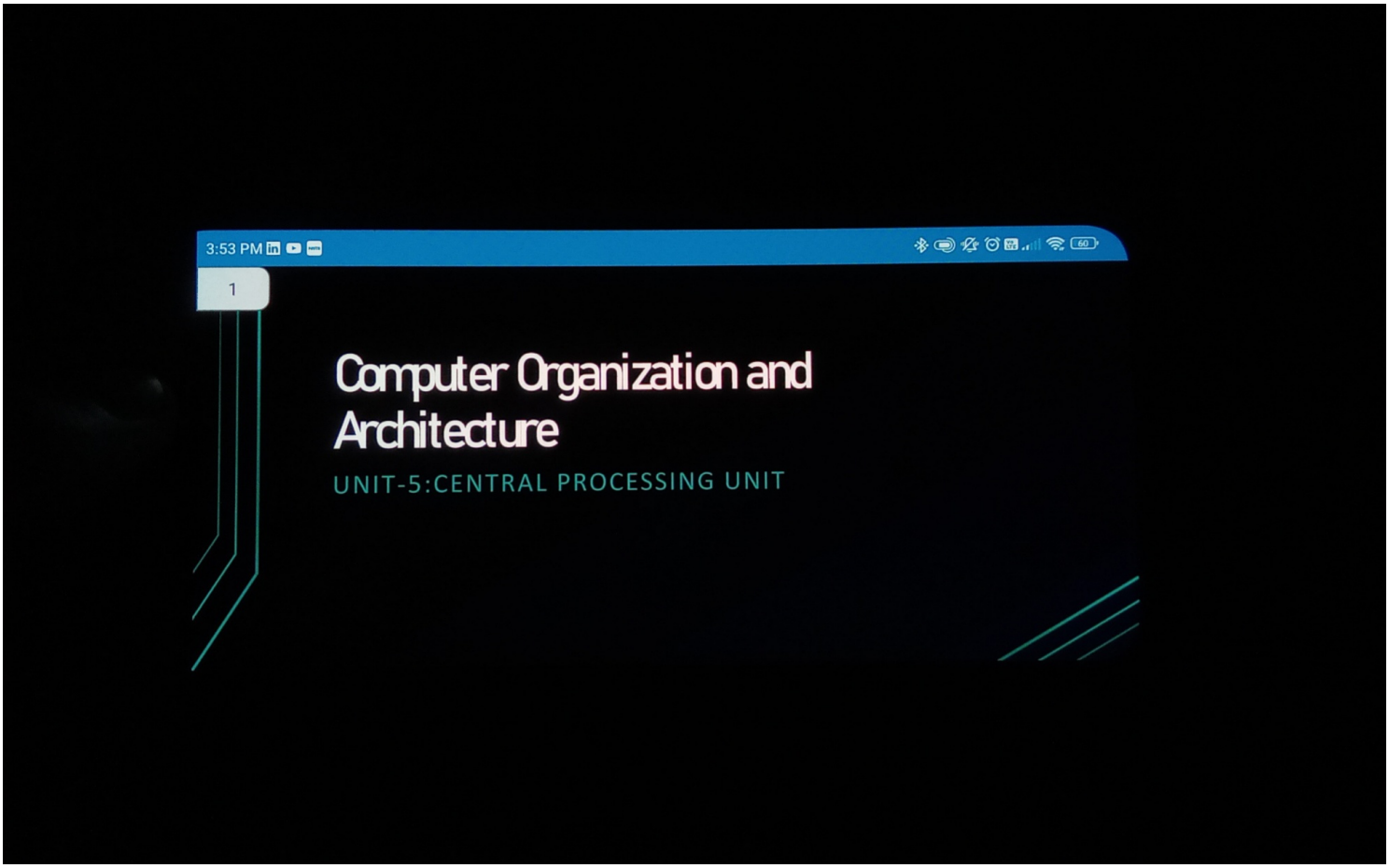
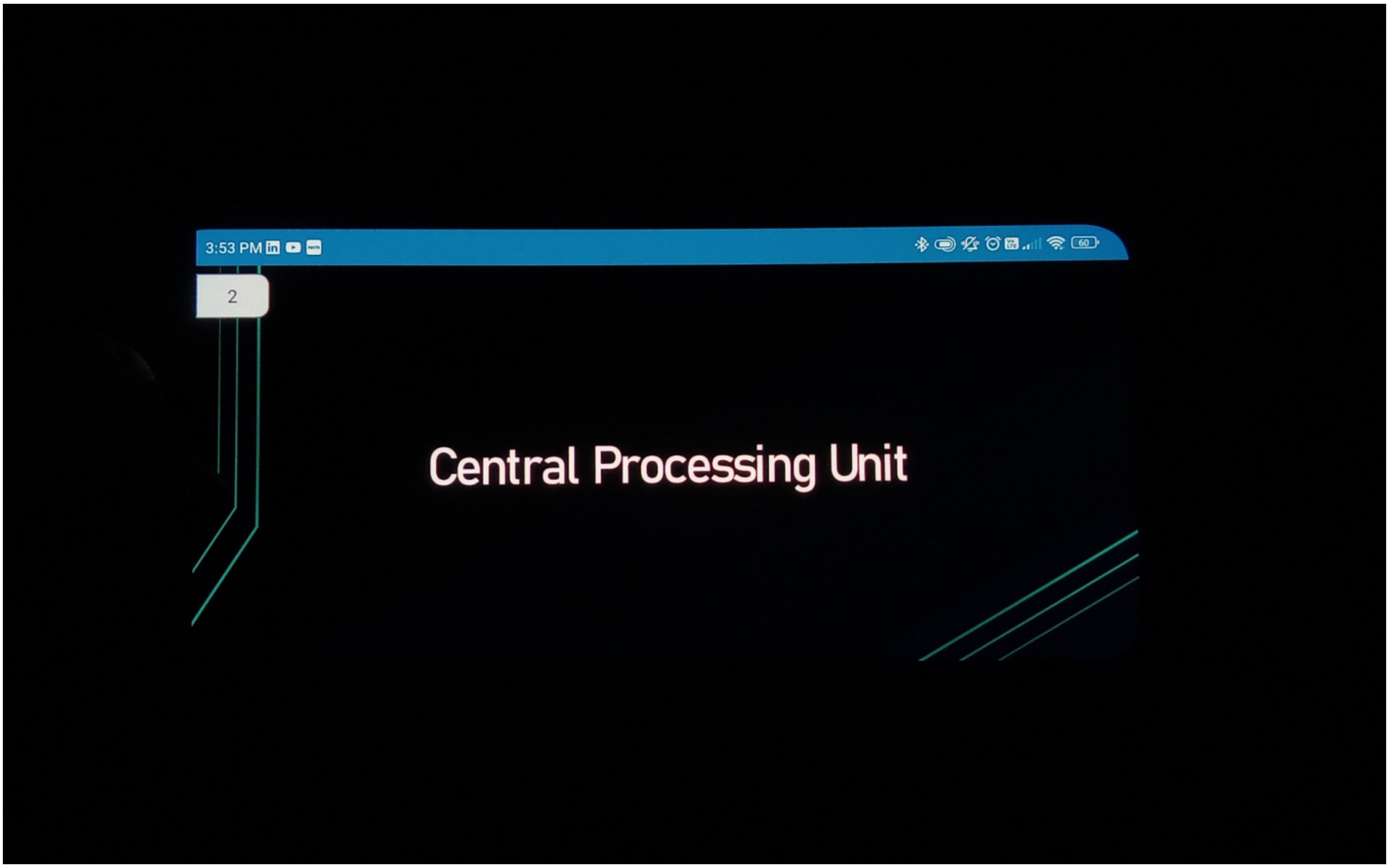
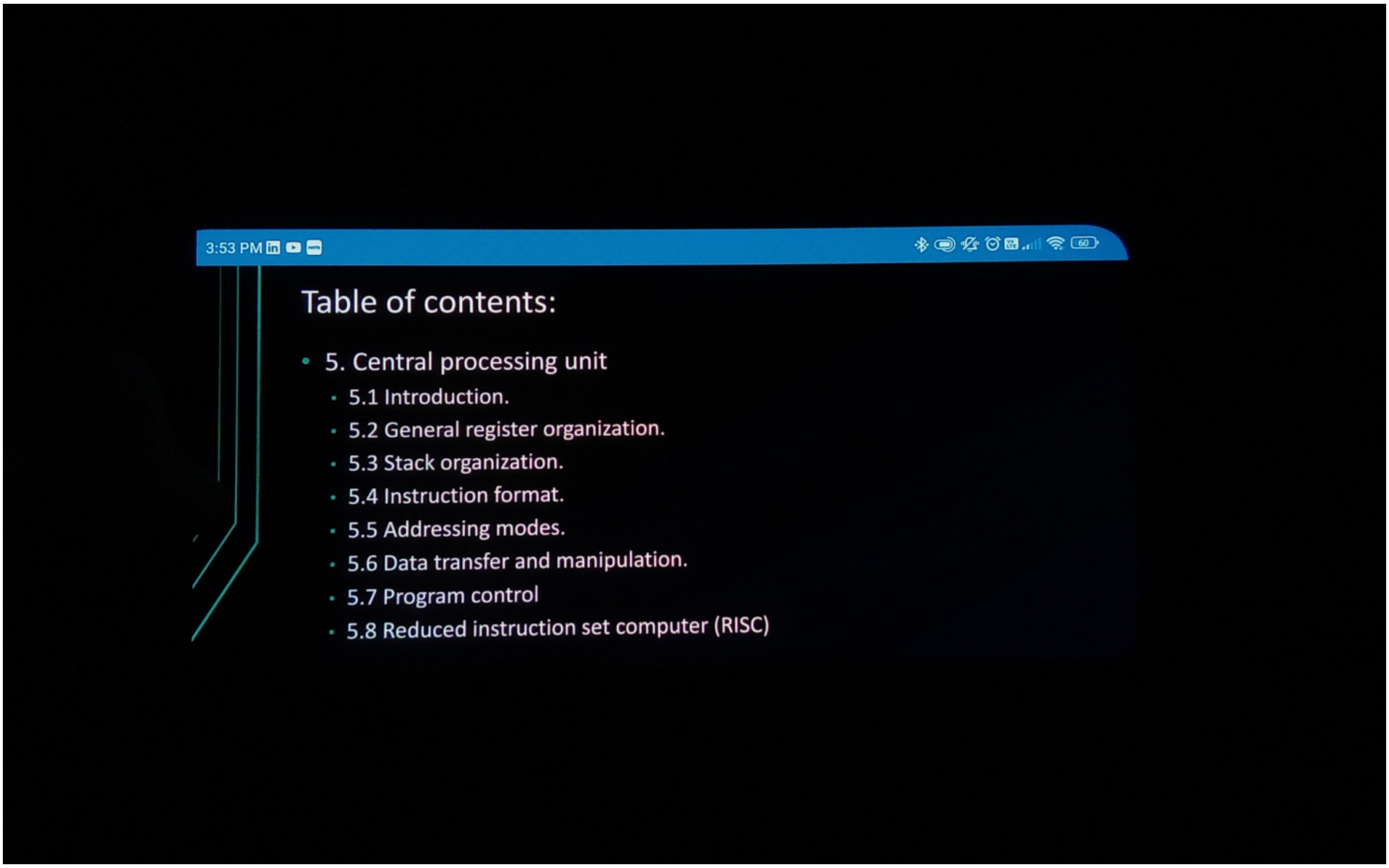
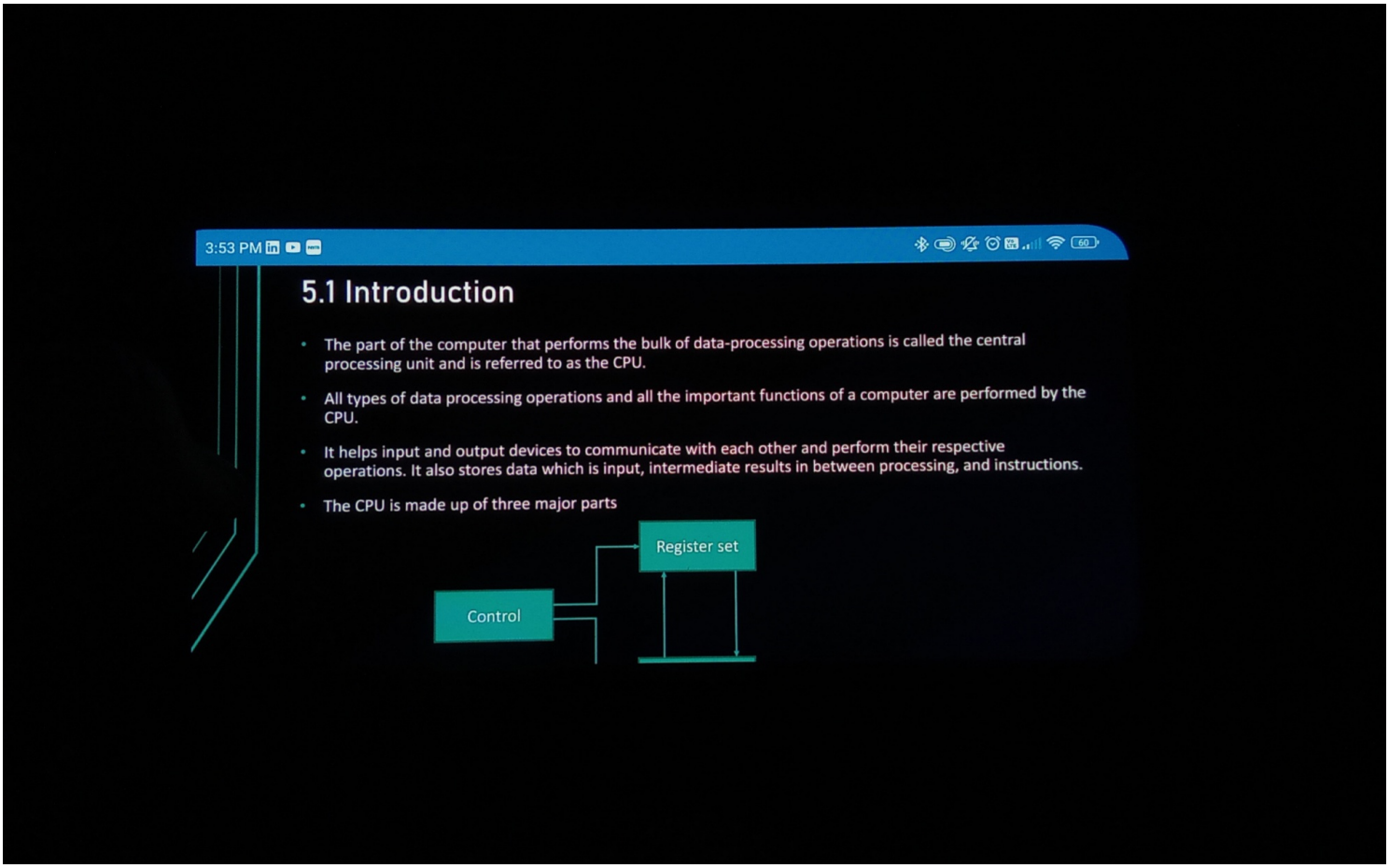
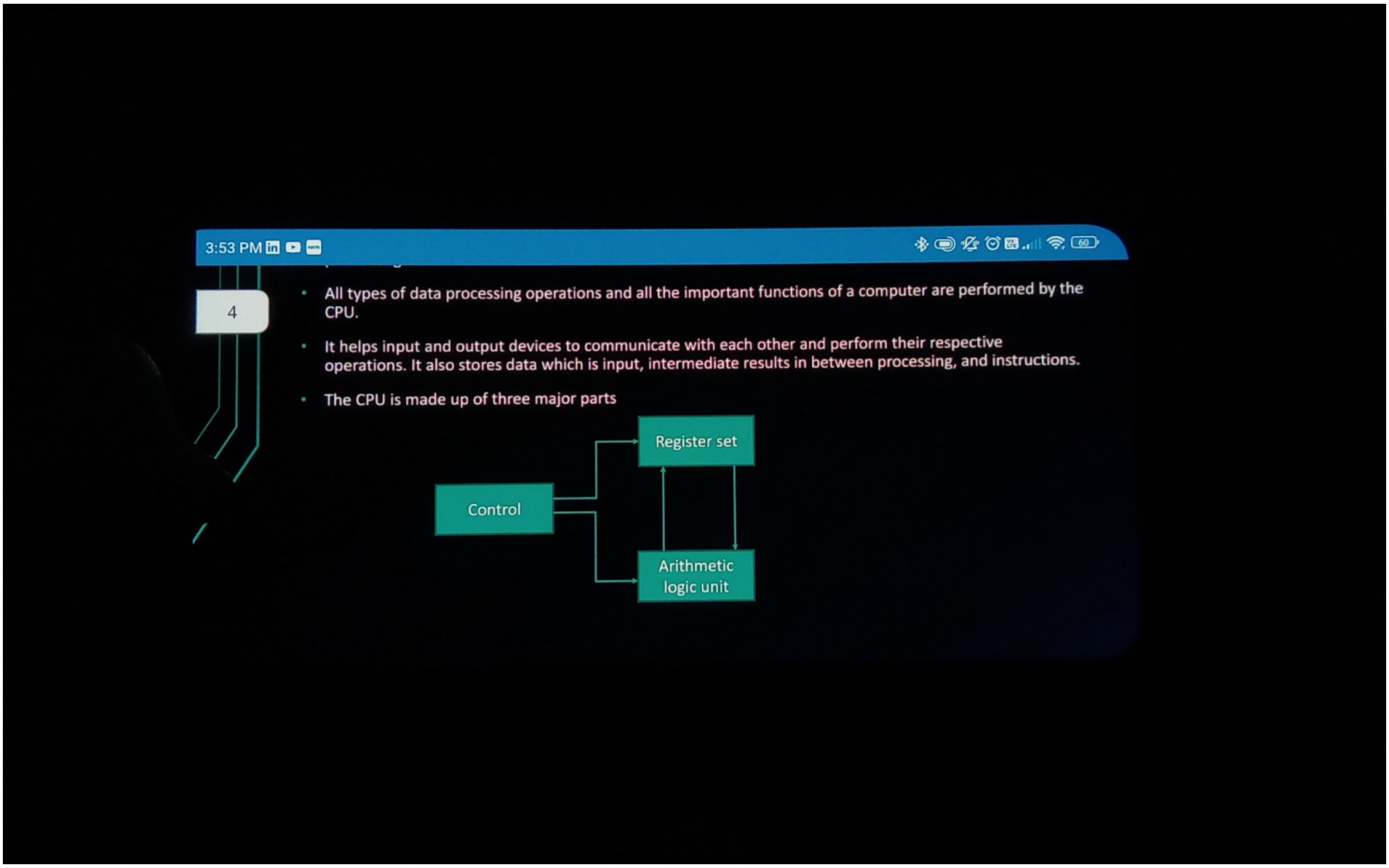
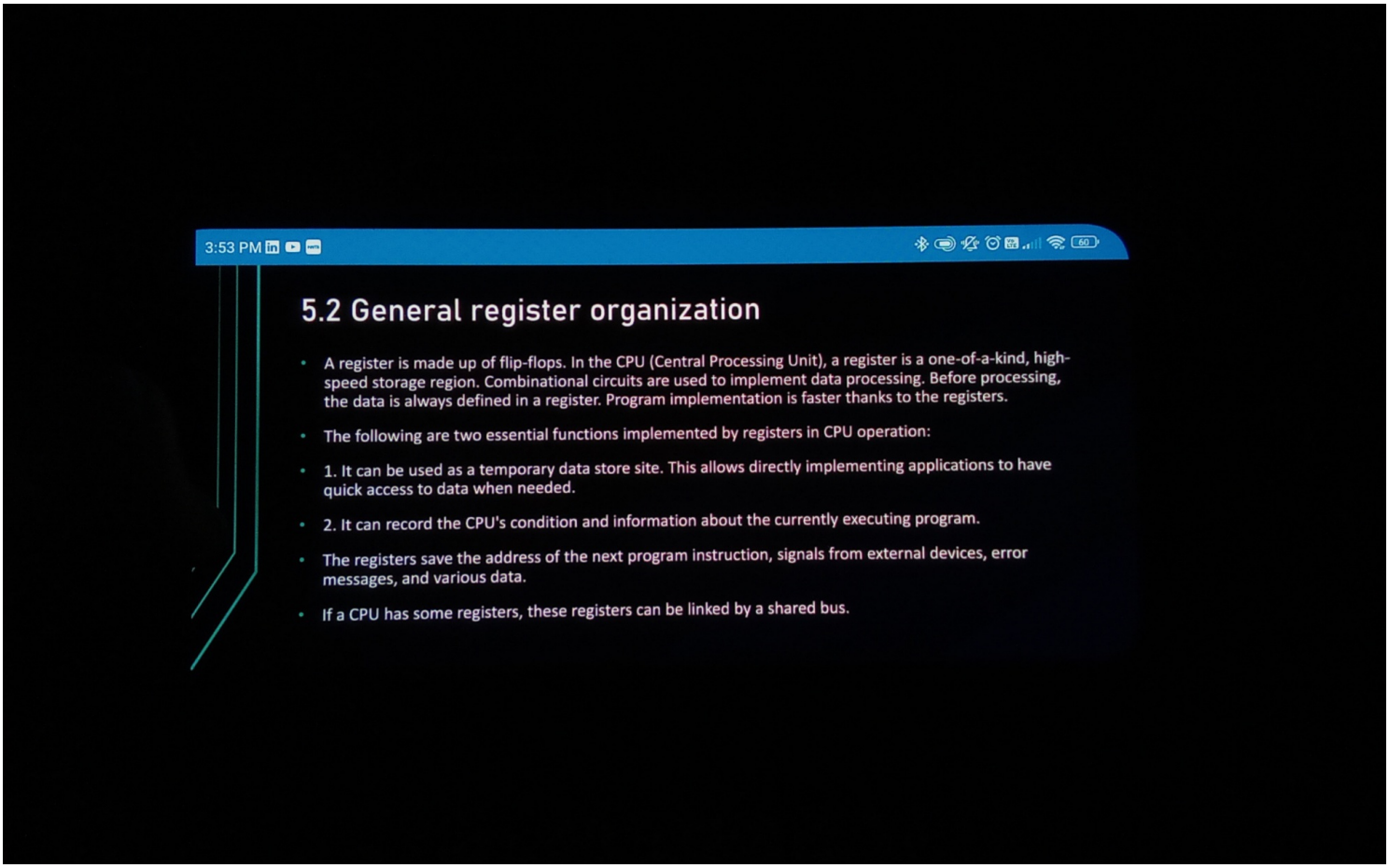

3:53 PM
1
Computer Organization and Architecture
UNIT-5:CENTRAL PROCESSING UNIT

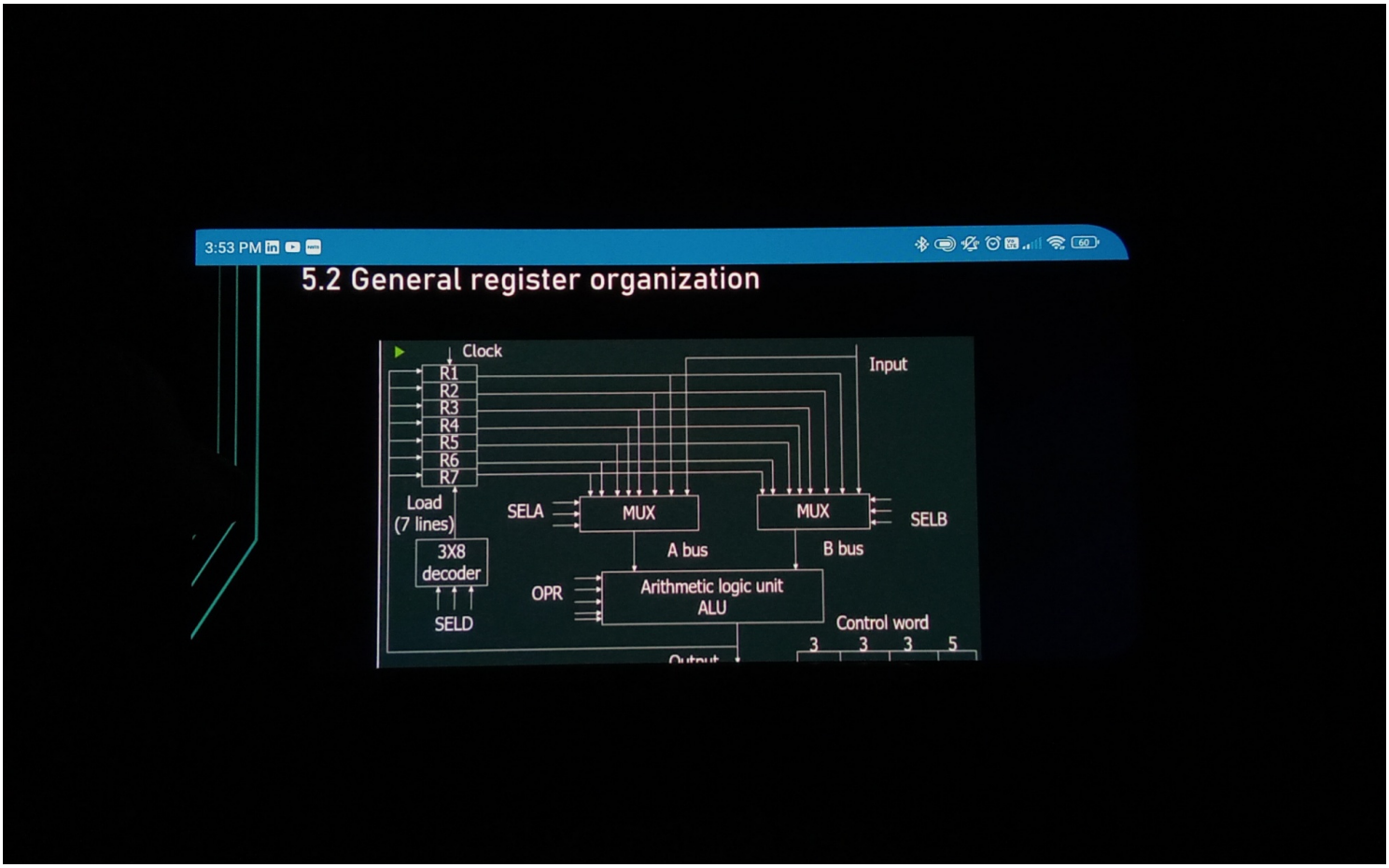

3:53 PM
2
Central Processing Unit

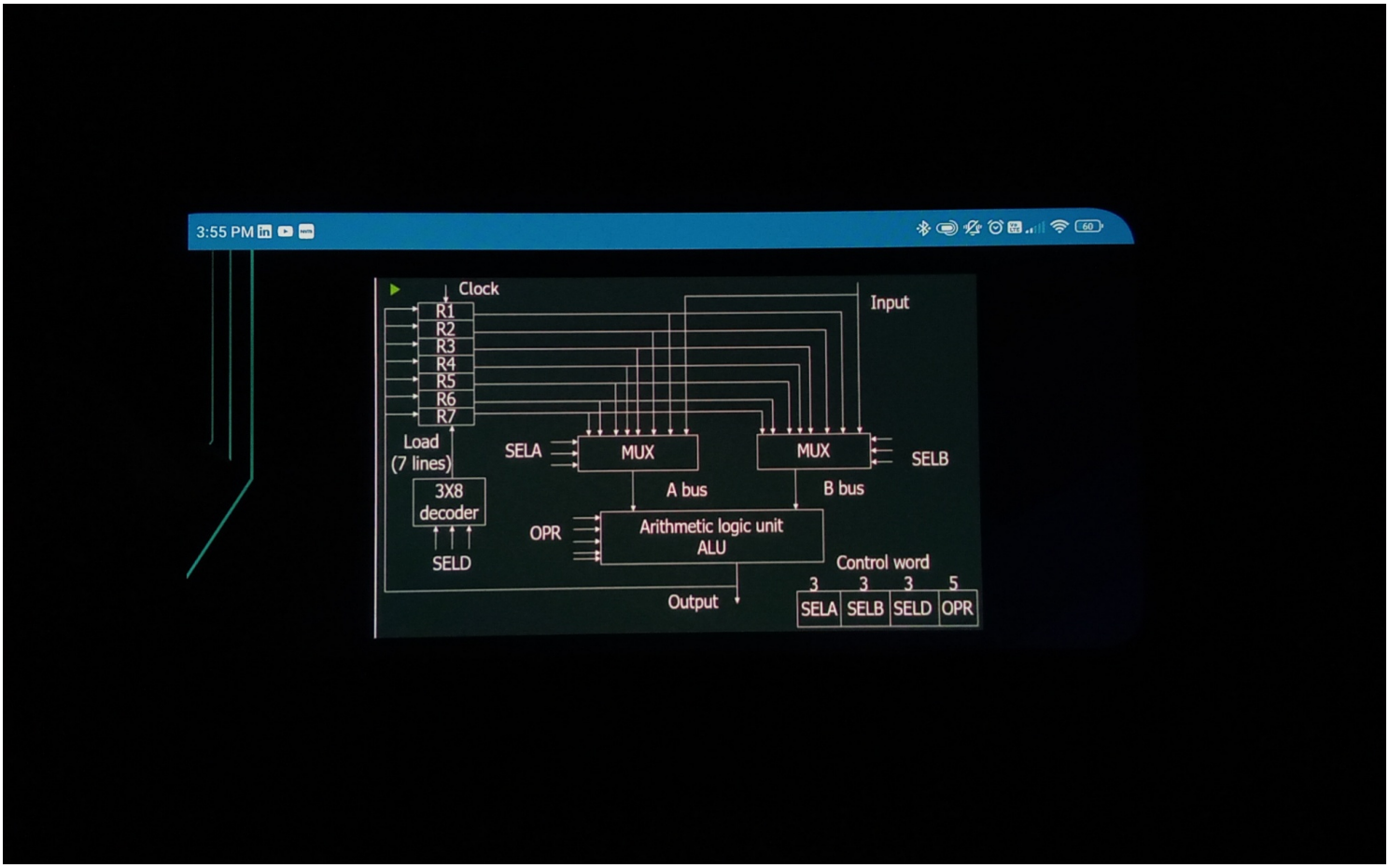

3:53 PM
Table of contents:
5. Central processing unit
5.1 Introduction.
5.2 General register organization.
5.3 Stack organization.
5.4 Instruction format.
5.5 Addressing modes.
5.6 Data transfer and manipulation.
5.7 Program control
5.8 Reduced instruction set computer (RISC)

