5. Execution Flow and Sequencing

This section connects all the previous concepts to illustrate how the microprogrammed control unit *runs* a program. The core of this operation is the mapping of high-level assembly instructions to low-level micro-routines, all managed by the micro-sequencer.

- **5.1 Mapping Assembly to Micro-routines**
  - The most important relationship in this design is the one-to-many mapping between an assembly instruction and its micro-routine. The main CPU's instruction set (the "assembly language") is not executed directly by the hardware; rather, each assembly instruction's opcode is a *command* that tells the micro-sequencer to find and run a specific "subroutine" of micro-instructions stored in the Control Store.
  - **Hypothetical Example:**
    - Consider a CPU with registers `R0`, `R1`, a `PC`, `MAR`, `IR`, and `RAM`.
    - A programmer writes the assembly instruction: `LOAD R0, [0x30]`
    - This translates to the 8-bit machine code `0001 0011` (Opcode `0001` for `LOAD R0`, and Address `0011` for `0x30`).
    - The Control Unit is designed to interpret this `0001` opcode. This single assembly instruction triggers a multi-step micro-routine:

Phase	Micro-operation	Keterangan
Fetch	`uAddr 0:` `PC_out, MAR_in`	(Fetch CC1) Send PC to memory.
	`uAddr 1:` `RAM_out, IR_in`	(Fetch CC2) Get instruction `00010011` into IR.
Decode	`uAddr 1:` (Sequencer Logic)	Sequencer sees `0001`, maps it to `uAddr 16`.
Execute	`uAddr 16:` `IR_addr_out, MAR_in`	(Execute CC1) Get address `0011` from IR.
	`uAddr 17:` `RAM_out, R0_in, PC_inc`	(Execute CC2) Get data from `RAM[0x30]` into `R0`. Increment PC.
	`uAddr 17:` (Sequencer Logic)	Sequencer sees `SEQ_FETCH`, sets `next_uPC = 0`.

- **5.2 The Fetch-Decode-Execute Cycle (Microprogram View)**
  - From the micro-sequencer's perspective, the fetch-decode-execute cycle is a continuous loop of branching within its own Control Store [2].
  - **1. Fetch:** The micro-sequencer is hardwired to always start at a fixed address (e.g., `uAddr 0`) when a new instruction is needed. It then executes the `Fetch` micro-routine (e.g., at `uAddr 0` and `uAddr 1`).

    - **2. Decode:** The final micro-instruction of the `Fetch` routine (at `uAddr 1` in our example) contains a special `SEQ_DECODE` sequencing command. This command tells the micro-sequencer to *stop* incrementing and instead use the `opcode` from the `IR` as an input to its logic. This logic acts as a "mapping ROM," translating the opcode (e.g., `0001`) into the starting address of its corresponding micro-routine (e.g., `uAddr 16`).
    - **3. Execute:** The micro-sequencer *jumps* to that new address (e.g., `uAddr 16`) and executes the micro-routine for the instruction. The final micro-instruction of *that* routine (e.g., at `uAddr 17`) then issues a `SEQ_FETCH` command, which forces the micro-sequencer to jump back to `uAddr 0` and begin the entire process again for the next assembly instruction.
- **5.3 Conditional Branching (Sequencing with Flags)**
    - The true power of a micro-sequencer is revealed in how it handles conditional branching (e.g., `JUMP_IF_ZERO`). This is not implemented with a new micro-instruction, but by adding logic to the **sequencer's `Decode` step**.
    - The `SEQ_DECODE` command is enhanced to not only look at the `opcode` but also at the CPU's `Status Flags` (like `Zero_Flag` and `Carry_Flag`).
    - **Example (Hypothetical `JUMP_ZERO [ADDR]`, Opcode `1010`):**
        - The programmer writes `CMP R0, R1` (which sets the `Zero_Flag`) followed by `JUMP_ZERO 0x05`.
        - When the `JUMP_ZERO` instruction (`1010 0101`) is fetched, the micro-sequencer's `DECODE` logic executes the following:

```
if (opcode == "1010") then
    -- This is a JUMP_ZERO instruction
    if (Zero_Flag == '1') then
        next_uPC = uAddr_JUMP_ROUTINE;  -- Jump to the JUMP micro-routine
    else
        next_uPC = uAddr_NOP_ROUTINE;   -- Jump to the NOP micro-routine (to
just inc PC)
    end if;
else
    -- (handle other opcodes...)
end if;
```

    - In this way, the micro-sequencer dynamically selects the correct micro-routine—either `JMP` or `NOP`—based on the *current state of the CPU flags*, effectively executing the conditional branch.
- **5.4 VHDL Code Example: A Hypothetical Microprogrammed Control Unit**

The following VHDL code is a complete, behavioral model of a hypothetical microprogrammed control unit. This example is simplified to clearly demonstrate the core concepts of sequencing, mapping, and conditional branching.

