Instruction format

- In general, based on the number of operands or reference made in the instructions. Instructions can be classified into the following types: -
  - 3 Address Instruction
  - 2 Address Instruction
  - 1 Address Instruction
  - 0 Address Instruction

3 Address Instruction

Mode/Opcode	Destination	Source$_1$	Source$_2$

- Three-address instruction is a format of machine instruction. It has one opcode and three address fields. One address field is used for destination and two address fields for source.

- X = (A + B) x (C + D)

ADD R_1, A, B	R_1 <- M[A] + M[B]
ADD R_2, C, D	R_2 <- M[C] + M[D]
MUL X, R_1, R_2	M[X] <- R_1 x R_2

- It produces short program much easier to understand, but it does not mean that program will run much faster because now instruction only contain more information but each micro operation (changing content of register, loading address in address bus etc.) will be performed in one cycle only.

- Many bits are needed for instructions encoded in binary to define three addresses. More advanced processing and decoding circuits are needed.

2 Address Instruction

Mode/Opcode	Destination/$Source_1$	$Source_2$

- Here two addresses can be specified in the instruction. In effort to reduce the size of instruction here we remove the reference for result and $Source_1$ is used for storing the result.
- X = (A + B) x (C + D)

MOV	R_1, A	$R_1 \leftarrow M[A]$
ADD	R_1, B	$R_1 \leftarrow R_1 + M[B]$
MOV	R_2, C	$R_2 \leftarrow C$
ADD	R_2, D	$R_2 \leftarrow R_2 + D$
MUL	R_1, R_2	$R_1 \leftarrow R_1 * R_2$
MOV	X, R	$M[X] \quad R$

- Less length compares to 3 address, more efficient, no of register required are less.

- Here code optimization is relatively difficult.

1 Address Instruction

Mode/Opcode	Destination/Source

- One address instruction uses an implied accumulator (AC) register for all data manipulation.
- One operand is in the accumulator and the other is in the register or memory location. Implied means that the CPU already knows that one operand is in the accumulator so there is no need to specify it.
- X = (A + B) x (C + D)

LOAD	A	AC ← M[A]
ADD	B	AC ← AC + M[B]
STORE	T	M[T] ← AC
LOAD	C	AC ← M[C]
ADD	D	AC ← AC + M[D]
MUL	T	AC ← AC * M[T]
STORE	X	M[X] ← AC

0 Address Instruction

Mode/Opcode

- Early computers which do not have complex circuit like ALU use few registers which are used as organised stack. While working on these stacks we do not require location of result as it is implied at the top of the stack. A stack organized computer does not use an address field for the instructions ADD and MUL. The push and pop instruction, however, need an address field to specify the operand that communicates with stack.
- X = (A + B) x (C + D)

PUSH	A	TOP ← A
PUSH	B	TOP ← B
ADD		TOP ← A+B
PUSH	C	TOP ← C
PUSH	D	TOP ← D
ADD		TOP ← C+D
MUL		TOP ← (C+D)*(A+B)
POP	X	M[X] ← TOP

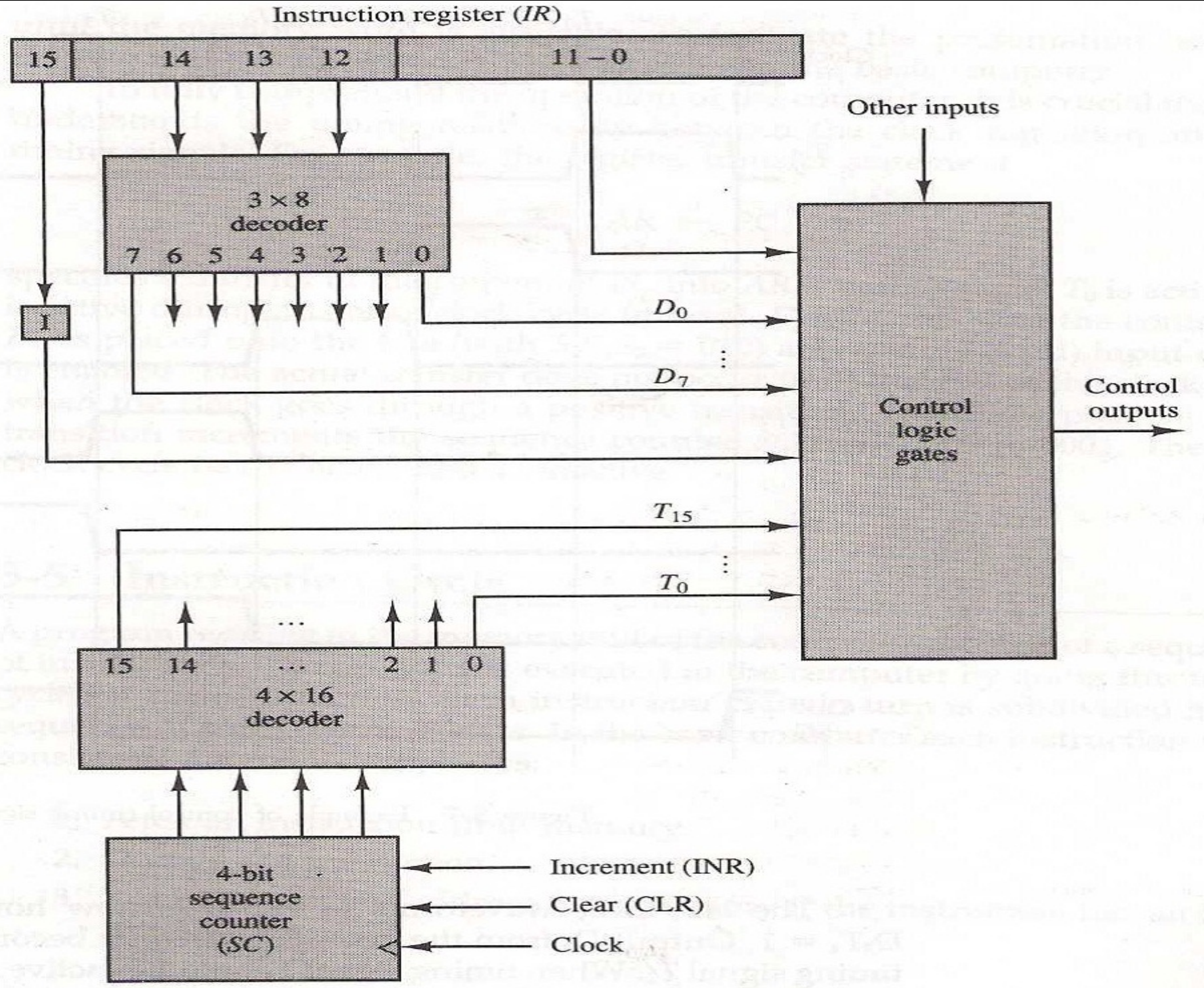
- To evaluate arithmetic expression in a stack computer, it is necessary to Covert the expression into reverse polish notation. The name zero-address is given to this type of computer because of the absences of an address field in the com.

- Here mode/opcode is sufficient to perform any operation.

- It is very simple and very short, but complex to design and speed is very poor.

