Instruction Codes

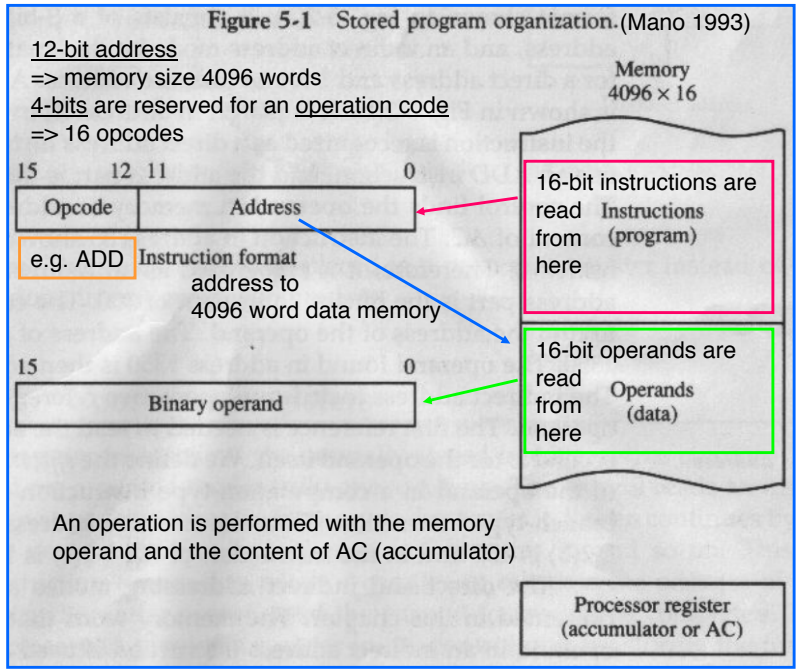
- The organization of a computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses.
- The internal organization of a digital system is defined by the sequence of micro-operations it performs on data stored in its registers.
- The general purpose digital computer is capable of executing various micro-operations and it can be instructed as to what specific instructions it must perform.

- The user of a computer can control the process by means of a program.
- A program is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- An instruction is a binary code that specifies a sequence of micro-operations.
- Instructions and data are stored in memory.
- The ability of store and execute instructions, the stored program concept (von Neumann architecture), is the most important property of a general-purpose computer.

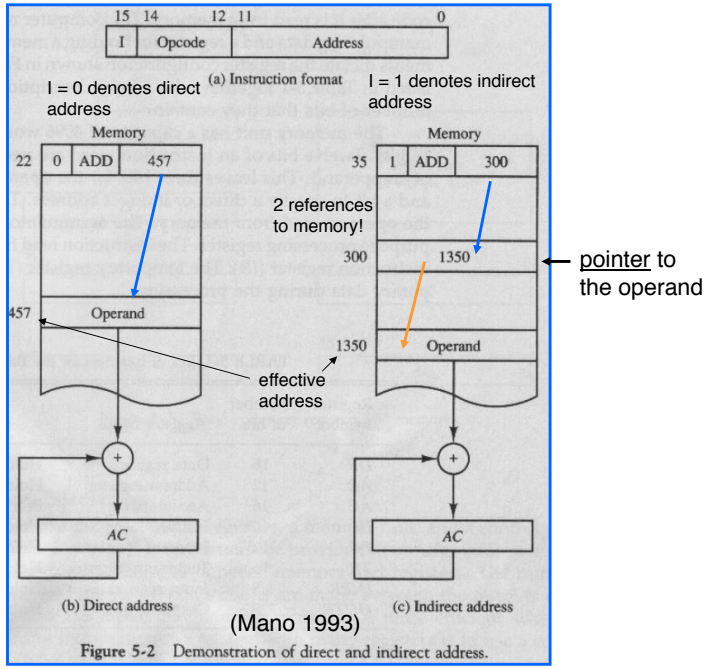
- An instruction code is a group of bits that instruct the computer to perform a specific operation (set of micro-operations).
- Operation code is a part of instruction code; a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.
- Control unit receives the instruction from memory and interprets the operation code bits. It the issues a sequence of control signals to initiate MOs in internal registers.

- For every operation code, the control issues a sequence of MOs.
- An operation code is called macro-operation because it specifies a set of micro-operations.
- An instruction code specify also the registers or the memory words for operands and results
  - Memory words can be specified by their address
  - Registers can be specified by a binary code of k bits specifying one of 2^k possible registers.
- Each computer (CPU) has its own instruction code format.

- A simple computer organization
  - One register
  - An instruction code format with two parts
    - Operation code
    - An address: tells the control where to find an operand from memory
- Fig. 5-1. Control reads 16-bit instruction from program memory. It uses the 12-bit address part of instruction to read 16-bit operand from data memory. It then executes the operation specified by the operation code.
- If an operation does not need an operand from memory, the address bits can be used for other purposes, e.g. for specifying other operations or an operand.



- When the second part of an instruction code specifies and operand, the instruction is said to have an immediate operand.
- When the second part specifies an address of an operand, the instruction is said to have a direct address.
- Indirect address: the second part specifies a memory location where the address of the operand is found.
- Indirect address increases addressable memory size => more bits for specifying addresses of operands.



Computer Registers

- Instructions are stored in consecutive memory locations and are executed sequentially one at a time.
- The control read an instruction from a specific address in memory and executes it: after that next instruction is read an executed, and so on.
- Register are needed for storing fetched instructions, and counters for computing the address of the next instruction.

- Computer needs processor registers for data manipulation and holding addresses (see. Fig. 5-3 and Table 5-1).
- Program counter (PC) goes through a counting sequence and causes the computer to read sequential instructions from memory.
- Instructions are read and executes in sequence unless a branch instruction is encountered
  - Calls for a transfer to a nonconsecutive instruction in the program
  - The address part of a branch instruction becomes the address of the next instruction in PC
  - Next instruction is read from the location indicated by PC

TABLE 5-1 List of Registers for the Basic Computer (Mano 1993)

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character (from an input device)
OUTR	8	Output register	Holds output character (for an output device)

