1.

The Next Address Generator (NAG) is a critical component in a microprogrammed control unit that determines the address of the next microinstruction to be fetched from the control memory. It enables the control unit to manage sequences of microoperations, supporting various control flows such as sequential execution, conditional branching, subroutine calls, and returns.

Functions of the Next Address Generator

The NAG computes the next microinstruction address using:

- Incrementing the current address for sequential execution.
- Conditional branching, based on status flags or ALU outputs.
- Subroutine call/return via a microprogram counter stack.
- Mapping from the opcode field in the instruction register.

Example Control Flow

Consider a microprogrammed control system with the following control memory address sources:

- Address from microinstruction address field.
- Output of a multiplexer (MUX) selecting one of: incremented address, branch address, or return address.

2.

In a microprogrammed control unit, the instruction code can be mapped to a corresponding microinstruction address using a Read-Only Memory (ROM). This approach is widely used in control systems where each instruction in the CPU's instruction set corresponds to a sequence of micro-operations that control various parts of the system, like the ALU, registers, and memory.

How ROM Maps Instruction Code to Microinstruction Address

A ROM-based control unit works by using the opcode (or part of the instruction) as the address to fetch the corresponding microinstruction from the control memory. Here's how this mapping works step-by-step:

4

- Instruction Fetch: The CPU fetches an instruction from memory. The instruction typically contains an opcode (operation code) that specifies which operation needs to be performed (e.g., ADD, SUB, etc.).
- Opcode as Address: The opcode, which is part of the instruction, is used as the address to access the ROM. In some systems, the entire instruction word may be used, or only the opcode field may be used if the rest of the instruction remains constant for a given operation.
- ROM Stores Microinstructions: The ROM contains pre-programmed microinstructions. Each location in the ROM corresponds to a specific operation and contains the sequence of micro-operations required to execute the operation. The number of bits in the ROM address determines the number of possible operations. For Example: The opcode for the ADD instruction might point to address 0x01 in ROM, where the corresponding microinstruction could specify signals to add two numbers in the

ALU, move the result to a register, etc. - Similarly, the opcode for the SUB instruction might point to address 0x02, with its associated microinstruction specifying the necessary control signals for subtraction.

- Microinstruction Fetch: Once the correct address is determined by the opcode, the microinstruction stored at that address is fetched from the ROM. This microinstruction will then be used to set the control lines, instructing the system on how to carry out the operation.

Advantages of Using ROM for Microinstruction Mapping

- Simplicity: Using ROM simplifies the design of the control unit, as the ROM can directly store all the microinstructions associated with various operations. This avoids the need for complex logic circuits.
- Flexibility and Ease of Modification: Microprogramming allows easy modification of control logic by simply updating the ROM. Changing the operation of a certain instruction can be done by rewriting its corresponding

microinstruction in ROM, without needing to modify any hardware components.

- Compactness: The ROM stores the microinstructions in a compact form. Since the microinstructions are retrieved using the opcode, there is no need for a large number of control circuits. Instead, the system can rely on a single ROM to handle all control logic.
- Speed: Since ROM is used for look-up, it is fast, and the access time to fetch a microinstruction is generally very short. This helps to ensure that the control unit can operate at the same speed as the rest of the CPU.

5.

A. Hardwired Control

In a hardwired control unit, control signals are generated by combinational logic circuits composed of gates, flip-flops, decoders, and other hardware components.

7

Characteristics

- Control logic is implemented directly using logic

gates.

- Signals are generated based on the current instruction and state.
- Typically faster since it avoids memory access.
- More complex to design and modify.

Advantages

- High performance: Generates control signals quickly with minimal delay.
- Low latency: No memory fetch needed for microinstructions.
- Efficient for simple or fixed instruction sets.

Disadvantages

- Difficult to design for complex instruction sets.
- Not easily modified or extended——any change requires redesigning the logic.
- Debugging and testing are harder due to complex wiring.

B. Microprogrammed Control

In microprogrammed control units, control signals are generated by executing microinstructions stored in control memory. Each instruction is executed as a sequence of microoperations.

Characteristics

- Uses control memory (like ROM or RAM) to store microinstructions.

- A control address register (CAR) determines the current microinstruction.
- A next address generator handles sequencing (increment, branch, return).
- Easier to implement complex or CISC architectures.

Advantages

- Easier to modify and update: Microcode can be changed or patched.
- Better suited for complex instruction sets (e.g., CISC).
- Supports conditional branching, subroutines, and loops in control logic.