This code directly illustrates the concepts from sections 5.1, 5.2, and 5.3.

- **Mapping (5.1):** The `init_rom` function shows how opcodes (like `"0001"`) are "mapped" to the starting addresses of micro-routines (like `uAddr 20`).
- **Fetch/Decode/Execute (5.2):** The `next_uPC` process (the Micro-Sequencer) implements this flow.
  - `SEQ_FETCH` (at `uAddr 0, 1`) is the `Fetch` phase.
  - `when SEQ_DECODE =>` is the `Decode` phase, which reads the `OPCODE_IN`.
  - The resulting jump (e.g., `next_uPC <= to_unsigned(20, 8)`) is the start of the `Execute` phase.
- **Conditional Branching (5.3):** The `SEQ_DECODE` block contains the most important logic. It explicitly checks the `Z_FLAG` *in addition* to the `OPCODE_IN` to implement a `JUMP_IF_ZERO` instruction.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Hypothetical_Micro_CU is
    port (
        CLK         : in  std_logic;
        RST         : in  std_logic;

        -- Inputs from Datapath
        OPCODE_IN   : in  std_logic_vector(3 downto 0);
        Z_FLAG      : in  std_logic; -- (for conditional jumps)

        -- Outputs to a hypothetical 10-signal datapath
        PC_OUT      : out std_logic := '0';
        PC_INC      : out std_logic := '0';
        MAR_IN      : out std_logic := '0';
        RAM_OUT     : out std_logic := '0';
        RAM_IN      : out std_logic := '0';
        IR_IN       : out std_logic := '0';
        ACC_IN      : out std_logic := '0';
        ACC_OUT     : out std_logic := '0';
        TEMP_IN     : out std_logic := '0';
        ALU_OUT     : out std_logic := '0'
    );
end entity;

architecture Behavioral of Hypothetical_Micro_CU is

    -- 3.2.2 Micro-Sequencer
```

```
    -- This is the "micro-Program Counter" (uPC)
    signal uPC      : unsigned(7 downto 0) := (others => '0');
    signal next_uPC : unsigned(7 downto 0);

    -- 3.2.3 Micro-instruction Register (uIR)
    -- 10 control bits + 2 sequencing bits
    signal uIR : std_logic_vector(11 downto 0);

    -- 4.2.2 Sequencing Field Constants
    constant SEQ_NEXT   : std_logic_vector(1 downto 0) := "00";
    constant SEQ_DECODE : std_logic_vector(1 downto 0) := "01";
    constant SEQ_FETCH  : std_logic_vector(1 downto 0) := "10";
    constant SEQ_HLT    : std_logic_vector(1 downto 0) := "11";

    -- 4.2 Format and Fields
    constant C_SEQ_BITS  : integer := 2;
    constant C_CTRL_BITS : integer := 10;
    constant C_UCODE_WIDTH : integer := C_SEQ_BITS + C_CTRL_BITS; -- 12 bits

    -- 3.2.1 Control Store (ROM)
    type t_control_store is array(0 to 255) of std_logic_vector(C_UCODE_WIDTH - 1 downto 0);

    -- Helper function to build a micro-instruction
    function to_ucode(
        ctrl : std_logic_vector(C_CTRL_BITS - 1 downto 0);
        seq  : std_logic_vector(C_SEQ_BITS - 1 downto 0)
    ) return std_logic_vector is
    begin
        return seq & ctrl; -- [Seq(1:0), Ctrl(9:0)]
    end function;

    -- Initialize the Control Store (ROM) with micro-routines
    function init_rom return t_control_store is
        variable rom : t_control_store := (others => (others => '0'));

        -- Control Field constants (10 bits)
        -- [PC_OUT, PC_INC, MAR_IN, RAM_OUT, RAM_IN, IR_IN, ACC_IN, ACC_OUT, TEMP_IN, ALU_OUT]
        constant C_FETCH1 : std_logic_vector(9 downto 0) := "1010000000"; -- PC_OUT, MAR_IN
        constant C_FETCH2 : std_logic_vector(9 downto 0) := "0001010000"; -- RAM_OUT, IR_IN
        constant C_LDA1   : std_logic_vector(9 downto 0) := "0010000000"; -- MAR_IN (assume
from IR)
```