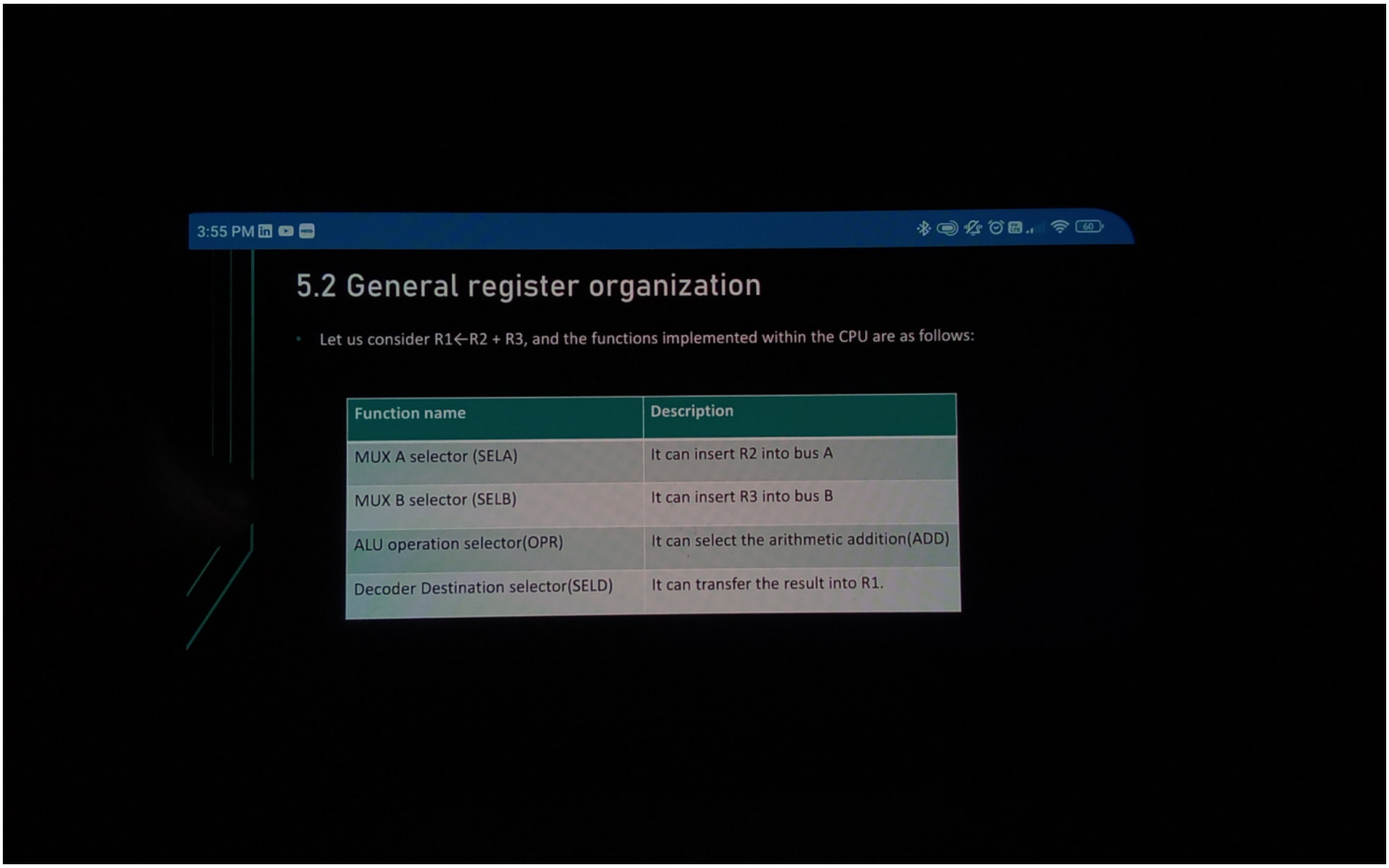

3:53 PM
5.1 Introduction
The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
All types of data processing operations and all the important functions of a computer are performed by the CPU.
It helps input and output devices to communicate with each other and perform their respective operations. It also stores data which is input, intermediate results in between processing, and instructions.
The CPU is made up of three major parts
Register set
Control

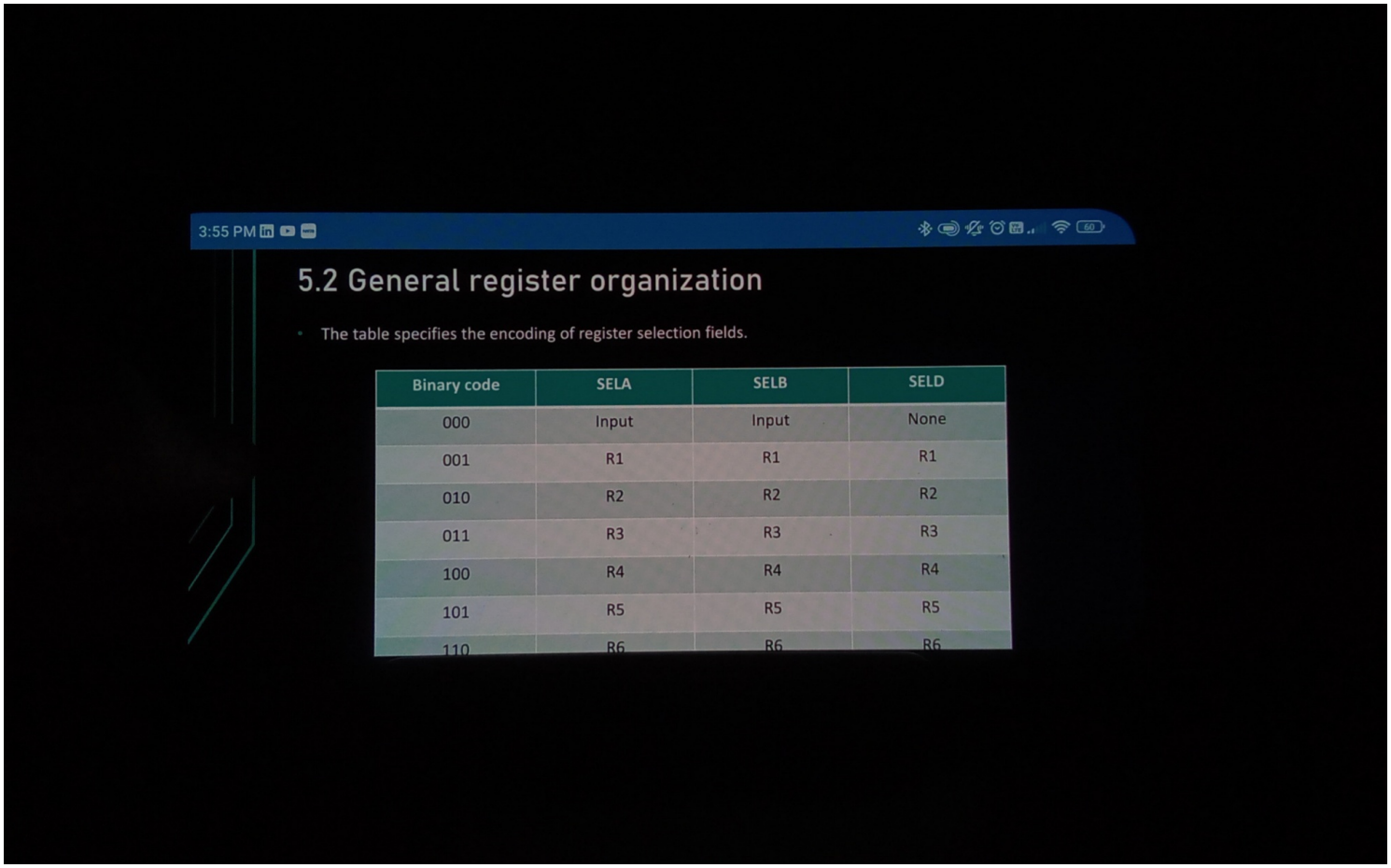

3:53 PM
4
All types of data processing operations and all the important functions of a computer are performed by the CPU.
It helps input and output devices to communicate with each other and perform their respective operations. It also stores data which is input, intermediate results in between processing, and instructions.
The CPU is made up of three major parts
Register set
Control
Arithmetic logic unit

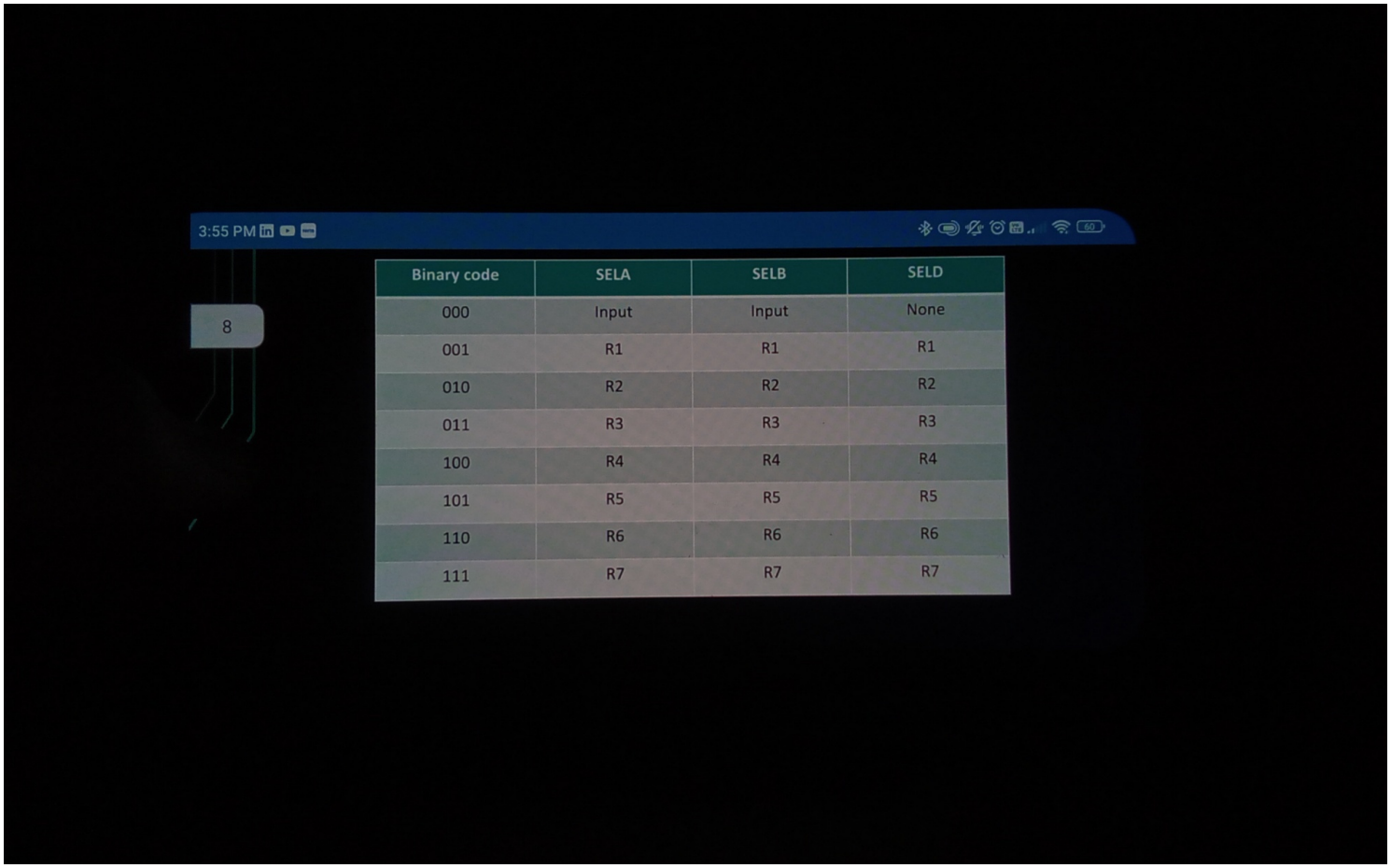

3:53 PM
5.2 General register organization
A register is made up of flip-flops. In the CPU (Central Processing Unit), a register is a one-of-a-kind, high-speed storage region. Combinational circuits are used to implement data processing. Before processing, the data is always defined in a register. Program implementation is faster thanks to the registers.
The following are two essential functions implemented by registers in CPU operation:
1. It can be used as a temporary data store site. This allows directly implementing applications to have quick access to data when needed.
2. It can record the CPU's condition and information about the currently executing program.
The registers save the address of the next program instruction, signals from external devices, error messages, and various data.
If a CPU has some registers, these registers can be linked by a shared bus.


3:53 PM
5.2 General register organization
Clock
Input
R1
R2
R3
R4
R5
R6
R7
Load
(7 lines)
SELA
MUX
MUX
SELB
3X8
decoder
A bus
B bus
OPR
Arithmetic logic unit
ALU
SELD
Control word
3
3
3
5

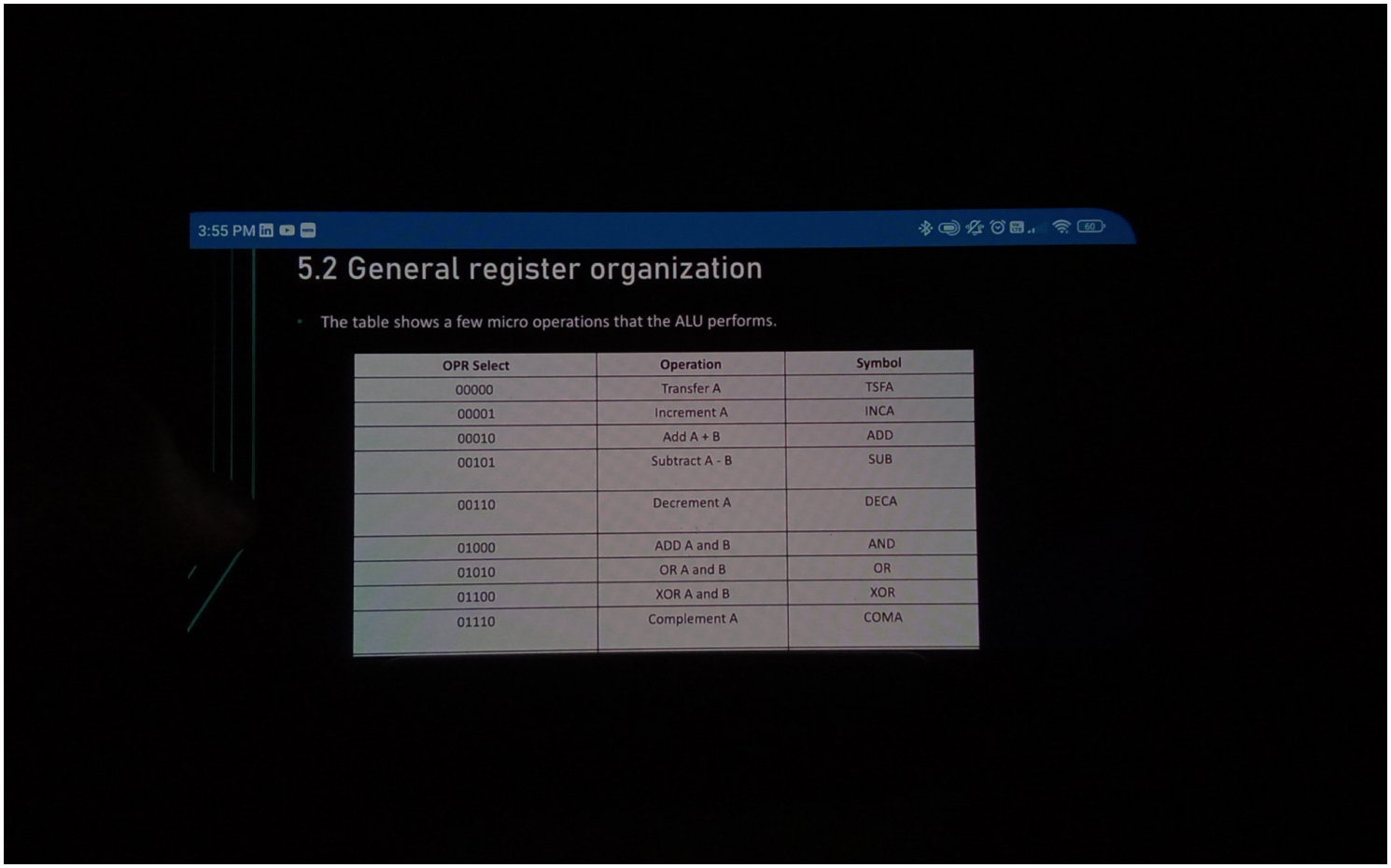

3:55 PM
Clock
R1
R2
R3
R4
R5
R6
R7
Input
Load
(7 lines)
SELA
MUX
MUX
SELB
3X8
decoder
A bus
B bus
OPR
Arithmetic logic unit
ALU
SELD
Control word
3
3
3
5
Output
SELA
SELB
SELD
OPR

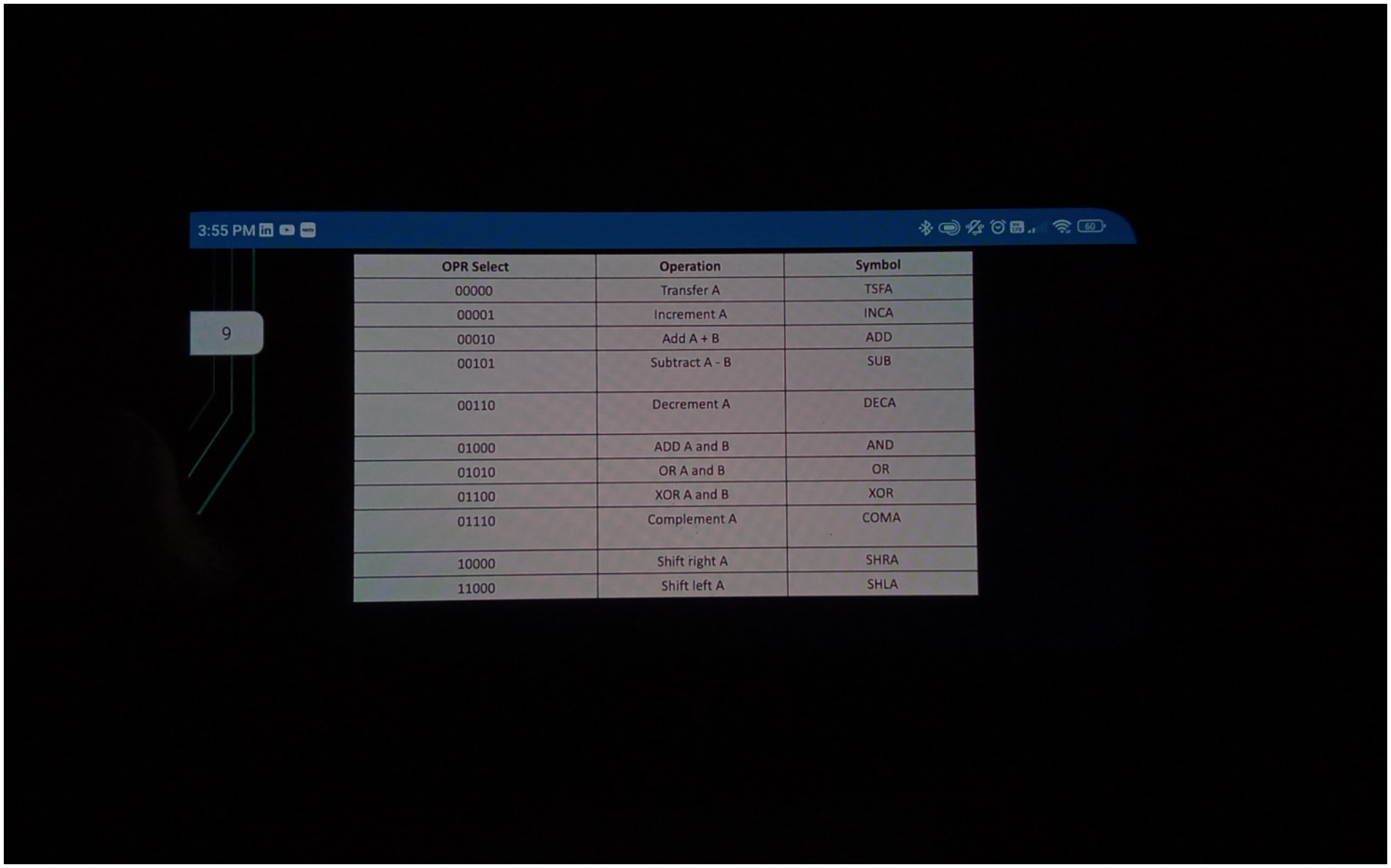

3:55 PM
5.2 General register organization
Let us consider R1←R2 + R3, and the functions implemented within the CPU are as follows:
Function name
Description
MUX A selector (SELA)
It can insert R2 into bus A
MUX B selector (SELB)
It can insert R3 into bus B
ALU operation selector(OPR)
It can select the arithmetic addition(ADD)
Decoder Destination selector(SELD)
It can transfer the result into R1.