Timing Circuit

- In order to perform a instruction we need to perform a number of micro-operators, and these micro-operations must be timed
  - $AR \leftarrow PC$
  - $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
- we should distinguish one clock pulse from another during the execution of instruction
- Sequence counter followed by decoder is used to generate time signals


Instruction register (IR)
15
14 13 12
11 – 0
Other inputs
3 × 8 decoder
7 6 5 4 3 2 1 0
D0
⋮
D7
1
Control logic gates
Control outputs
T15
⋮
T0
...
15 14 ... 2 1 0
4 × 16 decoder
4-bit sequence counter (SC)
Increment (INR)
Clear (CLR)
Clock

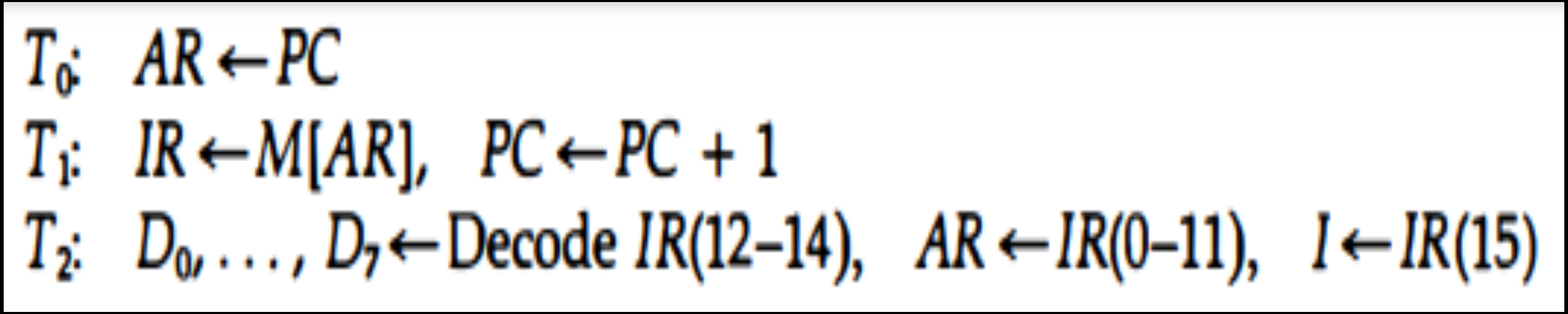
Instruction Cycle - A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases. from fetching of instruction to the completion of execution of instruction whatever happens is called instruction cycle. The reason we called this cycle because it will happen for every instruction.

- Each instruction cycle consists of the following phases:
  - Fetch an instruction from memory.
  - Decode the instruction.
  - Read the effective address from memory if the instruction has an indirect address.
  - Execute the instruction.
  - Fetch and Decode

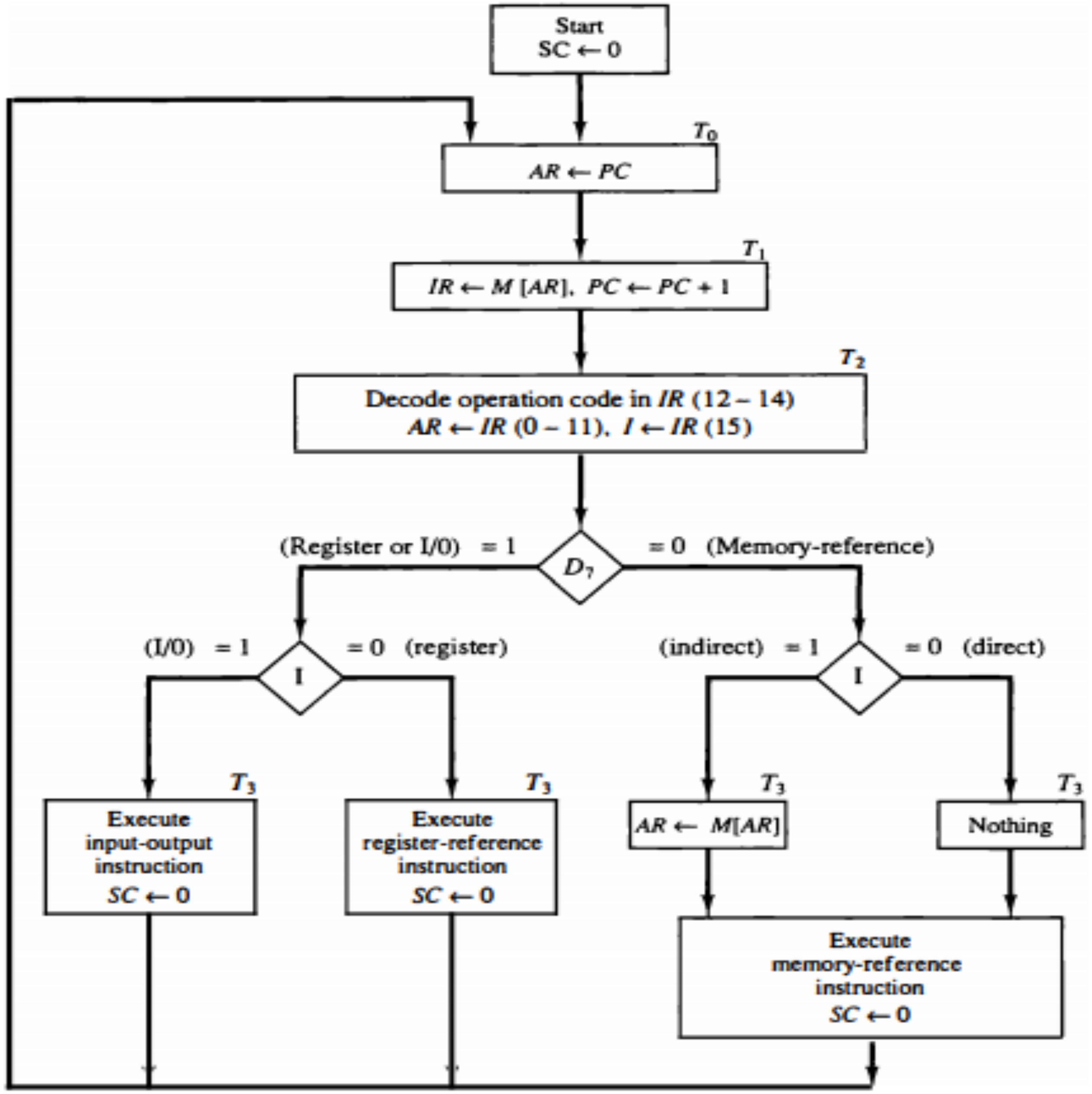
Micro-Operation

- A micro-operation is a simple operation that can be performed during one clock period.
- The result of this operation may replace the previous binary information of register or the result may be transferred to another register.
- Examples of micro-operations are shift, move, count, add and load etc.
- The micro-operations most often encountered in digital computers are classified into four categories:
    - **Register transfer micro-operations**: It transfer binary information from one register to another.
    - **Arithmetic micro-operations**: It perform arithmetic operation on numeric data stored in registers.
    - **Logic micro-operations**: It perform bit manipulation operations on non-numeric data stored in registers.
    - **Shift micro-operations**: It perform shift operations on data stored in registers.

- Initially, the program counter PC is loaded with the address of the first instruction in the program.
- The sequence counter SC is cleared to 0, providing a decoded timing signal T_0.
- After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on.
- The microoperations for the fetch and decode phases can be specified by the following register transfer statement

