

UNIT 4

Microprogrammed Control

(10 Hours) — Microprocessor and Computer Architecture (CACS155)

In any digital computer, the Control Unit is responsible for generating the sequence of control signals that direct the ALU, registers and memory to carry out each instruction. There are two fundamentally different approaches to designing a control unit: hardwired control, in which control signals are produced by fixed combinational logic circuits, and microprogrammed control, in which control signals are generated by a small, specialised "program" (the microprogram) stored in a Control Memory. This unit studies microprogrammed control in depth — beginning with the design of a simple accumulator-based processor, moving through control-memory address sequencing and microprogram representation, and concluding with the internal design of the control unit itself.

1. Basic Computer Design of Accumulator

A simple accumulator-based computer performs every arithmetic and logic operation using the Accumulator (AC) register as one operand source and as the destination of the result. Understanding how the AC register is controlled, and how it connects to the ALU, illustrates the general register-transfer principles that a microprogram must ultimately control.

1.1 Control of the AC Register

The AC register is not a passive storage element — it is controlled by a set of signals generated by the control unit that determine, on every clock pulse, what value it should load. Typical control functions associated with the AC register include:

Control Signal	Micro-operation	Effect
LD (Load)	$AC \leftarrow \text{ALU output}$	Loads the AC register with the result just computed by the ALU on the next clock edge.
INR (Increment)	$AC \leftarrow AC + 1$	Increments the contents of AC by one (used in counting and address-modification operations).
CLR (Clear)	$AC \leftarrow 0$	Resets all bits of AC to zero.
No signal asserted	$AC \leftarrow AC$ (no change)	AC retains its present value if none of its control inputs is active during that clock pulse.

Note: Each control signal (LD, INR, CLR) is itself the output of combinational logic driven by the microinstruction currently being executed — this is exactly the link between the microprogram and the actual hardware registers.

1.2 ALU Organization

The Arithmetic-Logic Unit (ALU) is a combinational circuit that performs a set of arithmetic, logic, and shift micro-operations on data supplied to it, typically from the AC register and the Data Register (DR). Its output is then optionally loaded back into AC.

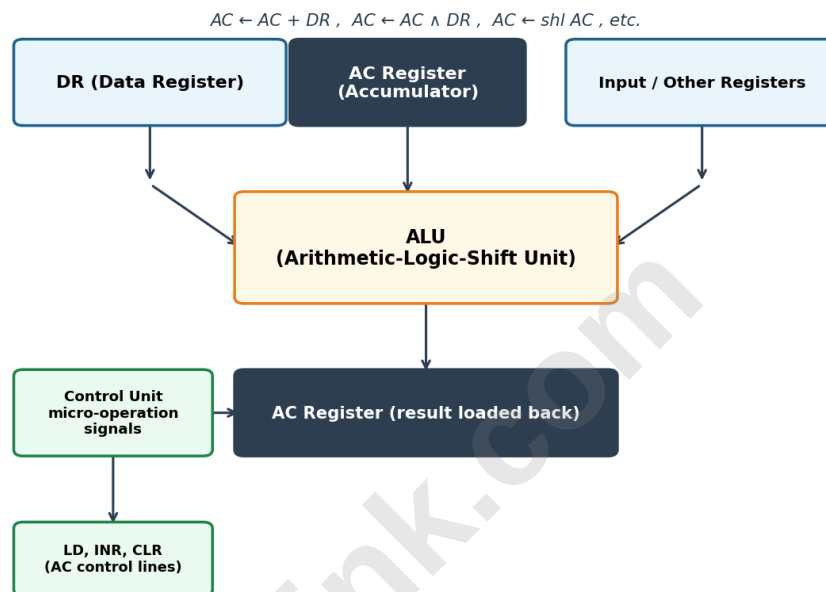


Figure 4.1 — Organization of the AC register with the ALU

Operation Type	Typical Micro-operations
Arithmetic	$AC \leftarrow AC + DR, AC \leftarrow AC - DR, AC \leftarrow AC + 1$
Logic	$AC \leftarrow AC \wedge DR$ (AND), $AC \leftarrow AC \vee DR$ (OR), $AC \leftarrow AC \oplus DR$ (XOR), $AC \leftarrow \overline{AC}$ (complement)
Shift	$AC \leftarrow shl AC$ (shift left), $AC \leftarrow shr AC$ (shift right)

The ALU is typically implemented as a set of parallel functional blocks (an adder/subtractor, logic-gate array, and shifter) whose outputs are selected by a multiplexer according to the operation code specified in the microinstruction currently active in the control unit. Because the ALU has no memory of its own, its output must be captured into AC (or another register) on the very next clock pulse, or it is lost.

