CEG2136
Computer Architecture I
Basic Computer Programming

uOttawa

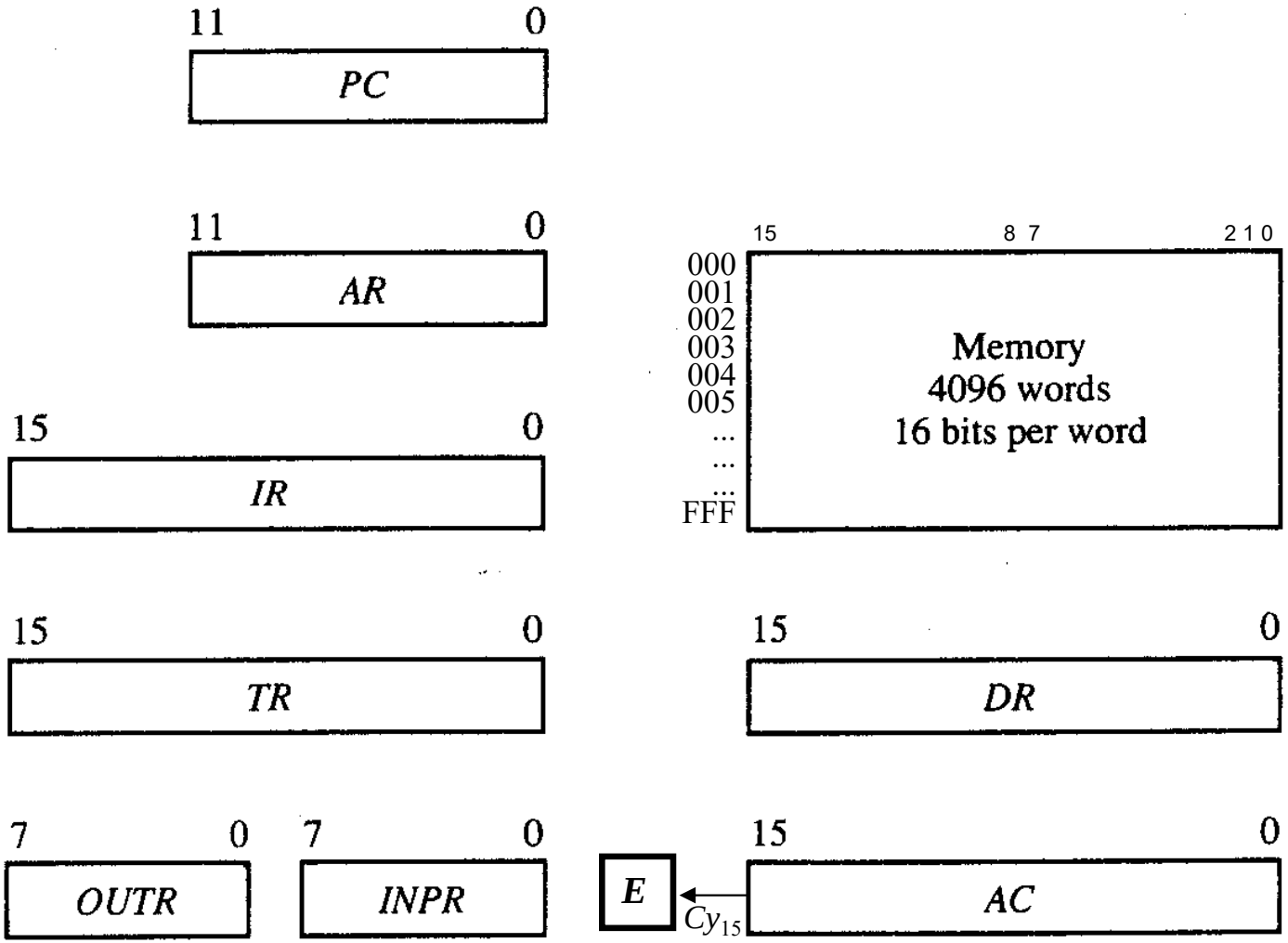
L'Université canadienne
Canada's university

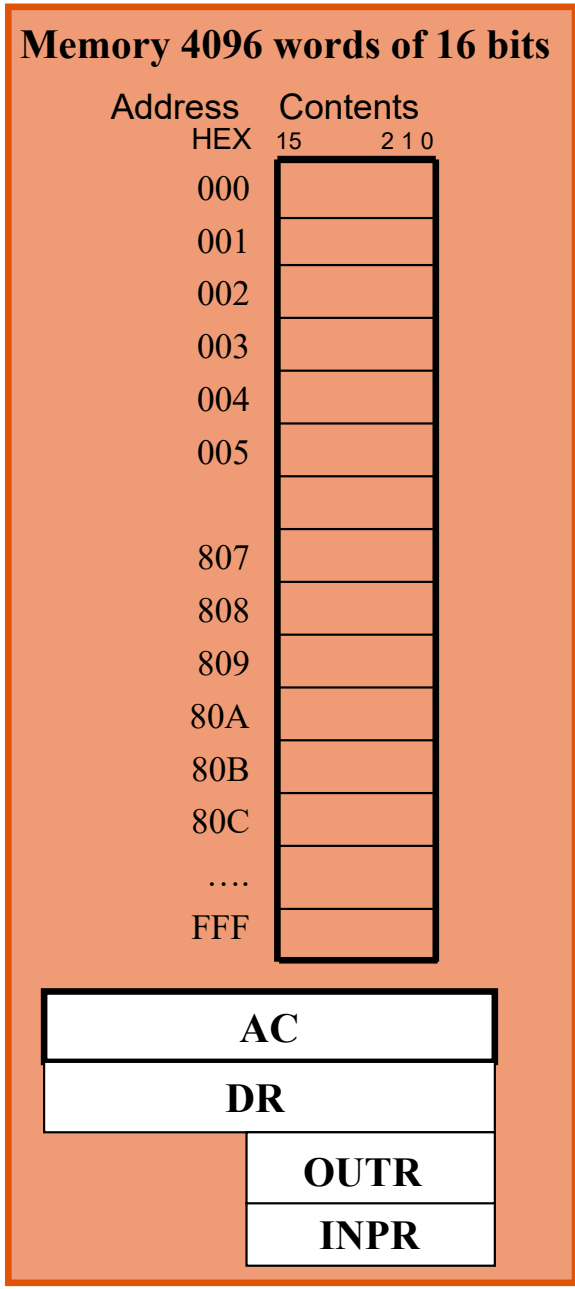
Voicu Groza
SITE Hall, Room 5017
562 5800 ext. 2159
VGroza@uOttawa.ca

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

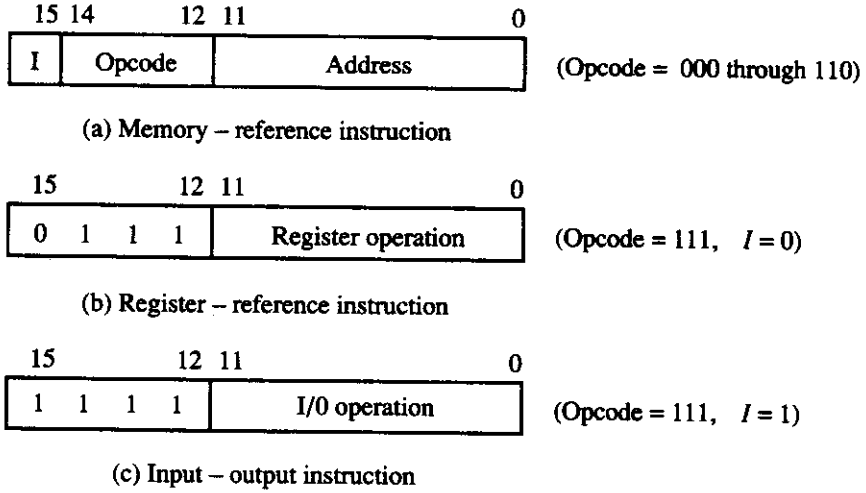
Basic Computer Registers and Memory





Basic Computer Instruction List (Hex)

In this chapter, we will use the same 25 instructions of the basic computer designed in Chapter 5.
the symbol *m* is used to used to denote the effective address,
the symbol *M* is used to refer to the memory word located at the effective address *m*.
Instruction word format:



Symbol	Hex code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m + 1
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC,
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

Machine Language

- Computer programs may be written in several programming languages.
- These programming languages can be divided into :
  1. *Machine language.* It is a sequence of instructions and operands represented in a Binary, Octal or hexadecimal code reflecting the exact content of the computer memory.
  2. *Assembly language* (Symbolic code). A sequence of instructions where each instruction is represented with one or more symbolic codes. A **compiler** is then necessary to translate each instruction into **one** binary-coded instruction (in machine language) to be executed by the computer.
  3. *High-level programming languages.* These are programming languages in which each instruction may be translated by the compiler into a sequence of binary-coded instructions. C, Java, and Fortran, are examples of such a type of programming languages.

Program to Add Two Numbers

Machine code			Symbolic program			Assembly Languages	
Address	Memory Content (Instruction binary code)	Instr hex code	Address	Instr. symbol opcode	Comments	**Instruction** / Comments	
						ORG 0 /Origin of program is location 0	
000	0010 0000 0000 0100	2004	000	LDA 4	Load first operand into AC	**LDA X** /Load **x** from location **X**	AC ← M[4]
001	0001 0000 0000 0101	1005	001	ADD 5	Add second operand to AC	**ADD Y** /Add **y** from location **Y**	AC←AC+M[5]
002	0011 0000 0000 0110	3006	002	STA 6	Store sum in location 006	**STA Z** /Store **z=x+y** at **Z**	M[6]←[AC]
003	0111 0000 0000 0001	7001	003	HLT	Halt computer	**HLT** /Halt computer	
004	0000 0000 0101 0011	0053	004	0053	First operand	**X, DEC 83** /Decimal operand **x**=83	M[4]
005	1111 1111 1110 1001	FFE9	005	FFE9	Second operand (negative)	**Y, DEC-23** /Decimal operand **y**=-23	M[5]
006	0000 0000 0000 0000	0000	006	0	Store sum here	**Z, DEC 0** / Sum **z** is stored here	M[6]
						END /End of symbolic program	
Table	6.2	6.3			6.4	6.5	

In assembly language, a *label* is a *symbolic address* which represents a *physical memory address*: e.g., label **X** represents *physical memory address X*=004 which refers to the value **x** stored at memory location M[4], i.e., M[X] = **x** ➔ M[4] = 83

Rules of the Assembly Language

- Each line of an assembly program is arranged in three columns (fields):
  [label,] instruction [/comment]

1. The *label* field may be empty or may specify a symbolic address.
2. The *instruction* field specifies a machine instruction or a pseudo-instruction.
3. The *comment* field may be empty or may include a comment.

- A symbolic address, in the label field, consists of one letter followed by no, one, or two, alphanumeric characters.
- The instruction field specifies one of the following:

1. A memory-reference instruction (MRI) consists of two or three symbols separated by spaces: opcode-symbol, symbolic-address and [I]
   - The opcode-symbol is a three-letter symbol defining an MRI operation code.
   - The second code is a symbolic address.
   - The third symbol is optional. If present, it is the letter “I".
- The presence of “I" in an MRI denotes an indirect address instruction, whereas its absence denotes a direct address instruction.

2. A register-reference or input-output instruction (non-MRI)
3. A pseudo-instruction with or without an operand

Pseudo-instructions

- A pseudo-instruction is not a machine instruction, and hence is not translated into a machine-coded instruction by the assembler.
- A pseudo-instruction is rather an instruction that provides the assembler with some type of information about a particular instruction in the assembly code.
- We will assume that the assembler defined in this chapter supports the four pseudo-instructions listed in Table 32.

Table 32: Definitions of pseudo-instructions

Symbol	Information for the assembler
ORG N	Hexadecimal number N is the memory location of the instruction or the operand listed in the following line. ☐ It informs the assembler that the instruction or operand in the following line is placed in the memory location specified by the *hexadecimal* number following ORG. ☐ It is possible to use ORG more than once in a program to specify more than one segment of memory.
END	Denotes the end of the symbolic program and it is placed to inform the assembler that the program is terminated
DEC N	Signed decimal number N to be converted to binary
HEX N	Signed hexadecimal number N to be converted to binary

Assembly Language Program to Subtract Two Numbers

- In order for a program to be executed by any computer it has to be converted into that computer's specific machine code by an appropriate compiler.
- An assembler is a compiler that translates each assembly instruction into its equivalent binary instruction.
- The translation of the assembly program for subtracting two numbers (presented in the previous example) into its equivalent machine code is shown in the following table.

	Assembly Language (source code)		**Address** (hex)	**Content** (hex)	Run time AC
	ORG 100	/Origin of program is location 100			
	LDA SUB	/Load subtrahend to AC	100	2107	0017
	CMA	/Complement AC	101	7200	FFE8
	INC	/Increment AC	102	7020	FFE9
	ADD MIN	/Add minuend to AC	103	1106	003C
	STA DIF	/Store difference	104	3108	003C
	HLT	/Halt computer	105	7001	003C
MIN,	DEC 83	/Minuend	106	0053	
SUB,	DEC 23	/Subtrahend	107	0017	
DIF,	HEX 0	/Difference stored here	108	0000	
	END	/End of symbolic program			
Table		6.8		6.9	

Compilation Passes

- The compilation of an assembly program into a machine code by the assembler is accomplished in two *passes* (program scans).
- During the *first pass*:
  1. a memory location (address) is assigned to each instruction and operand (ORG and END are not assigned any location as they are pseudo-instructions).
  2. an address symbol table defining the hexadecimal address value for each symbolic address is formed.
  - For instance, the address symbol table for the program of the previous example is as follows:

Address symbol	Hexadecimal address
MIN	106
SUB	107
DIF	108

  - No instruction code is derived during the first pass.
- During the *second pass* of the assembly program:
  - The address symbol table formed in the first pass is used to determine the address value of each instruction.
  - The complete instruction code for each instruction is derived.
- Example: the instruction LDA SUB is translated during the second pass into a binary code by:
  - getting the hexadecimal opcode of the operation LDA from Table 31 ⇒ 2,
  - determining the hexadecimal value of the label SUB from the address symbol table formed during the first pass ⇒ 107, and
  - reset bit 15 of the instruction code to 0 since the instruction is a direct address MRI (letter “I” is missing)
  - The complete hexadecimal instruction code is then formed by “assembling” the above segments: 2107

Program Loops

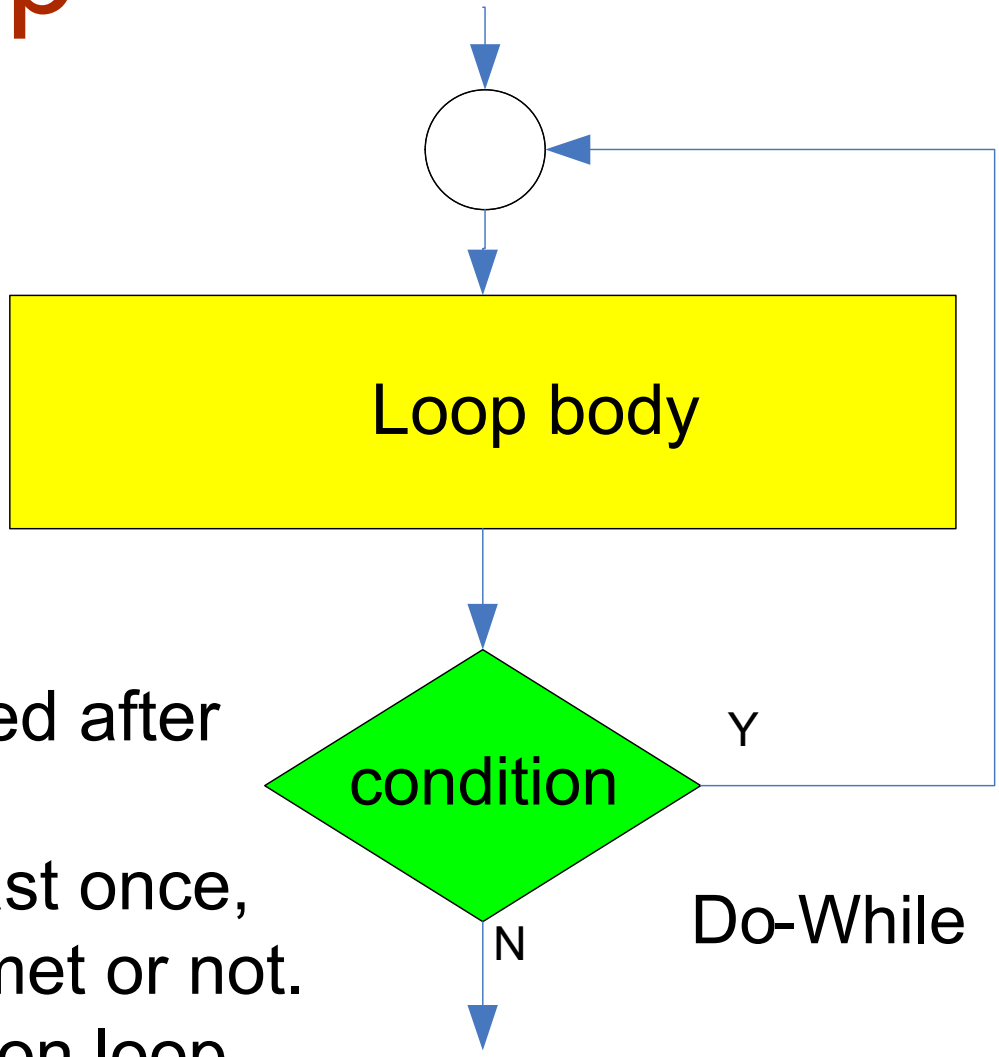
A **loop** is a programming structure that is used to repeat a sequence of instructions (loop body) until a specified condition is met.

