

Control Design

Digital system

- **Data Processing Unit** (DPU): capable of performing certain operations on data.
- **Control Unit** (CU): issues control signals or instructions to DPU
- CU signals **selects the functions** to be performed at specific times and route the data through the appropriate functional units.
- DPU is logically **reconfigured** by the CU to perform certain set of *(micro)operations*.
- **Sequence** of these micro-operations are important.

Control Design

Functions of CU:

- **Instruction sequencing:** methods by which instructions are selected for execution or control of the processor is transferred from one instruction to another.
- **Instruction Interpretation:** methods used for activating the control signals that cause the DPU to execute the instructions.

Instruction sequencing

- Each instruction can explicitly specify the address of its successor(s).
- It may increase the instruction length, which in turn increases the cost of memory where it is stored.
- I1 is stored in location A, and I2 is its successor.
- Let Program Counter (PC) contains A of I1, the address of I2 is can be determined by $PC \leftarrow PC+k$, where k is the word length of I1.

Instruction sequencing

- Increment of PC can be **automatic**.
- So, PC makes it unnecessary for an instruction to specify the address of its successor.
- **Branch Instruction**: Specifies implicitly/explicitly an instruction address X.
- **Unconditional branch** always alters the flow of control by causing $PC \leftarrow X$.

Instruction sequencing

- **Conditional branch** first tests the condition C (a result generated by earlier instruction, and stored in a status register).
- If C is **present**, then $PC \leftarrow X$, otherwise PC is incremented to point to the next consecutive instruction.

Program Control Transfer

- Often it is required to **temporal transfer** of control from P1 to P2 due to subroutine call or interrupt.
- **Subroutine call**: CALL X (X is address of the first instruction of P2)
- First contents of PC is stored (after fetching CALL) in a **predetermined location**, then X is loaded into PC.
- At then end of P2, transfer is **returned** back to P1.

Program Control Transfer

- In case of **interrupt**, call instruction is replaced by interrupt signal.
- Interrupt branch **address is fixed a priori**, or comes with the interrupt request.

Control Stack

- **Stack** is used to store **return** address.
- Also used to store **pass variables** (from P1 to P2), and variables that are local to P2.
- Each time a call is executed, return address is stored at **top of the stack**, PC is loaded with SUB address.

CALL SUB;

PUSH PC;

PC ← SUB;

Control Stack

- A return is effected by RETURN (equivalently to POP).
- LIFO organization of pushdown stack is used.
- No restriction on the use of recursive calls.
- Top of the stack varies dynamically; no interference with one another.

Control Stack

- Because return location is saved in top-of-stack, which varies dynamically, successive PUSH operations to save return addresses do not interfere with one another.

Begin

CALL SUB

X:

SUB:

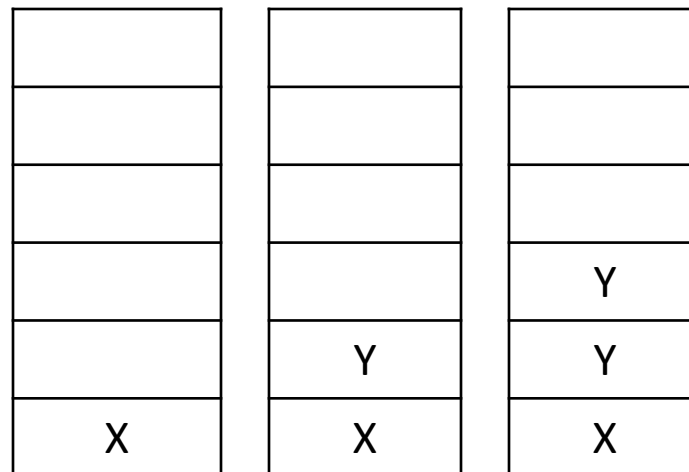
CALL SUB

Y:

RETURN

End

[For total k calls,
K-1 returns]



Use of pushdown stack to control recursive subroutine call

Stack Implementation

- A stack with n k -bit words is implemented using k n -bit shift registers having left and right-shift capabilities.
- One end of the shift register is defined as top-of-the-stack.
- Push/pop operation is done by activating the proper control line.

Stack Implementation

- Two possible error conditions may arise: *stack overflow* and *stack underflow*.
- **Error** is detected by a **counter** in the circuit to indicate the no. of words currently in the stack.
- **Counter** is incremented (decremented) by each push(pop) operation.
- **Combination** of push(pop) and count $n(0)$ results in *overflow(underflow)*.

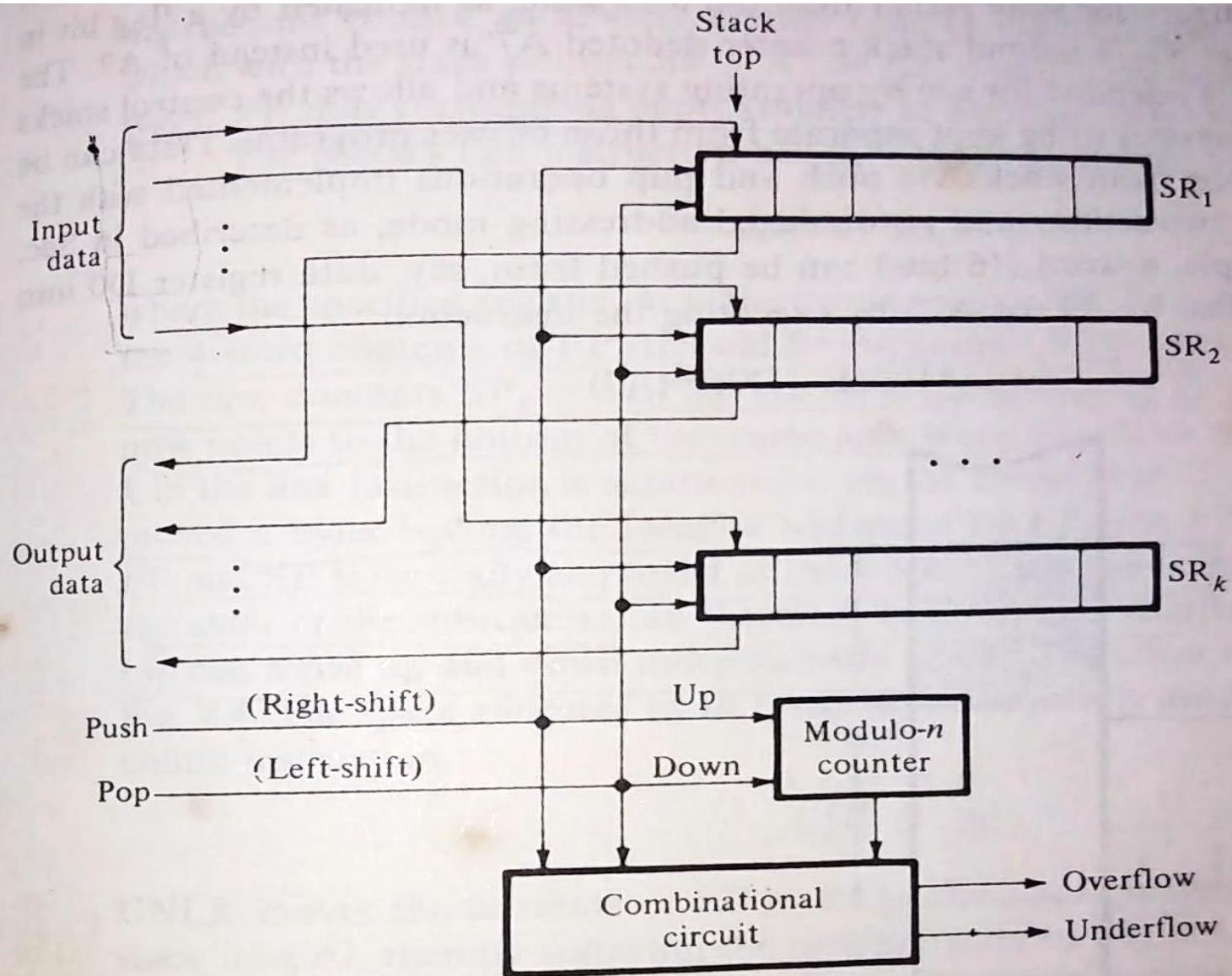


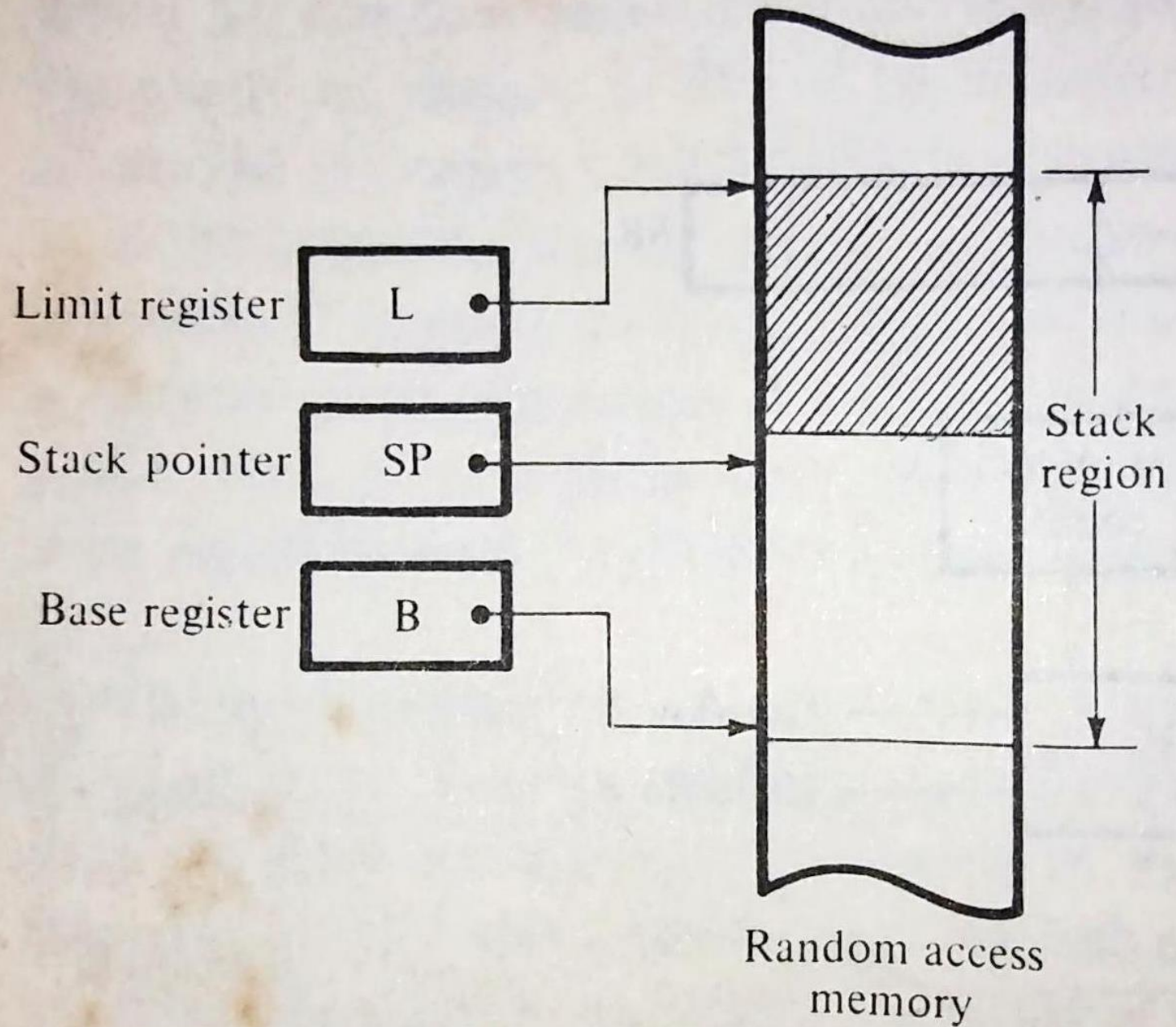
FIGURE 4.2
A stack constructed from shift registers.

