ECE 495

Computer Engineering Design Laboratory

Lab 7

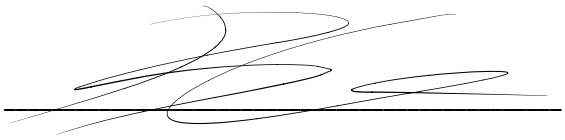
Microprogrammed CPU Design

Group 302

Bryan Galecio
Kevin Galarraga

December 11, 2025

On my honor, I pledge that I have not violated the provisions of the NJIT Academic Honor Code.



Kevin Galarraga

	Grade
Demo: 11/20 Part 1 A, B OK 11/25 Part 2 OK	
Verified: 12/11 #1-6 all OK	
Report:	

OBJECTIVE AND INTRODUCTION

The primary objective of this effort is to implement a microprogrammed CPU design on an FPGA using VHDL, the Library of Parameterized Modules (LPM), and structural modeling techniques. **Figure 1** shows a high-level block diagram illustrating how the FPGA will be used to realize the microprogrammed CPU.

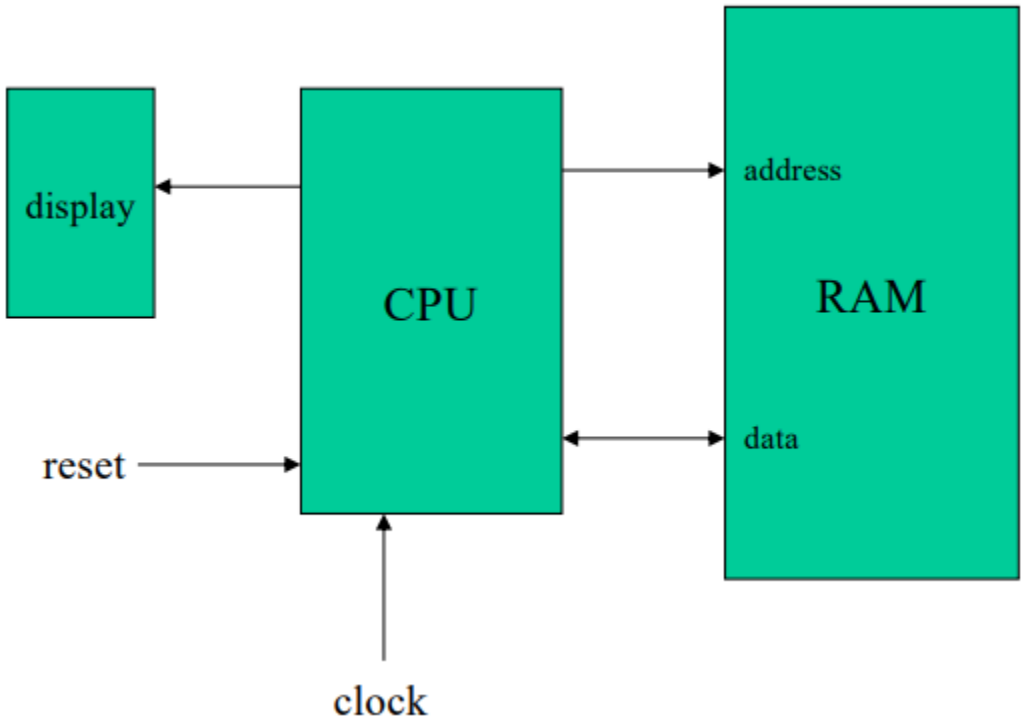


Figure 1: High-level block diagram showing how the CPU connects to an I/O device (display), and how the address and data lines interface with RAM, unidirectionally for the address bus and bidirectionally for the data bus, to fetch program instructions.

Processor control units can be designed in several ways, such as through hardwired control logic. However, a microprogrammed control unit offers significantly greater flexibility, easier debugging, simpler modification, and reduced design complexity. By storing control information in memory rather than embedding it in fixed hardware, a microprogrammed CPU enables a cleaner flow of control signals and is generally more adaptable to changes or additions in the instruction set.

The Library of Parameterized Modules provides the essential building blocks for the CPU design, including Read-Only Memory (ROM), Random-Access Memory (RAM), D flip-flops, Multiplexers, ALUs, and Counters. Because these modules are parameterized, their behavior and size can be adjusted during instantiation. This allows us to customize data widths, enable or disable optional features, and tailor each component to the specific needs of our architecture.

At the core of the microprogrammed design is the microsequencer, a component not included in the LPM library. Thus, before implementing the full CPU, we first designed a functional microsequencer capable of supplying the CPU with programmed control signals. **Figure 2** shows the block diagram of the microsequencer. The instruction register provides the opcode to the microsequencer, which uses this opcode, along with mapping bits, to access the starting address of the instruction's micro-operation in the ROM. From there, the microsequencer advances through consecutive micro-operations until the instruction is complete. **Figure 3** illustrates the ROM output format used by the microsequencer. Because the address line is 8 bits wide, the ROM contains 2^8 address locations. Each ROM word encodes the CPU control signals, a mapping bit, and the next address (from MSB to LSB), with the specific arrangement determined by the CPU design.

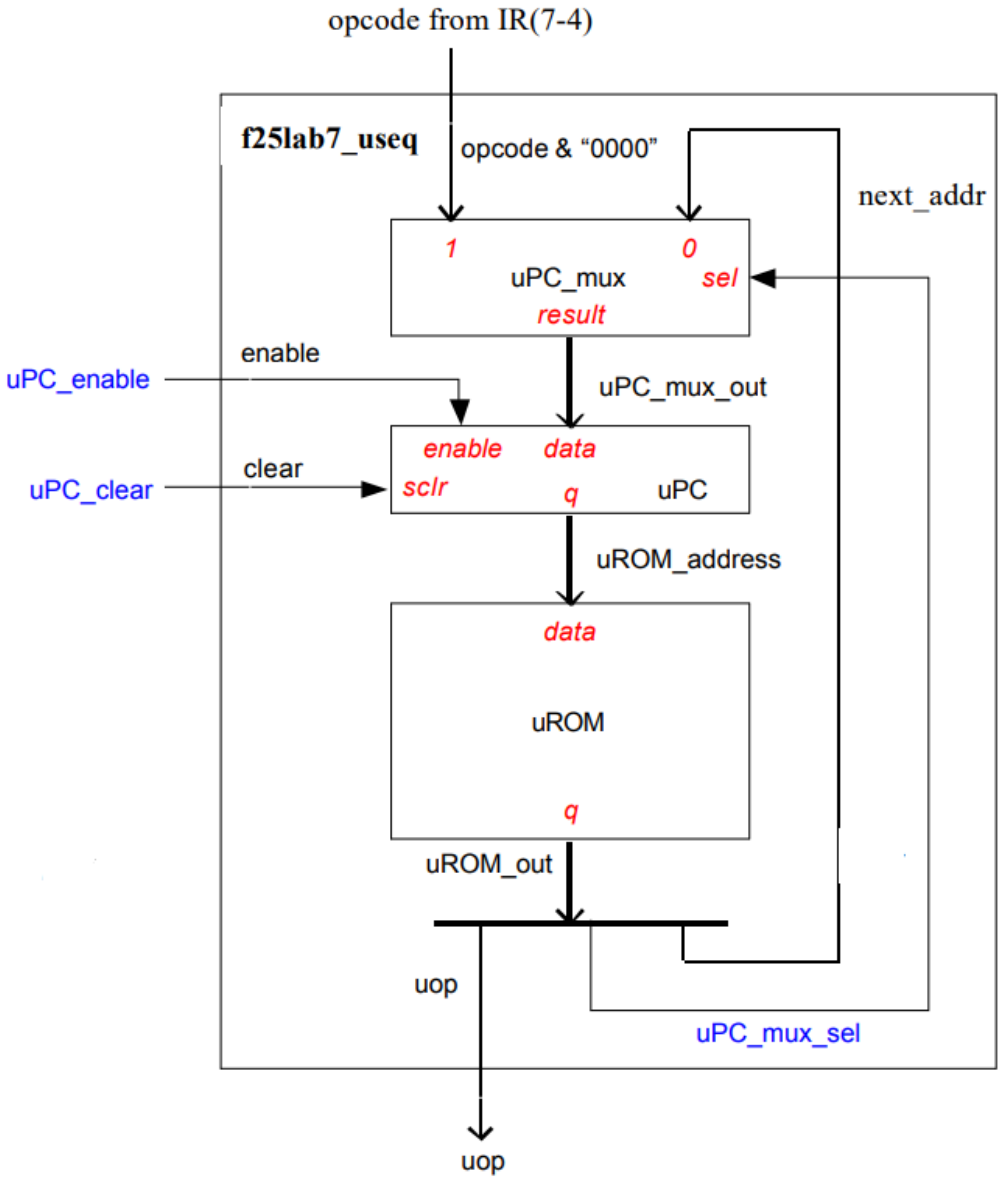


Figure 2: Block diagram of the microsequencer.

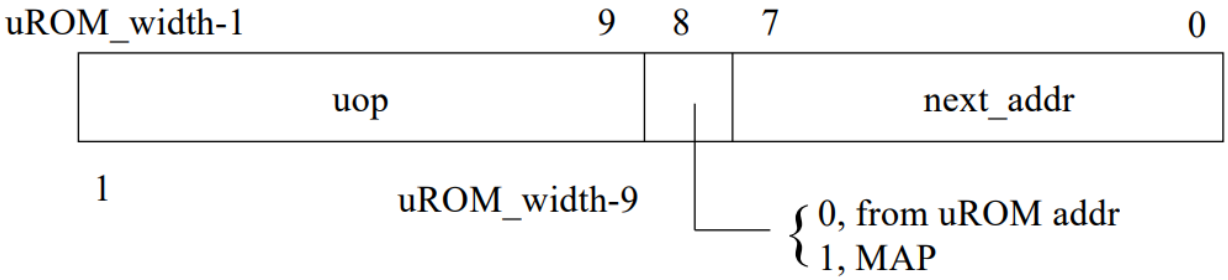


Figure 3: ROM output format used by the microsequencer.

With the microsequencer defined, we then needed a memory system capable of holding the program instructions. For this lab, we implemented a RAM block that includes address and data registers. This RAM stores the sequence of instructions that the CPU will execute during its fetch and execute cycles. The contents of RAM must adhere to the instruction set architecture (ISA) defined in **Figure 4**, which lists each instruction, its opcode, and its intended operation. **Figure 5** provides a block diagram of the RAM subsystem.

Instruction	**Instruction code**	**Operation**
NOP	00000000	No operation
LOADI Rn,X	0001000n X	Rn ← X
LOAD Rn,X	0010000n X	Rn ← M[X]
STORE X,Rm	0011000m X	M[X] ← Rm
MOVE Rn,Rm	0100000n	Rn ← Rm, m ≠ n
ADD Rn,Rm	0101000n	Rn ← Rn + Rm, m ≠ n
SUB Rn,Rm	0110000n	Rn ← Rn – Rm, m ≠ n
TESTNZ Rm	0111000m	Z ← not V, V = OR of the bits of Rm
TESTZ Rm	1000000m	Z ← V, V = OR of the bits of Rm
JUMP X	10010000 X	PC ← X
JUMPZ X	10100000 X	If (Z = 1) then PC ← X
LOADSP X	10110000 X	SP ← X
PEEP Rn	1100000n	Rn ← M[SP]
PUSH Rn	1101000n	M[--SP] ← Rn Pre-increment
POP Rn	1110000n	Rn ← M[SP++] Post-decrement
HALT	11110000	PC ← 0, stop microsequencer

Figure 4: Instruction set table showing the 16 CPU instructions, their instruction codes, and their corresponding operations.

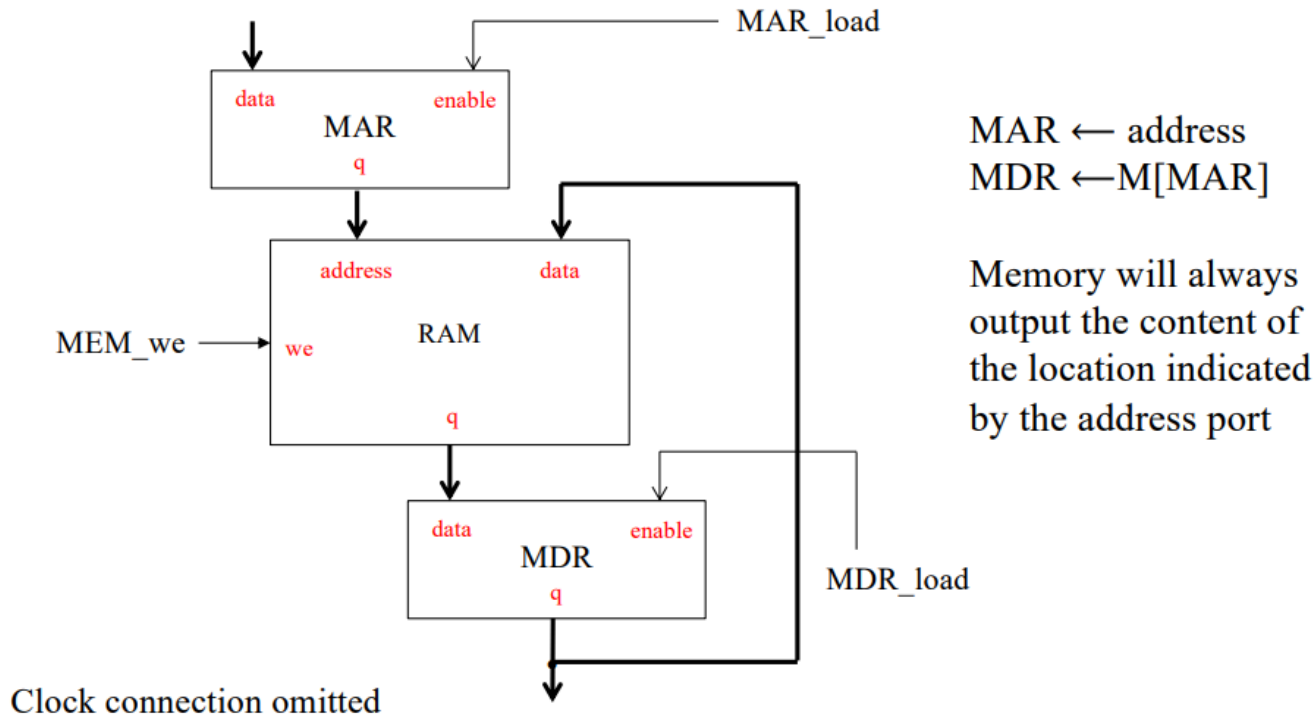


Figure 5: Block diagram of the RAM subsystem and its connections.