There are several types of loops

1. DO-WHILE Loop
2. WHILE Loop
3. FOR Loop

DO-WHILE Loop

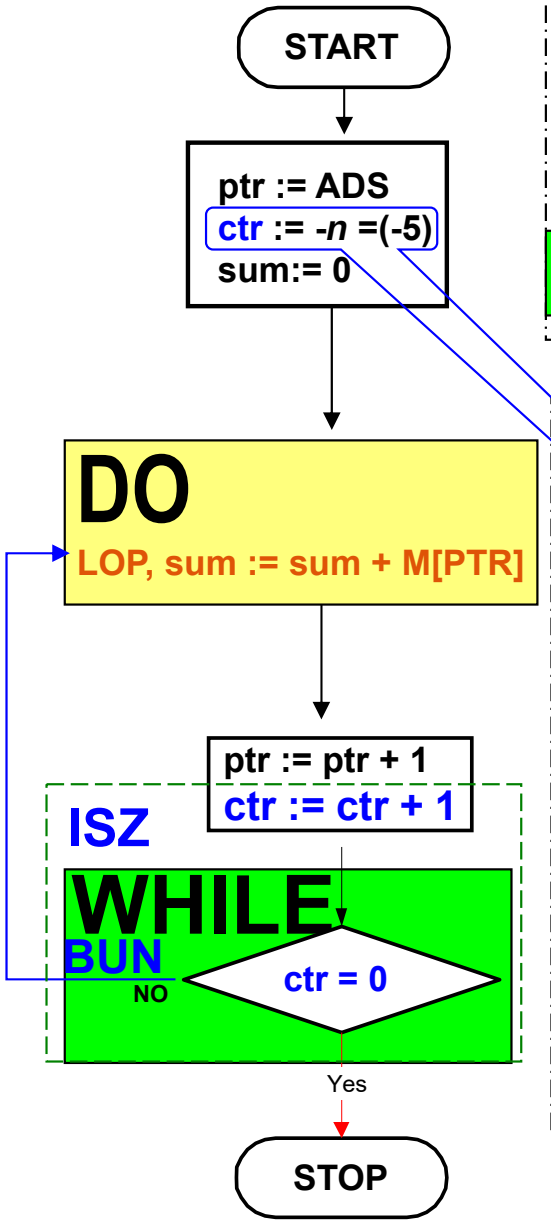
```
DO
    Loop body
ENDO
WHILE condition
```



Do-While

- The looping condition is tested after executing the loop body
- Loop body is executed at least once, no matter if the condition is met or not.
- Alternate names: exit-condition loop, exit-controlled loop, post-test loop.

Loops DO-WHILE in Assembly Language




```
DO          .... loop body
                ....
            Increment ctr
ENDO
WHILE ctr < 0 /condition
```

In Assembly Language, a variable, called counter (**ctr**) is stored at label **CTR** and keeps track of the number of iterations that were executed. Here the label **CTR** = 10E represents the physical address of the memory location where **ctr** is stored:

addr	MACHINE CODE memory	ASSEMBLY	
104	...		...
105	...	LOP	...
106	...		...
107	610E		ISZ CTR
108	4105		BUN LOP
109	7001		HLT
...	...		...
10E	FFFB	CTR	equ -5 / ctr

if ctr = 0

ctr = M[**CTR**]
FFFB = M[10E]
If a sequence of instructions is to be executed *n* times, the counter is initialized with –*n* and then incremented by 1 at every pass through the loop, until it reaches 0.

Matrices

A 1-dimension matrix with *nbr* = 100 elements (a_0, a_1, a_2,.. a_k,.. a_{99}) is stored in the memory starting at address ADS=HEX150; this *nbr* constant is stored at memory location labeled NBR. Write a program to find the sum of these numbers and store it in memory at label SUM: ***sum*** = $\sum_{k=0}^{99} a_k$

Elements of matrices or series of numbers (say a_0, a_1, a_2,… a_k, …) are stored at consecutive memory locations, starting with an initial address (say *ads*). In assembly language we do not use (a_0, a_1, a_2,… a_k, …) – like in high level languages, but rather we refer to them by the contents of the corresponding memory locations where they are stored:

M[*ads*]= a_0, M[*ads*+1]= a_1, M[*ads*+2] = a_2, … M[*ads*+k] = a_k, ….

A **pointer** variable (say ***ptr***) is stored in memory at location labeled PTR and carries the **address** {(*ads*+k), k=0,1,2, …} of the memory location where the element (a_k) that we currently refer to is stored and it goes through all values: ***ptr*** = {*ads*, *ads*+1, *ads*+2, … *ads*+k, …}. Again, the **pointer** carries at some moment the value ***ptr*** = *ads*+k = M[PTR], which is the address of the memory location M[*ads*+k] where the value of a_k is stored (a_k = M[*ads* +k]).

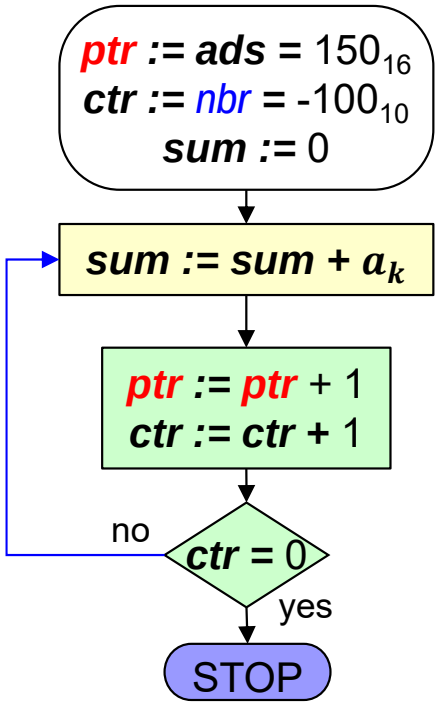
ptr = M[PTR]
ads = M[ADS]

Addr	Memory	Label,	operand	Comments:
10B	0150	ADS,	***ads***	/ M[ADS] = *ads* =150_{16} = *address of* a_0
10C	0000	PTR,	*ptr*	/ M[PTR] location is reserved for *ptr*; *ptr*'s **value** / is the **address** of the current element a_k
10D	FF9C	NBR,	*nbr*	/M[NBR]= *nbr* = -100_{10}= number of elements
10E				
10F	0000	SUM,	HEX 0	/ the final *sum* (= $\sum_{k=0}^{99} a_k$) is stored here
			ORG 150	/ Origin of operands string is HEX 150
150	004B	*ads,*	a_0	/ a_0 = 1st number, say $a_0 = 75_{10} = 4B_{16}$
151			…	…
150+*k*			a_k	/ a_k
			…	…
1B4	0017		a_{99}	/ a_{99} = last number, e.g., $a_{99} = 23_{10} = 17_{16}$

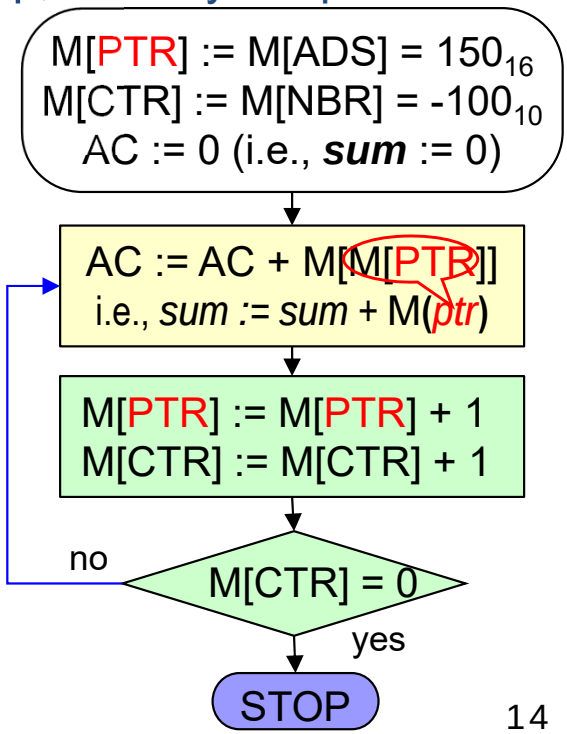
DO-WHILE Program Loop

A 1-dimension matrix with *nbr* = 100 elements (a_0, a_1, a_2,.. a_k,.. a_{99}) is stored in the memory starting at address ADS=HEX150; this *nbr* constant is stored at memory location labeled NBR. Write a program to find the sum of these numbers and store it in memory at label SUM: ***sum*** = $\sum_{k=0}^{99} a_k$

Loops are used to implement iterative processes. The indirect addressing mode was actually conceived to handle elements of matrices or series of numbers (say a_0, a_1, a_2,… a_k, …) which are stored at consecutive memory locations, starting with an initial address (say *ads*). The current iteration of a loop may deal with an element a_k = M[ADS+k] whose address ADS+k is given by the **pointer *ptr*** stored at a memory location with **label PTR**, ***ptr*** = ***ads*** +k = M[PTR]
So, a_k= M[*ads*+k] = M[M[PTR]] which we call indirect addressing mode, as in: LDA PTR I. The benefit is that by incrementing the contents of PTR (i.e., ***ptr***), we advance to the address where the next element a_{k+1} is stored, but still the instruction we use for it is LDA PTR I (which implements AC ← M[M[PTR]]) and we do not need to change the instruction in the loop, but only the pointer…



Addr	Mem.	Label,	Op'nd	Comments:
10B	0150	ADS,	*ads*	/ M[ADS] = ***ads*** =150$_{16}$ = *address of* a_0
10C	0000	PTR,	*ptr*	/ M[PTR] location for ***ptr***; ***ptr***'s **value** is / the **address** of the current element a_k
10D	FF9C	NBR,	*nbr*	/M[NBR]= *nbr* = -100$_{10}$ = # of elements
10E	0000	CTR,	*ctr*	/ M[CTR] location for the counter *ctr*
10F	0000	SUM,	HEX 0	/ final *sum* (= $\sum_{k=0}^{99} a_k$) is stored here
			ORG 150	/ Origin of operands string is HEX 150
150	004B	*ads*,	a_0	/ a_0 = 1st number, say a_0 = 75$_{10}$ = 4B$_{16}$
151			…	…
150+*k*			a_k	/ a_k
			…	…
1B4	0017		a_{99}	/ a_{99} = last #, e.g., a_{99} = 23$_{10}$ = 17$_{16}$



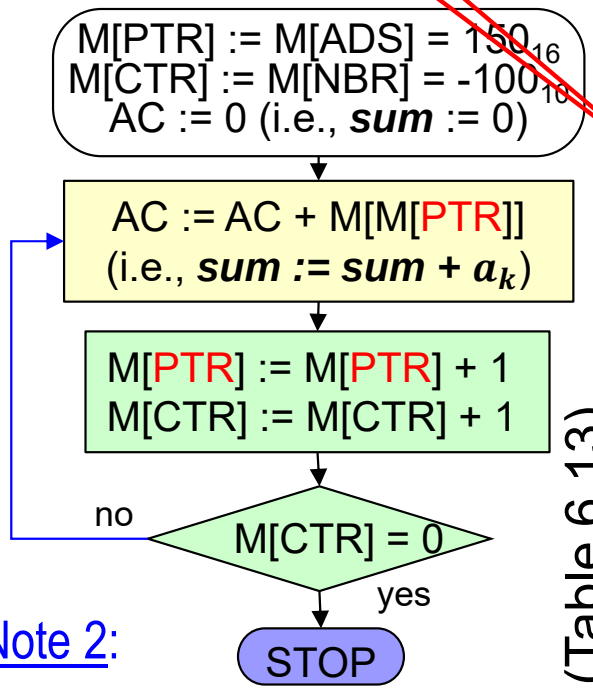
ptr = M[PTR] ; ***ads*** = M[ADS] ; ***ctr*** = M[CTR]; *nbr* = M[NBR] ; ***sum*** calculated in AC

DO-WHILE Loop

A 1-dimension matrix with 100 elements $(a_0, a_1, a_2, .. a_k, .. a_{99})$ is stored in the memory starting at address HEX150. Write a program to find the sum of these 100 numbers and store it into a word in the memory.

SUM = $\sum_{k=0}^{99} a_k$

Note1: labels represent addresses of corresponding memory locations; here, e.g., NBR = 10D is the address of memory location M[NBR] where *nbr* is stored & whose contents is FF9C = -100_{10} (*nbr* ≠ NBR, but *nbr* =M[NBR] i.e., FF9C ≠ 10D, but FF9C=M[10D])



(Table 6.13)

Memory contents before running the program

Add					
				ORG 100	/ Origin of program is HEX 100
100	210B	2		LDA ADS	/ AC←M[ADS] = *ads* := 150_{16}, i.e., address of a_0
101	310C	3		STA PTR	/ to initialize *ptr*=M[PTR] with a_0's address 150_{16}
102	210D	4		LDA NBR	/ Load AC with *nbr* = – 100_{10} = - DEC 100 and
103	310E	5		STA CTR	/ store it in counter: *ctr* = M[CTR] = FF9C
104	7800	6		CLA	/ Clear accumulator / initialize *sum* := 0
105	910C	7	LOP,	ADD PTR I	/ Add number a_k to [AC]: ***sum*** := ***sum*** + a_k
106	610C	8		ISZ PTR	/ Increment ***ptr***: M[PTR]:=M[PTR]+1
107	610E	9		ISZ CTR	/ Increment ***ctr***: M[CTR] := M[CTR]+1
108	4105	10		BUN LOP	/ Repeat loop again till M[CTR] = 0 (***ctr* = 0**)
109	310F	11		STA SUM	/Store final AC = $\sum_{k=0}^{99} a_k$ in memory @ SUM
10A	7001	12		HLT	/ STOP program
10B	0150	13	ADS,	HEX 150	/ M[ADS] *ads* =150_{16} =*address of* a_0
10C	0000	14	PTR,	HEX 0	/ M[PTR] location is reserved for *ptr*, *ptr*'s **value**
					/ is the **address** of the current element a_k
10D	FF9C	15	NBR,	**DEC -100**	/M[NBR] = *nbr* = -100_{10}= constant to initialize *ctr*
10E	0000	16	CTR,	HEX 0	/ M[CTR] = *ctr* location reserved for the counter
10F	0000	17	SUM,	HEX 0	/ the final *sum* (= $\sum_{k=0}^{99} a_k$) is stored here
		18		ORG 150	/ Origin of operands series is HEX 150
150	004B	19		DEC 75	/ a_0 = 1st number, say a_0 = 75
151		20	151	…	…
		19+k	150+k	…	/ a_k
		…		…	…
1B4	0017	118		DEC 23	/ a_{99} = last number, e.g., a_{99} = 23
		119		END	/End of symbolic program

