DESIGN PROJECT REPORT

EEX7436
PROCESSOR DESIGN
FUZZY MATRIX OPERATION UNIT

BY

M.N.M. FAZEEL

220260734

SUBMITED TO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING TECHNOLOGY

THE OPEN UNIVERSITY OF SRI LANKA

AT

COLOMBO

ON

17/11/2025

INTRODUCTION

What is an FPU?

Fuzzy Processing Unit (FPU) is a specialized hardware component designed to handle operations based on fuzzy logic, which do the reasoning which is approximating rather than fixed and exact. Unlike binary logic, where values are strictly 0 or 1, fuzzy logic allows values to exist on a continuum between 0 and 1.

Functions of FPU

- **Fuzzification**
  The process of converting crisp (precise, real-world) inputs into fuzzy values, which emphasize degrees of truth or belonging in fuzzy sets.

  Fuzzification uses membership functions (μ) for each fuzzy set.

  For example:

  - Crisp input: Temperature = 32°C
  - Fuzzy sets: Cold, Warm, Hot
  - Membership degrees:
    - $\mu_{Cold}(32) = 0.1$
    - $\mu_{Warm}(32) = 0.6$
    - $\mu_{Hot}(32) = 0.3$

  This outputs the input is mostly "Warm" and a bit "Hot".

  In FMU Values like 0.6, 0.3 are stored in matrix registers as 16-bit Q0.16 fuzzy values. Represented as fuzzy input matrices.

- **Fuzzy Rule Processing (Inference Engine)**

  The fuzzy logic engine applies rules and logic operators to the fuzzified inputs to determine fuzzy outputs.

  Processing fuzzy values to get an output:

  Let's Assume;
  Hot = 0.6
  Humid = 0.7

  Appling rules:

  ```
  IF Hot AND Humid THEN increaseSpeedFan
  → min(0.6, 0.7) = 0.6
  ```

  In this fan speed increased by the factor of 0.6

- **Defuzzification**
  Defuzzification converts the fuzzy output values like, fan = 0.6 high, 0.3 medium, 0.1 low into a single crisp number that a machine can work with like fan speed = 70%.

Common Defuzzification Methods:

Method	Description	Use Case
Centroid	Weighted average of membership areas	Most common
Max Membership	Choose output with highest membership	Simple, fast
Weighted Avg	Average of values weighted by membership	Good for linear outputs

Key Characteristics of FPU:

- Operates on fuzzy numbers; real values in the range [0.0, 1.0] to represent uncertainty, vagueness, and imprecise reasoning found in natural language, expert systems, and AI.
- Efficiently performs fuzzy matrix computations including addition (max), multiplication (min), subtraction enabling parallel and scalable rule-based decision making.
- Specifically designed to offload and accelerate fuzzy logic algorithms, which are computationally intensive and inefficient on general-purpose CPUs. Ideal for applications in:
    - Artificial Intelligence
    - Machine Learning
    - Robotics
    - Autonomous control systems
    - Natural Language Processing
- Can be embedded as a coprocessor within general-purpose processors, making it adaptable for embedded systems, SoCs, or FPGA/ASIC implementations.
- Simplified arithmetic enables high-speed and low-power computation, especially effective when working with large-scale fuzzy matrices or when multiple fuzzy rules must be evaluated simultaneously.

Requirements of Fuzzy Matrix Unit

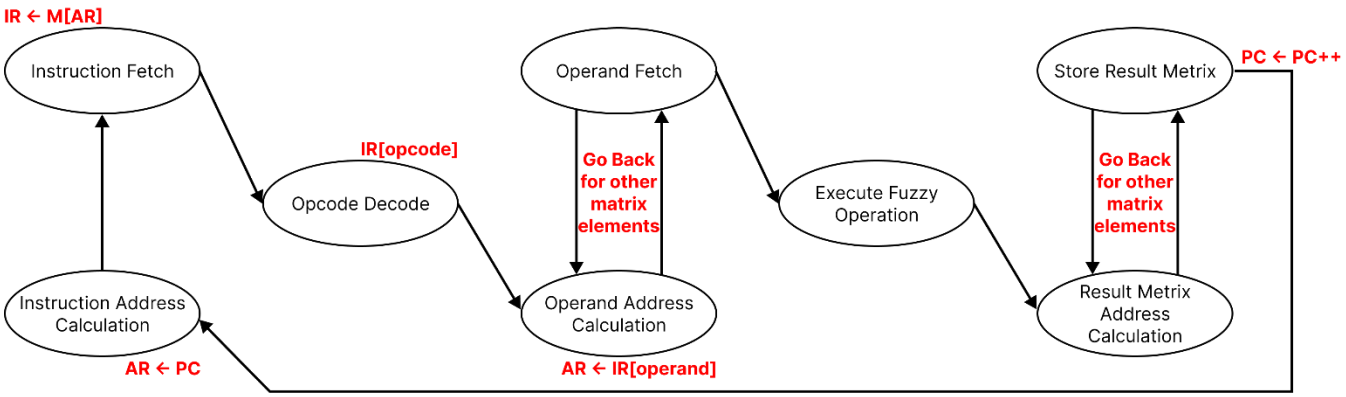
- Handle fuzzy numbers and perform operations based on fuzzy logic like:
  - Fuzzy addition: $a + b = max(a, b)$
  - Fuzzy subtraction: $a - b = \begin{cases} a & if\ a > b \\ 0 & if\ a \leq b \end{cases}$
  - Fuzzy multiplication: $a \cdot b = min(a, b)$
- Perform all calculations through registers or RAM to maximize performance.
- Provide ISA for:
  - Define matrices and manipulate data.
  - Perform fuzzy matrix operations.
  - Loading and storing matrices from and into memory.
  - Creating sub matrices and manipulation.
  - Branching.

Working Procedure of the FMU

The FMU operates as a specialized co-processor for the FPU, designed to handle fuzzy matrix operations. Its internal architecture contains:

- **Instruction Decoder:** Decodes FMU-specific instructions.
- **Matrix Register Bank:** Temporarily stores matrices at the processing for quick access and manipulation.
- **Fuzzy Arithmetic Logic Unit:** Performs fuzzy addition, multiplication, and subtraction based on pre-defined operations.
- **Control Unit:** Manages the sequencing of operations and coordinates between different FMU components.
- **Memory Interface Unit:** Handles data transfer between the FMU registers and system memory.

Basic Operation Flow



After completion the FMU signals the FPU, allowing the FPU to proceed with subsequent operations.

Design Steps

Proposed design uses RISC architecture to simplify the design process.

- RISC uses simple, uniform instructions (one instruction per operation).
- It is easy to implement on FPGA using VHDL because it uses fixed 32 bits instructions so the instruction decoding logic is simple to implement.
- Register based design and operand must be in registers so we can implement cleaner instruction encoding for ISA.