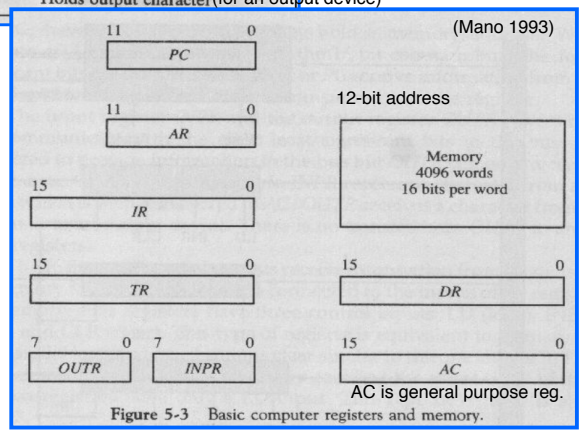
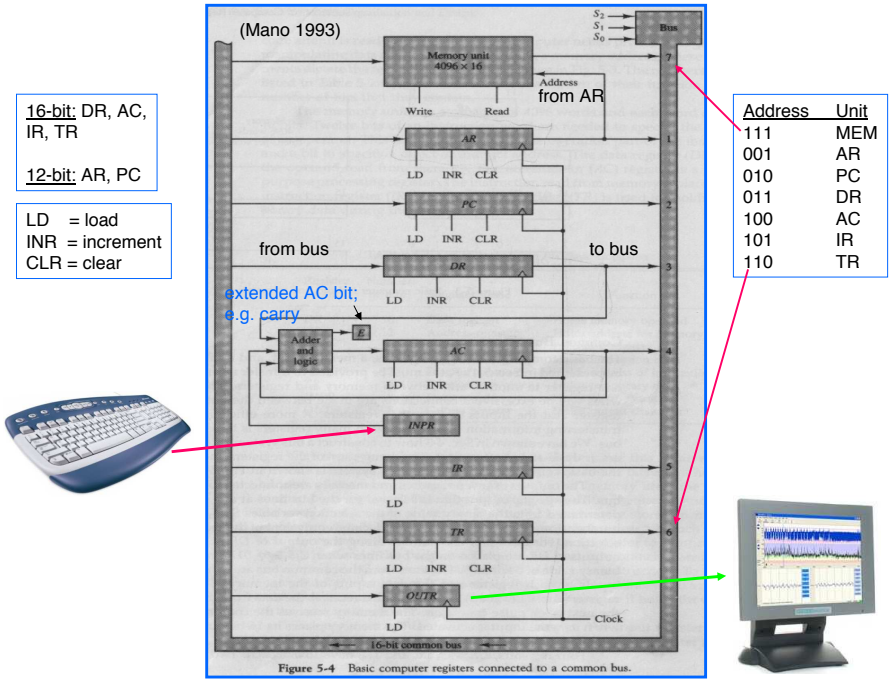


Figure 5-3 Basic computer registers and memory.

- The basic computer (introduced in this course) has (see Fig. 5-3):
  - 8 registers
  - 1 memory unit
  - 1 control unit
  - Common bus
- The outputs of 7 registers and memory are connected to the common bus.
- Connections to bus lines are specified by selection lines S0, S1, and S3.
- A register load during the next clock pulse transition is selected with a LD (load) input.
- Memory write/read is enabled with write/read signals.



- *INPR* receives a character from an input device.
- *OUTR* receives a character from *AC* and delivers it to an output device.
- Bus receives data from 6 registers and the memory unit.
- 5 registers have three control lines: LD (load), INR (increment), and CLR (clear): equivalent to a binary counter with parallel load and synchronous clear. 2 registers have only a LD input.
- AR is used to specify memory address: no need for an address bus.
- 16 inputs to AC come from an adder and logic circuit with three sets of inputs: *AC* output, *DR*, *INPR*.

- Content of any register can be applied onto the bus, and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated register and the output of the adder and logic circuit into *AC, e.g.:*

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

AC to the bus (S2S1S0 = 100), enabling the LD of *DR*, transferring *DR* into *AC* (through adder and logic unit), and enabling LD of *AC*, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

Computer Instructions

- The "basic computer" has three 16-bit instruction code formats (see. Fig. 5-5).
- Opcode contains 3 bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A memory-reference instruction uses 12 bits to specify address and one bit to specify addressing mode *I*.
- The register-reference instructions are recognized by opcode 111 with 0 in bit 15.
- A register-reference instruction specifies an operation on or test of the *AC* register. An operand is not needed: 12 bits are used for specifying the operation or test to be executed.

- Input-output instruction is recognized by the opcode 111 with 1 in bit 15. Remaining 12 bits are used to specify the type of input-output operation or test performed.
- Bits 12-15 are used to recognize the type of instruction.
- If Bits 12-14 are not 111 the instruction is a memory-reference type: *I* (bit 15) is taken as the addressing mode.
- If bits 12-14 are 111, bit 15 is inspected for the type of instruction: 0 for register-reference and 1 for input-output instruction.
- 25 instructions (see. Table 5-2).

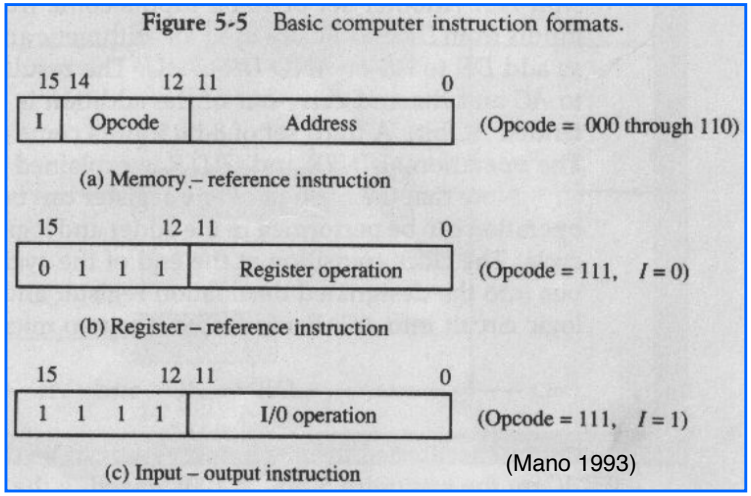


Figure 5-5 Basic computer instruction formats.

(Mano 1993)

Remember:
each hexadecimal digit corresponds 4-bit binary number!
e.g. 7 => 0111 and
F => 1111

TABLE 5-2 Basic Computer Instructions (Mano 1993)

address part

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

memory-reference instructions

register-reference instructions

Input-output instructions

- Instruction set completeness: sufficient set of instructions for computing any function known to be computable. Three categories of instructions:
  1. Arithmetic, logical, and shift instructions
  2. Instructions for moving information to and from memory and processor registers
  3. Program control instructions together with instructions that check status conditions
  4. Input and output instructions

- Arithmetic, logical, and shift instructions provide computational capabilities for processing data.
- All computation are done in processor registers: instructions for moving information between memory and registers are needed.
- Status checking (e.g. comparing magnitudes of two numbers) and program control instructions (e.g. branch) for altering the program flow.
- Input and output instructions for human-computer interaction: programs must be transferred into memory and the results of computations must be transferred to the user.
- Instructions in Table 5-2 constitute a minimum set.

- Addition and subtraction: ADD, CMA, INC.
- Shifts: CIR, CIL
- Multiplication and division: addition, subtraction, and shift.
- Logic: AND, CMA, CLA => NAND => all logic operations with two variables.
- Moving information: LDA, STA.
- Branching and status checking: BUN, BSA, ISZ, and skip operations.
- Input-output: INP, OUT

- The instruction set of the "basic computer" is complete, but not efficient.
- An efficient set of instructions includes separate instructions for frequently used operations in order to perform them fast. Examples: OR, exclusive-OR, subtract, multiply, divide. These operation must be programmed in the "basic computer".

Timing and Control

- Timing for all registers is controlled by a master clock generator.
- Clock is applied to all flip-flops and registers in the system.
- Clock pulses do not change the state of a register unless it is enable by a control signal generated in the control unit.