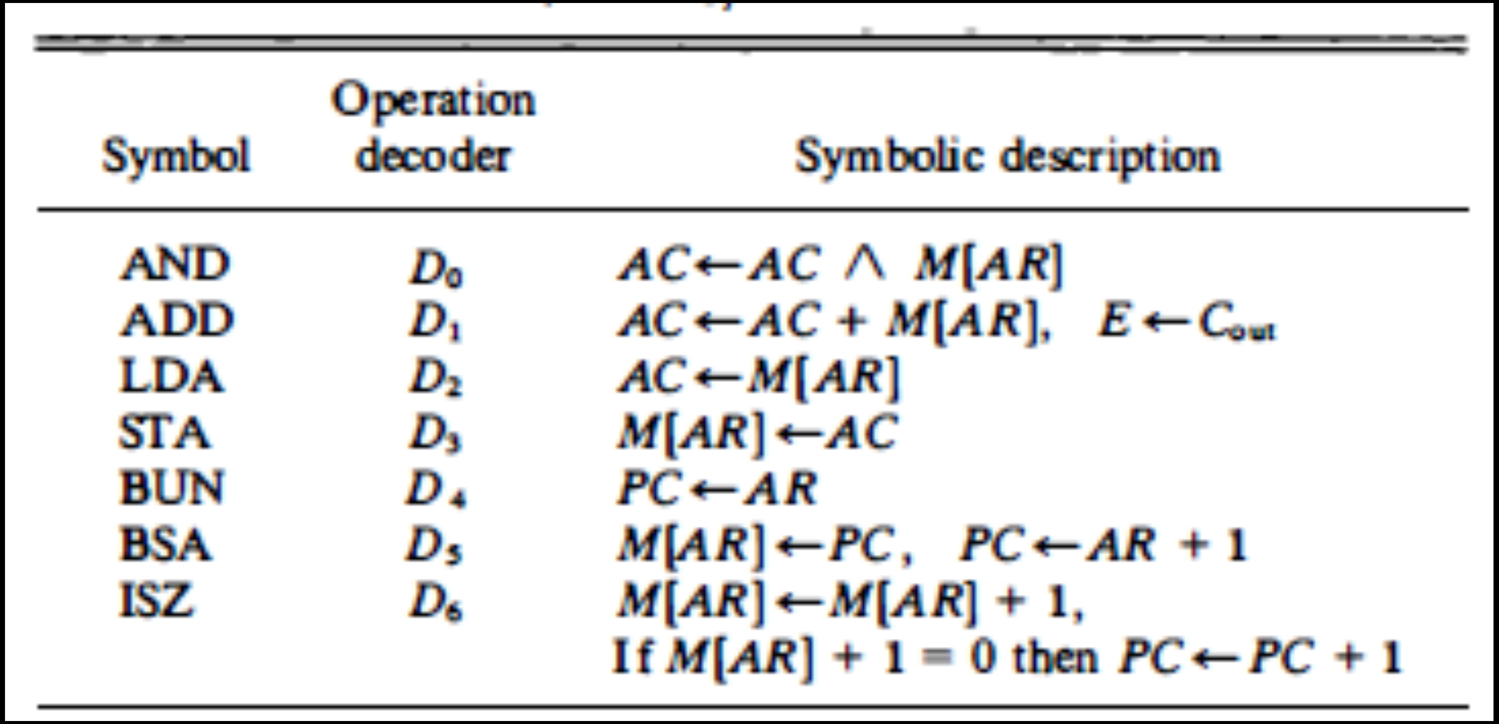


$$\begin{aligned} &T_0: \quad AR \leftarrow PC \\ &T_1: \quad IR \leftarrow M[AR], \quad PC \leftarrow PC + 1 \\ &T_2: \quad D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14), \quad AR \leftarrow IR(0\text{–}11), \quad I \leftarrow IR(15) \end{aligned}$$



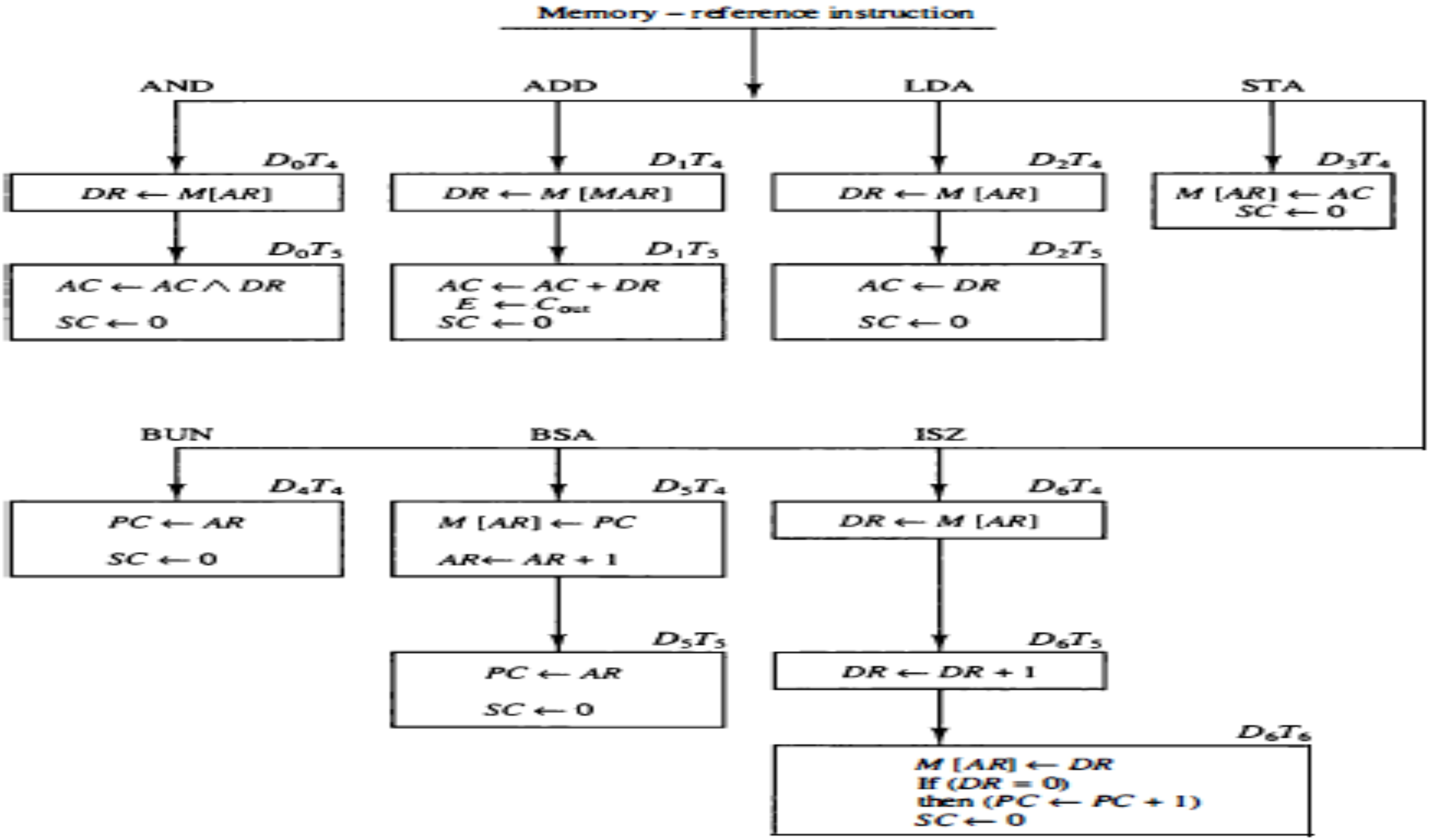
- Determine the Type of Instruction
- The timing signal that is active after the decoding is T_3.
- During time T_3, the control unit determines the type of instruction that was just read from memory.
- Decoder output D, is equal to 1 if the operation code is equal to binary 111.
- If D_7 = 1, the instruction must be a register-reference or input-output type.
- If D_7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.
- Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D_7 = 0 and I = 1, we have a memory reference instruction with an indirect address.
- In case of indirect address, it is necessary to read effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement
- AR ← M [AR]
- When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.
- The sequence counter SC is either incremented or cleared to 0 with every positive clock transition.

Memory-Reference Instructions



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

- The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I = 1. The execution of the memory-reference instructions starts with timing signal T4.
- Operation of each Instruction


Memory – reference instruction
AND
ADD
LDA
STA
D_0T_4
$DR \leftarrow M[AR]$
D_0T_5
$AC \leftarrow AC \wedge DR$
$SC \leftarrow 0$
D_1T_4
$DR \leftarrow M[MAR]$
D_1T_5
$AC \leftarrow AC + DR$
$E \leftarrow C_{out}$
$SC \leftarrow 0$
D_2T_4
$DR \leftarrow M[AR]$
D_2T_5
$AC \leftarrow DR$
$SC \leftarrow 0$
D_3T_4
$M[AR] \leftarrow AC$
$SC \leftarrow 0$
BUN
BSA
ISZ
D_4T_4
$PC \leftarrow AR$
$SC \leftarrow 0$
D_5T_4
$M[AR] \leftarrow PC$
$AR \leftarrow AR + 1$
D_5T_5
$PC \leftarrow AR$
$SC \leftarrow 0$
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 $(PC \leftarrow PC + 1)$
$SC \leftarrow 0$

- **AND to AC** - This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address.
- The result of the operation is transferred to AC. The microoperations that execute this instruction are:

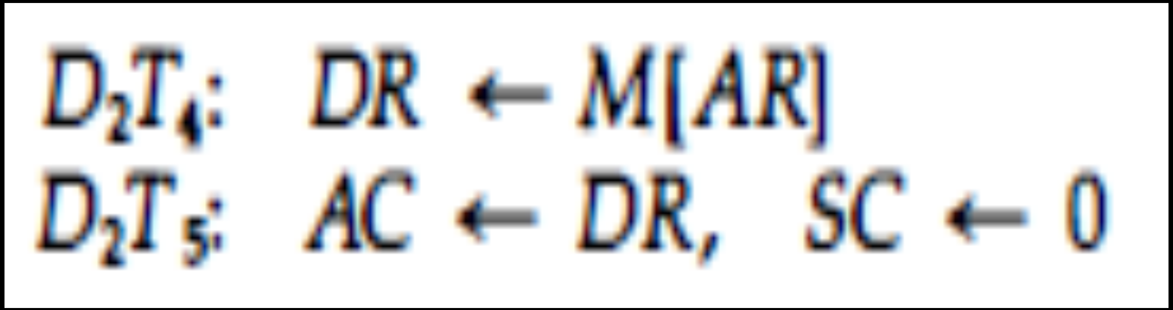
$$\begin{aligned} D_0T_4&: \quad DR \leftarrow M[AR] \\ D_0T_5&: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0 \end{aligned}$$

- **ADD to AC** - This instruction adds the content of the memory word specified by the effective address to the value of AC.