Defining Registers

- This design uses fixed registers for store matrix meta data and Buffer (RAM or Memory depending on the requirements and available resources) to store Matrices.
- There are 8 fixed registers (R0 – R4), each register stores:
  - Base Address (32 bits): Points to the first memory location of the matrix elements. Matrices are stored in the memory following the row-major order.
  - Row Count (8 bits): Used to specify the row count of the matrix.
  - Column Count (8 bits): Used to specify column count of the matrix.
- The benefit of using the register to cache identifiers is that it simplifies the instruction encoding because it does not require to encode full memory addresses in every instruction, we can simply call the register which holds the metrics meta data.
- We can easily expand the supported matrix size without adding extra registers.
- Accessing matrix element using matrix meta data stored in the register:

  $Matrix\ Register\ R0$:

  $Base\ address\ =\ 0x2000$
  $Rows\ =\ 3$
  $Columns\ =\ 3$

  Accessing matrix element at $Row\ i, Columns\ j$ (for 8-bit number and 8-bit memory block):

$$Address\ =\ Base\ +\ (i\ \times\ Cols\ +\ j)$$

Defining Fuzzy Number Format

- For this design, choose to go with slightly modified Fixed-point (Q1.7).
- Modification: Fuzzy numbers are in the range of 0 to 1 so for this design there is no need for a signed bit hence Fixed-point (Q0.8).
- **What is Fixed-point?**
  - Fixed-point represents fractional numbers using only scaled integers, without using floating-point hardware. It is very useful in this design because floating point numbers are in the range of 0 to 1.
- **Why Fixed-point (Q0.8)?**
  - Large enough range and high precision (Values from 0 to ~ 0.9961 with precision of 1/256).
  - Simple ALU operations (Only need integer logic + bit shifting no need of full floating-point unit).

- Better memory and bandwidth utilization (Required memory to store 8×8 matrix is 64 Bytes).
- No floating-point hardware and simple integer operations.

- **How to Represent Fuzzy Values Using Fixed-point (Q0.8)?**
  - To store fuzzy value f:

$$Stored\ Value = f \times 2^8$$

  - To retrieve the value:

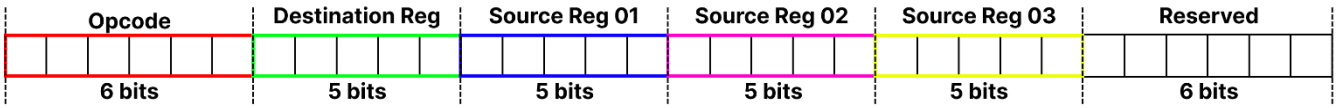
$$f = \frac{Stored\ Value}{2^8}$$

Decimal Value	Binary (8-bit)	Integer Stored
0.0	00000000	0
0.5	10000000	128
0.75	11000000	192
1.0	11111111	255

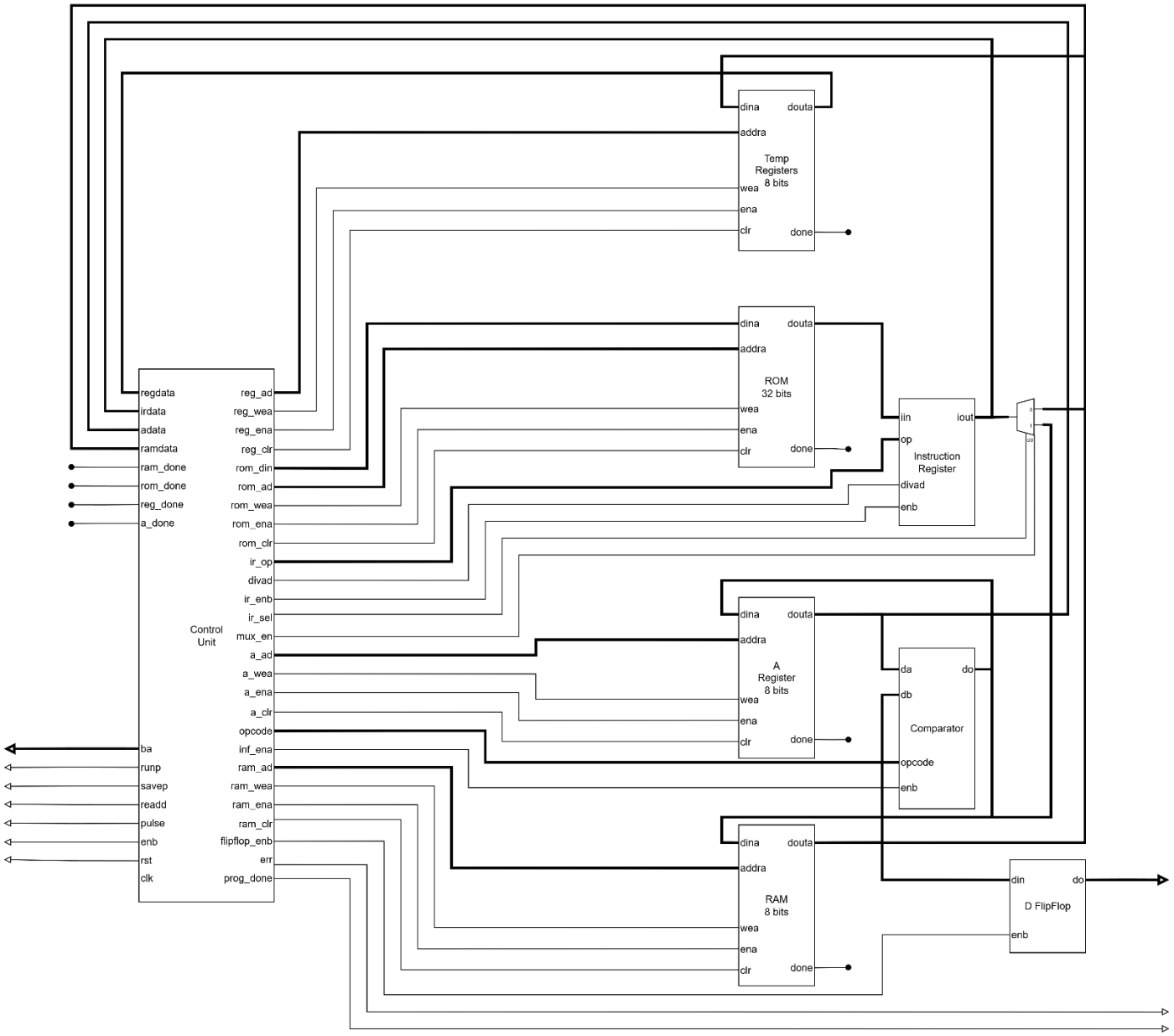
Assumptions for ISA

- Maximum matrix size: 32×32
- Fuzzy number size: $8\ bits$ (Fixed-point (Q0.8))
- Maximum Opcodes: 2^6 (Common in RISC architecture)
- Maximum possible matrix registers: 2^5 (fixed number of registers and memory stored matrices)
- Maximum possible buffer registers: 2^5 (can be used to modify matrix elements or act ass buffer for matrix operations)
- Matrix storage: Row major
- Matrix storage location: FPU RAM

Instruction Format



Block Diagram of the FMU



Components Designing of FMU

Memory Elements:

All memory elements of this processor are Block RAMs specially configured according to the functional needs of the FMU. This FMU uses Harvard architecture which has separate memories for data and program. This provides room for future development like pipelining because there is no memory bottleneck like in the Von-Neuman Architecture.

