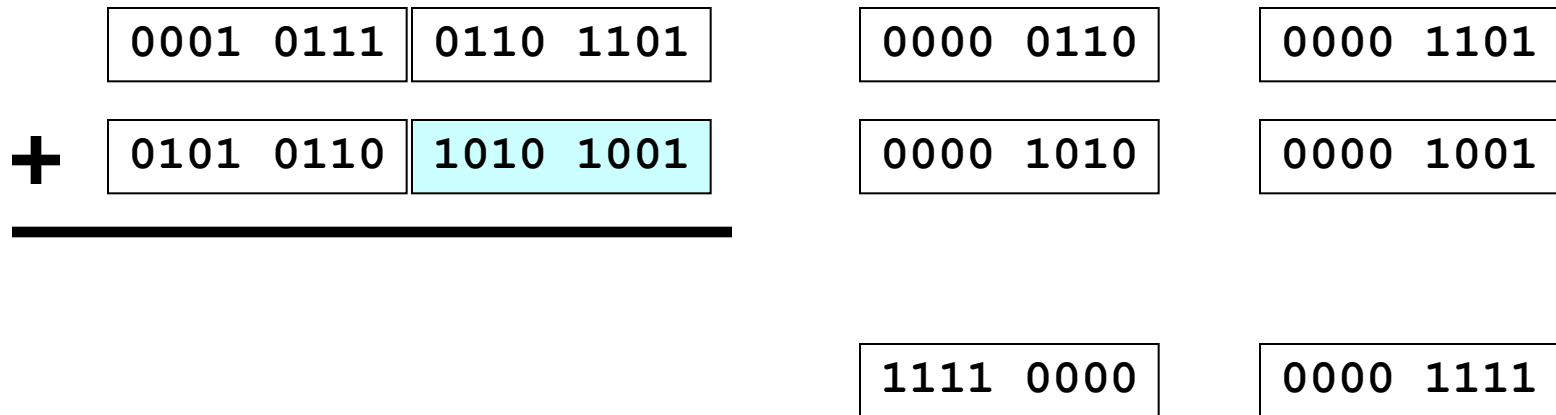
Use Mask to retain
high order bits
and clear low
order bits.

Then, shift right
four times.

Use Mask to
retain low
order bits and
clear high
order bits

Example 3: Double-Precision Addition (1b)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - First, deal with low order part of operands



Use Mask to retain and clear bits.
Shift high order part right four times.

Example 3: Double-Precision Addition (1c)

- **Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).**
 - **First, deal with low order part of operands**

	0001 0111	0110 1101
+	0101 0110	1010 1001
<hr/>		

1111 0000

0000 1111

0000 0110		0000 1101
0000 1010	+	0000 1001
<hr/>		

	0001 0110
0000 0001	0000 0110

Use Masks to retain and clear bits.
Shift high order part right four times.

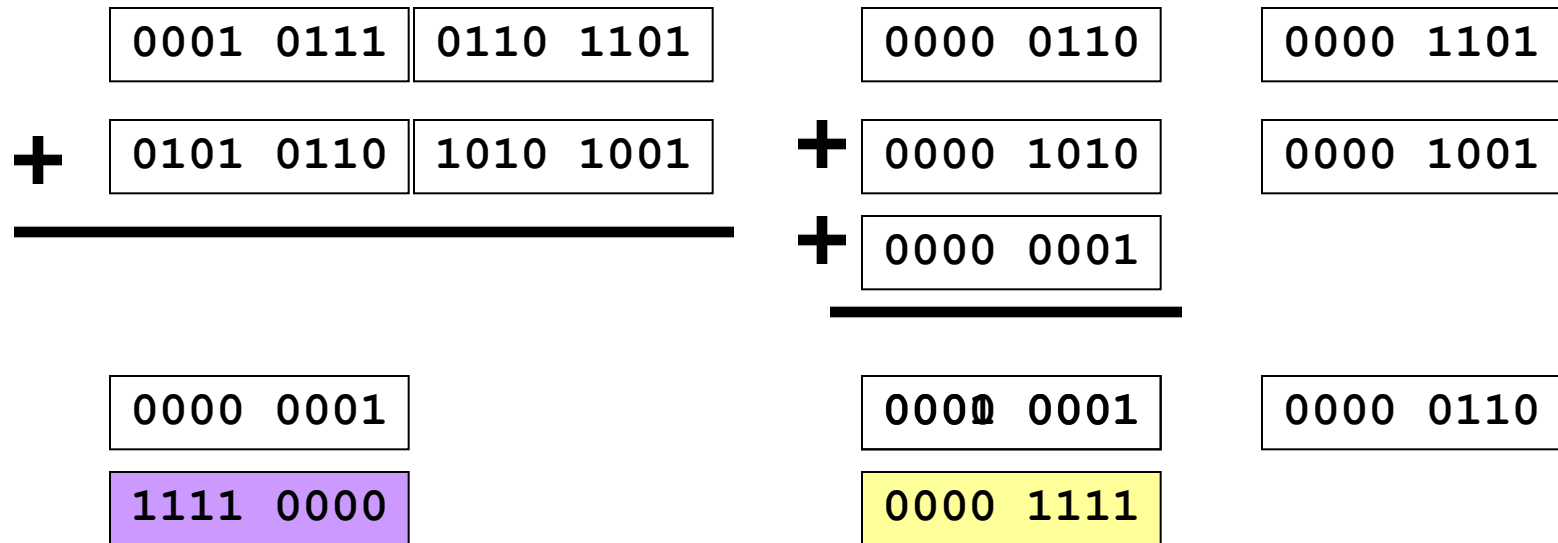
Example 3: Double-Precision Addition (1d)

- **Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).**
 - **First, deal with low order part of operands**

	<div>0001 0111</div>	<div>0110 1101</div>		<div>0000 0110</div>	<div>0000 1101</div>
+	<div>0101 0110</div>	<div>1010 1001</div>	+	<div>0000 1010</div>	<div>0000 1001</div>
<hr/>			+	<div>0000 0001</div>	
			<hr/>		
			<div>0000 0001</div>	<div>0000 0110</div>	

Example 3: Double-Precision Addition (1d)