- The sum is transferred into AC and the output carry Cout is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are:

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

<u>**LDA: Load to AC**</u> - This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are



D_2T_4: $DR \leftarrow M[AR]$

D_2T_5: $AC \leftarrow DR, \quad SC \leftarrow 0$

STA: Store AC - This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

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

<u>BUN: Branch Unconditionally</u> - This instruction transfers the program to the instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

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

BSA: Branch and Save Return Address - This instruction is useful for branching to a portion of the program called a subroutine or procedure

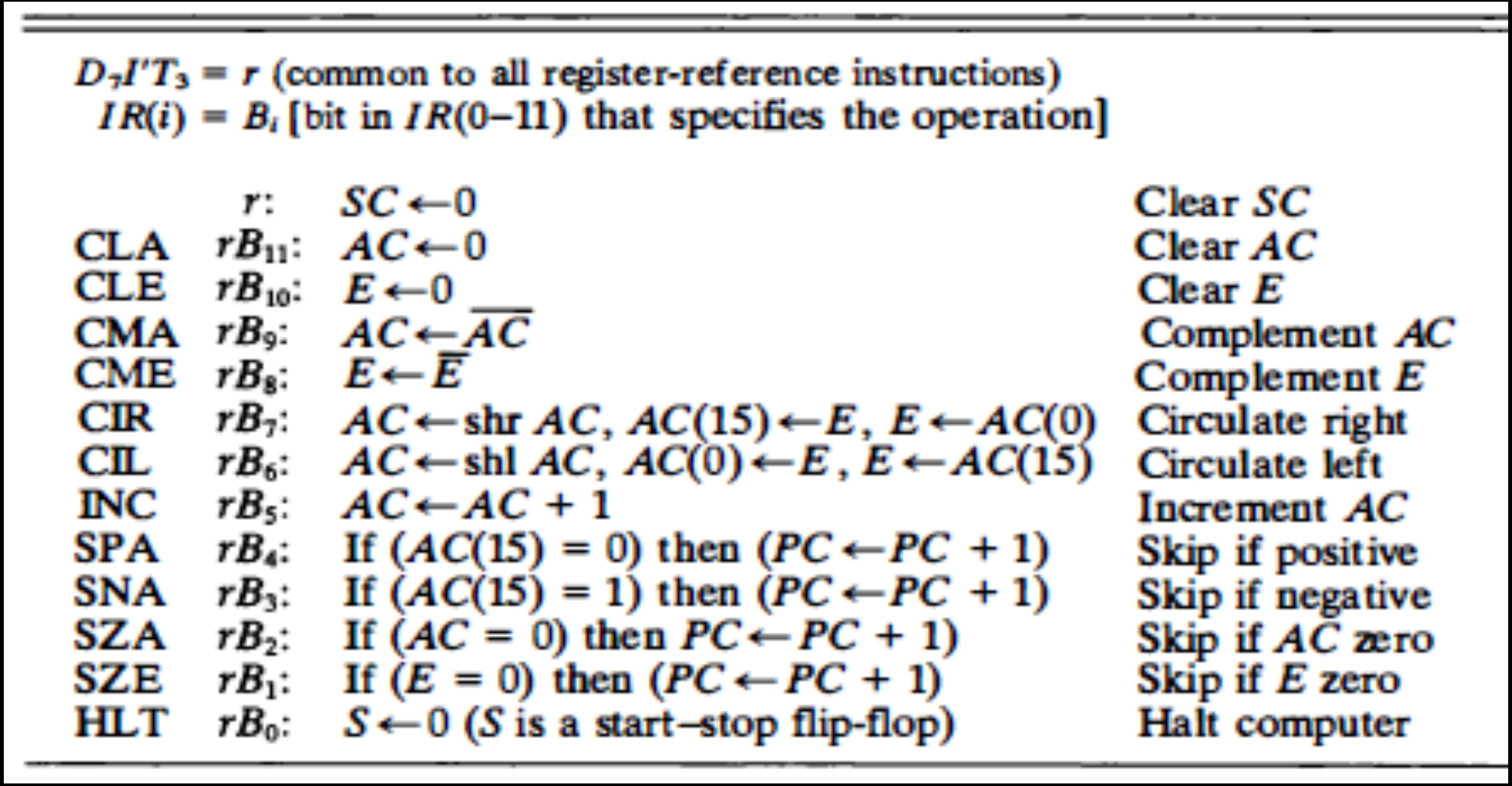
$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

<u>ISZ: Increment and Skip if Zero</u> - This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.

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

Register-Reference Instructions

- Register-reference instructions are recognized by the control when $D_7 = 1$ and $I = 0$.
- These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(0-11).
- These instructions are executed with the clock transition associated with timing variable T_3.
- Each control function needs the Boolean relation $D_7I'\ T_3$.

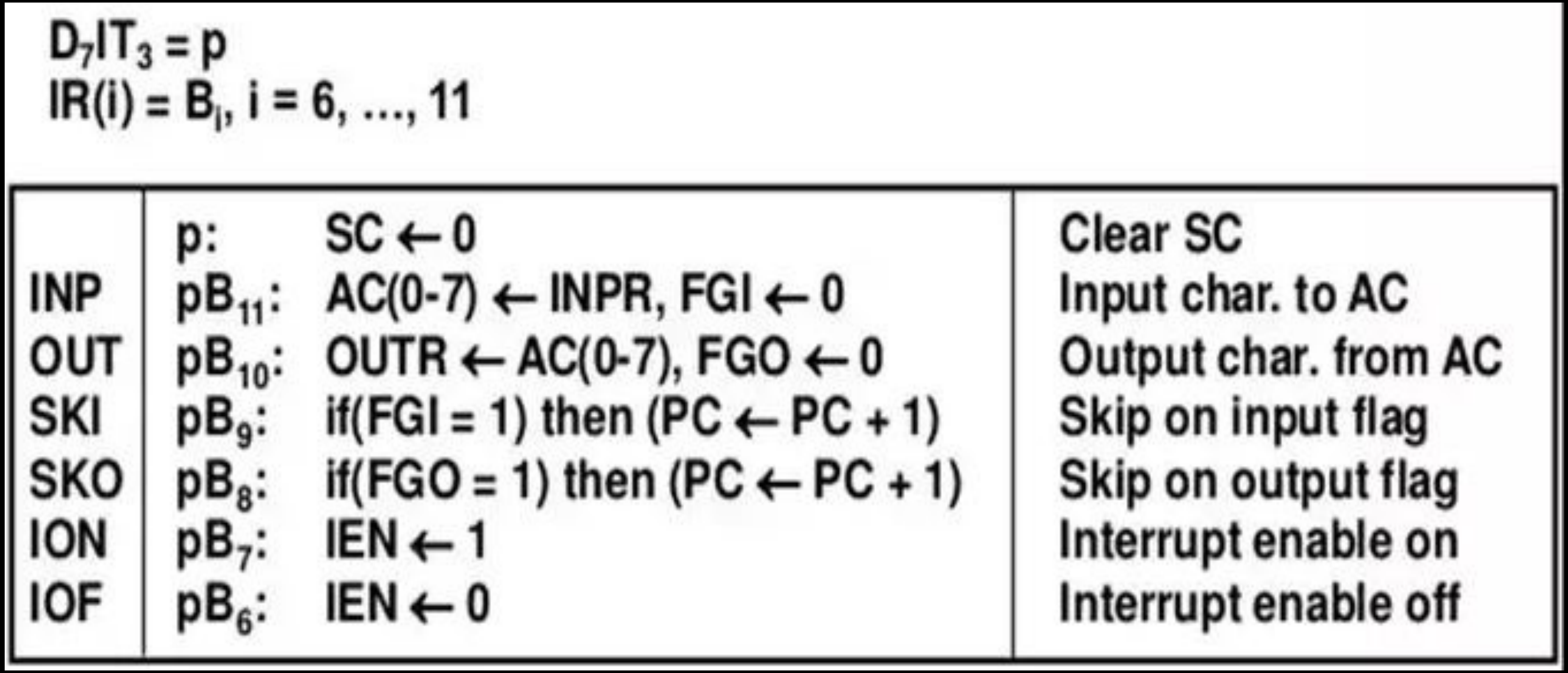


$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	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:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$	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