3:55 PM
5.2 General register organization
The table specifies the encoding of register selection fields.
Binary code
SELA
SELB
SELD
000
Input
Input
None
001
R1
R1
R1
010
R2
R2
R2
011
R3
R3
R3
100
R4
R4
R4
101
R5
R5
R5
110
R6
R6
R6

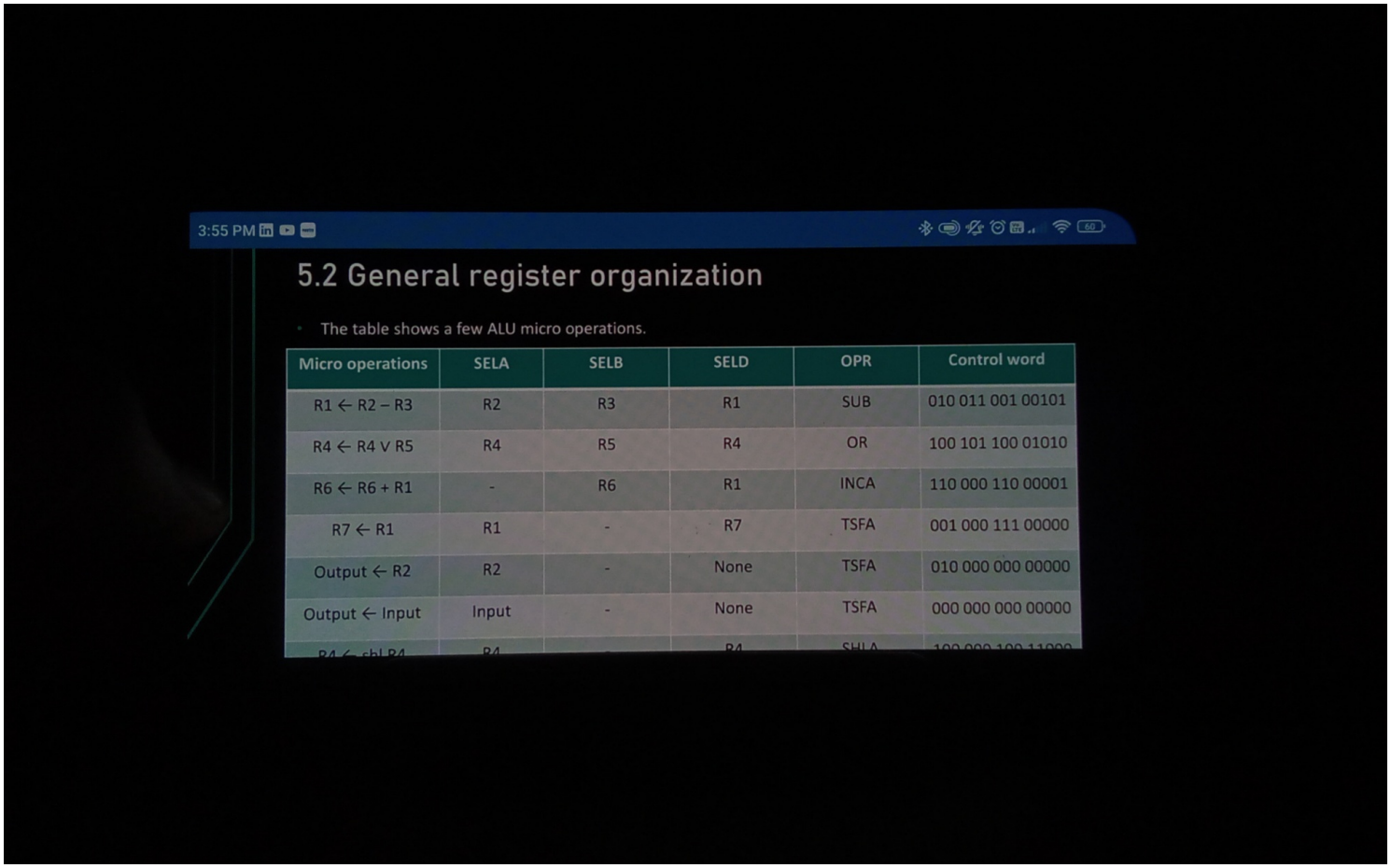


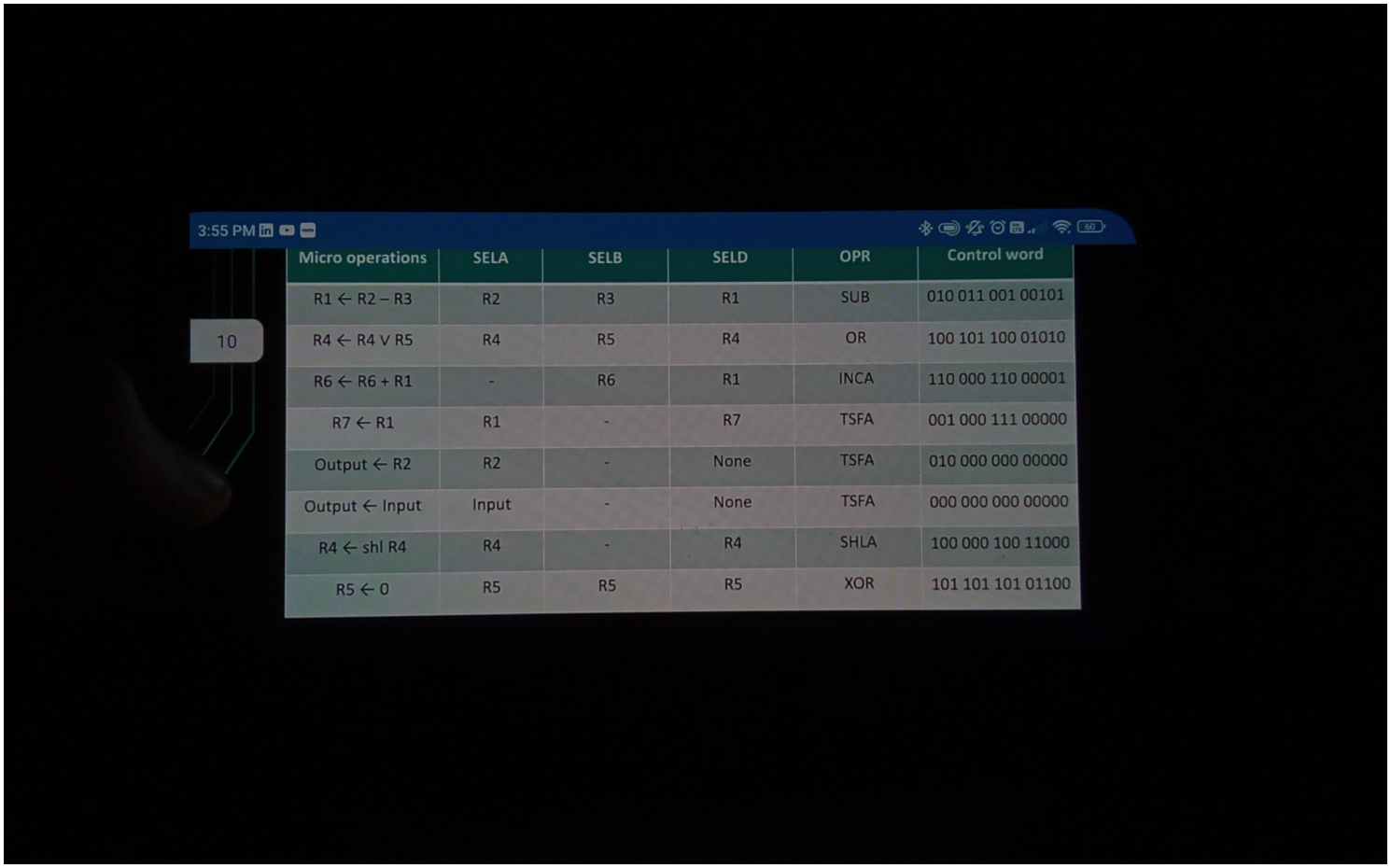
Binary code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

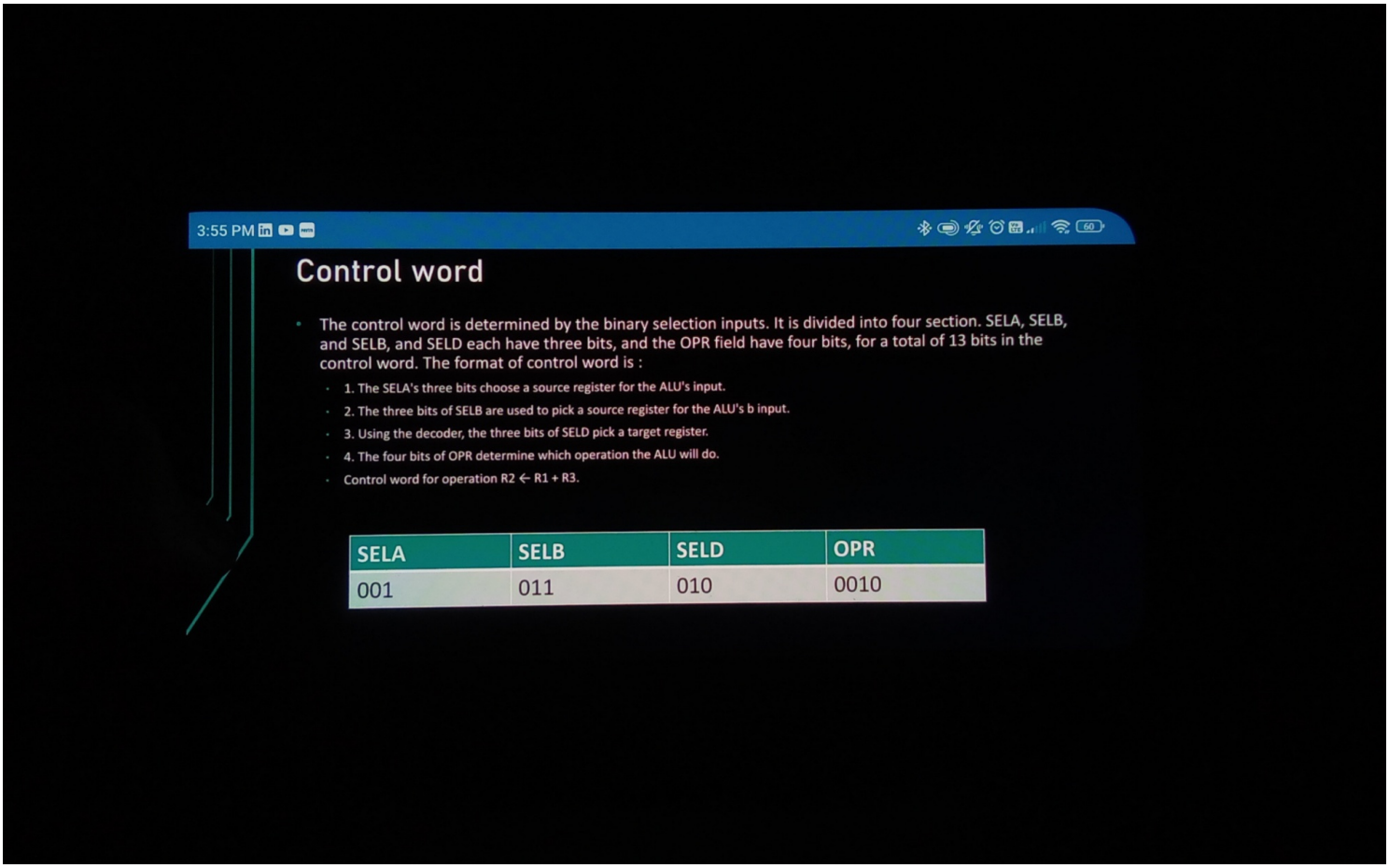

3:55 PM
5.2 General register organization
The table shows a few micro operations that the ALU performs.
OPR Select
Operation
Symbol
00000
Transfer A
TSFA
00001
Increment A
INCA
00010
Add A + B
ADD
00101
Subtract A - B
SUB
00110
Decrement A
DECA
01000
ADD A and B
AND
01010
OR A and B
OR
01100
XOR A and B
XOR
01110
Complement A
COMA

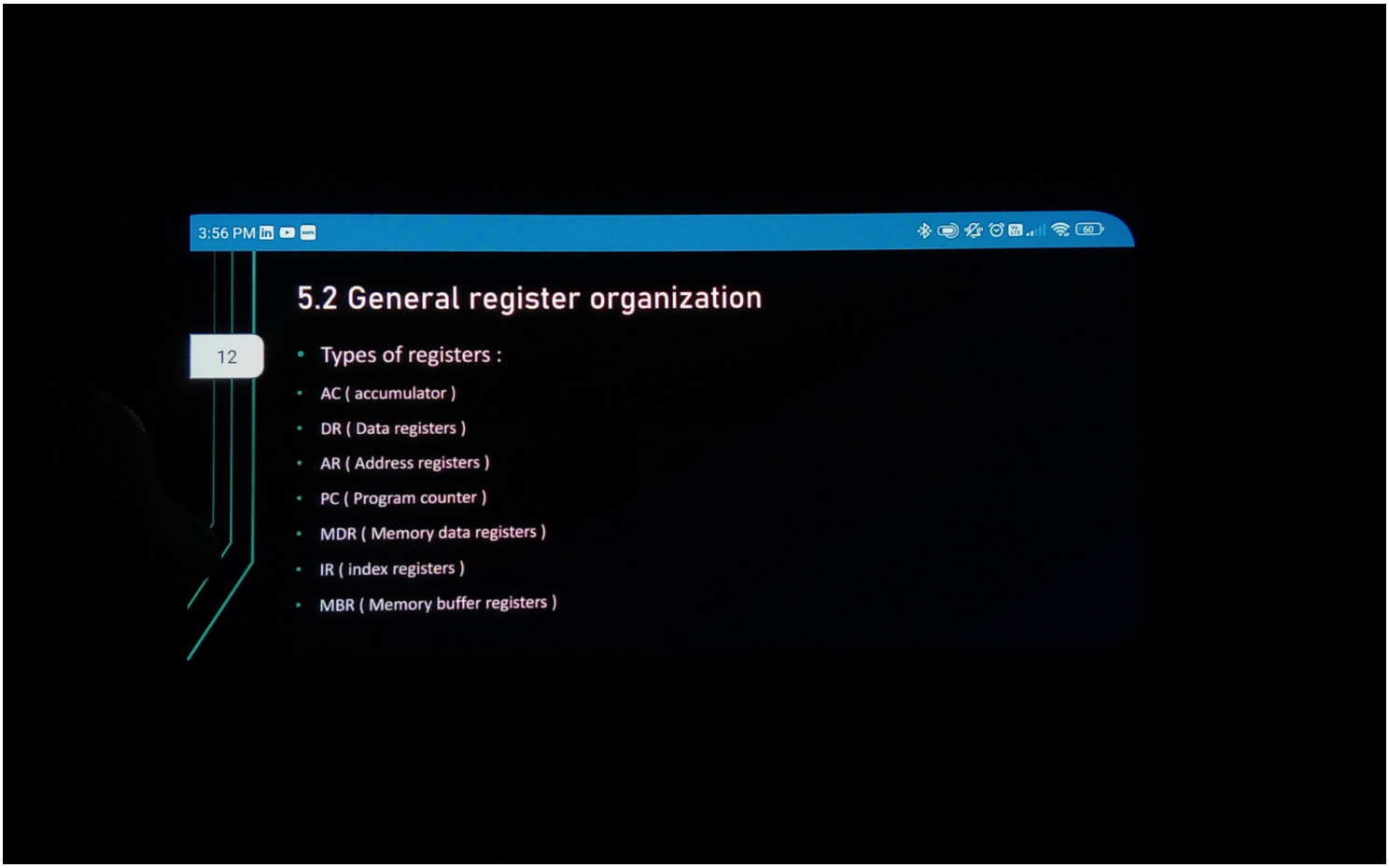


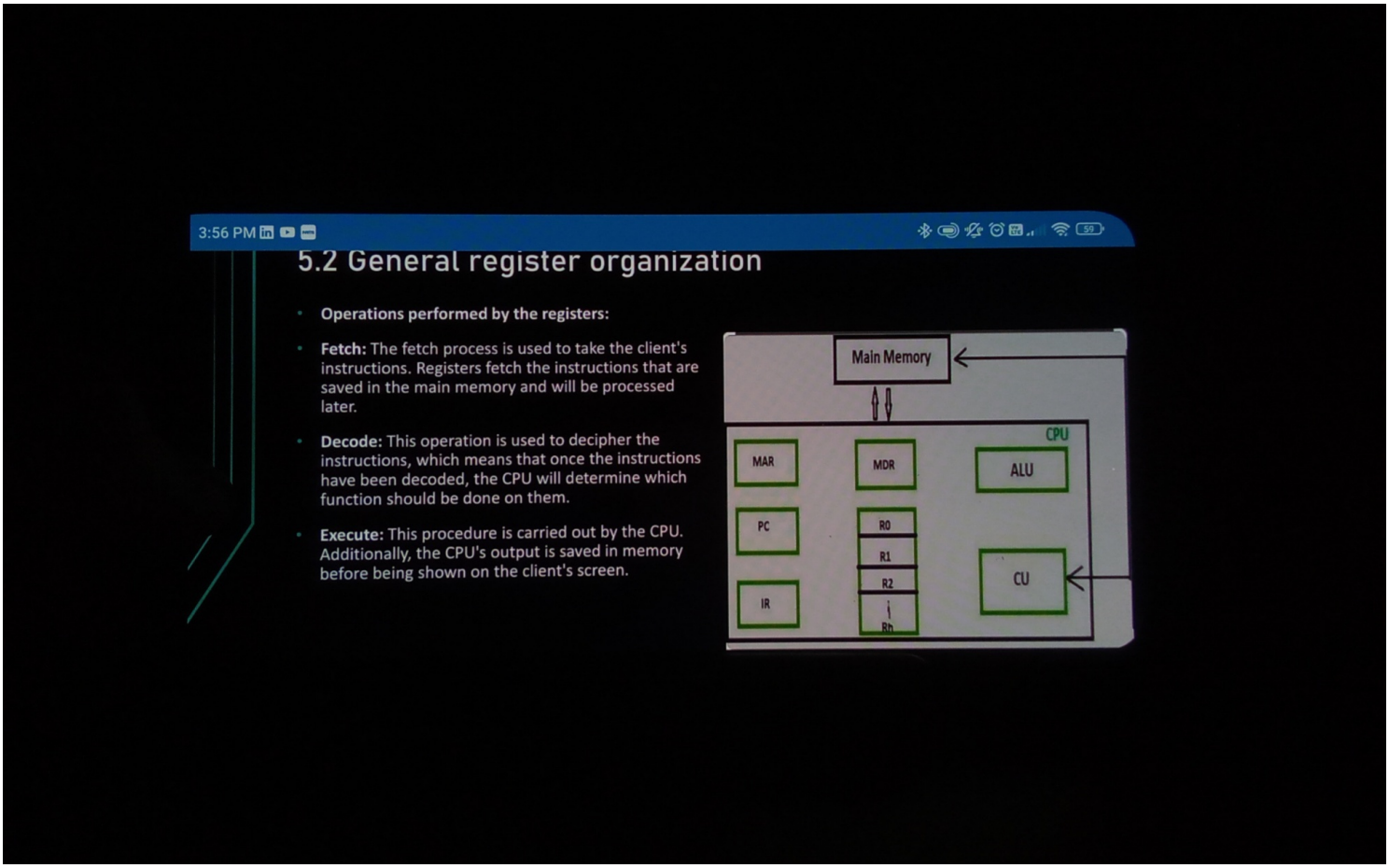
OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	ADD A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

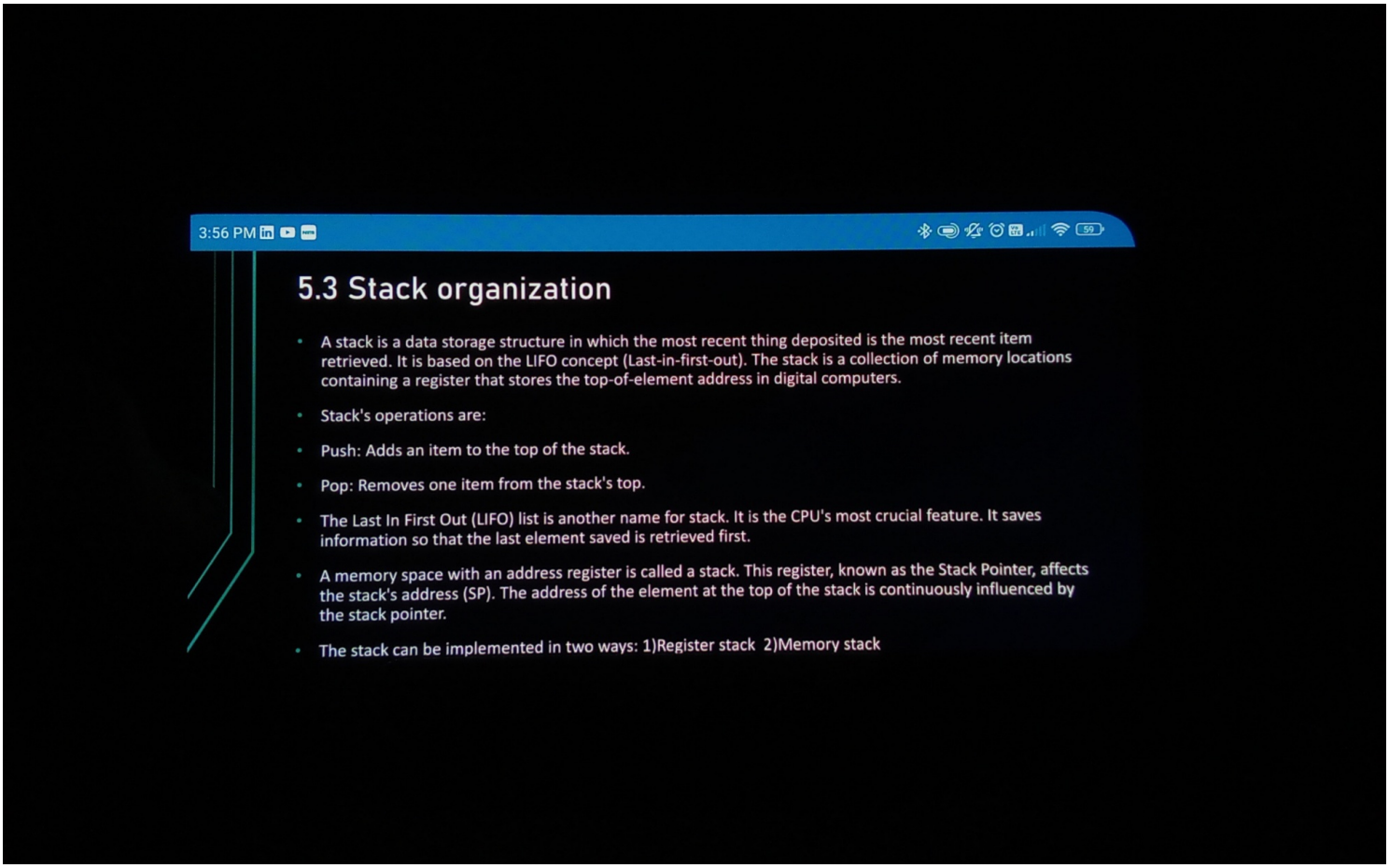

3:55 PM
5.2 General register organization
The table shows a few ALU micro operations.
Micro operations | SELA | SELB | SELD | OPR | Control word
R1 ← R2 – R3 | R2 | R3 | R1 | SUB | 010 011 001 00101
R4 ← R4 V R5 | R4 | R5 | R4 | OR | 100 101 100 01010
R6 ← R6 + R1 | - | R6 | R1 | INCA | 110 000 110 00001
R7 ← R1 | R1 | - | R7 | TSFA | 001 000 111 00000
Output ← R2 | R2 | - | None | TSFA | 010 000 000 00000
Output ← Input | Input | - | None | TSFA | 000 000 000 00000


3:55 PM
10
Micro operations | SELA | SELB | SELD | OPR | Control word
R1 ← R2 – R3 | R2 | R3 | R1 | SUB | 010 011 001 00101
R4 ← R4 V R5 | R4 | R5 | R4 | OR | 100 101 100 01010
R6 ← R6 + R1 | - | R6 | R1 | INCA | 110 000 110 00001
R7 ← R1 | R1 | - | R7 | TSFA | 001 000 111 00000
Output ← R2 | R2 | - | None | TSFA | 010 000 000 00000
Output ← Input | Input | - | None | TSFA | 000 000 000 00000
R4 ← shl R4 | R4 | - | R4 | SHLA | 100 000 100 11000
R5 ← 0 | R5 | R5 | R5 | XOR | 101 101 101 01100


3:55 PM
Control word
The control word is determined by the binary selection inputs. It is divided into four section. SELA, SELB, and SELB, and SELD each have three bits, and the OPR field have four bits, for a total of 13 bits in the control word. The format of control word is :
1. The SELA's three bits choose a source register for the ALU's input.
2. The three bits of SELB are used to pick a source register for the ALU's b input.
3. Using the decoder, the three bits of SELD pick a target register.
4. The four bits of OPR determine which operation the ALU will do.
Control word for operation R2 ← R1 + R3.
SELA SELB SELD OPR
001 011 010 0010


3:56 PM
5.2 General register organization
12
Types of registers :
AC (accumulator)
DR (Data registers)
AR (Address registers)
PC (Program counter)
MDR (Memory data registers)
IR (index registers)
MBR (Memory buffer registers)

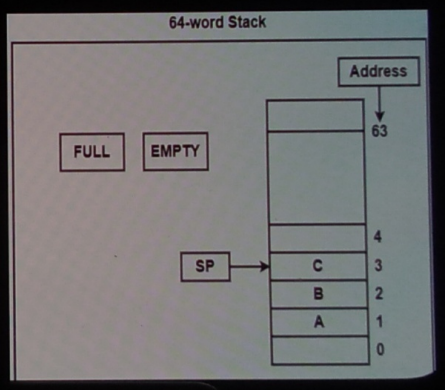

3:56 PM
5.2 General register organization
Operations performed by the registers:
Fetch: The fetch process is used to take the client's instructions. Registers fetch the instructions that are saved in the main memory and will be processed later.
Decode: This operation is used to decipher the instructions, which means that once the instructions have been decoded, the CPU will determine which function should be done on them.
Execute: This procedure is carried out by the CPU. Additionally, the CPU's output is saved in memory before being shown on the client's screen.
Main Memory
CPU
MAR
MDR
ALU
PC
R0
R1
R2
CU
IR


3:56 PM
5.3 Stack organization
A stack is a data storage structure in which the most recent thing deposited is the most recent item retrieved. It is based on the LIFO concept (Last-in-first-out). The stack is a collection of memory locations containing a register that stores the top-of-element address in digital computers.
Stack's operations are:
Push: Adds an item to the top of the stack.
Pop: Removes one item from the stack's top.
The Last In First Out (LIFO) list is another name for stack. It is the CPU's most crucial feature. It saves information so that the last element saved is retrieved first.
A memory space with an address register is called a stack. This register, known as the Stack Pointer, affects the stack's address (SP). The address of the element at the top of the stack is continuously influenced by the stack pointer.
The stack can be implemented in two ways: 1)Register stack 2)Memory stack

3:56 PM

Register Stack

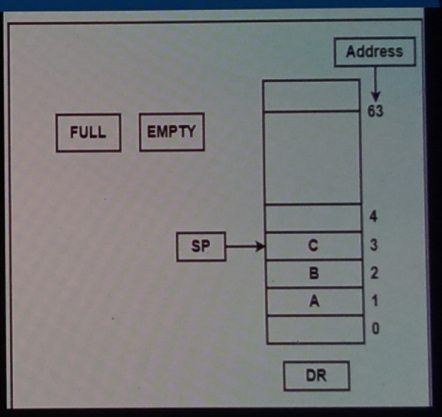
- A stack of memory words or registers may be placed on top of each other. Consider a 64-word register stack like the one shown in the diagram.
- A binary number, which is the address of the element at the top of the stack, is stored in the stack pointer register. The stack has the three elements A, B, and C.
- The stack pointer holds C's address, which is 3. C is at the top of the stack.
- The top element is removed from the stack by reading the memory word at address 3 and decreasing the stack pointer by one.
- As a result, B is at the top of the stack, and the SP is aware of B's address, which is 2. It may add a new