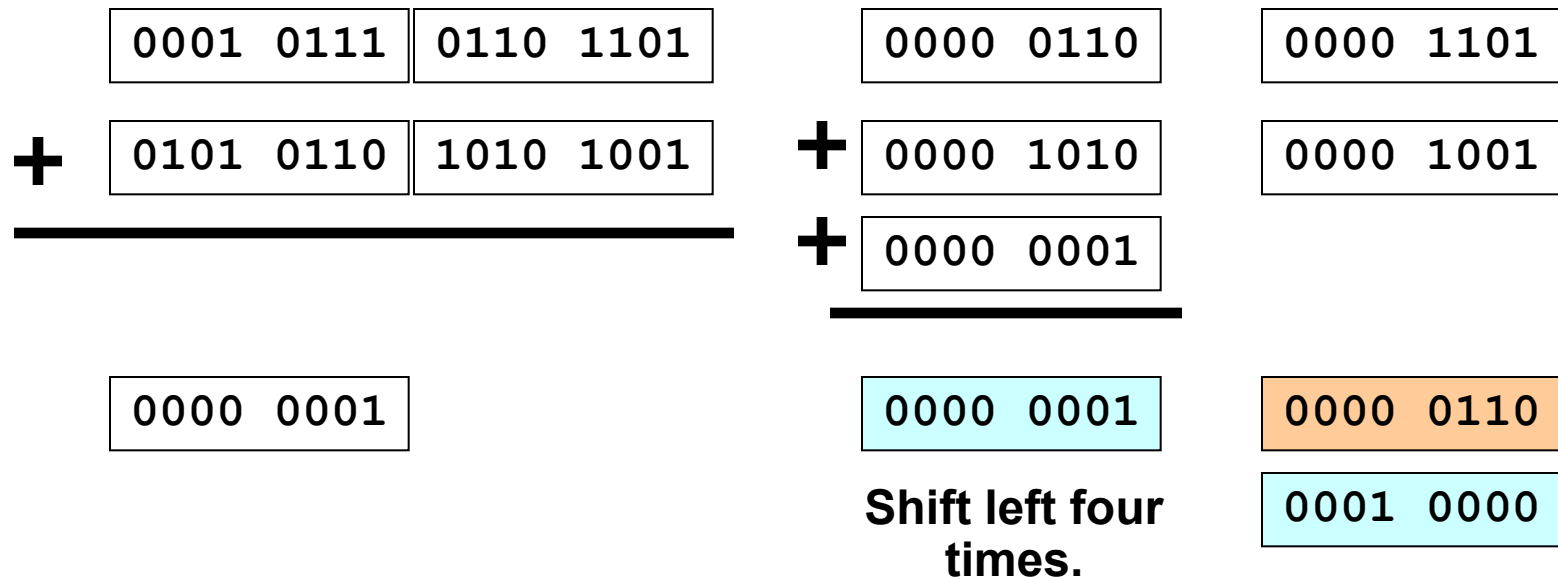
- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - First, deal with low order part of operands



Use Masks to retain and clear bits. Shift high order part right four times.

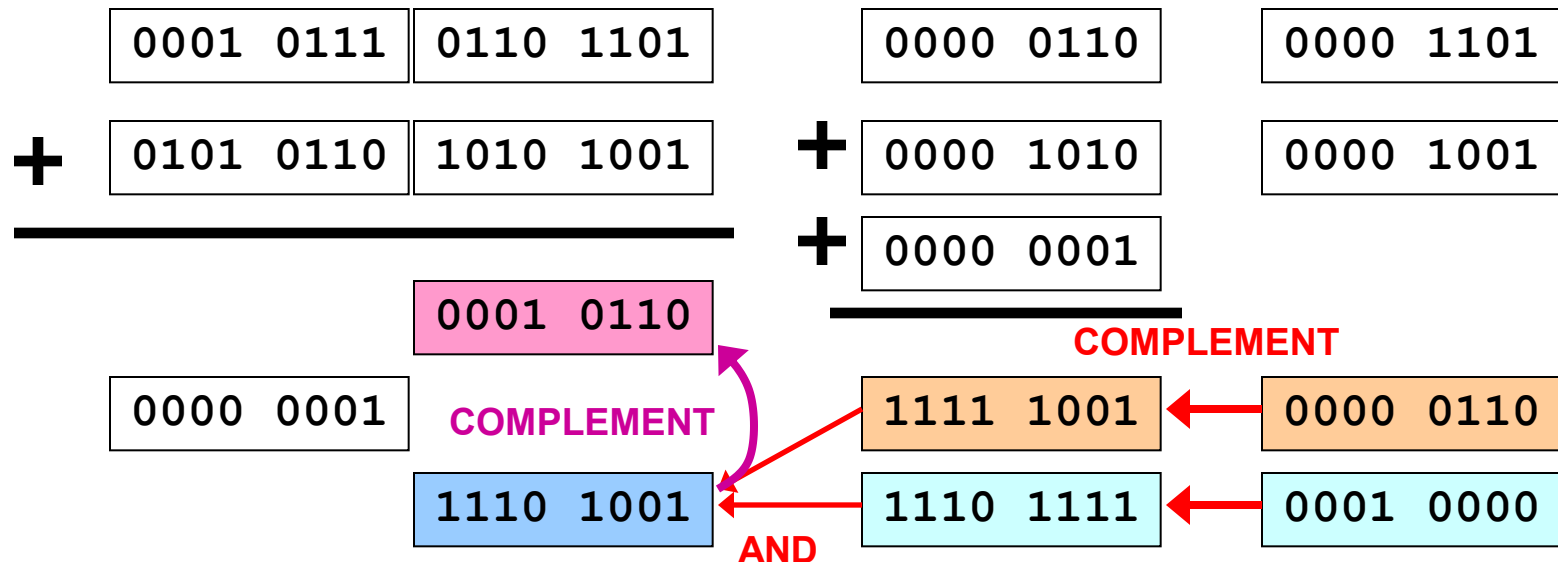
Example 3: Double-Precision Addition (1e)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - First, deal with low order part of operands



Example 3: Double-Precision Addition (1f)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - First, deal with low order part of operands



RECALL LOGIC: $A+B = (A+B)'' = (A'B')'$

Example 3: Double-Precision Addition (2)

- **Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).**
 - **Next, deal with high order parts, including carry**