Stack Implementation

- **Smaller stack: Shift register, Larger stack: RAM**
- Push: memory write, and pop: memory read
- **Stack pointer (SP):** this register contains address of the memory location **currently acting as top-of-the-stack**.
- A pop operation implies reading M using SP.
- SP either **decrements (pop)** or increments (push).

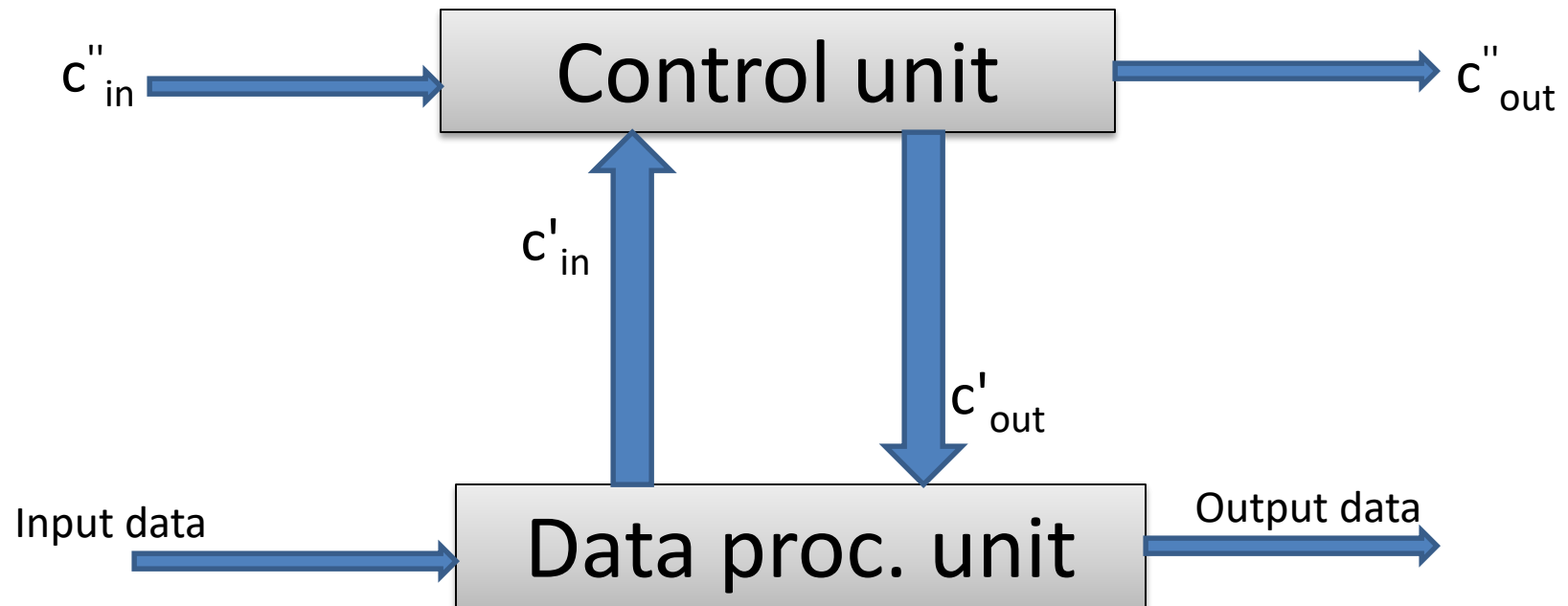
Stack Implementation

- **Limit register (L):** Contains **highest** stack address.
- **Base Register (B):** Contains **lowest** stack address.
- B, L and SP define the **boundary** of the stack.
- **SP > L** : Overflow and **SP < B** : Underflow.



■ Instruction Interpretation

Control signals -> via control lines -> outside



- Control and data are **relative**.
- **Same physical lines** can be used for either data or control at different levels.

Control Specifications :-

C'_{out}:- **Controls** the operation of DPU (main function of CU).

C'_{in}:- **Enables** the data being processed **to influence** the CU, and allows data-dependent decisions to be made.

e.g. error, overflow (unusual conditions)

C''_{out} :- Transmitted to other CUs, indicating the status such as 'busy' or 'operation completed'.

C''_{in} :- Received from the other CUs e.g. supervisory controller.
- Start/stop/timing information.

C''_{in} & C''_{out} : Synchronize with other CU

Behavior of CU

Flowchart/ Description language or both

- Once a DPU design has been completed, and the control points are identified, each micro operation $Z \leftarrow f(x)$ can be identified with a set of control lines $\{c_{ij}\}$ that must be activated in order to execute that micro operation .
- Formal specification of the input/output behavior

Implementation methods

1) **Hardwired CU (HCU):** Sequential logic circuit, which generates specific sequences of control signals.

- Minimizes the no. of components.
- Maximizes the speed of operation.

Once constructed, changes in behavior can be implemented by **re-designing** and physically **re-wiring** the unit.

=> **Lack of structure** makes HCU costly to design and debug.

2) Microprogramming :-

- Maurice V. Wilkes (1950)
- flexible & systematic

Activate control lines $\{c_{i,j}\}$

- A (micro)instruction stored in a special addressable memory called a **Control Memory (CM)**.

Microprogram:- Sequence of micro instructions (MI) needed to execute a particular operation.

- Fetch the MI from CM one at a time, and use them to activate the control lines directly.

- It is known as **Micro-programmed control unit**
- Firmware
- **Permanent** software programmed into a read-only memory.
- Design **changes** can easily be made by altering the **contents** of the CM
- **Emulation:-** A micro-programmed CPU can execute programs written in the machine language of **several different** computers.

Limitation:-

Costly than HCU due to CM & its circuitry.

Slower due to fetch from CM.

Since the improvements in memory technology, it is now a **standard method** of designing CU


Hardwired control :-

Trade-off: Amount of hardware, speed and cost.

Design methods in practice are ad hoc, heuristic, cannot easily be formalized (due to large no. of instructions, & complex interdependency).

3 main methods (apt for small CU like RISC):

- 1) State- table method
- 2) Delay-Element method
- 3) Sequence counter method



No one can
win over
other

State-table Method

- Like any finite-state sequential machine

Input Combinations c_{in}				
State	I_1	I_2		I_m
S_1	$S_{1,1}, Z_{1,1}$		$S_{1,m}, Z_{1,m}$
S_2	$S_{2,1}, Z_{2,1}$		$S_{2,m}, Z_{2,m}$
.	.			.
.	.			.
S_n	$S_{n,1}, Z_{n,1}$			$S_{n,m}, Z_{n,m}$

C_{in} : Input

C_{out} : output

Row: Set of internal states $\{S_i\}$

Internal state: Information stored at discrete point of time

Column: Set of external signals to CU

Entry in row S_i and column I_j : $S_{i,j}$, $Z_{i,j}$

$S_{i,j}$: Next state of CU

$Z_{i,j}$: Set of output signals $Z_{i,j}$ from C_{out} that are activated by the application of I_j to CU when it is in state S_i .

Pros: Systematic technique for minimizing no. of gates, flip-flops.

Cons:

- i) No. of states, i/p conditions may be so huge that **size** of the table & computation **time** become excessive.
- ii) Tends to conceal useful info about circuits behavior e.g. **loop**.
- iii) Due to random structure, debugging and maintenance become difficult.

DELAY-ELEMENT METHOD

- Generate the sequence of signals at times t_1, t_2, \dots, t_n using a Hardwired Control Unit

t_1 : Activate $\{ C_{1,j} \};$

t_2 : Activate $\{ C_{2,j} \};$

.....

.....

t_n : Activate $\{ C_{n,j} \};$

- Initial signal $START(t_1)$ is fanned out to $\{c_{1,j}\}$ to perform first micro operation.

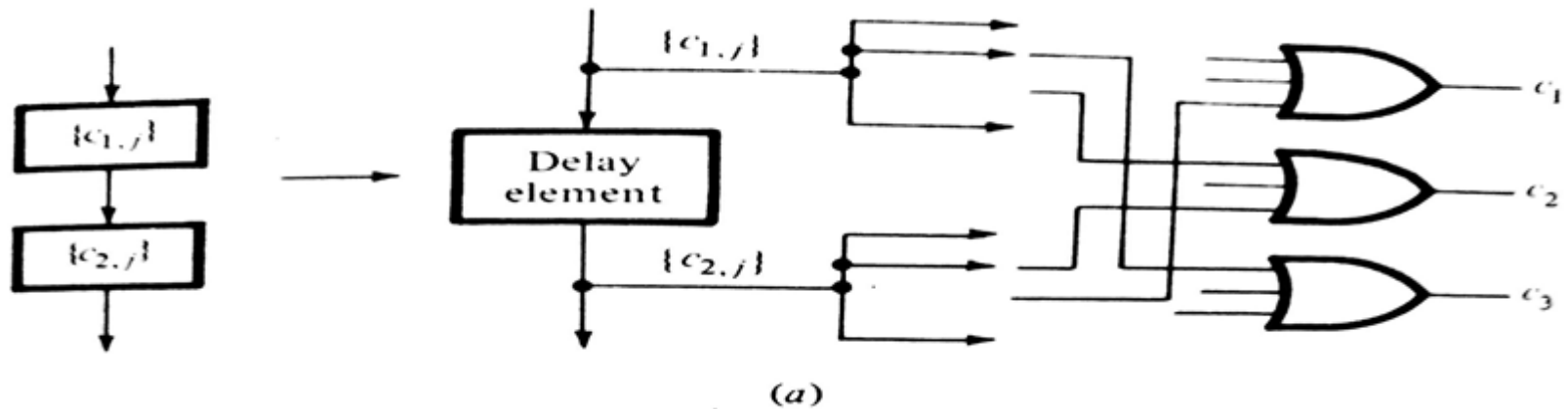
If $START(t_1)$ is also entered into a time delay element of delay $t_2 - t_1$, the output of that circuit, $START(t_2)$ can be used to activate $\{c_{2,j}\}$.

Likewise, another delay element of delay $t_3 - t_2$, with input $START(t_3)$ can be used to activate $\{c_{3,j}\}$.

- Sequence of delay-elements.

- D flip-flop is used to ensure synchronous operation.
- Can be constructed from flow-chart.
- Circuit mirrors the flow of control through the flow-chart.

DELAY-ELEMENT METHOD

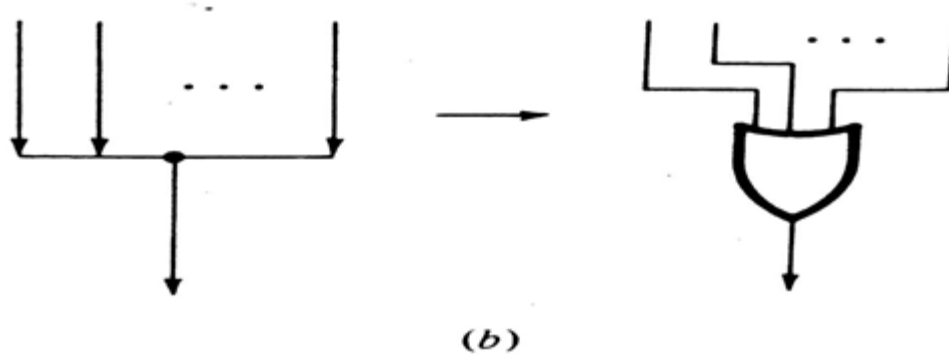


Each sequence of **two successive micro operations** require a delay element.

Signals that are intended to activate the **same control line** C_i are fed to an **OR gate** whose output is C_i .

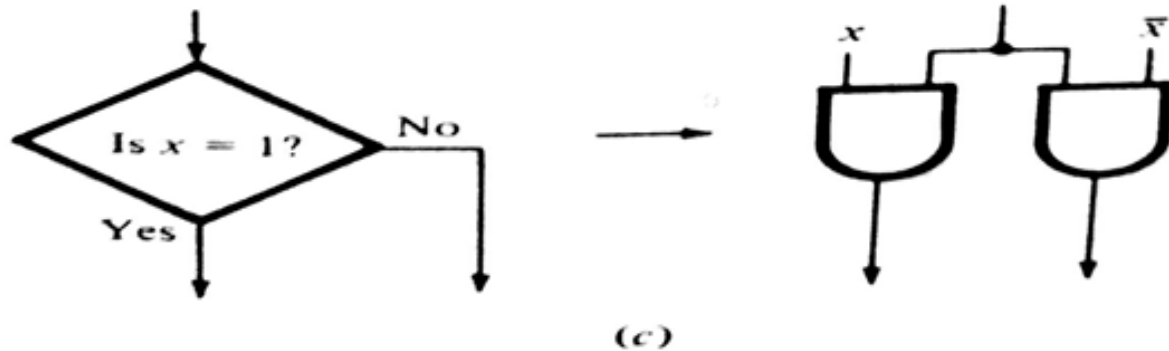
This line may then be **connected** to the control point it activates.

