MODULE 1

1. Microprogrammed Control

The control unit of a CPU generates the sequence of control signals needed to execute instructions. In hardwired control, these signals are produced by fixed logic circuitry, which is fast but difficult to modify. In microprogrammed control, control signals are generated by a small, specialized program (microprogram) stored in a dedicated control memory — making the control unit flexible, systematic, and easy to modify or extend.

1.1 Control Memory

Control memory is a fast, dedicated storage unit that holds microprograms consisting of microinstructions. Each microinstruction specifies one or more microoperations to be executed during one control step (clock cycle).

- Control Word (CW): The bit pattern within a microinstruction that specifies one or more microoperations for the components of the computer.
- Control Address Register (CAR): Holds the address of the microinstruction currently being fetched from control memory.
- Control Memory can be implemented as a ROM (fixed microprogram) or as a writable control store (WCS), which allows the microprogram to be altered.
- A sequencer (next-address generator) determines the address of the next microinstruction, based on status bits, the current microinstruction, and mapping logic.

Note: A computer that employs microprogrammed control has two levels: the microprogram level, which defines the primitive control functions, and the machine instruction level, whose instructions are interpreted by a sequence of microinstructions.

1.2 Address Sequencing

Address sequencing refers to the techniques used to determine the address of the next microinstruction in control memory. Common address-sequencing requirements include:

- Incrementing the control address register to fetch the next microinstruction in sequence.
- Unconditional and conditional branching, based on status (condition) bits, to alternate microprogram paths.
- A mapping process that converts the bits of the machine instruction's operation code (opcode) into a control-memory address — this is how the control unit locates the start of the microroutine for each machine instruction.
- A facility for subroutine call and return within the microprogram, typically implemented using a small internal stack to hold return addresses.

A typical sequencer design uses a multiplexer to select the next address source, based on control bits within the current microinstruction, from among: the incremented CAR, a branch address field, a subroutine-return address, or a mapped address derived from the opcode.

1.2.1 Mapping of Instructions to Microroutines

A common mapping technique reserves a fixed field of control-memory address bits taken directly from the opcode bits of the machine instruction, with the remaining address bits set to zero. This provides a simple, direct correspondence between an operation code and the starting address of its microroutine in control memory.

1.3 Microprogram Example

A microprogram example typically illustrates how a simple computer's instruction set is translated into sequences of microinstructions stored in control memory. A representative microprogram for a fetch–decode–execute cycle includes:

1. 1. Fetch microroutine: transfers the instruction from memory (addressed by the Program Counter) into the Instruction Register, and increments the Program Counter.
2. 2. Decode / mapping step: the opcode field is used to branch to the appropriate microroutine for that instruction.
3. 3. Execute microroutine: performs the sequence of microoperations that implement the instruction (e.g., register transfers, ALU operations, memory read/write).
4. 4. Return to the fetch microroutine to process the next machine instruction.

Each step in the microprogram corresponds to one microinstruction, and the fields of that microinstruction directly enable the datapath control signals (e.g., register-select lines, ALU function-select lines, memory read/write lines) needed for that step.

1.4 Design of Control Unit

The design of a microprogrammed control unit involves defining the format of microinstructions and the hardware needed to generate control signals and next-microinstruction addresses.

1.4.1 Microinstruction Format

- Horizontal microinstruction format: Each bit in the control word directly controls one microoperation/control line; this gives maximum parallelism and flexibility but requires a wide control memory word.
- Vertical microinstruction format: Control signals are encoded into compact fields, requiring a decoder to generate the actual control lines; this reduces control-word width at the cost of some parallelism and added decoding delay.

1.4.2 Basic Control Unit Organization

A microprogrammed control unit is generally organized around: a control memory (holding the microprogram), a control address register (CAR) and next-address generation logic, a control data/microinstruction register to hold the currently executing microinstruction, and decoding logic that converts control-word fields into actual control signals sent to the datapath.

Note: The key trade-off in control unit design is hardwired control (faster, less flexible) versus microprogrammed control (slower per microinstruction, but far easier to design, debug, and modify).

MODULE 2

2. Central Processing Unit

The Central Processing Unit (CPU) is the part of the computer that executes instructions, performing arithmetic, logic, control, and data-movement operations. This module examines how CPU registers are organized, how machine instructions are formatted, how operands are addressed, and the categories of operations a CPU performs.

2.1 General Register Organization

Rather than using a single accumulator, most modern CPUs contain a set of general-purpose registers that can be used to hold operands, intermediate results, and addresses, improving performance by reducing memory accesses.

- A common bus system connects the general registers, the ALU, and other CPU components, with multiplexers used to select the source of data placed on the bus.
- Control signals select which registers act as ALU inputs and which register receives the ALU output, allowing a single ALU and bus structure to support many different register-to-register operations.
- Typical registers include general-purpose registers (R0–Rn), an ALU with a function-select input, and often a stack pointer and status/flag register.