Input-Output Configuration

- The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code.
- The serial information from the keyboard is shifted into the input register INPR.
- The serial information for the printer is stored in the output register OUTR.
- These two registers communicate with a communication interface serially and with the AC in parallel.

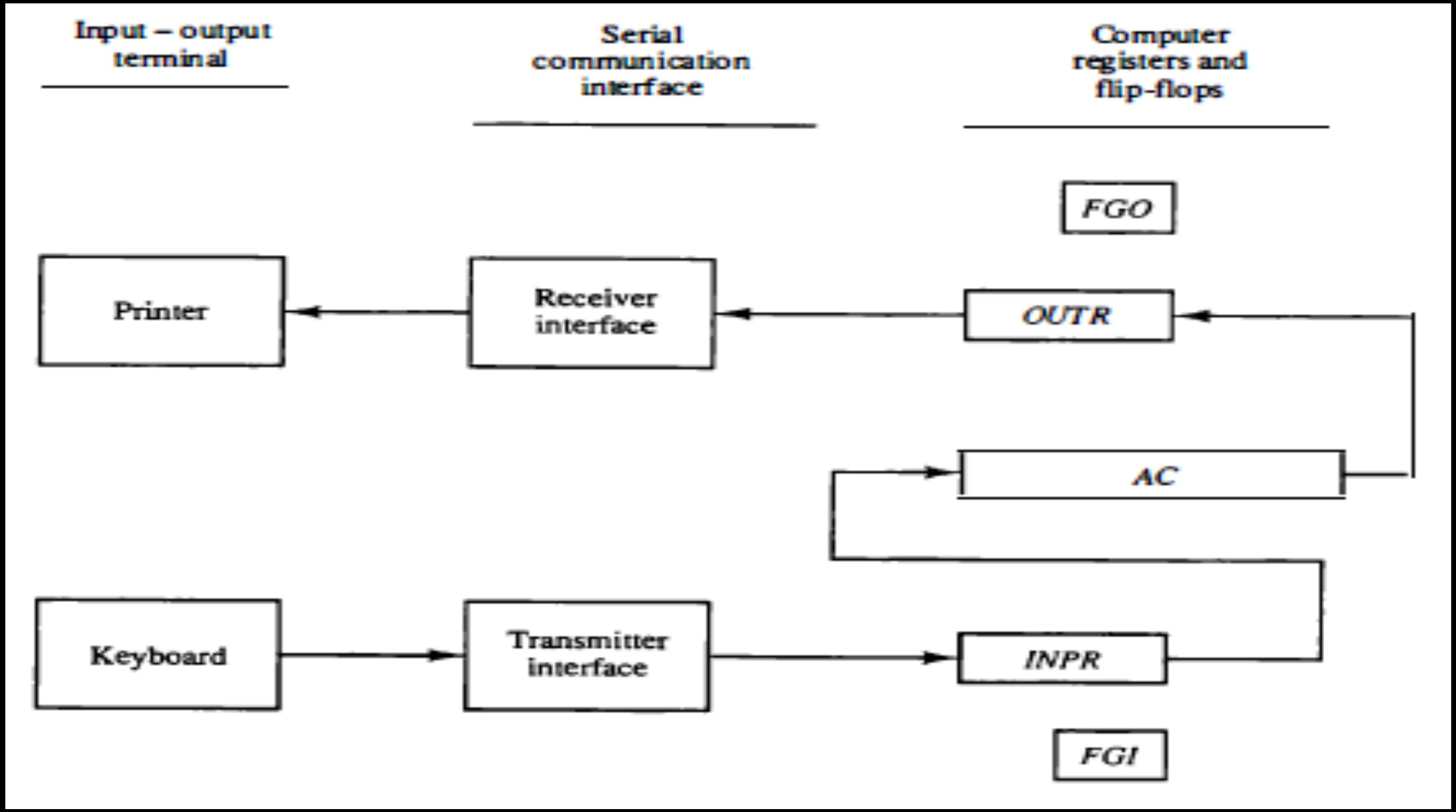


$D_7IT_3 = p$
$IR(i) = B_i,\ i = 6, \ldots, 11$

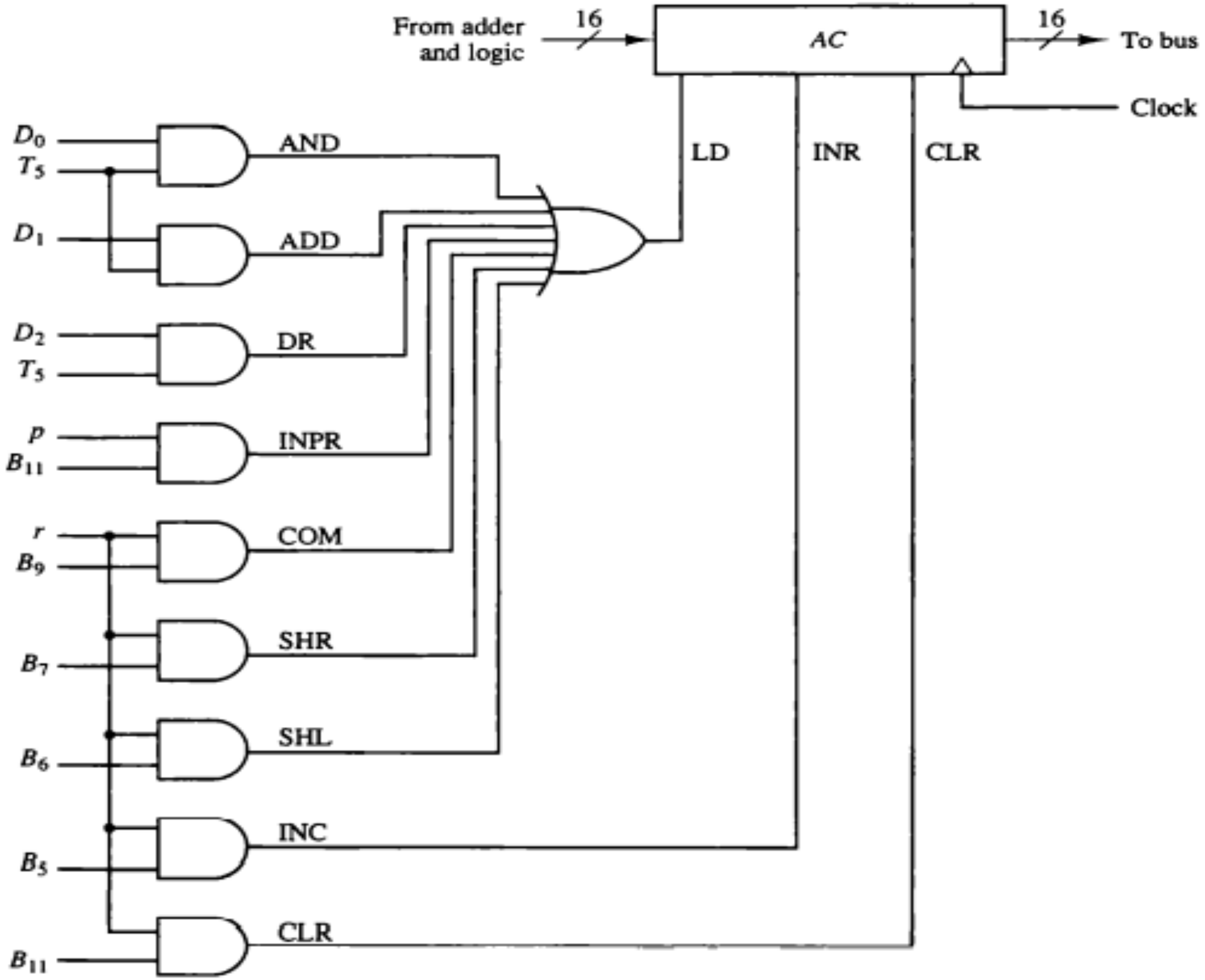
	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR,\ FGI \leftarrow 0$	Input char. to AC
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7),\ FGO \leftarrow 0$	Output char. from AC
SKI	pB_9:	$\text{if}(FGI = 1) \text{ then } (PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	$\text{if}(FGO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