Having established the foundational components, we next considered how binary control signals (1s and 0s) orchestrate CPU behavior during instruction execution. Each instruction consists of a sequence of micro-operations that are activated by specific control signals. To determine these control signals and their order, we rely on Register Transfer Language (RTL). RTL provides a clear abstraction of the data transfers and operations that occur in each cycle of an instruction, and serves as the most direct guide for generating the corresponding microinstructions.

With the hardware requirements, microsequencer design, ISA definition, and RTL descriptions in place, we are prepared to develop the complete microprogrammed CPU. The design, RTL code, and memory structures will be implemented in VHDL and synthesized onto the *Altera Cyclone® IV EP4CE115F29C7* FPGA using *Intel® Quartus® II*.

THEORY AND METHODS

Based on the microsequencer block diagram shown in **Figure 2**, we designed a VHDL program that synthesizes its functionality using components from the Library of Parameterized Modules (LPM), including multiplexers, D flip-flops, and ROM. The micro-operation (UOP) size was set to the ROM output size minus 9, since 8 bits encode the next address and 1 bit represents the internal mapping signal. The resulting VHDL program is parametrized, allowing it to be instantiated as a reusable component for the microprogrammed CPU design, as shown in **Figure 6**.

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
 );
end uSequencer;

architecture structural of uSequencer is
signal uROM_address: std_logic_vector (7 downto 0);
```

```
signal uROM_out: std_logic_vector (uROM_width-1 downto 0);
signal uPC_mux_data: std_logic_2D(1 downto 0, 7 downto 0);
signal uPC_mux_sel: std_logic_vector(0 to 0);
signal uPC_mux_out: std_logic_vector(7 downto 0);
signal temp: std_logic_vector(7 downto 0);

begin
 temp <= opcode & "0000";

 for_label: for i in 0 to 7 generate
    uPC_mux_data(0, i) <= uROM_out(i);
    uPC_mux_data(1, i) <= temp(i);
 end generate;

 uPC_mux_sel(0) <= uROM_out(8);

 uPC_mux: lpm_mux
        generic map (lpm_width=>8, lpm_size=>2, lpm_widths=>1)
        port map (result=>uPC_mux_out, data=>uPC_mux_data, sel=>uPC_mux_sel);
 uPC: lpm_ff
        generic map (lpm_width=>8)
        port map (clock=>clock, data=>uPC_mux_out, q=>uROM_address, sclr=>clear,
enable=>enable);
 uROM: lpm_rom
        generic map (lpm_widthad=>8, lpm_width=>uROM_width, lpm_file=>uROM_file)
        port map (address=>uROM_address, q=>uROM_out, inclock=>clock,
outclock=>clock);

 uop <= uROM_out(uROM_width-1 downto 9);
 debug_map_addr <= uROM_out(8 downto 0);
```

```
end structural;
```

Figure 6: Parameterized VHDL program of the microsequencer.

To verify the microsequencer, we tested it in two separate VHDL programs (**Figures 7** and **8**). Because an instruction register was not yet implemented, we hardwired the opcode 0011 and used a memory initialization file (.mif) in *Intel Quartus® II* to load the ROM. In the program of **Figure 7**, a counter enables the microsequencer every four clock pulses, advancing through addresses until the mapping signal is set high. The ROM then transitions to address "00110000" (0x30 in hex or 48 in decimal), where the microsequencer continues until cleared, as the mapping signal is no longer declared.

In **Figure 8**, we added a register representing the CPU's memory address reference point. This setup tests the microsequencer's ability to correctly set control signals in conjunction with a pseudo instruction. The corresponding .mif files for these tests are shown in **Figures 9** and **10**. At address 0x30, control signals are activated to load and increment the program counter (PC), which we confirmed using a seven-segment display component developed in previous labs.

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity test_uSequencer is
	port(
		button, clear: in std_logic;
		useqEnOut: out std_logic;
		ctrlSignals: out std_logic_vector(0 to 7);
		M_disp, disp1, disp0: out std_logic_vector(0 to 6)
```

```
        );
end test_uSequencer;

architecture structural of test_uSequencer is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
 );
end component;

component seven_segment_display is
        port(
                digit: in std_logic_vector(3 downto 0);
                display: out std_logic_vector(0 to 6)
        );
end component;

signal useqEnable, useqClear, clk: std_logic;
signal uop: std_logic_vector(0 to 7);
signal address_out: std_logic_vector(8 downto 0);
signal M, addr_hex1, addr_hex0: std_logic_vector(3 downto 0);
begin
```

```
	clk <= not(button);
	delay: lpm_counter generic map(lpm_width=>2) port map(clock=>clk,
cout=>useqEnable);

	useqClear <= clear;
	useqEnOut <= useqEnable;

	labelG: for i in 0 to 7 generate
		ctrlSignals(i) <= uop(i) and useqEnable;
	end generate;

	uSeq: uSequencer
		generic map(uROM_width=>17, uROM_file=>"test_uSequencer_file.mif")
		port map(opcode=>"0011", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);



	M <= "000" & address_out(8);
	addr_hex1 <= address_out(7 downto 4);
	addr_hex0 <= address_out(3 downto 0);
	display1: seven_segment_display port map(digit=>M, display=>M_disp);
	display2: seven_segment_display port map(digit=>addr_hex1, display=>disp1);
	display3: seven_segment_display port map(digit=>addr_hex0, display=>disp0);
end structural;
```

Figure 7: VHDL test program for the microsequencer using a counter and hardwired opcode.

```
library ieee;
use ieee.std_logic_1164.all;
```

```
library lpm;
use lpm.lpm_components.all;

entity test_uSequencer_PC is
	port(
		button, clear: in std_logic;
		useqEnOut: out std_logic;
		ctrlSignals: out std_logic_vector(0 to 7);
		pc_Seg1, pc_Seg0, M_disp, disp1, disp0: out std_logic_vector(0 to 6)
	);
end test_uSequencer_PC;

architecture structural of test_uSequencer_PC is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
 );
end component;

component seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
```

```
                display: out std_logic_vector(0 to 6)
        );
end component;

signal useqEnable, useqClear, clk: std_logic;
signal pcInc, pcLoad, pcClear: std_logic;
signal pcOut: std_logic_vector(7 downto 0);
signal uop: std_logic_vector(0 to 7);
signal address_out: std_logic_vector(8 downto 0);
signal M, addr_hex1, addr_hex0, pc_hex1, pc_hex0: std_logic_vector(3 downto 0);
begin
        clk <= not(button);
        delay: lpm_counter
                generic map(lpm_width=>2)
                port map(clock=>clk, cout=>useqEnable);

        useqClear <= clear;
        useqEnOut <= useqEnable;

        labelG: for i in 0 to 7 generate
                ctrlSignals(i) <= uop(i) and useqEnable;
        end generate;

        pcInc <= uop(0) and useqEnable;
        pcLoad <= uop(1) and useqEnable;
        pcClear <= uop(2) and useqEnable;

        uSeq: uSequencer
                generic map(uROM_width=>17,
uROM_file=>"test_uSequencer_PC_file.mif")
```

```
        port map(opcode=>"0011", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);

    pc: lpm_counter
        generic map(lpm_width=>8)
        port map(sload=>pcLoad, data=> "10101010", q=>pcOut, cnt_en=>pcInc,
sclr=>pcClear, clock=>clk);

    M <= "000" & address_out(8);
    addr_hex1 <= address_out(7 downto 4);
    addr_hex0 <= address_out(3 downto 0);
    pc_hex1 <= pcOut(7 downto 4);
    pc_hex0 <= pcOut(3 downto 0);
    display1: seven_segment_display port map(digit=>M, display=>M_disp);
    display2: seven_segment_display port map(digit=>addr_hex1, display=>disp1);
    display3: seven_segment_display port map(digit=>addr_hex0, display=>disp0);
    display4: seven_segment_display port map(digit=>pc_hex1, display=>pc_Seg1);
    display5: seven_segment_display port map(digit=>pc_hex0, display=>pc_Seg0);

end structural;
```

Figure 8: VHDL test program for the microsequencer, including a pseudo instruction register.

```
1   -- uROM Test File
2   -- 8-bit control signals
3   WIDTH=17;
4   DEPTH=256;
5   ADDRESS_RADIX=HEX;
6   DATA_RADIX=BIN;
7   CONTENT BEGIN
8    [0..FF]: 00000000000000000;
9   --   uop
10  --   01234567M01234567
11   00: 00000000000000001; -- control signal = 00000000, next address = 0x01
12   01: 00000001000000010; -- control signal = 00000001, next address = 0x02
13   02: 00000010000000011; -- control signal = 00000010, next address = 0x03
14   03: 00000011000000100; -- control signal = 00000011, next address = 0x04
15   04: 00000100100000000; -- control signal = 00000100, use MAP, go to address 0x30
16   30: 00000101000110001; -- control signal = 00000101, next address = 0x31
17   31: 00000110000110010; -- control signal = 00000110, next address = 0x32
18   32: 00000111000110011; -- control signal = 00000111, next address = 0x33
19   33: 00001000000000001; -- control signal = 00001000, next address = 0x01
20  END;
```

Figure 9: Memory initialization file (.mif) for the microsequencer test in **Figure 7**.

```
1   -- uROM Test File
2   -- 8-bit control signals
3   WIDTH=17;
4   DEPTH=256;
5   ADDRESS_RADIX=HEX;
6   DATA_RADIX=BIN;
7   CONTENT BEGIN
8    [0..FF]: 00000000000000000;
9   --   uop
10  --   01234567M01234567
11   00: 00000000000000001; -- control signal = 00000000, next address = 0x01
12   01: 00100001000000010; -- control signal = 00100001, next address = 0x02, clear PC, PC = 0x00
13   02: 10000010000000011; -- control signal = 10000010, next address = 0x03, inc PC, PC = 0x01
14   03: 10000011000000100; -- control signal = 10000011, next address = 0x04, inc PC, PC = 0x02
15   04: 00000100100000000; -- control signal = 00000100, use MAP, go to address 0x30
16   30: 01000101000110001; -- control signal = 01000101, next address = 0x31, load PC, PC = 0xAA
17   31: 00000110000110010; -- control signal = 00000110, next address = 0x32
18   32: 10000111000110011; -- control signal = 10000111, next address = 0x33, inc PC, PC = 0xAB
19   33: 00001000000000001; -- control signal = 00001000, next address = 0x01,
20  END;
```

Figure 10: Memory initialization file (.mif) for the microsequencer test in **Figure 8**.

In addition to the microsequencer, the RAM subsystem is a critical component for instruction storage and execution. **Figure 11** shows the VHDL implementation of the RAM subsystem. We tested the RAM using initialized ROM and RAM .mif files: the ROM .mif provided control signals to load the MAR and MDR registers, while the RAM .mif verified data at specific address locations. These .mif files are illustrated in **Figure 12**.

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity test_ram is
	port(
		button, clear: in std_logic;
		useqEnOut: out std_logic;
		marData: in std_logic_vector(7 downto 0); -- connected to switches, input to MAR
		marSegHi, marSegLo, mdrSegHi, mdrSegLo, nextAddrHi, nextAddrLo: out std_logic_vector(0 to 6);
		ctrlSignals: out std_logic_vector(0 to 7)
	);
end test_ram;

architecture structural of test_ram is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
```

```
 );
end component;

component seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
		display: out std_logic_vector(0 to 6)
	);
end component;

signal clk, useqEnable, useqClear, marLoad, mdrLoad, memWE: std_logic;
signal uop: std_logic_vector(0 to 7);
signal marADig, marBDig, mdrADig, mdrBDig, nextAADig, nextABDig: std_logic_vector(3
downto 0);
signal marOut, memOut, mdrOut: std_logic_vector(7 downto 0);
signal address_out: std_logic_vector(8 downto 0);
begin
	clk <= not(button);
	useqClear <= clear;
	useqEnOut <= useqEnable;
	marLoad <= uop(0) and useqEnable;
	mdrLoad <= uop(1) and useqEnable;
	memWE <= uop(2) and useqEnable;

	marADig <= marOut(7 downto 4);
	marBDig <= marOut(3 downto 0);
	mdrADig <= mdrOut(7 downto 4);
	mdrBDig <= mdrOut(3 downto 0);
	nextAADig <= address_out(7 downto 4);
	nextABDig <= address_out(3 downto 0);
```

```
	labelG: for i in 0 to 7 generate
		ctrlSignals(i) <= uop(i) and useqEnable;
	end generate;

	delay: lpm_counter generic map(lpm_width=>2) port map(clock=>clk,
cout=>useqEnable);
	microSequencer: uSequencer
		generic map(uROM_width=>17, uROM_file=>"test_rom_file.mif")
		port map(opcode=>"0000", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);
	marReg: lpm_ff
		generic map(lpm_width=>8)
		port map(data=>marData, q=>marOut, clock=>clk, enable=>marLoad);
	mdrReg: lpm_ff
		generic map(lpm_width=>8)
		port map(data=>memOut, q=>mdrOut, clock=>clk, enable=>mdrLoad);
	ram: lpm_ram_dq
		generic map(lpm_widthad=>8, lpm_width=>8,
lpm_file=>"test_ram_file.mif")
		port map(data=>mdrOut, address=>marOut, q=>memOut, inclock=>clk,
outclock=>clk, we=>memWE);
	marA: seven_segment_display port map(digit=>marADig, display=>marSegHi);
	marB: seven_segment_display port map(digit=>marBDig, display=>marSegLo);
	mdrA: seven_segment_display port map(digit=>mdrADig, display=>mdrSegHi);
	mdrB: seven_segment_display port map(digit=>mdrBDig, display=>mdrSegLo);
	nextAddressA: seven_segment_display port map(digit=>nextAADig,
display=>nextAddrHi);
	nextAddressB: seven_segment_display port map(digit=>nextABDig,
display=>nextAddrLo);
```

```
end structural;
```

Figure 11: VHDL implementation of the RAM subsystem and its connections.