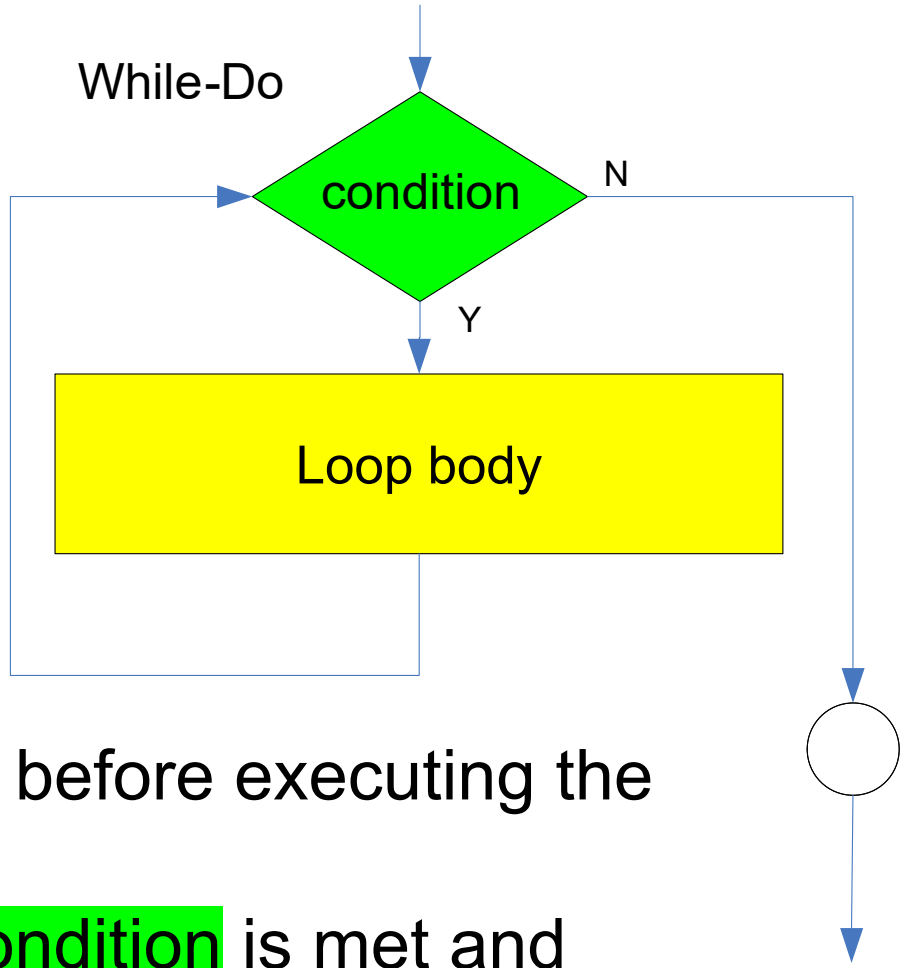
if CTR = 0

ptr's address

Note 2:

This program can run with a different number of elements, stored at different locations by changing NBR & ADS respectively

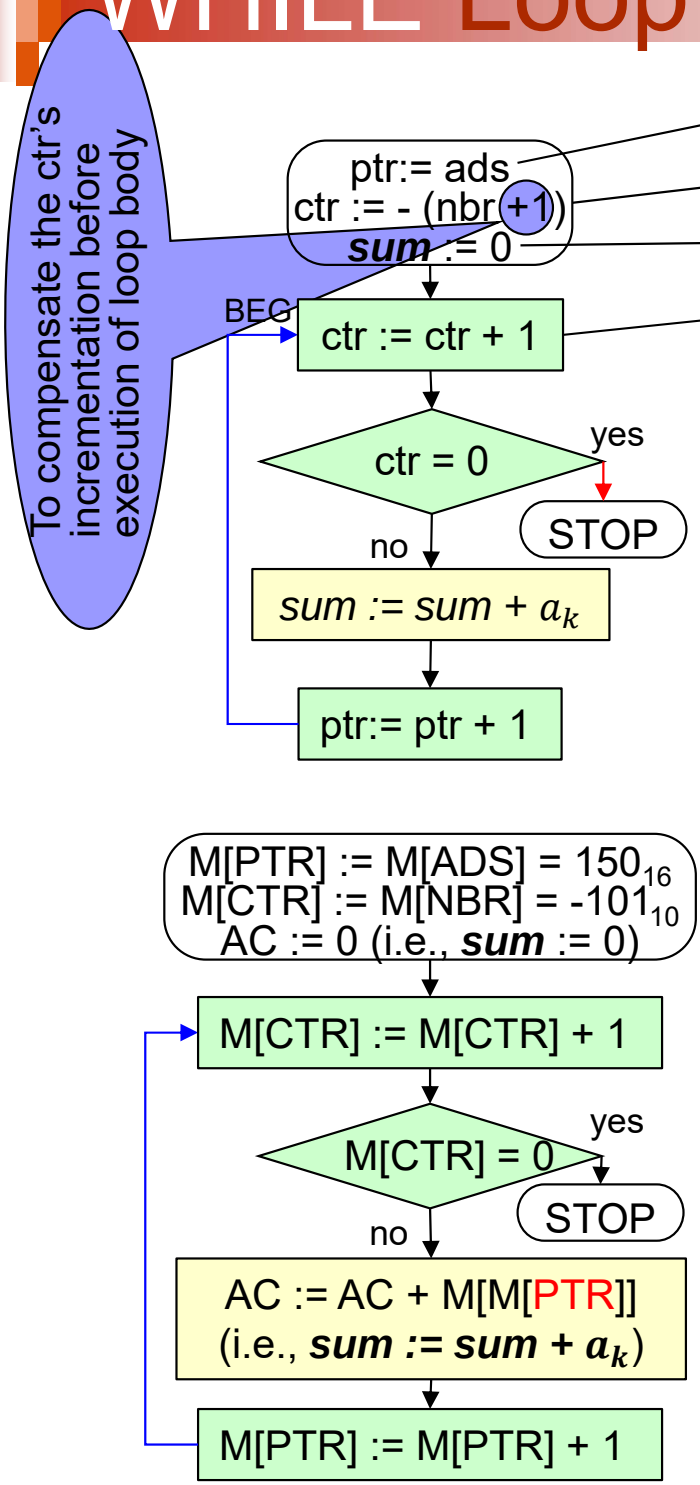
WHILE-DO Loop



WHILE condition
DO
    Loop body
ENDO

- The looping condition is tested before executing the loop body
- Loop body is executed if the condition is met and terminates otherwise; subsequently, if no needed, the body might not be executed at all.
- Alternate names: entry-condition loop, entry-controlled loop, pre-test loop.

WHILE Loop



Addr				
	1		ORG 100	/ Origin of program is HEX 100
100	2		LDA ADS	/ AC←M[ADS] = *ads* := 150_{16}, i.e., address of a_0
101	3		STA PTR	/ initialize *ptr* =M[PTR] =*ads* = 150_{16}, a_0's address
102	4		LDA NBR	/ Load AC with *nbr* = – 101_{10} = - DEC 100 and
103	5		STA CTR	/ store it in counter: *ctr* = M[CTR] = FF9C
104	6		CLA	/ Clear accumulator / initialize *sum* := 0
105	7	BEG	ISZ CTR	/ Increment ***ctr***: M[CTR] := M[CTR]+1
106	8		BUN LOP	/ ctr ≠ 0 ➔ start the loop
107	9		STA SUM	/ Store final AC = $\sum_{k=0}^{99} a_k$ in memory @ SUM
108	10		HLT	/ stop the program since ctr = 0
109	11	LOP,	ADD PTR I	/ Add number a_k to [AC]: ***sum*** := ***sum*** + a_k
10A	12		ISZ PTR	/ Increment ***ptr***: M[PTR]:=M[PTR]+1
10B	13		BUN BEG	
10C	14	ADS,	HEX 150	/ M[ADS] = *ads* =150_{16} =*address of* a_0
10D	15	PTR,	HEX 0	/***ptr*** = M[PTR] location reserved for ***ptr's* value**
			ptr's address	/ is the **address** of the current element a_k
10E	16	NBR,	**DEC -101**	/M[NBR]= *nbr* = -101_{10}= constant to initialize *ctr*
10F	17	CTR,	HEX 0	/ M[CTR] location is reserved for the counter ***ctr***
	18	SUM,	HEX 0	/ the final *sum* (= $\sum_{k=0}^{99} a_k$) is stored here
			ORG 150	/ Origin of operands string is HEX 150
150	19		DEC 75	/ a_0 = 1st number, say a_0 = 75
151	20	151	…	…
	19+k	150+k	…	/ a_k
	…		…	…
1B4	118		DEC 23	/ a_{99} = last number, e.g., a_{99} = 23
	119		END	/End of symbolic program

Programming Arithmetic and Logic Operations

- Operations not included in the set of machine instructions must be implemented by a program.
- Operations that can be performed with one machine instruction are said to be implemented by hardware.
- Operations that are performed through a program (set of instructions) are said to be implemented by software.
- Hardware implementation is more costly than software implementation but leads to a faster execution of operations.

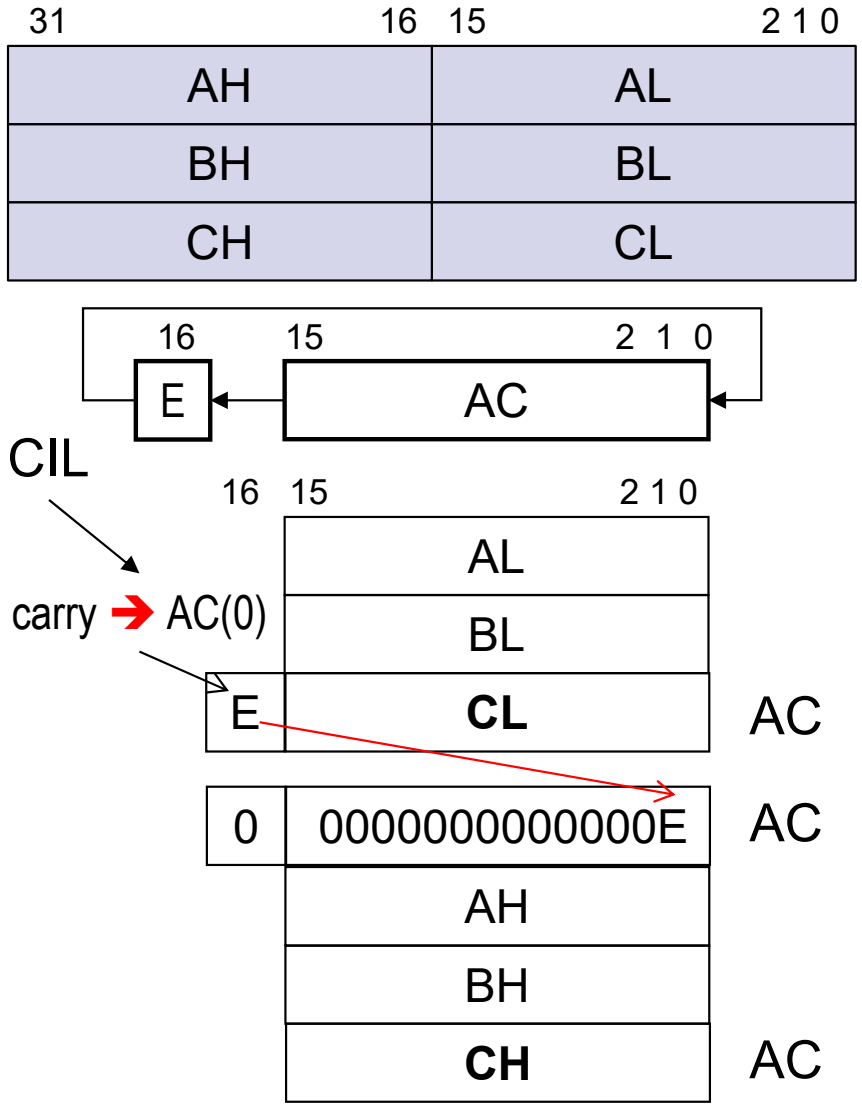
Double-Precision Addition

- An assembly program to add two double-precision numbers is shown
- The **little-endian** approach is used

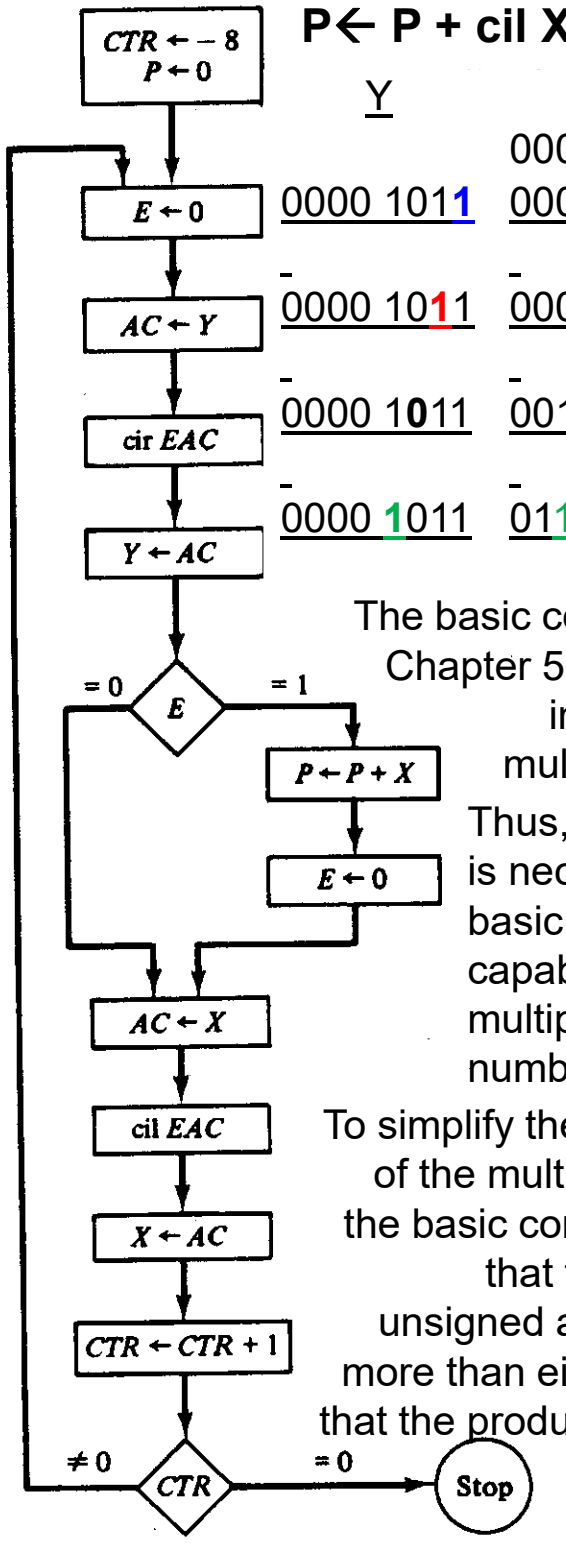
	LDA AL	/Load A low
	ADD BL	/ Add B low, carry in E
	STA CL	/Store in C low
	CLA	/Clear AC
	CIL	/Circulate to bring carry from E to AC(0)
	ADD AH	/ Add A high and carry
	ADD BH	/Add B high
	STA CH	/Store in C high
	HLT	
AL,	-	/Location of operands
AH,	-	
BL,	-	
BH,	-	
CL,	-	
CH	-	

Table 6.15

- To increase the accuracy of an operation, operands may be stored in two or more memory words rather than only one.
- A number stored in two memory words is said to have **double precision**.



Multiplication



P← P + cil X if X_i = 1, i=1,4

Y	cil X	P
	0000 1111	**0000 0000**+
0000 1011	0000 1111	0000 1111
		0000 1111+
0000 1011	0001 1110	0001 1110
		0010 1101+
0000 1011	0011 1100	0000 0000
		0010 1101+
0000 1011	0111 1000	0111 1000
		1010 0101

The basic computer designed in Chapter 5 does not contain an instruction to perform multiplication operation.

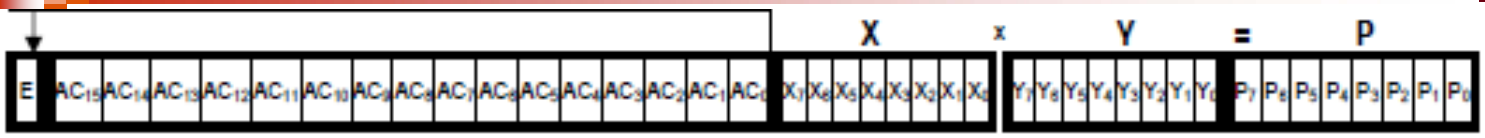
Thus, a SW implementation is necessary to provide the basic computer with the capability to perform a multiplication of two numbers.

To simplify the SW implementation of the multiplication operation in the basic computer, it is assumed that the two operands are unsigned and that they have no more than eight significant bits so that the product cannot exceed the 16-bit word capacity (**Table 6.14**).