2. Control Memory — Address Sequencing

In a microprogrammed control unit, all control signals for every machine instruction are stored, in encoded form, as a sequence of microinstructions inside a fast Read-Only Memory called the Control Memory (CM). The address of the microinstruction currently being read is held in the Control Address Register (CAR). The process of determining, cycle by cycle, which address CAR should hold next is called address sequencing.

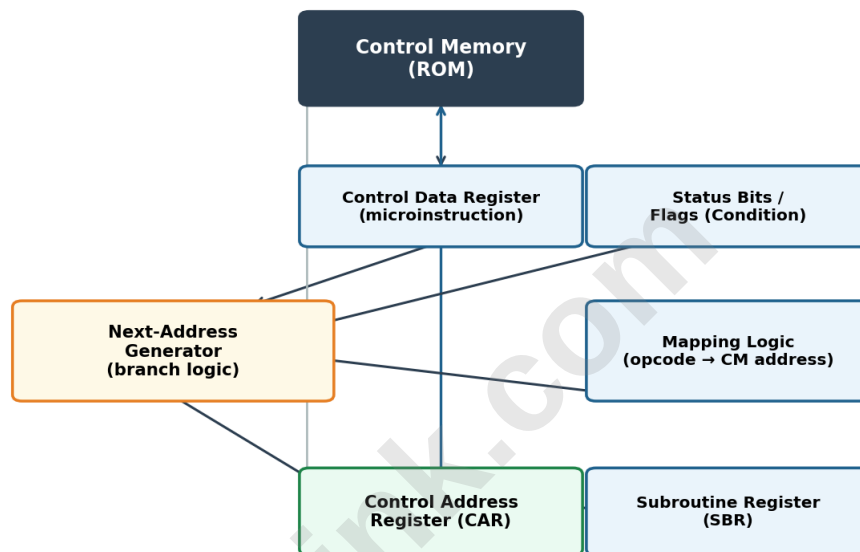


Figure 4.2 — Control memory address-sequencing organization

In general, the next control-memory address can come from one of three sources:

- Sequential execution — CAR is simply incremented ($CAR \leftarrow CAR + 1$) to fetch the next microinstruction in sequence, just as a program counter advances through ordinary machine instructions.
- Branching — CAR is loaded with a branch address that is itself specified as a field within the current microinstruction, taken conditionally or unconditionally.
- Mapping — when a new machine-language instruction is fetched, CAR is loaded with the starting control-memory address of that instruction's microroutine, generated by a mapping process from the instruction's opcode.

2.1 Conditional Branching

Instead of providing a separate microinstruction for every possible outcome of a decision (which would be wasteful), the control unit tests one status (condition) bit at a time and branches

accordingly. Typical status bits tested include the ALU's Carry flag, the Zero flag, or an instruction's addressing-mode bit (direct/indirect).

Field in Microinstruction	Purpose
Condition select	Chooses which status bit (e.g., Carry, Zero, indirect-bit I) is to be tested.
Branch address	Specifies the control-memory address to jump to if the tested condition is true.
Branch control (BR)	Encodes the type of address-selection to use: e.g., unconditional branch, conditional branch, mapping, or simple increment (JMP/CALL/RET-like function).

Example 4.1

A control unit tests the indirect-mode bit I of the current instruction. If $I = 0$, control should proceed to microinstruction address 100; if $I = 1$ (indirect addressing), it should first branch to an indirect-address-resolution routine at address 300. Explain how conditional branching accomplishes this.

Solution

The current microinstruction's Branch field specifies address 300 as the conditional target and selects the status bit I as the condition to test.

If $I = 1$, the next-address logic loads CAR with 300 (the branch address).

If $I = 0$, the next-address logic instead loads CAR with 100 (the address following the current microinstruction, i.e., normal sequencing).

This single conditional-branch microinstruction therefore replaces what would otherwise require two separate hard-wired paths.

2.2 Mapping of Instructions to Microroutines

When a machine-language instruction is fetched into the Instruction Register, its opcode must be translated into the starting address, in control memory, of the microroutine that implements it. This translation is called mapping, and is usually performed by a small, fast piece of dedicated hardware (a mapping ROM or a simple hard-wired transformation) rather than by the general-purpose microprogram itself, since it must execute in a single clock cycle right after instruction fetch.

A common and simple mapping technique places the opcode bits directly into the low-order bits of the control-memory address, with a fixed prefix (e.g., 0) in the high-order bits:

Example 4.2 — Simple Mapping Scheme

For a 4-bit opcode (0000 to 1111) and a control memory with 7-bit addresses, one simple mapping rule is:

Control memory address = 0, 0, Opcode(3), Opcode(2), Opcode(1), Opcode(0), 0

That is, the opcode bits are placed in the middle of the 7-bit address, with two fixed 0-bits on the left and one fixed 0-bit on the right (leaving room for a short microroutine at each mapped address before it must branch out).