```
-- uROM Test File
-- 8-bit control signals

WIDTH=17;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
   [0..FF]: 00000000000000000;
--     uop
--     01234567M01234567
   00: 00000000000000001; -- control signal (cs) = 00000000, next address (na) = 0x01
   01: 10000001000000010; -- cs = 10000001, na = 0x02, Load MAR from sw, MAR = 0x54
   02: 01000010000000011; -- cs = 01000010, na = 0x03, MDR <- M[MAR], MDR = 0xFC
   03: 10000011000000100; -- cs = 10000011, na = 0x04, Load MAR from sw, MAR = 0x55
   04: 01000100000000101; -- cs = 01000100, na = 0x05, MDR <- M[MAR], MDR = 0xDD
   05: 10000101000000110; -- cs = 10000101, na = 0x06, Load MAR from sw, MAR = 0x56
   06: 00100110000000111; -- cs = 00100110, na = 0x07, M[MAR] <- MDR
   07: 10000111000001000; -- cs = 10000111, na = 0x08, Load MAR from sw, MAR = 0X54
   08: 01001000000001001; -- cs = 01001000, na = 0x09, MDR <- M[MAR], MDR = 0xFC
   09: 10001001000001010; -- cs = 10001001, na = 0x0A, Load MAR from sw, MAR = 0x56
   0A: 01001010000000000; -- cs = 01001010, na = 0x00, MDR <- M[MAR], MDR = 0xDD
END;
```

```
-- RAM Test File

WIDTH=8;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
   [0..FF]: 0;
   54: FC;
   55: DD;
   56: AA;
END;
```

Figure 12: .mif files for testing RAM and microsequencer interactions.

Once the basic register, microsequencer, and RAM subsystems were implemented, we developed Register Transfer Level (RTL) statements for each CPU instruction. These RTL

statements guided the construction of a full CPU block diagram and the creation of a ROM file containing the micro-operations for each instruction. **Figures 13**, **14**, and **15** show the RTL statements, the CPU block diagram, and the ROM .mif file exported from a spreadsheet, respectively, ensuring consistency with the RTL logic.

```
# Develop the RTL statements for the CPU instructions:

# All instructions do Instruction Fetch:
F1: MAR ← PC # Load address of instruction
F2: MDR ← M[MAR], PC ← PC + 1 # Fetch instruction and increment PC
F3: IR ← MDR # Decode instruction

nop                     00000000
- None -

loadi Rn, X            0001000n X
LI1: MAR ← PC    # Load memory address of immediate operand
LI2: MDR ← M[MAR], PC ← PC + 1   # Fetch immediate operand and increment PC
LI3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1     # Load immediate value into R0 if
IR(0) = 0 else if IR(0) = 1 into R1

load Rn, X             0010000n X
L1: MAR ← PC    # Load memory address of operand
L2: MDR ← M[MAR], PC ← PC + 1   # Fetch operand address and increment PC
L3: MAR ← MDR   # Load effective address
L4: MDR ← M[MAR]    # Fetch data from memory
L5: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1    # Load data into R0 if IR(0) = 0 else if
IR(0) = 1 load into R1

store X, Rn            0011000n X
```

```
ST1: MAR ← PC    # Load memory address of operand
ST2: MDR ← M[MAR], PC ← PC + 1   # Fetch operand address and increment PC
ST3: MAR ← MDR   # Load effective address
ST4: MDR ← R0 if IR(0) = 0, MDR ← R1 if IR(0) = 1    # Load data from R0 if IR(0) = 0 else if IR(0) = 1 load from R1
ST5: M[MAR] ← MDR    # Store data into memory

move Rn, Rm              0100000n m!=n
M1: R0 ← R1 if IR(0) = 0, R1 ← R0 if IR(0) = 1    # Move value from R1 to R0 if IR(0) = 0 else if IR(0) = 1 move from R0 to R1

add Rn, Rm               0101000n m!=n
A1: R0 ← R0 + R1 if IR(0) = 0, R1 ← R0 + R1 if IR(0) = 1    # R0 plus R1 if IR(0) = 0 else if IR(0) = 1 R1 plus R0

sub Rn, Rm               0110000n m!=n
S1: R0 ← R0 - R1 if IR(0) = 0, R1 ← R1 - R0  if IR(0) = 1   # R0 minus R1 if IR(0) = 0 else if IR(0) = 1 R1 minus R0

testnz Rn                0111000n
TN1: Z ← NOT(R0(0) OR ... OR R0(7)) if IR(0) = 0, ← NOT(R1(0) OR ... OR R1(7)) if IR(0) = 1    # Z = 1 if R0 or R1 is nonzero

testz Rn                 1000000n
T1: Z ← R0(0) OR ... OR R0(7) if IR(0) = 0, ← R1(0) OR ... OR R1(7) if IR(0) = 1    # Z = 1 if R0 or R1 is nonzero

jump X                   10010000 X
J1: MAR ← PC    # Load memory address of jump target
J2: MDR ← M[MAR]   # Fetch jump target address
```

```
J3: PC ← MDR    # Jump to target address

jumpz X                 10100000 X
JZ1: MAR ← PC # Load memory address of jump target
JZ2: MDR ← M[MAR], PC <- PC + 1   # Fetch jump target address and increment PC
JZ3: PC ← MDR if Z=1   # Jump if Z flag is set

loadsp X                10110000 X
LS1: MAR ← PC   # Load memory address of operand
LS2: MDR ← M[MAR], PC ← PC + 1  # Fetch operand and increment PC
LS3: SP ← MDR   # Load operand into SP

peep Rn                 1100000n
P1: MAR ← SP    # Load address from stack pointer
P2: MDR ← M[MAR]   # Fetch value from stack
P3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1    # Peek value from stack into R0 if
IR(0) = 0 else if IR(0) = 1 into R1

push Rn                 1101000n
PU1: SP ← SP - 1  # Decrement stack pointer
PU2: MAR ← SP   # Load address from stack pointer
PU3: MDR ← R0 if IR(0) = 0, MDR ← R1 if IR(0) = 1   # Load value from R0 if IR(0) = 0
else if IR(0) = 1 from R1
PU4: M[MAR] ← MDR   # Push value onto stack

pop Rn                  1110000n
PO1: MAR ← SP   # Load address from stack pointer
PO2: MDR ← M[MAR]   # Fetch value from stack
PO3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1   # Load value into R0 if IR(0) = 0 else
if IR(0) = 1 into R1
```

```
PO4: SP ← SP + 1   # Increment stack pointer


halt                        11110000
H1: PC ← 0  # Stop execution by resetting PC
```

Figure 13: RTL statements for the CPU instructions.

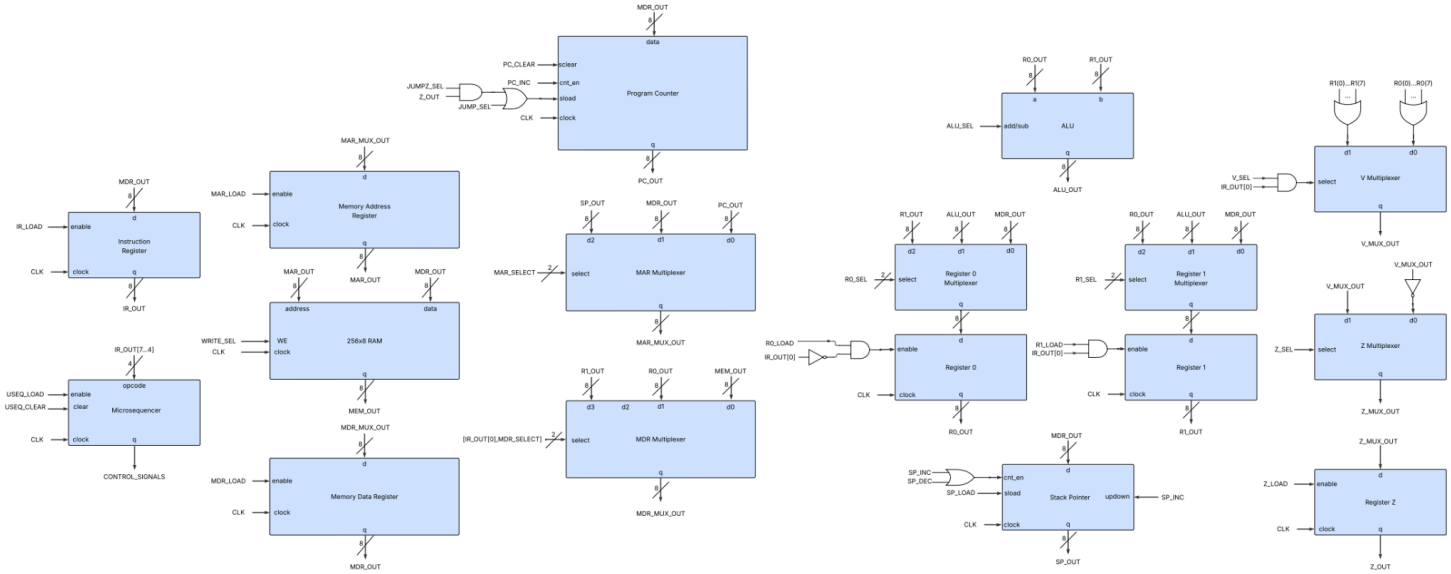


Figure 14: Block diagram of the complete microprogrammed CPU design

Microsequencer ROM → ROM Memory Initialization File ↓

```
WIDTH=33;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

-- Current Address: UOP(1-24)MAP(8)NEXTADDRESS(7-0)
-- 0xADDRESS: 123456789(10)(11)(12)(13)(14)(15)(16)(17)(18)(19)(20)(21)(22)(23)(24)876543210