A representative micro-operation using this organization: R3 ← R1 + R2, performed by placing R1 and R2 on the input buses of the ALU, setting the ALU function to ADD, and enabling R3 to load the ALU output.

2.2 Instruction Formats

The instruction format defines the layout of bits within a machine instruction, typically including an operation code (opcode) field and one or more operand-address fields. The number of address fields used gives rise to different instruction organizations:

Format	Description	Example (typical form)
Three-address	Specifies two source operand addresses and one destination address.	ADD R1, R2, R3 (R1 ← R2 + R3)
Two-address	One address doubles as both a source and the destination.	ADD R1, R2 (R1 ← R1 + R2)
One-address	Uses an implied accumulator (AC) as the second operand and destination.	ADD M (AC ← AC + M)
Zero-address	Uses a stack; operands are implicitly the top elements of the stack.	ADD (push result of popping two operands and adding)

Reducing the number of address fields tends to shorten instructions (saving memory and fetch bandwidth) but generally requires more instructions to perform the same computation.

2.3 Addressing Modes

An addressing mode specifies the rule for interpreting or modifying the address field of an instruction before the actual operand is referenced. Addressing modes provide flexibility for accessing data in different ways without needing separate instructions for each access pattern.

Addressing Mode	Description
Immediate	The operand is given explicitly in the instruction itself (no memory reference needed).
Register	The operand is held in a CPU register, specified by a register field in the instruction.
Register Indirect	The named register holds the address (in memory) of the operand, rather than the operand itself.
Direct (Absolute)	The address field of the instruction directly gives the memory address of the operand.
Indirect	The address field gives the address of a memory location that itself contains the address of the operand.
Relative	The effective address is obtained by adding the program counter (PC) to the address field, useful for position-independent branch targets.
Indexed	The effective address is obtained by adding the contents of an index register to the address field, useful for accessing array elements.
Base Register	The effective address is obtained by adding the contents of a base register to the address field, useful for relocatable programs.
Autoincrement / Autodecrement	Similar to register indirect, but the register is automatically incremented or decremented after (or before) it is used, useful for stepping through sequential data.

2.4 Data Transfer and Manipulation

Most machine instructions fall into one of three broad functional categories: data transfer, data manipulation, and program control. This section focuses on the first two.

2.4.1 Data Transfer Instructions

Data transfer instructions move data from one location to another without changing its value — between registers, between a register and memory, or between memory locations. Typical operations include Load, Store, Move, Push, Pop, and Exchange.

2.4.2 Data Manipulation Instructions

Data manipulation instructions perform operations that change the value of data, and are grouped into three subcategories:

- Arithmetic instructions: Add, Subtract, Multiply, Divide, Increment, Decrement, and Negate (complement).

- Logical and bit-manipulation instructions: AND, OR, XOR, NOT (Complement), Clear, and Set — used for masking, testing, and manipulating individual bits.
- Shift instructions: Logical shift, Arithmetic shift, and Rotate (with or without carry) — used to reposition bits, perform quick multiplication/division by powers of two, and support serial data operations.

2.5 Program Control

Program control instructions alter the normal sequential flow of instruction execution, enabling decision-making, loops, and subroutine calls.

- Branch / Jump: Transfers control unconditionally to a specified instruction address.
- Conditional Branch: Transfers control only if a specified condition (based on status/flag bits such as zero, carry, sign, overflow) is satisfied.
- Call and Return: Call transfers control to a subroutine while saving the return address (typically on a stack); Return restores control to the instruction following the original call.
- Compare and Test instructions: Perform a subtraction or logical operation to set status flags, without storing the result, for use by subsequent conditional branches.
- Interrupt / Trap instructions: Transfer control to a special handling routine in response to an exceptional condition or explicit software request.

Note: A status (flag) register typically holds condition-code bits such as Carry (C), Zero (Z), Sign (S), and Overflow (V), which are tested by conditional program-control instructions.

MODULE 3

3. Computer Arithmetic

Computer arithmetic deals with the algorithms and hardware used to perform arithmetic operations — addition, subtraction, multiplication, and division — on both fixed-point (integer) and floating-point (real) numbers, as well as decimal (BCD) representations.

3.1 Addition and Subtraction

3.1.1 Signed-Magnitude Addition and Subtraction

In signed-magnitude representation, the sign and magnitude are handled separately: to add two numbers with the same sign, their magnitudes are added and the common sign is kept; to add numbers with different signs, the smaller magnitude is subtracted from the larger, and the sign of the larger-magnitude number is used for the result.