- There are two major types of control organization: hardwire control and microprogrammed control
  - Hardwire organization (see Fig. 5-6): the control logic is implemented with gates, flip-flops, decoders, and other digital circuits
    - Can be optimized to produce fast mode of operation
    - Requires changes in the wiring if the design has to be modified
  - Microprogrammed organization: the control information is stored in a control memory (store)
    - The control memory is programmed to initiate the required sequence of micro-operations
    - Any required modifications can be done by updating the micro-program in control memory.

A microprogram is a program consisting of microcode that controls the different parts of a computer's central processing unit (CPU). The memory in which it resides is called a control store.

- Block diagram of the (hardwire) control unit is shown in Fig. 5-6 (control logic derived later)
  - *IR* contains an instruction read from memory
    - three parts: I-bit, opcode, bits 0-11
    - Opcode is decoded with a 3 x 8 decoder (outputs D0-D7)
  - *I* is transferred to a flip-flop
  - 4-bit sequence counter (*SC*) provide the sequence of 16 timing signals
    - synchronous clear and increment
    - When required, SC can be cleared (CLR signal enabled) by a suitable control logic, e.g. (see Fig. 5-7):

$$D_3T_4: SC \leftarrow 0$$

- Control outputs are a function of all incoming signals to the control logic gates. *SC* enables sequential control outputs.

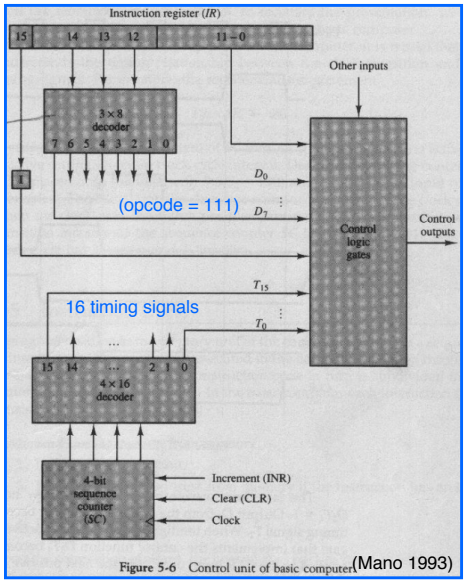


Figure 5-6 Control unit of basic computer (Mano 1993)

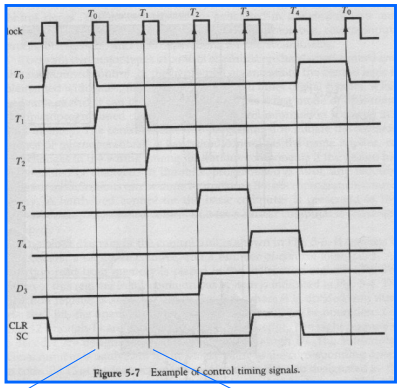


Figure 5-7 Example of control timing signals.

timing signal is active for one clock cycle

control outputs = F(control inputs), where control inputs are a function of time due to sequence counter (*SC*)
=> an instruction can produce a sequence of control signals, although it itself remains constant in an instruction register (*IR*).

- Memory read/write are initiated by a rising clock edge.
- It is assumed that memory access is completed in one clock cycle
  - assumption is often not valid in real computers because the memory cycle is usually longer that the clock cycle => wait cycles (states) must be provided until the memory word is available.
  - No wait cycles in "basic computer" introduced here.
- Next rising edge will load the memory word into a register.

- It is important to understand the timing relationship between clock transition and the timing signals.
- For example, the register transfer statement:

  $$T_0: AR \leftarrow PC$$

  specifies a transfer of the content *PC* into *AR* if the timing signal T_0 is active. T_0 is active an entire clock cycle. During this time interval the content of PC is placed onto the bus and LD input of AR is enabled. The actual transfer occurs at the end of the clock cycle when the clock goes through a positive transition (latches inputs to flip-flops). This same transition increments SC: the next clock cycle has T_1 active and T_0 inactive.

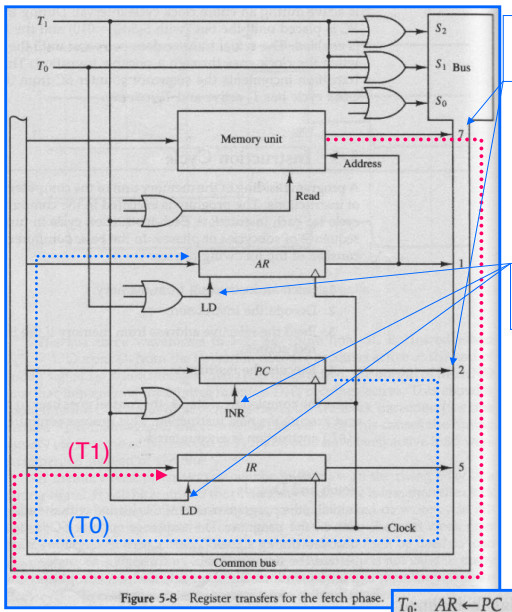
Instruction Cycle

- A program consists of a sequence of instruction, and it resides in the memory.
- Each instruction cycle in "basic computer" has following phases:
  1. Fetch an instruction from memory
  2. Decode the instruction
  3. Read the effective address from memory if instruction defines an indirect address
  4. Execute the instruction
- After phase 4, the control jumps back to phase 1. This process continues until HALT instruction is encountered.

Fetch and Decode

- Initially program counter *PC* in loaded with the address of the first instruction in the program.
- *SC* is cleared (i.e. timing signal T_0 is active). *SC* is incremented after each clock pulse.
- Fetch and decode phases can be specified by following register transfer statements:

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$



T0: 010 => 2
T1: 111 => 7

T0: LD (AR)
T1: LD (IR), INR (PC)

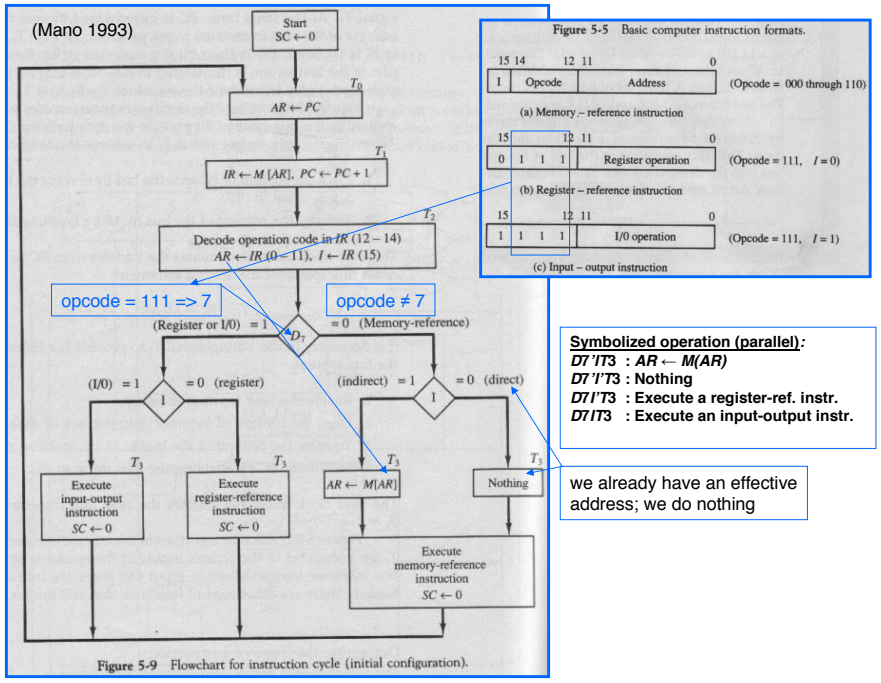
Figure 5-8 Register transfers for the fetch phase.