Disadvantages

- Slower than hardwired control due to memory fetch delay.
- Requires more hardware (control memory, address sequencing logic).
- Higher latency in executing control paths.

C. Comparison Table

Aspect	Hardwired Control	Microprogrammed Control
Design Method	Combinational logic	Microinstructions in control memory

Speed Fast Slower due to memory access
Flexibility Low (inflexible) High (easy to modify)
Complexity Difficult for complex ISAs Easier for complex ISAs
Cost Lower for simple systems Higher due to memory needs
Modifiability Difficult Simple (firmware updates possible)
Application RISC architectures CISC architectures, firmware control
D. Hybrid Designs
Some modern systems use hybrid control units. For example, the main instruction decodingmight be hardwired for speed, while complex operations (like floating-point instructions or I/O routines) may be handled by a microprogrammed controller.
Both hardwired and microprogrammed controls have their merits. Hardwired is preferred for speed and simplicity in RISC CPUs, while microprogrammed control offers flexibility and ease of management in complex CISC architectures. The choice depends on performance requirements, design complexity, and flexibility

goals.

7.

1. Direct Mapping

In this scheme, each block of main memory maps to only one possible cache line.

How it works:

- Assume main memory has 2n blocks and cache has 2m lines.
- A block address from main memory is divided into:

— Tag $(n - m)$ bits

— Index (m) bits —— specifies the cache line

- The block is stored only in the cache line corresponding to the index.

Advantages:

- Simple and easy to implement.
- Fast access time due to one-to-one mapping.

Disadvantages:

- High conflict misses —— different blocks may compete for the same line.

Example: Main memory = 64 KB, Cache = 1 KB, block size = 1 byte $\Rightarrow$ Main memory has 216 blocks, cache has 210 lines $\Rightarrow$ Index = 10 bits, Tag = 6 bits

---

2. Associative Mapping

In associative mapping, any block from main memory can be loaded into any cache line.

How it works:

- Entire address is used as tag.
- On access, all cache tags are compared in parallel (associative search).

12

Advantages:

- Flexible — any block can be placed anywhere in cache.
- Minimizes conflict misses.

Disadvantages:

- Hardware-intensive and costly due to parallel tag comparisons.

Example: With 64 KB memory and 1 KB cache, the tag size is 16 bits (entire address), and no index field is needed.

---

3. Set-Associative Mapping

This is a hybrid of direct and associative mapping. The cache is divided into sets, and each set contains multiple lines.

How it works:

- A memory block maps to a specific set, but can occupy any line within that set.
- If cache has 2^s sets and k lines per set (k-way set associative):
  - Index (s bits) to select the set
  - Tag = Total address bits - (s + block offset bits)

Advantages:

- Balances flexibility and simplicity.
- Reduces conflict misses compared to direct mapping.

Disadvantages:

- Slightly more complex than direct mapping.
- Some associative logic required within sets.

Example: Cache = 1 KB, 4-way set associative, block size = 1 byte $\Rightarrow$ Cache has 256 sets (2^8), 8-bit index, and 8-bit tag (for 16-bit address)

13

Comparison Summary

Mapping Type	Flexibility	Hit Ratio	Hardware Cost
Direct Mapping	Low	Moderate	Low

Associative Mapping	High	High	High
Set-Associative Mapping	Medium	High	Medium

The choice of cache mapping technique depends on the system's performance requirements and available hardware. Direct mapping is simple and fast, while associative mapping offers maximum flexibility. Set-associative mapping strikes a practical balance and is widely used in modern processors.

9.

Associative Memory (also known as Content Addressable Memory or CAM) allows data retrieval based on content rather than explicit addresses. This type of memory is particularly useful in cache tag matching, translation lookaside buffers (TLBs), and networking hardware.

Working Principle

Instead of supplying a memory address to retrieve data, a key or pattern is provided. The memory compares the key against its stored contents in all locations simultaneously and

returns matching entries.

15

Multiple Matches

In some scenarios, multiple entries in associative memory may match the input key.

- This is common in cases where wildcard or masked comparisons are allowed.
- Multiple matches can create ambiguity if the system expects a single result.

Resolution Strategies:

1. Priority Encoder: Chooses the match with the highest priority (typically lowest index).
2. Match Lines: A separate match line is activated for each successful match.
3. Multiple Outputs: The system may allow multiple outputs to be processed simultaneously if hardware permits.
4. Parallel Read Logic: If multiple matches are allowed, special logic reads and processes all matched entries in parallel.

Use Cases

- Cache Tag Matching: To identify if the requested memory block exists in the cache.