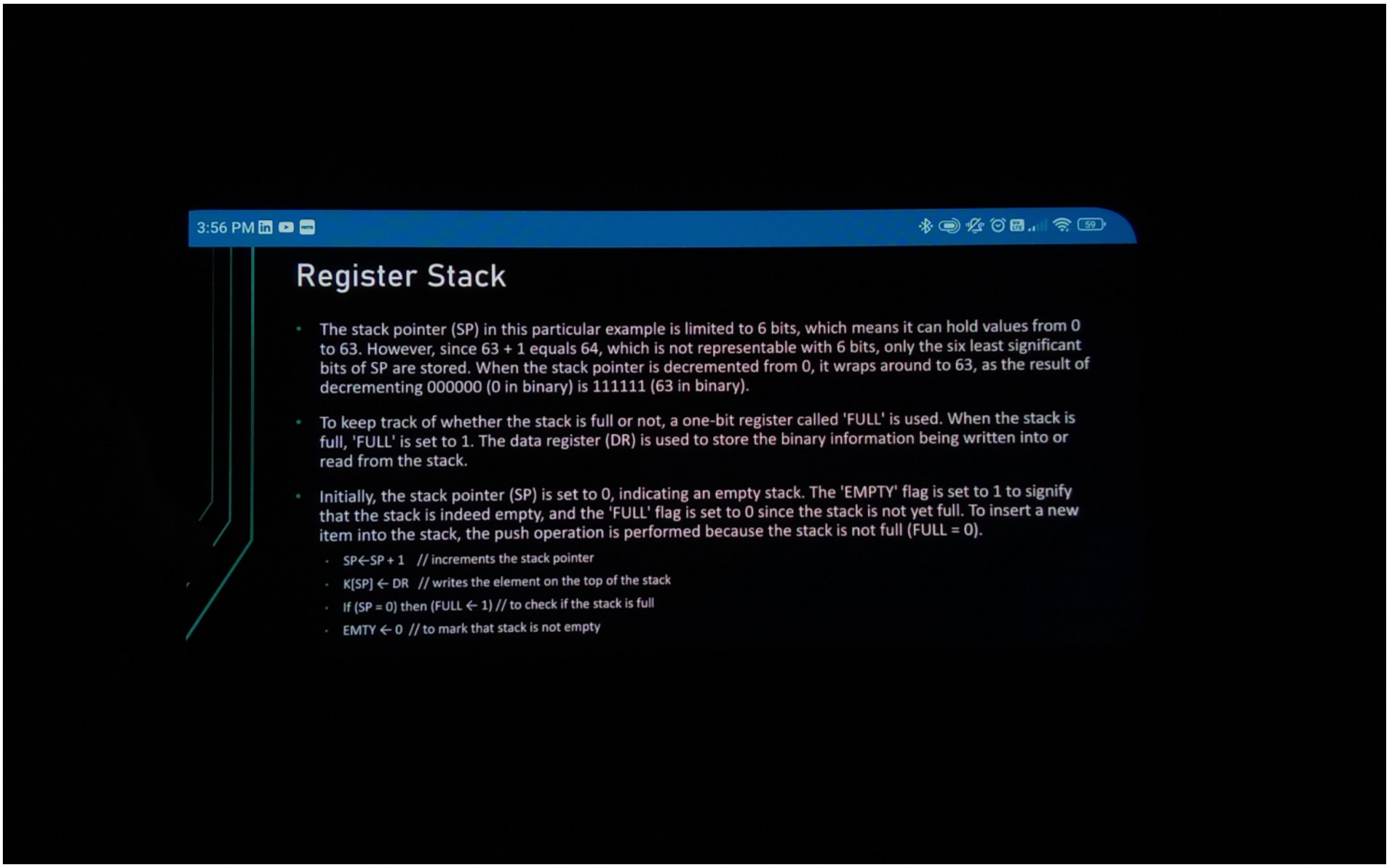


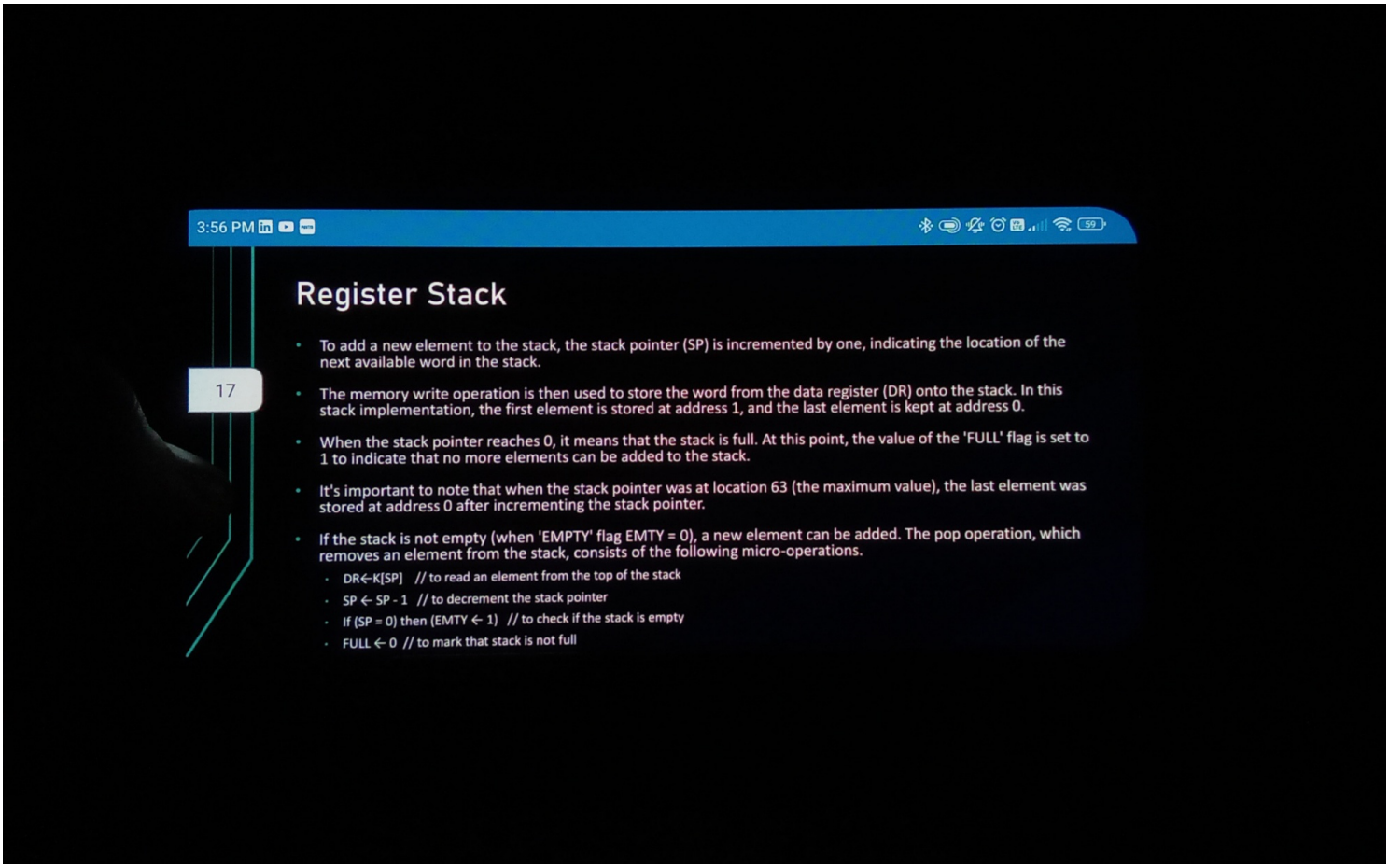
3:56 PM

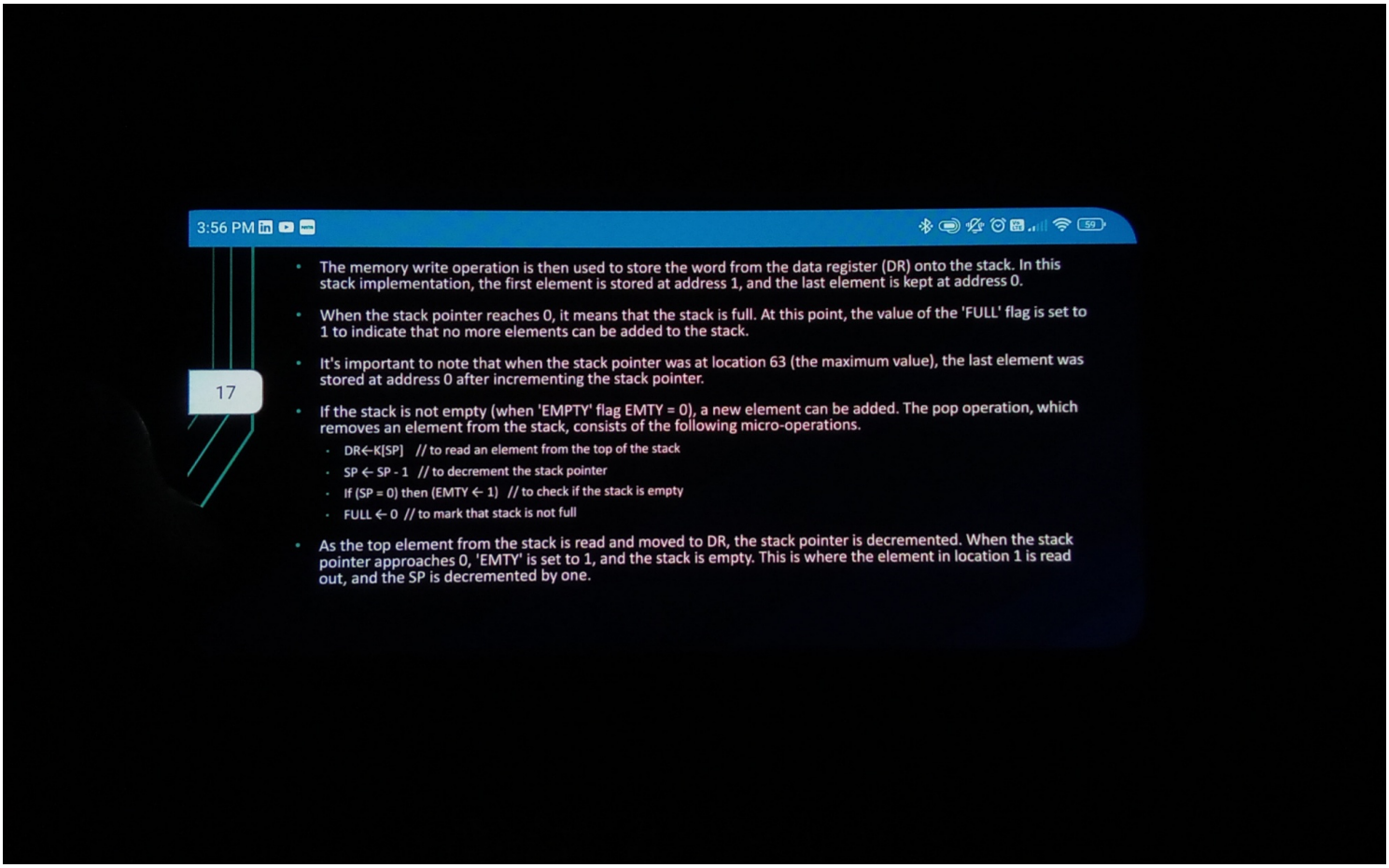
on top of each other. Consider a 64-word register stack like the one shown in the diagram.

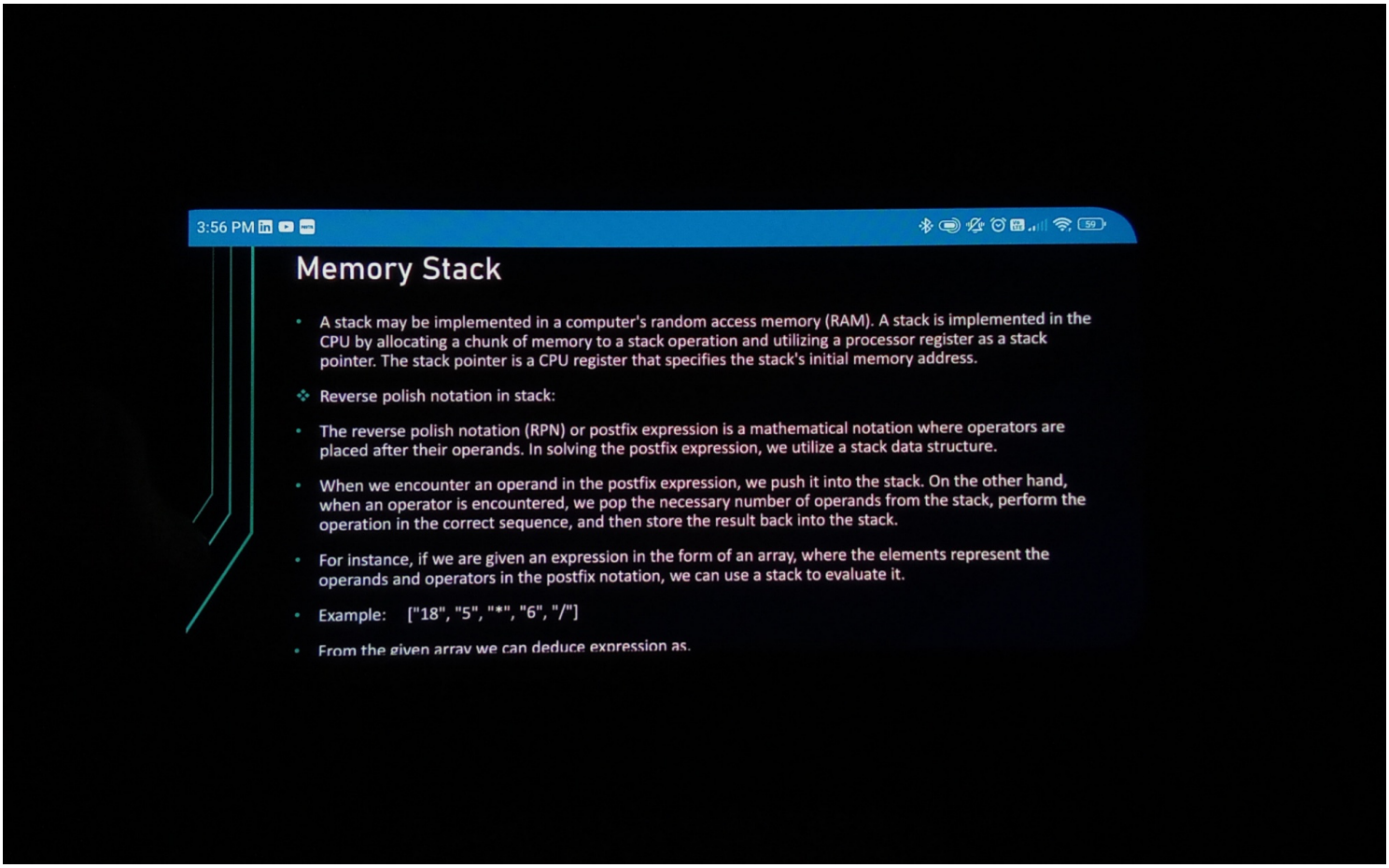
- A binary number, which is the address of the element at the top of the stack, is stored in the stack pointer register. The stack has the three elements A, B, and C.
- The stack pointer holds C's address, which is 3. C is at the top of the stack.
- The top element is removed from the stack by reading the memory word at address 3 and decreasing the stack pointer by one.
- As a result, B is at the top of the stack, and the SP is aware of B's address, which is 2. It may add a new word to the stack by increasing the stack pointer and inserting a word in the newly increased location.

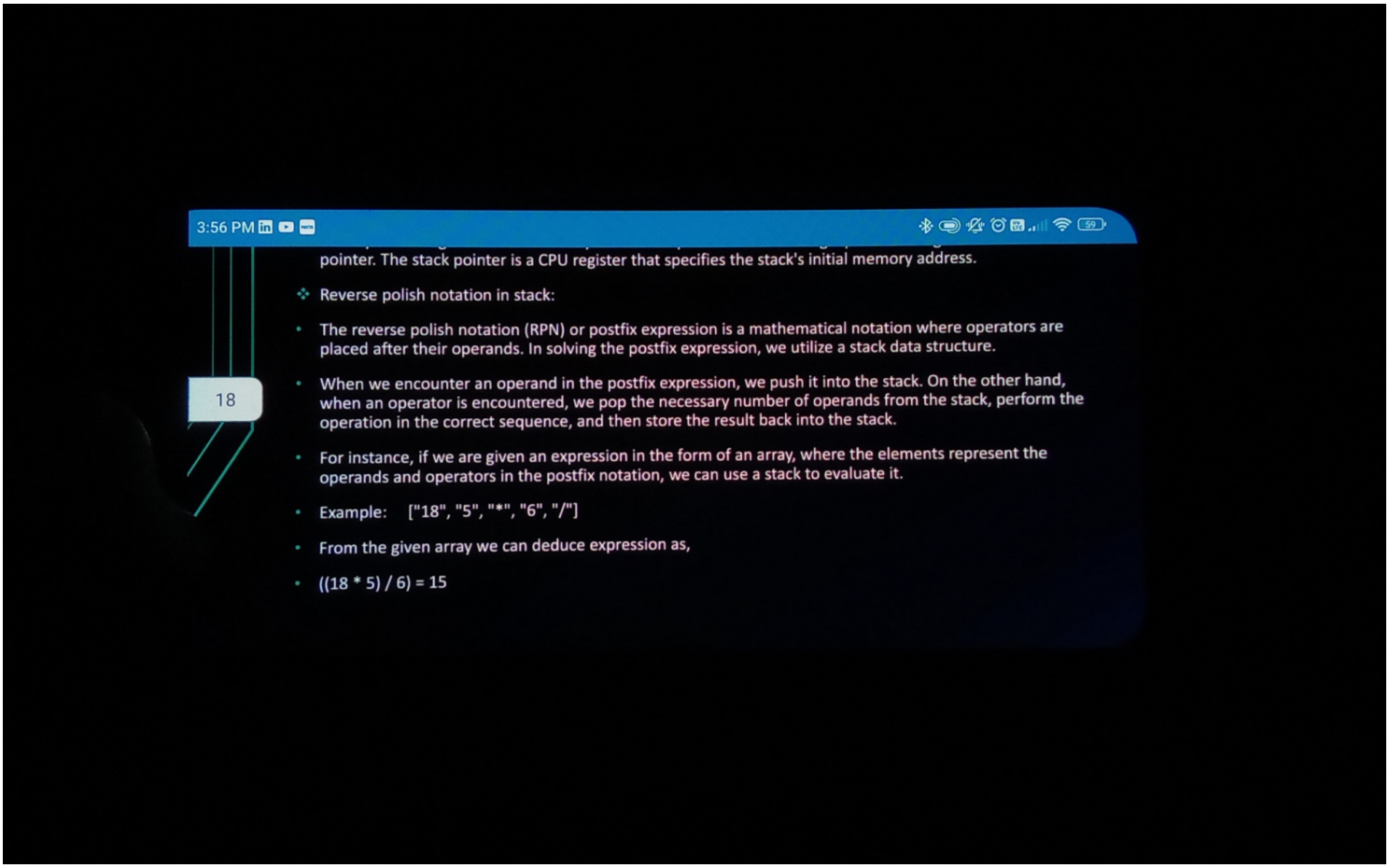


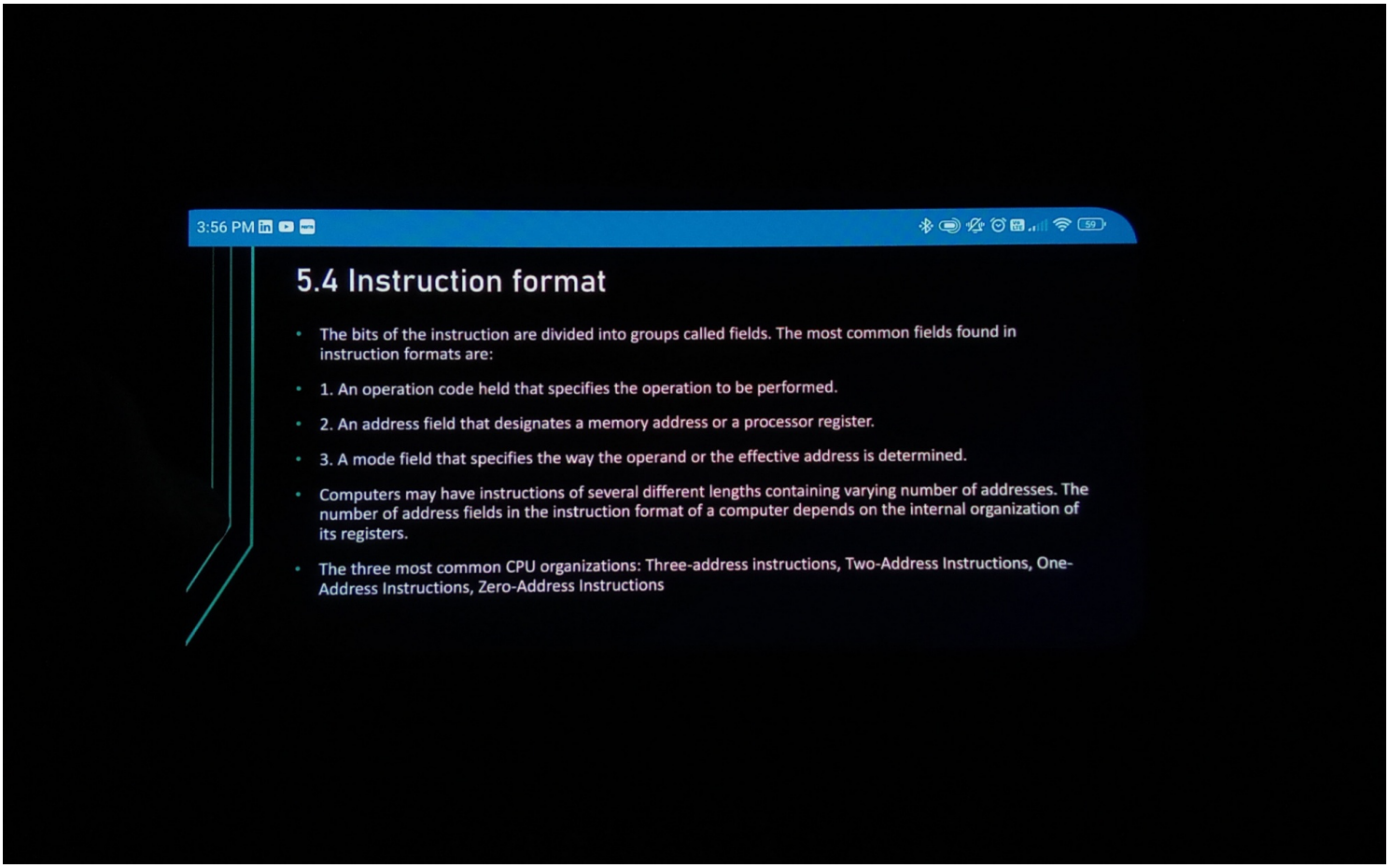

3:56 PM
Register Stack
The stack pointer (SP) in this particular example is limited to 6 bits, which means it can hold values from 0 to 63. However, since 63 + 1 equals 64, which is not representable with 6 bits, only the six least significant bits of SP are stored. When the stack pointer is decremented from 0, it wraps around to 63, as the result of decrementing 000000 (0 in binary) is 111111 (63 in binary).
To keep track of whether the stack is full or not, a one-bit register called 'FULL' is used. When the stack is full, 'FULL' is set to 1. The data register (DR) is used to store the binary information being written into or read from the stack.
Initially, the stack pointer (SP) is set to 0, indicating an empty stack. The 'EMPTY' flag is set to 1 to signify that the stack is indeed empty, and the 'FULL' flag is set to 0 since the stack is not yet full. To insert a new item into the stack, the push operation is performed because the stack is not full (FULL = 0).
SP←SP + 1 // increments the stack pointer
K[SP] ← DR // writes the element on the top of the stack
If (SP = 0) then (FULL ← 1) // to check if the stack is full
EMTY ← 0 // to mark that stack is not empty

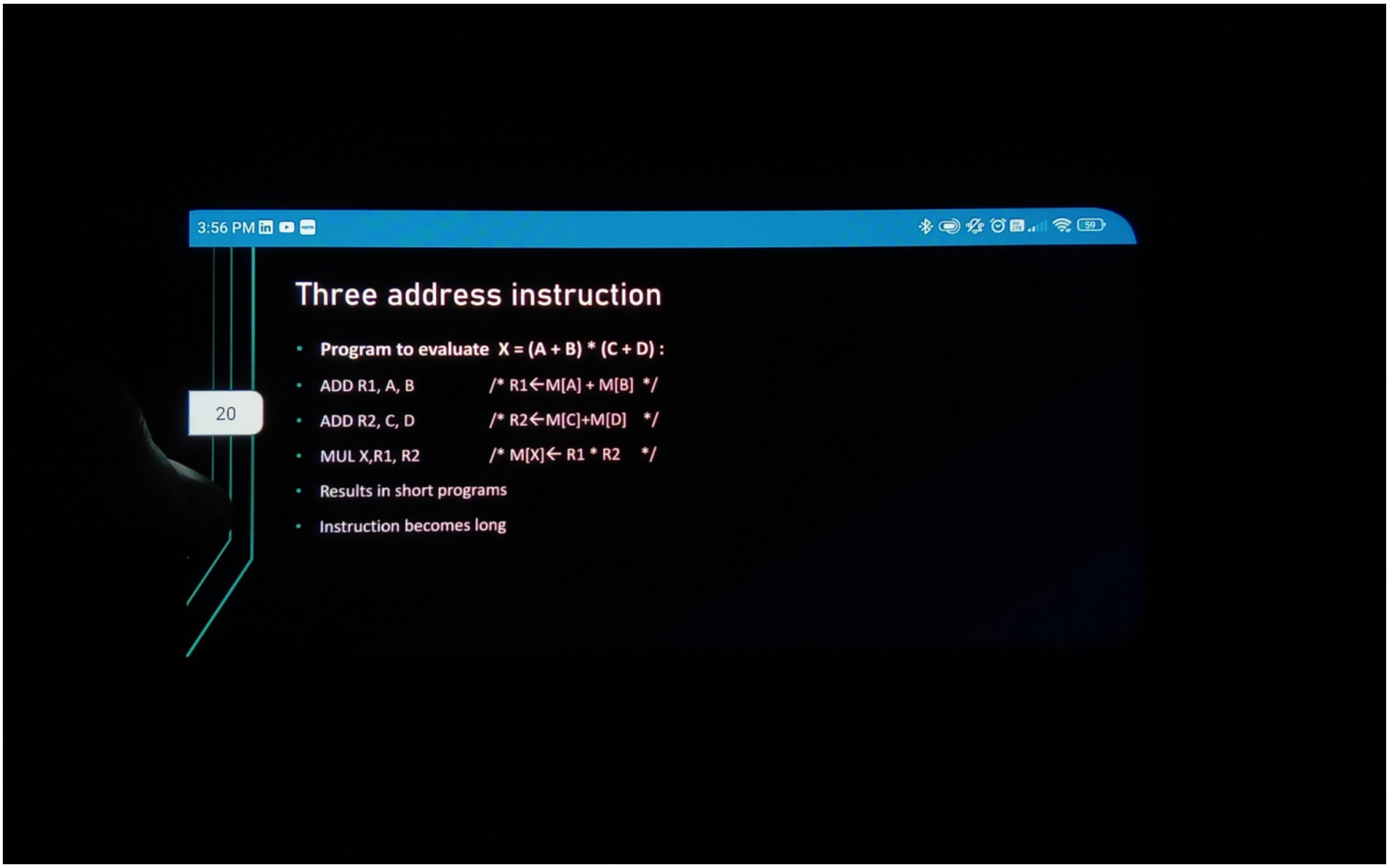

3:56 PM
17
Register Stack
To add a new element to the stack, the stack pointer (SP) is incremented by one, indicating the location of the next available word in the stack.
The memory write operation is then used to store the word from the data register (DR) onto the stack. In this stack implementation, the first element is stored at address 1, and the last element is kept at address 0.
When the stack pointer reaches 0, it means that the stack is full. At this point, the value of the 'FULL' flag is set to 1 to indicate that no more elements can be added to the stack.
It's important to note that when the stack pointer was at location 63 (the maximum value), the last element was stored at address 0 after incrementing the stack pointer.
If the stack is not empty (when 'EMPTY' flag EMTY = 0), a new element can be added. The pop operation, which removes an element from the stack, consists of the following micro-operations.
DR←K[SP] // to read an element from the top of the stack
SP ← SP - 1 // to decrement the stack pointer
If (SP = 0) then (EMTY ← 1) // to check if the stack is empty
FULL ← 0 // to mark that stack is not full