CONTENT BEGIN
   [0..FF]: 000000000000000000000000000000000;
   00: 000000000000000000000000000000001;
   01: 000000010000000000000000000000010;
   02: 010000001000000000000000000000011;
   03: 001000000000000000000000100000001;
   10: 000000100000000000000000000010001;
   11: 010000001000000000000000000010010;
   12: 000000000001001000000000000000001;
   20: 000000100000000000000000000100001;
   21: 010000001000000000000000000100010;
   22: 000001100000000000000000000100011;
   23: 000000001000000000000000000100100;
   24: 000000000001001000000000000000001;
   30: 000000100000000000000000000110001;
   31: 010000001000000000000000000110010;
   32: 000101100000000000000000000110011;
   33: 000000011000000000000000000110100;
   34: 000100000000000000000000000000001;
   40: 000000000101101000000000000000001;
   50: 000000000011011100000000000000001;
   60: 000000000011011000000000000000001;
   70: 000000000000000010100000000000001;
   80: 000000000000000011100000000000001;
   90: 000000100000000000000000000010001;
   91: 000000001000000000000000010010010;
   92: 000000000000000000010000000000001;
   A0: 000000100000000000000000010100001;
   A1: 010000001000000000000000000010100010;
   A2: 000000000000000000010000000000001;
   B0: 000000100000000000000000000010110001;
   B1: 010000001000000000000000000010110010;
   B2: 000000000000000000000100000000001;
   C0: 000010100000000000000000000011000001;
   C1: 000000001000000000000000000011000010;
   C2: 000000000001001000000000000000000001;
   D0: 000000000000000000000000001011010001;
   D1: 000010100000000000000000000011010010;
   D2: 000000011000000000000000000011010011;
   D3: 000100000000000000000000000000000001;
   E0: 000010100000000000000000000011100001;
   E1: 000000001000000000000000000011100010;
   E2: 000000000001001000000000000011100011;
   E3: 000000000000000000000000100000000001;
   F0: 100000000000000000000000000000000001;
END;
```

Figure 15: ROM .mif file containing micro-operations for each instruction.

With all components and the ROM initialization file prepared, the final step was synthesizing the microprogrammed CPU on the FPGA. **Figure 16** shows the VHDL implementation of the complete CPU design, which employs structural modeling by instantiating all components with appropriate sizes, pins, control signals, processes, and loops according to the block diagram. A seven-segment display was included to visualize critical values such as register contents, the stack pointer, and the PC.

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
```

```
use lpm.lpm_components.all;

entity cpu is
	port(
		button, clear: in std_logic;
		Z_FLAG: out std_logic_vector(0 to 0);
		R0SegHi, R0SegLo, R1SegHi, R1SegLo, pcSegHi, pcSegLo, spSegHi, spSegLo:
out std_logic_vector(0 to 6)
	);
end cpu;

architecture structural of cpu is
component uSequencer is
	generic(
		uROM_width: integer := 10;
		uROM_file: string := ""
	);
	port(
		opcode: in std_logic_vector(3 downto 0);
		uop: out std_logic_vector(1 to (uROM_width-9));
		debug_map_addr: out std_logic_vector(8 downto 0);
		enable, clear: in std_logic;
		clock: in std_logic
	);
end component;

component seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
		display: out std_logic_vector(0 to 6)
```

```
        );
end component;

-- CLOCK / CONTROL SIGNALS
signal CLOCK, useqEnable, useqClear: std_logic;
signal PC_LOAD, PC_CLEAR, PC_INC, IR_LOAD, WRITE_SEL, MAR_LOAD, MDR_SEL,
MDR_LOAD: std_logic;
signal R0_ENABLE, R0_LOAD, R1_ENABLE, R1_LOAD, ALU_SEL, Z_LOAD, JUMP_SEL,
JUMPZ_SEL: std_logic;
signal SP_ENABLE, SP_LOAD, SP_INC, SP_DEC, M: std_logic;

-- DATA PATH SIGNALS (8-bit registers / ALU)
signal PC_OUT, IR_OUT, MAR_OUT, MEM_OUT, MDR_OUT: std_logic_vector(7 downto 0);
signal MAR_MUX_OUT, MDR_MUX_OUT: std_logic_vector(7 downto 0);
signal R0_OUT, R0_MUX_OUT, R1_OUT, R1_MUX_OUT: std_logic_vector(7 downto 0);
signal ALU_A, ALU_B, ALU_OUT: std_logic_vector(7 downto 0);
signal SP_OUT: std_logic_vector(7 downto 0);

-- MUX SELECT SIGNALS
signal MAR_SEL, MDR_MUX_SEL, R0_SEL, R1_SEL: std_logic_vector(1 downto 0);
signal V_MUX_SEL, V_SEL, Z_SEL: std_logic_vector(0 to 0);

-- 1-BIT MUX
signal V_MUX_0, V_MUX_1: std_logic;
signal V_MUX_OUT, Z_MUX_OUT, Z_OUT: std_logic_vector(0 to 0);

-- MUX DATA ARRAYS
signal MAR_MUX_DATA, MDR_MUX_DATA, R0_MUX_DATA, R1_MUX_DATA: std_logic_2D(3
downto 0, 7 downto 0);
signal V_MUX_DATA, Z_MUX_DATA: std_logic_2D(1 downto 0, 0 to 0);
```

```
-- MICROSEQUENCER / UOP SIGNALS
signal uop: std_logic_vector(1 to 24);
signal m_next_address: std_logic_vector(8 downto 0);

begin
	CLOCK <= not(button);
	delay: lpm_counter
		generic map(lpm_width=>2)
		port map(clock=>CLOCK, cout=>useqEnable);

	useqClear <= clear;
	PC_CLEAR <= uop(1) and useqEnable;
	PC_INC <= uop(2) and useqEnable;
	IR_LOAD <= uop(3);
	WRITE_SEL <= uop(4) and useqEnable;
	MAR_SEL(1) <= uop(5) and useqEnable;
	MAR_SEL(0) <= uop(6) and useqEnable;
	MAR_LOAD <= uop(7) and useqEnable;
	MDR_SEL <= uop(8) and useqEnable;
	MDR_LOAD <= uop(9) and useqEnable;
	R0_SEL(1) <= uop(10) and useqEnable;
	R0_SEL(0) <= uop(11) and useqEnable;
	R0_LOAD <= uop(12) and useqEnable;
	R1_SEL(1) <= uop(13) and useqEnable;
	R1_SEL(0) <= uop(14) and useqEnable;
	R1_LOAD <= uop(15) and useqEnable;
	ALU_SEL <= uop(16) and useqEnable;
	V_SEL(0) <= uop(17) and useqEnable;
	Z_SEL(0) <= uop(18) and useqEnable;
```

```
	Z_LOAD <= uop(19) and useqEnable;
	JUMP_SEL <= uop(20) and useqEnable;
	JUMPZ_SEL <= uop(21) and useqEnable;
	SP_LOAD <= uop(22) and useqEnable;
	SP_INC <= uop(23) and useqEnable;
	SP_DEC <= uop(24) and useqEnable;

	PC_LOAD <= (JUMPZ_SEL and Z_OUT(0)) or JUMP_SEL;
	R0_ENABLE <= R0_LOAD and not(IR_OUT(0));
	R1_ENABLE <= R1_LOAD and IR_OUT(0);
	SP_ENABLE <= SP_INC or SP_DEC;
	V_MUX_0 <= R0_OUT(0) OR R0_OUT(1) OR R0_OUT(2) OR R0_OUT(3) OR
R0_OUT(4) OR R0_OUT(5) OR R0_OUT(6) OR R0_OUT(7);
	V_MUX_1 <= R1_OUT(0) OR R1_OUT(1) OR R1_OUT(2) OR R1_OUT(3) OR
R1_OUT(4) OR R1_OUT(5) OR R1_OUT(6) OR R1_OUT(7);
	V_MUX_SEL(0) <= V_SEL(0) and IR_OUT(0);

	process (IR_OUT)
	begin
		if IR_OUT(0) = '0' then
			ALU_A <= R0_OUT;
			ALU_B <= R1_OUT;
		else
			ALU_A <= R1_OUT;
			ALU_B <= R0_OUT;
		end if;
	end process;

	PC: lpm_counter
		generic map(lpm_width=>8)
```

```
		port map(sload=>PC_LOAD, data=>MDR_OUT, q=>PC_OUT,
cnt_en=>PC_INC, sclr=>PC_CLEAR, clock=>CLOCK);

	IR: lpm_ff
		generic map(lpm_width=>8)
		port map(enable=>IR_LOAD, data=>MDR_OUT, q=>IR_OUT,
clock=>CLOCK);

	uSeq: uSequencer
		generic map(uROM_width=>33, uROM_file=>"rom.mif")
		port map(opcode=>IR_OUT(7 downto 4), uop=>uop,
debug_map_addr=>m_next_address, enable=>useqEnable, clear=>useqClear,
clock=>CLOCK);

	MAR_MUX_LOOP: for i in 0 to 7 generate
		MAR_MUX_DATA(0, i) <= PC_OUT(i);
		MAR_MUX_DATA(1, i) <= MDR_OUT(i);
		MAR_MUX_DATA(2, i) <= SP_OUT(i);
		MAR_MUX_DATA(3, i) <= '0';
	end generate;

	MAR_MUX: lpm_mux
		generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
		port map(data=>MAR_MUX_DATA, result=>MAR_MUX_OUT,
sel=>MAR_SEL);

	MAR: lpm_ff
		generic map(lpm_width=>8)
		port map(enable=>MAR_LOAD, data=>MAR_MUX_OUT, q=>MAR_OUT,
clock=>CLOCK);
```

```
	RAM: lpm_ram_dq
		generic map(lpm_widthad=>8, lpm_width=>8, lpm_file=>"ram.mif")
		port map(address=>MAR_OUT, data=>MDR_OUT, q=>MEM_OUT,
inclock=>CLOCK, outclock=>CLOCK, we=>WRITE_SEL);

	MDR_MUX_LOOP: for i in 0 to 7 generate
		MDR_MUX_DATA(0, i) <= MEM_OUT(i);
		MDR_MUX_DATA(1, i) <= R0_OUT(i);
		MDR_MUX_DATA(2, i) <= MEM_OUT(i);
		MDR_MUX_DATA(3, i) <= R1_OUT(i);
	end generate;

	MDR_MUX: lpm_mux
		generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
		port map(data=>MDR_MUX_DATA, result=>MDR_MUX_OUT,
sel=>(IR_OUT(0) & MDR_SEL));

	MDR: lpm_ff
		generic map(lpm_width=>8)
		port map(enable=>MDR_LOAD, data=>MDR_MUX_OUT, q=>MDR_OUT,
clock=>CLOCK);

	ALU: lpm_add_sub
		generic map (lpm_width=>8)
		port map (dataa=>ALU_A, datab=>ALU_B, result=>ALU_OUT,
add_sub=>ALU_SEL);

	R0_MUX_LOOP: for i in 0 to 7 generate
		R0_MUX_DATA(0, i) <= MDR_OUT(i);
```

```
                R0_MUX_DATA(1, i) <= ALU_OUT(i);
                R0_MUX_DATA(2, i) <= R1_OUT(i);
                R0_MUX_DATA(3, i) <= '0';
        end generate;

        R0_MUX: lpm_mux
                generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
                port map(data=>R0_MUX_DATA, result=>R0_MUX_OUT, sel=>R0_SEL);

        REG0: lpm_ff
                generic map(lpm_width=>8)
                port map(enable=>R0_ENABLE, data=>R0_MUX_OUT, q=>R0_OUT,
clock=>CLOCK);

        R1_MUX_LOOP: for i in 0 to 7 generate
                R1_MUX_DATA(0, i) <= MDR_OUT(i);
                R1_MUX_DATA(1, i) <= ALU_OUT(i);
                R1_MUX_DATA(2, i) <= R0_OUT(i);
                R1_MUX_DATA(3, i) <= '0';
        end generate;

        R1_MUX: lpm_mux
                generic map(lpm_width=> 8, lpm_size=> 4, lpm_widths=> 2)
                port map(data=>R1_MUX_DATA, result=>R1_MUX_OUT, sel=>R1_SEL);

        REG1: lpm_ff
                generic map(lpm_width=>8)
                port map(enable=>R1_ENABLE, data=>R1_MUX_OUT, q=>R1_OUT,
clock=>CLOCK);
```

```
	STACK_POINTER: lpm_counter
		generic map(lpm_width=>8)
		port map(sload=>SP_LOAD, data=>MDR_OUT, q=>SP_OUT,
cnt_en=>SP_ENABLE, updown=>SP_INC, clock=>CLOCK);

	V_MUX_DATA(0, 0) <= V_MUX_0;
	V_MUX_DATA(1, 0) <= V_MUX_1;

	V_MUX: lpm_mux
		generic map(lpm_width=>1, lpm_size=>2, lpm_widths=>1)
		port map(data=>V_MUX_DATA, result=>V_MUX_OUT, sel=>V_MUX_SEL);

	Z_MUX_DATA(0, 0) <= not V_MUX_OUT(0);
	Z_MUX_DATA(1, 0) <= V_MUX_OUT(0);

	Z_MUX: lpm_mux
		generic map(lpm_width=>1, lpm_size=>2, lpm_widths=>1)
		port map(data=>Z_MUX_DATA, result=>Z_MUX_OUT, sel=>Z_SEL);

	REG_Z: lpm_ff
		generic map(lpm_width=>1)
		port map(enable=>Z_LOAD, data=>Z_MUX_OUT, q=>Z_OUT,
clock=>CLOCK);

	Z_FLAG <= Z_OUT;

	R0Hi: seven_segment_display port map(digit=>R0_OUT(7 downto 4),
display=>R0SegHi);
	R0Lo: seven_segment_display port map(digit=>R0_OUT(3 downto 0),
display=>R0SegLo);
```

```
        R1Hi: seven_segment_display port map(digit=>R1_OUT(7 downto 4),
display=>R1SegHi);
        R1Lo: seven_segment_display port map(digit=>R1_OUT(3 downto 0),
display=>R1SegLo);
        pcHi: seven_segment_display port map(digit=>PC_OUT(7 downto 4),
display=>pcSegHi);
        pcLo: seven_segment_display port map(digit=>PC_OUT(3 downto 0),
display=>pcSegLo);
        spHi: seven_segment_display port map(digit=>SP_OUT(7 downto 4),
display=>spSegHi);
        spLo: seven_segment_display port map(digit=>SP_OUT(3 downto 0),
display=>spSegLo);

end structural;
```

Figure 16: VHDL implementation of the full microprogrammed CPU on the FPGA, including seven-segment displays.

FPGA Pin Layout

Figure 17: Signal Number and its corresponding Pin Number on the *Altera Cyclone® IV EP4CE115F29C7* FPGA.

Node Name	Direction	Location
button	Input	PIN_M21
clear	Input	PIN_Y23
R0SeqHi[0]	Output	PIN_M24
R0SeqHi[1]	Output	PIN_Y22
R0SeqHi[2]	Output	PIN_W21
R0SeqHi[3]	Output	PIN_W22
R0SeqHi[4]	Output	PIN_W25
R0SeqHi[5]	Output	PIN_U23
R0SeqHi[6]	Output	PIN_U24
R0SeqLo[0]	Output	PIN_G18
R0SeqLo[1]	Output	PIN_F22
R0SeqLo[2]	Output	PIN_E17
R0SeqLo[3]	Output	PIN_L26
R0SeqLo[4]	Output	PIN_L25
R0SeqLo[5]	Output	PIN_J22
R0SeqLo[6]	Output	PIN_H22
R1SeqHi[0]	Output	PIN_V21
R1SeqHi[1]	Output	PIN_U21
R1SeqHi[2]	Output	PIN_AB20
R1SeqHi[3]	Output	PIN_AA21
R1SeqHi[4]	Output	PIN_AD24
R1SeqHi[5]	Output	PIN_AF23
R1SeqHi[6]	Output	PIN_Y19
R1SeqLo[0]	Output	PIN_AA25
R1SeqLo[1]	Output	PIN_AA26
R1SeqLo[2]	Output	PIN_Y25
R1SeqLo[3]	Output	PIN_W26
R1SeqLo[4]	Output	PIN_Y26
R1SeqLo[5]	Output	PIN_W27
R1SeqLo[6]	Output	PIN_W28
Z_FLAG[0]	Output	PIN_H15
pcSeqHi[0]	Output	PIN_AD18
pcSeqHi[1]	Output	PIN_AC18
pcSeqHi[2]	Output	PIN_AB18
pcSeqHi[3]	Output	PIN_AH19
pcSeqHi[4]	Output	PIN_AG19
pcSeqHi[5]	Output	PIN_AF18
pcSeqHi[6]	Output	PIN_AH18
pcSeqLo[0]	Output	PIN_AB19
pcSeqLo[1]	Output	PIN_AA19
pcSeqLo[2]	Output	PIN_AG21
pcSeqLo[3]	Output	PIN_AH21
pcSeqLo[4]	Output	PIN_AE19
pcSeqLo[5]	Output	PIN_AF19
pcSeqLo[6]	Output	PIN_AE18
spSeqHi[0]	Output	PIN_AD17
spSeqHi[1]	Output	PIN_AE17
spSeqHi[2]	Output	PIN_AG17
spSeqHi[3]	Output	PIN_AH17
spSeqHi[4]	Output	PIN_AF17
spSeqHi[5]	Output	PIN_AG18
spSeqHi[6]	Output	PIN_AA14
spSeqLo[0]	Output	PIN_AA17
spSeqLo[1]	Output	PIN_AB16
spSeqLo[2]	Output	PIN_AA16
spSeqLo[3]	Output	PIN_AB17
spSeqLo[4]	Output	PIN_AB15
spSeqLo[5]	Output	PIN_AA15
spSeqLo[6]	Output	PIN_AC17

Figure 18: Final Pin Assignments for the Microprogrammed CPU Design Implementation on the FPGA Board.

RESULTS AND DISCUSSION

Before implementing the full CPU design in VHDL, we first verified the functionality of the microsequencer independently. Using the tests described in the Theory and Methods section, we hardwired opcodes and initialized the ROM with .mif files to exercise specific sequences of micro-operations. During these tests, the seven-segment display was used to visualize the values of key registers, the program counter (PC), and outputs, providing a straightforward way to confirm that control signals were being correctly generated and that the microsequencer executed microinstructions in the expected order. These tests confirmed that the microsequencer correctly handled address sequencing, mapping signals, and control signal activation, forming a reliable foundation for the CPU implementation.

Following the successful verification of the microsequencer, we proceeded to test the RAM subsystem. By initializing RAM with .mif files and running simple load and store operations in conjunction with the microsequencer, we were able to validate proper data transfer between registers and memory, as well as the correct operation of address and data buses. The seven-segment display continued to serve as a key debugging tool, allowing real-time monitoring of register values, memory contents, and intermediate outputs.

Once both the microsequencer and RAM were verified, we integrated these components into the full CPU VHDL design. To validate the complete system, we utilized the collection of RAM files provided by our professor, which contained sequences covering all 16 CPU instructions (**Figure 19**). Running the CPU design with each of these files allowed us to verify that every instruction executed correctly without any modifications to the VHDL code. This demonstrates that our design is truly general-purpose: the CPU executes the full instruction set as intended without being tailored to specific instructions, reflecting the behavior expected of a functional, programmable processor.

We presented our design in person to our professor, demonstrating the CPU running each RAM file and verifying proper functionality across all instructions. The professor provided written approval as confirmation of successful verification. The ability to execute all

instructions with a single, unmodified CPU design highlights the correctness and robustness of our implementation, as well as the effectiveness of our testing methodology using the microsequencer, RAM, and seven-segment display for debugging.

ram_files

Figure 19: Folder containing RAM files provided by the professor, used to test the CPU design with all 16 instructions and verify functionality across the full instruction set.