ROM (32bits word and 65KB depth)

ROM acts as the program memory of the processor which holds the instructions. Instructions can be written to the ROM using the programing mod of the processor.

Temp Register (8bits word and 16 Byte depth)

This register used to temporarily store the matrix meta data such as RAM address, row and column count of the matrix. This data is then used to access individual elements of the second matrix which that is processing. There are 4 registers namely Ra, Rb, Rc, Rd and each register has 4 bytes of memory first two memory locations hold the High byte of the RAM address and Low byte nest Row count and Column count respectively.

A Register (8bits word and 2048 Byte depth)

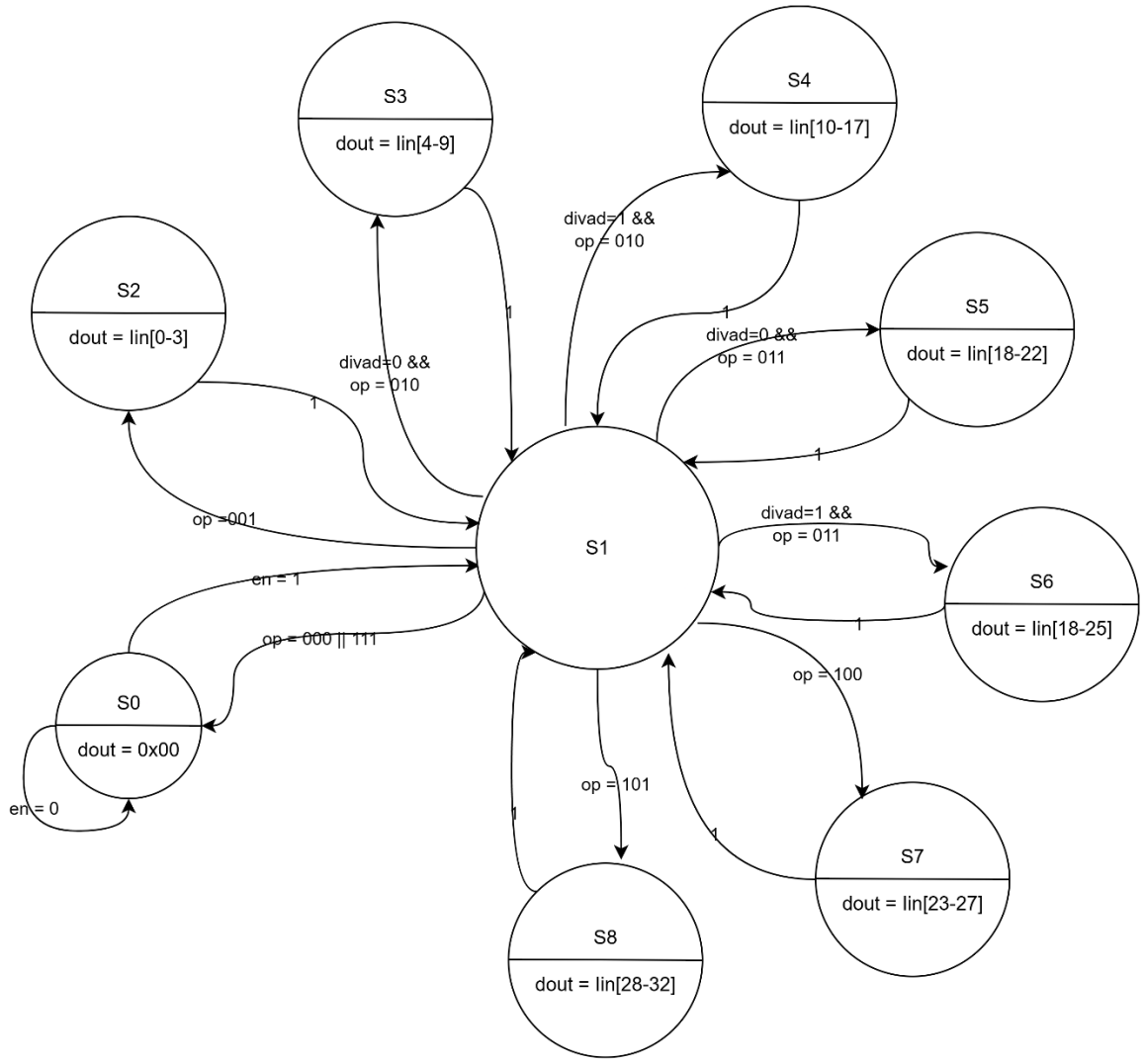
This acts as the accumulator of this processing unit. One matrix operand comes from the A register and other operand is accessed from the data memory using the meta data stored in the Temp Registers. Then after processing matrix element is written back to the accumulator and matrix should be manually written back to the data memory if needed. This is also advantage when it comes to pipelining later.

RAM (8bits word and 65KB depth)

This acts as the Data Memory of the FMU. This is used to store the matrixes and used to access the second operand of currently processing matrix.

Instruction Register

This is a 32bits wide register used to decode the instructions stored in the ROM when running a program. Memory decoding process is done as per the opcodes dispatched by the Control Unit.



Signals and Their Functions:

op → 000 and 111 resets the 8bit output to 0x00

001 outputs the opcode

010 outputs the first register address (destination) / high byte of the RAM address depending on the divad.

011 outputs the second register address / row count / low byte of the RAM address depending on the instruction or the divad.