Input-Output Configuration

- The transmitter interface receives serial information from the keyboard and transmits it to INPR.
- The receiver interface receives information from OUTR and sends it to the printer serially.
- The 1-bit input flag FGI is a control flip-flop.
- The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.
- The flag is needed to synchronize the timing rate difference between the input device and the computer.



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


From adder and logic
16
AC
16
To bus
Clock
LD
INR
CLR
D_0
T_5
AND
D_1
ADD
D_2
T_5
DR
p
B_{11}
INPR
r
B_9
COM
B_7
SHR
B_6
SHL
B_5
INC
B_{11}
CLR

Reduced Instruction Set Computer (RISC)

- Computers that use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often are classified as RISC.
- Relatively few instructions and few addressing modes
- Memory access limited to load and store instructions as all operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format, single-cycle instruction execution.
- Hardwired rather than microprogrammed control.
- The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, Thus, each operand is brought into a processor register with a load instruction. All computations are done among the data stored in processor registers. Results are transferred to memory by means of store instructions
- A RISC computer has a small set of simple and general instructions, rather than a large set of complex and specialized ones.

- Another success from this era was IBM's effort that eventually led to the IBM POWER instruction set architecture, PowerPC, and Power ISA. As these projects matured, a variety of similar designs flourished in the late 1980s and especially the early 1990s, representing a major force in the Unix workstation market as well as for embedded processors in laser printers, routers and similar products.
- The many varieties of RISC designs include ARC, Alpha, Am29000, ARM, Atmel AVR, Blackfin, i860, i960, M88000, MIPS, PA-RISC, Power ISA (including PowerPC), RISC-V, SuperH, and SPARC.
- The use of ARM architecture processors in smartphones and tablet computers such as the iPad and Android devices provided a wide user base for RISC-based systems. RISC processors are also used in supercomputers such as Summit, which, as of January 2020, is the world's fastest supercomputer as ranked by the TOP500 project.

Complex Instruction Set Computer (CISC)

- A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC.
- A large number of instructions-typically from 100 to 250 instructions
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes-typically from 5 to 20 different modes
- Variable-length instruction formats
- Instructions that manipulate operands in memory

- A **complex instruction set computer** (**CISC**) is a computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

- The term was retroactively coined in contrast to reduced instruction set computer (RISC) and has therefore become something of an umbrella term for everything that is not RISC, from large and complex mainframe computers to simplistic microcontrollers where memory load and store operations are not separated from arithmetic instructions.

- A modern RISC processor can therefore be much more complex than, say, a modern microcontroller using a CISC-labeled instruction set, especially in the complexity of its electronic circuits, but also in the number of instructions or the complexity of their encoding patterns.

- The only typical differentiating characteristic is that most RISC designs use uniform instruction length for almost all instructions, and employ strictly separate load/store-instructions.

- Examples of instruction set architectures that have been retroactively labeled CISC are System/360 through z/Architecture, the PDP-11 and VAX architectures, Data General Nova and many others.
- Well known microprocessors and microcontrollers that have also been labeled CISC in many academic publications include the Motorola 6800, 6809 and 68000-families; the Intel 8080, iAPX432 and x86-family; the Zilog Z80, Z8 and Z8000-families; the National Semiconductor 32016 and NS320xx-line; the MOS Technology 6502-family; the Intel 8051-family; and others.
- Some designs have been regarded as borderline cases by some writers. For instance, the Microchip Technology PIC has been labeled RISC in some circles and CISC in others. The 6502 and 6809 have both been described as "RISC-like", although they have complex addressing modes as well as arithmetic instructions that operate on memory, contrary to the RISC-principles.

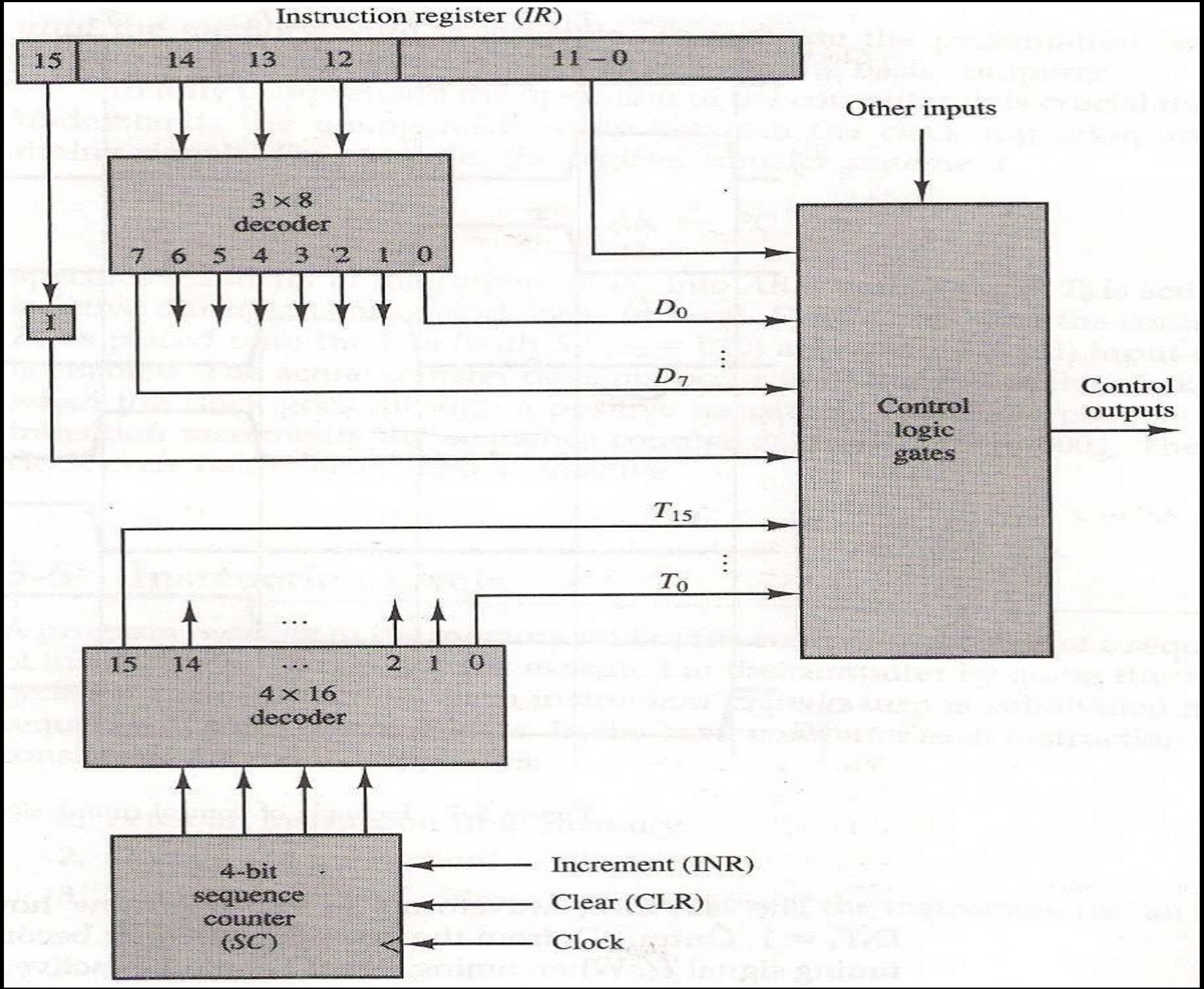
Complex instruction set format (CISC)	Reduced instruction set format (RISC)
Large number of instructions (around 1000) Application point of view they are useful but for system designer it is a headache	Relatively Few numbers of instruction, only those instructions which are strictly required, and in case of implementation of a complex function a set of simple instruction will perform together
Some instruction is there which perform specific task and are used infrequently for e.g. instructions for testing	Few instructions will be there which will be preforming more frequent work
Large number of addressing mode, leading to lengthen instruction, but compilation and translation will be faster	Number of instruction mode will be less, hardly 3 to 4
Variable length instruction format	Fixed length easy to decode instruction format
Instruction that manipulate operand in memory	Do not perform operation directly in memory, first load them in register and then perform, so all operations will be done with in the registers of CPU
Powerful but costly	Relatively less powerful but cheap
Microprogrammed control unit, relatively slow	Hardwired control unit, so fast