E				A	C			

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

Multiplication



E	AC15 AC14 AC13 AC12 AC11 AC10 AC9 AC8 AC7 AC6 AC5 AC4 AC3 AC2 AC1 AC0	X7 X6 X5 X4 X3 X2 X1 X0	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	P7 P6 P5 P4 P3 P2 P1 P0				
0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	0 0 0 0 1 0 1 1	0 0 0 0 0 0 0 0	AC <-Y	CHECK Y	LDA Y	/Load multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	0 0 0 0 0 1 0 1	0 0 0 0 0 0 0 0	CIR Y <-AC		CIR STA Y	/transfer multiplier bit to E /Store shifted multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 0 1 0 1	0 0 0 0 0 0 0 0	AC <-X	ADD X	LDA X	/Load multiplicand
0	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	P <-P+X E <-0		ADD P STA P CLE	/Add to partial product /Store partial product /Clear E
0	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	AC <- X	SHIFT X	LDA X	/Load multiplicand
0	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	CIL		CIL	/Shift left
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	AC <-Y	CHECK Y	LDA Y	/Load multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	CIR Y <-AC		CIR STA Y	/transfer multiplier bit to E /Store shifted multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 0 0 1 1 1 1	AC <-X	ADD X	LDA X	/Load multiplicand
0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	P <-P+X E <-0		ADD P STA P CLE	/Add to partial product /Store partial product /Clear E
0	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0	0 0 0 1 1 1 1 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	AC <- X	SHIFT X	LDA X	/Load multiplicand
0	0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	CIL		CIL	/Shift left
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0 0 1 1 1 1 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	AC <-Y	CHECK Y	LDA Y	/Load multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	CIR Y <-AC		CIR STA Y	/transfer multiplier bit to E /Store shifted multiplier
0	0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	AC <- X	SHIFT X only	LDA X	/Load multiplicand
0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	CIL	NO ADD	CIL	/Shift left
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 0 1 0 1 1 0 1	AC <-Y	CHECK Y	LDA Y	/Load multiplier
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 1 1 1 1 0 0 0	0 0 0 0 0 0 0 0	0 0 1 0 1 1 0 1	CIR Y <-AC		CIR STA Y	/transfer multiplier bit to E /Store shifted multiplier
	0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0	0 1 1 1 1 0 0 0	0 0 0 0 0 0 0 0	1 0 1 0 0 1 0 1	AC <-X P <-P+X E <-0	ADD X	LDA X ADD P STA P CLE	/Load multiplicand /Add to partial product /Store partial product /Clear E
	0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0	1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0	1 0 1 0 0 1 0 1	AC <- X	SHIFT X	LDA X	/Load multiplicand
	0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0	1 0 1 0 0 1 0 1	CIL		CIL	/Shift left

Logic Operations

- A logic operation can also be implemented in SW in case it is not supported directly by the ALU.
- The basic computer's ALU does not have a HW implementation for an OR operation, but does have a HW implementation for AND and NOT operations.
- Since $x \vee y = (x' \wedge y')'$ (De Morgan theorem), the OR operation can be supported through a SW implementation. A symbolic program to this is shown below.

```
        LDA A       /Load first operand x, stored at address A (AC←M[A] = x)
        CMA         /Complement to get x'
        STA TMP     /Store x' in a temporary location
        LDA B       /Load second operand y, stored at address B (AC←M[B] = y)
        CMA         /Complement to get y'
        AND TMP     /Add with y' to get (x' ∧ y')
        CMA         /Complement to get (x' ∧ y')' = (x ∨ y )
        ORG FF0
A, x                / first operand x
B, y                / second operand y
TMP,
```

Shift Operations

- The basic computer has a HW implementation for a circular shift.
- A **logic shift right** operation can then be accomplished in SW by the following instructions:

```
CLE
CIR
```

- A **logic shift left** operation can be accomplished in SW by the following instructions:

```
CLE
CIL
```

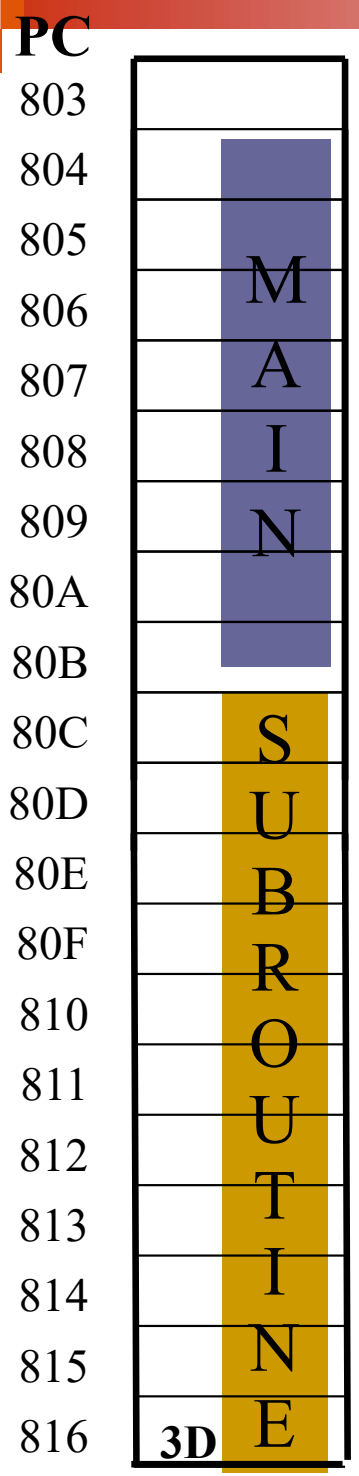
- The **arithmetic shift right** operation can be accomplished in SW by the following symbolic program:

```
CLE     /Clear E to 0
SPA     /Skip if AC > 0; E remains 0
CME     /AC < 0; set E to 1
CIR     /Circulate  E and AC
```

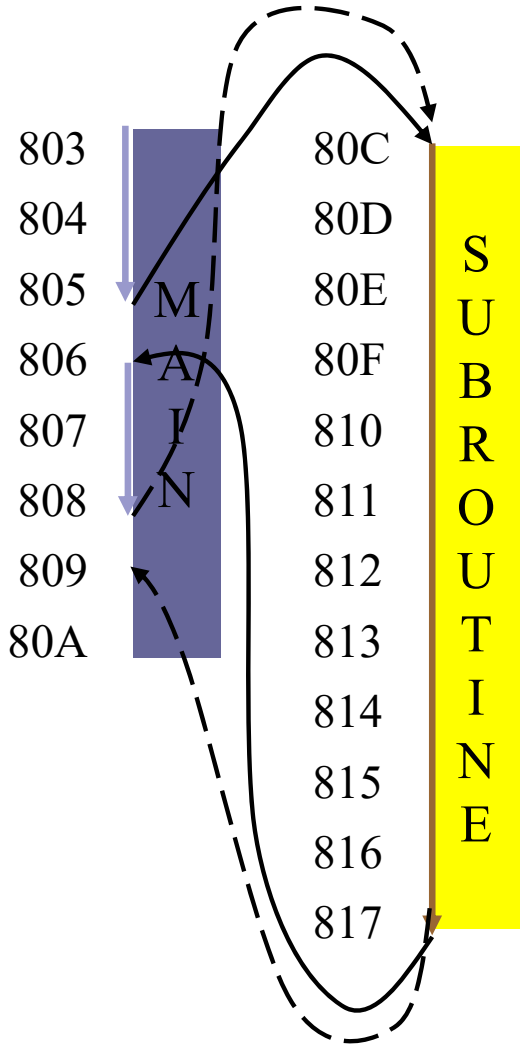
How …

1. … DOES A SUBROUTINE WORK?
2. … TO DEFINE IT?
3. … TO USE IT?

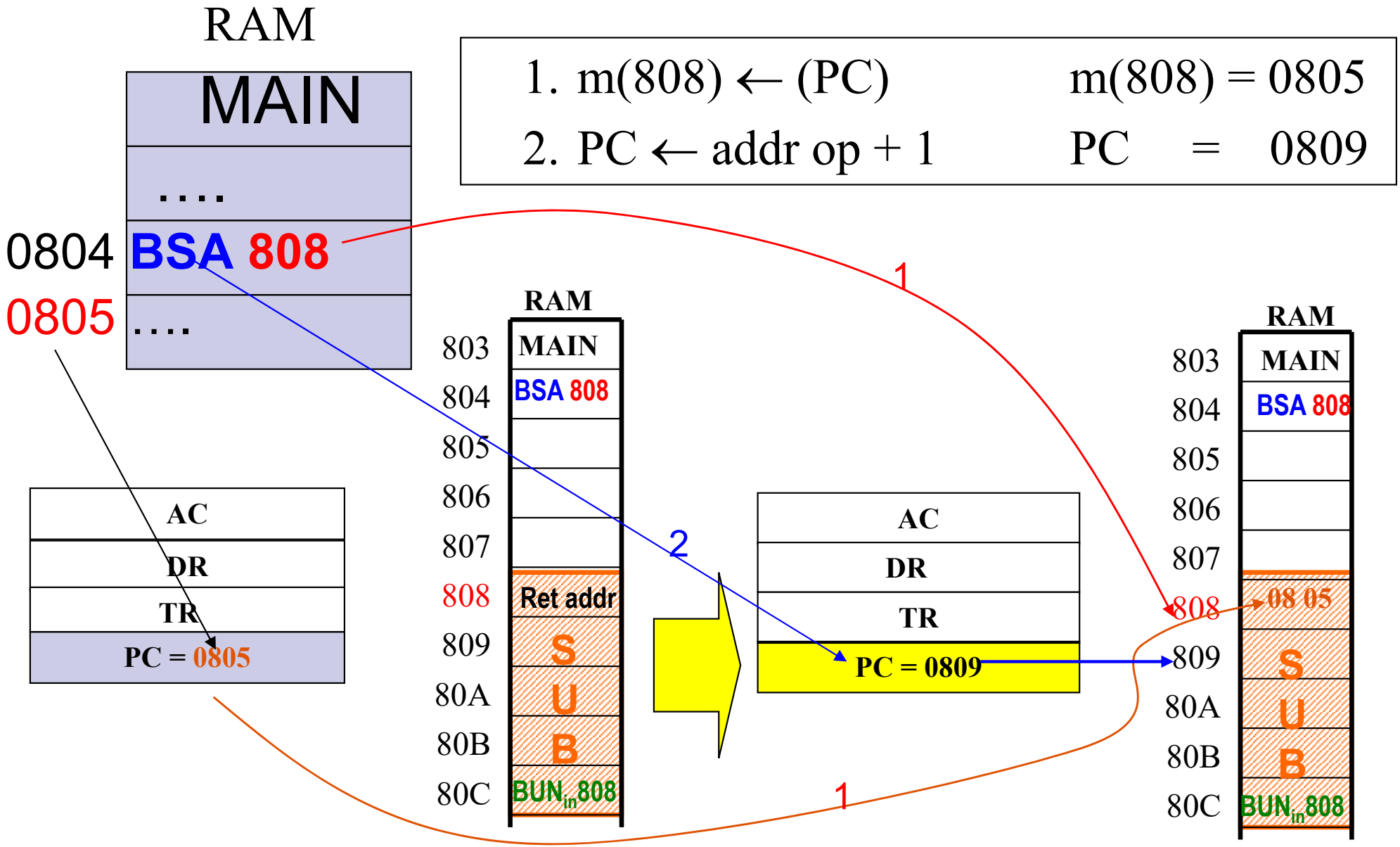
HOW DOES IT WORK?



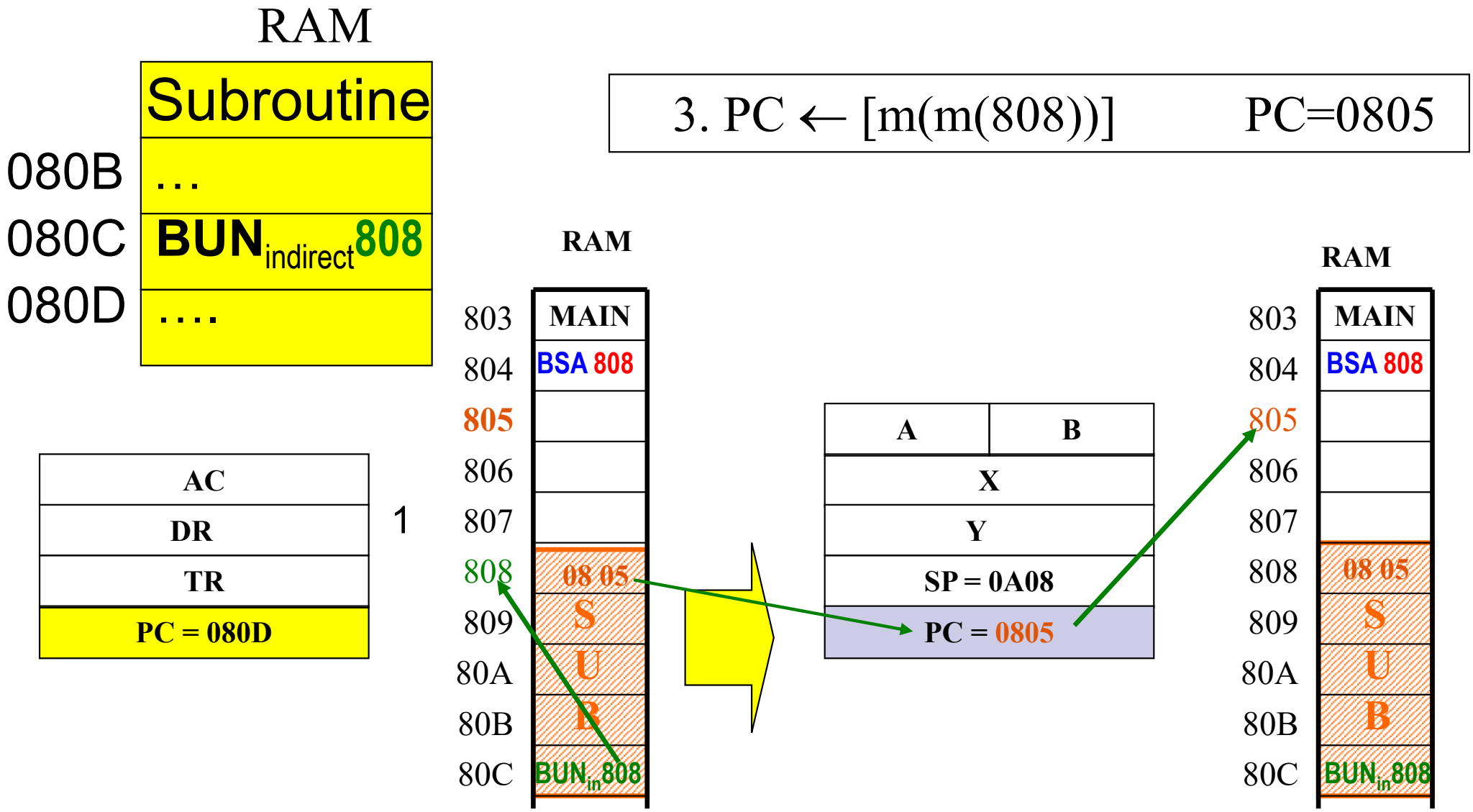
- A subroutine is a program module that is independent of the main program
- To use it, the main program transfers control to the subroutine
- The subroutine performs its function and then returns control to the main program



MAIN CALLS SUBROUTINE



SUBROUTINE **RETURNS** TO MAIN



BSA: Branch and Save Return Address

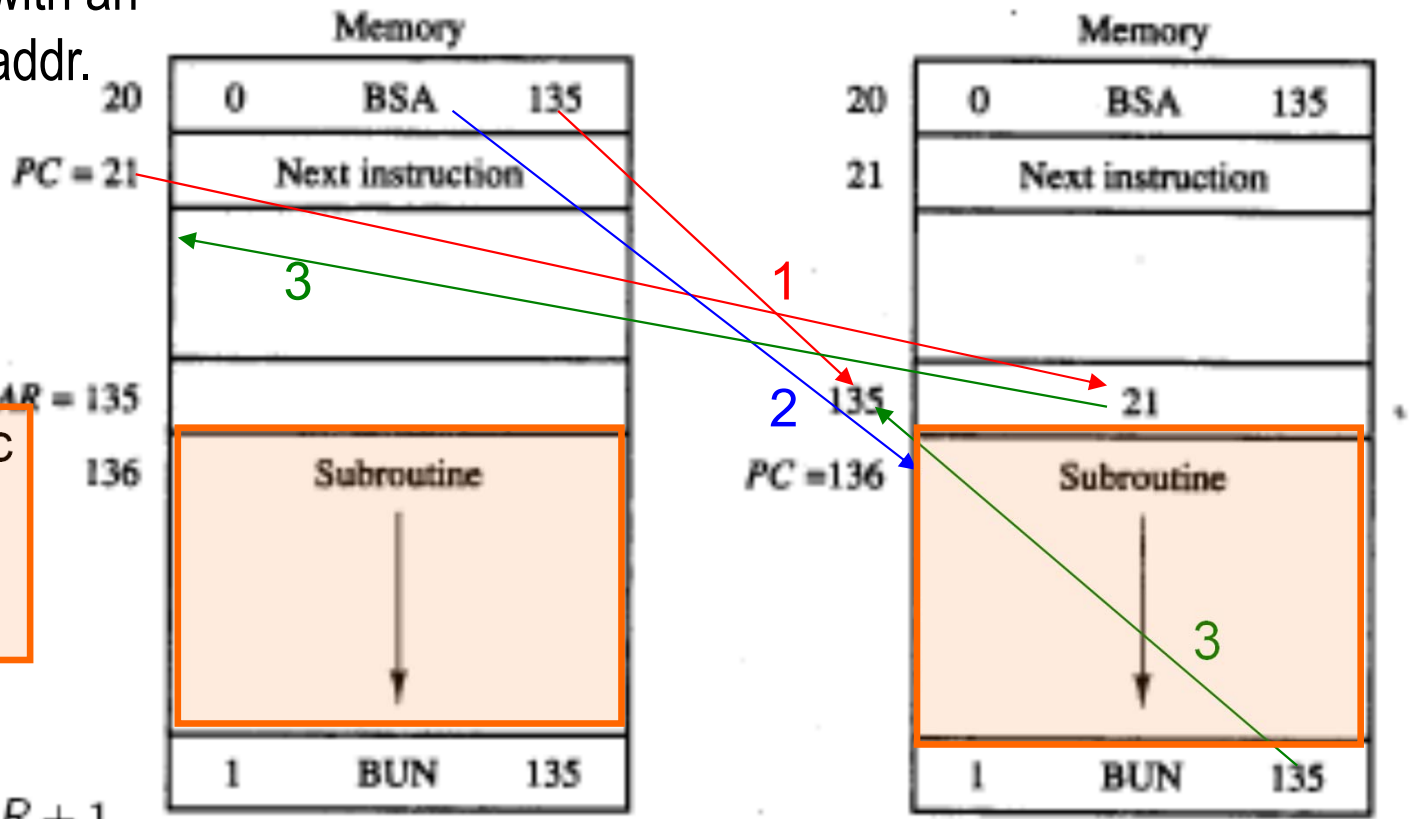
The BSA instruction

1. ***stores the return address*** (i.e., the address of the instruction to be run after the subroutine is executed, 21 in our example; this address is already prepared in PC) into the memory location specified by its effective address (135).
2. **branches to the first instruction of the *subroutine*** which is stored in the memory at the next address (136) after the stored return address.
3. The subroutine has to end with an **indirect BUN** to the return addr.