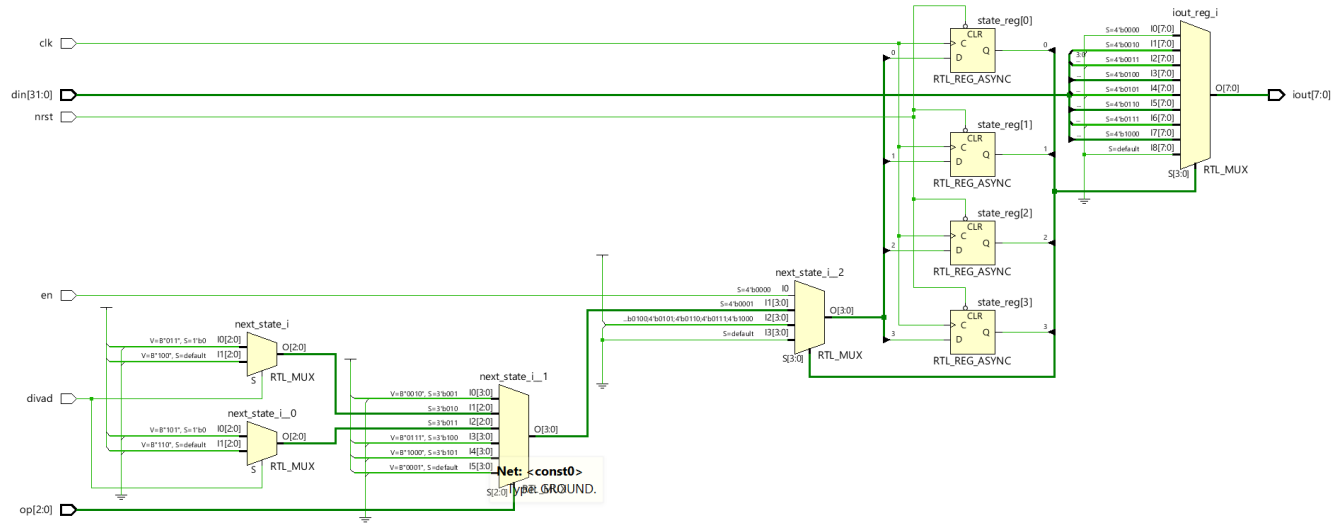
100 outputs the third register address / column count depending on the instruction.

101 outputs the 8bit data

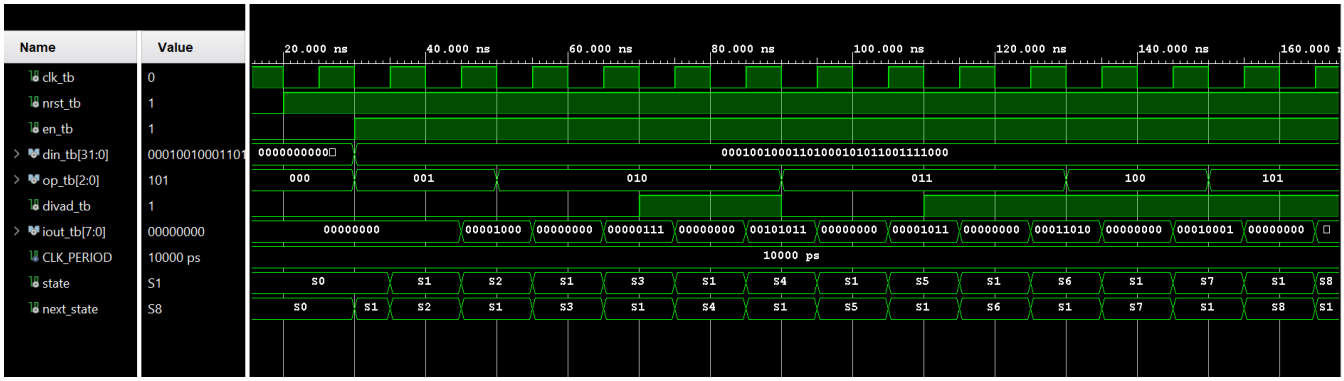
divad → When high and op = 010 outputs the high byte of the RAM address in the instruction.
When high and op = 011 outputs the low byte of the RAM address in the instruction.

enb → Enables the Instruction Register.

RTL Schematic of the Instruction Register



Timing Diagram of the Instruction Register



Comparator / Inference Unit

Handle fuzzy numbers and perform operations based on fuzzy logic:

- Fuzzy addition: $a + b = max(a, b)$
- Fuzzy subtraction: $a - b = \begin{cases} a & if\ a > b \\ 0 & if\ a \leq b \end{cases}$
- Fuzzy multiplication: $a \cdot b = min(a, b)$

Signals and Their Functions:

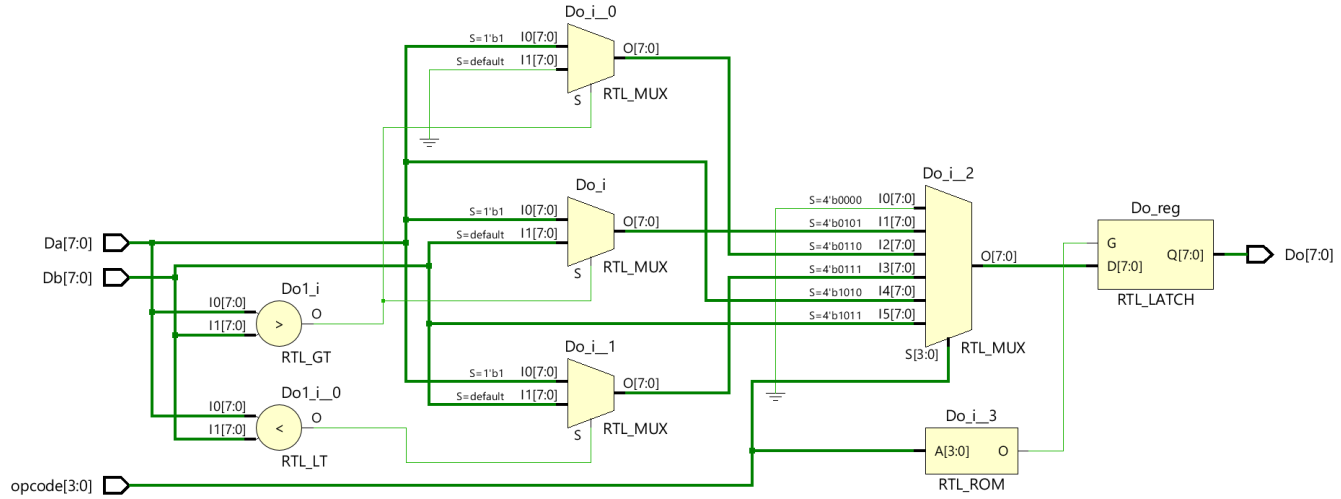
Da → First operand 8bit wide, which comes from the A register.

Db → Second operand 8 bit wide, which comes from the RAM.