- TLBs: Associative memory is used to map virtual addresses to physical addresses.
- Networking: CAM is used for high-speed lookup tables in routers and switches.

16

Associative memories are powerful tools for content-based searching. Handling multiple matches effectively is essential for ensuring deterministic system behavior. Using match lines and priority logic ensures reliable output even when multiple candidates match the input pattern.

11.

- RAM (Random Access Memory):
  - Volatile memory (loses data when power is off)
  - Types: SRAM (Static RAM) and DRAM (Dynamic RAM)
  - Read/write operations possible
  - Used for temporary data storage (main memory, cache)
- ROM (Read Only Memory):
  - Non-volatile memory (retains data without

power)
— Types: PROM, EPROM, EEPROM, Flash
— Typically read-only (some can be reprogrammed)
— Used for firmware, BIOS, and permanent storage

Chip Select Functionality

The chip select (CS) signal plays a crucial role in memory organization:

- Enables/disables a specific memory chip in a system with multiple chips
- Determines which chip responds to read/write operations
- Implemented using address decoding logic
- Helps expand memory capacity by combining multiple chip

Address Decoding and Memory Mapping

- Full Decoding:
  — All address lines used for chip selection
  — Each chip occupies unique address range
  — NO address space wasted
- Partial Decoding:
  — Only some address lines used for chip selection
  — Memory chips appear mirrored in address space
  — Simpler but less efficient addressing

Key Considerations

- Chip select signals prevent bus contention

- Proper decoding ensures correct memory access
- Memory expansion requires careful address range assignment
- Modern systems often use memory controllers for this function

13.

Programmed I/O

Programmed I/O is a method of data transfer between the CPU and peripheral devices in which the CPU is responsible for directly managing all input and output operations. In this scheme, the CPU continuously checks (polls) the status of an I/O device to determine whether it is ready for data transfer.

- The CPU issues a read/write command to the I/O module.
- It then enters a busy-wait loop, repeatedly checking the status register of the I/O device.
- Once the device is ready (i.e., sets a flag), the CPU performs the data transfer.
- After the transfer, the CPU resumes the execution of the program.

Advantages:

- Simple and easy to implement.

21

- Full control by the CPU over I/O timing and operations.

Disadvantages:

- Highly inefficient——CPU cycles are wasted while polling.
- Not suitable for multitasking or real-time systems.

Use Case: Suitable for simple embedded systems or when I/O latency is minimal.

Interrupt Initiated I/O

Interrupt initiated I/O improves efficiency by allowing the CPU to execute other tasks while waiting for an I/O device to become ready. In this approach, the I/O device interrupts the CPU when it is ready for data transfer.

- The CPU issues an I/O command and continues with other instructions.
- The I/O device performs the requested operation independently.
- When the operation is complete or the device is ready, it sends an interrupt signal to

the CPU.
- The CPU suspends its current task, saves the context, and invokes an interrupt service routine (ISR) to handle the I/O request.
- After servicing the interrupt, the CPU resumes the suspended task.

Advantages:
- Efficient use of CPU time——no busy-waiting.
- Better suited for multitasking environments.

Disadvantages:
- Slightly more complex to implement.
- Frequent interrupts can lead to context-switching overhead.

Use Case: Widely used in modern systems where responsiveness and efficient CPU utilization are required.

15.

The process of data transfer between an I/O device and the CPU typically follows a standardized interface using control, status, and data lines. One of the fundamental mechanisms for synchronizing this transfer is the use of a flag bit, which signals whether the I/O device

is ready to send or receive data.

Components Involved

- I/O Device: External peripheral that communicates with the CPU (e.g., keyboard, printer).
- I/O Interface: Hardware logic that connects the I/O device to the system bus.
- CPU: Initiates or responds to data transfer requests.
- Control Lines: Carry signals like read/write commands.
- Data Bus: Transfers actual data.
- Status Register with Flag Bit: Indicates the current state of the device (ready or not ready).

Diagram: Data Transfer with Flag Bit

CPU

Control Unit

Registers

I/O Interface

Data Register

Control Register

Status Register

(Includes Flag Bit)
I/O Device
Control Signals
Data (to/from)
Data
Status Update
Procedure for Setting and Clearing the Flag Bit
1. Setting the Flag Bit (Device Ready)
- When the I/O device has data available or is ready for a new command, it sets the flag bit in the status register.
- This flag notifies the CPU that the device is ready for data transfer.
- The CPU periodically checks this bit (polling) or gets interrupted when it's set (interruptdriven).

Example:
- A keyboard buffer gets a new keypress.
- The device sets the Input Ready bit (flag) in the status register.
- CPU detects the flag and reads data from the data register.