CONCLUSIONS

Throughout this project, we leveraged the seven-segment display as a key debugging tool, frequently changing which values were displayed to monitor the inputs and outputs of specific components as we progressed through testing the RAM files. This iterative approach allowed us to identify and correct potential issues at the microsequencer, RAM, and CPU levels, ensuring that each component functioned as intended.

Ultimately, our final VHDL implementation successfully executed every instruction in the CPU's instruction set, regardless of how the instructions were programmed in the RAM. This demonstrated that our design is fully general-purpose and capable of handling a complete range of operations without modification, reflecting the principles of a functional and programmable CPU.

This project provided an invaluable hands-on experience in computer architecture and digital design. By designing, testing, and debugging a microprogrammed CPU on an FPGA, we gained practical insight into the interaction between control units, memory, and the execution of instructions. Such experience is crucial for computer engineering students, as it bridges theoretical knowledge with practical skills needed to understand and implement the core elements of modern processors.

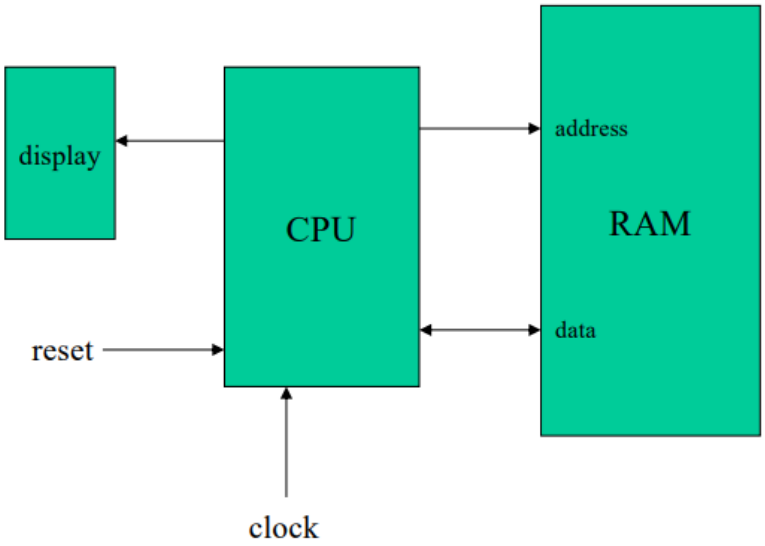
REFERENCES

1. ECE495 Slides by Edwin Hou

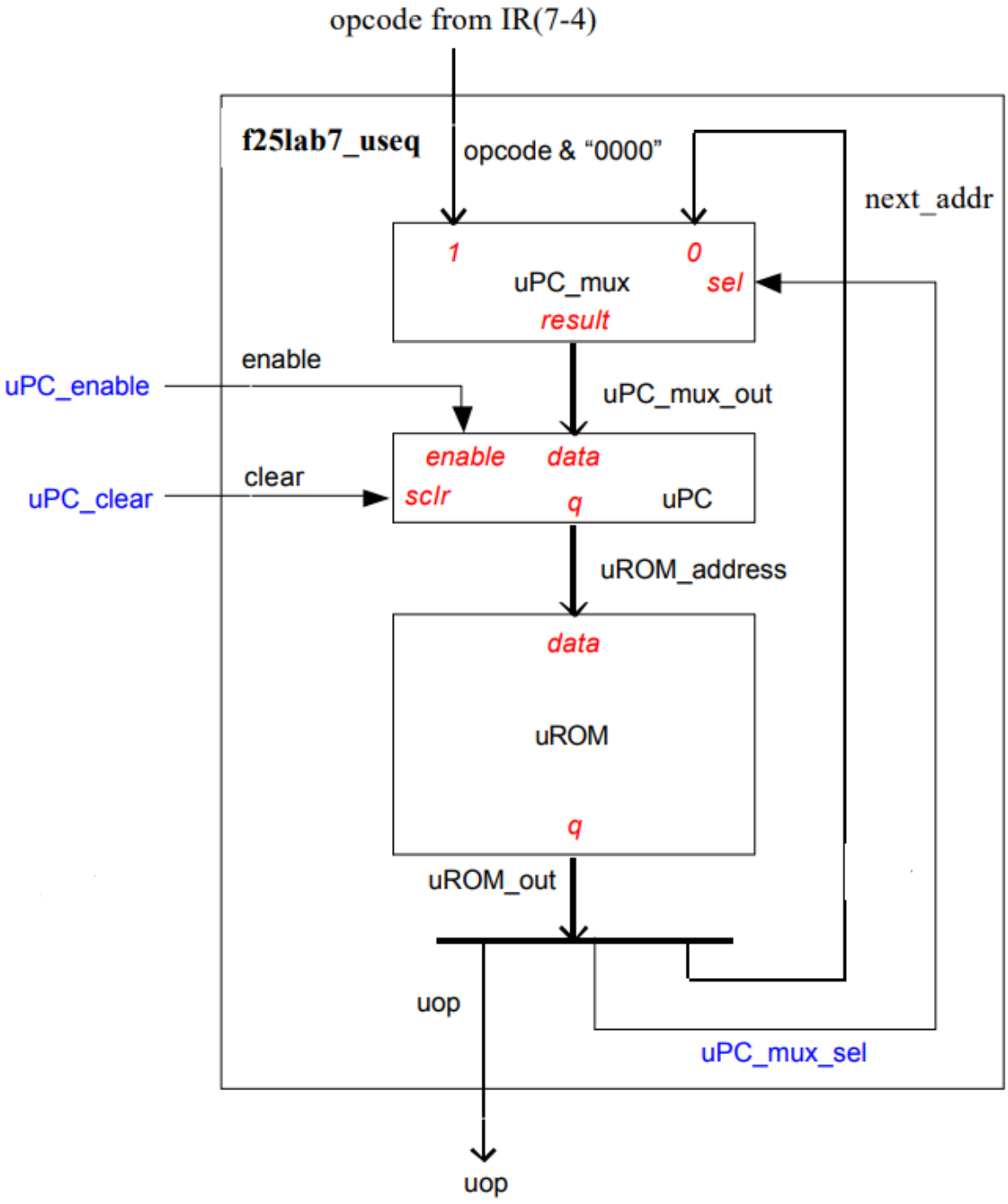
APPENDIX

- Block Diagrams

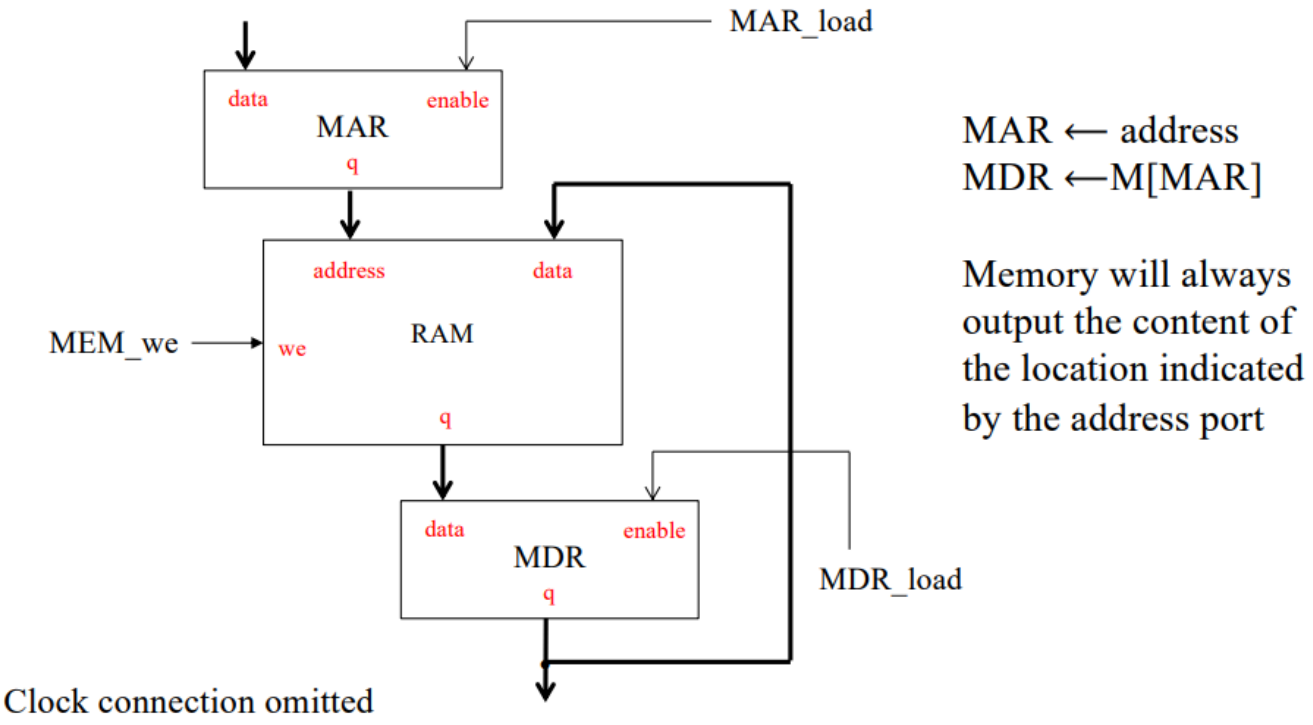
CPU Block Diagram



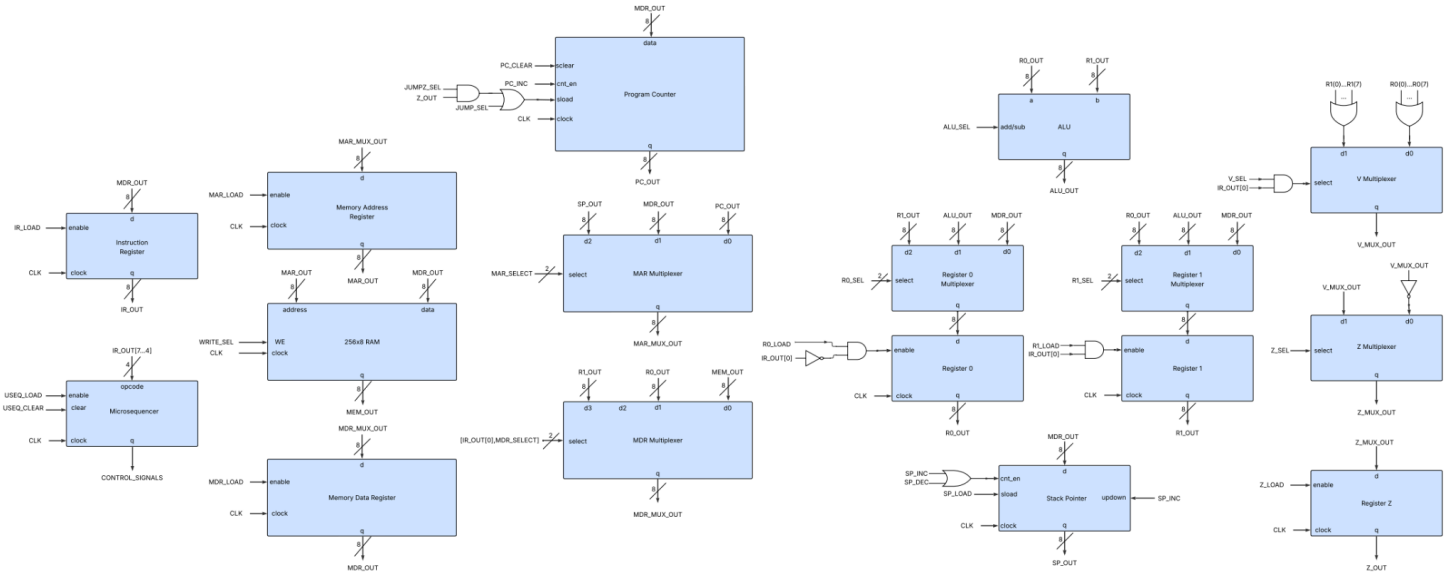
Microsequencer Block Diagram



RAM Block Diagram



Microprogrammed CPU Design Block Diagram



- Source Code

Microsequencer Component

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
 );
end uSequencer;

architecture structural of uSequencer is
signal uROM_address: std_logic_vector (7 downto 0);
signal uROM_out: std_logic_vector (uROM_width-1 downto 0);
signal uPC_mux_data: std_logic_2D(1 downto 0, 7 downto 0);
signal uPC_mux_sel: std_logic_vector(0 to 0);
signal uPC_mux_out: std_logic_vector(7 downto 0);
signal temp: std_logic_vector(7 downto 0);
```

```
begin
  temp <= opcode & "0000";

  for_label: for i in 0 to 7 generate
      uPC_mux_data(0, i) <= uROM_out(i);
      uPC_mux_data(1, i) <= temp(i);
  end generate;

  uPC_mux_sel(0) <= uROM_out(8);

  uPC_mux: lpm_mux
          generic map (lpm_width=>8, lpm_size=>2, lpm_widths=>1)
          port map (result=>uPC_mux_out, data=>uPC_mux_data, sel=>uPC_mux_sel);
  uPC: lpm_ff
          generic map (lpm_width=>8)
          port map (clock=>clock, data=>uPC_mux_out, q=>uROM_address, sclr=>clear,
enable=>enable);
  uROM: lpm_rom
          generic map (lpm_widthad=>8, lpm_width=>uROM_width, lpm_file=>uROM_file)
          port map (address=>uROM_address, q=>uROM_out, inclock=>clock,
outclock=>clock);

  uop <= uROM_out(uROM_width-1 downto 9);
  debug_map_addr <= uROM_out(8 downto 0);
end structural;
```

Test Microsequencer without PC Register

```
library ieee;
use ieee.std_logic_1164.all;
```

```
library lpm;
use lpm.lpm_components.all;

entity test_uSequencer is
	port(
		button, clear: in std_logic;
		useqEnOut: out std_logic;
		ctrlSignals: out std_logic_vector(0 to 7);
		M_disp, disp1, disp0: out std_logic_vector(0 to 6)
	);
end test_uSequencer;

architecture structural of test_uSequencer is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
 );
end component;

component seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
```

```
                display: out std_logic_vector(0 to 6)
        );
end component;

signal useqEnable, useqClear, clk: std_logic;
signal uop: std_logic_vector(0 to 7);
signal address_out: std_logic_vector(8 downto 0);
signal M, addr_hex1, addr_hex0: std_logic_vector(3 downto 0);
begin
        clk <= not(button);
        delay: lpm_counter generic map(lpm_width=>2) port map(clock=>clk,
cout=>useqEnable);

        useqClear <= clear;
        useqEnOut <= useqEnable;

        labelG: for i in 0 to 7 generate
                ctrlSignals(i) <= uop(i) and useqEnable;
        end generate;

        uSeq: uSequencer
                generic map(uROM_width=>17, uROM_file=>"test_uSequencer_file.mif")
                port map(opcode=>"0011", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);