	0001 0111	0110 1101
+	0101 0110	1010 1001
<hr/>		
		0001 0110
CARRY	0000 0001	

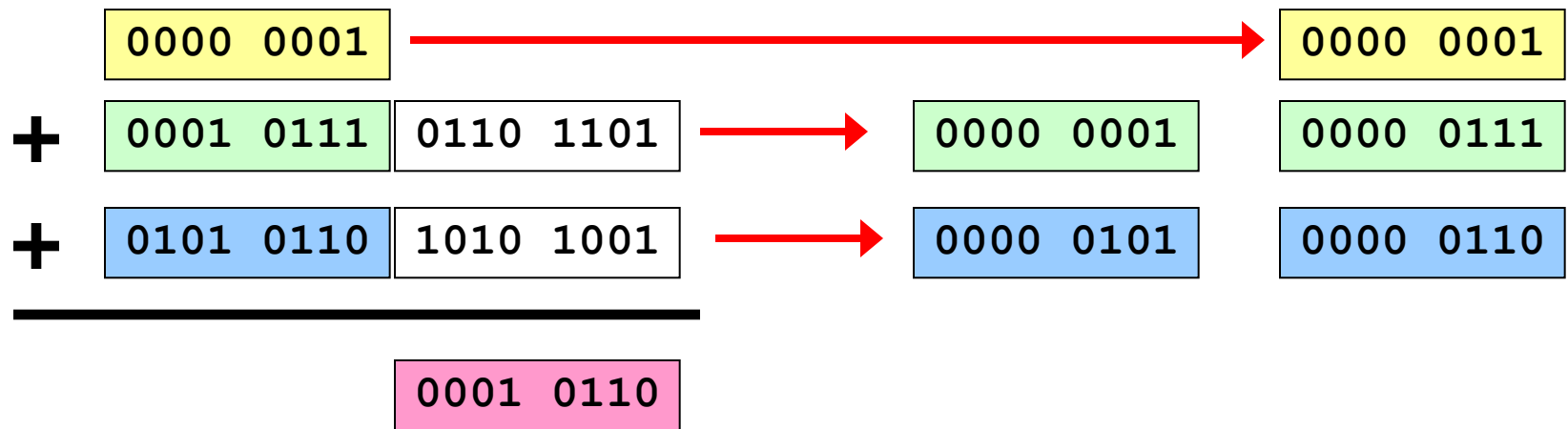
Example 3: Double-Precision Addition (2)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry

	0000 0001	
+	0001 0111	0110 1101
+	0101 0110	1010 1001
<hr/>		
		0001 0110
CARRY	0000 0001	

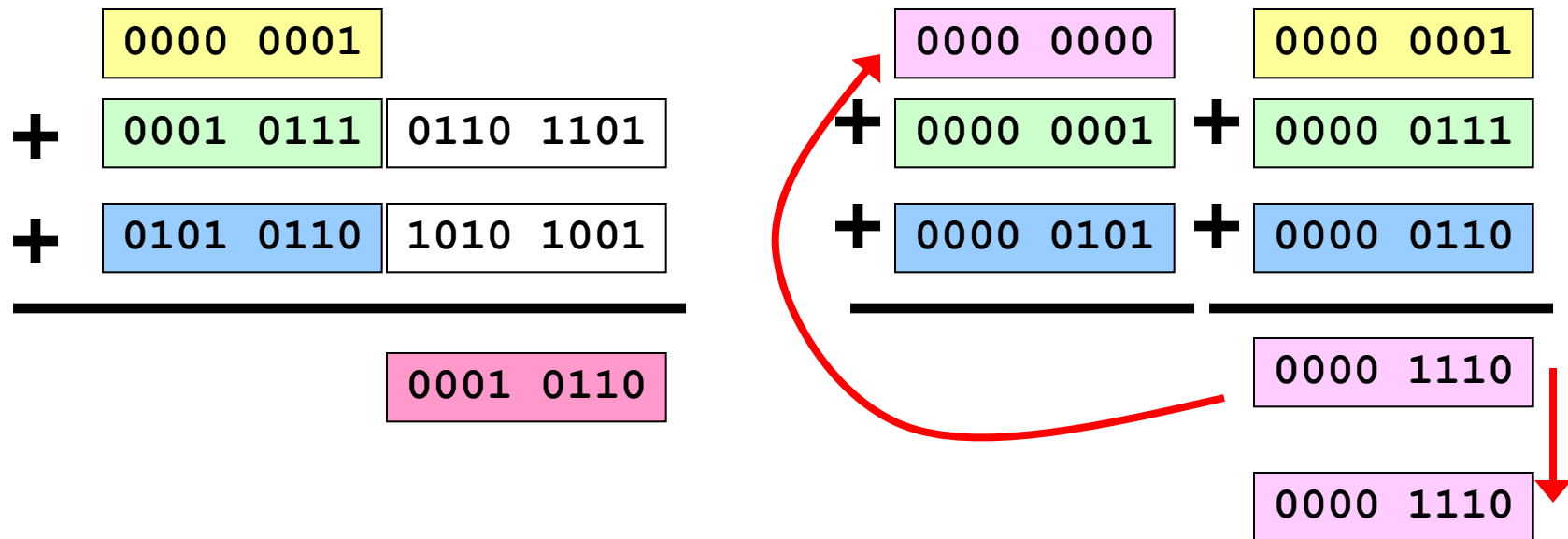
Example 3: Double-Precision Addition (2a)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry



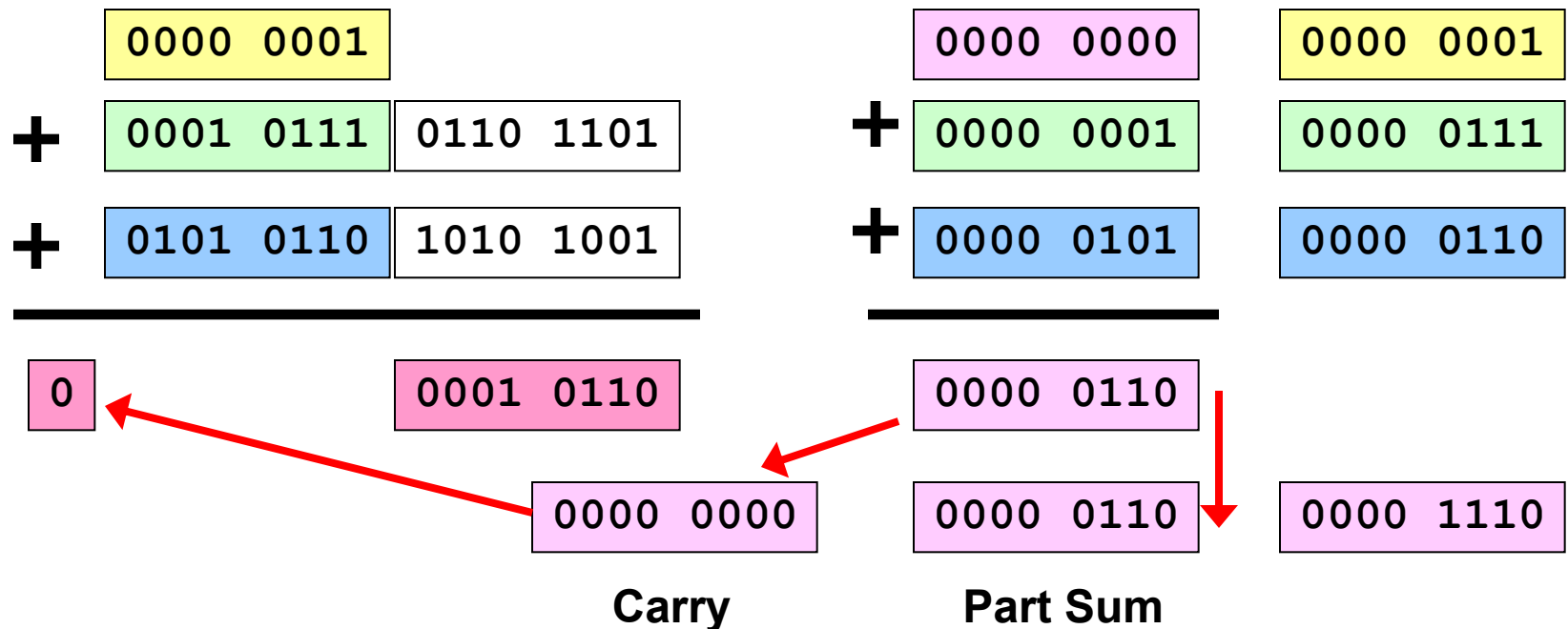
Example 3: Double-Precision Addition (2b)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry



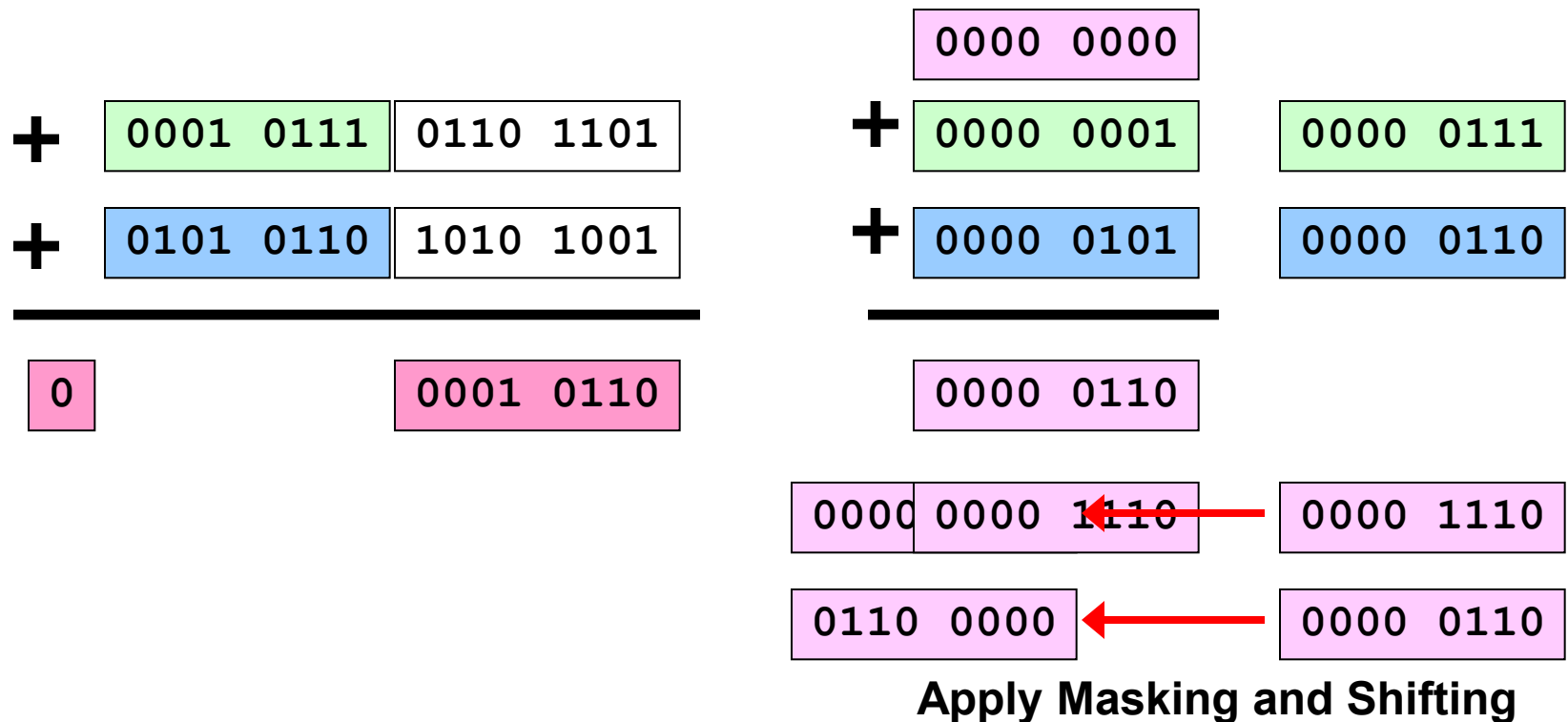
Example 3: Double-Precision Addition (2c)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry



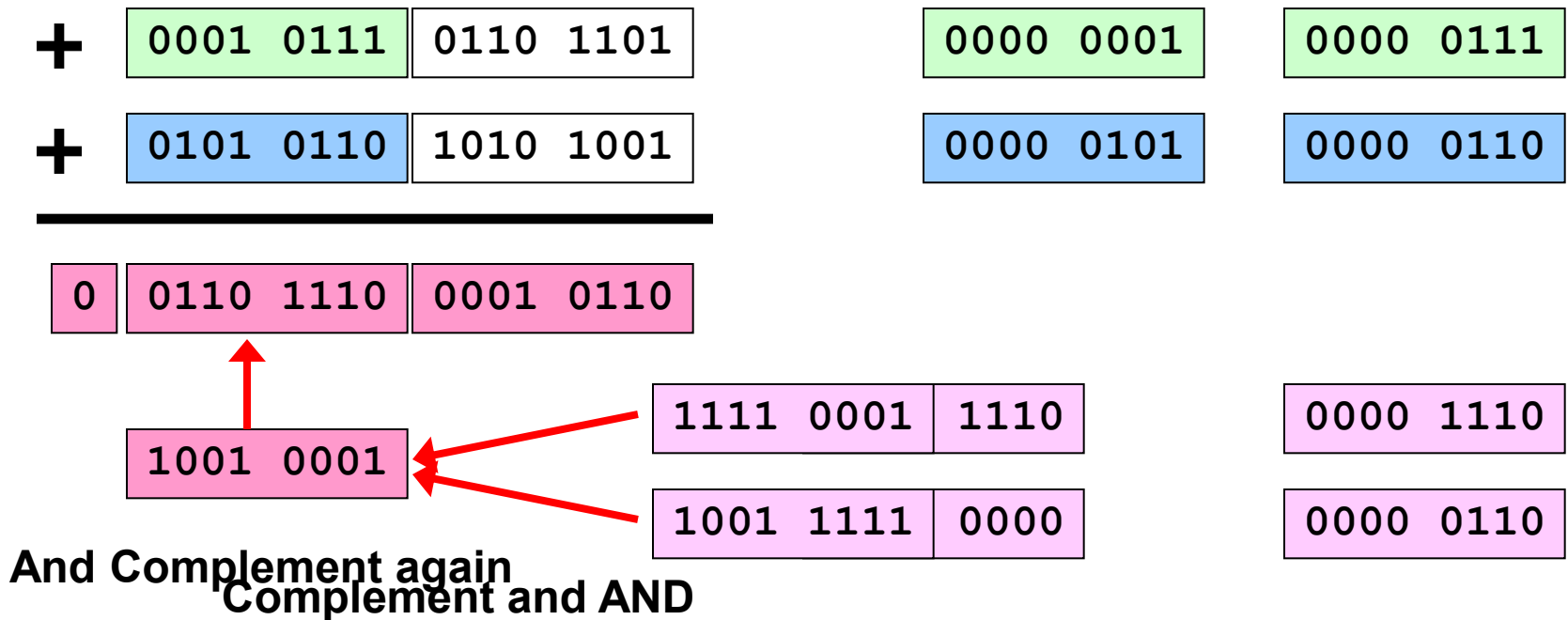
Example 3: Double-Precision Addition (2d)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry



Example 3: Double-Precision Addition (2e)

- Deal with each part of the extended representation separately, starting from the low order end to handle the carry bit (ripple addition simulation).
 - Next, deal with high order parts, including carry



Example 3: Double-Precision Addition (3)

- Here are codes to illustrate some of the points raised

```
/SIMPLE SUBROUTINE FOR SEPARATING
/  HIGH-LOW PARTS OF 16bit INTEGER
MDO, HEX 0
      BUN MD1      /JUMP OVER DATA
M,    DEC 0        /VALUE PASSED
MH,   DEC 0        /HIGH PART
ML,   DEC 0        /LOW PART
MSH,  HEX FF00     /HIGH MASK
MSL,  HEX 00FF     /LOW MASK
MD1,  LDA M        /LOAD M
      AND MSH      /CLEAR LOW BITS
      CIR
      CIR
      CIR
      CIR          /SHIFT RIGHT 4 TIMES
      STA MH       /STORE HIGH PART
      LDA M        /LOAD M
      AND MSL      /CLEAR HIGH BITS
      STA ML       /STORE LOW PART
      BUN MDO I    /RETURN
```

```
/A OR B USING COMPLEMENT and AND
/   $A+B = (A+B)'' = (A'B')'$ 
      LDA B        /PREPARE B
      CMA          /COMPLEMENT B
      STA B        /STORE B
      LDA A        /PREPARE A
      CMA          /COMPLEMENT A
      AND B        /A' AND B'
      CMA          /COMPLEMENT (A'B')'
      STA A        /STORE A
```

Example 3: Double-Precision Addition (4)

- **There are many algorithms for multiple precision arithmetic**
- **All rely on the essential hardware for efficiency and complexity**
- **We focused on simplicity, based on availability of single operand ADD, AND and shifting instructions, as well as the ability to perform counting loops.**
- **Mano provides a detailed example and assembly language coding for this problem**

Example 4: Subroutines (0)

- **Most high level languages support the notion of a subroutine**
 - A procedure that accepts input data, performs a specific action and returns data
 - May be referenced (called) from many locations and always returns to the point of reference.
- **Non-recursive** subroutines are relatively simple to implement
 - Need to **CALL** the subroutine by Name (symbol ref.)
 - Need to **pass parameters** to subroutine (Parameter Linkage convention)
 - Need to **return values** from subroutine
 - Need to **return from subroutine** to the next instruction following the CALLing instruction (Subroutine Linkage convention)

Example 4: Subroutines (1)

- **We consider a simple subroutine that adds two integers and returns the sum**
 - **Must design a strategy for passing data to and from the subroutine, regardless of which instruction invokes it.**
 - **Copy two integers**
 - **Return one integer**
 - **Must be able to access the passed data from within the subroutine.**
 - **Use pointer technique**
 - **Must be able to return to the proper return point address to continue execution of the program section that invoked the subroutine.**
 - **Store return point address (or base reference)**
 - **We are NOT concerned about trapping exceptions.**

Example 4: Subroutines (2)

- **First, we consider the issues of invoking (calling) the subroutine**
 - **Preparing the values (parameters)**
 - A responsibility of the programmer
 - For high level languages this is accomplished by the compiler
 - **Passing control to the subroutine**
 - Must ensure that the return address is specified
 - Mano provides this using the BSA instruction
 - *BSA constrains* the approach to subroutine design and use

```
. . .  
LDA X      /PREPARE INPUT DATA  
STA PM1    / USING STACK  
LDA Y      /   BASED  
STA PM2    /   STRUCTURE  
BSA SUB     /JUMP TO SUBROUTINE  
RTV, DEC 0  /RETURN VALUE  
PM1, DEC 0  /FIRST PARAMETER VALUE  
PM2, DEC 0  /NEXT PARAMETER VALUE  
RPA, LDA RTV /RETURN POINT ADDRESS  
. . .
```

Choose a data structure based on number and type of data.

Locate the data structure at the address in PC when BSA is executed. This address is passed to the subroutine.

Example 4: Subroutines (3)

- **First, we consider the issues of invoking (calling) the subroutine**
 - **Preparing the values (parameters)**
 - A responsibility of the programmer
 - For high level languages this is accomplished by the compiler
 - **Passing control to the subroutine**
 - Must ensure that the return address is specified
 - Mano provides this using the BSA instruction
 - *BSA constrains* the approach to subroutine design and use

```
• • •  
LDA X      /PREPARE INPUT DATA  
STA PM1    / USING STACK  
LDA Y      /   BASED  
STA PM2    /   STRUCTURE  
BSA SUB    /JUMP TO SUBROUTINE  
RTV, DEC 0  /RETURN VALUE  
PM1, DEC 0  /FIRST PARAMETER VALUE  
PM2, DEC 0  /NEXT PARAMETER VALUE  
RPA, LDA RTV /RETURN POINT ADDRESS  
• • •
```

Be sure to transfer all data into the data structure before calling the subroutine.

Here we use *Call by Address*.

Example 4: Subroutines (4)

- First, we consider the issues of invoking (calling) the subroutine
 - Preparing the values (parameters)
 - A responsibility of the programmer
 - For high level languages this is accomplished by the compiler
 - Passing control to the subroutine
 - Must ensure that the return address is specified
 - Mano provides this using the BSA instruction
 - BSA *constrains* the approach to subroutine design and use

```

. . .
LDA X      /PREPARE INPUT DATA
STA PM1    / USING STACK
LDA Y      /   BASED
STA PM2    /   STRUCTURE
BSA SUB     /JUMP TO SUBROUTINE
RTV, DEC 0  /RETURN VALUE
PM1, DEC 0  /FIRST PARAMETER VALUE
PM2, DEC 0  /NEXT PARAMETER VALUE
RPA, LDA RTV /RETURN POINT ADDRESS
. . .
```

Call the subroutine – pass control using BSA.

It is assumed that the programmer has correctly implemented the return procedure and has provided the return value in the proper location.