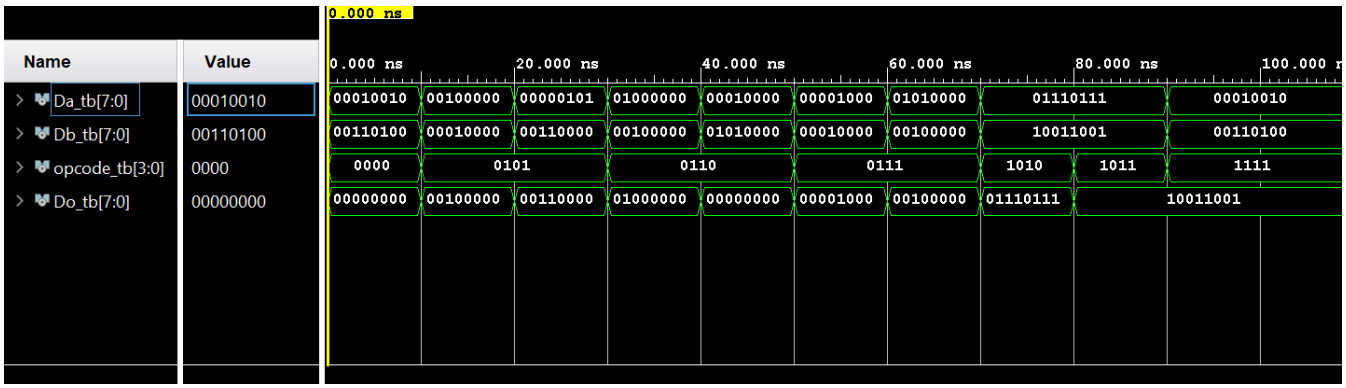
Opcode → Opcode this is used to identify the operation that has to do.

Do → Output of the unit. This is connected to the A register for the write back.

RTL Schematic of the Inference Unit



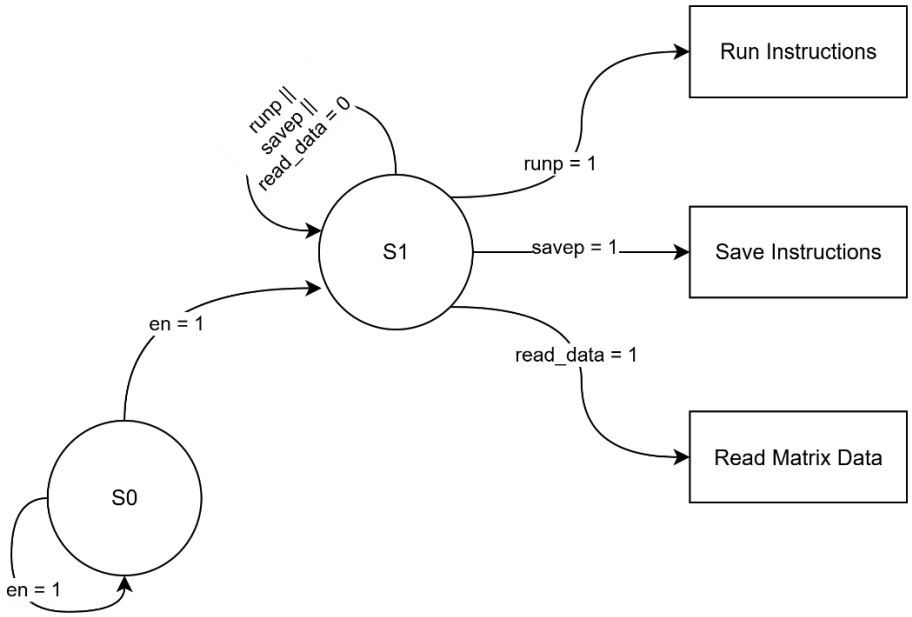
Timing Diagram of the Inference Unit



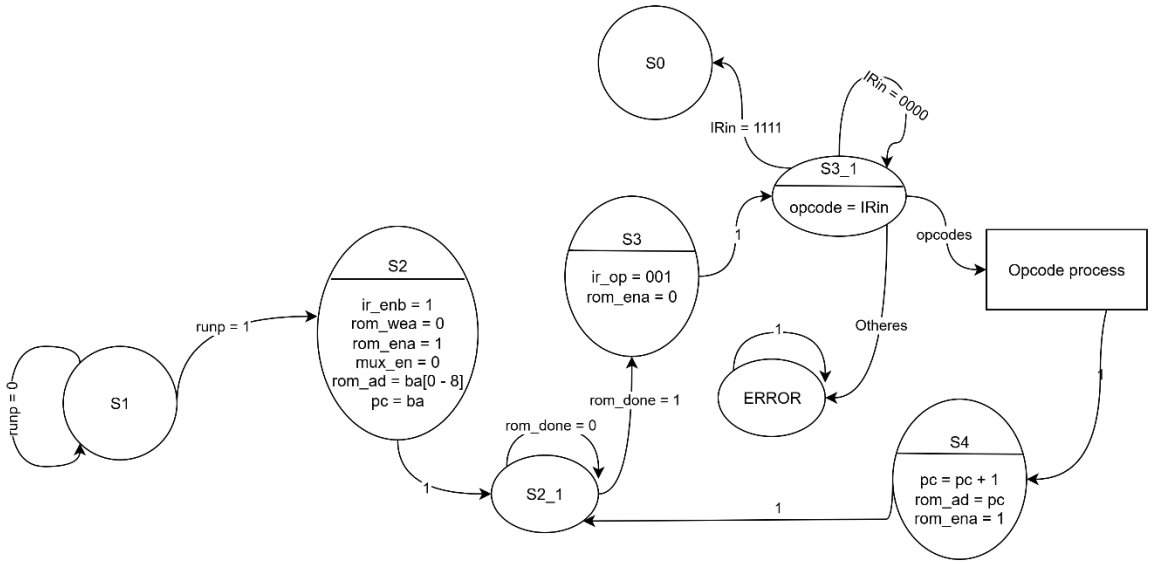
Control Unit

Coordinate and control the components of the FMU using different signals. Specialty of this control unit is the address decoding logic is built into the CU because there are many matrix elements to process offloading the address decoding logic to separate Address Register is difficult and increase the signals required to process and increase the clock cycles required for single instruction to execute drastically. By integrating the address decoding logic into CU I was able to keep maximum clock cycles per instruction 20 in best case. Otherwise, it requires additional 8 clock cycles at least.