        M <= "000" & address_out(8);
        addr_hex1 <= address_out(7 downto 4);
        addr_hex0 <= address_out(3 downto 0);
        display1: seven_segment_display port map(digit=>M, display=>M_disp);
```

```
        display2: seven_segment_display port map(digit=>addr_hex1, display=>disp1);
        display3: seven_segment_display port map(digit=>addr_hex0, display=>disp0);
end structural;
```

Test Microsequencer with PC Register

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity test_uSequencer_PC is
        port(
                button, clear: in std_logic;
                useqEnOut: out std_logic;
                ctrlSignals: out std_logic_vector(0 to 7);
                pc_Seg1, pc_Seg0, M_disp, disp1, disp0: out std_logic_vector(0 to 6)
        );
end test_uSequencer_PC;

architecture structural of test_uSequencer_PC is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
```

```
    enable, clear: in std_logic;
    clock: in std_logic
  );
end component;

component seven_segment_display is
        port(
                digit: in std_logic_vector(3 downto 0);
                display: out std_logic_vector(0 to 6)
        );
end component;

signal useqEnable, useqClear, clk: std_logic;
signal pcInc, pcLoad, pcClear: std_logic;
signal pcOut: std_logic_vector(7 downto 0);
signal uop: std_logic_vector(0 to 7);
signal address_out: std_logic_vector(8 downto 0);
signal M, addr_hex1, addr_hex0, pc_hex1, pc_hex0: std_logic_vector(3 downto 0);
begin
        clk <= not(button);
        delay: lpm_counter
                generic map(lpm_width=>2)
                port map(clock=>clk, cout=>useqEnable);

        useqClear <= clear;
        useqEnOut <= useqEnable;

        labelG: for i in 0 to 7 generate
                ctrlSignals(i) <= uop(i) and useqEnable;
        end generate;
```

```
	pcInc <= uop(0) and useqEnable;
	pcLoad <= uop(1) and useqEnable;
	pcClear <= uop(2) and useqEnable;

	uSeq: uSequencer
		generic map(uROM_width=>17,
uROM_file=>"test_uSequencer_PC_file.mif")
		port map(opcode=>"0011", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);

	pc: lpm_counter
		generic map(lpm_width=>8)
		port map(sload=>pcLoad, data=> "10101010", q=>pcOut, cnt_en=>pcInc,
sclr=>pcClear, clock=>clk);

	M <= "000" & address_out(8);
	addr_hex1 <= address_out(7 downto 4);
	addr_hex0 <= address_out(3 downto 0);
	pc_hex1 <= pcOut(7 downto 4);
	pc_hex0 <= pcOut(3 downto 0);
	display1: seven_segment_display port map(digit=>M, display=>M_disp);
	display2: seven_segment_display port map(digit=>addr_hex1, display=>disp1);
	display3: seven_segment_display port map(digit=>addr_hex0, display=>disp0);
	display4: seven_segment_display port map(digit=>pc_hex1, display=>pc_Seg1);
	display5: seven_segment_display port map(digit=>pc_hex0, display=>pc_Seg0);

end structural;
```

Read and Write to/from RAM using MicroSequencer

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity test_ram is
	port(
		button, clear: in std_logic;
		useqEnOut: out std_logic;
		marData: in std_logic_vector(7 downto 0); -- connected to switches, input to MAR
		marSegHi, marSegLo, mdrSegHi, mdrSegLo, nextAddrHi, nextAddrLo: out std_logic_vector(0 to 6);
		ctrlSignals: out std_logic_vector(0 to 7)
	);
end test_ram;

architecture structural of test_ram is
component uSequencer is
 generic(
  uROM_width: integer := 10;
  uROM_file: string := ""
 );
 port(
  opcode: in std_logic_vector(3 downto 0);
  uop: out std_logic_vector(1 to (uROM_width-9));
  debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
  enable, clear: in std_logic;
  clock: in std_logic
```

```
 );
end component;

component seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
		display: out std_logic_vector(0 to 6)
	);
end component;

signal clk, useqEnable, useqClear, marLoad, mdrLoad, memWE: std_logic;
signal uop: std_logic_vector(0 to 7);
signal marADig, marBDig, mdrADig, mdrBDig, nextAADig, nextABDig: std_logic_vector(3
downto 0);
signal marOut, memOut, mdrOut: std_logic_vector(7 downto 0);
signal address_out: std_logic_vector(8 downto 0);
begin
	clk <= not(button);
	useqClear <= clear;
	useqEnOut <= useqEnable;
	marLoad <= uop(0) and useqEnable;
	mdrLoad <= uop(1) and useqEnable;
	memWE <= uop(2) and useqEnable;

	marADig <= marOut(7 downto 4);
	marBDig <= marOut(3 downto 0);
	mdrADig <= mdrOut(7 downto 4);
	mdrBDig <= mdrOut(3 downto 0);
	nextAADig <= address_out(7 downto 4);
	nextABDig <= address_out(3 downto 0);
```

```
	labelG: for i in 0 to 7 generate
		ctrlSignals(i) <= uop(i) and useqEnable;
	end generate;

	delay: lpm_counter generic map(lpm_width=>2) port map(clock=>clk,
cout=>useqEnable);
	microSequencer: uSequencer
		generic map(uROM_width=>17, uROM_file=>"test_rom_file.mif")
		port map(opcode=>"0000", uop=>uop, debug_map_addr=>address_out,
enable=>useqEnable, clear=>useqClear, clock=>clk);
	marReg: lpm_ff
		generic map(lpm_width=>8)
		port map(data=>marData, q=>marOut, clock=>clk, enable=>marLoad);
	mdrReg: lpm_ff
		generic map(lpm_width=>8)
		port map(data=>memOut, q=>mdrOut, clock=>clk, enable=>mdrLoad);
	ram: lpm_ram_dq
		generic map(lpm_widthad=>8, lpm_width=>8,
lpm_file=>"test_ram_file.mif")
		port map(data=>mdrOut, address=>marOut, q=>memOut, inclock=>clk,
outclock=>clk, we=>memWE);
	marA: seven_segment_display port map(digit=>marADig, display=>marSegHi);
	marB: seven_segment_display port map(digit=>marBDig, display=>marSegLo);
	mdrA: seven_segment_display port map(digit=>mdrADig, display=>mdrSegHi);
	mdrB: seven_segment_display port map(digit=>mdrBDig, display=>mdrSegLo);
	nextAddressA: seven_segment_display port map(digit=>nextAADig,
display=>nextAddrHi);
	nextAddressB: seven_segment_display port map(digit=>nextABDig,
display=>nextAddrLo);
```

```
end structural;
```

Seven Segment Display Component

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

entity seven_segment_display is
	port(
		digit: in std_logic_vector(3 downto 0);
		display: out std_logic_vector(0 to 6)
	);
end seven_segment_display;

architecture behavior of seven_segment_display is
begin
	with digit select
		display <= "0000001" when "0000",
		"1001111" when "0001",
		"0010010" when "0010",
		"0000110" when "0011",
		"1001100" when "0100",
		"0100100" when "0101",
		"0100000" when "0110",
		"0001111" when "0111",
		"0000000" when "1000",
		"0000100" when "1001",
		"0001000" when "1010",
		"1100000" when "1011",
		"0110001" when "1100",
```

```
                "1000010" when "1101",
                "0110000" when "1110",
                "0111000" when "1111",
                "1111111" when others;
end behavior;
```

Microsequencer-based CPU

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity cpu is
        port(
                button, clear: in std_logic;
                Z_FLAG: out std_logic_vector(0 to 0);
                R0SegHi, R0SegLo, R1SegHi, R1SegLo, pcSegHi, pcSegLo, spSegHi, spSegLo:
out std_logic_vector(0 to 6)
        );
end cpu;

architecture structural of cpu is
component uSequencer is
        generic(
                uROM_width: integer := 10;
                uROM_file: string := ""
        );
        port(
                opcode: in std_logic_vector(3 downto 0);
```

```
                uop: out std_logic_vector(1 to (uROM_width-9));
                debug_map_addr: out std_logic_vector(8 downto 0);
                enable, clear: in std_logic;
                clock: in std_logic
        );
end component;

component seven_segment_display is
        port(
                digit: in std_logic_vector(3 downto 0);
                display: out std_logic_vector(0 to 6)
        );
end component;

-- CLOCK / CONTROL SIGNALS
signal CLOCK, useqEnable, useqClear: std_logic;
signal PC_LOAD, PC_CLEAR, PC_INC, IR_LOAD, WRITE_SEL, MAR_LOAD, MDR_SEL,
MDR_LOAD: std_logic;
signal R0_ENABLE, R0_LOAD, R1_ENABLE, R1_LOAD, ALU_SEL, Z_LOAD, JUMP_SEL,
JUMPZ_SEL: std_logic;
signal SP_ENABLE, SP_LOAD, SP_INC, SP_DEC, M: std_logic;

-- DATA PATH SIGNALS (8-bit registers / ALU)
signal PC_OUT, IR_OUT, MAR_OUT, MEM_OUT, MDR_OUT: std_logic_vector(7 downto 0);
signal MAR_MUX_OUT, MDR_MUX_OUT: std_logic_vector(7 downto 0);
signal R0_OUT, R0_MUX_OUT, R1_OUT, R1_MUX_OUT: std_logic_vector(7 downto 0);
signal ALU_A, ALU_B, ALU_OUT: std_logic_vector(7 downto 0);
signal SP_OUT: std_logic_vector(7 downto 0);

-- MUX SELECT SIGNALS
```

```
signal MAR_SEL, MDR_MUX_SEL, R0_SEL, R1_SEL: std_logic_vector(1 downto 0);
signal V_MUX_SEL, V_SEL, Z_SEL: std_logic_vector(0 to 0);

-- 1-BIT MUX
signal V_MUX_0, V_MUX_1: std_logic;
signal V_MUX_OUT, Z_MUX_OUT, Z_OUT: std_logic_vector(0 to 0);

-- MUX DATA ARRAYS
signal MAR_MUX_DATA, MDR_MUX_DATA, R0_MUX_DATA, R1_MUX_DATA: std_logic_2D(3
downto 0, 7 downto 0);
signal V_MUX_DATA, Z_MUX_DATA: std_logic_2D(1 downto 0, 0 to 0);

-- MICROSEQUENCER / UOP SIGNALS
signal uop: std_logic_vector(1 to 24);
signal m_next_address: std_logic_vector(8 downto 0);

begin
	CLOCK <= not(button);
	delay: lpm_counter
		generic map(lpm_width=>2)
		port map(clock=>CLOCK, cout=>useqEnable);

	useqClear <= clear;
	PC_CLEAR <= uop(1) and useqEnable;
	PC_INC <= uop(2) and useqEnable;
	IR_LOAD <= uop(3);
	WRITE_SEL <= uop(4) and useqEnable;
	MAR_SEL(1) <= uop(5) and useqEnable;
	MAR_SEL(0) <= uop(6) and useqEnable;
	MAR_LOAD <= uop(7) and useqEnable;
```

```
        MDR_SEL <= uop(8) and useqEnable;
        MDR_LOAD <= uop(9) and useqEnable;
        R0_SEL(1) <= uop(10) and useqEnable;
        R0_SEL(0) <= uop(11) and useqEnable;
        R0_LOAD <= uop(12) and useqEnable;
        R1_SEL(1) <= uop(13) and useqEnable;
        R1_SEL(0) <= uop(14) and useqEnable;
        R1_LOAD <= uop(15) and useqEnable;
        ALU_SEL <= uop(16) and useqEnable;
        V_SEL(0) <= uop(17) and useqEnable;
        Z_SEL(0) <= uop(18) and useqEnable;
        Z_LOAD <= uop(19) and useqEnable;
        JUMP_SEL <= uop(20) and useqEnable;
        JUMPZ_SEL <= uop(21) and useqEnable;
        SP_LOAD <= uop(22) and useqEnable;
        SP_INC <= uop(23) and useqEnable;
        SP_DEC <= uop(24) and useqEnable;

        PC_LOAD <= (JUMPZ_SEL and Z_OUT(0)) or JUMP_SEL;
        R0_ENABLE <= R0_LOAD and not(IR_OUT(0));
        R1_ENABLE <= R1_LOAD and IR_OUT(0);
        SP_ENABLE <= SP_INC or SP_DEC;
        V_MUX_0 <= R0_OUT(0) OR R0_OUT(1) OR R0_OUT(2) OR R0_OUT(3) OR
R0_OUT(4) OR R0_OUT(5) OR R0_OUT(6) OR R0_OUT(7);
        V_MUX_1 <= R1_OUT(0) OR R1_OUT(1) OR R1_OUT(2) OR R1_OUT(3) OR
R1_OUT(4) OR R1_OUT(5) OR R1_OUT(6) OR R1_OUT(7);
        V_MUX_SEL(0) <= V_SEL(0) and IR_OUT(0);

        process (IR_OUT)
        begin
```

```
                if IR_OUT(0) = '0' then
                        ALU_A <= R0_OUT;
                        ALU_B <= R1_OUT;
                else
                        ALU_A <= R1_OUT;
                        ALU_B <= R0_OUT;
                end if;
        end process;

        PC: lpm_counter
                generic map(lpm_width=>8)
                port map(sload=>PC_LOAD, data=>MDR_OUT, q=>PC_OUT,
cnt_en=>PC_INC, sclr=>PC_CLEAR, clock=>CLOCK);

        IR: lpm_ff
                generic map(lpm_width=>8)
                port map(enable=>IR_LOAD, data=>MDR_OUT, q=>IR_OUT,
clock=>CLOCK);

        uSeq: uSequencer
                generic map(uROM_width=>33, uROM_file="rom.mif")
                port map(opcode=>IR_OUT(7 downto 4), uop=>uop,
debug_map_addr=>m_next_address, enable=>useqEnable, clear=>useqClear,
clock=>CLOCK);