3.1.2 Signed 2's Complement Addition and Subtraction

Addition: the two numbers, including their sign bits, are added directly as unsigned binary numbers; any carry out of the sign bit is discarded.

Subtraction: A − B is performed by taking the 2's complement of B and adding it to A (i.e., A + (−B)), again discarding any carry out of the sign bit.

Note: Overflow in 2's complement addition/subtraction is detected when the carry into the sign bit differs from the carry out of the sign bit (or equivalently, when two operands of the same sign produce a result of the opposite sign).

3.2 Multiplication Algorithms

3.2.1 Booth's Multiplication Algorithm

Booth's algorithm multiplies two signed numbers in 2's complement form by examining pairs of adjacent multiplier bits and a Booth-recoding rule, allowing strings of consecutive 1s in the multiplier to be handled with a single addition and subtraction rather than a series of additions.

- If the current bit pair is 0,0 or 1,1: no arithmetic operation; simply shift.
- If the current bit pair is 0,1: add the multiplicand to the partial product, then shift.
- If the current bit pair is 1,0: subtract the multiplicand from the partial product, then shift.

Booth's algorithm reduces the number of addition operations needed, especially for multipliers containing long runs of 1s, and directly supports signed operands.

3.2.2 Array Multiplier

An array multiplier is a combinational hardware structure built from an array of AND gates (to form partial products) and adders arranged so all partial products are added in parallel, offering higher speed at the cost of increased hardware compared to sequential shift-and-add multiplication.

3.3 Division Algorithms

3.3.1 Restoring Division

In restoring division, the divisor is repeatedly subtracted from the partial remainder; if the result is negative, the divisor is added back (restoring the previous value) and the corresponding quotient bit is set to 0; if the result is non-negative, the quotient bit is set to 1 and the subtraction result is retained as the new partial remainder.

3.3.2 Non-Restoring Division

Non-restoring division avoids the extra restoring addition step: if the previous partial remainder was non-negative, the divisor is subtracted; if it was negative, the divisor is added instead. The corresponding quotient bit is set based on the sign of the result at each step, and only a single final correction (if needed) is applied at the end.

Note: Non-restoring division is generally faster than restoring division because it avoids the extra restore operation at every step where the partial remainder goes negative.

3.4 Floating-Point Arithmetic Operations

Floating-point numbers are represented in the general form (−1)^S × Mantissa × 2^Exponent (see IEEE 754 format). Arithmetic on floating-point numbers requires special handling of the exponent and mantissa fields.

3.4.1 Floating-Point Addition and Subtraction

5. 1. Compare exponents of the two operands; align (shift) the mantissa of the number with the smaller exponent to match the larger exponent.
6. 2. Add or subtract the aligned mantissas.
7. 3. Normalize the result so that the mantissa satisfies the standard normalized form (adjusting the exponent accordingly).
8. 4. Round the result to fit the available mantissa width, and check for overflow or underflow in the exponent.

3.4.2 Floating-Point Multiplication and Division

- Multiplication: multiply the mantissas, add the exponents (subtracting the bias once, since it would otherwise be counted twice), determine the sign by XOR-ing the operand signs, then normalize and round.
- Division: divide the mantissas, subtract the exponents (adding the bias back), determine the sign by XOR-ing the operand signs, then normalize and round.

3.5 Decimal Arithmetic Unit

A decimal arithmetic unit performs arithmetic directly on numbers represented in BCD (binary-coded decimal) form, which is important in applications (such as financial and commercial computing) where exact decimal representation is required and binary rounding errors are unacceptable.

- A decimal arithmetic unit typically consists of parallel decimal adder stages (one per BCD digit), each including the binary-adder-plus-correction logic described for BCD addition, along with decimal-carry propagation between digit stages.
- Additional logic is included to detect invalid (non-BCD) digit combinations and to apply the +6 correction whenever a digit sum exceeds 9 or a carry is generated, as described in BCD addition.

3.6 Decimal Arithmetic Operations

3.6.1 BCD Addition

As described previously, BCD digits are added using a 4-bit binary adder per digit pair; if the 4-bit result exceeds 9 or produces a carry, 0110 (6) is added to correct the digit and generate the correct decimal carry into the next digit position.

3.6.2 BCD Subtraction

BCD subtraction can be performed using 9's complement or 10's complement arithmetic, analogous to 1's and 2's complement subtraction in binary: the 9's (or 10's) complement of the subtrahend is added to the minuend, and a correction is applied to the result based on whether an end-around carry occurs.

3.6.3 Decimal Multiplication and Division

Decimal multiplication and division are typically implemented using repeated decimal addition/subtraction and shifting, analogous to the binary shift-and-add / shift-and-subtract algorithms, but operating digit-by-digit on BCD-encoded values rather than bit-by-bit on pure binary values.