Designing of Control Unit

- The Control signal can be generated either by hardware (hardware Control unit design) or by (memory + h/w) (micro-program control unit design)
- In the hardware control unit design, each control signal is expressed as SOP expression and realized by digital hardware.
- The sequence of the operation carried out by this machine is determined by the wiring of the logic elements and hence named as “hardwired”.
- Fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.

Q Consider a hypothetical control unit implemented by hardware it uses three, 8-bits register A, B, C. It supports the two instruction I_1 and I_2, the following table gives control signals required for each micro-operation for both instructions?

Micro-operation	Control Signal	
	I_1	I_2
T_1	A_{in}, B_{out}	A_{in}, A_{out}
T_2	B_{in}, C_{out}	C_{in}, A_{out}
T_3	B_{in}, B_{out}	C_{in}, C_{out}
T_4	A_{in}, A_{out}	A_{in}, B_{out}


Instruction register (IR)
15
14
13
12
11 – 0
Other inputs
3 × 8 decoder
7 6 5 4 3 2 1 0
1
D_0
D_7
Control logic gates
Control outputs
T_{15}
T_0
15 14 ... 2 1 0
4 × 16 decoder
4-bit sequence counter (SC)
Increment (INR)
Clear (CLR)
Clock

Conclusion

- It is the fastest control unit design and it is suitable for real time application. (The RISC machine employ hardware control unit), Hardwired control is faster than micro-programmed control.
- A controller that uses this approach can operate at high speed. RISC architecture is based on hardwired control unit.
- Limitation: any modification to the design require re-design & re-connection of H/W component. it is not flexible; hence hardware control unit is not suitable for design & de-bugging environment. This problem is resolved using Micro-programmed control unit design.

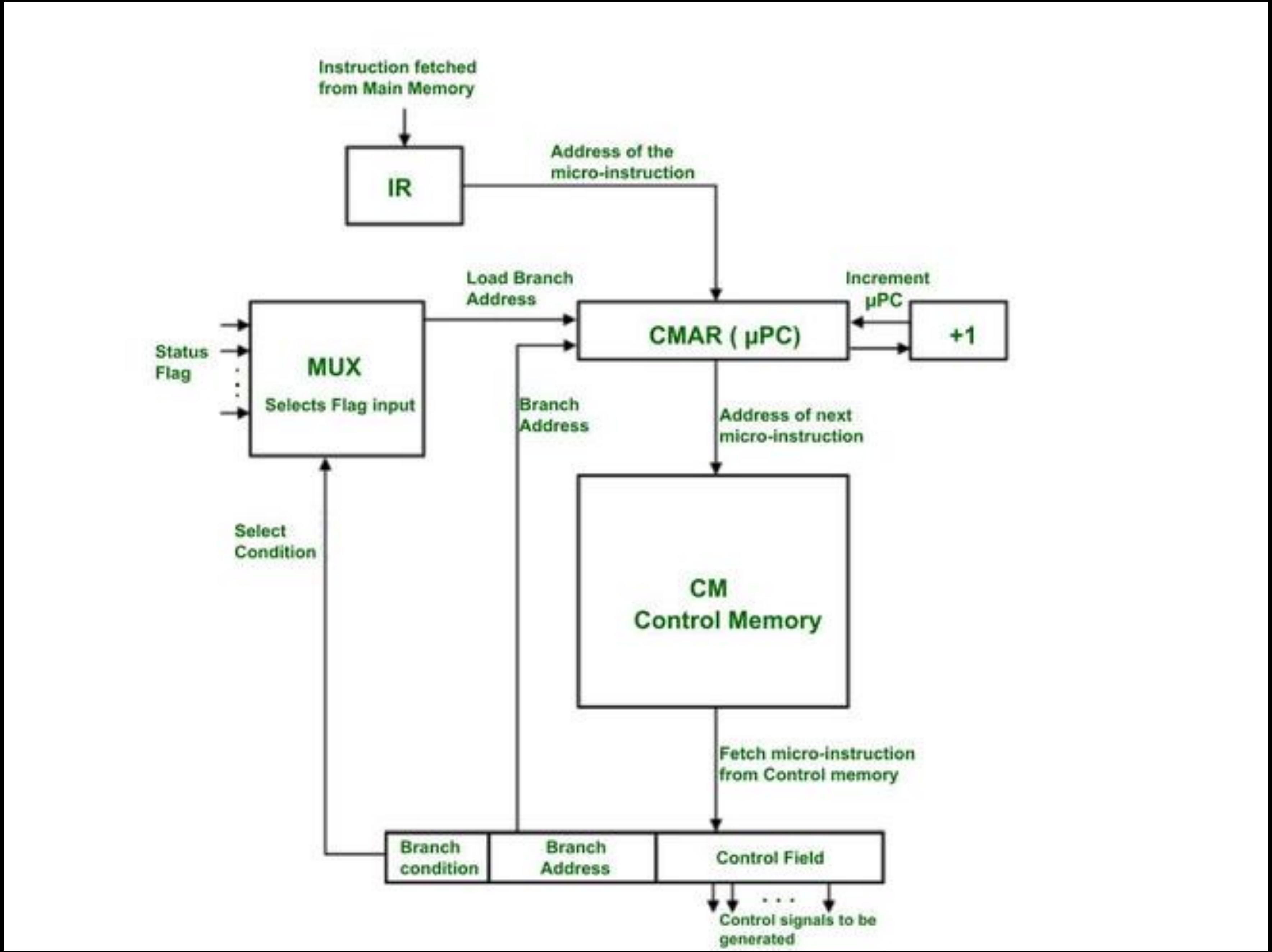
Micro-Programmed Control Unit

- The idea of microprogramming was introduced by Maurice Wilkes in 1951 as an intermediate level to execute computer program instructions.
- Microprograms were organized as a sequence of *microinstructions* and stored in special control memory.
- The main advantage of the microprogram control unit is the simplicity of its structure. Outputs of the controller are organized in microinstructions and they can be easily replaced.
- In this, the binary patterns of control signals are stored in control memory. After accessing word from the control memory, hardware is used to generate the control signals.