20	BSA 135
21	next instruction
22	……
….	…….
135	stored ret. addr (21)
136	Subroutine 1st instruc
137	…..
….	…..
159	Subroutine last instr
160	$BUN_{indirect}$ 135

$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$

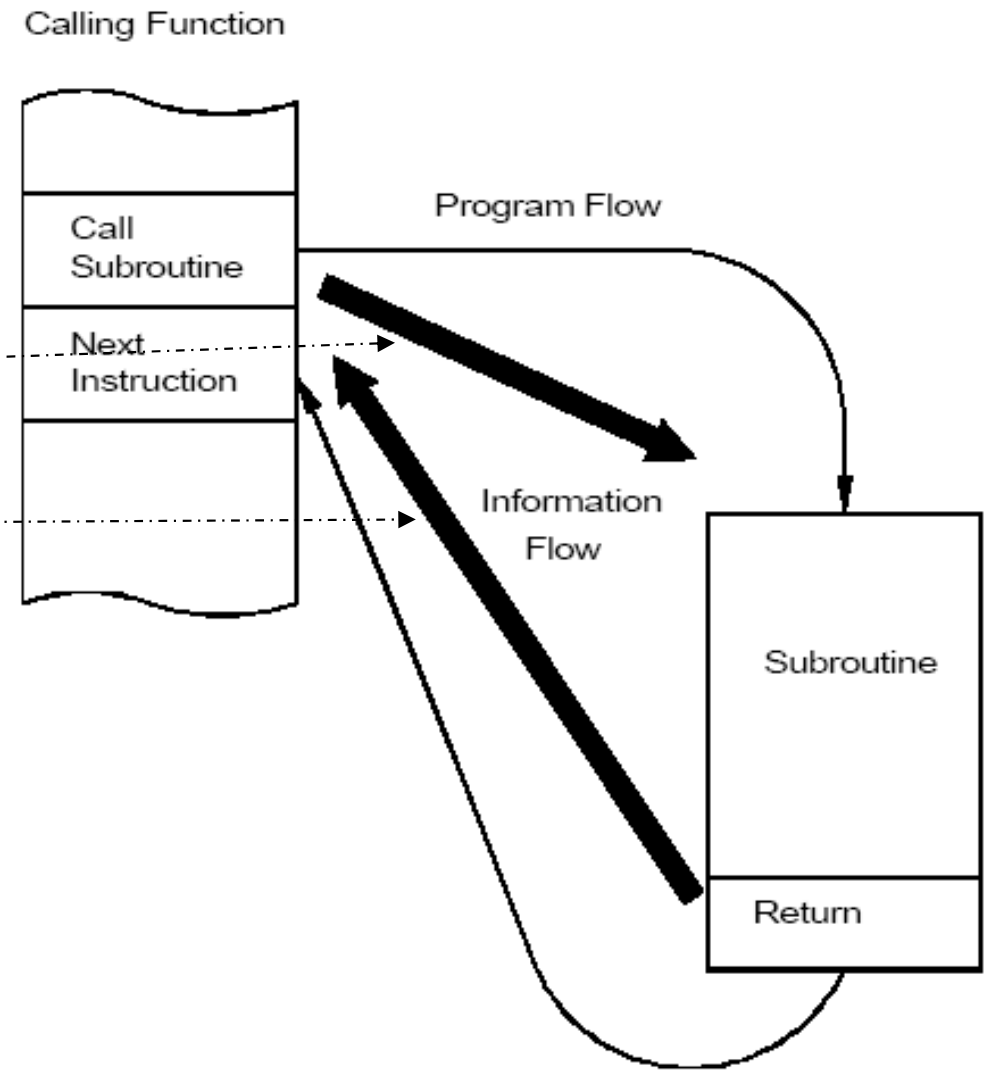
$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$



(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

Parameter Passing

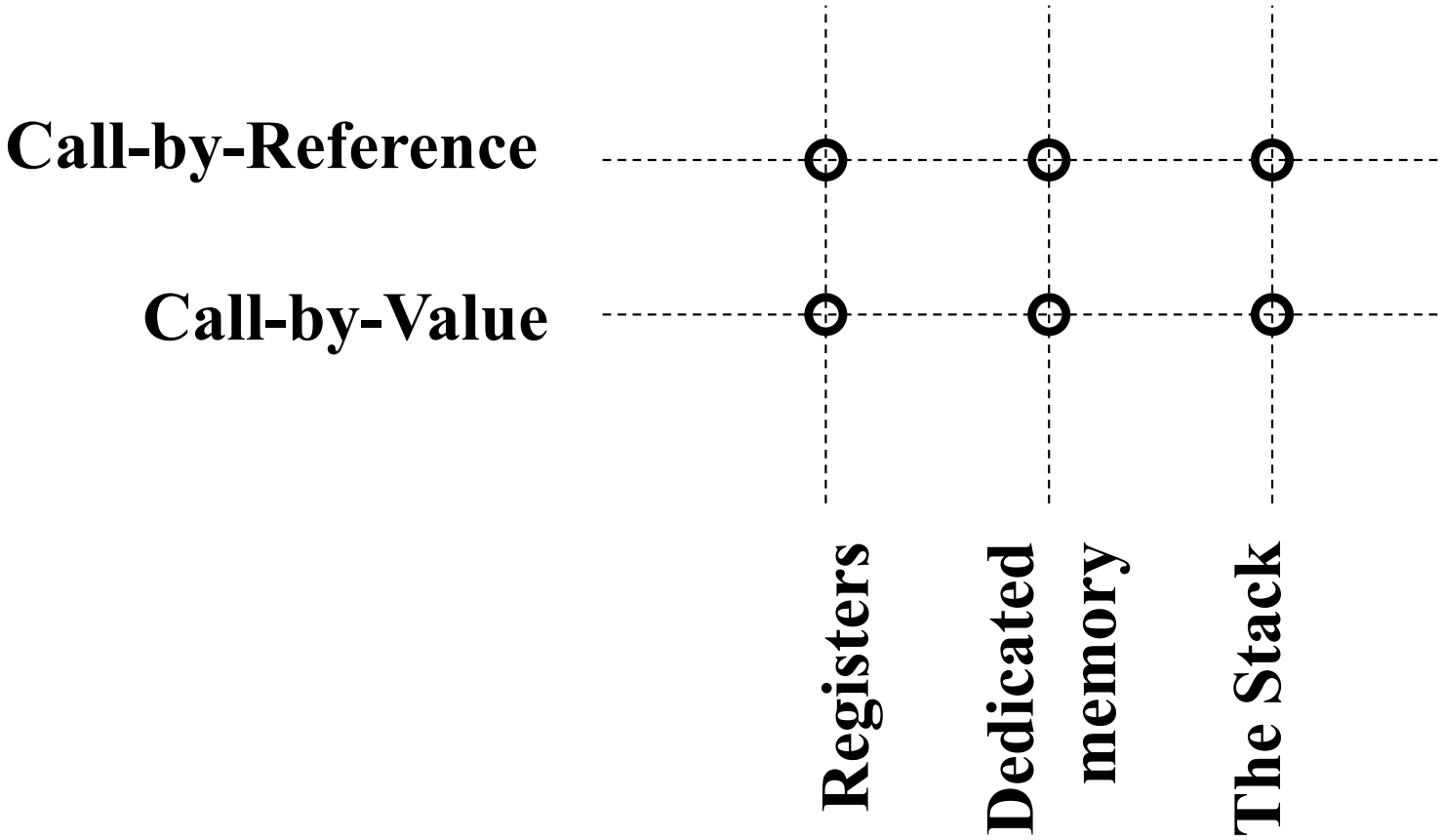
- Data passed between subroutine and calling code
  - Data is passed to the subroutine
  - Data is returned from the subroutine
- A number of approaches can be used for passing parameters



PARAMETER PASSING

- **CALL-BY-VALUE**: MAIN gives the SUBROUTINE **copies** of the data values, such that it cannot change the originals
- **CALL-BY-REFERENCE:** MAIN gives the reference to the **original** values (their address) => if something goes wrong, one can loose data
- In both techniques, parameters (data or their references) are passed using:
  - Microprocessor's Registers
  - The Stack
  - Global Data Areas (Dedicated memory)

PARAMETERS PASSING



Parameter Passing - Registers

- Using registers
  - Parameters are placed in predetermined register locations
  - Simple, efficient and fast
  - Number of parameters is limited by the number of "free" registers
  - General (data in memory not affected)
  - But only a few registers available (AC, E)

Parameter Passing - Global Data Areas

- A series of (predetermined) memory locations are used to hold the parameters
- Simple, but added overhead due to memory operations
- Possible to pass many parameters
- Can be reached from any part of the program
- May be difficult to find offending code when bug is detected
- Increases coupling between modules
- Passing addresses to global data is common

CALL-BY-VALUE

Operation:

- Before giving control to the SUBROUTINE (SUB), MAIN places data to be processed by SUB in registers (e.g., AC) or in memory and then executes BSA SUB
- SUB processes data and before returning control to MAIN, places the results into the microprocessor registers or memory and then executes BUN SUB I

It's the most popular solution for a small number of parameters

Example Subroutine

- Calling a subroutine within a program is accomplished through the "BSA" instruction.
- To illustrate the use of the "BSA" instruction in this context, **subroutine SH4** that *multiplies the content of AC by 16 (by logic shift AC four bits to the left*) is shown in the following.
- The subroutine is called twice in the program. Once in line $(101)_{16}$ and once in line $(104)_{16}$.
- When the first "BSA" instruction is executed to call subroutine SH4, the control unit stores the return address 102 into the location address symbolically defined by SH4 (i.e., SH4 = $(109)_{16}$).
- It also stores the value SH4+1 (= $(10A)_{16}$) into the program counter *PC* to start executing the subroutine.
- The last subroutine instruction in location $(10F)_{16}$ (called the subroutine **return** instruction) makes an indirect unconditional branching back to location (102)16 .
- The "BSA" is referred to as the subroutine **call** instruction.

SUBROUTINE Example

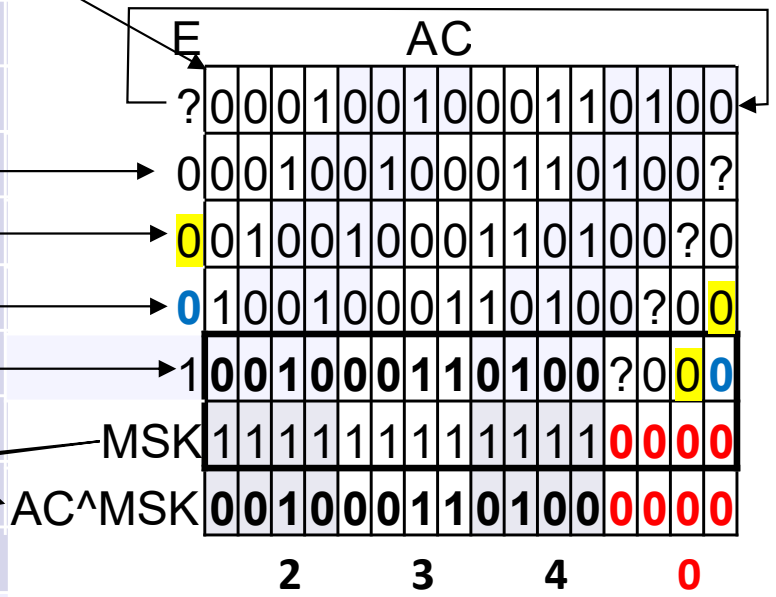
uOttawa

Subroutine SH4

Table 6.16

		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shift left 4 times
109	**SH4,**	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	/Circulate left second time
10C		CIL	/Circulate left third time
10D		CIL	/Circulate left fourth time
10B		AND MSK	/Set AC (3-0) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

- **Call-by-value**
- Parameters passed /registers
  - MAIN loads the number to be shifted in AC
  - Subroutine SH4 returns the shifted number in AC
- Subroutine SH4 **multiplies** AC by 2^4, i.e. logically shifts AC 4 times to the left
- MAIN calls SUB twice to multiply by 16 numbers X & Y



Parameter Linkage

Subroutine OR performs logic OR between 2 operands

$$Y = W \vee X = \overline{\overline{W} \cdot \overline{X}}$$

Call-by-value

1st operand (X) passed
➔ registers (AC)

2nd operand (W) passed
➔ memory = the location following the BSA instruction!

Table 6.17

		ORG 200	
200		LDA X	/Load first operand into AC
201		BSA OR	/Branch to subroutine OR
202	W,	HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand X
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand W
20B		CMA	/Complement second operand W'
20C		AND TMP	/AND complemented first operand
200		CMA	/Complement again to get OR
20E		ISZ OR	/Increment to get return address
20F		BUN OR I	/Return to main program
210	TMP,	HEX 0	/Temporary storage
		END	

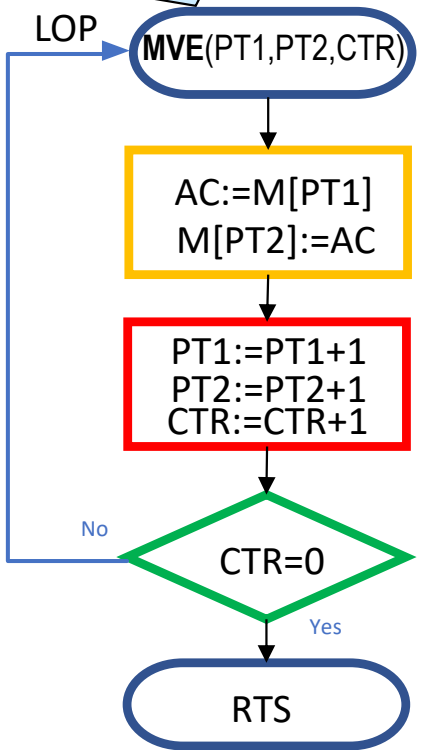
CALL-BY-REFERENCE

- **Pointer to memory**
  - Pass the address of the parameters in memory to the subroutine by
    - a register or
    - memory = global variable or parameter linkage
  - More complex, extra overhead for use of the pointer

Call-by-reference

Pointers to memory locations where data are stored; passed by memory / parameter linkage.