        MAR_MUX_LOOP: for i in 0 to 7 generate
                MAR_MUX_DATA(0, i) <= PC_OUT(i);
                MAR_MUX_DATA(1, i) <= MDR_OUT(i);
                MAR_MUX_DATA(2, i) <= SP_OUT(i);
                MAR_MUX_DATA(3, i) <= '0';
```

```
	end generate;

	MAR_MUX: lpm_mux
		generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
		port map(data=>MAR_MUX_DATA, result=>MAR_MUX_OUT,
sel=>MAR_SEL);

	MAR: lpm_ff
		generic map(lpm_width=>8)
		port map(enable=>MAR_LOAD, data=>MAR_MUX_OUT, q=>MAR_OUT,
clock=>CLOCK);

	RAM: lpm_ram_dq
		generic map(lpm_widthad=>8, lpm_width=>8, lpm_file=>"ram.mif")
		port map(address=>MAR_OUT, data=>MDR_OUT, q=>MEM_OUT,
inclock=>CLOCK, outclock=>CLOCK, we=>WRITE_SEL);

	MDR_MUX_LOOP: for i in 0 to 7 generate
		MDR_MUX_DATA(0, i) <= MEM_OUT(i);
		MDR_MUX_DATA(1, i) <= R0_OUT(i);
		MDR_MUX_DATA(2, i) <= MEM_OUT(i);
		MDR_MUX_DATA(3, i) <= R1_OUT(i);
	end generate;

	MDR_MUX: lpm_mux
		generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
		port map(data=>MDR_MUX_DATA, result=>MDR_MUX_OUT,
sel=>(IR_OUT(0) & MDR_SEL));

	MDR: lpm_ff
```

```
                generic map(lpm_width=>8)
                port map(enable=>MDR_LOAD, data=>MDR_MUX_OUT, q=>MDR_OUT,
clock=>CLOCK);

        ALU: lpm_add_sub
                generic map (lpm_width=>8)
                port map (dataa=>ALU_A, datab=>ALU_B, result=>ALU_OUT,
add_sub=>ALU_SEL);

        R0_MUX_LOOP: for i in 0 to 7 generate
                R0_MUX_DATA(0, i) <= MDR_OUT(i);
                R0_MUX_DATA(1, i) <= ALU_OUT(i);
                R0_MUX_DATA(2, i) <= R1_OUT(i);
                R0_MUX_DATA(3, i) <= '0';
        end generate;

        R0_MUX: lpm_mux
                generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
                port map(data=>R0_MUX_DATA, result=>R0_MUX_OUT, sel=>R0_SEL);

        REG0: lpm_ff
                generic map(lpm_width=>8)
                port map(enable=>R0_ENABLE, data=>R0_MUX_OUT, q=>R0_OUT,
clock=>CLOCK);

        R1_MUX_LOOP: for i in 0 to 7 generate
                R1_MUX_DATA(0, i) <= MDR_OUT(i);
                R1_MUX_DATA(1, i) <= ALU_OUT(i);
                R1_MUX_DATA(2, i) <= R0_OUT(i);
                R1_MUX_DATA(3, i) <= '0';
```

```
        end generate;

        R1_MUX: lpm_mux
                generic map(lpm_width=> 8, lpm_size=> 4, lpm_widths=> 2)
                port map(data=>R1_MUX_DATA, result=>R1_MUX_OUT, sel=>R1_SEL);

        REG1: lpm_ff
                generic map(lpm_width=>8)
                port map(enable=>R1_ENABLE, data=>R1_MUX_OUT, q=>R1_OUT,
clock=>CLOCK);

        STACK_POINTER: lpm_counter
                generic map(lpm_width=>8)
                port map(sload=>SP_LOAD, data=>MDR_OUT, q=>SP_OUT,
cnt_en=>SP_ENABLE, updown=>SP_INC, clock=>CLOCK);

        V_MUX_DATA(0, 0) <= V_MUX_0;
        V_MUX_DATA(1, 0) <= V_MUX_1;

        V_MUX: lpm_mux
                generic map(lpm_width=>1, lpm_size=>2, lpm_widths=>1)
                port map(data=>V_MUX_DATA, result=>V_MUX_OUT, sel=>V_MUX_SEL);

        Z_MUX_DATA(0, 0) <= not V_MUX_OUT(0);
        Z_MUX_DATA(1, 0) <= V_MUX_OUT(0);

        Z_MUX: lpm_mux
                generic map(lpm_width=>1, lpm_size=>2, lpm_widths=>1)
                port map(data=>Z_MUX_DATA, result=>Z_MUX_OUT, sel=>Z_SEL);
```

```
        REG_Z: lpm_ff
                generic map(lpm_width=>1)
                port map(enable=>Z_LOAD, data=>Z_MUX_OUT, q=>Z_OUT,
clock=>CLOCK);

        Z_FLAG <= Z_OUT;

        R0Hi: seven_segment_display port map(digit=>R0_OUT(7 downto 4),
display=>R0SegHi);
        R0Lo: seven_segment_display port map(digit=>R0_OUT(3 downto 0),
display=>R0SegLo);
        R1Hi: seven_segment_display port map(digit=>R1_OUT(7 downto 4),
display=>R1SegHi);
        R1Lo: seven_segment_display port map(digit=>R1_OUT(3 downto 0),
display=>R1SegLo);
        pcHi: seven_segment_display port map(digit=>PC_OUT(7 downto 4),
display=>pcSegHi);
        pcLo: seven_segment_display port map(digit=>PC_OUT(3 downto 0),
display=>pcSegLo);
        spHi: seven_segment_display port map(digit=>SP_OUT(7 downto 4),
display=>spSegHi);
        spLo: seven_segment_display port map(digit=>SP_OUT(3 downto 0),
display=>spSegLo);

end structural;
```

- Register-Transfer Logic

Micro-operation Level RTL for CPU Instructions

```
# Develop the RTL statements for the CPU instructions:
```

```
# All instructions do Instruction Fetch:
F1: MAR ← PC # Load address of instruction
F2: MDR ← M[MAR], PC ← PC + 1 # Fetch instruction and increment PC
F3: IR ← MDR # Decode instruction

nop                    00000000
- None -

loadi Rn, X            0001000n X
LI1: MAR ← PC    # Load memory address of immediate operand
LI2: MDR ← M[MAR], PC ← PC + 1   # Fetch immediate operand and increment PC
LI3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1    # Load immediate value into R0 if
IR(0) = 0 else if IR(0) = 1 into R1

load Rn, X             0010000n X
L1: MAR ← PC    # Load memory address of operand
L2: MDR ← M[MAR], PC ← PC + 1   # Fetch operand address and increment PC
L3: MAR ← MDR   # Load effective address
L4: MDR ← M[MAR]    # Fetch data from memory
L5: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1    # Load data into R0 if IR(0) = 0 else if
IR(0) = 1 load into R1

store X, Rn            0011000n X
ST1: MAR ← PC    # Load memory address of operand
ST2: MDR ← M[MAR], PC ← PC + 1   # Fetch operand address and increment PC
ST3: MAR ← MDR   # Load effective address
ST4: MDR ← R0 if IR(0) = 0, MDR ← R1 if IR(0) = 1    # Load data from R0 if IR(0) = 0 else
if IR(0) = 1 load from R1
ST5: M[MAR] ← MDR    # Store data into memory
```

```
move Rn, Rm             0100000n m!=n
M1: R0 ← R1 if IR(0) = 0, R1 ← R0 if IR(0) = 1    # Move value from R1 to R0 if IR(0) = 0
else if IR(0) = 1 move from R0 to R1

add Rn, Rm              0101000n m!=n
A1: R0 ← R0 + R1 if IR(0) = 0, R1 ← R0 + R1 if IR(0) = 1    # R0 plus R1 if IR(0) = 0 else if
IR(0) = 1 R1 plus R0

sub Rn, Rm              0110000n m!=n
S1: R0 ← R0 - R1 if IR(0) = 0, R1 ← R1 - R0  if IR(0) = 1   # R0 minus R1 if IR(0) = 0 else if
IR(0) = 1 R1 minus R0

testnz Rn               0111000n
TN1: Z ← NOT(R0(0) OR ... OR R0(7)) if IR(0) = 0, ← NOT(R1(0) OR ... OR R1(7)) if IR(0)
= 1    # Z = 1 if R0 or R1 is nonzero

testz Rn                1000000n
T1: Z ← R0(0) OR ... OR R0(7) if IR(0) = 0, ← R1(0) OR ... OR R1(7) if IR(0) = 1    # Z = 1 if
R0 or R1 is nonzero

jump X                  10010000 X
J1: MAR ← PC    # Load memory address of jump target
J2: MDR ← M[MAR]   # Fetch jump target address
J3: PC ← MDR    # Jump to target address

jumpz X                 10100000 X
JZ1: MAR ← PC # Load memory address of jump target
JZ2: MDR ← M[MAR], PC <- PC + 1   # Fetch jump target address and increment PC
JZ3: PC ← MDR if Z=1   # Jump if Z flag is set
```

```
loadsp X                10110000 X
LS1: MAR ← PC   # Load memory address of operand
LS2: MDR ← M[MAR], PC ← PC + 1  # Fetch operand and increment PC
LS3: SP ← MDR   # Load operand into SP

peep Rn                 1100000n
P1: MAR ← SP    # Load address from stack pointer
P2: MDR ← M[MAR]   # Fetch value from stack
P3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1    # Peek value from stack into R0 if
IR(0) = 0 else if IR(0) = 1 into R1

push Rn                 1101000n
PU1: SP ← SP - 1  # Decrement stack pointer
PU2: MAR ← SP   # Load address from stack pointer
PU3: MDR ← R0 if IR(0) = 0, MDR ← R1 if IR(0) = 1   # Load value from R0 if IR(0) = 0
else if IR(0) = 1 from R1
PU4: M[MAR] ← MDR   # Push value onto stack

pop Rn                  1110000n
PO1: MAR ← SP   # Load address from stack pointer
PO2: MDR ← M[MAR]   # Fetch value from stack
PO3: R0 ← MDR if IR(0) = 0, R1 ← MDR if IR(0) = 1   # Load value into R0 if IR(0) = 0 else
if IR(0) = 1 into R1
PO4: SP ← SP + 1   # Increment stack pointer