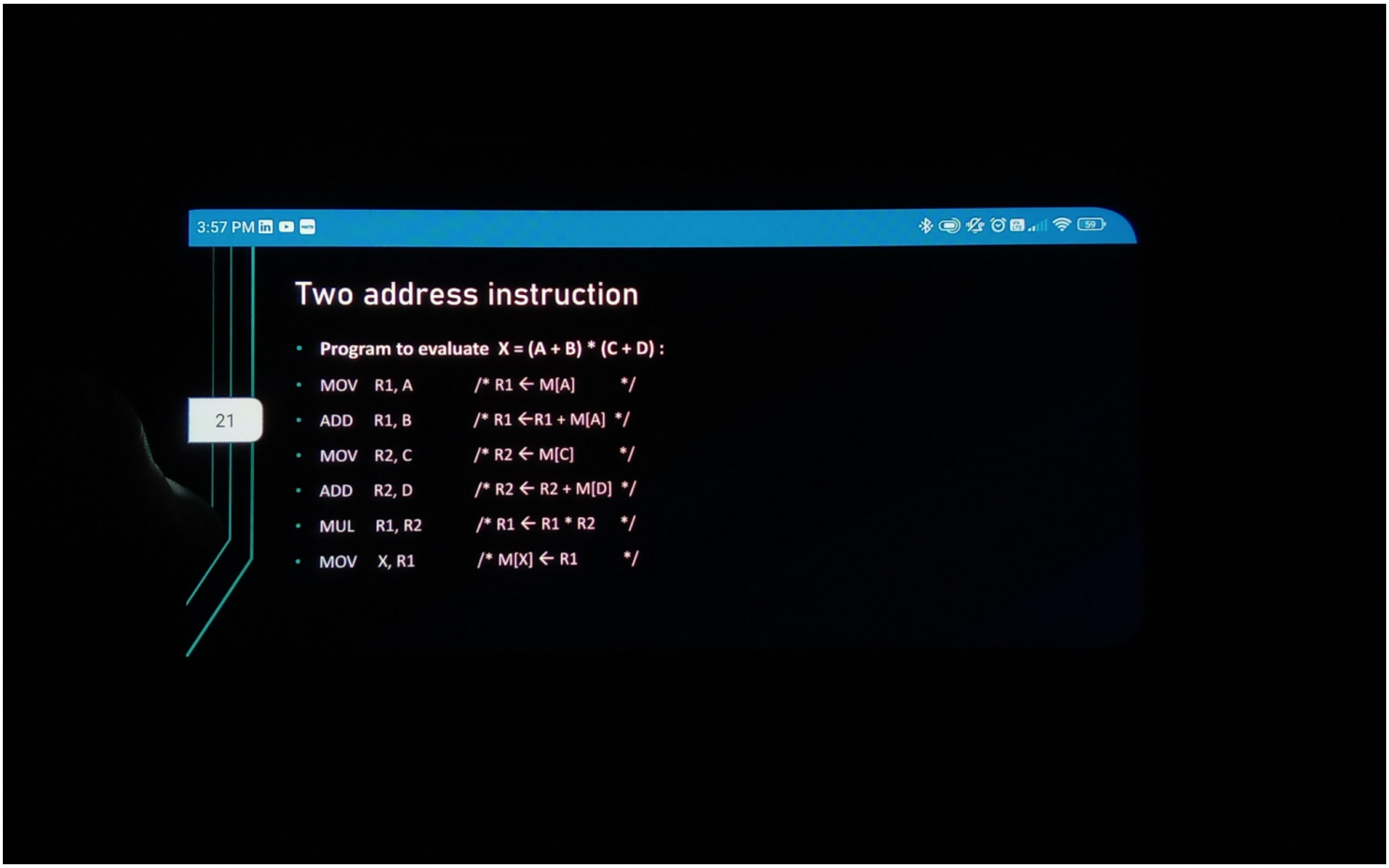

3:56 PM
17
The memory write operation is then used to store the word from the data register (DR) onto the stack. In this stack implementation, the first element is stored at address 1, and the last element is kept at address 0.
When the stack pointer reaches 0, it means that the stack is full. At this point, the value of the 'FULL' flag is set to 1 to indicate that no more elements can be added to the stack.
It's important to note that when the stack pointer was at location 63 (the maximum value), the last element was stored at address 0 after incrementing the stack pointer.
If the stack is not empty (when 'EMPTY' flag EMTY = 0), a new element can be added. The pop operation, which removes an element from the stack, consists of the following micro-operations.
DR←K[SP] // to read an element from the top of the stack
SP ← SP - 1 // to decrement the stack pointer
If (SP = 0) then (EMTY ← 1) // to check if the stack is empty
FULL ← 0 // to mark that stack is not full
As the top element from the stack is read and moved to DR, the stack pointer is decremented. When the stack pointer approaches 0, 'EMTY' is set to 1, and the stack is empty. This is where the element in location 1 is read out, and the SP is decremented by one.

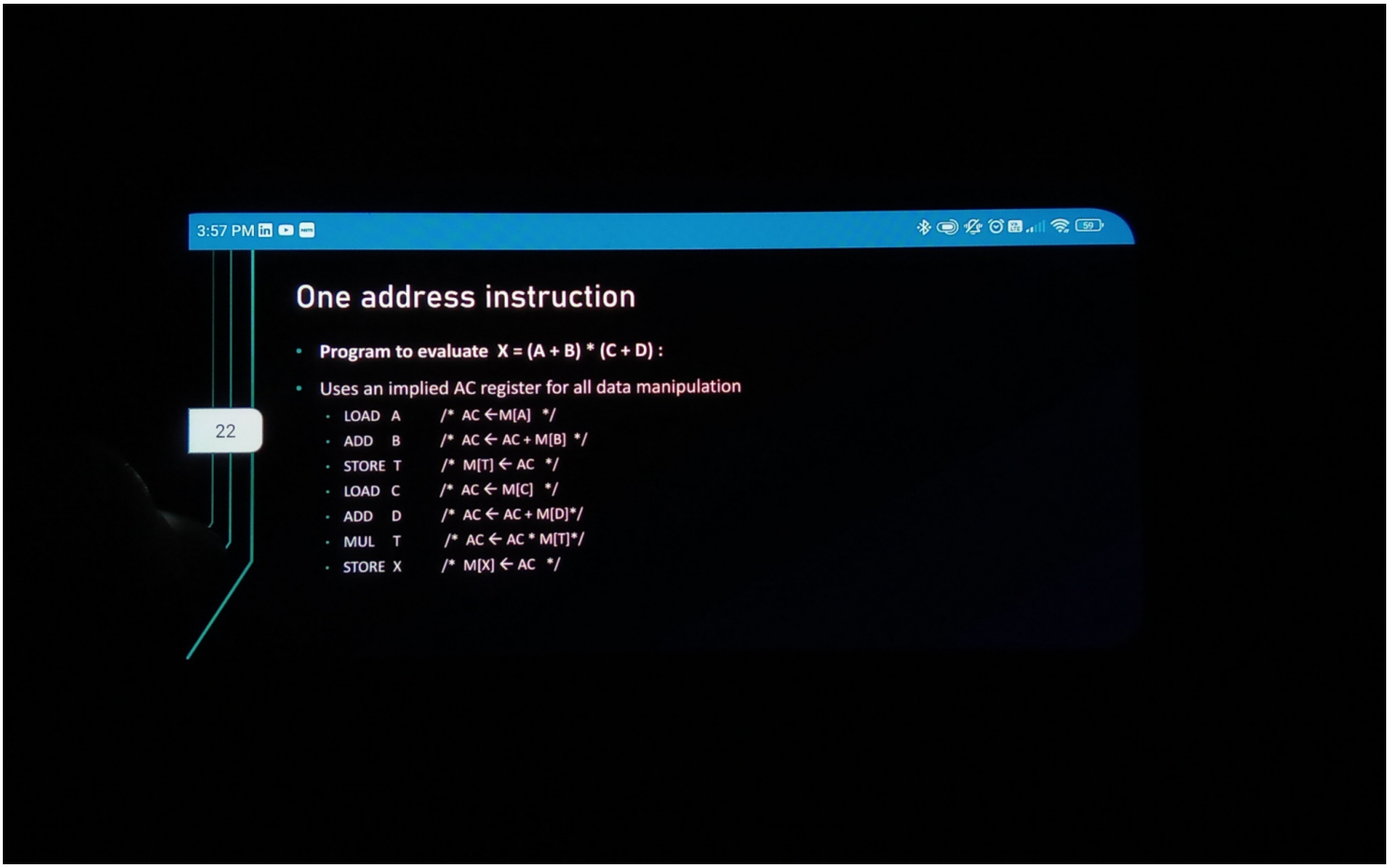

3:56 PM
Memory Stack
A stack may be implemented in a computer's random access memory (RAM). A stack is implemented in the CPU by allocating a chunk of memory to a stack operation and utilizing a processor register as a stack pointer. The stack pointer is a CPU register that specifies the stack's initial memory address.
Reverse polish notation in stack:
The reverse polish notation (RPN) or postfix expression is a mathematical notation where operators are placed after their operands. In solving the postfix expression, we utilize a stack data structure.
When we encounter an operand in the postfix expression, we push it into the stack. On the other hand, when an operator is encountered, we pop the necessary number of operands from the stack, perform the operation in the correct sequence, and then store the result back into the stack.
For instance, if we are given an expression in the form of an array, where the elements represent the operands and operators in the postfix notation, we can use a stack to evaluate it.
Example: ["18", "5", "*", "6", "/"]
From the given array we can deduce expression as.

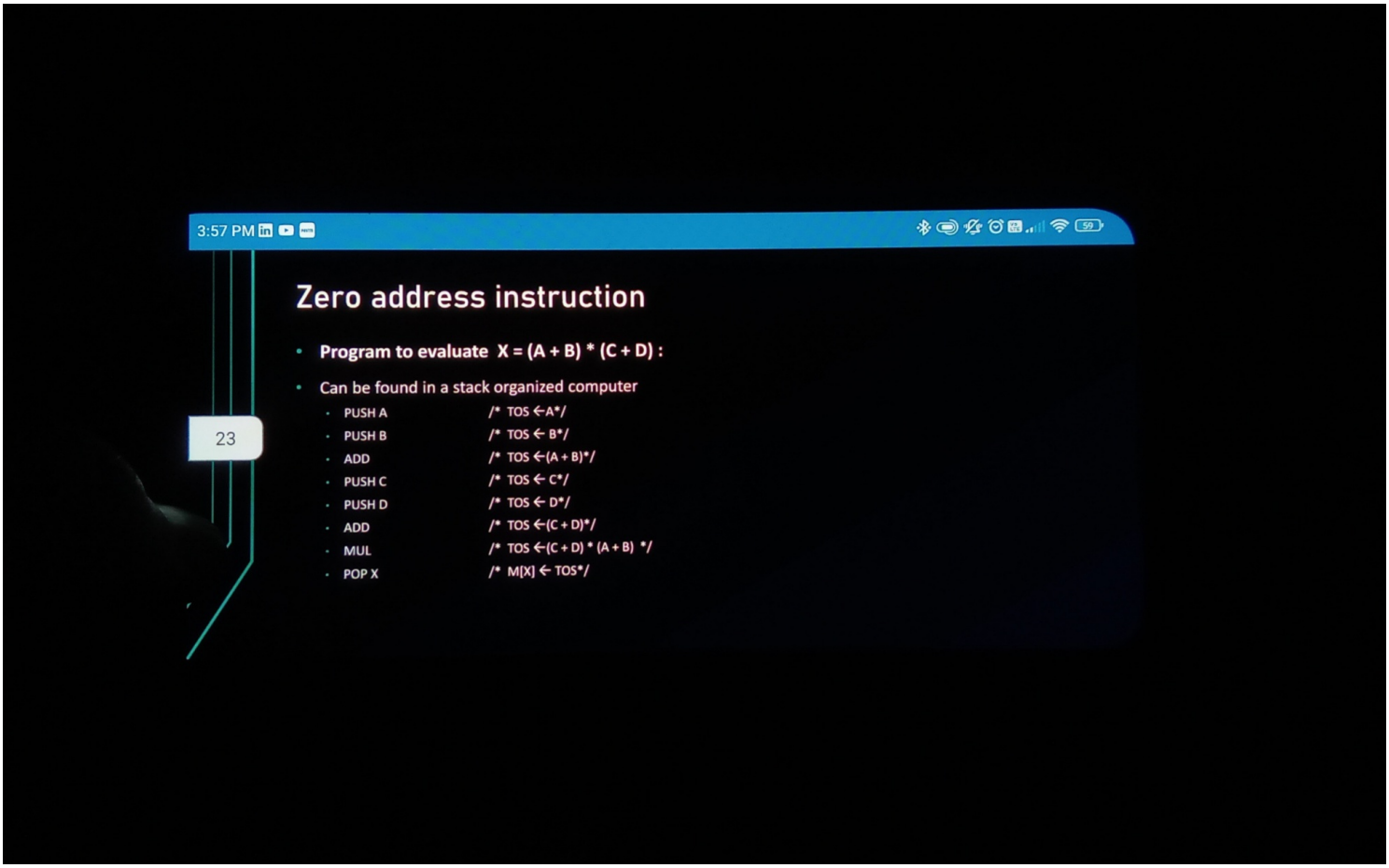

3:56 PM
pointer. The stack pointer is a CPU register that specifies the stack's initial memory address.
Reverse polish notation in stack:
The reverse polish notation (RPN) or postfix expression is a mathematical notation where operators are placed after their operands. In solving the postfix expression, we utilize a stack data structure.
18
When we encounter an operand in the postfix expression, we push it into the stack. On the other hand, when an operator is encountered, we pop the necessary number of operands from the stack, perform the operation in the correct sequence, and then store the result back into the stack.
For instance, if we are given an expression in the form of an array, where the elements represent the operands and operators in the postfix notation, we can use a stack to evaluate it.
Example: ["18", "5", "*", "6", "/"]
From the given array we can deduce expression as,
((18 * 5) / 6) = 15

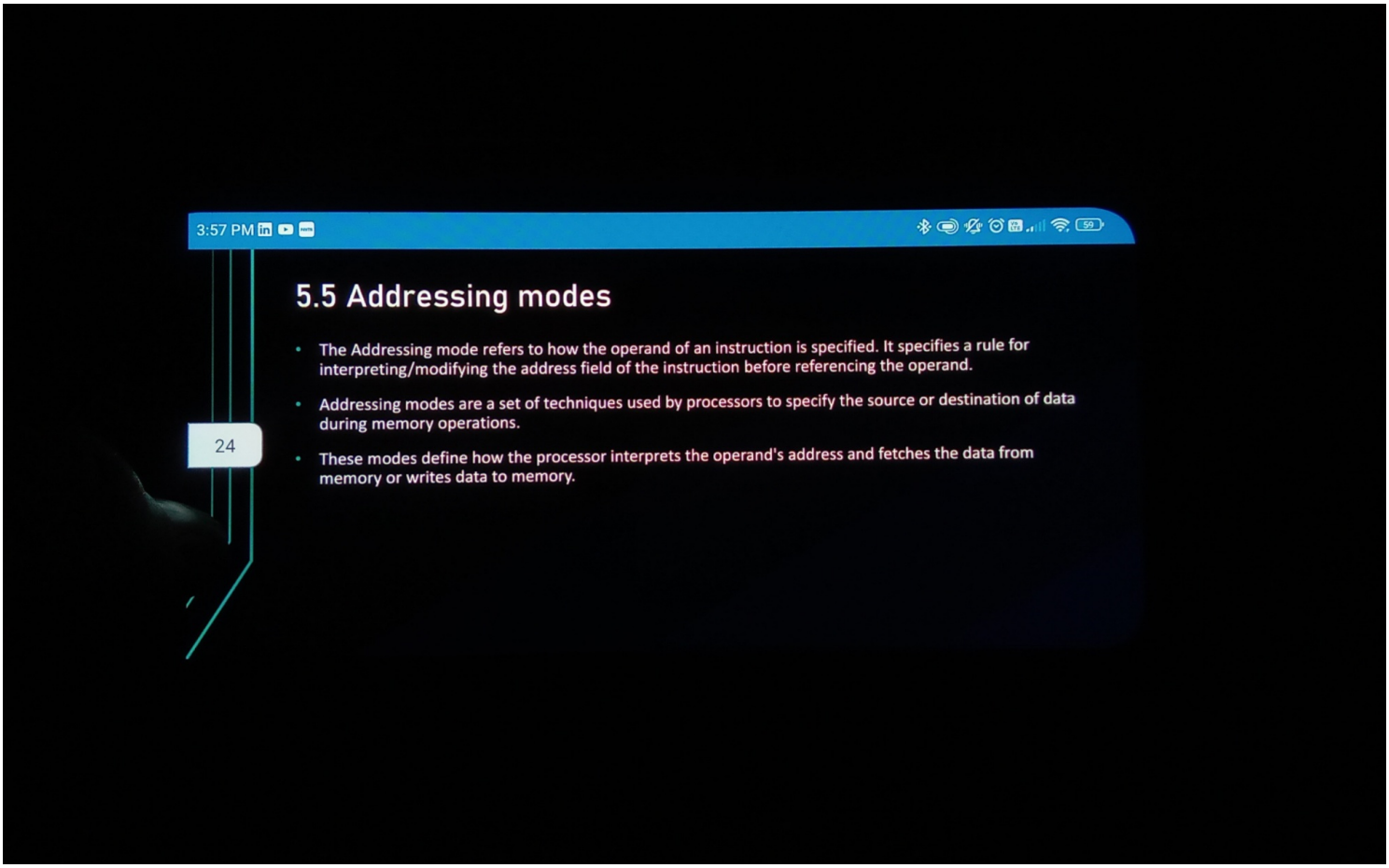

3:56 PM
5.4 Instruction format
The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:
1. An operation code held that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.
Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
The three most common CPU organizations: Three-address instructions, Two-Address Instructions, One-Address Instructions, Zero-Address Instructions

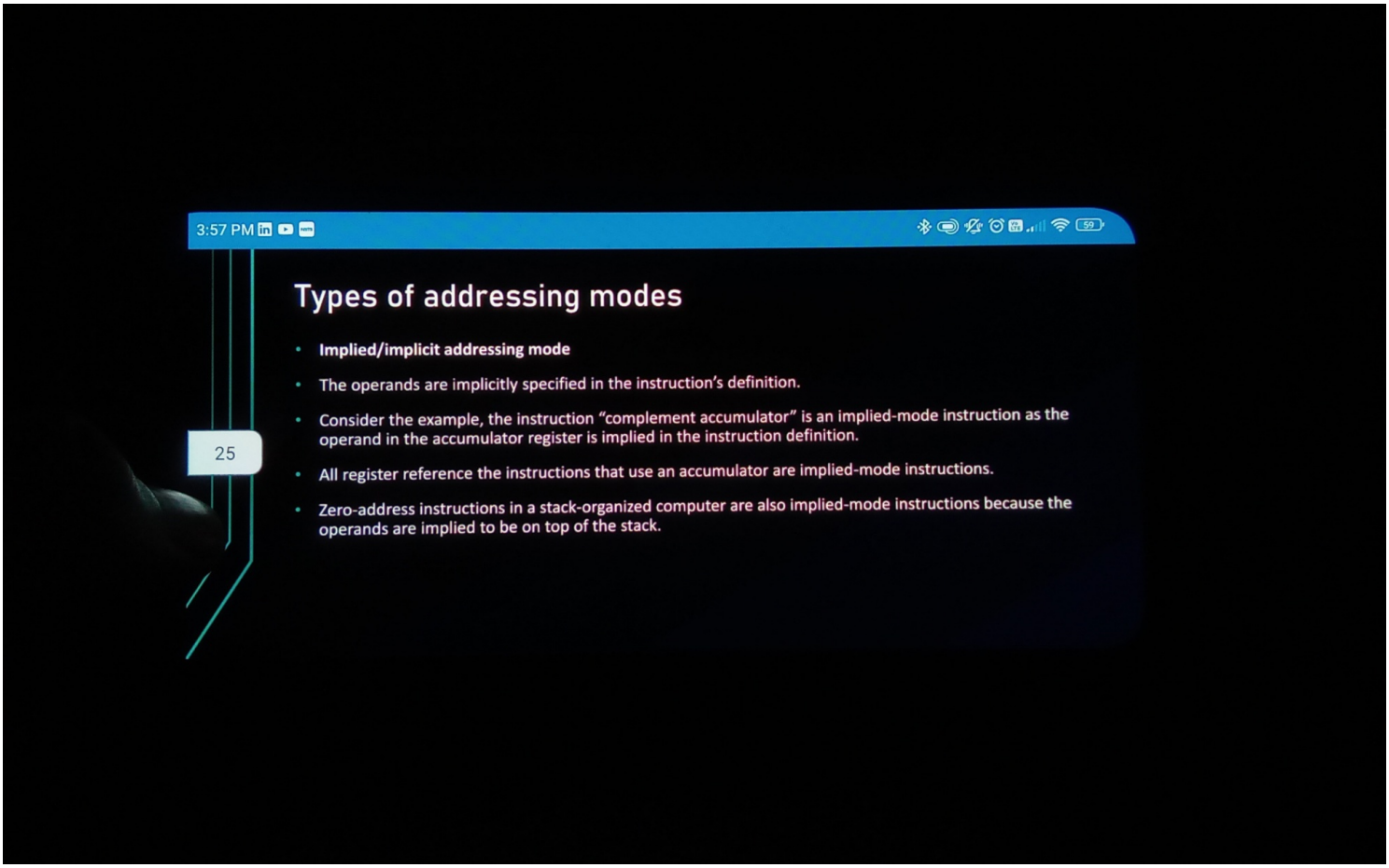

3:56 PM
Three address instruction
Program to evaluate X = (A + B) * (C + D) :
ADD R1, A, B /* R1←M[A] + M[B] */
ADD R2, C, D /* R2←M[C]+M[D] */
MUL X,R1, R2 /* M[X]← R1 * R2 */
Results in short programs
Instruction becomes long
20

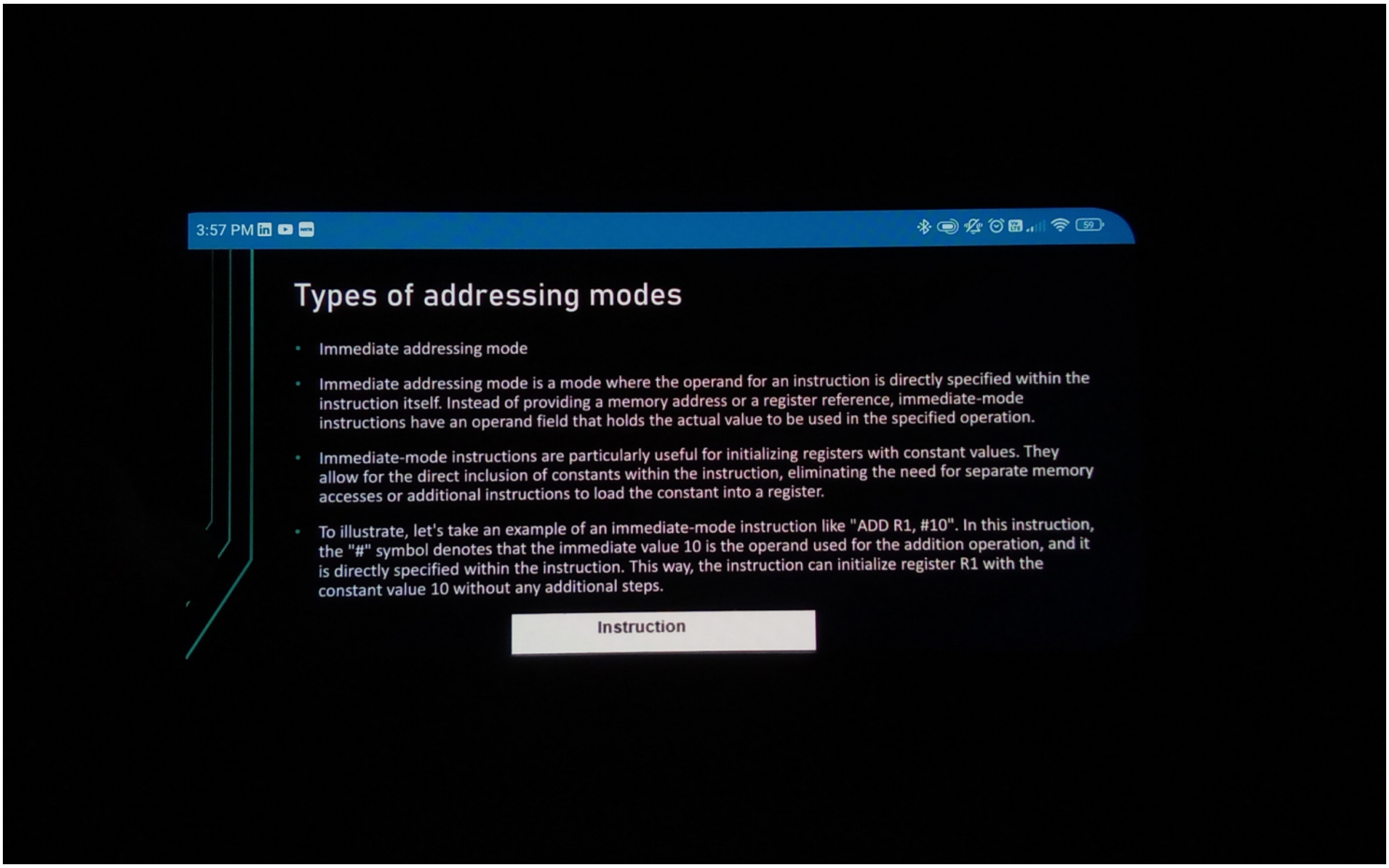

3:57 PM
Two address instruction
Program to evaluate X = (A + B) * (C + D) :
MOV R1, A /* R1 ← M[A] */
ADD R1, B /* R1 ←R1 + M[A] */
MOV R2, C /* R2 ← M[C] */
ADD R2, D /* R2 ← R2 + M[D] */
MUL R1, R2 /* R1 ← R1 * R2 */
MOV X, R1 /* M[X] ← R1 */
21