Write and test subroutine that copies a block of 16 16-bit words {a_0, a_1,…a_{15}} from memory locations starting at 100 HEX to 200 HEX




```
Addr                ORG 0          /Main program
 0                  BSA MVE        /Branch to subroutine
 1                  HEX 100        / 1st parameter = address of first source data
 2                  HEX 200        / 2nd parameter = address of first destination data
 3                  DEC -16        / 3rd parameter (FFF0) = Number of words to move
 4                  HLT
 5  MVE,            HEX 0          /To store address provided by BSA MVE (1)
 6                  LDA MVE I      /Copy 1st parameter, i.e., address of source (100),
 7                  STA PT1        /    into the first pointer (PT1:=100)
 8                  ISZ MVE        /Increment M[5] to point to 2 = addr. of 2nd param.
 9                  LDA MVE I      /Bring address of destination (200)
 A                  STA PT2        /    and store it in the second pointer (PT2:=200)
 B                  ISZ MVE        /Increment M[5] to point to 3 = addr. of 3rd param.
 C                  LDA MVE I      /Bring the number of words to be copied (-16)
 D                  STA CTR        / and initialize the word counter with it (CTR:=-16)
 E                  ISZ MVE        /Increment M[5] to obtain correct return address
 F  LOP,            LDA PT1 I      /Load source item: AC← a_k @ M[PT1]
10                  STA PT2 I      /Store in destination: M[PT2]← a_k @ AC
11                  ISZ PT1        /Increment source pointer
12                  ISZ PT2        /Increment destination pointer
13                  ISZ CTR        /Increment counter
14                  BUN LOP        /Repeat 16 times
15                  BUN MVE I      /Return to main program
16 PT1,             /              / pointer to the source data
17 PT2,             /              / pointer to the destination data
18 CTR,

                    ORG 100
100                 a0
101                 a1
...                 ...
10F                 a15
200                 ORG 200
                    ...
                    END
```

Main program

parameters

Subroutine

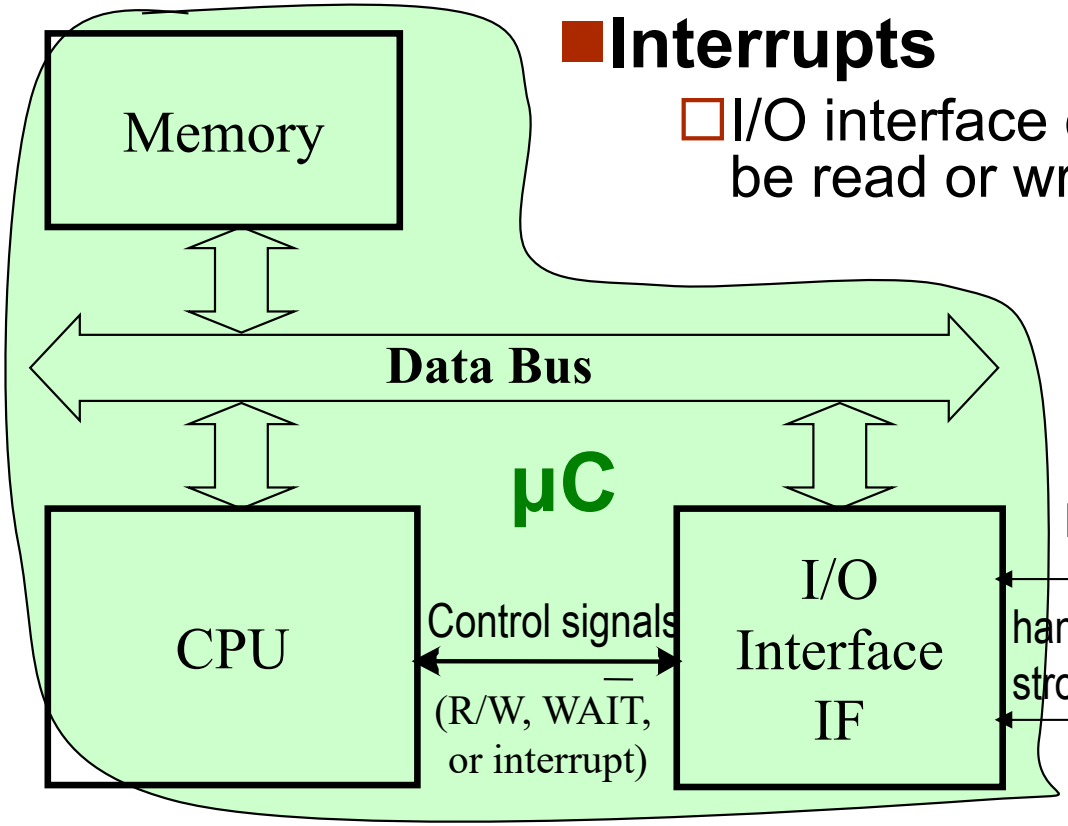
Data

Table 6.18

After running the program
PT1 = 110
PT2 = 210
CTR = 0
{a_0, a_1, … a_{15}} will be at both {100..10F} and {200..20F}

Synchronizing CPU & I/O Interface

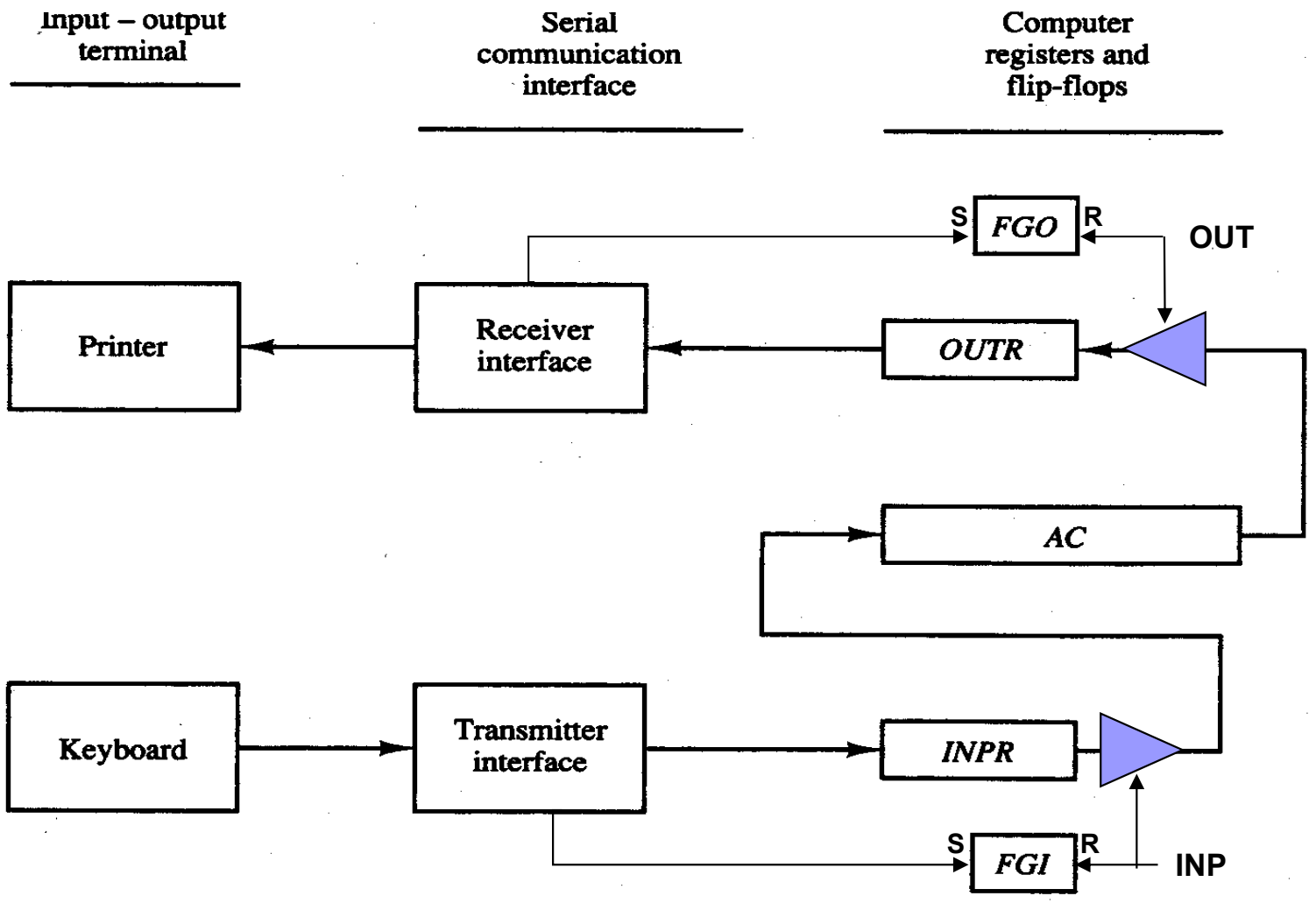
- Hardware provides an **interface** to the I/O device. Contains:
  - data register (for data transfer)
  - status register
  - control register
- **SW Polling**
  - CPU monitors the status register
  - When data is ready, CPU reads or writes from/to the data register
- **Interrupts**
  - I/O interface can interrupt the CPU when data can be read or written (will study later)
- **DMA (direct-memory access):**
  - the IF takes over the control of the system's Data, Address & Control Buses and transfers sequences of data direct to the memory.



Basic Computer I/O

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off



SW Polling

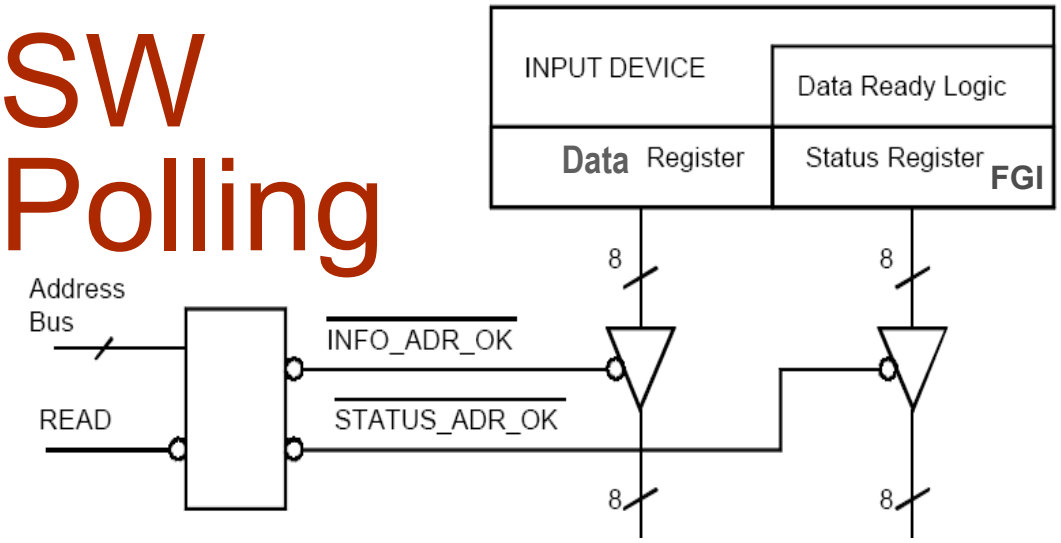


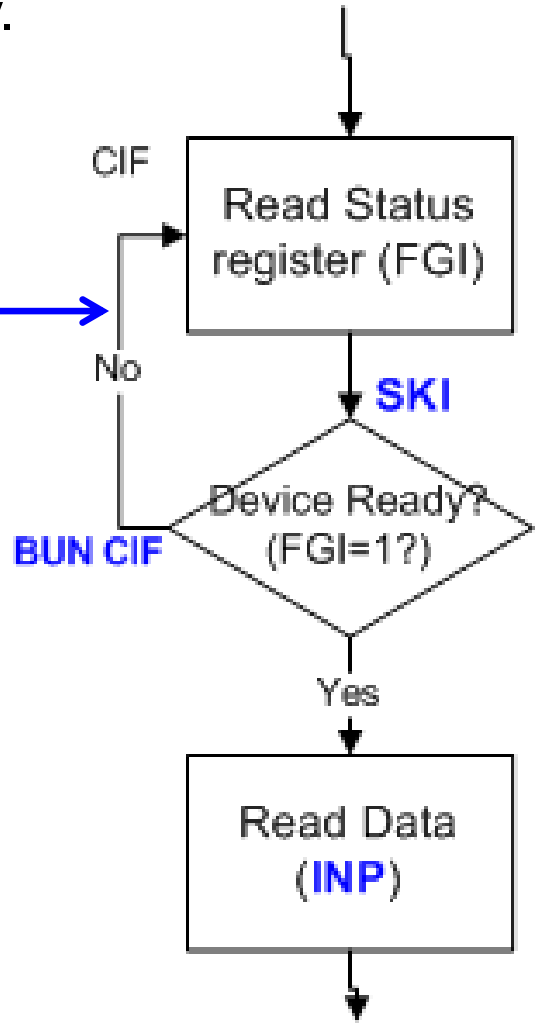
Table (a) lists the instructions needed to input a character, print it, and then store it in memory.
Table (b) lists the instructions needed to print a character initially stored in memory.

(a) Input a character & store it at M[CHR]:

```
CIF, SKI                    /Check input flag
     BUN CIF      /Flag=0, branch to check again
     INP          /Flag=1, input character
     OUT          /Print character
     STA CHR      /Store character
     HLT
CHR, -            /Store character here
```

(b) Output one character from M[CHR]:

```
     LDA CHR      /Load character into AC
COF, SKO          /Check output flag
     BUN COF      /Flag=0, branch to check again
     OUT /Flag= 1, /output character
     HLT
CHR, HEX 0057     /Character is “W"
```



Character Manipulation

Table 6.20

```
IN2,    -          / Subroutine entry
FST     SKI
        BUN FST
        INP        / Input 1'st character & reset FGI
        OUT
        BSA SH4    / Shift left four times
        BSA SH4    / Shift left four more  times
SCD,    SKI
        BUN SCD
        INP        / Input second character & reset FGI
        OUT
        BUN IN2 I  / Return
```

SUBROUTINE IN2 inputs 2 bytes from INPR (say ASCII characters) and packs them into one 16-bit word in AC to be returned to the calling program (**call-by-value / register**).

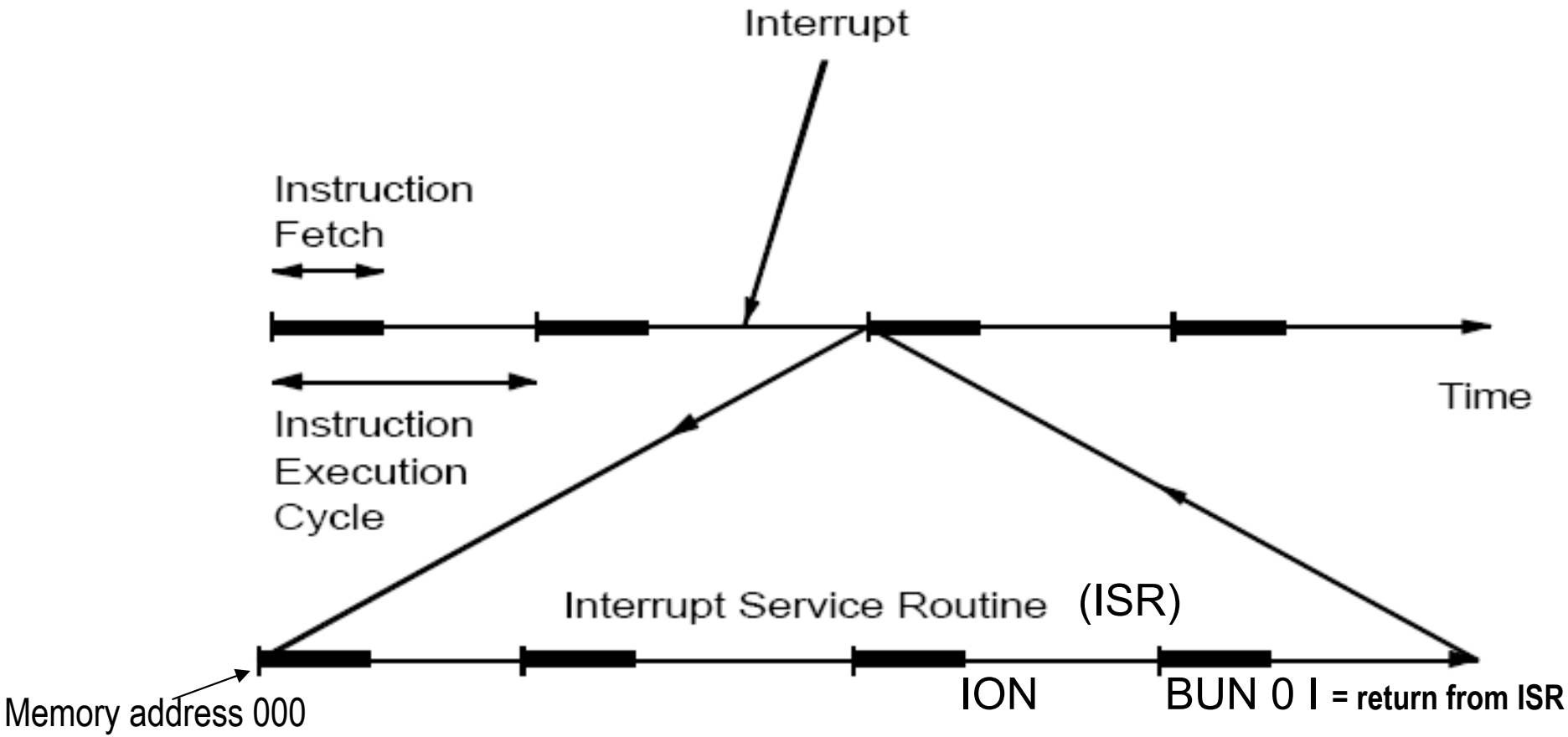
Table 6.21

```
        LDA ADS    /Load first address of buffer
        STA PTR    /Initialize pointer
LOP,    BSA IN2    /Go to subroutine IN2 (Table 6-20)
        STA PTR I  /Store double character word in buffer
        ISZ PTR    /Increment pointer
        BUN LOP    /Branch to input more characters
        HLT
ADS,    HEX 500    /First address of buffer
PTR,    HEX 0      /Location for pointer
```