Note: This method guarantees that every instruction's microroutine begins at a distinct, easily-computed control-memory address, without requiring a large, slow lookup table.

bcalink.com

3. Micro Program

A microprogram is the set of microinstructions stored in control memory that together implement the complete instruction set of the computer. Just as a machine-language program is written using mnemonics before being assembled into binary, a microprogram is normally first written in a readable symbolic form, and then translated into the binary form that is actually loaded into the control memory ROM.

3.1 Symbolic Microprogram

A symbolic microprogram is written using an assembly-language-like notation, typically with one line per microinstruction, broken into fields corresponding to different parts of the microinstruction format (for example: an F1, F2, F3 micro-operation field, a CD condition field, a BR branch-type field, and an AD address field). A microinstruction may specify several micro-operations that are to occur simultaneously, since many register-transfer operations can share the same clock pulse.

Symbolic Field	Meaning	Example Value
Micro-operation field(s)	One or more symbols denoting simultaneous register-transfer micro-operations	$AC \leftarrow AC + DR$
CD (Condition)	Which status bit to test for a conditional branch	I (indirect bit)
BR (Branch)	Type of next-address control: JMP, CALL, RET, MAP	CALL
AD (Address)	Symbolic label of the branch target address	INDRCT

Example 4.3 — Symbolic Microprogram Fragment

ORG 0

FETCH: PC $\rightarrow$ MAR CALL INDRCT ; if indirect, resolve address first

MDR $\leftarrow$ M[MAR], PC $\leftarrow$ PC + 1

MDR $\rightarrow$ IR MAP ; map opcode to microroutine

INDRCT: MAR $\leftarrow$ IR(ADDR)

MDR $\leftarrow$ M[MAR]

MDR $\rightarrow$ IR(ADDR) RET ; return to caller (mapping step)

3.2 Binary Microprogram

Each symbolic microinstruction, once written, is translated field-by-field into its binary equivalent, based on a fixed encoding table defined for the machine — exactly analogous to how

an assembler converts mnemonics like MOV or ADD into binary opcodes. The resulting binary words are what is physically stored, one per control-memory address, in the Control Memory ROM.

Example 4.4 — Binary Encoding of Example 4.3

Suppose the microinstruction format allocates: 3 bits for F1, 3 bits for F2, 3 bits for F3, 2 bits for CD, 2 bits for BR, and 7 bits for AD (total 20 bits). If F1 = 101 (PC→MAR), CD = 00 (unconditional), BR = 11 (CALL) and AD = 0100000 (INDRCT = address 32):

Solution

The binary microinstruction word would be encoded as:

F1(101) F2(000) F3(000) CD(00) BR(11) AD(0100000)

= 101 000 000 00 11 0100000

This 20-bit binary pattern is what is actually stored at the corresponding control-memory address — the symbolic form exists only for the programmer's convenience while writing and debugging the microprogram.

Note: Because control memory is read-only and very narrow-purpose, binary microprograms are typically generated automatically by a microassembler rather than hand-encoded, in the same way ordinary assemblers generate machine code.

4. Design of Control Unit

4.1 Basic Requirements of a Control Unit

Whether hardwired or microprogrammed, every control unit must satisfy the same fundamental requirements — it must be able to:

- Generate, for every machine instruction, the correct time-ordered sequence of control signals needed to carry it out.
- Sequence correctly through multi-cycle instructions (fetch, decode, execute, and any indirect-addressing or interrupt-handling steps).
- Respond to status/condition information from the datapath (flags, addressing-mode bits) by altering the sequence of control signals as required (conditional branching).
- Be efficiently and reliably implementable, and ideally be easy to modify or extend when new instructions are added to the instruction set.

Note: It is this last requirement — ease of modification — where microprogrammed control has a decisive advantage over hardwired control: adding or changing an instruction typically only requires editing the contents of the control memory ROM, not redesigning combinational logic circuits.

4.2 Structure of the Control Unit

A microprogrammed control unit is built around three principal hardware elements, working together with the address-sequencing logic already described in Section 2:

Component	Role
Control Memory (CM)	A ROM (or writable control store) holding all microinstructions; addressed by CAR, its output is the current microinstruction.
Control Address Register (CAR)	Holds the address of the microinstruction currently being read from control memory; updated every clock pulse by the next-address logic.
Control Data Register (CDR / pipeline register)	Optionally holds the microinstruction just read from CM, allowing the next address to be computed while the current microinstruction's control signals are still being applied (pipelining the control unit itself).
Next-Address (Sequencing) Logic	Combinational circuit that computes the value to be loaded into CAR next, based on the BR/CD fields of the current microinstruction, status bits, and the mapping logic output.

4.3 Micro-Program Sequencer

The microprogram sequencer is the specific unit within the control unit that implements the next-address logic described above. It typically uses multiplexers to select, on every clock pulse, which of several possible address sources should be loaded into CAR.