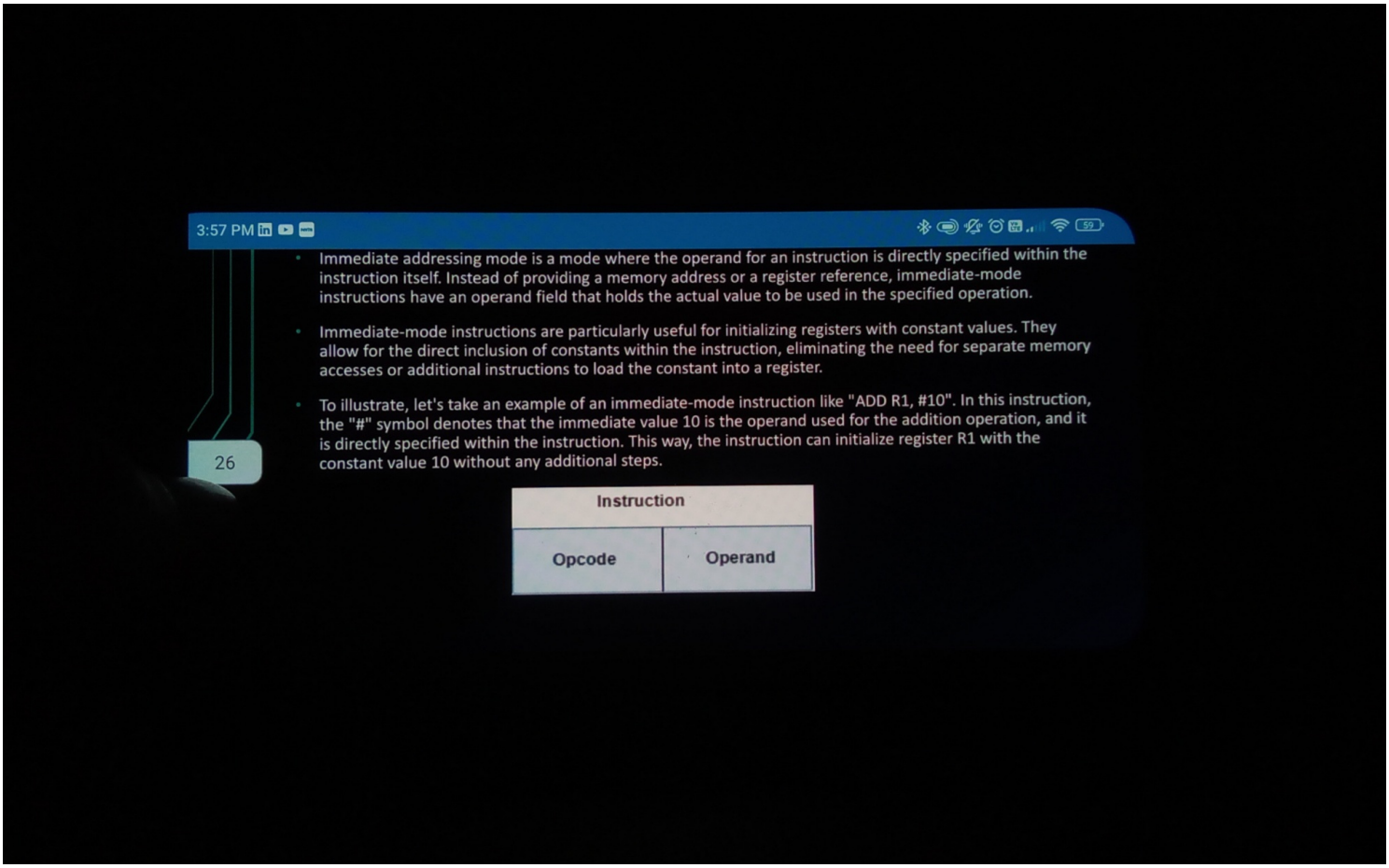

3:57 PM
One address instruction
Program to evaluate X = (A + B) * (C + D) :
Uses an implied AC register for all data manipulation
LOAD A /* AC ←M[A] */
ADD B /* AC ← AC + M[B] */
STORE T /* M[T] ← AC */
LOAD C /* AC ← M[C] */
ADD D /* AC ← AC + M[D]*/
MUL T /* AC ← AC * M[T]*/
STORE X /* M[X] ← AC */
22

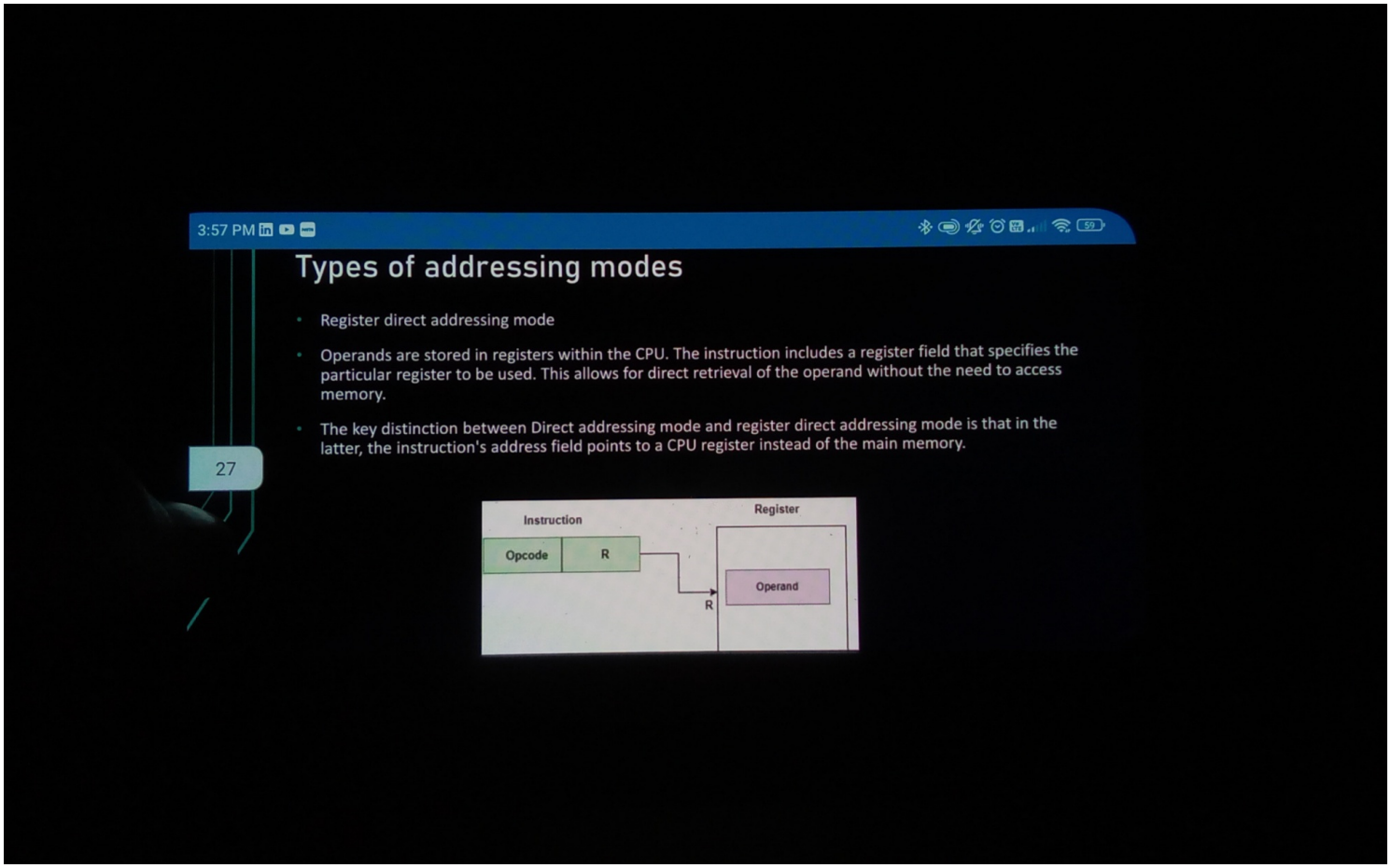

3:57 PM
Zero address instruction
Program to evaluate X = (A + B) * (C + D) :
Can be found in a stack organized computer
PUSH A /* TOS ←A*/
PUSH B /* TOS ← B*/
ADD /* TOS ←(A + B)*/
PUSH C /* TOS ← C*/
PUSH D /* TOS ← D*/
ADD /* TOS ←(C + D)*/
MUL /* TOS ←(C + D) * (A + B) */
POP X /* M[X] ← TOS*/
23


3:57 PM
5.5 Addressing modes
The Addressing mode refers to how the operand of an instruction is specified. It specifies a rule for interpreting/modifying the address field of the instruction before referencing the operand.
Addressing modes are a set of techniques used by processors to specify the source or destination of data during memory operations.
These modes define how the processor interprets the operand's address and fetches the data from memory or writes data to memory.
24


3:57 PM
Types of addressing modes
Implied/implicit addressing mode
The operands are implicitly specified in the instruction's definition.
Consider the example, the instruction "complement accumulator" is an implied-mode instruction as the operand in the accumulator register is implied in the instruction definition.
All register reference the instructions that use an accumulator are implied-mode instructions.
Zero-address instructions in a stack-organized computer are also implied-mode instructions because the operands are implied to be on top of the stack.
25

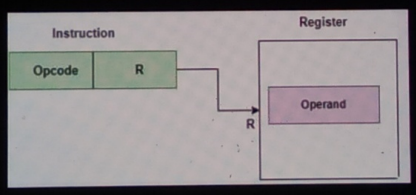

3:57 PM
Types of addressing modes
Immediate addressing mode
Immediate addressing mode is a mode where the operand for an instruction is directly specified within the instruction itself. Instead of providing a memory address or a register reference, immediate-mode instructions have an operand field that holds the actual value to be used in the specified operation.
Immediate-mode instructions are particularly useful for initializing registers with constant values. They allow for the direct inclusion of constants within the instruction, eliminating the need for separate memory accesses or additional instructions to load the constant into a register.
To illustrate, let's take an example of an immediate-mode instruction like "ADD R1, #10". In this instruction, the "#" symbol denotes that the immediate value 10 is the operand used for the addition operation, and it is directly specified within the instruction. This way, the instruction can initialize register R1 with the constant value 10 without any additional steps.
Instruction


3:57 PM
26
Immediate addressing mode is a mode where the operand for an instruction is directly specified within the instruction itself. Instead of providing a memory address or a register reference, immediate-mode instructions have an operand field that holds the actual value to be used in the specified operation.
Immediate-mode instructions are particularly useful for initializing registers with constant values. They allow for the direct inclusion of constants within the instruction, eliminating the need for separate memory accesses or additional instructions to load the constant into a register.
To illustrate, let's take an example of an immediate-mode instruction like "ADD R1, #10". In this instruction, the "#" symbol denotes that the immediate value 10 is the operand used for the addition operation, and it is directly specified within the instruction. This way, the instruction can initialize register R1 with the constant value 10 without any additional steps.
Instruction
Opcode
Operand


3:57 PM
Types of addressing modes
Register direct addressing mode
Operands are stored in registers within the CPU. The instruction includes a register field that specifies the particular register to be used. This allows for direct retrieval of the operand without the need to access memory.
The key distinction between Direct addressing mode and register direct addressing mode is that in the latter, the instruction's address field points to a CPU register instead of the main memory.
27
Instruction
Register
Opcode
R
Operand
R

3:57 PM

- Operands are stored in registers within the CPU. The instruction includes a register field that specifies the particular register to be used. This allows for direct retrieval of the operand without the need to access memory.
- The key distinction between Direct addressing mode and register direct addressing mode is that in the latter, the instruction's address field points to a CPU register instead of the main memory.



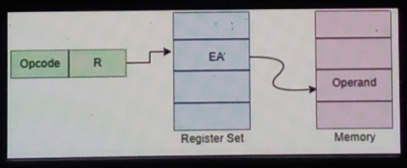
27

3:57 PM

Types of addressing modes

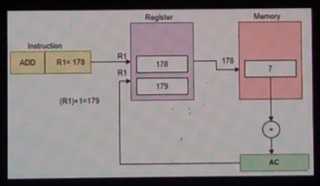
- Register indirect addressing mode
- Indirect addressing mode is a mode where an instruction specifies a CPU register to store the memory address of the operand.
- This mode requires only one memory reference to fetch the operand. Instead of directly containing the operand, the specified register holds the address of the operand in memory.
- The only distinction between Indirect addressing mode and register indirect addressing mode is that in the latter, the instruction's address field refers to a CPU register.

28



3:58 PM

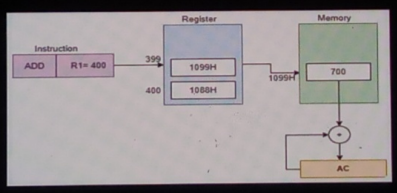
- Auto increment addressing mode
- This mode is a special Register Indirect Addressing Mode case. The register is incremented/decremented after or before its value is utilized.
- EA = content of the register.
- The content of the register is increment automatically by step size 'd' after accessing the operand, where the step size 'd' depends on the size of the accessed operand. Only one reference memory is required to fetch the operand.
- Example:



- Here, the Effective Address (R) =178, and the operand in AC are 7. After loading R1 is incremented by 1 and becomes 179.

3:58 PM

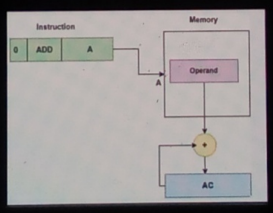
- Auto decrement addressing mode
- This mode is a specific case of Register Indirect Addressing Mode. In this case, the effective address (EA) is calculated by subtracting a step size ('d') from the content of the register.
- After accessing the operand, the content of the register is decremented by the step size 'd', which depends on the size of the operand being accessed. This mode requires only one memory reference to fetch the operand.
- Example:



- Here, register R1 is decremented by 1 (400-1=399) prior to the instruction execution which implies that the operand loaded to the AC is of address 1099H instead of 1088H. Hence, the Effective Address is 1099H.

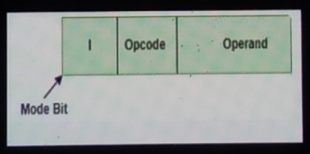
3:58 PM

- Direct addressing mode
- The effective address of the operand is directly stored in the address field of the instruction.
- The operand itself is located in memory, and the address field in the instruction specifies the memory address of the operand. This mode, known as absolute addressing mode, requires only one memory reference to fetch the operand, without any additional calculations needed to determine the effective address.
- For instance, consider the instruction "ADD X." In this case, the value stored in the accumulator is incremented by the value stored at the memory address X. The address field in the instruction directly points to the memory location where the operand resides, enabling the addition operation to be performed without any further computations.



3:58 PM

- Indirect addressing mode
- In this addressing mode, the address field of the instruction points to a memory location that holds the effective address of the operand.
- To fetch the operand, two memory references are needed. Initially, the control unit retrieves the instruction from memory, and then it uses the address part of the instruction to access the memory location that stores the effective address. T
- his mode of addressing slows down the execution process because it requires multiple memory lookups to obtain the operand.

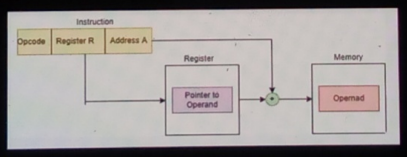


3:58 PM

Types of addressing modes

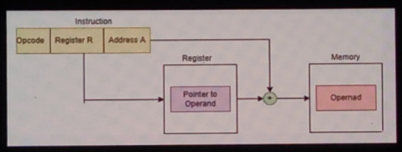
- Displacement addressing mode
- In this addressing mode, the content of an indexed register is added to the address part of the instruction. This addition operation yields the effective address of the operand.
- By combining the indexed register content with the instruction's address part, the processor determines the memory location where the operand can be found. This allows for efficient retrieval of the operand by calculating the effective address using the indexed register.

33



3:58 PM

- By combining the indexed register content with the instruction's address part, the processor determines the memory location where the operand can be found. This allows for efficient retrieval of the operand by calculating the effective address using the indexed register.



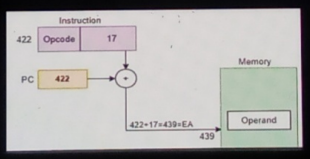
33

- EA = A + (R)
- Here, the address field holds two values, A: Base value R: displacement value.

Types of addressing modes

3:58 PM

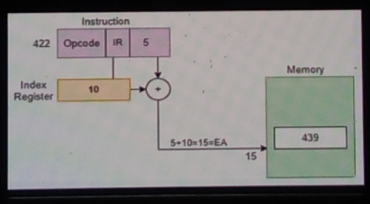
- Relative addressing mode
- It is like the another version of the displacement mode
- In this addressing mode, the effective address (EA) is obtained by adding the content of the program counter (PC) to the address part of the instruction. The equation EA = A + (PC) represents this calculation, where EA represents the effective address and PC represents the program counter.
- Typically, the address part of the instruction is a signed number that can be positive or negative. Once the instruction's address is fetched, the value of the program counter immediately increases, regardless of whether the fetched instruction has been executed or not. The program counter holds the address of the next instruction to be executed in the program sequence.
- The program counter contains the number 422, and the address part of the instruction contains the number 17. The instruction at location 421 is read during the fetch phase, and the program counter is then incremented by one to 422 + 17 = 439.



3:58 PM

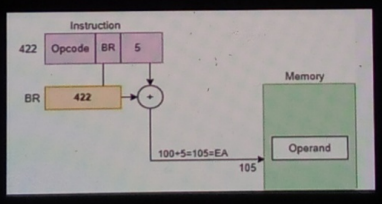
Types of addressing modes

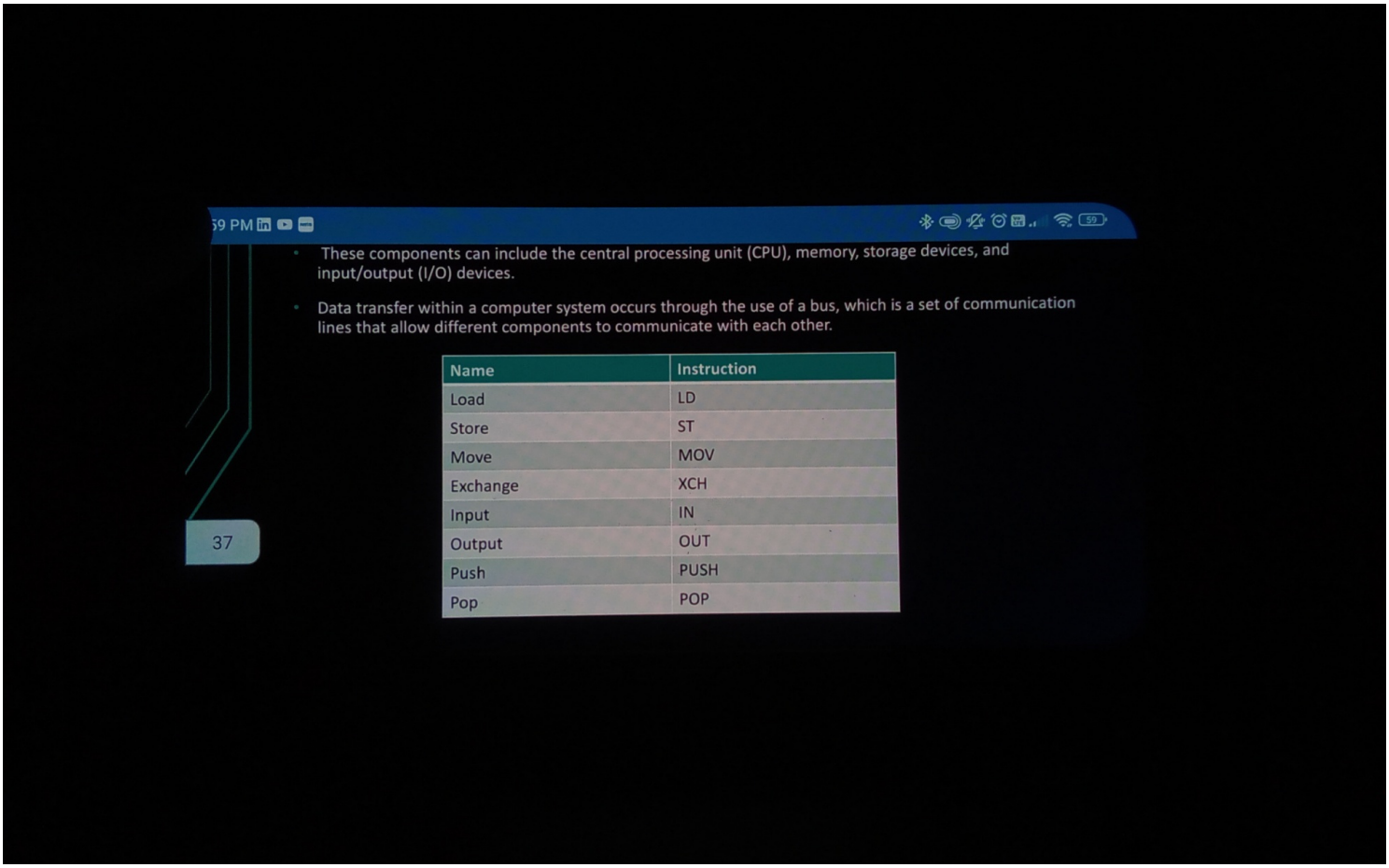
- Indexed addressing mode
- In this addressing mode, the effective address (EA) is obtained by adding the content of an index register to the address part of the instruction. The equation EA = content of index register + Instruction address part represents this calculation.
- By adding the index register's content to the address part of the instruction, the processor determines the effective address where the operand can be found. This allows for efficient memory access by incorporating the index register value into the address calculation.