DELAY-ELEMENT METHOD



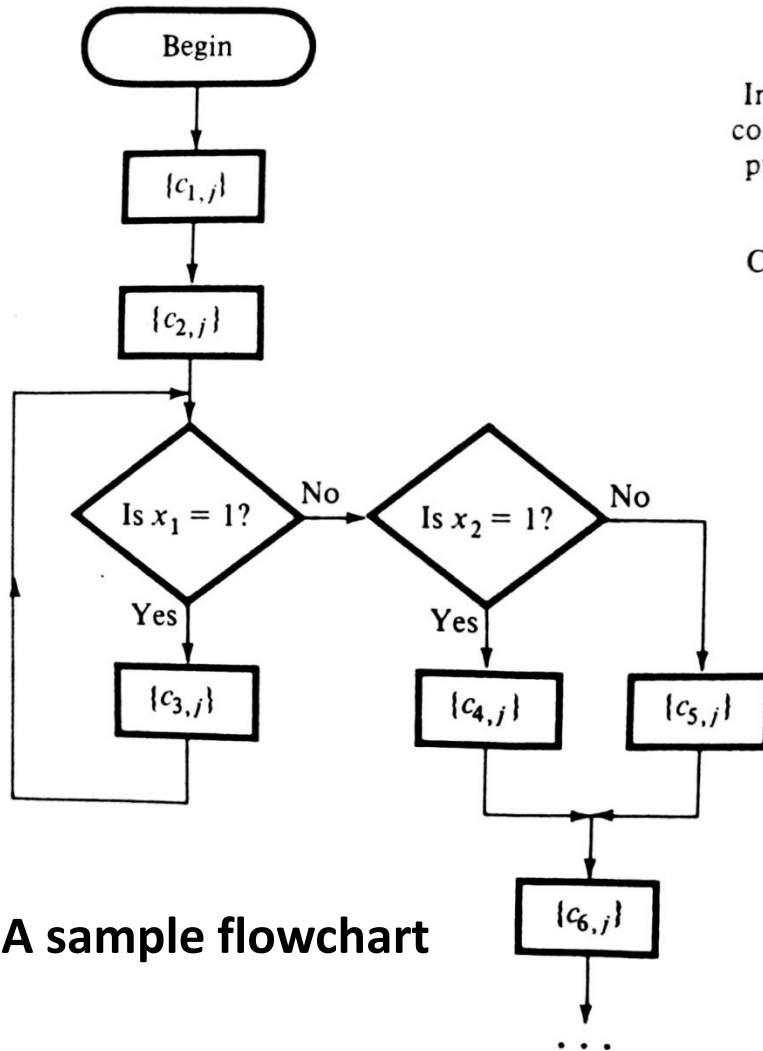
K-lines in the flowchart that **merge** to a common line are transformed into a **k-input OR gate**.

DELAY-ELEMENT METHOD

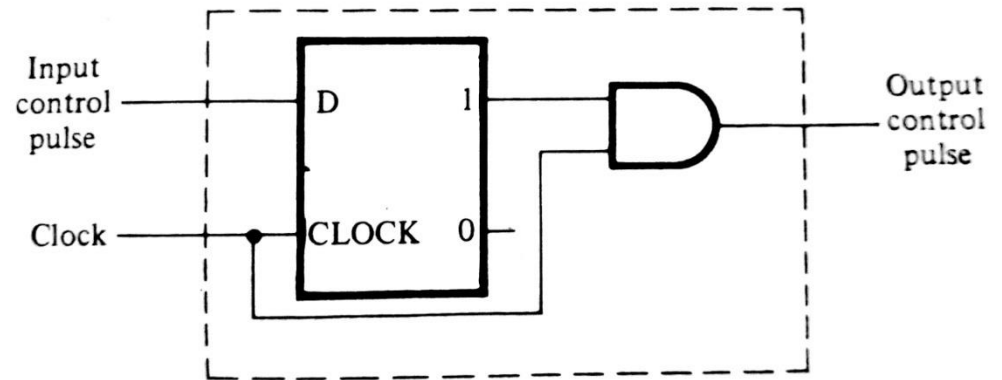


A **decision box** can be implemented by two AND gates. It forms a 1-bit de-multiplexer controlled by the test variable x (or a Boolean function $f(x)$).

An example :



A sample flowchart



If all delays are **synchronous** then a clocked D-type master-slave flip-flop can be used.

- **Complex circuit** may be required if an input becomes significant amount of **out of phase** with the clock due to propagation delays between delay elements.

Cons: 1) No. of delay elements needed is approximately equal to the no. of states.

2) Each delay element is a sequential circuit of equal or greater complexity than a flip flop.

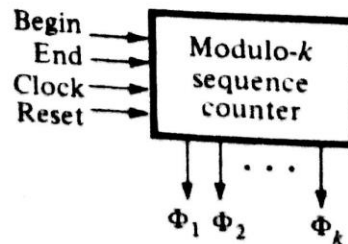
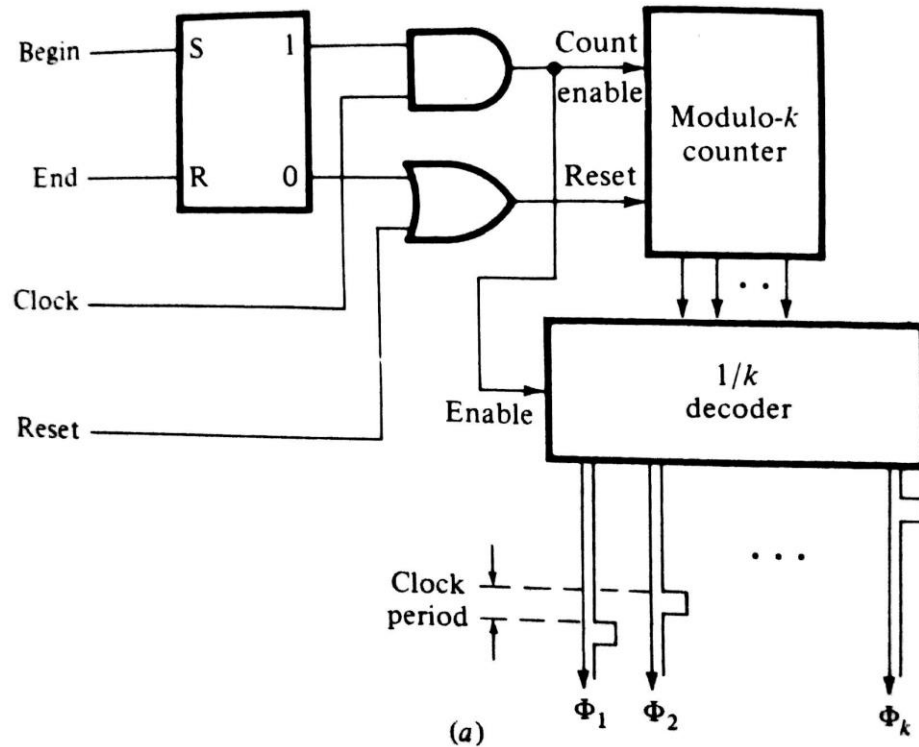
3) **Synchronization** becomes difficult.

Sequence Counter Method

Output of **Modulo-k sequence counter** is connected to $1/k$ clocked decoder.

- If the count enable input is connected to a clock source, **counter cycles continually** through its k states.
- Decoder generates **k -pulse signals $\{\Phi_i\}$** on its output lines, separated by **one clock** period.
 - called **phase signals**.
- Two additional input lines and a flip-flop are provided for turning the counter on and off.

Sequence Counter Method



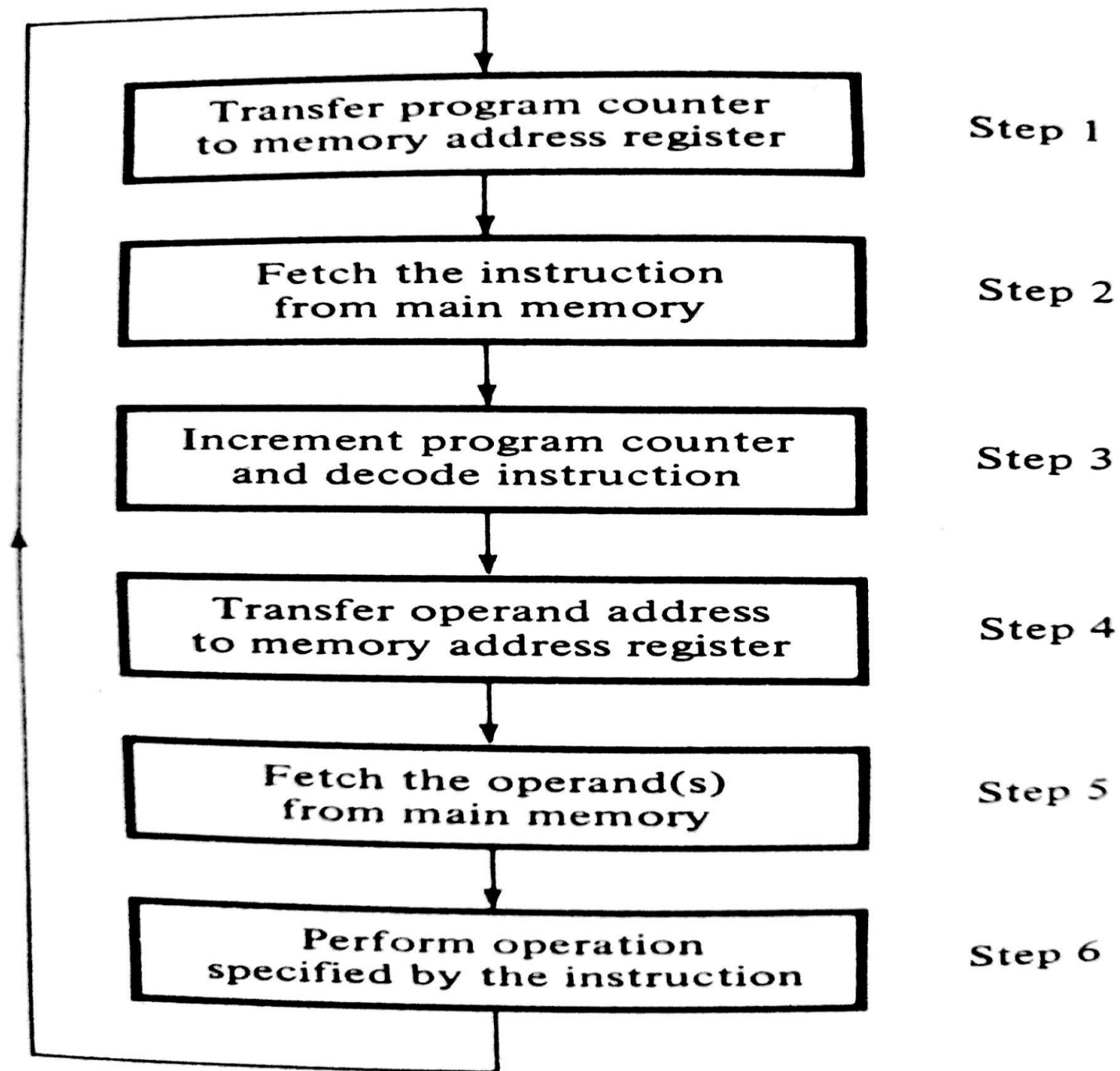
USEFULLNESS - Many digital circuits are designed to perform a relatively **small number of actions repeatedly**.

This type of behavior can be described by a flowchart consisting of a **single closed-loop containing k-steps**.