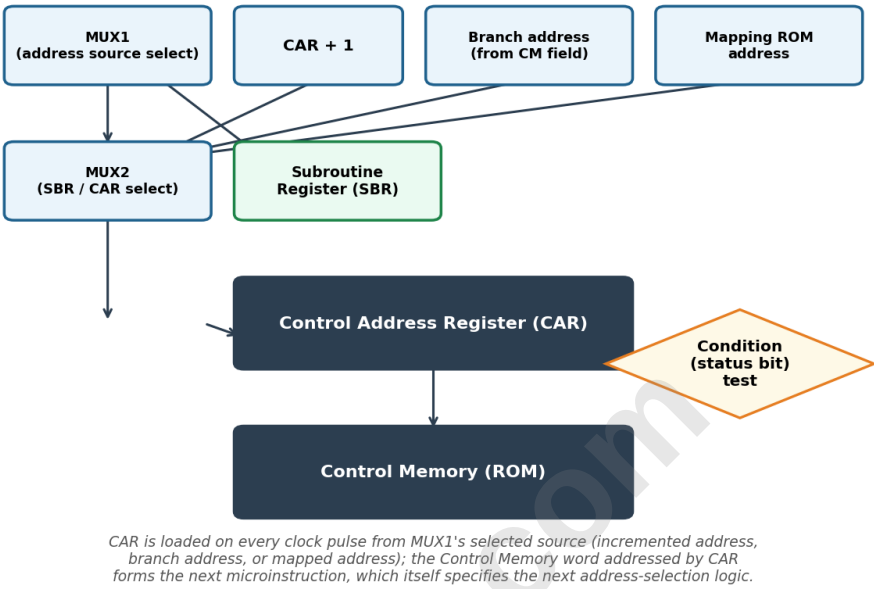


Figure 4.3 — Typical microprogram sequencer structure

Element	Function
MUX1	Selects between an in-page address (CAR + 1) and an out-of-page branch/mapped address, based on the BR field.
MUX2	Selects between the Subroutine Register (SBR) value (used for RET) and other candidate addresses, enabling subroutine call/return within the microprogram.
Subroutine Register (SBR)	Saves the return address when a CALL-type microinstruction is executed, so that a later RET can resume the calling microroutine — the microprogram-level analogue of a machine-level CALL/RET pair.
Condition (status-bit) test logic	Evaluates the selected status bit and produces a single true/false control signal that steers MUX1's selection for conditional branches.

The BR (branch) field of each microinstruction typically encodes one of four basic sequencer operations, closely mirroring ordinary program-control instructions:

BR Code	Operation	Next CAR Value
JMP	Unconditional (or conditional) branch	Branch address field of the microinstruction
CALL	Branch to a microsubroutine, saving return address	Branch address field; CAR+1 saved into SBR

BR Code	Operation	Next CAR Value
RET	Return from microsubroutine	Value previously saved in SBR
MAP	Map opcode of new instruction to its microroutine address	Output of the mapping logic

Example 4.5

Trace the sequencer's behaviour for the fragment of Example 4.3: at FETCH, a CALL to INDRCT is executed, followed eventually by a RET, and then a MAP to begin the next instruction's microroutine.

Solution

1. At FETCH's CALL microinstruction: $SBR \leftarrow CAR + 1$ (return address saved); $CAR \leftarrow$ address of INDRCT.
2. The INDRCT microroutine executes its own microinstructions in sequence (CAR incremented each cycle).
3. At INDRCT's RET microinstruction: $CAR \leftarrow SBR$ (control returns to the instruction right after the original CALL).
4. At the MAP microinstruction: $CAR \leftarrow$ mapping-logic output, i.e., the control-memory address corresponding to the opcode of the newly fetched instruction, and execution of that instruction's microroutine begins.

Key Takeaways

- ✓ The AC register is controlled by explicit LD, INR and CLR signals; the ALU performs arithmetic, logic and shift micro-operations on AC and DR, feeding its result back into AC.
- ✓ Control-memory address sequencing has three sources: sequential increment ($CAR+1$), conditional/unconditional branching, and opcode-to-microroutine mapping.
- ✓ Conditional branching lets one microinstruction test a single status bit and choose between two next addresses, avoiding duplicated hardware paths.
- ✓ A symbolic microprogram is written field-by-field (micro-operations, CD, BR, AD) for readability, then translated into a binary microprogram for storage in control memory — just as assembly language is translated into machine code.
- ✓ A control unit must correctly sequence multi-cycle instructions, respond to status conditions, and (ideally) be easy to extend — microprogrammed control excels at the latter by confining changes to control-memory contents.
- ✓ The microprogram sequencer uses multiplexers (MUX1, MUX2), a Subroutine Register (SBR) and condition-test logic to implement JMP, CALL, RET and MAP next-address operations.

— *End of Unit 4* —

bcalink.com