- Any changes can affects only the control word but not the hardware
- The design is flexible & it is used in CISC system.
- Based on the no of words in the control word, the micro-programming may be
    - Horizontal micro-Program
    - Vertical Micro-program

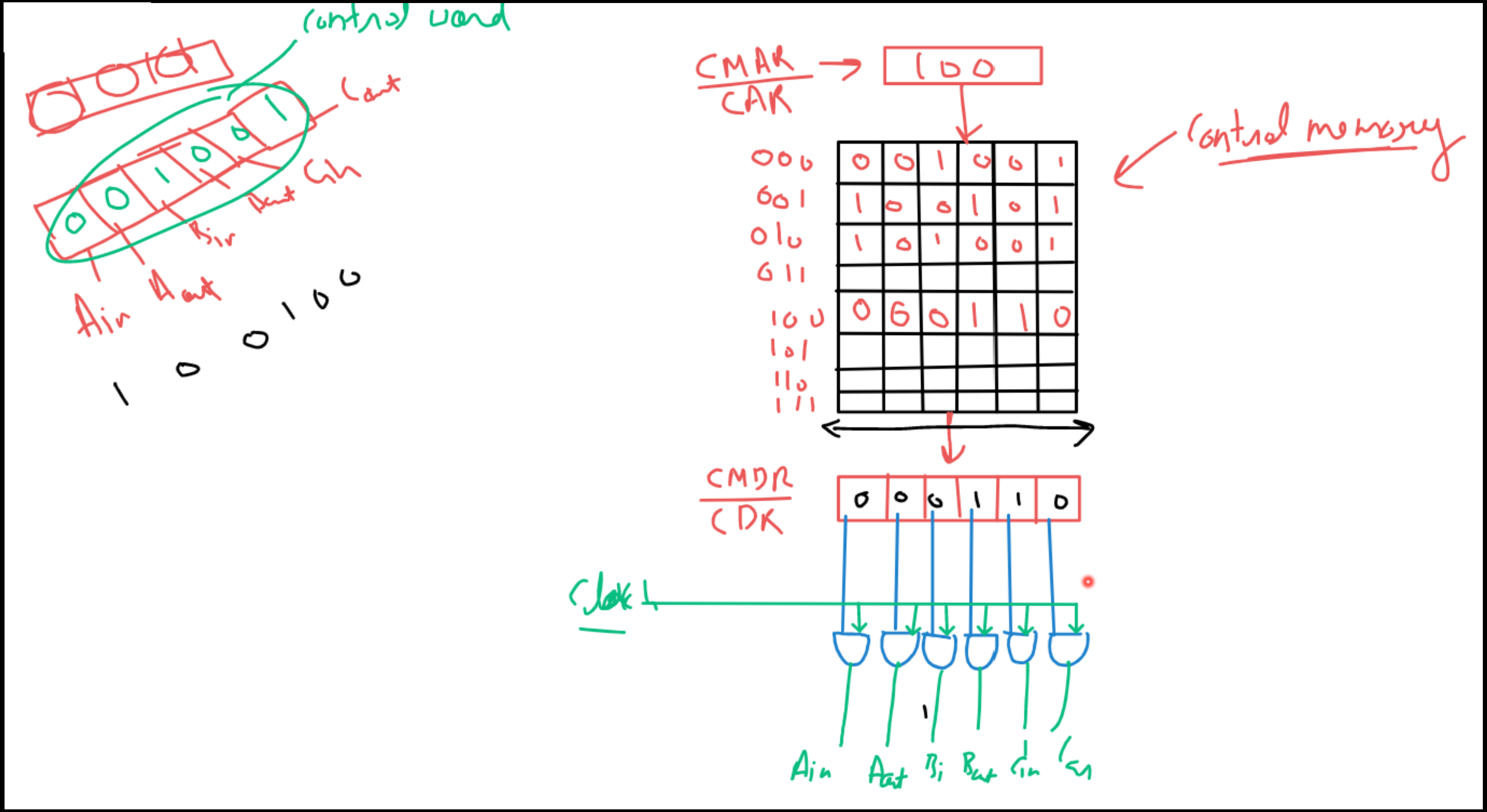
Characteristics	**Hardwired**	**Micro-programmed Control**
Speed	Fast	Slow
Implementation	Hardware	Software
Flexibility	Not Flexible	Flexible
Ability to handle complex instruction set	Diffcult	Easier
Design process	Diffcult for more opertion	Easy
Memory	Not Used	Control memory used
Chip are efficiency	Uses less area	Uses more area
Used in	RISC	CISC

Horizontal Micro-Programmed Control Unit


Instruction fetched from Main Memory
IR
Address of the micro-instruction
Load Branch Address
Increment μPC
CMAR (μPC)
+1
Status Flag
MUX
Selects Flag input
Branch Address
Address of next micro-instruction
Select Condition
CM
Control Memory
Fetch micro-instruction from Control memory
Branch condition
Branch Address
Control Field
Control signals to be generated

Horizontal micro-Program

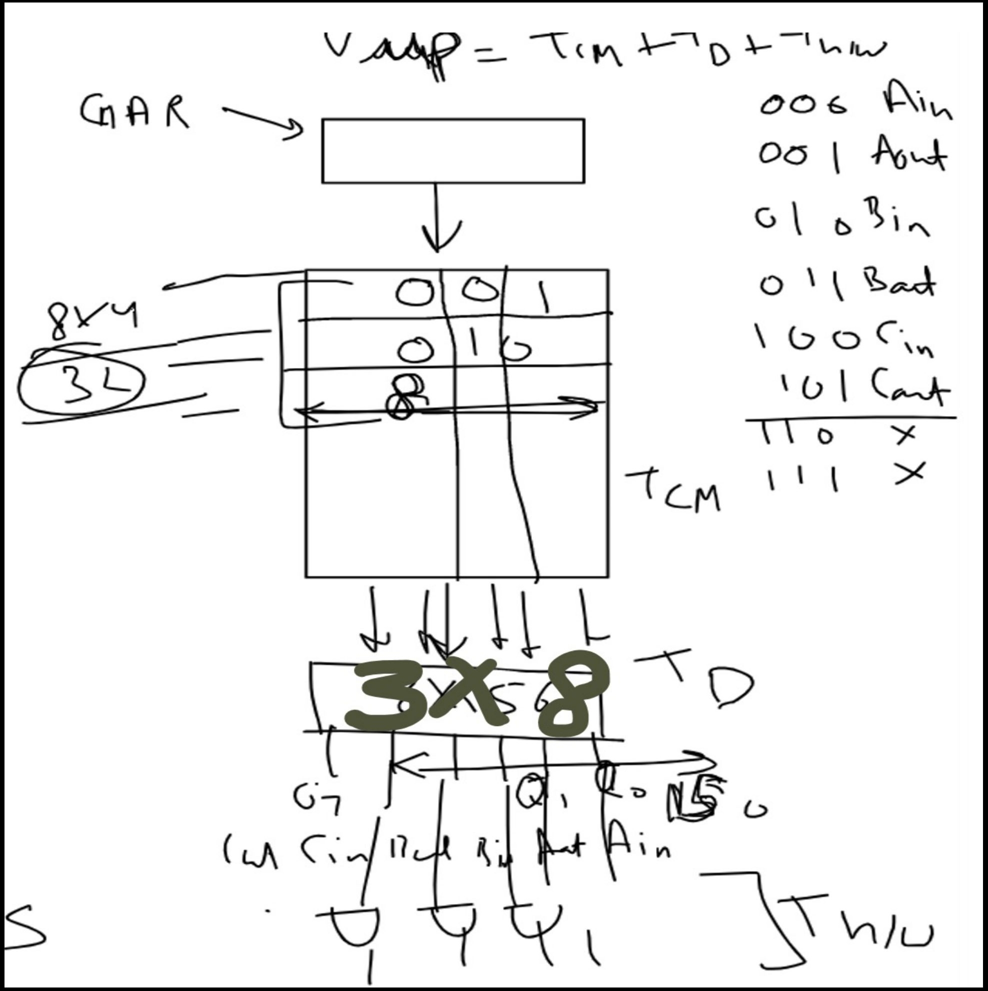
- The horizontal micro-programmed provides higher degree of parallelism & it is suitable in multi-processor system. it requires more bits for control word (1 it for every control signal)



000	0	0	1	0	0	1
001	1	0	0	1	0	1
010	1	0	1	0	0	1
011						
100	0	0	0	1	1	0
101						
110						
111						

Vertical Micro-program

- The vertical micro-programming reduces the size of control words by encoding, control signal pattern before it is stored in control memory. it offers more flexibility than horizontal micro-programming.
- The pattern with vertical-programming is the maximum degree of parallelism is 1(due to the decoder).



Horizontal	Vertical
Long Format	Short Format
Ability to express a high degree of parallelism	Limited ability to express parallel micro-operations
Little encoding of control information	Considerable encoding of the control information
Usefull when higher operating speed is desired	Slower operating speed

Conclusion

- The combinational Horizontal micro-Program and Vertical Micro-program controlled unit is preferred in most application.

Control word sequencing

- In order to implement, the micro-program, control words are to be sequentially from control memory. One address instruction is used for performing the control word sequencing.

Flag	Control Signal	Address