PROGRAM that uses subroutine IN2 to input a stream of characters from the keyboard, pack every 2 characters in one word, and store them in a buffer starting from address $(500)_{16}$ of the memory. No counter is used in the program, so characters will be read as long as they are available or until the buffer reaches location 0 (after location FFF)

Asynchronous Interrupt - CPU Timing

- Interrupts can occur at any time during an instruction cycle

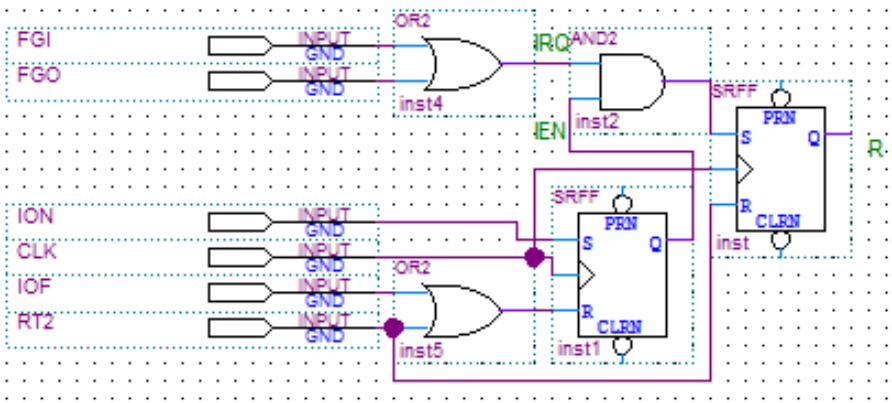


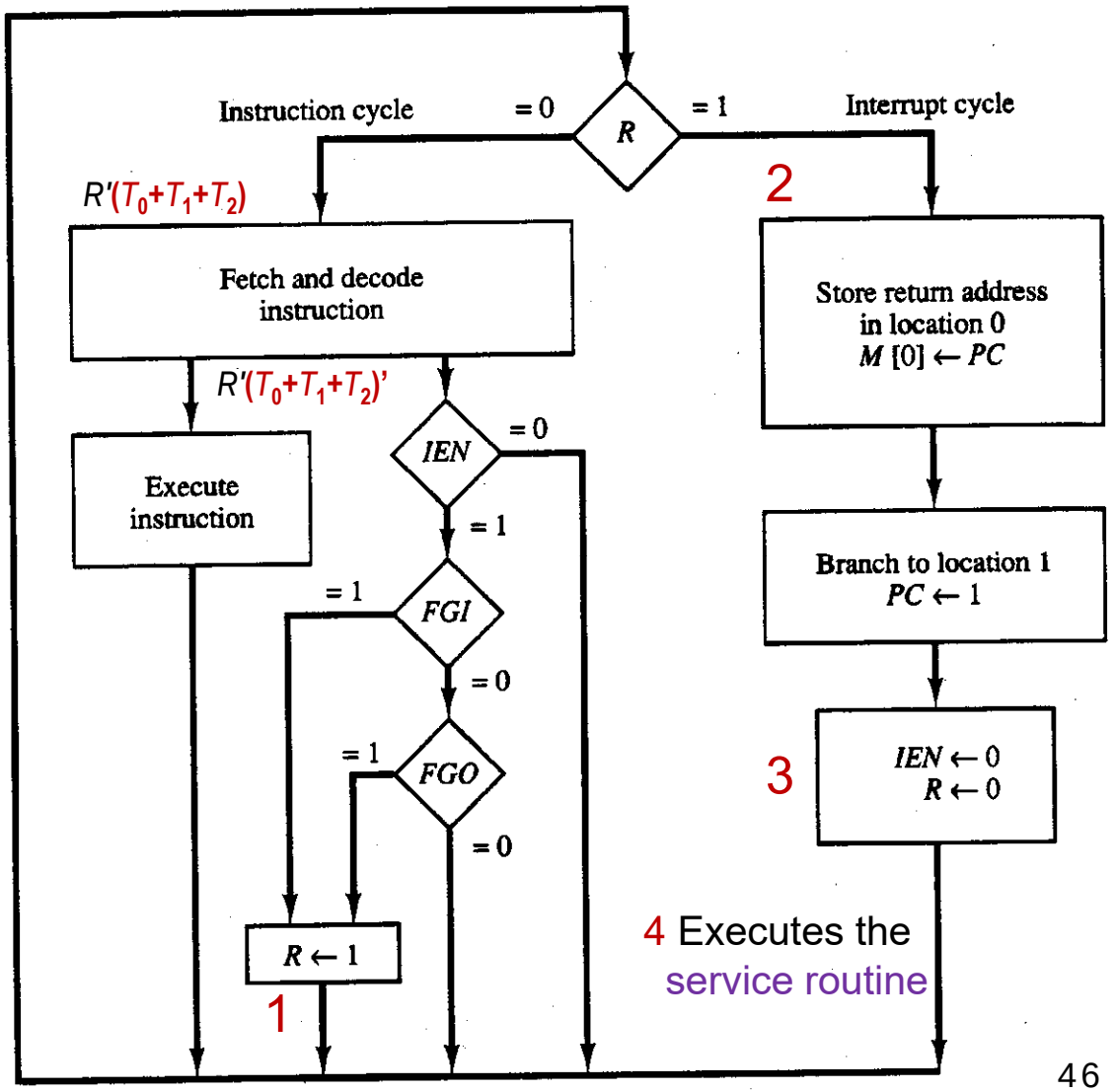
Interrupt System Specification

- Allow for asynchronous events to occur and be recognized.
- Wait for the current instruction to finish before servicing an interrupt.
- Service interrupt with a sub-routine (ISR) and returns to interrupted code.
- Enabling and disabling interrupts (IEN)
- Multiple (here 2: FGO, FGI) sources of interrupts
  - Simultaneous interrupts.

Internal CPU Interrupt Hardware

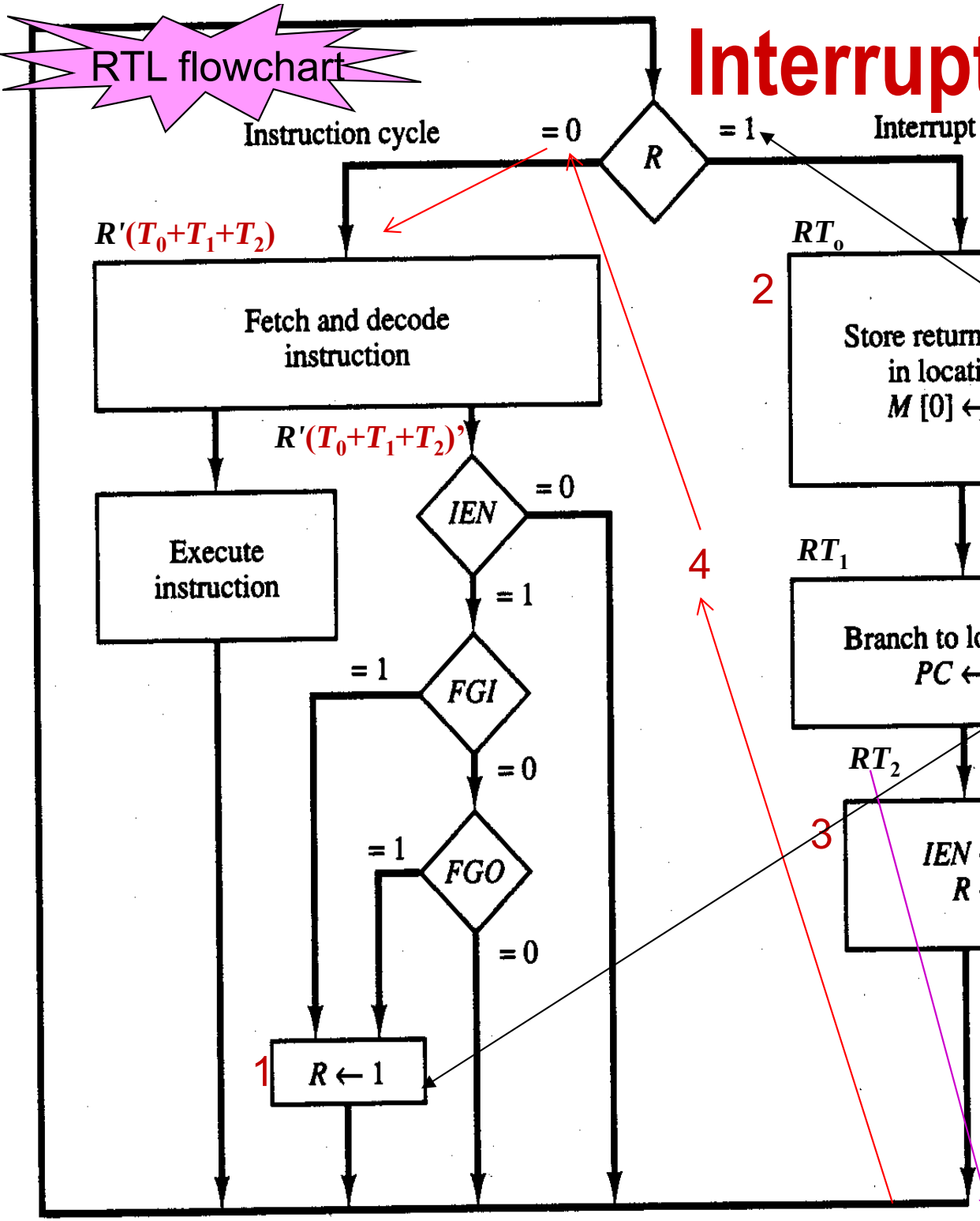
- Use a flip-flop (R) to catch IRQ (FGI, FGO)
  - Multiple device interrupt lines (FGI, FGO) can be OR-wired together
  - CPU waits until the current instruction is executed and then can process interrupt
- Can enable/disable interrupt using enable-disable flip-flop (IEN)
  - When interrupt is acknowledged by CPU HW, interrupts are disabled (IEN←0) for the duration of interrupt servicing
- The source of interrupt is determined by FGI or FGO → the Interrupt Service Routine (ISR) has to check which I/O device generated the IRQ
- At the end of ISR
  - ION is executed to re-arm the interrupt system: IEN←1
  - BUN 0 I (i.e., $BUN_{indirect}0$) is executed to return to the interrupted program



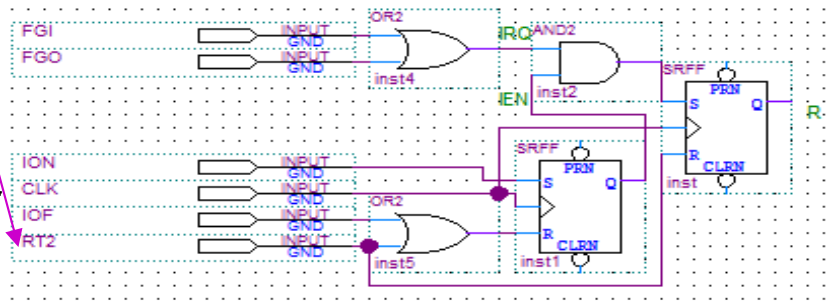


Interrupt Cycle

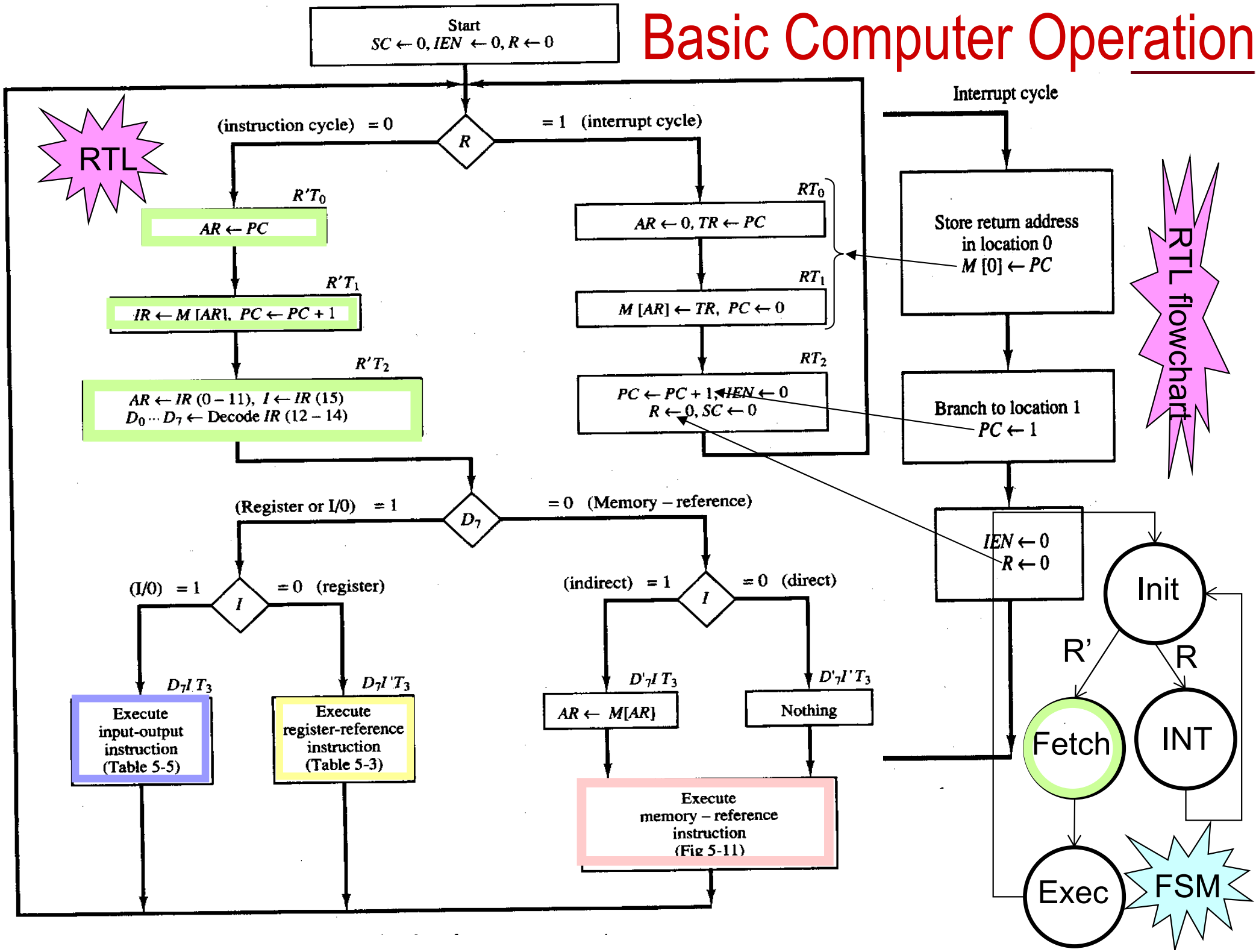
RTL flowchart



1. While an instruction is in execution phase $(T_0+T_1+T_2)'$, if a flag (*FGI* or *FGO*) is set to 1 by an enabled (IEN) interrupt, the computer stores the *Interrupt **R**equest* in ***R*** $(T_0+T_1+T_2)'(IEN)(FGI+FGO)$: $R \leftarrow 1$ and completes the execution of the instruction in progress;
2. next goes to an interrupt cycle (R=1), where
   - stores return address in memory location 0 and
   - puts SVR address in PC
3. At RT_2 disables the interrupt system (IEN=0) and resets R (since the system already takes care of this interrupt);
4. Since now R=0, the next cycle will be a normal instruction fetch from memory address 1, which may be the 1st instruction of ISR or a direct unconditional branching to the ISR base address.