```
        constant C_LDA2   : std_logic_vector(9 downto 0) := "0101001000"; -- PC_INC, RAM_OUT,
ACC_IN
        constant C_JUMP1  : std_logic_vector(9 downto 0) := "0000000000"; -- (Assume PCI/IRO
logic)
        constant C_NOP1   : std_logic_vector(9 downto 0) := "0100000000"; -- PC_INC

    begin
        -- 5.2 Fetch Cycle
        rom(0) := to_ucode(C_FETCH1, SEQ_NEXT);   -- uAddr 0
        rom(1) := to_ucode(C_FETCH2, SEQ_DECODE); -- uAddr 1

        -- 5.1 Mapping Assembly to Micro-routines
        -- These are the "Execute" routines

        -- Map Opcode "0001" (LOAD_ACC) to uAddr 20
        rom(20) := to_ucode(C_LDA1, SEQ_NEXT);
        rom(21) := to_ucode(C_LDA2, SEQ_FETCH);

        -- Define targets for conditional branches
        rom(16) := to_ucode(C_NOP1, SEQ_FETCH);  -- uAddr 16 (NOP routine)
        rom(30) := to_ucode(C_JUMP1, SEQ_FETCH); -- uAddr 30 (JUMP routine)

        -- ... the rest of the ROM for other opcodes (ADD, STA, etc.) ...

        return rom;
    end function;

    -- Create the constant ROM
    constant Control_Store : t_control_store := init_rom;

    -- Define the jump targets for conditional branches
    constant uAddr_NOP_ROUTINE : unsigned(7 downto 0) := to_unsigned(16, 8);
    constant uAddr_JMP_ROUTINE : unsigned(7 downto 0) := to_unsigned(30, 8);

begin

    -- 1. Micro-Program Counter (uPC) Register (The "State")
    process(CLK)
    begin
        if rising_edge(CLK) then
            if RST = '1' then
```

```
            uPC <= (others => '0'); -- On reset, go to Fetch (uAddr 0)
        else
            uPC <= next_uPC;
        end if;
    end if;
end process;

-- 2. ROM Read (Concurrent)
uIR <= Control_Store(to_integer(uPC));

-- 3. Micro-Sequencer (Next uPC Logic)
-- This process is the "brain" of the Control Unit
process(uPC, uIR, OPCODE_IN, Z_FLAG)
    variable seq_control : std_logic_vector(1 downto 0);
begin
    seq_control := uIR(C_UCODE_WIDTH - 1 downto C_CTRL_BITS);

    -- Default: always increment
    next_uPC <= uPC + 1;

    case seq_control is
        when SEQ_NEXT =>
            next_uPC <= uPC + 1;

        when SEQ_FETCH =>
            next_uPC <= (others => '0'); -- Go back to Fetch (uAddr 0)

        when SEQ_HLT =>
            next_uPC <= uPC; -- Halt by looping on this address

        when SEQ_DECODE =>
            -- This is the 5.2 Decode step
            -- It maps the opcode to a micro-routine address
            case OPCODE_IN is
                when "0001" => -- LOAD_ACC
                    next_uPC <= to_unsigned(20, 8);

                -- ... other non-branching opcodes (ADD, STA, etc.) ...

                -- 5.3 Conditional Branching Logic
                when "1010" => -- JUMP_IF_ZERO
```

```
                        if Z_FLAG = '1' then
                            next_uPC <= uAddr_JMP_ROUTINE;
                        else
                            next_uPC <= uAddr_NOP_ROUTINE;
                        end if;

                    -- ... other conditional branches (JNE, JLT, etc.) ...

                    when others =>
                        next_uPC <= uAddr_NOP_ROUTINE; -- Invalid opcode
                end case;

            when others =>
                next_uPC <= (others => '0'); -- Safe state
        end case;
    end process;

    -- 4. Output Logic (Concurrent)
    -- Map the 10 control bits from the uIR to the output ports
    PC_OUT  <= uIR(9);
    PC_INC  <= uIR(8);
    MAR_IN  <= uIR(7);
    RAM_OUT <= uIR(6);
    RAM_IN  <= uIR(5);
    IR_IN   <= uIR(4);
    ACC_IN  <= uIR(3);
    ACC_OUT <= uIR(2);
    TEMP_IN <= uIR(1);
    ALU_OUT <= uIR(0);

end Behavioral;
```

Revision #1
Created 2025-11-11 18:13:44 UTC by AX
Updated 2025-11-11 18:25:21 UTC by AX