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$

- during T0:
  1. Place the content of PC onto bus (S2S1S0 = 010 => 2)
  2. Transfer the content of the bus to AR (enable LD input of AR)
  - The next clock transition initiates transfer from PC to AR
- during T1:
  1. Enable the read input of memory
  2. Place the content of memory onto the bus (S2S1S0 = 111 => 7)
  3. Transfer the content of bus to IR (enable LD input of IR)
  4. Increment PC (enable INR input of PC)
  - The next clock transition initiates the read and increment operations
- during T2:
  1. Opcode is decoded by the 3 x 8 decoder
  2. IR(0-11) is transferred to AR (address register)
  3. IR(15) is latched to flip-flop I
  - 2 and 3 occur at the end of the clock cycle

Determine the Type of Instruction

- Timing signal is T3 (after decoding)
- During T3, the control unit determines the type instruction that was just read from memory (see Fig. 5-9).
- After the instruction has been executed SC is cleared and control returns to fetch phase with T0 = 1.
- It is assumed (not explicitly shown in transfer statements) that SC is incremented with every positive clock transition.
- When SC is cleared, $SC \leftarrow 0$ statement is included.



Register-Reference Instruction

- Recognized by the control when $D7 = 1$ and $I = 0$.
- Uses bits 0-11 of the instruction code to specify one of 12 instructions.
- The 12 bits are available in *IR*(0-11) and they were transferred to *AR* during time *T*2.
- See Table 5-3 for control functions and microoperations for the register-reference instructions.
  - Each control function share Boolean relation $D7I'T3$ (denoted by *r*)
  - The particular control function is indicated by one of the bits in IR(0-11)
  - The execution of a register-reference instruction is completed at time *T*3: the sequence counter is cleared to 0 and control goes back to fetch the next instruction with timing signal *T*1.

TABLE 5-3 Execution of Register-Reference Instructions (Mano 1993)

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

must be recognized as part of control conditions!

stops *SC* from counting

Memory-Reference Instructions

- Table 5-4 lists the seven memory-reference instructions: the execution of each instruction requires a sequence of microoperations because data is stored in memory and cannot be processed directly.
- The effective address resides in AR and was placed there during timing signal $T2$ when $I = 0$, and $T3$ when $I = 1$ (see Fig. 5-9).

TABLE 5-4 Memory-Reference Instructions (Mano 1993)

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

- AND to AC: pair wise AND to bits in AC and the memory word specified by the effective address

D_0T_4: $DR \leftarrow M[AR]$
D_0T_5: $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

output of the operation decoder = 0

- ADD to AC: adds the content of the memory word specified by the effective address to the value of AC

D_1T_4: $DR \leftarrow M[AR]$
D_1T_5: $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$

extended accumulator

output of the operation decoder = 1

- LDA: load a memory word from a specified effective address to AC

D_2T_4: $DR \leftarrow M[AR]$
D_2T_5: $AC \leftarrow DR, \quad SC \leftarrow 0$

  - See Fig. 5-4: no direct path from the bus to *AC*: memory word is first read into *DR* whose content is then transferred into *AC*.
- STA: store the content of AC into the memory word specified by the effective address

D_3T_4: $M[AR] \leftarrow AC, \quad SC \leftarrow 0$

- BUN: branch unconditionally – transfers the program to the instruction specified by the effective address. The next instruction is fetched and executed from the memory address given by the new value in *PC*

D_4T_4: $PC \leftarrow AR, \quad SC \leftarrow 0$

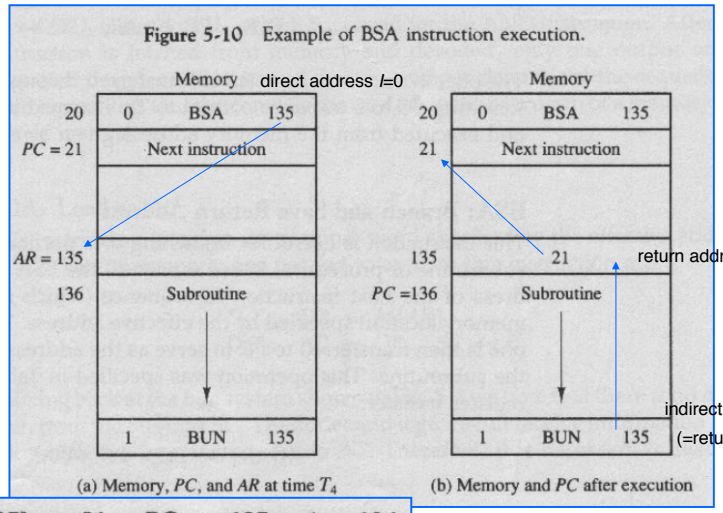
- BSA: branch and save return address – this instruction is useful for branching to a portion of a program called a subroutine or procedure

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

address of the next instruction in sequence (return address)

address of the first instruction in the subroutine

- Numerical example of BSA instruction (subroutine call):



$M[135] \leftarrow 21, \quad PC \leftarrow 135 + 1 = 136$

- The BSA instruction performs the function usually referred to as a subroutine call.
- The indirect BUN instruction at the end of the subroutine performs the function referred as a subroutine return.
- In most commercial computers, the return address associated with the subroutine is stored in either a processor register of in a portion of memory called a stack.

A stack is a data structure that works on the principle of Last In First Out (LIFO). This means that the last item put on the stack is the first item that can be taken off, like a physical stack of plates.
A stack-based computer system is one that is based on the use of stacks, rather than being register based.

- The BSA instruction must be executed with a sequence of two microoperations:

D_5T_4: $M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$
D_5T_5: $PC \leftarrow AR, \quad SC \leftarrow 0$

  - Timing signal *T*4 initiates a memory write operation, places the content of *PC* onto the bus, and enables the INR input of *AR*.
  - Memory write operation is completed and *AR* is incremented by the time the next clock transition occurs.
  - The bus is used at *T*5 to transfer the content or *AR* to *PC*.

- ISZ: increment the word specified by the effective address, and if incremented value is equal to 0, PC is incremented by 1

D_6T_4: $DR \leftarrow M[AR]$
D_6T_5: $DR \leftarrow DR + 1$
D_6T_6: $M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$

  - Programmer usually stores a negative number (in 2's complement) in the memory word. Repeated increments will eventually clear the memory word to 0. At that time PC is incremented by one in order to skip the next instruction in the program => can be used to create loops (shown later).