Example 4: Subroutines (5)

- Next, we consider the subroutine design issues of
 - Access to the input data
 - Processing it according to the subroutine algorithm
 - Returning the final value
 - Returning from the subroutine

BSA places the location of RTV into SUB and points the PC at the instruction at ENT.

```
    . . .  
    LDA X  
    STA PM1  
    LDA Y  
    STA PM2  
    BSA SUB  
RTV, DEC 0  
PM1, DEC 0  
PM2, DEC 0  
RPA, LDA RTV  
    . . .
```

```
SUB, HEX 0      /RETURN VALUE LOCATION (RTV)  
ENT, LDA SUB    /LOAD RETURN VALUE LOCATION  
STA RVA        /STORE RETURN VALUE LOC.  
ISZ SUB        /INCREMENT POINTER TO PM1  
LDA SUB I      /FETCH PM1  
ISZ SUB        /INCREMENT POINTER TO PM2  
ADD SUB I      /FETCH AND ADD PM2  
STA RVA I      /STORE TO RET. VAL. LOC.  
ISZ SUB        /INCREMENT POINTER TO RPA  
BUN SUB I      /RETURN TO CALLING POINT  
RVA, HEX 0      /RETURN VALUE LOCATION
```

Example 4: Subroutines (6)

- Next, we consider the subroutine design issues of
 - Access to the input data
 - Processing it according to the subroutine algorithm
 - Returning the final value
 - Returning from the subroutine

It is often a good idea to store the return address locally.

```
. . .  
LDA X  
STA PM1  
LDA Y  
STA PM2  
BSA SUB  
RTV, DEC 0  
PM1, DEC 0  
PM2, DEC 0  
RPA, LDA RTV  
. . .
```

```
SUB, HEX 0 /RETURN VALUE LOCATION (RTV)  
ENT, LDA SUB /LOAD RETURN VALUE LOCATION  
STA RVA /STORE RETURN VALUE LOC.  
ISZ SUB /INCREMENT POINTER TO PM1  
LDA SUB I /FETCH PM1  
ISZ SUB /INCREMENT POINTER TO PM2  
ADD SUB I /FETCH AND ADD PM2  
STA RVA I /STORE TO RET. VAL. LOC.  
ISZ SUB /INCREMENT POINTER TO RPA  
BUN SUB I /RETURN TO CALLING POINT  
RVA, HEX 0 /RETURN VALUE LOCATION
```

Example 4: Subroutines (7)

- Next, we consider the subroutine design issues of
 - Access to the input data
 - Processing it according to the subroutine algorithm
 - Returning the final value
 - Returning from the subroutine

Note how the input data values are accessed indirectly by incrementing the pointer through the data structure. The return value is placed in the proper return location, RTV.

```
. . .  
LDA X  
STA PM1  
LDA Y  
STA PM2  
BSA SUB
```

```
RTV, DEC 0  
PM1, DEC 0  
PM2, DEC 0  
RPA, LDA RTV  
. . .
```

```
SUB, HEX 0      /RETURN VALUE LOCATION (RTV)  
ENT, LDA SUB    /LOAD RETURN VALUE LOCATION  
STA RVA         /STORE RETURN VALUE LOC.  
ISZ SUB         /INCREMENT POINTER TO PM1  
LDA SUB I       /FETCH PM1  
ISZ SUB         /INCREMENT POINTER TO PM2  
ADD SUB I       /FETCH AND ADD PM2  
STA RVA I       /STORE TO RET. VAL. LOC.  
ISZ SUB         /INCREMENT POINTER TO RPA  
BUN SUB I       /RETURN TO CALLING POINT  
RVA, HEX 0      /RETURN VALUE LOCATION
```

Example 4: Subroutines (8)

- Next, we consider the subroutine design issues of

- Access to the input data
- Processing it according to the subroutine algorithm
- Returning the final value
- Returning from the subro

Finally, the return is implemented using the modified address provided by BSA.

Note that this requires indirect addressing.

```
      . . .  
      LDA X  
      STA PM1  
      LDA Y  
      STA PM2  
      BSA SUB  
RTV, DEC 0  
PM1, DEC 0  
PM2, DEC 0  
RPA, LDA RTV  
      . . .
```

```
SUB, HEX 0      /RETURN VALUE LOCATION (RTV)  
ENT, LDA SUB    /LOAD RETURN VALUE LOCATION  
      STA RVA    /STORE RETURN VALUE LOC.  
      ISZ SUB    /INCREMENT POINTER TO PM1  
      LDA SUB I  /FETCH PM1  
      ISZ SUB    /INCREMENT POINTER TO PM2  
      ADD SUB I  /FETCH AND ADD PM2  
      STA RVA I  /STORE TO RET. VAL. LOC.  
      ISZ SUB    /INCREMENT POINTER TO RPA  
      BUN SUB I  /RETURN TO CALLING POINT  
RVA, HEX 0      /RETURN VALUE LOCATION
```

Example 4: Subroutines (9)

- **When designing compilers it is vital to define very precisely the convention (protocol) for creating proper subroutine calls and data exchange mechanisms and structures**
 - **Creates transparency of implementation and permits programmers to concentrate on high level expression of logical intention**
- **When programming at the assembly language level, all members of the programming team MUST know the linkage conventions**
 - **Otherwise, chaos results**
 - **Very difficult to debug**
- **Stacks are often used to implement data exchange**
 - **Use of *stack frames* to hold data being passed, as well as *process state* information (CPU registers)**
 - **Can be used to implement re-entrancy and shared usage to support recursive subroutine calls.**

Example 5: Input/Output (0)

- **General purpose computers support a wide variety of devices**
- **At a basic level one expects to have:**
 - **Keyboard input (STDIN)**
 - C language: `scanf(“%c”, &charbuff) ;`
 - **Terminal (Printer) output (STDOUT)**
 - C language: `printf(“%c”, charbuff) ;`
- **Mano’s machine supports basic I/O of 8-bit data values.**
 - **This automatically supports the ASCII data set.**

Example 5: Input/Output (1)

- **We concentrate on the principal issues of performing I/O using Mano's instruction set**
- **We defer discussion of interrupts until the next Example**
- **We consider the problem of inputting an Assembly Language program (source code) and storing it in memory**
 - **This would be a necessary first step in writing an Assembler program using the instruction set itself.**

Example 5: Input/Output (2)

• Def

```
CIF, HEX 0      /RETURN VALUE
CF0, SKI        /WAIT FOR INPUT
BUN CF0         /FGI=0, LISTEN AGAIN
INP             /FGI=1, INPUT CHAR
OUT            /OUTPUT CHAR
STA CIF I      /STORE RETURN VALUE
ISZ CIF        /OBTAIN RETURN ADDRESS
BUN CIF I      /RETURN
```

er from device

acter passed

```
BSA CIF
CHR, HEX 0
RCI, LDA CHR
```

```
LDA CH1
STA CHR
BSA COF
CHR, HEX 0
RCO, continue
```

```
COF, HEX 0      /RETURN ADDRESS
CO0, SKO        /WAIT FOR OUTPUT CHANNEL
BUN CO0         /FGO=0, LISTEN AGAIN
OUT            /FGO=1, OUTPUT CHAR
BUN COF I      /RETURN
```