Example

CPU behavior represents as a:

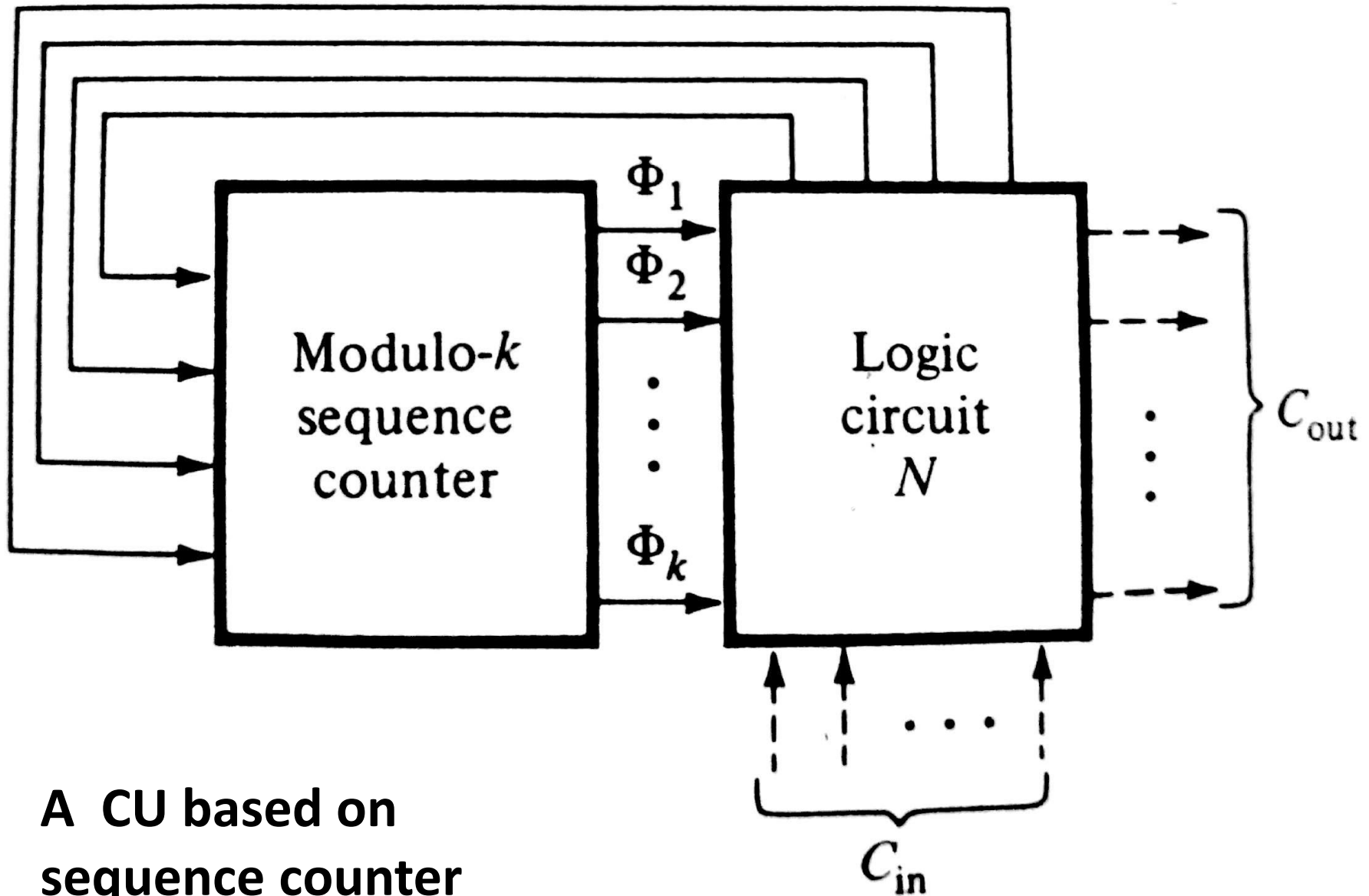
- Each pass through the loop: 1 instruction cycle
- Each step, 1 clock period (approximately)
- Modulo-6 sequence counter
- Each signal Φ_i activates some set of control lines in step i of every instruction cycle
- Single closed-loop



It is required to vary the operations performed in step i depending on certain control signals

$$C_{in} = \{ C'_{in}, C''_{in} \}$$

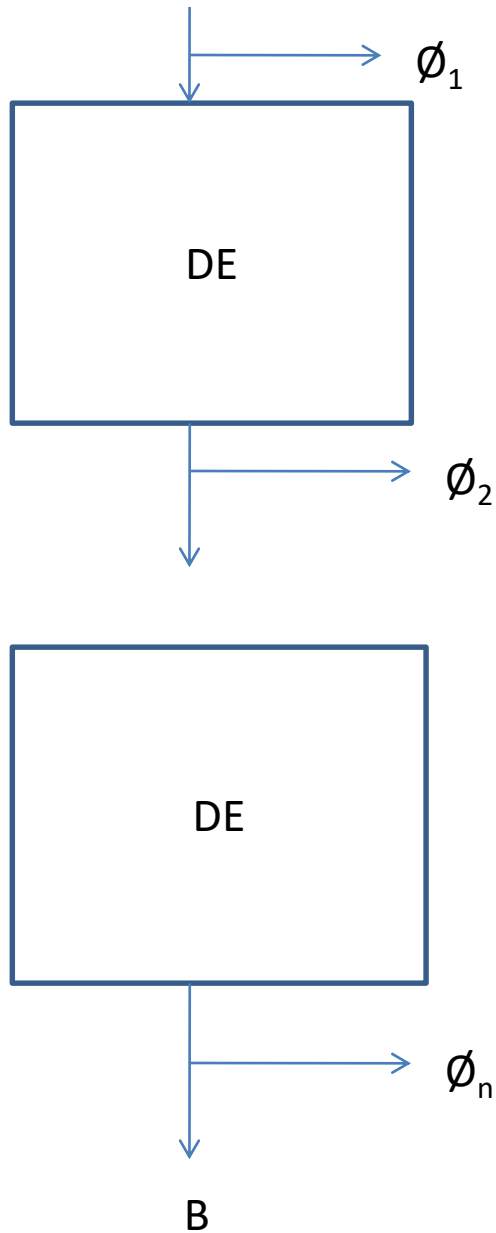
A logic circuit N combines C_{in} with the timing signals $\{\Phi_i\}$ generated by the sequence counter.



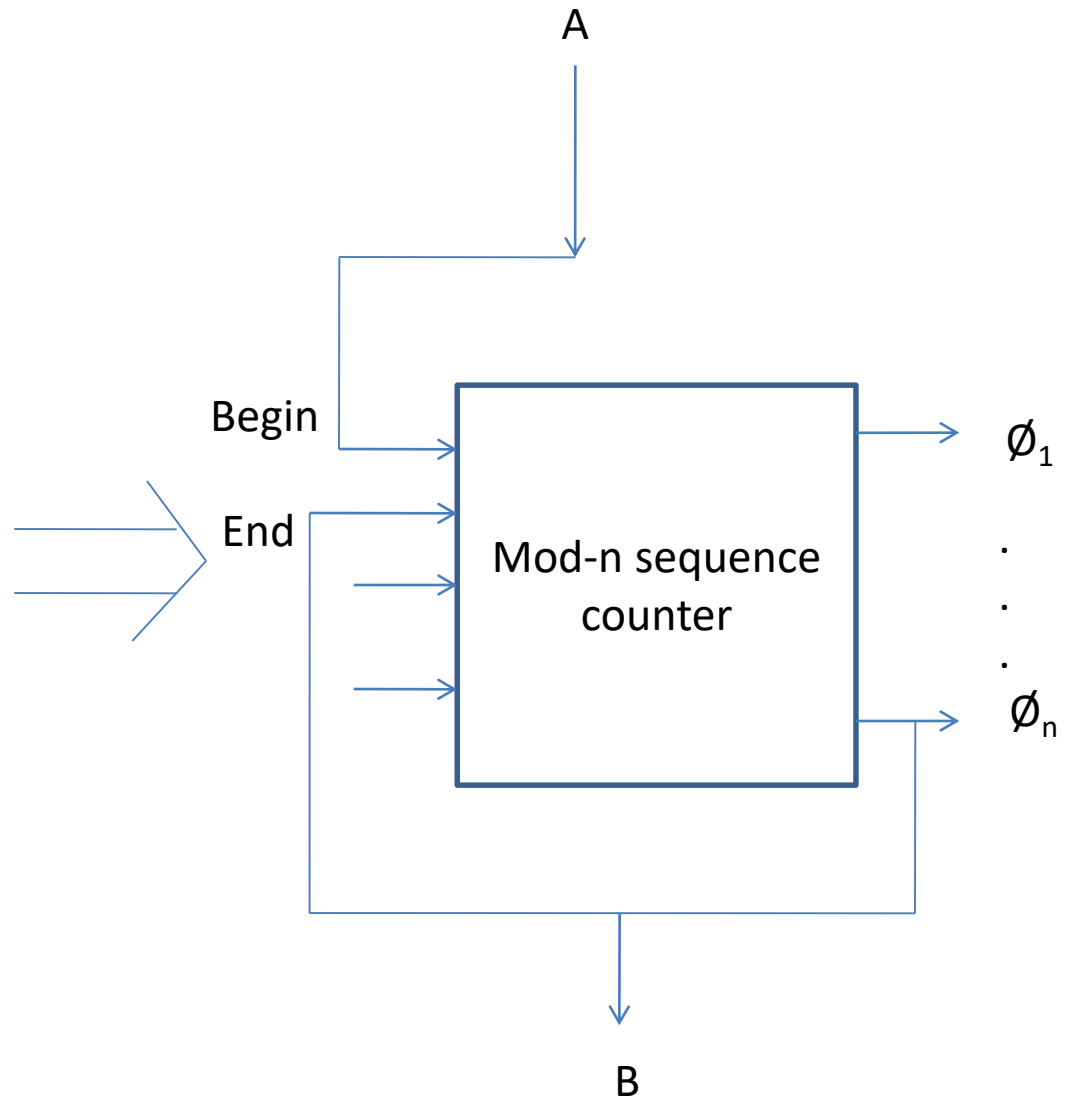
**A CU based on
sequence counter**

Relationship- Sequence counter and Delay-element .

- Modulo-k sequence counter is **equivalent** to cascade of k-s delay elements
- Connect k^{th} output line Φ_k to the **end line**, so that counter shuts itself off after one complete cycle.



Cascade of DEs

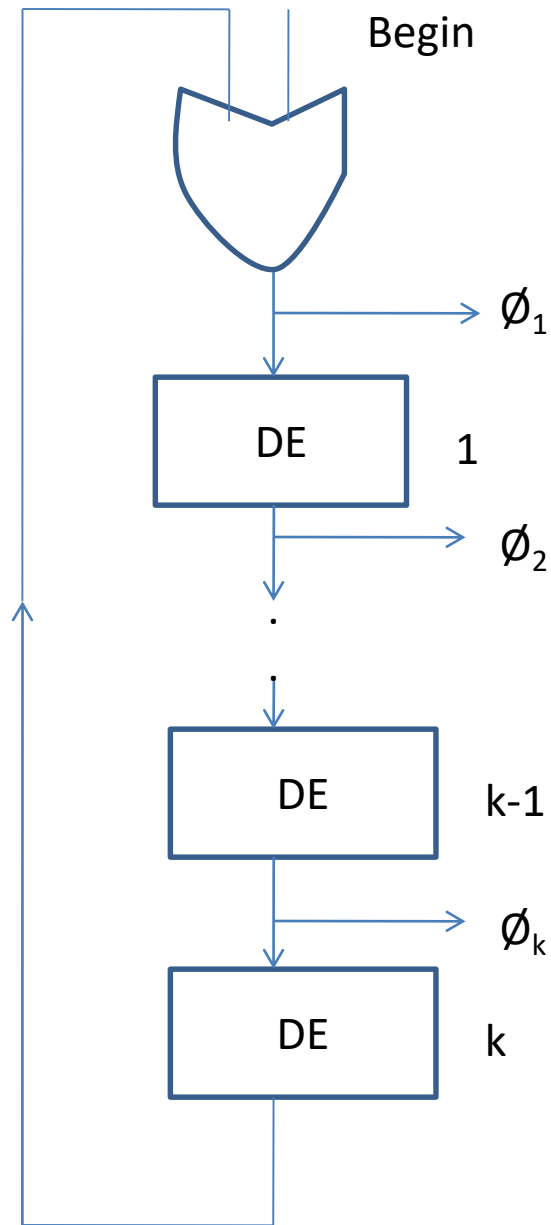


Equivalent sequence counter

➤ Also a **cascade of k-DEs** can be made to behave like a **sequence counter** by connecting its output to its input via an additional DE and an OR gate

➤ Modulo-k ring counter

➤ A cascade of identical DEs is essentially a shift register.