2. Clearing the Flag Bit (Data Read/Write Acknowledged)
- Once the CPU reads from or writes to the data

register, it acknowledges the action by clearing the flag bit.

- This can be done automatically by hardware or explicitly by the CPU depending on the system.
- This tells the I/O device that the data has been consumed and it may prepare the next data or enter idle.

17.

1. SISD (Single Instruction, Single Data)

- A single control unit fetches one instruction at a time and applies it to a single data element.
- Conventional sequential architecture.
- Execution: Linear instruction-by-instruction.
- Example: Traditional uniprocessor systems.

2. SIMD (Single Instruction, Multiple Data)

- A single instruction is executed on multiple data elements simultaneously.
- Used in applications with high data parallelism (e.g., matrix operations, image processing).
- Execution: Parallel data-level operations under one control unit.

- Example: GPUs, vector processors.

3. MISD (Multiple Instruction, Single Data)

- Multiple instructions operate on the same data stream.
- Theoretical model; rarely used in practice due to limited applicability.
- Use-case: Fault-tolerant systems where redundant computations improve reliability.
- Example: Spacecraft control systems (hypothetical).

4. MIMD (Multiple Instruction, Multiple Data)

- Each processor executes its own instruction stream on its own data set.
- Highly versatile and widely used in multi-core, multiprocessor, and distributed systems.
- Execution: True parallel processing with independent tasks.
- Example: Cloud servers, supercomputers, parallel computing systems.

Flynn's classification helps in understanding how parallelism is achieved in different computer architectures. Among these, SIMD and MIMD are most commonly used in modern parallel and high-performance computing systems

due to their scalability and efficiency in handling complex computational tasks.

19.

Floating point addition and subtraction are more complex than integer operations due to the need to align exponents, handle sign bits, normalize results, and round them appropriately. These operations benefit significantly from pipelining.

31

Stages of Floating Point Addition/Subtraction Pipeline

The floating-point addition/subtraction operation is typically broken down into the following pipeline stages:

1. Exponent Difference Calculation: Compute the difference between the exponents of the two operands to determine alignment requirements.
2. Mantissa Alignment: Right-shift the mantissa of the operand with the smaller exponent so the exponents match.
3. Mantissa Operation (Add/Subtract): Depending on the signs of the operands, add or subtract the aligned mantissas.

4. Normalization: Normalize the result by shifting the mantissa and adjusting the exponent if needed.
5. Rounding: Round the result to the required precision (based on IEEE 754 standards).
6. Result Write-back: Store the final floating-point result into the registers or memory.

Advantages of Arithmetic Pipelining

- Improves throughput by overlapping operations.
- Allows high-speed floating point computation.
- Efficient use of hardware functional units.
- Reduces latency in repeated arithmetic computations.

21.

Pipeline processing is a technique used to enhance the throughput of arithmetic operations by breaking the operation into multiple stages. The operation $A_i + B_i + C_i$ involves three operands: A_i, B_i, and C_i, and is repeated for $i = 1, 2, 3, \ldots 7. \ldots$,

We assume a three-segment pipeline where each segment performs part of the operation, reducing the overall time required to complete all operations. The three segments in this case could be:

1. Fetch operands: Fetch A_i, B_i, and C_i.
2. Add operands: Perform the addition operation $A_i + B_i$.
3. Finalize result: Add the third operand C_i to the result from the second segment.

Each operation will take 3 clock cycles, and the goal is to determine how many clock cycles are needed to process all 7 operations and visualize the pipeline flow.

Pipeline Operation

Each of the three segments in the pipeline processes different parts of the operation for each index i. As soon as the first operation has completed one full pass through the pipeline, the

next operation can begin. Clock Cycles and Calculation

The pipeline operates in the following manner:

1. At Cycle 1, the first operation A1 + B1 + C1 starts the Fetch stage.
2. At Cycle 2, the second operation A2+B2+C2 enters the pipeline, and the first operation moves to the Add stage.
3. The process continues, with each subsequent operation moving through the pipeline in the next available stage.
4. After the first operation completes all three stages, the next operation is processed in the next clock cycle, and this continues for all 7 operations.

In a 3-segment pipeline: - The first instruction starts at Cycle 1 and completes at Cycle 3. - The second instruction enters the pipeline at Cycle 2 and completes at Cycle 4. - This continues in a staggered fashion, so that after the pipeline is filled, each instruction completes every 1 clock cycle.

Total Clock Cycles Calculation:

For $n = 7$ operations: - The pipeline starts to