Example 5: Input/Output (3)

- The program obtains an 8-bit input character from the keyboard.
- Since two characters are to be output (or output characters) – Use two calls to the program.

```

CI2,  HEX 0      /RETURN VALUE
CF0,  SKI        /WAIT FOR INPUT
      BUN CF0     /FGI=0, LISTEN AGAIN
      INP         /FGI=1, INPUT CHAR
      OUT         /OUTPUT CHAR
      BSA SL4     /SHIFT LEFT 4 BITS
      BSA SL4     /SHIFT LEFT 4 BITS
CF1,  SKI        /WAIT FOR INPUT
      BUN CF1     /FGI=0, LISTEN AGAIN
      INP         /FGI=1, INPUT CHAR
      OUT         /OUTPUT CHAR
      BUN CI2 I   /RETURN
    
```

Note that AC is ONLY modified in bits 0-7 (low order) by INP.

```

SL4,  HEX 0      /RETURN ADDRESS
      CIL
      CIL
      CIL
      CIL
      AND ML4
      BUN SL4 I   /RETURN
ML4,  HEX FFF0
    
```

Requires that AC is NOT modified by the call to SL4.

Example 5: Input/Output (4)

- **Mano provides several additional examples to illustrate character processing**
 - **Simple string manipulation**

Example 6: Interrupt Handling (0)

- **The needs for interrupts arise in different contexts**
 - **Trapping exceptions from instructions**
 - Integer addition/subtraction overflow
 - Divide by zero
 - Addressing/Protection
 - **Responding to events (event handling)**
 - Clock cycles for scheduled O/S operations
 - Listening to communications ports
 - Communicating with Input/Output devices
- **Mano's machine implements an interrupt capability to enable asynchronous I/O**
 - **The basic idea can be extended to handling of a wide variety of interrupts**

Example 6: Interrupt Handling (1)

Time scale differences

Between a human typist entering data (1 keystroke per byte) and CPU processing of the input

In one second, the typist enters about 10 bytes of data while the CPU executes several hundreds of millions of instructions.

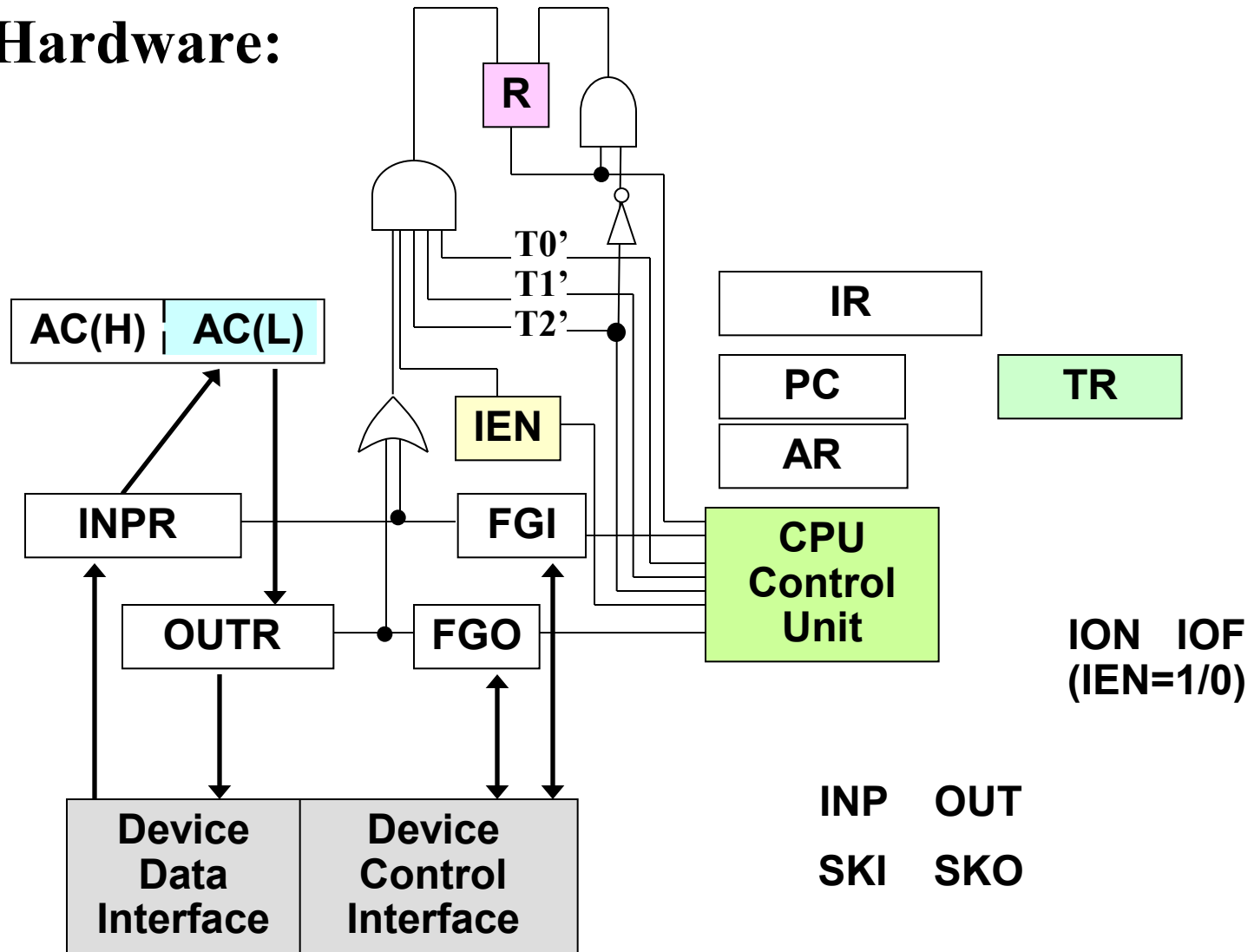
This means that waiting for I/O before using the CPU wastes considerable (ie. Huge) amounts of potential instruction cycles.

Interrupts are special device interface circuits designed to permit CPU processing of instructions to proceed while anticipated I/O events are still pending.

When the events occur, the interrupt triggers and enables immediate (and rapid) handling of the I/O data and subsequent return to the interrupted process, with its interrupted state fully restored.