A DE circuit that behaves like a sequence counter.

MICROPROGRAMMED CONTROL

- Every instruction in a CPU is implemented by a **sequence** of one/more set of concurrent **micro operations**.
- Each micro operation is associated with a specific **set of control lines** which, when **activated**, **cause** that micro operation to take place.

- Since the no. of instructions and control lines are huge, HCU could be exceedingly complicated.
- Information stored in ROM/RAM, called **Control Memory (CM)**.
- Control signals to be **activated** at any time are **specified** by a microinstruction, which is **fetched** from CM.

➤ Each microinstruction explicitly or implicitly **specifies** the **next** microinstruction to be used.

➤ Sequencing

➤ A set of related microinstructions: **micro program**.

➤ Relatively **easy to change** the design.

➤ A micro-programmed computer C1 used to execute programs written in m/c language L2 (other m/c C1) by placing an **emulator** for L2 in the CM of C1.

➤ C1 is then said to be capable of emulating C2

➤ Comparable with assembly language, but requires more detail **knowledge** of the **processor h/w**.

➤ Symbolic language->micro-assembly language

➤ Micro-assembler

Wilkes' Design

Microinstruction:

1. **Control Field**: Indicates the control lines to be activated.
2. **Address Field**: Indicates the address in the CM of the next microinstructions to be executed.

K_i, C_i : if $K_i=1, C_i$ is activated

CM: Row-> microinstruction

Col -> Control/Address lines

CMAR -> control memory address register which stores the address of the current instruction.

➤ May be loaded from an external source as well as from an address fields

➤ External source provides the starting address microprogram stored in the CM

- CM is organized as a program logic array of diodes.
- This is partial matrix & consists of 2 components - control signals & address of next microinstruction.
- CMAR can be loaded by the IR or by the address field of CM. On getting an input from the IR, CM provides a 3 bit address to the 3 x 8 decoder.
- This is an entry point address to the CM.

➤ On the basis of this address, **decoder activates one** of the eight output lines (horizontal).

➤ This activated line, in turn, generates **control signals** and the **address** of the next microinstruction to be executed.

➤ This address is once again **fed to the CMAR** resulting in activation of another control line and address field.

➤ This cycle is **repeated till the execution** of the instruction is achieved.

- Ability to respond to external signal
- S flip-flop is used to indicate an external condition
- Conditional jump: One of the two possible address fields to be selected.

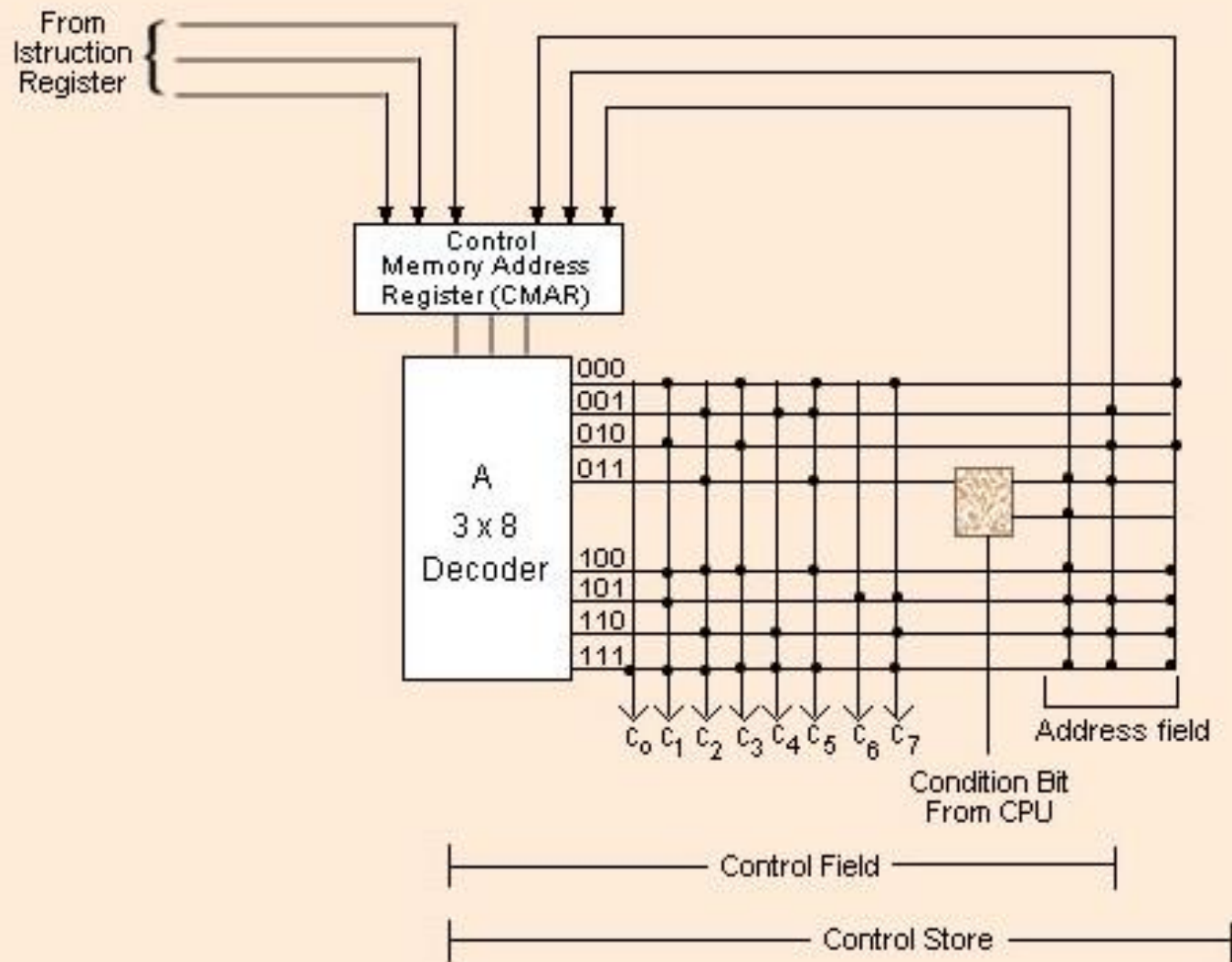


Figure 1: Wilkes Control Unit

Area of concern: word length of microinstruction
(influence the cost and size of CM)

1. Max no. of simultaneous micro-operations that must be specified (i.e., degree of parallelism)
2. The way control information is represented
3. The way the next microinstruction address is specified

Control Memory (early design): ROM or ROS (Read Only Store)

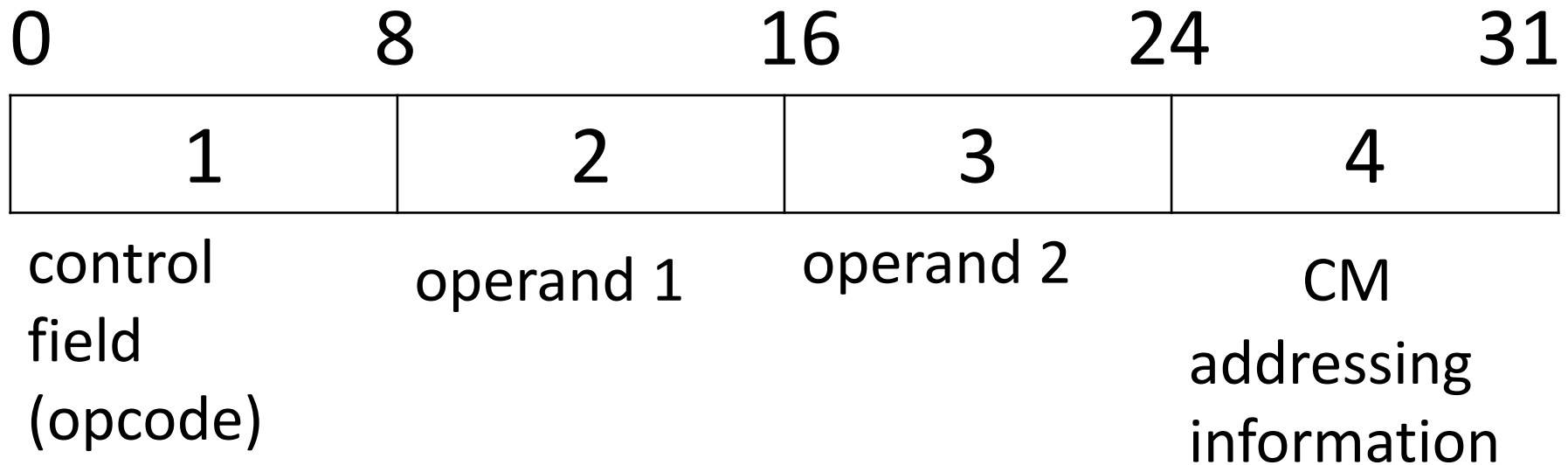
- Because ROM (i.e. diode) provides faster access than RAM (i.e. ferrite core)
- Microprogrammed processor was viewed as permanent except for correction or minor change (enhancement)

Wilkes Idea: Writable CM (WCM)

- allows instruction set to be changed by changing the microprograms that interpret the instruction opcode
- It can provide same machine with different instruction sets which may be tailored for specific applications.
- Computer with WCM implies no instruction set in usual sense (dynamically microprogrammable)

Parallelism in Microinstructions:-

Micro-programmable processor characterized by the **max. no. of micro-operations** that can be specified by a single microinstruction (several hundred)



Microinstruction format (IBM sys./370 model 145):