(Mano 1993)

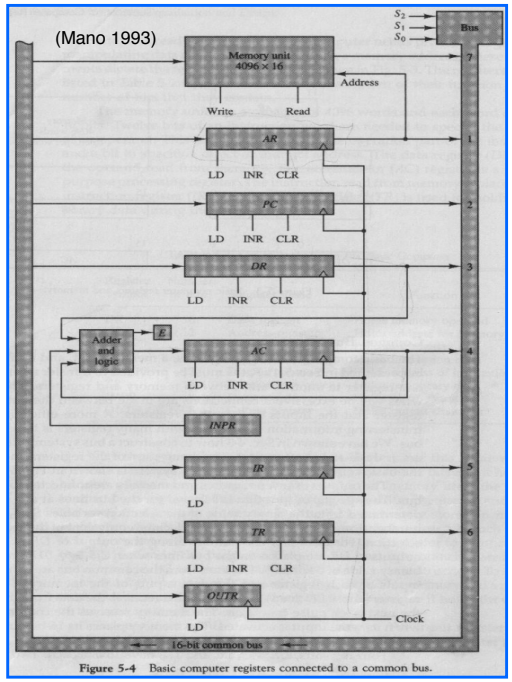
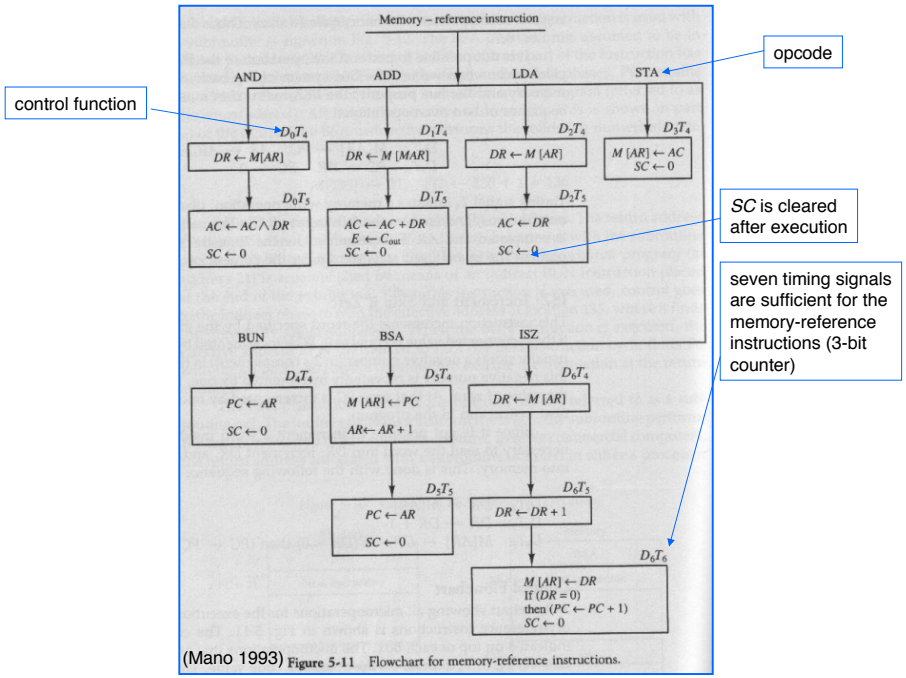


Figure 5-4 Basic computer registers connected to a common bus.

- Flow chart showing microoperations for the seven memory-reference instructions is shown in Fig. 5-11.



(Mano 1993) **Figure 5-11** Flowchart for memory-reference instructions.

Input-Output and Interrupt

- Computer has to communicate with the external environment is order to be useful: instructions and data come from some input device.
- Computational results must be transmitter to the user through some output device.
- Most basic requirements for input and output communication will be illustrated with a terminal unit with a keyboard and printer (see Fig. 5-12).

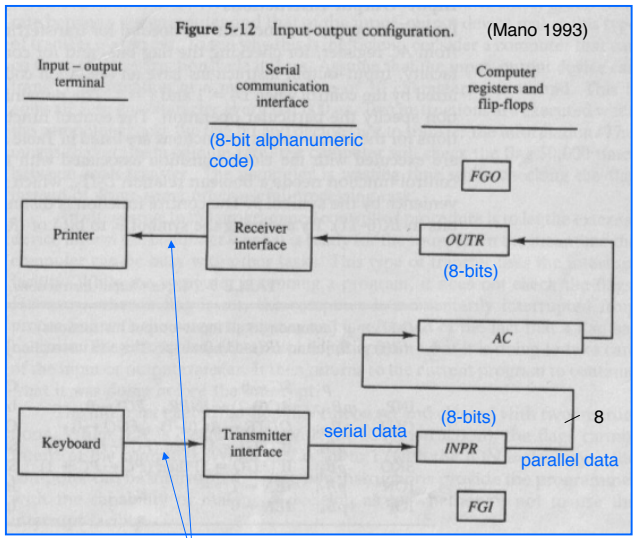
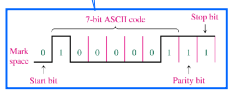


Figure 5-12 Input-output configuration. (Mano 1993)

Serial data *e.g.*:



- INPR and OUTR communicate serially (8-bit alphanumeric code) with transmitter, and receiver interfaces, respectively.
- Connection to AC is parallel (8 significant bits).
- 1-bit FGI=1 when INPR holds new data. FGI=0 when data has been accepted (read) by the computer => needed for synchronizing the timing rate difference between the input device and the computer
  - When a key is pressed 8-bit code is sent to INPR and FGI is set to 1.
  - Computer checks the FGI. If it is 1, computer reads the new data and clears the flag (FGI=0): new data can be shifted into INPR.

- OUTR works similarly but the direction of information flow is reversed
  - Initially FGO=1
  - Computer checks the FGO; if it is 1, the data from AC is transferred in parallel to OUTR and FGO is cleared to 0.
  - The output device accepts the coded data, prints the character, and when completed sets the FGO to 1.
  - The computer does not load a new character to OUTR as long as FGO is 0: output device is processing (printing) the last character.

- Input and output instructions are needed for checking the flag bits (FGO and FGI), and for controlling and interrupt the interrupt facility.
- The control functions and microoperations for input-output instructions are listed in Table 5-5.
- All control functions share the Boolean relation $D7IT3$, which is designated, for convenience, with p.

TABLE 5-5 Input-Output Instructions (Mano 1993)

$D_7IT_3 = p$ (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

e.g. bit 6 of *IR*

Program interrupts

- Interrupts were originated to avoid wasting the computer's valuable time in software loops (called polling loops) waiting for electronic events.
- The computer was able to do other useful work while the event was pending.
- The interrupt would signal the computer when the event occurred, allowing efficient accommodation for slow mechanical devices.