Process Breck Down of the Control Unit

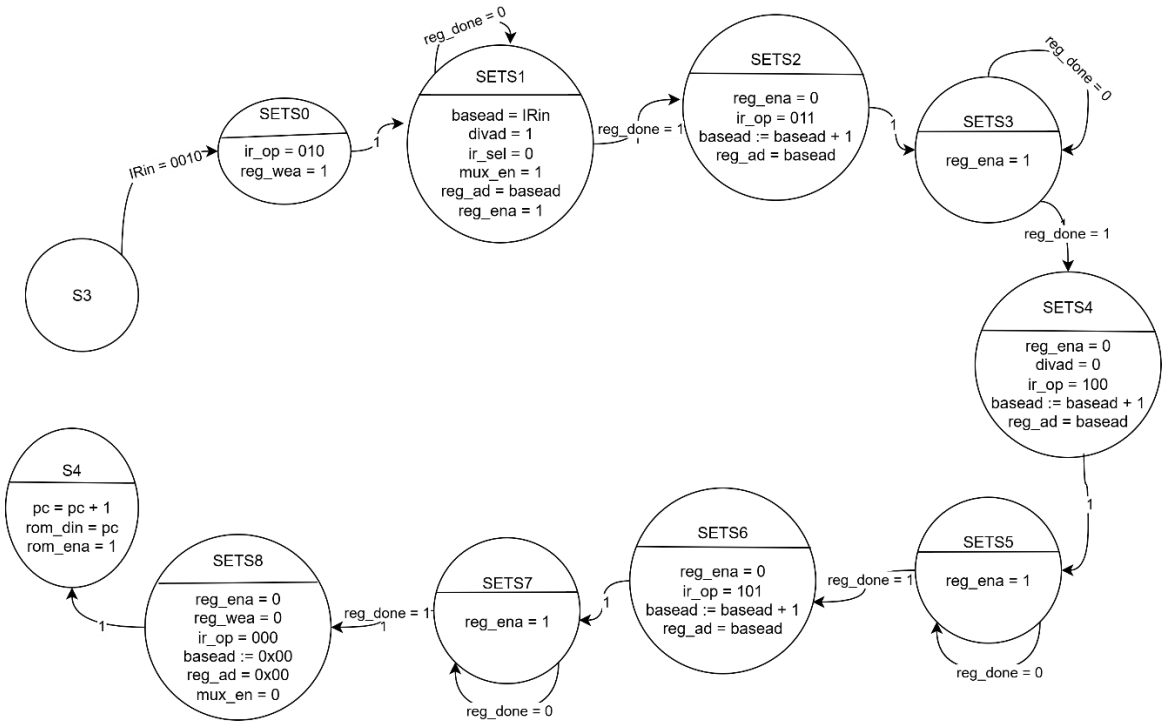


At the beginning it waits for enable signal to go high and then according to the selected mode it goes to respective processes to execute instructions / save instructions / read matrix data.



In run instruction process CU send signals to ROM and Instruction Register to fetch instruction and get the opcode. If the opcode is invalid outputs a error signal and system trap in an ERROR state until reset.

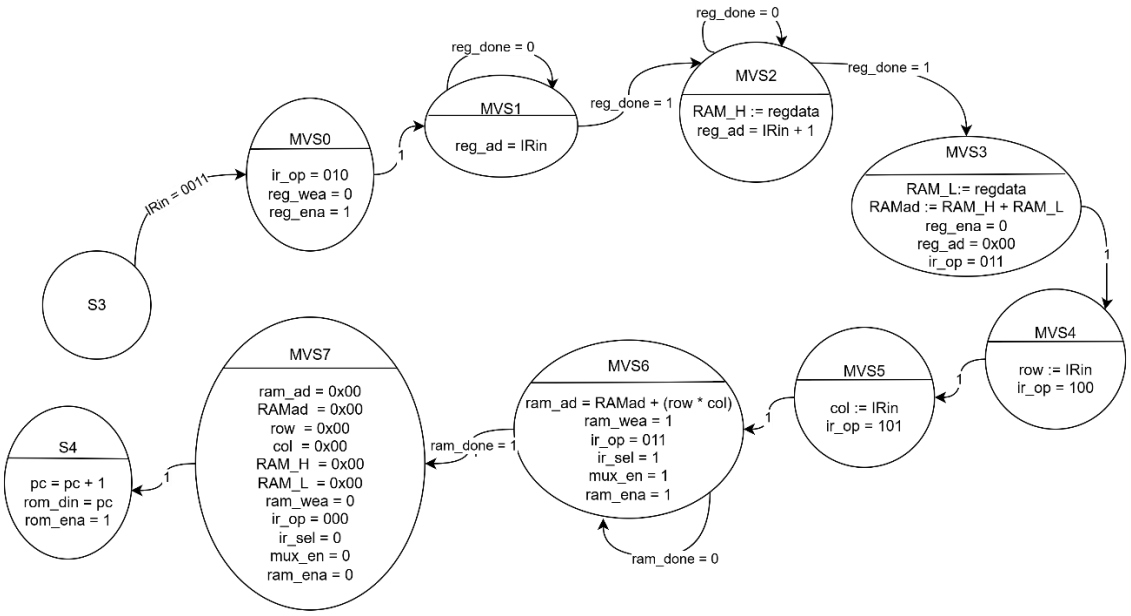
SETMX Opcode: 0001, R, RAMad, [row], [col]



Opcode 0001:

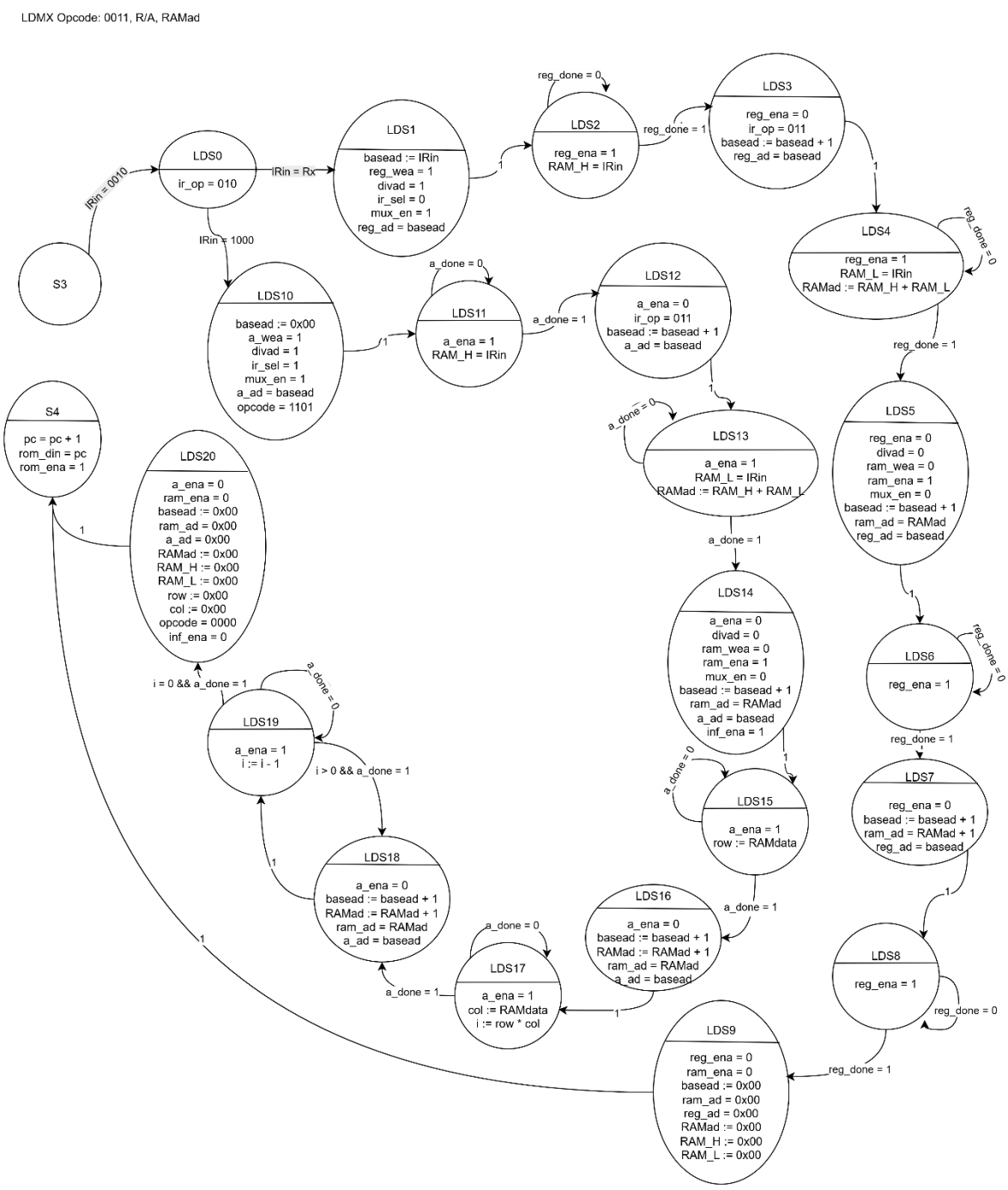
If the opcode is 0001 CU dispatch signals Temp Register and Instruction Register. In this operation the matrix metadata provided in the instruction is stored in the destination register.

MOVME Opcode: 0010, R, [row], [col], val



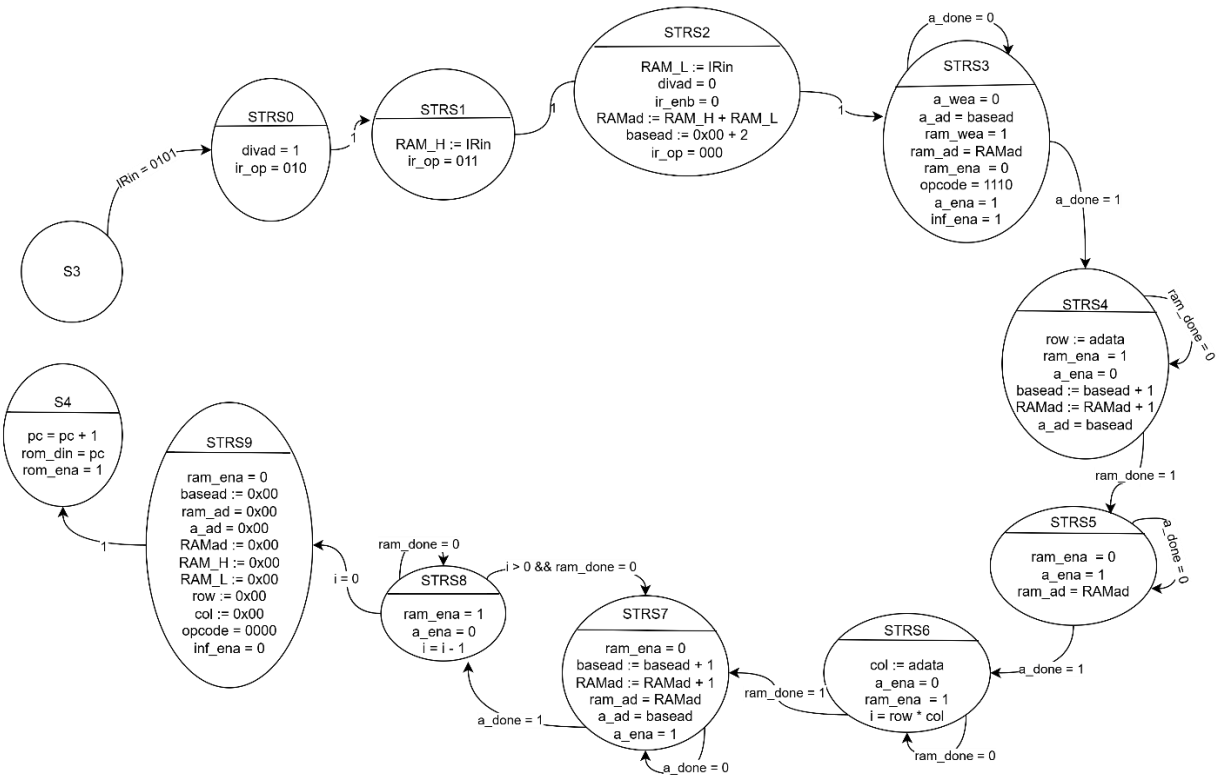
Opcode 0010:

In this operation 8bit value provided in the instruction moved to matrix pointed by the register R. Element number calculated using the row and column value provided in the instruction.