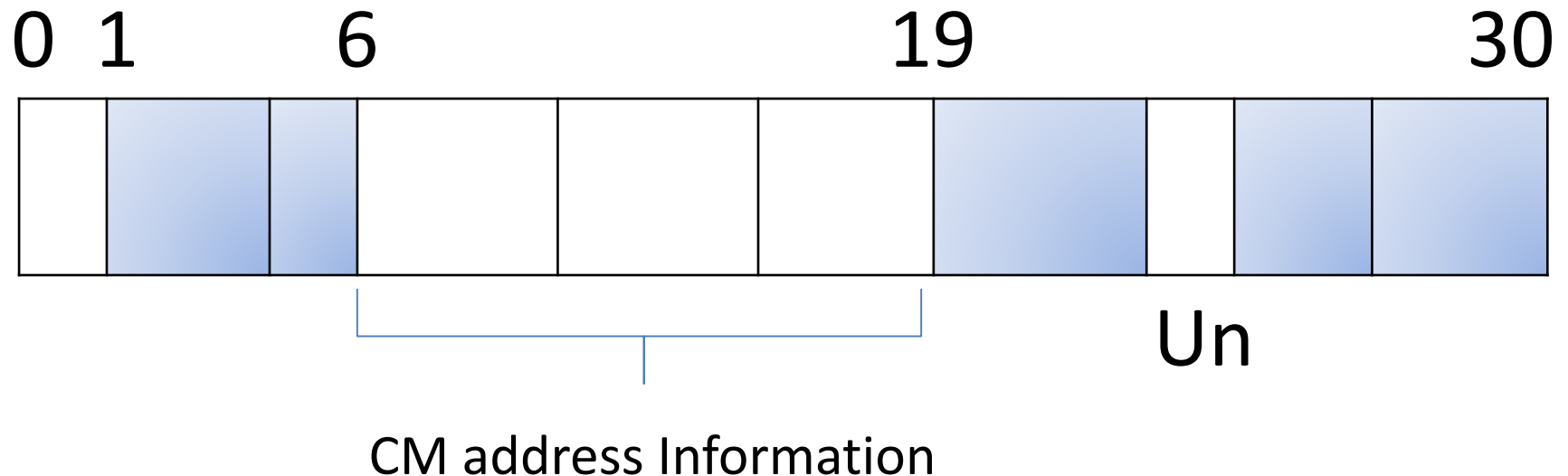
1: Operations to be performed

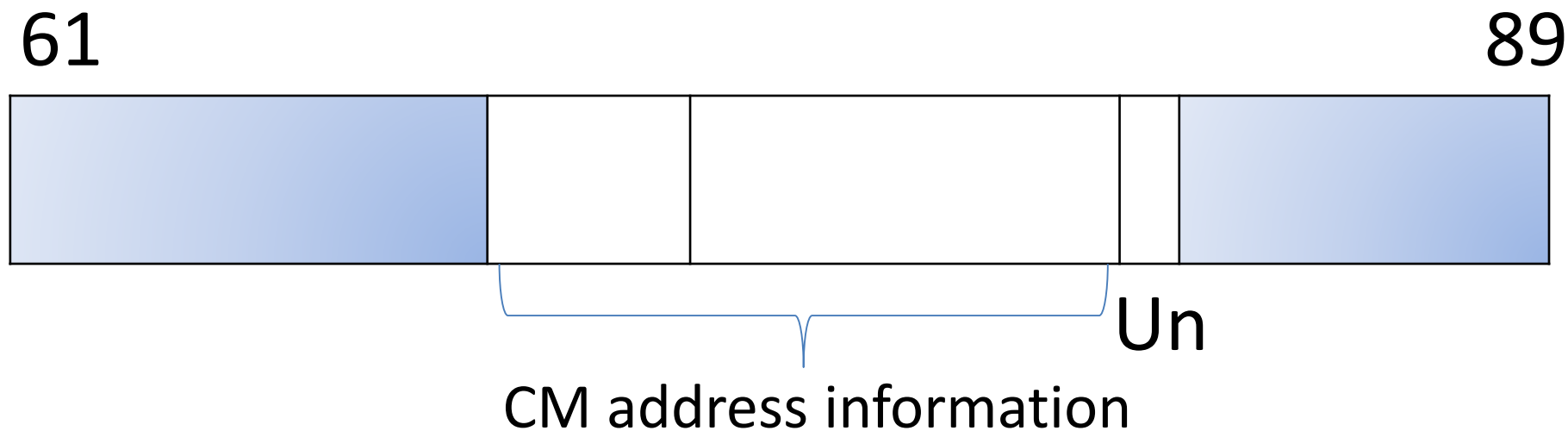
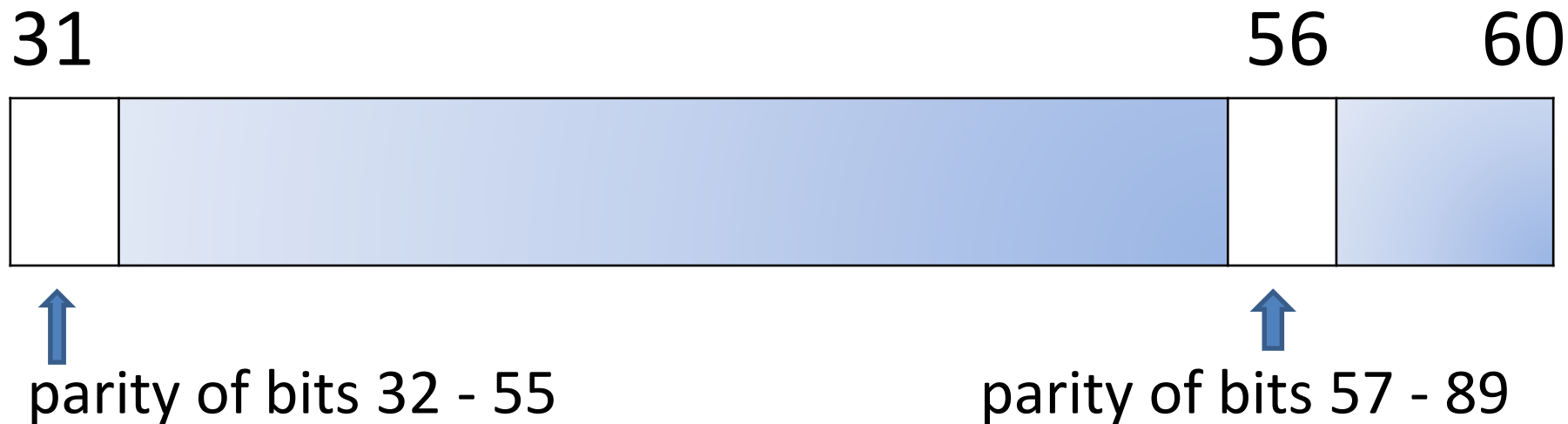
2-3: Mostly addresses of CPU registers

- If **all combination** of parallel micro-operations were specified by **single** opcode, the no. of opcode would be **enormous**.
- **Decoder** circuit was needed then (added complexity)

Solⁿ :- **Divide** the microinstruction specification port into k disjoint control fields.

Each control field is associated with a set of micro-operations any of which can be performed simultaneously with the micro-operations specified by the remaining control fields.





- 90-bit microinstruction format of IBM sys/360 Model 50 (shaded areas are control fields)
- 3-bit control fields 65-67 which controls the *right input to the CPU adder*. This field indicates which of the several possible registers should be connected to the right input of the CPU adder.
- 68-72 define different operations of adder such as *binary or decimal*.
- 21 other control fields : used for different purposes

- The scheme in which there is a **control field for every control line** is **wasteful** of *CM space*, since many combinations of control signals are never used.

- In general, any *n-independent* control signals can be **encoded** in a control field of $\lceil \log_2(n+1) \rceil$ bits, assuming that it is necessary to specify a **no-operation** condition when no control signals are to be activated.

Un-encoded format: Control signals may be derived **directly** from the microinstruction.

Suppose MI is loaded into CM data register, called *microinstruction register* **μIR**. The outputs of the control part of μIR are the control lines.

When encoded fields are used, each control field must be connected to a **decoder** from which control signals are derived.

Horizontal Microinstructions:-

- 1) Long formats
- 2) Ability to express a high degree of parallelism
- 3) Little encoding of the control information

Vertical Microinstructions:-

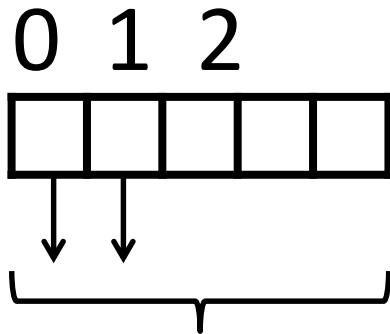
- 1) Short formats
- 2) Limited ability to express parallel micro operations
- 3) Considerable encoding of the control information

A horizontal microinstruction format allows *no encoding* of control information, whereas a vertical form *does*.

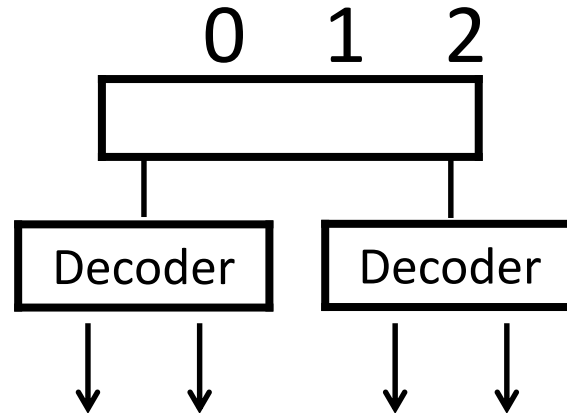
A vertical microinstruction can specify only *one* micro operation (*no parallelism*), while a horizontal one can specify *many* micro operations.

Not entirely independent

control fields

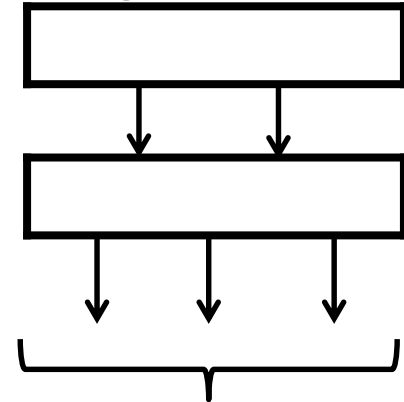


No encoding



Some encoding

Single control field



Control Lines

Complete Encoding

Microinstruction Addressing:

- Each MI contains the CM address of the next MI to be executed.
- For branching, two possible next address are included.

Advantage: No time lost in address generation

Limitation: Wasteful of CM space

Solution: microprogram counter (μ PC) for branching

- Analogous to PC at instruction level

Since only MIs have to be fetched from CM, the μ PC is also used as CMAR.

Conditional Branching :

- The condition to be tested is a *condition variable* or *flag* generated by the DPU.
- If several such conditions exist, a *condition select* subfield is included in the microinstruction.
- The branch address may be contained in the MI itself, in which case it is loaded into CMAR, when a branch condition is satisfied.

Another Approach:-

Allow the **condition variables** to modify the contents of the CMAR directly, thus **eliminating** wholly or partly the need for branch addresses in MIs.

Example: overflow when $v=1$

No overflow when $v=0$

Skip on overflow: logically connect v to *count enable input* of μPC at an appropriate point in the MI cycle. Allows *overflow* condition to increment μPC using an extra time, performing *skip* operation.

Micro operation Timing :-

- Each MI generates a **set of CSs**, which are active for duration of MIs' **execution cycle**.
- **Monophase**: a **single** clock pulse synchronizes all control signals.
- **Polyphase**: The number of MIs needed to specify a particular operation can be reduced by dividing the MI cycle into several sequential phases.
- It *adds complexity* as requirement of **specifying the phase** during which a control signal is to be activated.