- Interrupts remain in modern computers because they permit a computer to have prompt responses to electronic events, while performing other work.
- Typically, the user can configure the machine using hardware registers so that different types of interrupts are enabled or disabled, depending on what the user wants
  - the interrupt signals are AND'ed with a mask, thus allowing only desired interrupts to occur. Some interrupts cannot be disabled - these are referred to as nonmaskable interrupts.

- In "Basic Computer" the interrupt enable flip-flop *IEN* can be set (ION) or cleared (IOF) to enable, or disable interrupts, respectively.

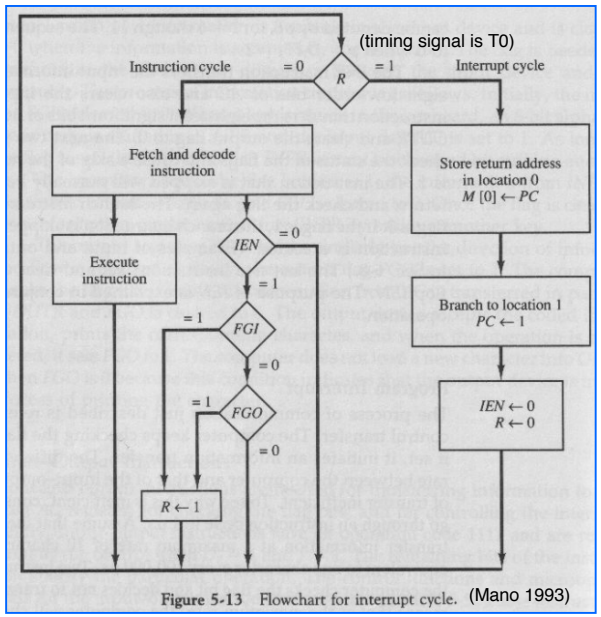


Figure 5-13 Flowchart for interrupt cycle. (Mano 1993)

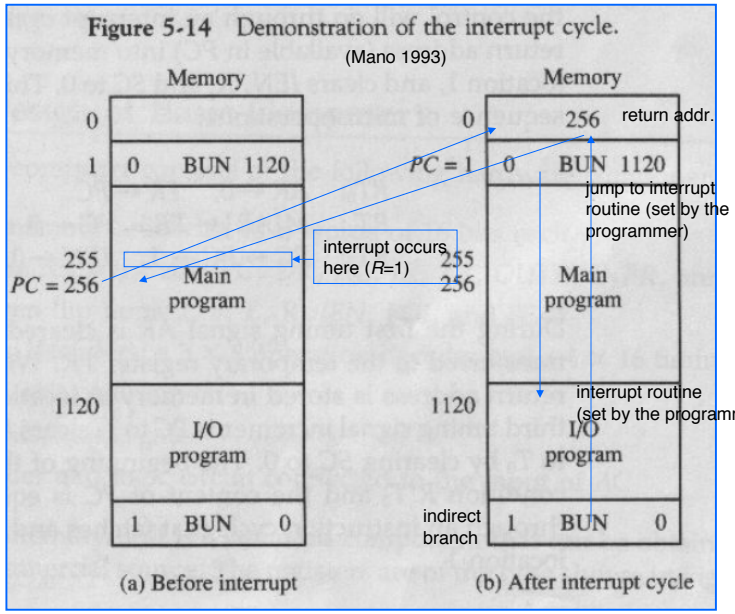


Figure 5-14 Demonstration of the interrupt cycle. (Mano 1993)

E.g.: Interrupt vectors of Atmel's 8-bit microcontroller ATmega32

```
Address   Labels   Code            Comments
$000              jmp   RESET      ; Reset Handler
$002              jmp   EXT_INT0   ; IRQ0 Handler
$004              jmp   EXT_INT1   ; IRQ1 Handler
$006              jmp   EXT_INT2   ; IRQ2 Handler
$008              jmp   TIM2_COMP  ; Timer2 Compare Handler
$00A              jmp   TIM2_OVF   ; Timer2 Overflow Handler
$00C              jmp   TIM1_CAPT  ; Timer1 Capture Handler
$00E              jmp   TIM1_COMPA ; Timer1 CompareA Handler
$010              jmp   TIM1_COMPB ; Timer1 CompareB Handler
$012              jmp   TIM1_OVF   ; Timer1 Overflow Handler
$014              jmp   TIM0_COMP  ; Timer0 Compare Handler
$016              jmp   TIM0_OVF   ; Timer0 Overflow Handler
$018              jmp   SPI_STC    ; SPI Transfer Complete Handler
$01A              jmp   USART_RXC  ; USART RX Complete Handler
$01C              jmp   USART_UDRE ; UDR Empty Handler
$01E              jmp   USART_TXC  ; USART TX Complete Handler
$020              jmp   ADC        ; ADC Conversion Complete Handler
$022              jmp   EE_RDY     ; EEPROM Ready Handler
$024              jmp   ANA_COMP   ; Analog Comparator Handler
$026              jmp   TWI        ; Two-wire Serial Interface Handler
$028              jmp   SPM_RDY    ; Store Program Memory Ready Handler
```

(Atmel's ATmega32)

Table 18. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

Bit	7	6	5	4	3	2	1	0	
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TWIE bit in *TWCR* register enables TWI interrupt

- Interrupt cycle
  - the condition for setting flip-flop R can be expressed:

$$T_0' T_1' T_2' (IEN)(FGI + FGO): \quad R \leftarrow 1$$

where *T*0'*T*1'*T*2' indicates a timing signal after the fetch and decode phase.

  - Modified fetch phase (valid for $R = 1$, the standard fetch phase is valid for $R = 0$ => *R'*)

$$\begin{aligned} RT_0: &\quad AR \leftarrow 0, \quad TR \leftarrow PC \\ RT_1: &\quad M[AR] \leftarrow TR, \quad PC \leftarrow 0 \\ RT_2: &\quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0 \end{aligned}$$

$$\begin{aligned} R'T_0: &\quad AR \leftarrow PC \\ R'T_1: &\quad IR \leftarrow M[AR], \quad PC \leftarrow PC + 1 \\ R'T_2: &\quad D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14), \quad AR \leftarrow IR(0\text{–}11), \quad I \leftarrow IR(15) \end{aligned}$$

Complete Computer Description

- The final flowchart of the instruction cycle for the basic computer is shown in Fig. 5-15.
- Interrupt flip-flop (flag) can be set at any time after timing signal *T*2.
- After *SC* is cleared to 0, control returns to timing signal *T*0.
- When R=1 we have the interrupt cycle.
- When R=0 we have an instruction cycle.