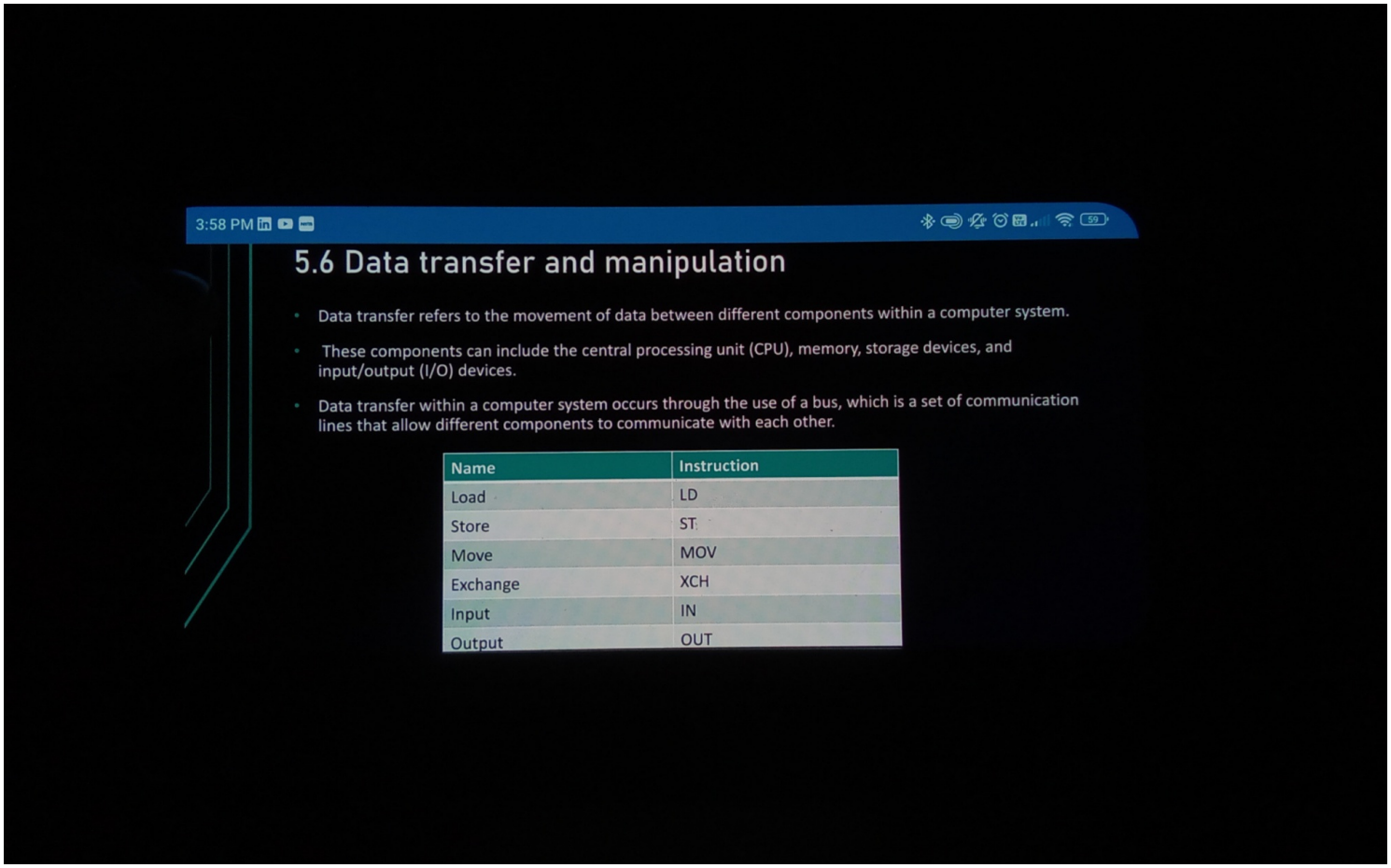


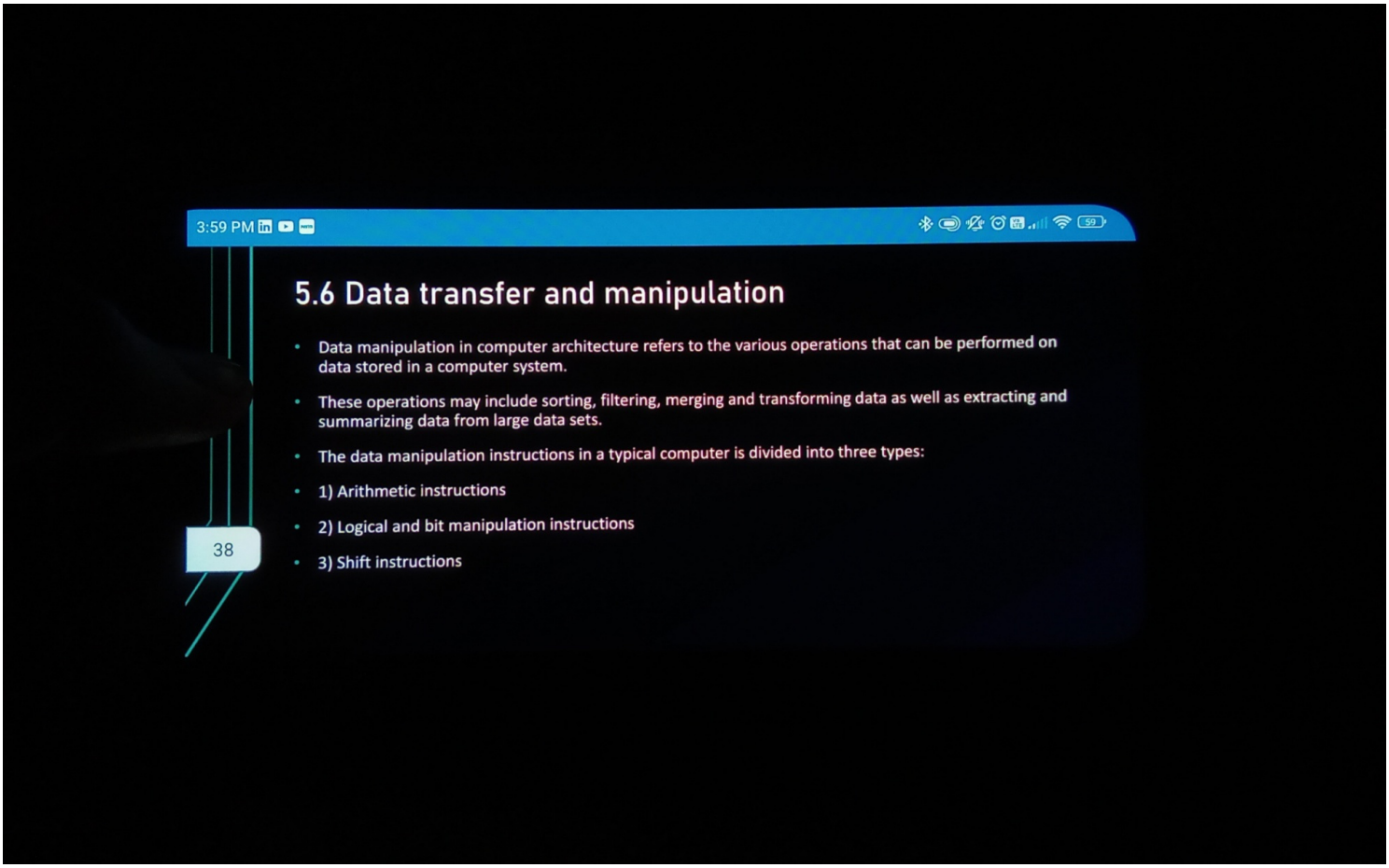
3:58 PM

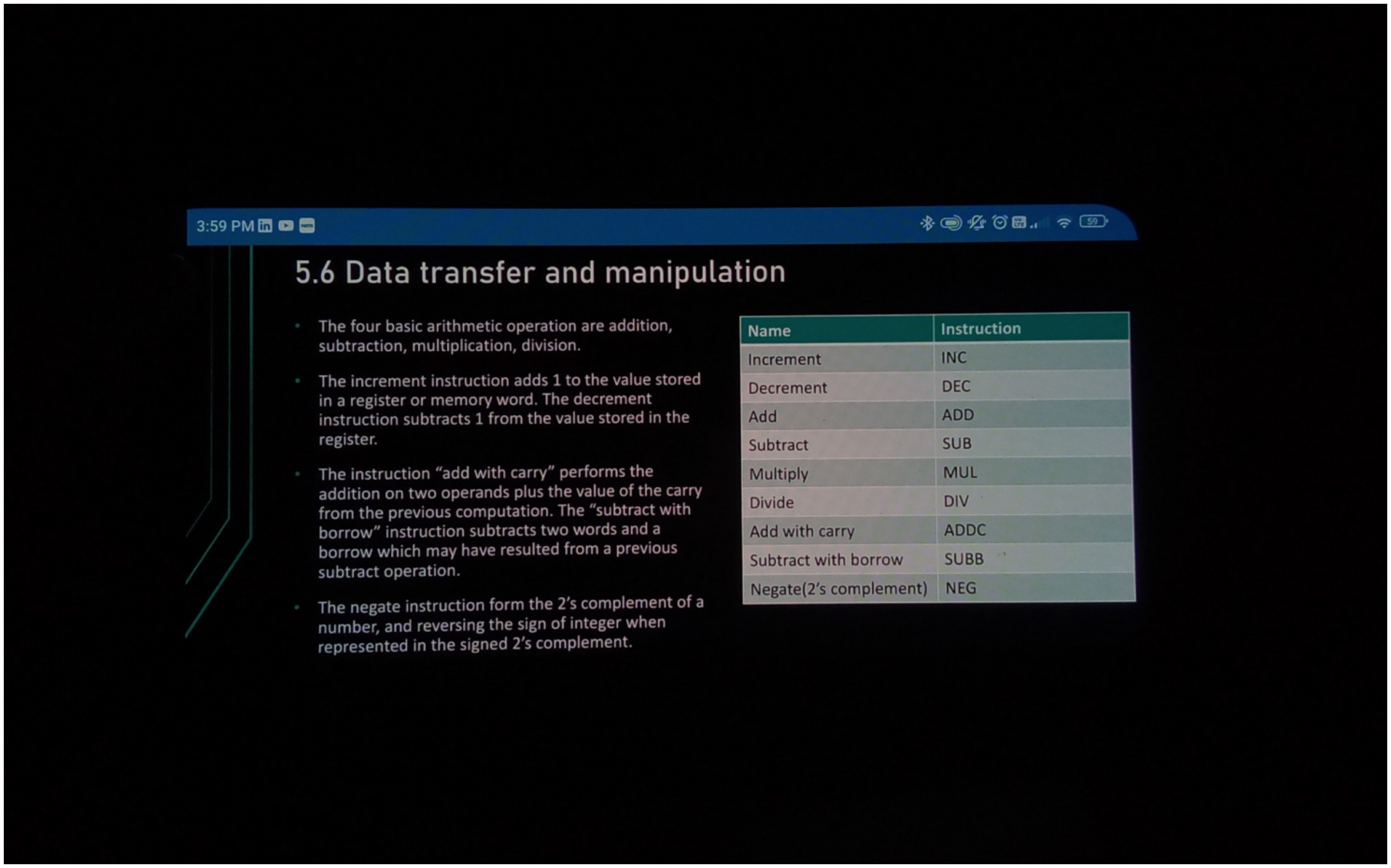
- Base register addressing mode
- In this addressing mode, known as displacement address mode, the effective address (EA) is obtained by adding the content of a base register to the address part of the instruction. The equation EA = A + (R) represents this calculation.
- Here, A represents the displacement value and R represents the pointer to the base address. By adding the base register's content to the instruction's address, the processor determines the effective address where the operand is located. This mode allows for flexible memory access by incorporating a displacement value with the base address pointed to by the base register.

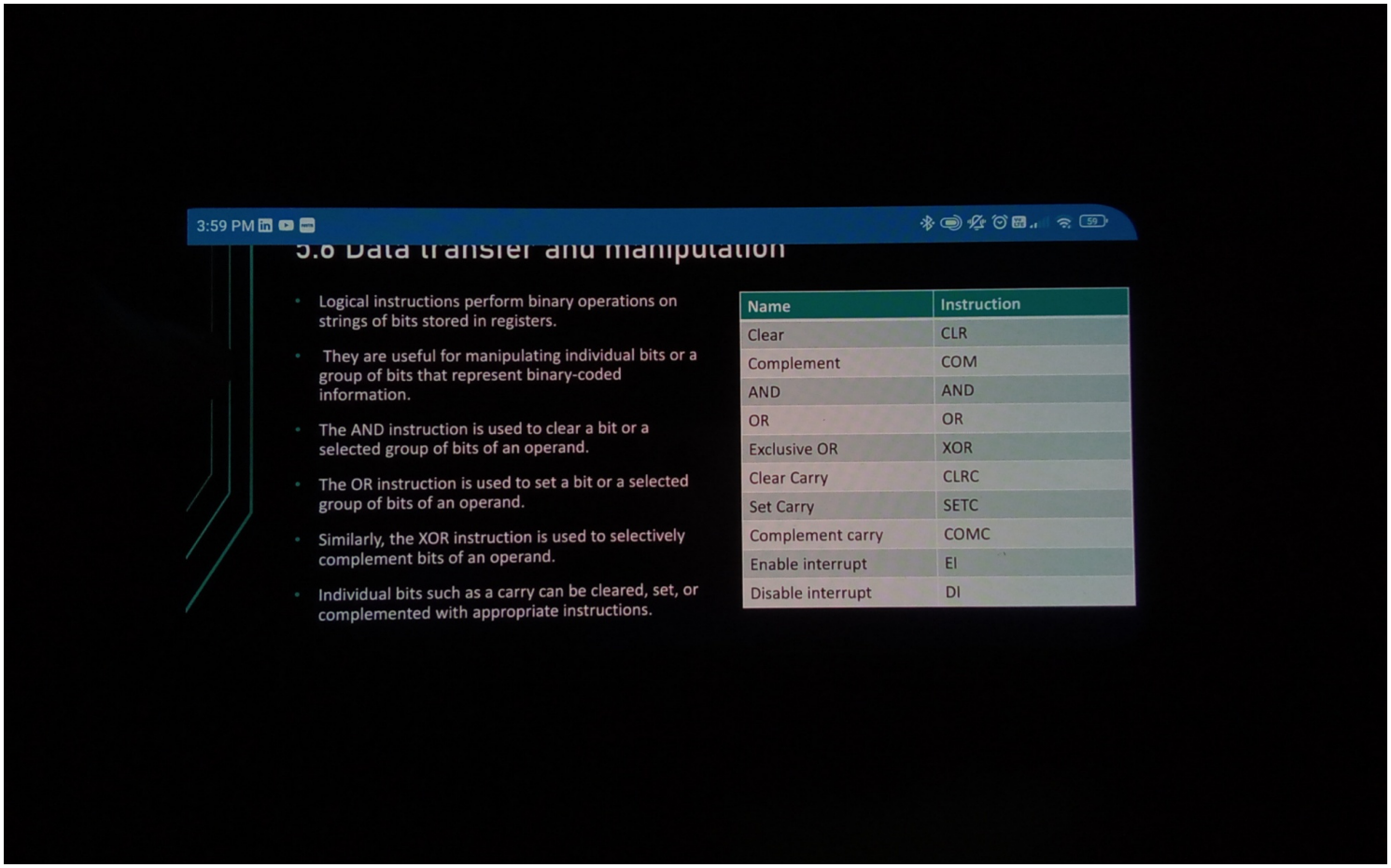


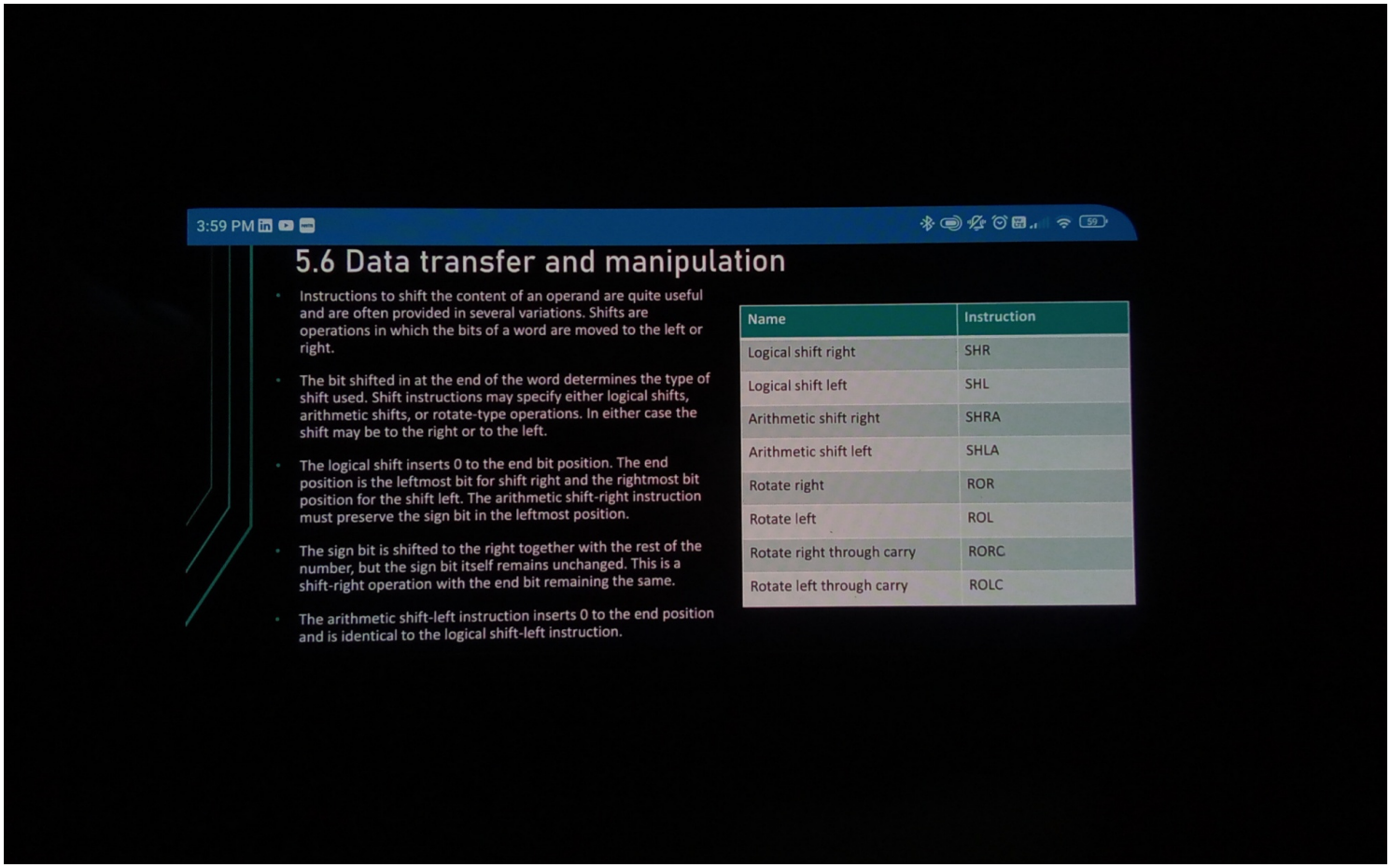

9 PM
These components can include the central processing unit (CPU), memory, storage devices, and input/output (I/O) devices.
Data transfer within a computer system occurs through the use of a bus, which is a set of communication lines that allow different components to communicate with each other.
Name
Instruction
Load
LD
Store
ST
Move
MOV
Exchange
XCH
Input
IN
Output
OUT
Push
PUSH
Pop
POP
37

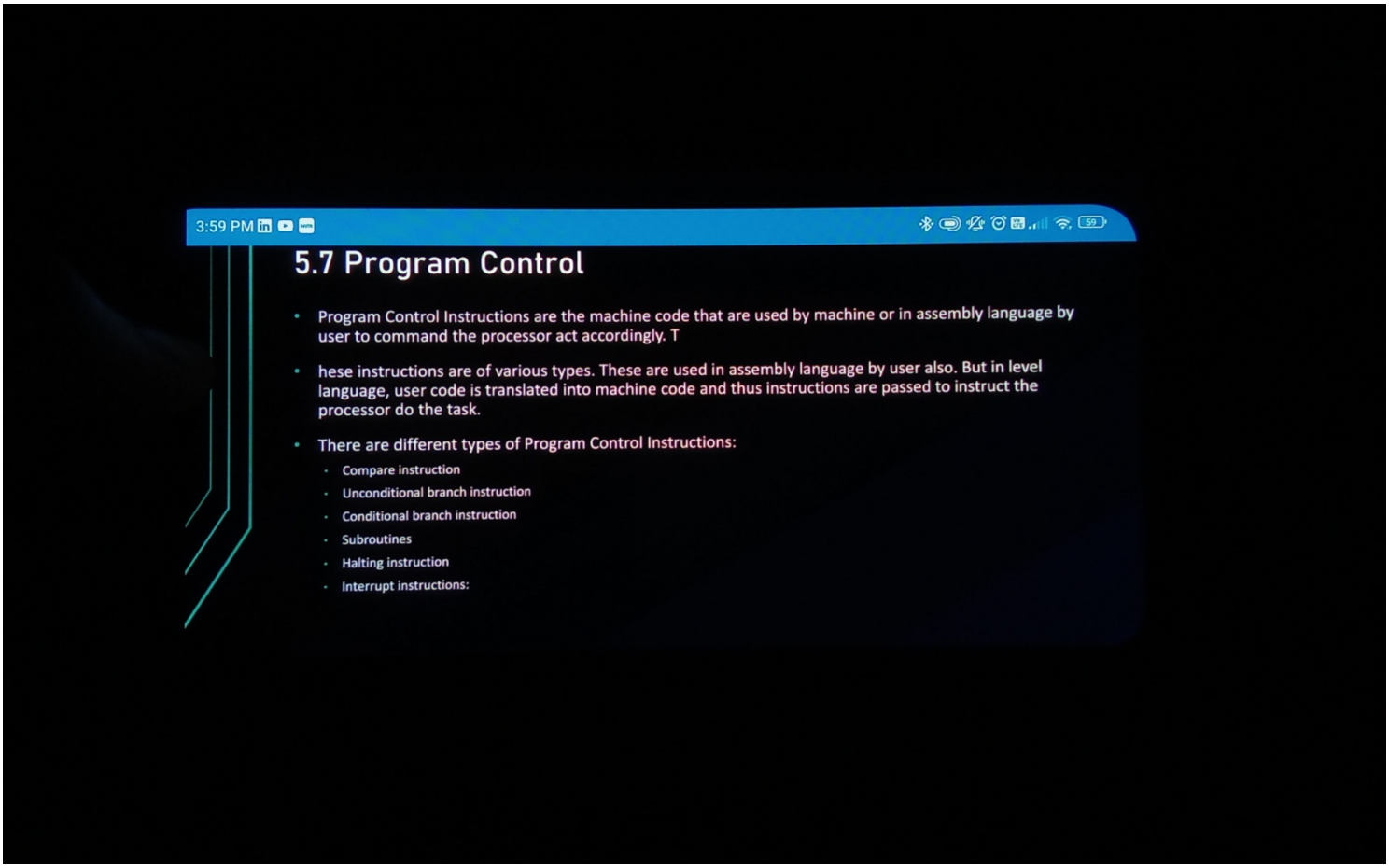

3:58 PM
5.6 Data transfer and manipulation
Data transfer refers to the movement of data between different components within a computer system.
These components can include the central processing unit (CPU), memory, storage devices, and input/output (I/O) devices.
Data transfer within a computer system occurs through the use of a bus, which is a set of communication lines that allow different components to communicate with each other.
Name
Instruction
Load
LD
Store
ST
Move
MOV
Exchange
XCH
Input
IN
Output
OUT

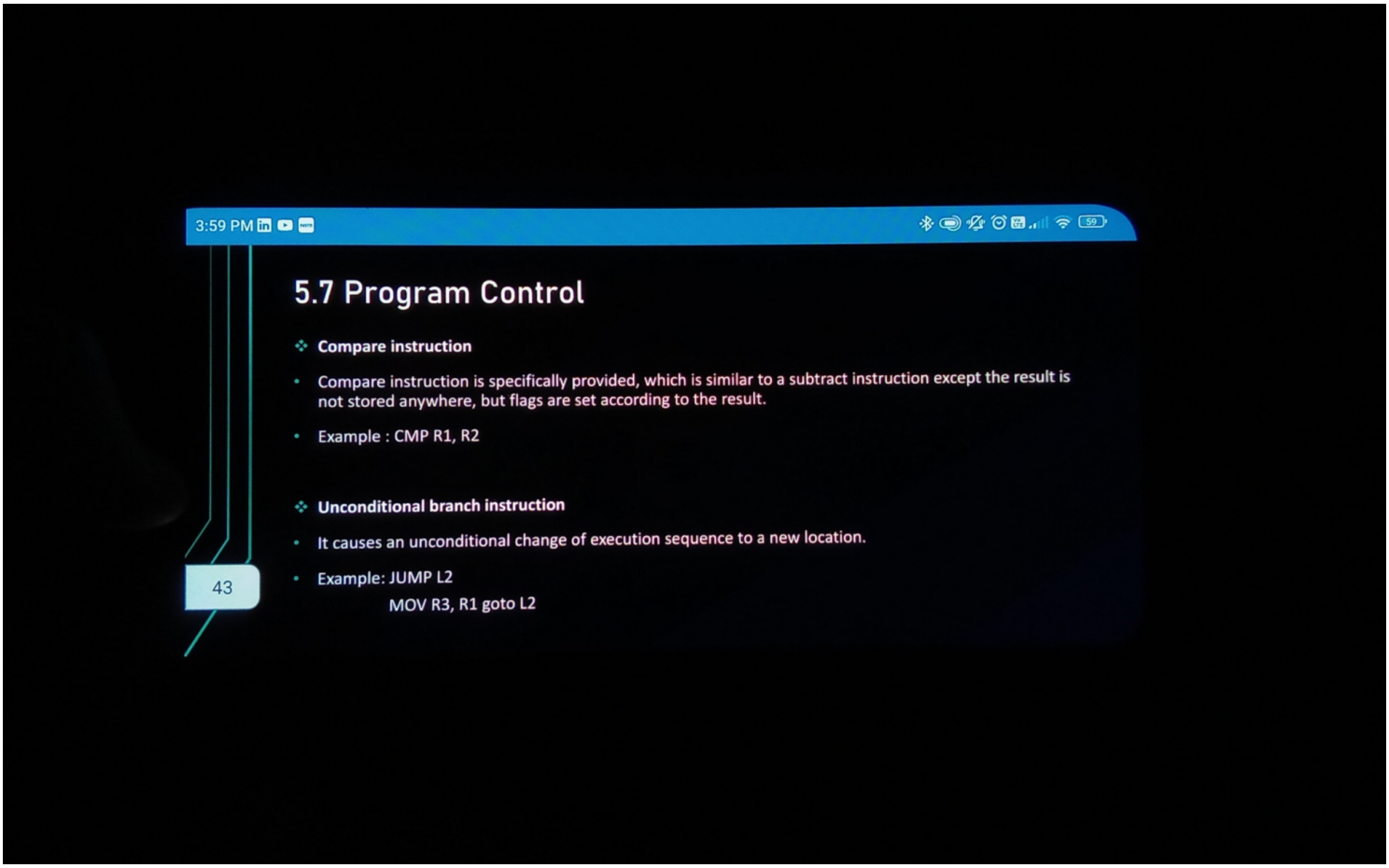

3:59 PM
5.6 Data transfer and manipulation
Data manipulation in computer architecture refers to the various operations that can be performed on data stored in a computer system.
These operations may include sorting, filtering, merging and transforming data as well as extracting and summarizing data from large data sets.
The data manipulation instructions in a typical computer is divided into three types:
1) Arithmetic instructions
2) Logical and bit manipulation instructions
3) Shift instructions
38

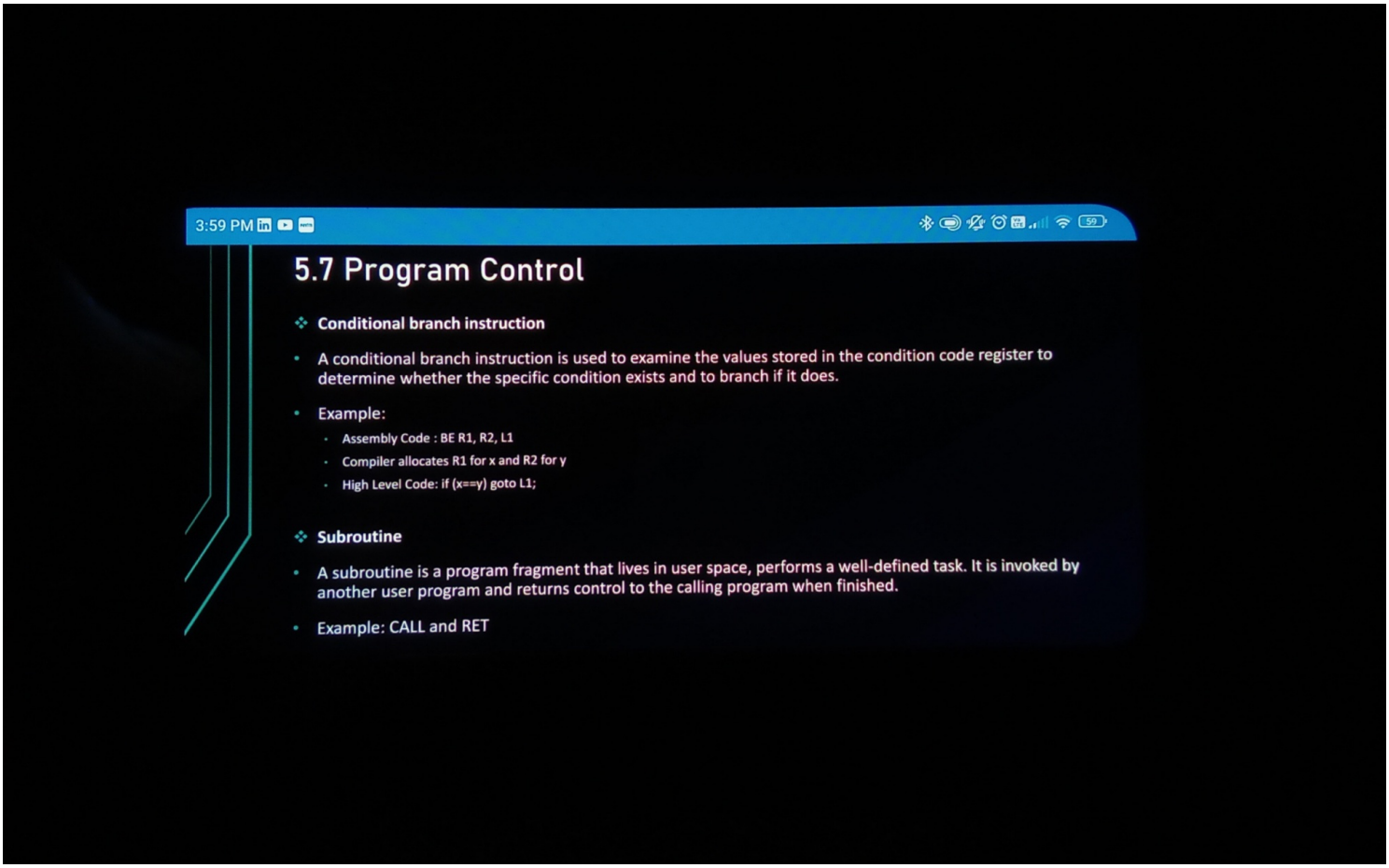

3:59 PM
5.6 Data transfer and manipulation
The four basic arithmetic operation are addition, subtraction, multiplication, division.
The increment instruction adds 1 to the value stored in a register or memory word. The decrement instruction subtracts 1 from the value stored in the register.
The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation. The "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation.
The negate instruction form the 2's complement of a number, and reversing the sign of integer when represented in the signed 2's complement.
Name
Instruction
Increment
INC
Decrement
DEC
Add
ADD
Subtract
SUB
Multiply
MUL
Divide
DIV
Add with carry
ADDC
Subtract with borrow
SUBB
Negate(2's complement)
NEG