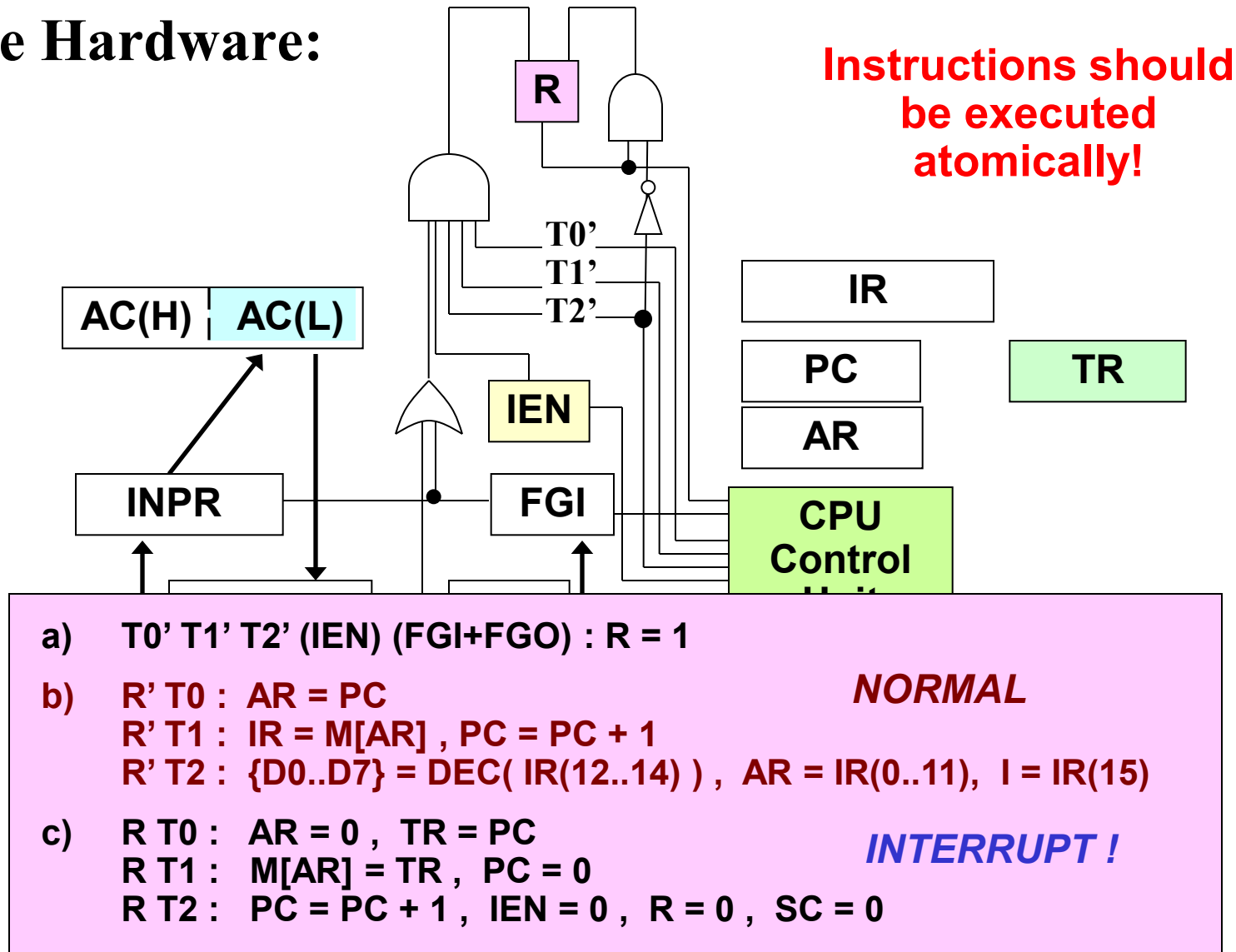
Example 6: Interrupt Handling (2)

- **The Hardware:**



Example 6: Interrupt Handling (3)

- The Hardware:



Example 6: Interrupt Handling (4)

- **When an I/O interrupt is performed, it is necessary to**
 - **Save the contents of CPU registers**
 - To enable recovery of the original state at the time of interrupt
 - Include
 - **Check whether the interrupt is for**
 - FGI or FGO
 - **Service the device whose flag is set**
 - Input (FGI) or Output (FGO)
 - **Before returning from the interrupt handling routine, restore the contents of the CPU registers (AC, E)**
 - **Turn the interrupt facility on**
 - IEN
 - **Return to the running program**

**Study Mano's example in
Table 6-23, page 207.**

Example 6: Interrupt Handling (5)

- **In Mano's machine there is a single interrupt flip-flop, R, and it is controlled directly by the I/O circuits (black boxes), and the limited ION and IOF instructions.**
- **To increase the number of recognizable, specific interrupts, it is necessary to impose an enumeration scheme**
 - **Each unique interrupt type must be assigned a code value, in contiguous sequence, starting at 0**
 - **Code is used as selector input to interrupt control circuitry**
- **The value of the interrupt code may be used to directly access an Interrupt Vector, located at address 0.**
 - **This vector consists of data structures that provide access to interrupt handling routines (functions)**
 - **May return control to the user program**
 - **Or, may default back to operating system control and abort the user program.**

Summary

- We discussed programming of Mano's model of computer architecture (Mano – Chapter 6)
- Assembly language
- Assembler
 - One-pass
 - Two-pass
- Language extensions
- Example assembly language programs
- **Towards conclusion**
 - A glimpse at future topics to study and consider in Digital Design and Computer Architecture

End of Formal Lectures



Removing the mystery of *magic* should not spoil the enjoyment or marvel at its display.

Into the Future

- **Future topics in**
 - **Digital Design**
 - **Minimization of logic circuit complexity (Quine-McCluskey-Petrick)**
 - **Computer Architecture**
 - **Expanding memory address space**
 - **Expanding CPU register set**
 - **Extending the instruction set, number of operands, addressing modes**
 - **Multiprocessors**
 - **Multi-core processing (inline channel parallelism)**
 - **SMP – Symmetric Multi-Processing**
 - **Multiple CPUs, Memories**
 - **Complex bus architectures**
 - **Extending Input and Output capabilities**



Summary of Goals

- **Course: 60-265 Digital Design & Computer Architecture**
- **Concepts**
 - Design, Integration, Boolean Algebra/Calculus, Optimization, Hardware constraints, Control, Machine language
- **Skills**
 - Technical : Circuit design, circuit optimization, comprehension of machine specifications
 - Analytical : Boolean Algebra, intricacies of circuits
- **Understanding**
 - Computer and Information Scientists should understand the precise nature and limitations of the machines used to run software and be guided in this knowledge
 - The concepts and techniques of Boolean Logic used in this context also apply in every aspect of information processing.
- **Application**
 - Group Project invested significant time and skill in developing a complete computer design architecture for a challenging, complete machine specification.
- **Professional Development**
 - Individual & Team qualities were challenged and improved through interaction that emphasized consensus building, decision making, discipline, project and team management, and deliverables.

Postscript

- I hope
 - You have enjoyed the lectures
 - What you have learned will provide a stronger foundation for deeper understanding.
- Best of luck with continuation of your academic program
 - Study hard – make your own luck !
- See you at the Final Examination.