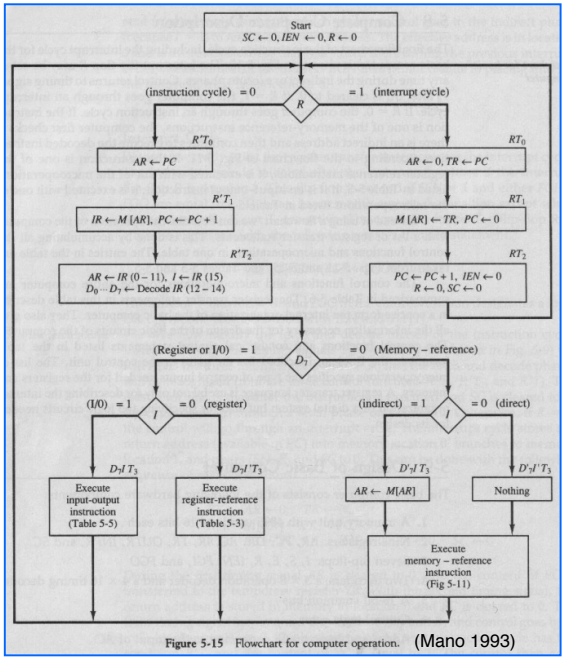


Figure 5-15 Flowchart for computer operation. (Mano 1993)

- The control functions and microoperations for the "Basic Computer" are summarized in Table 5-6.
- The register transfer statements describe in a concise form the internal organization of the basic computer.
- The control functions and conditional control statements formulate the Boolean functions for the gates in the control unit.
- The list of microoperations specifies the type of control inputs needed for the register and memory.

TABLE 5-6 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	D_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:		
	$T_0T_1T_2(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		
		$D_7I'T_3 = r$ (common to all register-reference instructions)
		$IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
		$D_7IT_3 = p$ (common to all input–output instructions)
		$IR(i) = B_i$ $(i = 6, 7, 8, 9, 10, 11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

(Mano 1993)

Design of Basic Computer

- Hardware components
  1. A memory unit with 4096 words of 16 bits each (4096x16).
  2. 9 registers: *AR*, *PC*, *DR*, *AC*, *IR*, *TR*, *OUTR*, *INPR*, and *SC*
     - the registers are similar to 74164 IC:
       http://www.alldatasheet.com/datasheet-pdf/pdf/FAIRCHILD/74163.html
  3. 7 flip-flops: *I*, *S*, *E*, *R*, *IEN*, *FGI*, and *FGO*.
  4. 2 decoders: a 3x8 operation decoder and a 4 x 16 timing decoder.
  5. A 16-bit common bus
     - can be *e.g.* constructed with sixteen 8x1 multiplexers (Fig. 4-3).
  6. Control logic gates
  7. Adder and logic circuit connected to the input of *AC*.

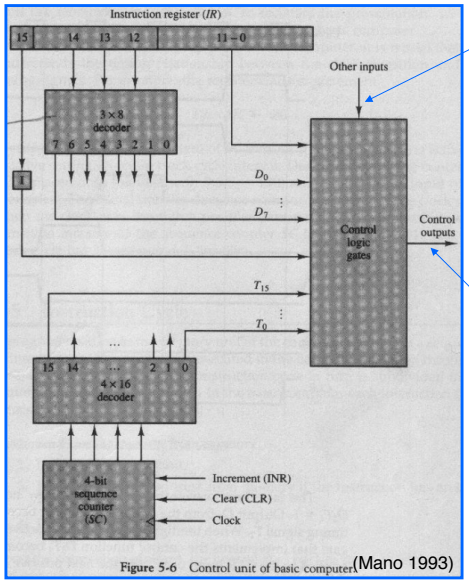


Figure 5-6 Control unit of basic computer

(Mano 1993)

Other inputs to control and logic gates: AC bits 0-15, DR bits 0-15, seven flip-flops.

Outputs:
1. Signals to control the inputs of the 9 registers
2. Control inputs for memory reads/writes
3. Signals to set, clear, or complement the flip-flops.
4. Signals for *S2*, *S1*, and *S0* to select a register for the bus.
5. Signals to control the *AC* adder and logic circuit.

- Control of Registers and Memory
  - The control inputs of registers are LD (load), INR (increment), and CLR (clear).
  - Suppose that we want to derive a gate structure for controlling the inputs of *AR*.
  - From Table 5-6: all statements that change the content or *AR* (*i.e.* involve signals for LD,INR,CLR inputs of *AR*):

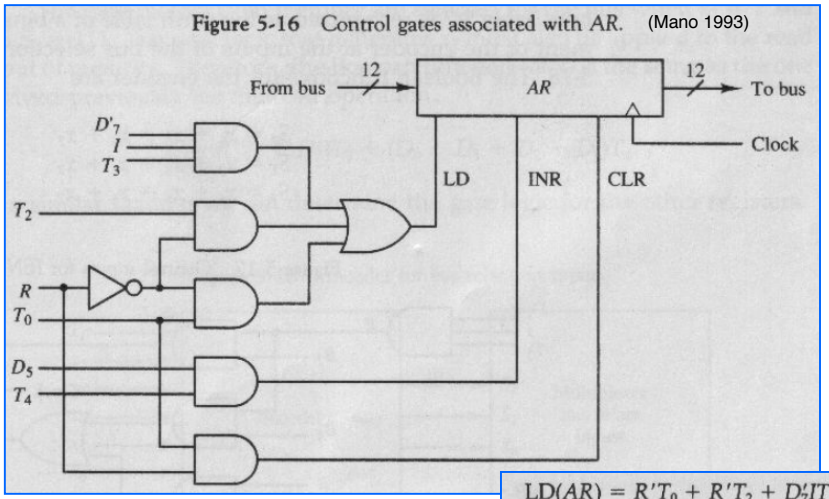
$R'T_0$:	$AR \leftarrow PC$	LD
$R'T_2$:	$AR \leftarrow IR(0\text{–}11)$	LD
$D_7'IT_3$:	$AR \leftarrow M[AR]$	LD
RT_0:	$AR \leftarrow 0$	CLR
D_5T_4:	$AR \leftarrow AR + 1$	INR

- We can combine the control functions into three Boolean expressions (one for each control input of *AR*):

$$\begin{aligned} \text{LD}(AR) &= R'T_0 + R'T_2 + D_7'IT_3 \\ \text{CLR}(AR) &= RT_0 \\ \text{INR}(AR) &= D_5T_4 \end{aligned}$$

these Boolean functions can be translated directly into a control gate (combinational logic) (Fig. 5-16)

Figure 5-16 Control gates associated with AR. (Mano 1993)



$$\begin{aligned} \text{LD}(AR) &= R'T_0 + R'T_2 + D_7'IT_3 \\ \text{CLR}(AR) &= RT_0 \\ \text{INR}(AR) &= D_5T_4 \end{aligned}$$

- In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory.
- Memory read/write:
  - From table 5-16 (read is recognized from $\leftarrow M[AR]$):

$$\text{Read} = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

  - The output (Read) of the Boolean function above is connected to the read input of memory.