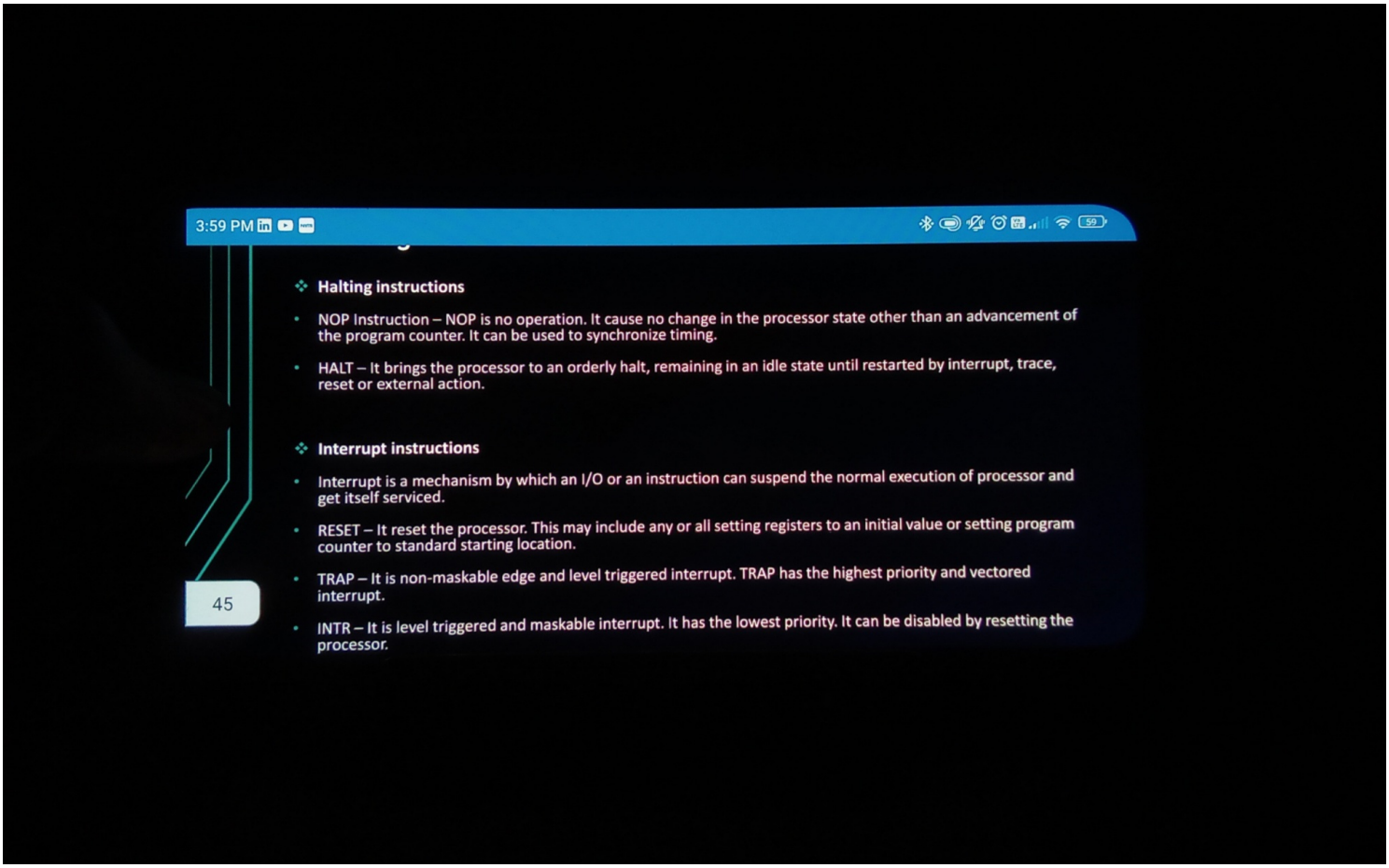

3:59 PM
5.8 Data transfer and manipulation
Logical instructions perform binary operations on strings of bits stored in registers.
They are useful for manipulating individual bits or a group of bits that represent binary-coded information.
The AND instruction is used to clear a bit or a selected group of bits of an operand.
The OR instruction is used to set a bit or a selected group of bits of an operand.
Similarly, the XOR instruction is used to selectively complement bits of an operand.
Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions.
Name
Instruction
Clear
CLR
Complement
COM
AND
AND
OR
OR
Exclusive OR
XOR
Clear Carry
CLRC
Set Carry
SETC
Complement carry
COMC
Enable interrupt
EI
Disable interrupt
DI


3:59 PM
5.6 Data transfer and manipulation
Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right.
The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.
The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same.
The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.
Name
Instruction
Logical shift right
SHR
Logical shift left
SHL
Arithmetic shift right
SHRA
Arithmetic shift left
SHLA
Rotate right
ROR
Rotate left
ROL
Rotate right through carry
RORC
Rotate left through carry
ROLC


3:59 PM
5.7 Program Control
Program Control Instructions are the machine code that are used by machine or in assembly language by user to command the processor act accordingly. T
hese instructions are of various types. These are used in assembly language by user also. But in level language, user code is translated into machine code and thus instructions are passed to instruct the processor do the task.
There are different types of Program Control Instructions:
Compare instruction
Unconditional branch instruction
Conditional branch instruction
Subroutines
Halting instruction
Interrupt instructions:


3:59 PM
5.7 Program Control
Compare instruction
Compare instruction is specifically provided, which is similar to a subtract instruction except the result is not stored anywhere, but flags are set according to the result.
Example : CMP R1, R2
Unconditional branch instruction
It causes an unconditional change of execution sequence to a new location.
Example: JUMP L2
MOV R3, R1 goto L2
43

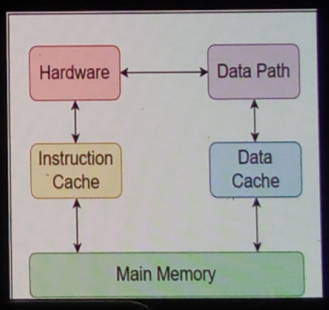

3:59 PM
5.7 Program Control
Conditional branch instruction
A conditional branch instruction is used to examine the values stored in the condition code register to determine whether the specific condition exists and to branch if it does.
Example:
Assembly Code : BE R1, R2, L1
Compiler allocates R1 for x and R2 for y
High Level Code: if (x==y) goto L1;
Subroutine
A subroutine is a program fragment that lives in user space, performs a well-defined task. It is invoked by another user program and returns control to the calling program when finished.
Example: CALL and RET


3:59 PM
Halting instructions
NOP Instruction – NOP is no operation. It cause no change in the processor state other than an advancement of the program counter. It can be used to synchronize timing.
HALT – It brings the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset or external action.
Interrupt instructions
Interrupt is a mechanism by which an I/O or an instruction can suspend the normal execution of processor and get itself serviced.
RESET – It reset the processor. This may include any or all setting registers to an initial value or setting program counter to standard starting location.
TRAP – It is non-maskable edge and level triggered interrupt. TRAP has the highest priority and vectored interrupt.
INTR – It is level triggered and maskable interrupt. It has the lowest priority. It can be disabled by resetting the processor.
45

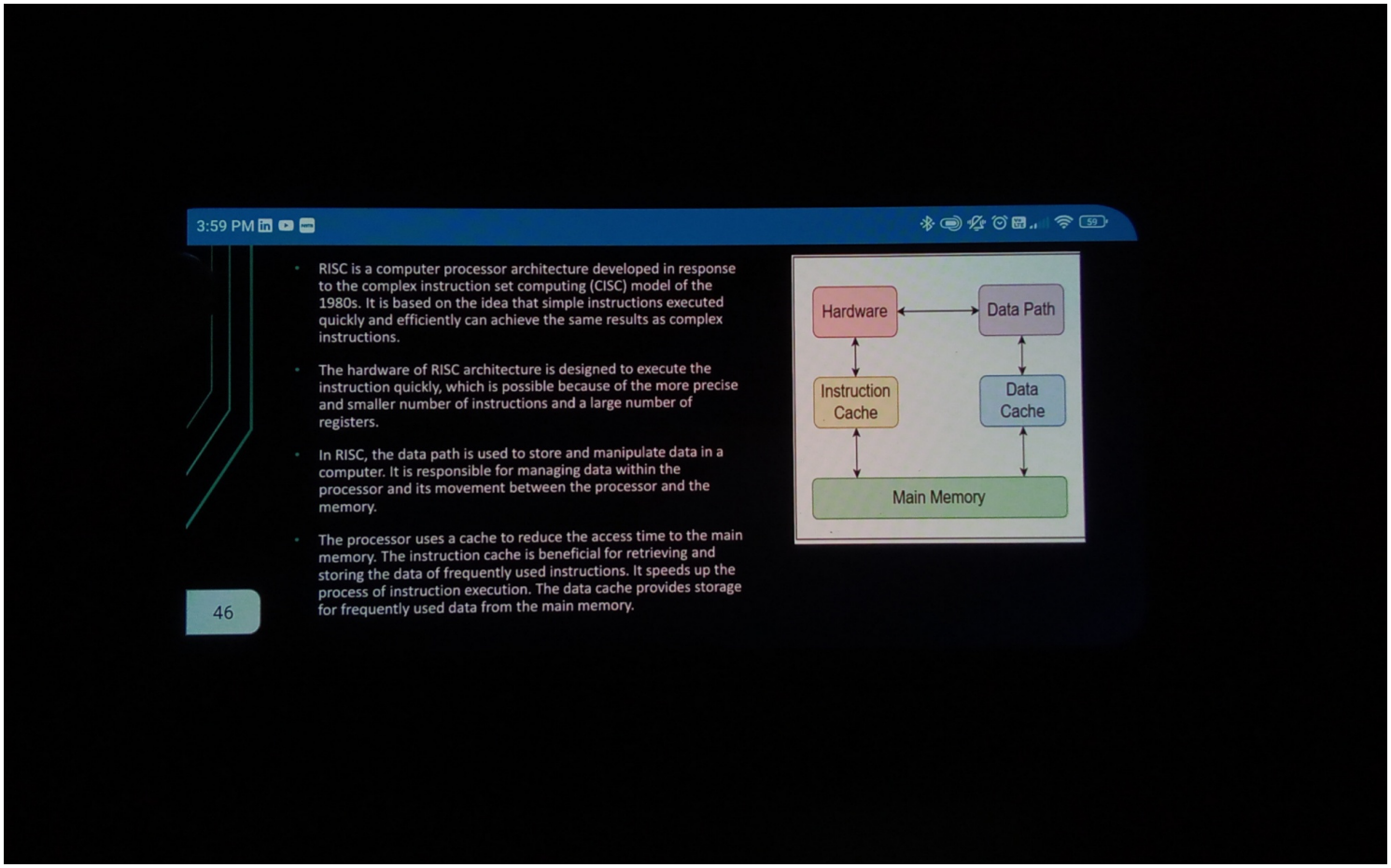
3:59 PM

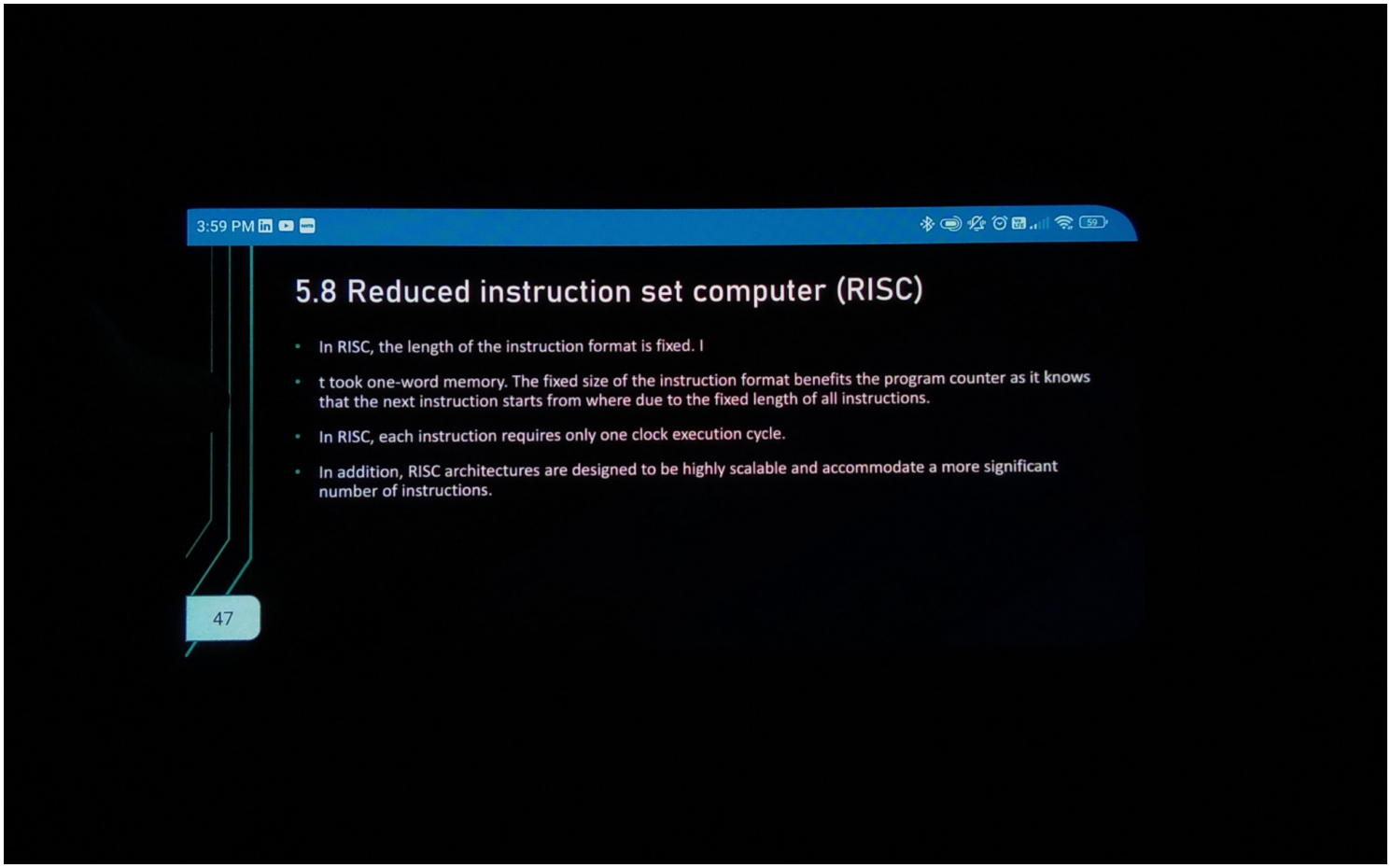
5.8 Reduced instruction set computer (RISC)

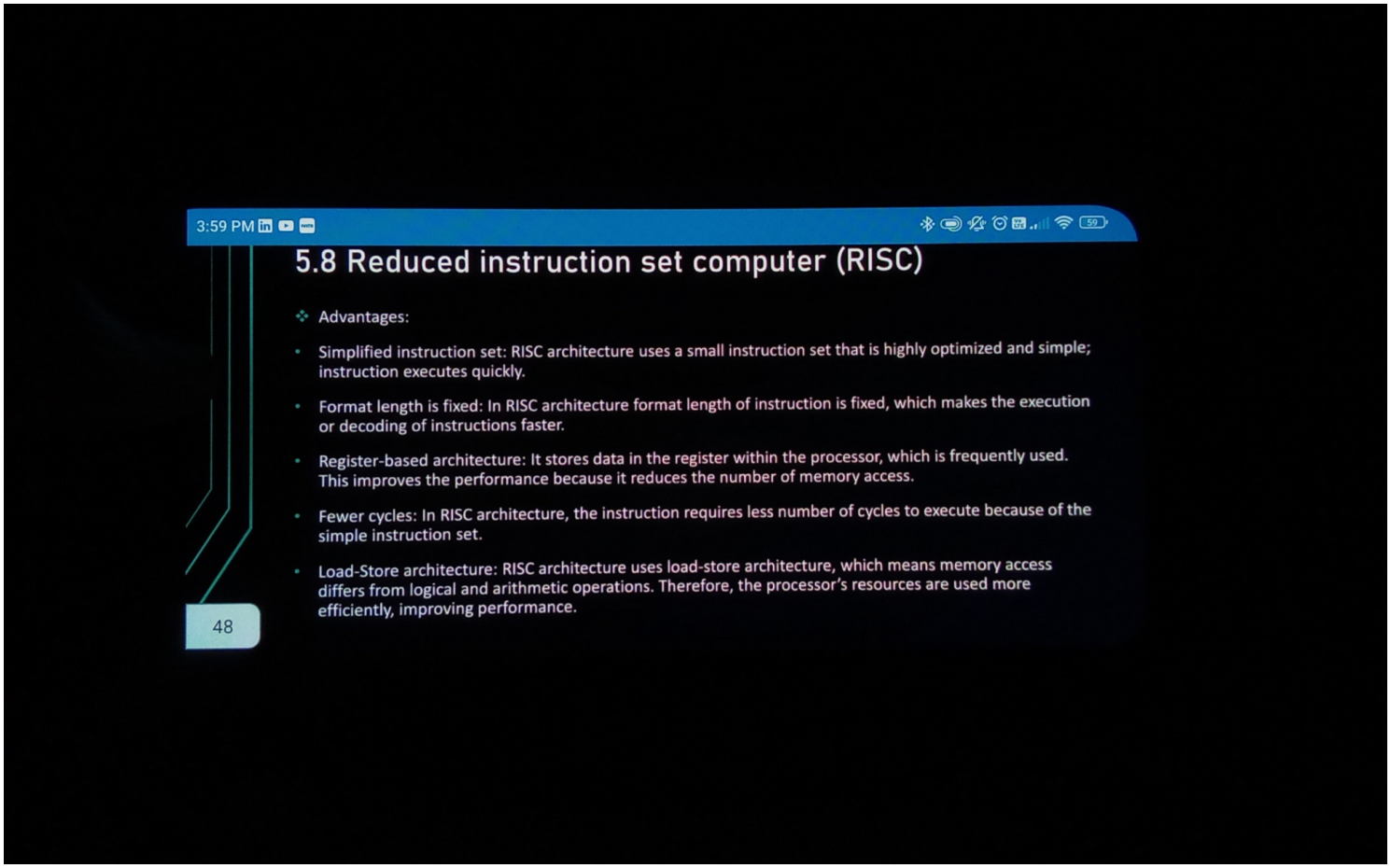
- The Reduced Instruction Set Computer (RISC) architecture reduces execution time by simplifying the instruction set.
- RISC is a computer processor architecture developed in response to the complex instruction set computing (CISC) model of the 1980s. It is based on the idea that simple instructions executed quickly and efficiently can achieve the same results as complex instructions.
- The hardware of RISC architecture is designed to execute the instruction quickly, which is possible because of the more precise and smaller number of instructions and a large number of registers.
- In RISC, the data path is used to store and manipulate data in a computer. It is responsible for managing data within the processor and its movement between the processor and the memory.

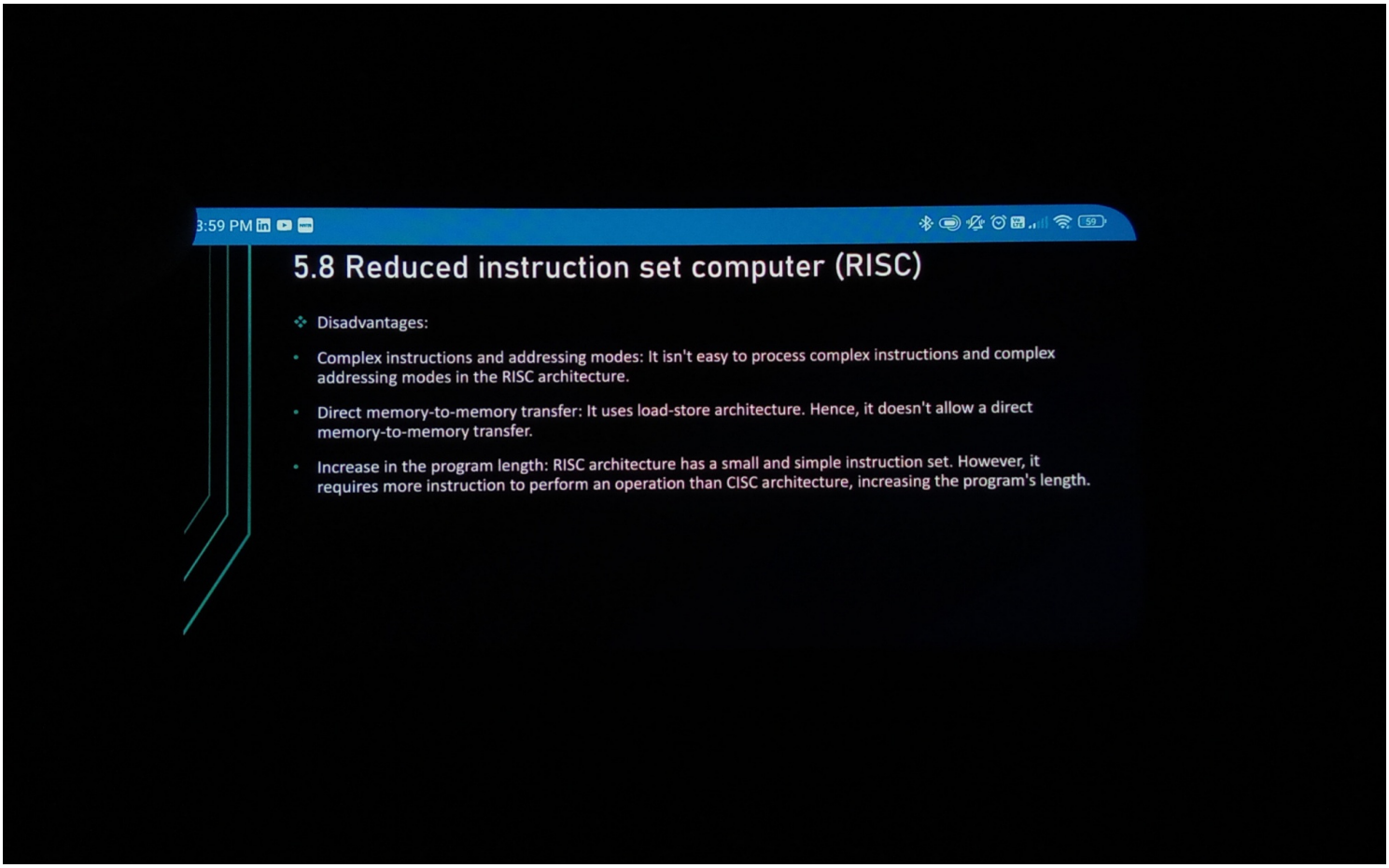


46


3:59 PM
RISC is a computer processor architecture developed in response to the complex instruction set computing (CISC) model of the 1980s. It is based on the idea that simple instructions executed quickly and efficiently can achieve the same results as complex instructions.
The hardware of RISC architecture is designed to execute the instruction quickly, which is possible because of the more precise and smaller number of instructions and a large number of registers.
In RISC, the data path is used to store and manipulate data in a computer. It is responsible for managing data within the processor and its movement between the processor and the memory.
The processor uses a cache to reduce the access time to the main memory. The instruction cache is beneficial for retrieving and storing the data of frequently used instructions. It speeds up the process of instruction execution. The data cache provides storage for frequently used data from the main memory.
46
Hardware
Data Path
Instruction Cache
Data Cache
Main Memory


3:59 PM
5.8 Reduced instruction set computer (RISC)
In RISC, the length of the instruction format is fixed. I
t took one-word memory. The fixed size of the instruction format benefits the program counter as it knows that the next instruction starts from where due to the fixed length of all instructions.
In RISC, each instruction requires only one clock execution cycle.
In addition, RISC architectures are designed to be highly scalable and accommodate a more significant number of instructions.
47


3:59 PM
5.8 Reduced instruction set computer (RISC)
Advantages:
Simplified instruction set: RISC architecture uses a small instruction set that is highly optimized and simple; instruction executes quickly.
Format length is fixed: In RISC architecture format length of instruction is fixed, which makes the execution or decoding of instructions faster.
Register-based architecture: It stores data in the register within the processor, which is frequently used. This improves the performance because it reduces the number of memory access.
Fewer cycles: In RISC architecture, the instruction requires less number of cycles to execute because of the simple instruction set.
Load-Store architecture: RISC architecture uses load-store architecture, which means memory access differs from logical and arithmetic operations. Therefore, the processor's resources are used more efficiently, improving performance.
48


3:59 PM
5.8 Reduced instruction set computer (RISC)
Disadvantages:
Complex instructions and addressing modes: It isn't easy to process complex instructions and complex addressing modes in the RISC architecture.
Direct memory-to-memory transfer: It uses load-store architecture. Hence, it doesn't allow a direct memory-to-memory transfer.
Increase in the program length: RISC architecture has a small and simple instruction set. However, it requires more instruction to perform an operation than CISC architecture, increasing the program's length.