halt                    11110000
H1: PC ← 0  # Stop execution by resetting PC
```

- Pin Assignments

Pin Assignments for testing the Microsequencer without a PC Register

Node Name	Direction	Location
addr_hex0_disp0[0]	Output	PIN_G18
addr_hex0_disp0[1]	Output	PIN_F22
addr_hex0_disp0[2]	Output	PIN_E17
addr_hex0_disp0[3]	Output	PIN_L26
addr_hex0_disp0[4]	Output	PIN_L25
addr_hex0_disp0[5]	Output	PIN_J22
addr_hex0_disp0[6]	Output	PIN_H22
addr_hex1_disp1[0]	Output	PIN_M24
addr_hex1_disp1[1]	Output	PIN_Y22
addr_hex1_disp1[2]	Output	PIN_W21
addr_hex1_disp1[3]	Output	PIN_W22
addr_hex1_disp1[4]	Output	PIN_W25
addr_hex1_disp1[5]	Output	PIN_U23
addr_hex1_disp1[6]	Output	PIN_U24
button	Input	PIN_R24
clear	Input	PIN_AB28
ctrlSignals[0]	Output	PIN_H19
ctrlSignals[1]	Output	PIN_J19
ctrlSignals[2]	Output	PIN_E18
ctrlSignals[3]	Output	PIN_F18
ctrlSignals[4]	Output	PIN_F21
ctrlSignals[5]	Output	PIN_E19
ctrlSignals[6]	Output	PIN_F19
ctrlSignals[7]	Output	PIN_G19
M_disp[0]	Output	PIN_AA25
M_disp[1]	Output	PIN_AA26
M_disp[2]	Output	PIN_Y25
M_disp[3]	Output	PIN_W26
M_disp[4]	Output	PIN_Y26
M_disp[5]	Output	PIN_W27
M_disp[6]	Output	PIN_W28
useqEnOut	Output	PIN_H15

Pin Assignments for testing the Microsequencer with a PC Register

Node Name	Direction	Location
addr_hex0_disp0[0]	Output	PIN_G18
addr_hex0_disp0[1]	Output	PIN_F22
addr_hex0_disp0[2]	Output	PIN_E17
addr_hex0_disp0[3]	Output	PIN_L26
addr_hex0_disp0[4]	Output	PIN_L25
addr_hex0_disp0[5]	Output	PIN_J22
addr_hex0_disp0[6]	Output	PIN_H22
addr_hex1_disp1[0]	Output	PIN_M24
addr_hex1_disp1[1]	Output	PIN_Y22
addr_hex1_disp1[2]	Output	PIN_W21
addr_hex1_disp1[3]	Output	PIN_W22
addr_hex1_disp1[4]	Output	PIN_W25
addr_hex1_disp1[5]	Output	PIN_U23
addr_hex1_disp1[6]	Output	PIN_U24
button	Input	PIN_R24
clear	Input	PIN_AB28
ctrlSignals[0]	Output	PIN_H19
ctrlSignals[1]	Output	PIN_J19
ctrlSignals[2]	Output	PIN_E18
ctrlSignals[3]	Output	PIN_F18
ctrlSignals[4]	Output	PIN_F21
ctrlSignals[5]	Output	PIN_E19
ctrlSignals[6]	Output	PIN_F19
ctrlSignals[7]	Output	PIN_G19
M_disp[0]	Output	PIN_AA25
M_disp[1]	Output	PIN_AA26
M_disp[2]	Output	PIN_Y25
M_disp[3]	Output	PIN_W26
M_disp[4]	Output	PIN_Y26
M_disp[5]	Output	PIN_W27
M_disp[6]	Output	PIN_W28
pc_Seg0[0]	Output	PIN_AB19
pc_Seg0[1]	Output	PIN_AA19
pc_Seg0[2]	Output	PIN_AG21
pc_Seg0[3]	Output	PIN_AH21
pc_Seg0[4]	Output	PIN_AE19
pc_Seg0[5]	Output	PIN_AF19
pc_Seg0[6]	Output	PIN_AE18
pc_Seg1[0]	Output	PIN_AD18
pc_Seg1[1]	Output	PIN_AC18
pc_Seg1[2]	Output	PIN_AB18
pc_Seg1[3]	Output	PIN_AH19
pc_Seg1[4]	Output	PIN_AG19
pc_Seg1[5]	Output	PIN_AF18
pc_Seg1[6]	Output	PIN_AH18
useqEnOut	Output	PIN_H15

Pin Assignments for reading and writing to/from RAM using a Microsequencer

Node Name	Direction	Locati
clear	Input	PIN_AB28
ctrlSignals[0]	Output	PIN_H19
ctrlSignals[1]	Output	PIN_J19
ctrlSignals[2]	Output	PIN_E18
ctrlSignals[3]	Output	PIN_F18
ctrlSignals[4]	Output	PIN_F21
ctrlSignals[5]	Output	PIN_E19
ctrlSignals[6]	Output	PIN_F19
ctrlSignals[7]	Output	PIN_G19
marData[7]	Input	PIN_Y23
marData[6]	Input	PIN_Y24
marData[5]	Input	PIN_AA22
marData[4]	Input	PIN_AA23
marData[3]	Input	PIN_AA24
marData[2]	Input	PIN_AB23
marData[1]	Input	PIN_AB24
marData[0]	Input	PIN_AC24
marSeqHi[0]	Output	PIN_M24
marSeqHi[1]	Output	PIN_Y22
marSeqHi[2]	Output	PIN_W21
marSeqHi[3]	Output	PIN_W22
marSeqHi[4]	Output	PIN_W25
marSeqHi[5]	Output	PIN_U23
marSeqHi[6]	Output	PIN_U24
marSeqLo[0]	Output	PIN_G18
marSeqLo[1]	Output	PIN_F22
marSeqLo[2]	Output	PIN_E17
marSeqLo[3]	Output	PIN_L26
marSeqLo[4]	Output	PIN_L25
marSeqLo[5]	Output	PIN_J22
marSeqLo[6]	Output	PIN_H22
mdrSeqHi[0]	Output	PIN_V21
mdrSeqHi[1]	Output	PIN_U21
mdrSeqHi[2]	Output	PIN_AB20
mdrSeqHi[3]	Output	PIN_AA21
mdrSeqHi[4]	Output	PIN_AD24
mdrSeqHi[5]	Output	PIN_AF23
mdrSeqHi[6]	Output	PIN_Y19
mdrSeqLo[0]	Output	PIN_AA25
mdrSeqLo[1]	Output	PIN_AA26
mdrSeqLo[2]	Output	PIN_Y25
mdrSeqLo[3]	Output	PIN_W26
mdrSeqLo[4]	Output	PIN_Y26
mdrSeqLo[5]	Output	PIN_W27
mdrSeqLo[6]	Output	PIN_W28
nextAddrHi[0]	Output	PIN_AD18
nextAddrHi[1]	Output	PIN_AC18
nextAddrHi[2]	Output	PIN_AB18
nextAddrHi[3]	Output	PIN_AH19
nextAddrHi[4]	Output	PIN_AG19
nextAddrHi[5]	Output	PIN_AF18
nextAddrHi[6]	Output	PIN_AH18
nextAddrLo[0]	Output	PIN_AB19
nextAddrLo[1]	Output	PIN_AA19
nextAddrLo[2]	Output	PIN_AG21
nextAddrLo[3]	Output	PIN_AH21
nextAddrLo[4]	Output	PIN_AE19
nextAddrLo[5]	Output	PIN_AF19
nextAddrLo[6]	Output	PIN_AE18
useqEnOut	Output	PIN_H15

Pin assignments for the Microsequencer-based CPU implementation

Node Name	Direction	Location
button	Input	PIN_M21
clear	Input	PIN_Y23
R0SegHi[0]	Output	PIN_M24
R0SegHi[1]	Output	PIN_Y22
R0SegHi[2]	Output	PIN_W21
R0SegHi[3]	Output	PIN_W22
R0SegHi[4]	Output	PIN_W25
R0SegHi[5]	Output	PIN_U23
R0SegHi[6]	Output	PIN_U24
R0SegLo[0]	Output	PIN_G18
R0SegLo[1]	Output	PIN_F22
R0SegLo[2]	Output	PIN_E17
R0SegLo[3]	Output	PIN_L26
R0SegLo[4]	Output	PIN_L25
R0SegLo[5]	Output	PIN_J22
R0SegLo[6]	Output	PIN_H22
R1SegHi[0]	Output	PIN_V21
R1SegHi[1]	Output	PIN_U21
R1SegHi[2]	Output	PIN_AB20
R1SegHi[3]	Output	PIN_AA21
R1SegHi[4]	Output	PIN_AD24
R1SegHi[5]	Output	PIN_AF23
R1SegHi[6]	Output	PIN_Y19
R1SegLo[0]	Output	PIN_AA25
R1SegLo[1]	Output	PIN_AA26
R1SegLo[2]	Output	PIN_Y25
R1SegLo[3]	Output	PIN_W26
R1SegLo[4]	Output	PIN_Y26
R1SegLo[5]	Output	PIN_W27
R1SegLo[6]	Output	PIN_W28
Z_FLAG[0]	Output	PIN_H15
pcSegHi[0]	Output	PIN_AD18
pcSegHi[1]	Output	PIN_AC18
pcSegHi[2]	Output	PIN_AB18
pcSegHi[3]	Output	PIN_AH19
pcSegHi[4]	Output	PIN_AG19
pcSegHi[5]	Output	PIN_AF18
pcSegHi[6]	Output	PIN_AH18
pcSegLo[0]	Output	PIN_AB19
pcSegLo[1]	Output	PIN_AA19
pcSegLo[2]	Output	PIN_AG21
pcSegLo[3]	Output	PIN_AH21
pcSegLo[4]	Output	PIN_AE19
pcSegLo[5]	Output	PIN_AF19
pcSegLo[6]	Output	PIN_AE18
spSegHi[0]	Output	PIN_AD17
spSegHi[1]	Output	PIN_AE17
spSegHi[2]	Output	PIN_AG17
spSegHi[3]	Output	PIN_AH17
spSegHi[4]	Output	PIN_AF17
spSegHi[5]	Output	PIN_AG18
spSegHi[6]	Output	PIN_AA14
spSegLo[0]	Output	PIN_AA17
spSegLo[1]	Output	PIN_AB16
spSegLo[2]	Output	PIN_AA16
spSegLo[3]	Output	PIN_AB17
spSegLo[4]	Output	PIN_AB15
spSegLo[5]	Output	PIN_AA15
spSegLo[6]	Output	PIN_AC17

- Memory Initialization Files

Memory Initialization File for testing the Microsequencer without a PC Register

```
-- uROM Test File
-- 8-bit control signals
WIDTH=17;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
 [0..FF]: 00000000000000000;
--   uop
--   01234567M01234567
 00: 00000000000000001; -- control signal = 00000000, next address = 0x01
 01: 00000001000000010; -- control signal = 00000001, next address = 0x02
 02: 00000010000000011; -- control signal = 00000010, next address = 0x03
 03: 00000011000000100; -- control signal = 00000011, next address = 0x04
 04: 00000100100000000; -- control signal = 00000100, use MAP, go to address 0x30
 30: 00000101000110001; -- control signal = 00000101, next address = 0x31
 31: 00000110000110010; -- control signal = 00000110, next address = 0x32
 32: 00000111000110011; -- control signal = 00000111, next address = 0x33
 33: 00001000000000001; -- control signal = 00001000, next address = 0x01
END;
```

Memory Initialization File for testing the Microsequencer with a PC Register

```
-- uROM Test File
-- 8-bit control signals
WIDTH=17;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
 [0..FF]: 00000000000000000;
--   uop
--   01234567M01234567
 00: 00000000000000001; -- control signal = 00000000, next address = 0x01
 01: 00100001000000010; -- control signal = 00100001, next address = 0x02, clear PC, PC = 0x00
 02: 10000010000000011; -- control signal = 10000010, next address = 0x03, inc PC, PC = 0x01
 03: 10000011000000100; -- control signal = 10000011, next address = 0x04, inc PC, PC = 0x02
 04: 00000100100000000; -- control signal = 00000100, use MAP, go to address 0x30
 30: 01000101000110001; -- control signal = 01000101, next address = 0x31, load PC, PC = 0xAA
 31: 00000110000110010; -- control signal = 00000110, next address = 0x32
 32: 10000111000110011; -- control signal = 10000111, next address = 0x33, inc PC, PC = 0xAB
 33: 00001000000000001; -- control signal = 00001000, next address = 0x01,
END;
```

Memory Initialization Files for reading and writing to/from RAM using a Microsequencer

```
-- uROM Test File
-- 8-bit control signals

WIDTH=17;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
   [0..FF]: 00000000000000000;
--     uop
--     01234567M01234567
   00: 00000000000000001; -- control signal (cs) = 00000000, next address (na) = 0x01
   01: 10000001000000010; -- cs = 10000001, na = 0x02, Load MAR from sw, MAR = 0x54
   02: 01000010000000011; -- cs = 01000010, na = 0x03, MDR <- M[MAR], MDR = 0xFC
   03: 10000011000000100; -- cs = 10000011, na = 0x04, Load MAR from sw, MAR = 0x55
   04: 01000100000000101; -- cs = 01000100, na = 0x05, MDR <- M[MAR], MDR = 0xDD
   05: 10000101000000110; -- cs = 10000101, na = 0x06, Load MAR from sw, MAR = 0x56
   06: 00100110000000111; -- cs = 00100110, na = 0x07, M[MAR] <- MDR
   07: 10000111000001000; -- cs = 10000111, na = 0x08, Load MAR from sw, MAR = 0X54
   08: 01001000000001001; -- cs = 01001000, na = 0x09, MDR <- M[MAR], MDR = 0xFC
   09: 10001001000001010; -- cs = 10001001, na = 0x0A, Load MAR from sw, MAR = 0x56
   0A: 01001010000000000; -- cs = 01001010, na = 0x00, MDR <- M[MAR], MDR = 0xDD
END;
```

```
-- RAM Test File

WIDTH=8;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
   [0..FF]: 0;
   54: FC;
   55: DD;
   56: AA;
END;
```

Memory Initialization Files for the ROM of the Microsequencer inside of our CPU

```
WIDTH=33;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

-- Current Address: UOP(1-24)MAP(8)NEXTADDRESS(7-0)
-- 0xADDRESS: 123456789(10)(11)(12)(13)(14)(15)(16)(17)(18)(19)(20)(21)(22)(23)(24)876543210

CONTENT BEGIN
   [0..FF]: 000000000000000000000000000000000;
   00: 000000000000000000000000000000001;
   01: 000000100000000000000000000000010;
   02: 010000001000000000000000000000011;
   03: 001000000000000000000000100000001;
   10: 000000100000000000000000000010001;
   11: 010000001000000000000000000010010;
   12: 000000000001001000000000000000001;
   20: 000000100000000000000000000100001;
   21: 010000001000000000000000000100010;
   22: 000001100000000000000000000100011;
   23: 000000001000000000000000000100100;
   24: 000000000001001000000000000000001;
   30: 000000100000000000000000000110001;
   31: 010000001000000000000000000110010;
   32: 000101100000000000000000000110011;
   33: 000000011000000000000000000110100;
   34: 000100000000000000000000000000001;
   40: 000000000101101000000000000000001;
   50: 000000000011011100000000000000001;
   60: 000000000011011000000000000000001;
   70: 000000000000000010100000000000001;
   80: 000000000000000011100000000000001;
   90: 000000100000000000000000000010001;
   91: 000000010000000000000000000010010;
   92: 000000000000000000010000000000001;
   A0: 000000100000000000000000000010100001;
   A1: 010000001000000000000000000010100010;
   A2: 000000000000000000010000000000001;
   B0: 000000100000000000000000000010110001;
   B1: 010000001000000000000000000010110010;
   B2: 000000000000000000010000000100000001;
   C0: 000010100000000000000000000011000001;
   C1: 000000001000000000000000000011000010;
   C2: 000000000001001000000000000000001;
   D0: 000000000000000000000000000001011010001;
   D1: 000010100000000000000000000011010010;
   D2: 000000011000000000000000000011010011;
   D3: 000100000000000000000000000000000001;
   E0: 000010100000000000000000000011100001;
   E1: 000000001000000000000000000011100010;
   E2: 000000000001001000000000000011100011;
   E3: 000000000000000000000000000100000001;
   F0: 100000000000000000000000000000000001;
END;
```

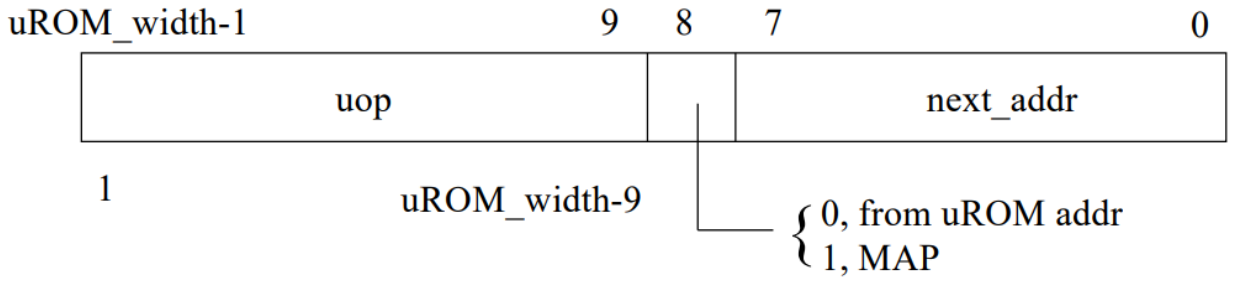
ram_files

- Tables

CPU Instructions

Instruction	Instruction code	Operation
NOP	00000000	No operation
LOADI Rn,X	0001000n X	Rn ← X
LOAD Rn,X	0010000n X	Rn ← M[X]
STORE X,Rm	0011000m X	M[X] ← Rm
MOVE Rn,Rm	0100000n	Rn ← Rm, m ≠ n
ADD Rn,Rm	0101000n	Rn ← Rn + Rm, m ≠ n
SUB Rn,Rm	0110000n	Rn ← Rn – Rm, m ≠ n
TESTNZ Rm	0111000m	Z ← not V, V = OR of the bits of Rm
TESTZ Rm	1000000m	Z ← V, V = OR of the bits of Rm
JUMP X	10010000 X	PC ← X
JUMPZ X	10100000 X	If (Z = 1) then PC ← X
LOADSP X	10110000 X	SP ← X
PEEP Rn	1100000n	Rn ← M[SP]
PUSH Rn	1101000n	M[--SP] ← Rn Pre-increment
POP Rn	1110000n	Rn ← M[SP++] Post-decrement
HALT	11110000	PC ← 0, stop microsequencer

ROM Output



- Spreadsheets

Microsequencer ROM

FPGA Pin Layout