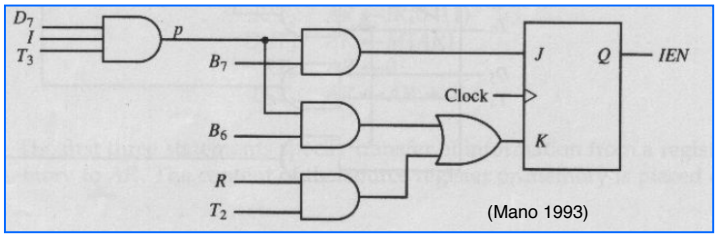
- Control of single flip-flops is derived in a similar manner, e.g. *IEN*:
  - From Table 5-6 (ION and IOF may change the *IEN*)

pB_7:	$IEN \leftarrow 1$
pB_6:	$IEN \leftarrow 0$
RT_2:	$IEN \leftarrow 0$

$p = D7IT3$

at the end of interrupt cycle IEN is cleared

  - The corresponding control gate logic is shown in Fig. 5-17.



(Mano 1993)

- Control of common bus
  - 16-bit bus is controlled by selection inputs S2, S1, and S0 (Table 5-7):

truth table of binary encoder

TABLE 5-7 Encoder for Bus Selection Circuit (Mano 1993)

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	*AR*
0	1	0	0	0	0	0	0	1	0	*PC*
0	0	1	0	0	0	0	0	1	1	*DR*
0	0	0	1	0	0	0	1	0	0	*AC*
0	0	0	0	1	0	0	1	0	1	*IR*
0	0	0	0	0	1	0	1	1	0	*TR*
0	0	0	0	0	0	1	1	1	1	Memory

$S_0 = x_1 + x_3 + x_5 + x_7$
$S_1 = x_2 + x_3 + x_6 + x_7$
$S_2 = x_4 + x_5 + x_6 + x_7$

Boolean functions for the encoder

multiplexer bus select inputs

- To determine the logic for encoder inputs (x1...x7), the control functions that place the corresponding register onto the bus must be find: each encoder input signal (x1…x7) corresponds a one register (or memory) connected to the common bus.
- *E.g.*: find a logic that makes x1 = 1 (*i.e.* connects AR onto the bus)
  - From register transfer statements in Table 5-6 we select the statements that have *AR* as a source (we do not have to consider the cases in which *AR* is a destination because those cases are handled with LD control input of *AR*):

D_4T_4: $PC \leftarrow AR$
D_5T_5: $PC \leftarrow AR$

=> the Boolean function for x1 is

$$x_1 = D_4T_4 + D_5T_5$$

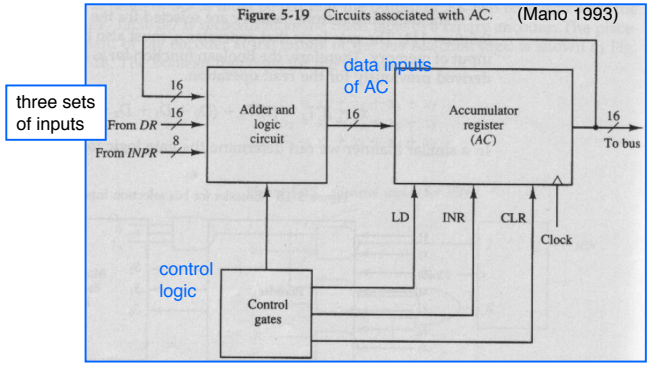
- The data output from memory is selected for the bus when x7=1, *i.e.* S2S1S0 = 111.
- The gate logic for x7 must also be applied to the read input of memory (we are reading memory when it's outputs are selected onto the bus) => Boolean function is the same as the one derived previously for the read operation:

$$\text{Read} = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

- Gate logic for other registers can be derived in a similar manner.

Design of Accumulator Logic

- The circuits associated with AC register are shown in Fig. 5-19:

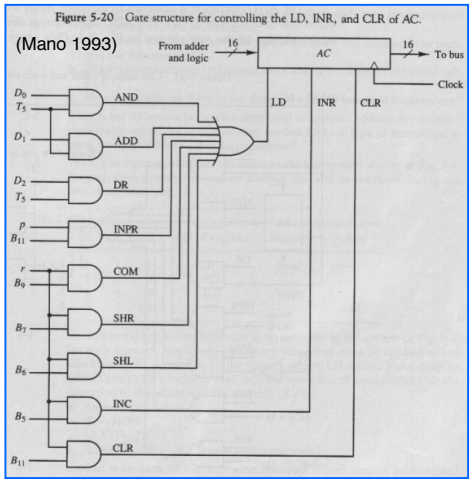


- In order to design control logic associated with *AC* we need to find the register transfer statements (from Table 5-6) that change the content of *AC*, *i.e.* utilize control inputs LD, INR, or CLR of *AC*:

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

- From the list above we can derive the control logic gates (Fig. 5-20) and the adder and logic circuit.

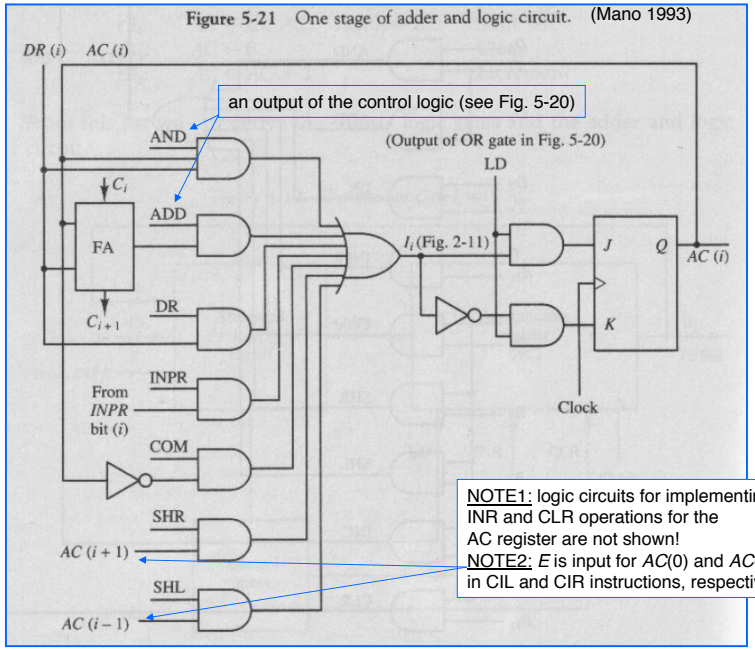
- Control of *AC* register
  - The logic controlling the LD,INR, and CLR inputs of AC is shown in Fig. 5-20



$p = D7IT3$
$r = D7I'T3$

- Adder and Logic Circuit
  - The adder and logic circuit can be subdivided into 16 stages.
  - Each stage corresponds one bit of AC.
  - Internal construction of the register is as show in Fig. 2-11.
  - Figure 5-21 shows one adder and logic circuit stage for AC (or gates removed: required for CLR and INR operations, and are shown in Fig. 2-11).
  - One stage consist of:
    - 7 AND gates
    - 1 OR gate
    - 1 full-adder (FA)

Figure 5-21 One stage of adder and logic circuit. (Mano 1993)



J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

NOTE1: logic circuits for implementing INR and CLR operations for the AC register are not shown!
NOTE2: *E* is input for *AC*(0) and *AC*(15) in CIL and CIR instructions, respectively.