Example: Register Transfer operation

$R \leftarrow f(R_1, R_2)$ R can be R_1 or R_2

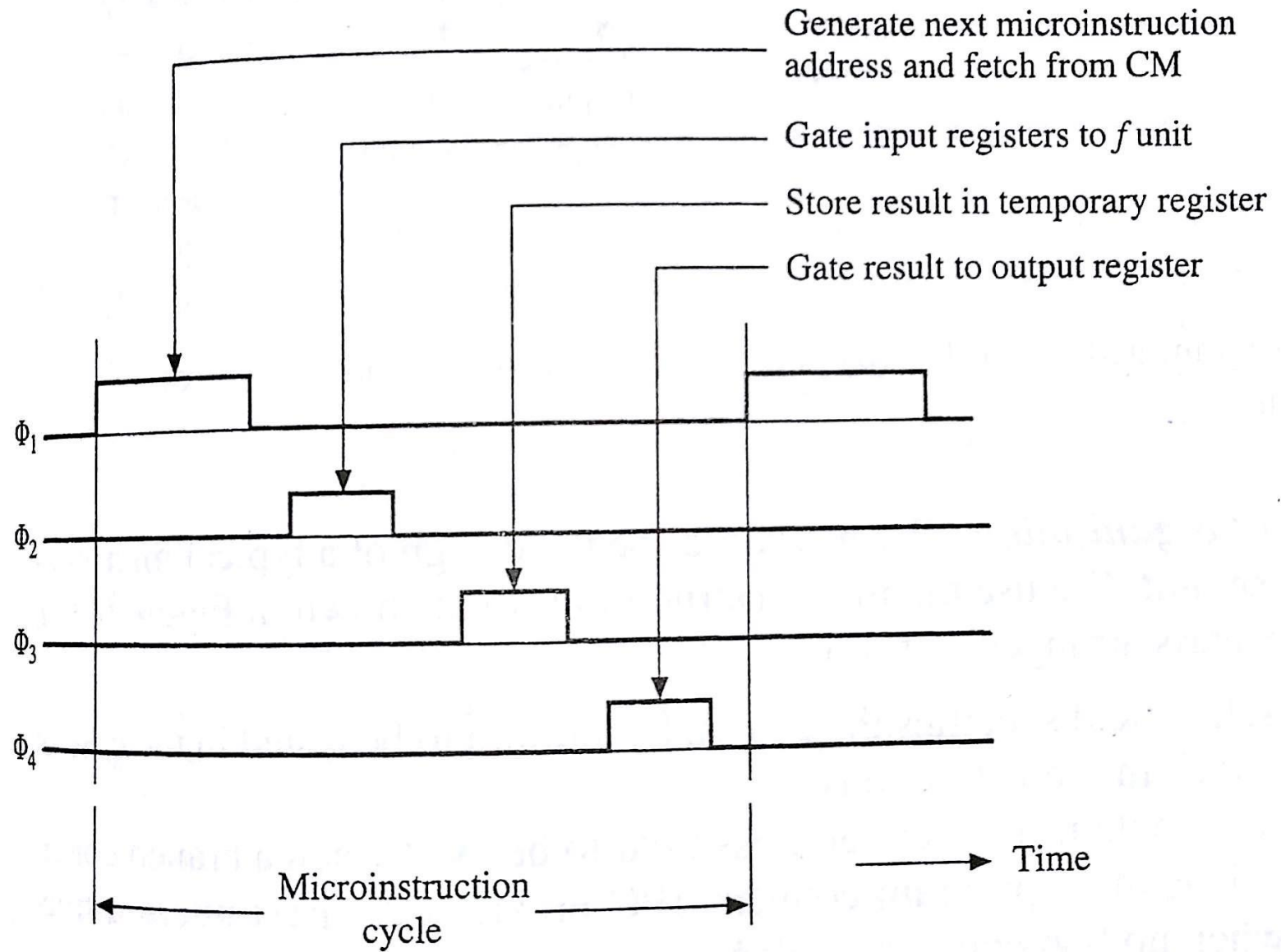
←

Phase Ø1: Transfer the contents of the R_1 & R_2 to the inputs of the f unit

Phase Ø2: Store the result in a temporary register/latch (L).

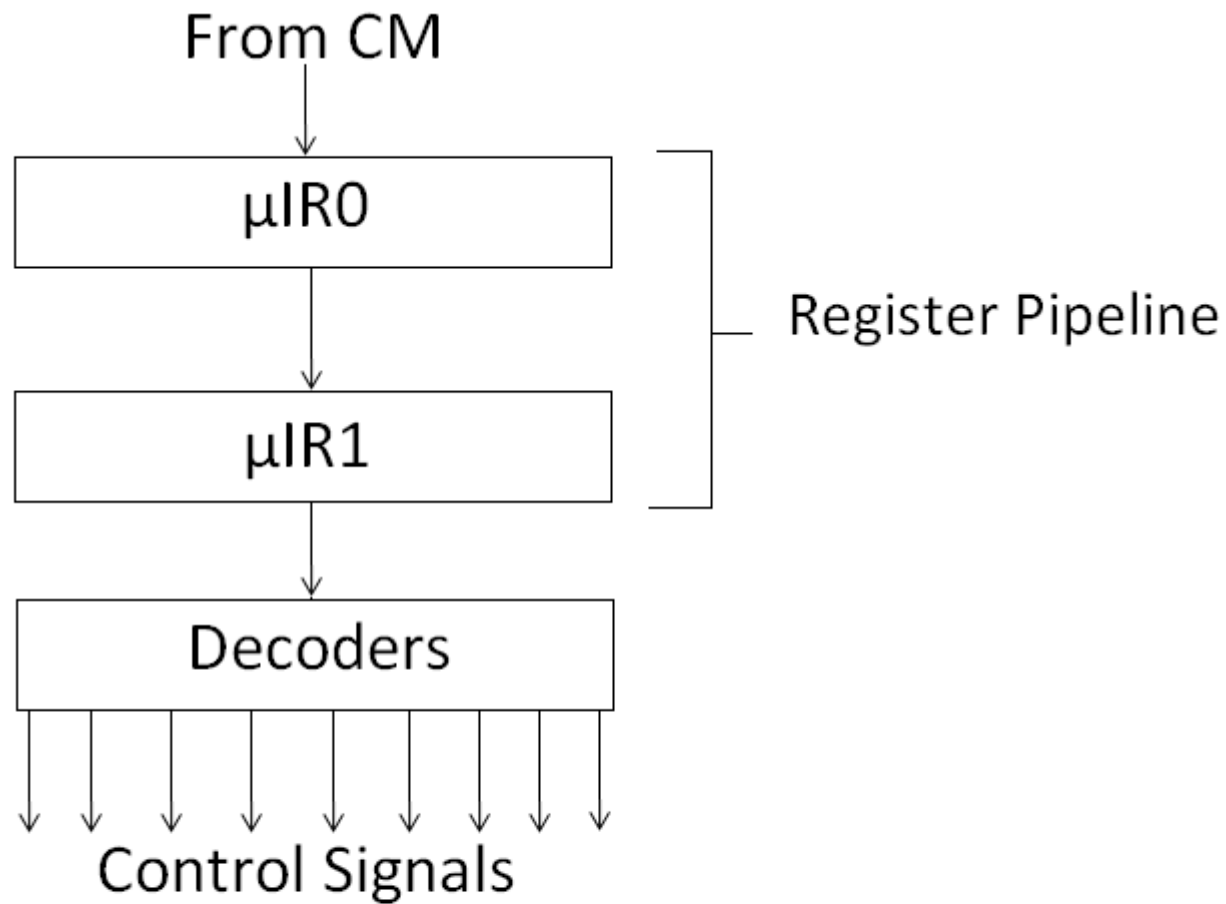
Phase Ø3: Transfer the contents of L to destination register R

Phase Ø4: Fetch the next MI from CM.



Micro operation Timing

- Fetch & execute steps can be overlapped
- Replace μIR by a *pair of registers* forming a two-segment **pipeline**, while one MI in μIR_1 is being executed, the next MI can be fetched and placed in μIR_0 .



Residual Control

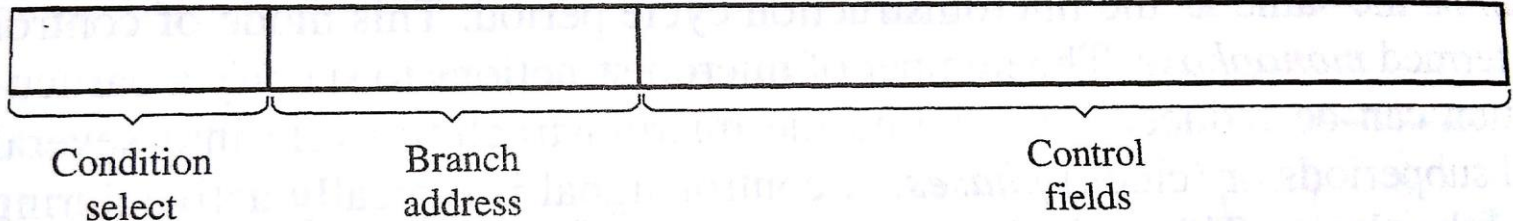
- **Store** the Control field in a **register** which **continues** to exercise until it is modified by a subsequent MI.
- **Application:** When MI is used to allocate a resource e.g. **connection between two units** may be established by an MI and maintained for a long period via Residual Control.

Control Unit Organization

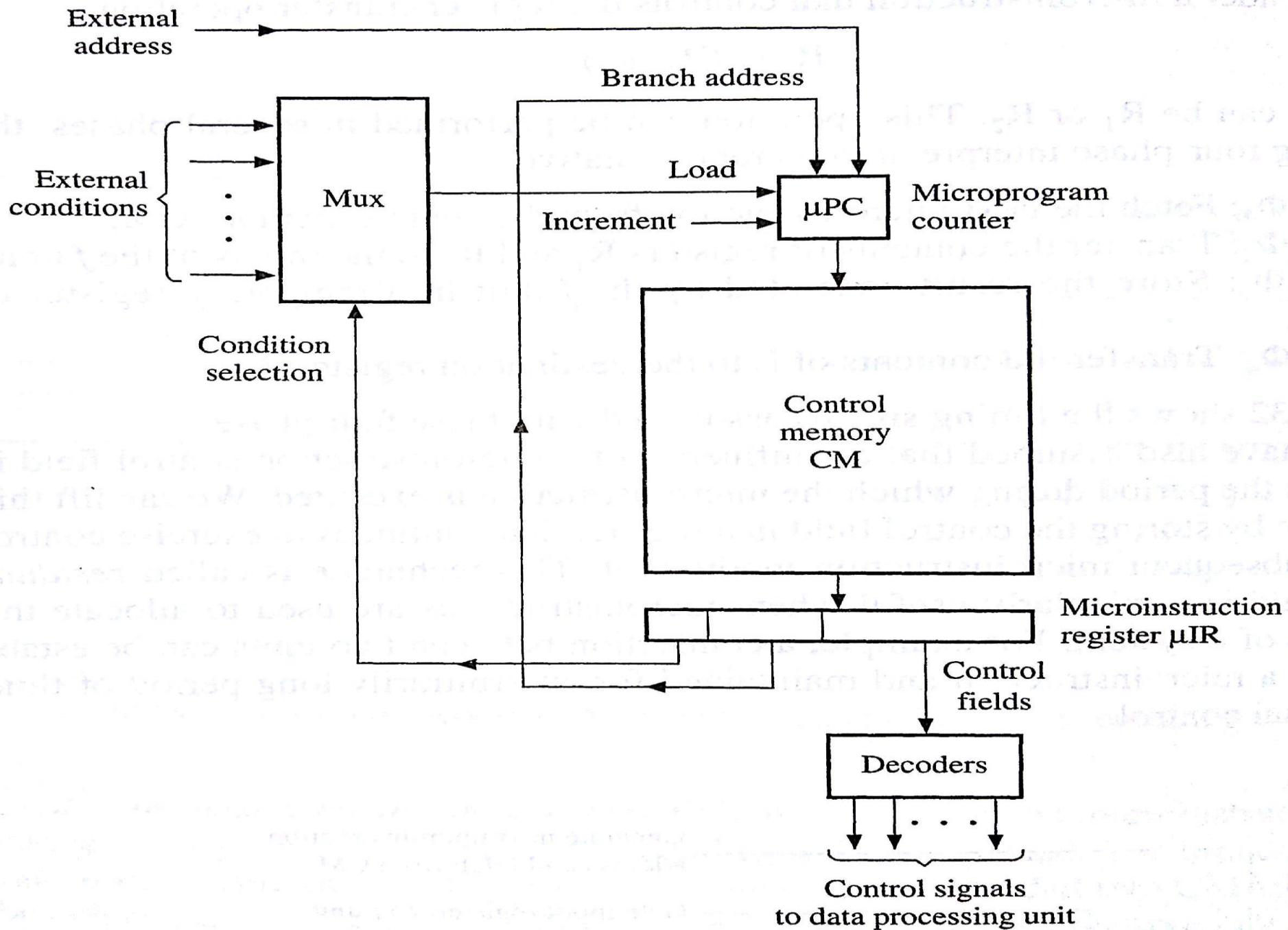
- A *Condition Select* field is used to specify the external condition to be tested in case of conditional branch MI.
- An *Address field* contains the next address to be used when a branch condition is satisfied (μ PC provides next MI address when no branching)

Control Unit Organization

- Rest specifies (un/encoded) format of the CS that must be activated to perform the desired micro operation.



MI format



- The **contents** of the addressed word in CM are transferred to the μ IR
- Control fields are **decoded** (if required) and used to generate CS for DPU
- μ PC is then **incremented**.
- For **branch** instruction in μ IR, the **contents** of the MI Address field are **loaded** in μ PC.
- MUX activates **parallel-load** control i/p of μ PC based on **status of the external** condition variables

s_0	s_1	Meaning
0	0	No branching
0	1	Branch if $v_1 = 1$
1	0	Branch if $v_2 = 1$
1	1	Unconditional branch

MUX: x_0, x_1, x_2, x_3 (4 i/ps)
 x_i routed to MUX o/p when
 $S_0S_1 = i$

- A provision is made for loading μ PC with an address from an external source, used for loading the starting address of the desired micro program in cases where CM contains more than one micro program.