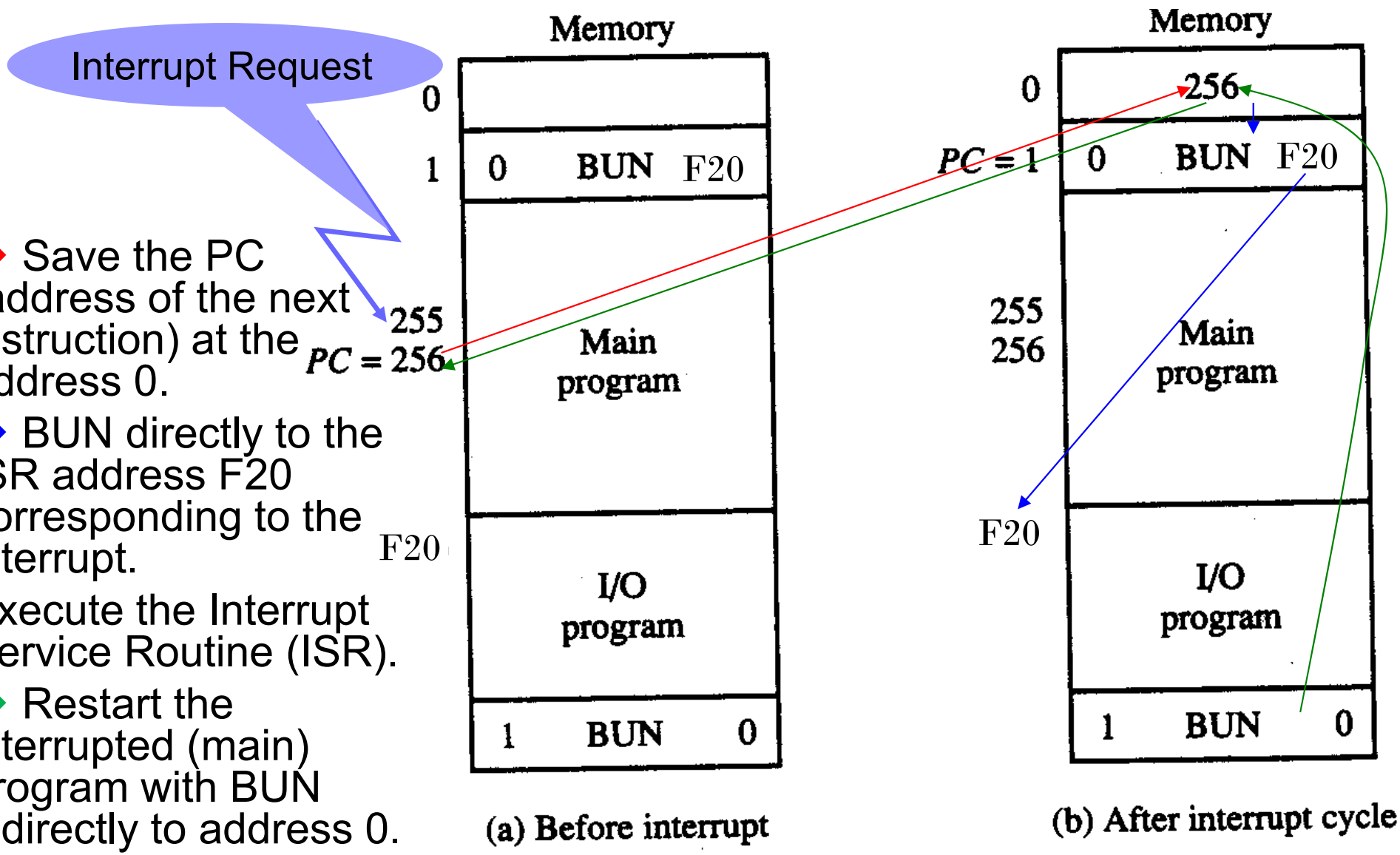


Basic Computer Operation



Interrupt Service Cycle

1. → Save the PC (address of the next instruction) at the address 0.
2. → BUN directly to the ISR address F20 corresponding to the interrupt.
3. Execute the Interrupt Service Routine (ISR).
4. → Restart the interrupted (main) program with BUN indirectly to address 0.



Interrupt Service Routine

An example of a program that services an **ASCII Character I/O interrupt** is listed next.

- Location 0 is reserved for the return address.
- Here, Location 1 contains a direct unconditional branching to the interrupt service routine (**ISR**) SRV.
- The **running program** starts at location $(100)_{16}$ of the memory.
- **ISR** (interrupt service routine = SRV) starts at location $(200)_{16}$ of the memory.
- The running program contains an "ION" instruction at the beginning of the program to turn the interrupt facility on.
- The interrupt system is turned OFF (**disarmed**) in hardware by the interrupt (micro-operation RT_2: IEN ← 0) just before the execution of the **ISR**.
- Thus, it is important to turn the interrupt facility back ON (**arm**) just before the end of the service routine. This is done in SW by including the instruction "ION".

000	ZRO,		/ Return address stored here
001		BUN SRV	/Branch to service routine
.....	.....	.....	.....
100		CLA	/ Portion of running program
101		ION	/ Turn on interrupt facility
102		LDA X	
103		ADD Y	/ Interrupt occurs here
104		STA Z	/ Program returns here after interrupt
.....	.....	.....	.....
			/ ISR
200	SRV,	....	
		....	
		ION	/ Turn interrupt on
		BUN ZRO I	/ Return from ISR to running program
		END	

Table 6.23

ISR

The service routine (ISR) must contain instructions to perform the following tasks:

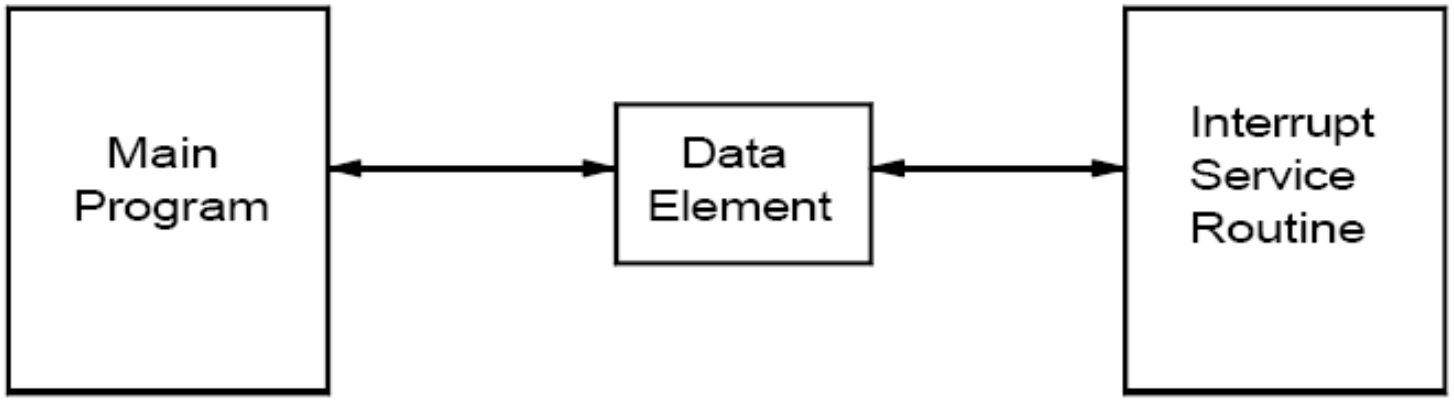
1. Save the contents of processor registers (AC and E in this basic computer)
2. Check which input-output flag is set (FGI, FGO)
3. Service the device whose flag is set
4. Restore the contents of processor registers
5. Turn ON the interrupt facility
6. Return to the running program

000	ZRO,		/ Return address stored here	
001		BUN SRV	/ Branch to service routine	
•••••	•••••	•••••••	•••••••	
100		CLA	/Portion of running program	
101		ION	/ Turn on interrupt facility	
102		LDA X		
103		ADD Y	/ Interrupt occurs here	
104		STA Z	/ Program returns here after intr	
•••••	•••••	•••••••	/ **Interrupt service routine**	
200	SRV,	STA SAC	/ Store content of AC	1
		CIR	/ Move E into AC(15)	
		STA SE	/ Store content of E	
		SKI	/ Check input flag	2
		BUN NXT	/ If flag is OFF, check next flag	
		INP	/ Flag ON, input char & reset FGI	3
		OUT	/ Print character	
		STA PT1 I	/ Store it in input buffer	
		ISZ PT1	/ Increment input pointer	
	NXT,	SKO	/ Check output flag	2
		BUN EXT	/ Flag is OFF, exit	
		LDA PT2 I	/ Load char from output buffer	3
		OUT	/ Output character	
		ISZ PT2	/ Increment output pointer	
	EXT,	LDA SE	/ Restore value of AC(15)	4
		CIL	/ Shift it to E	
		LDA SAC	/ Restore content of AC	
		ION	/ Turn interrupt on	5
		BUN ZRO I	/ Return from ISR to run'g prog	6
	SAC,		/ AC is stored here	
	SE,		/ E is stored here	
	PT1,		/ Pointer of input buffer	
	PT2,		/ Pointer of output buffer	

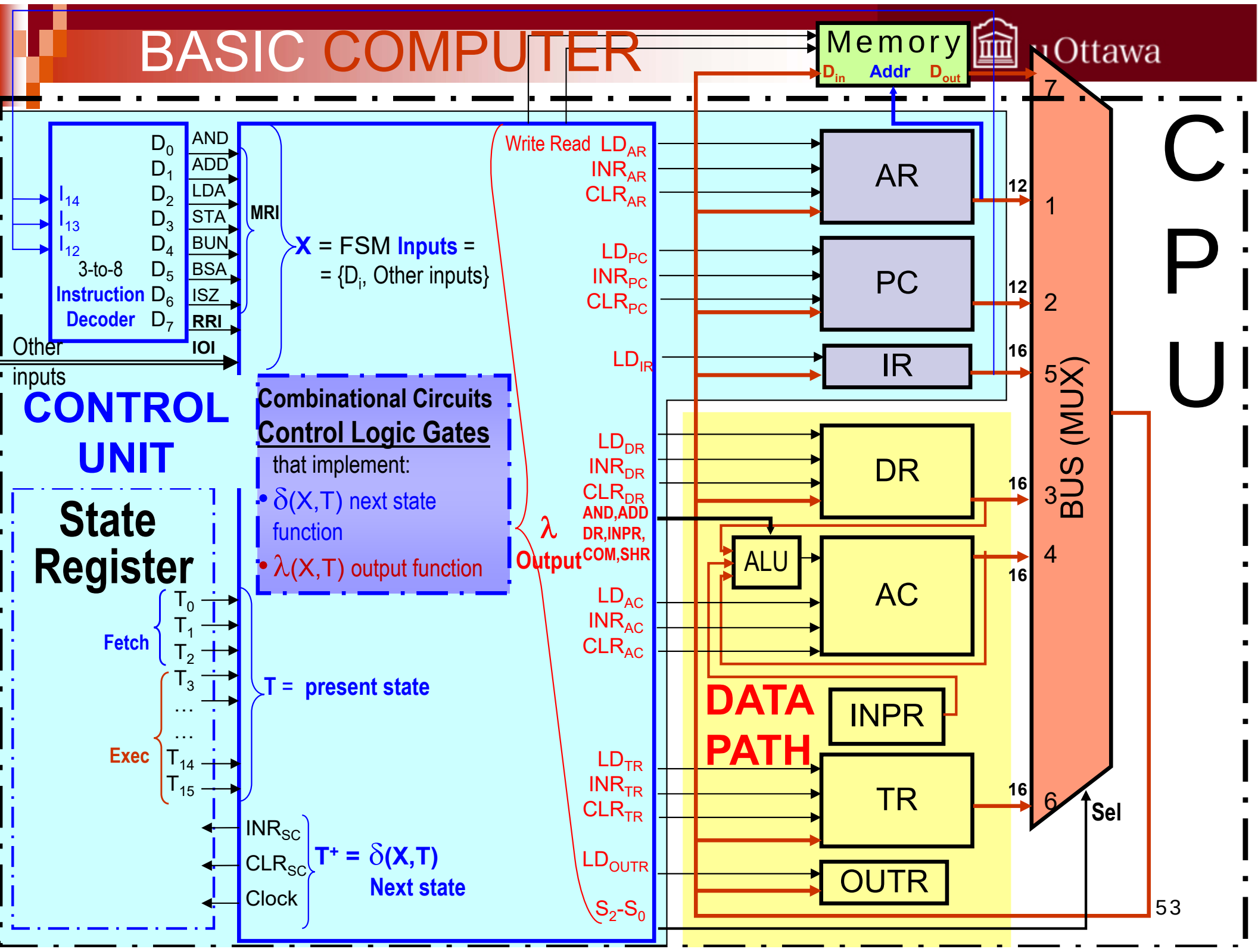
Table 6.23

Data Exchange with ISRs

- Use global data
- May need to disable interrupts for critical regions of code using this data



BASIC COMPUTER



Basic Computer Instruction List (Binary)

D_7		I_{15}=0	I_{15}=1	op code			I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
0	*mem*	**MRI Direct Addr.**	**MRI** Indirect Addr.	*x*	*x*	*x*	Memory				A	D	D	R	E	S	S	
1	*reg*	**RRI**	**IOI**	**1**	**1**	**1**	C	o	d	e	Extension							

D_7	*Instr.* *Dec*	Addressing Mode I_{15}=0	I_{15}=1	op code I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
		MRI Direct Addr.	**MRI** Indirect Addr.				Memory				A	D	D	R	E	S	S	
D_7=0 ***mem***	D_0	**AND**=$0addr	AND$_i$=$8(addr)	*0*	*0*	*0*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_1	**ADD**=$1addr	ADD$_i$=$9(addr)	*0*	*0*	*1*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_2	**LDA** =$2addr	LDA$_i$=$A(addr)	*0*	*1*	*0*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_3	**STA** =$3addr	STA$_i$=$B(addr)	*0*	*1*	*1*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_4	**BUN** =$4addr	BUN$_i$=$C(addr)	*1*	*0*	*0*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_5	**BSA** =$5addr	BSA$_i$=$D(addr)	*1*	*0*	*1*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
	D_6	**ISZ** =$6addr	ISZ$_i$=$E(addr)	*1*	*1*	*0*	AR_{11}	AR_{10}	AR_9	AR_8	AR_7	AR_6	AR_5	AR_4	AR_3	AR_2	AR_1	AR_0
D_7=1 ***reg***		**RRI**	**IOI**	**1**	**1**	**1**	C	o	d	e	Extension							
	D_7	**CLA** =$7800	INP = $F800	*1*	*1*	*1*	**1**	0	0	0	0	0	0	0	0	0	0	0
	D_7	**CLE** =$7400	OUT = $F400	*1*	*1*	*1*	0	**1**	0	0	0	0	0	0	0	0	0	0
	D_7	**CMA** =$7200	SKI = $F200	*1*	*1*	*1*	0	0	**1**	0	0	0	0	0	0	0	0	0
	D_7	**CME** =$7100	SKO = $F100	*1*	*1*	*1*	0	0	0	1	0	0	0	0	0	0	0	0
	D_7	**CIR** =$7080	ION = $F080	*1*	*1*	*1*	0	0	0	0	**1**	0	0	0	0	0	0	0
	D_7	**CIL** =$7040	IOF = $F040	*1*	*1*	*1*	0	0	0	0	0	**1**	0	0	0	0	0	0
	D_7	**INC** =$7020	n/a	*1*	*1*	*1*	0	0	0	0	0	0	**1**	0	0	0	0	0
	D_7	**SPA** =$7010	n/a	*1*	*1*	*1*	0	0	0	0	0	0	0	**1**	0	0	0	0
	D_7	**SNA** =$7008	n/a	*1*	*1*	*1*	0	0	0	0	0	0	0	0	**1**	0	0	0
	D_7	**SZA** =$7004	n/a	*1*	*1*	*1*	0	0	0	0	0	0	0	0	0	**1**	0	0
	D_7	**SZE** =$7002	n/a	*1*	*1*	*1*	0	0	0	0	0	0	0	0	0	0	**1**	0
	D_7	**HLT** =$7001	n/a	*1*	*1*	*1*	0	0	0	0	0	0	0	0	0	0	0	**1**

Binary encoding **One-hot** (*bit-per-state*) encoding "addr" = 12 bit address of the operand
"(addr)" = address of the operand address