Nanoprogrammed Computer

- In some machines, the **MI**s do not directly issue the **signals** that control the hardware.
- They are used to access **2nd CM**, called nanoControl Memory (**nCM**).
- First used in 1970 by Nanodata Corp.

Nanoprogrammed Computer

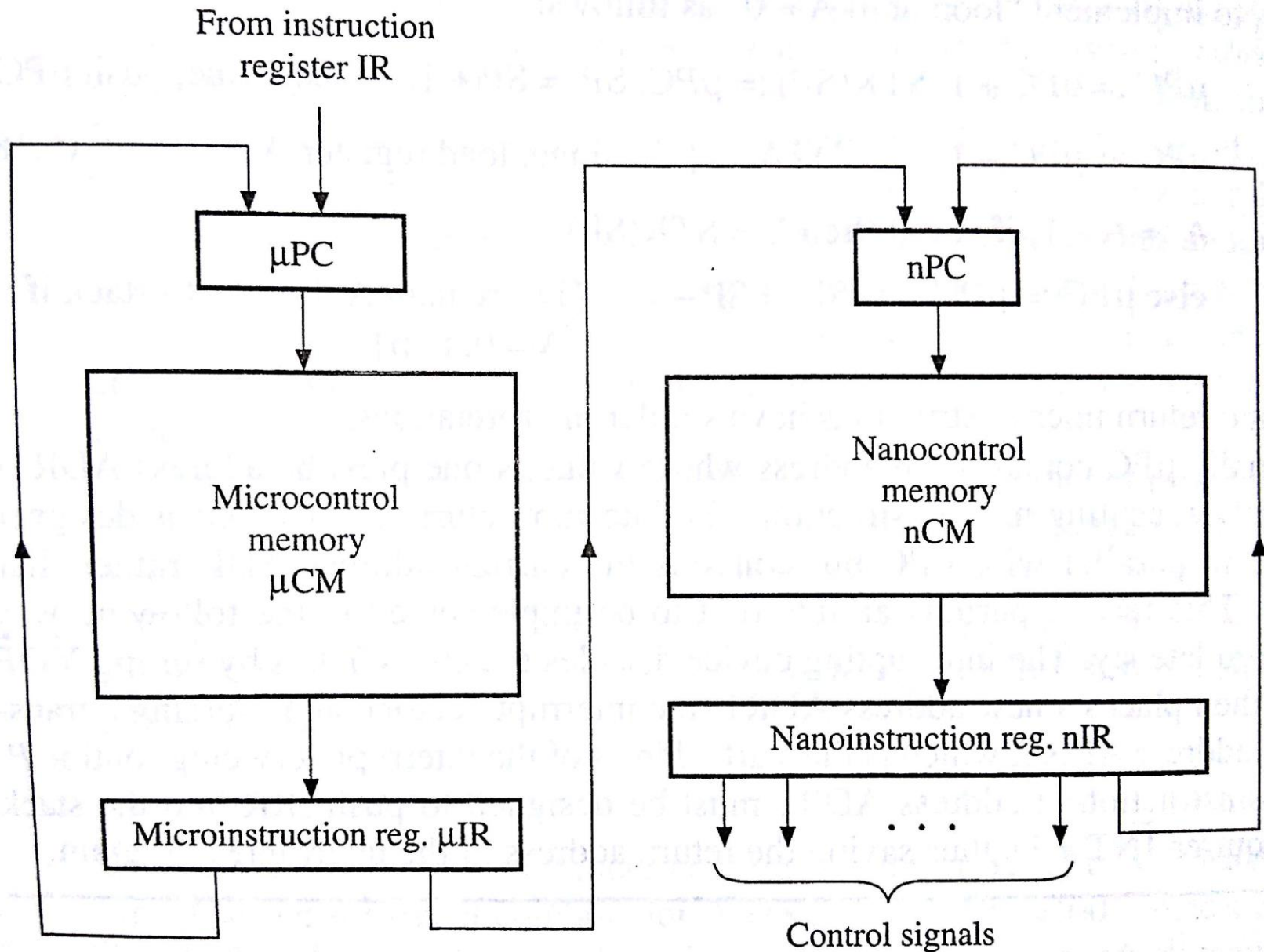
Two levels of CMs:

1. Higher level: Microcontrol memory μCM (MI)
2. Lower Level: Nanocontrol memory $n\text{CM}$
(nanoinstruction - NI)

Consider the dimensions:

μCM : $H_m \times W_m$

$n\text{CM}$: $H_n \times W_n$



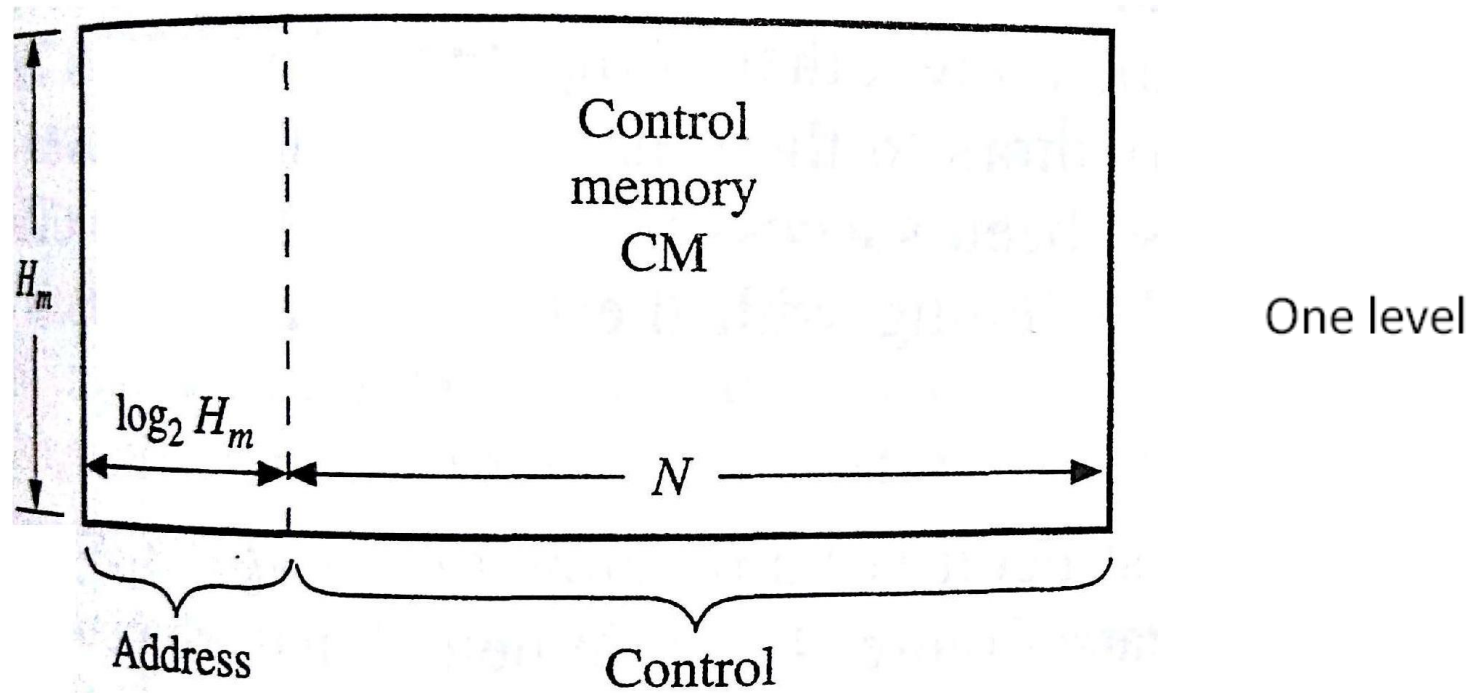
Two level control store organization for nanoprogramming

Nanoprogrammed Computer

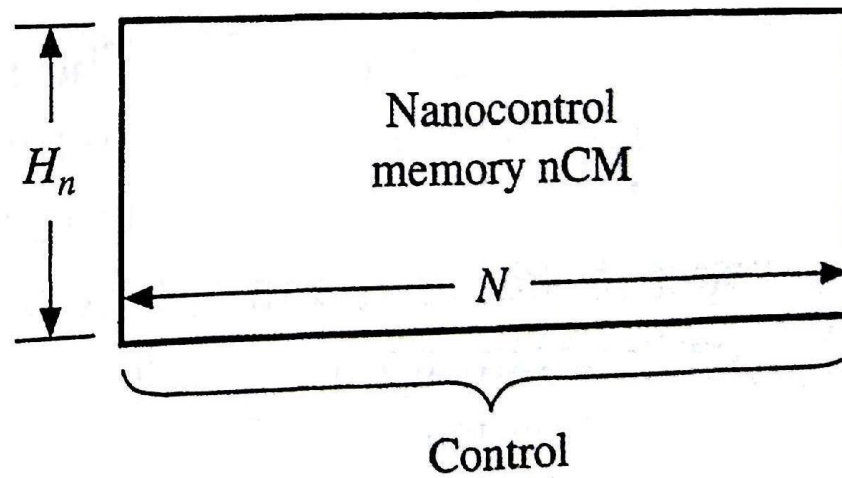
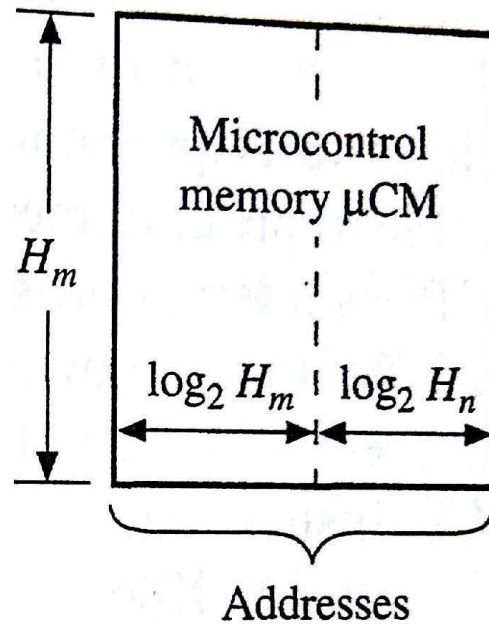
- Typically, **MIS** are encoded in a **vertical format**, so that although H_m is large, W_m is small.
- **NIs** usually have a highly parallel **horizontal** format making W_n large.
- If many MIs can be interpreted by the same nanoprogram, then H_n can be kept relatively small.

Advantages

- It can reduce the total size of CMs needed, this translates to smaller chip area.
- Greater design flexibility resulting from the loosening the bonds between instruction and hardware with two intermediate levels of control rather than one.



$$S1 = H_m (N + \lceil \log_2 H_m \rceil) \quad \dots\dots\dots(i)$$



Two level

- In 2-level design, μCM again stores H_m MIs, but N -bit Control fields are transferred to $n\text{CM}$.
- In place of latter, each MI in μCM contains $[\log_2 H_n]$ bit address to specify any NI location in $n\text{CM}$.

Hence size becomes,

$$S_2 = H_m ([\log_2 H_m] + [\log_2 H_n]) + NH_n$$

(Assuming no branching, so no explicit addressing)

$$S_2 = H_m ([\log_2 H_m] + [\log_2 H_n]) + NH_n$$

Suppose all control bit patterns are different, it can be written as

$$H_n = r H_m$$

where r = ratio of the number of unique control states to the total number of H_n of control states needed to implement all instructions

So,

$$\begin{aligned} S_2 &= H_m ([\log_2 H_m] + [\log_2 r H_m]) + NrH_m \\ &= H_m (2[\log_2 H_m] + [\log_2 r] + Nr) \quad \text{.....(ii)} \end{aligned}$$

If $N=70$, $H_m=650$, $r=0.4$ so that $H_n=260$

$$S_1 = 52,240$$

$$S_2 = 30,550$$

That means, $52450-30550=21850$ bits of control storage (42% of S_1) are saved.

In general, 2-level design requires less memory space if

$$S_2 < S_1$$

Hence, from equation (i) & (ii), the following inequality must be satisfied

$$N \geq [\log_2 H_m] + [\log_2 r] + rN$$

Disadvantage

Reduction in speed due to extra memory access for nCM and a more complex CU organization.