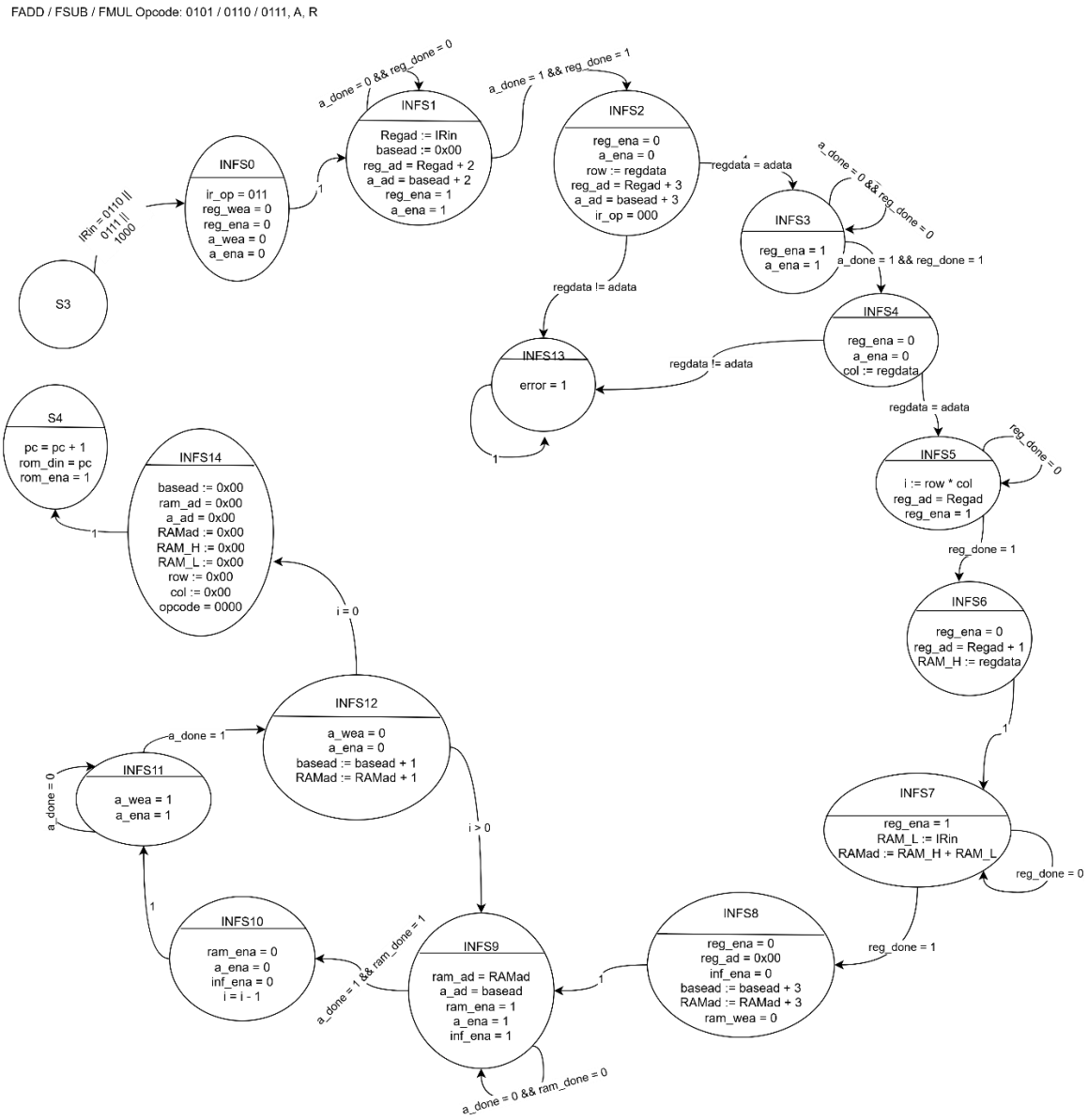
Opcode 0011:

In this operation if the destination is a Temp Register RAM address and the row and column data stored at that memory location is moved to the register R. If it is A register all the matrix elements also move along with the meta data.



Opcode 0100:

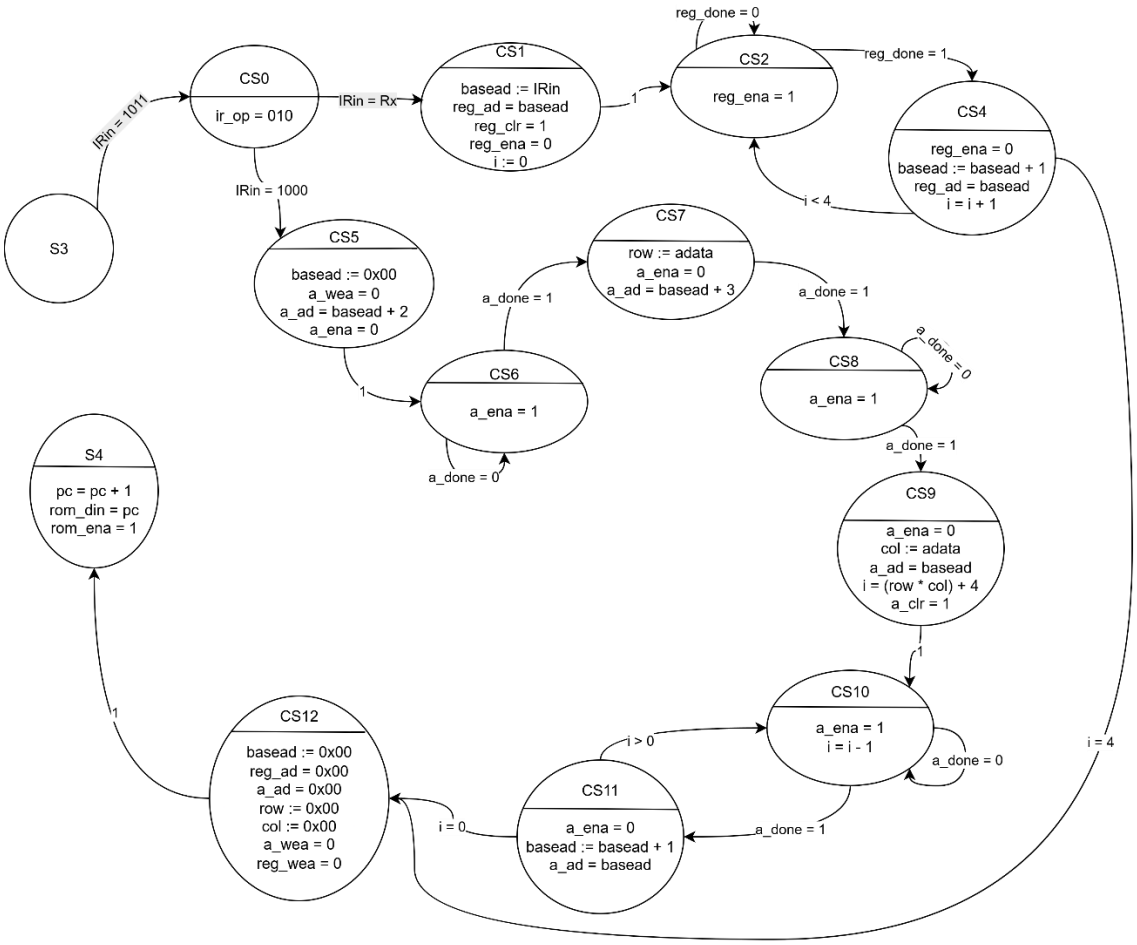
In this operation matrix stored in A Register is moved in to the RAM starting with the RAM address provided in the instruction.



Opcode 0101 / 0110 / 0111:

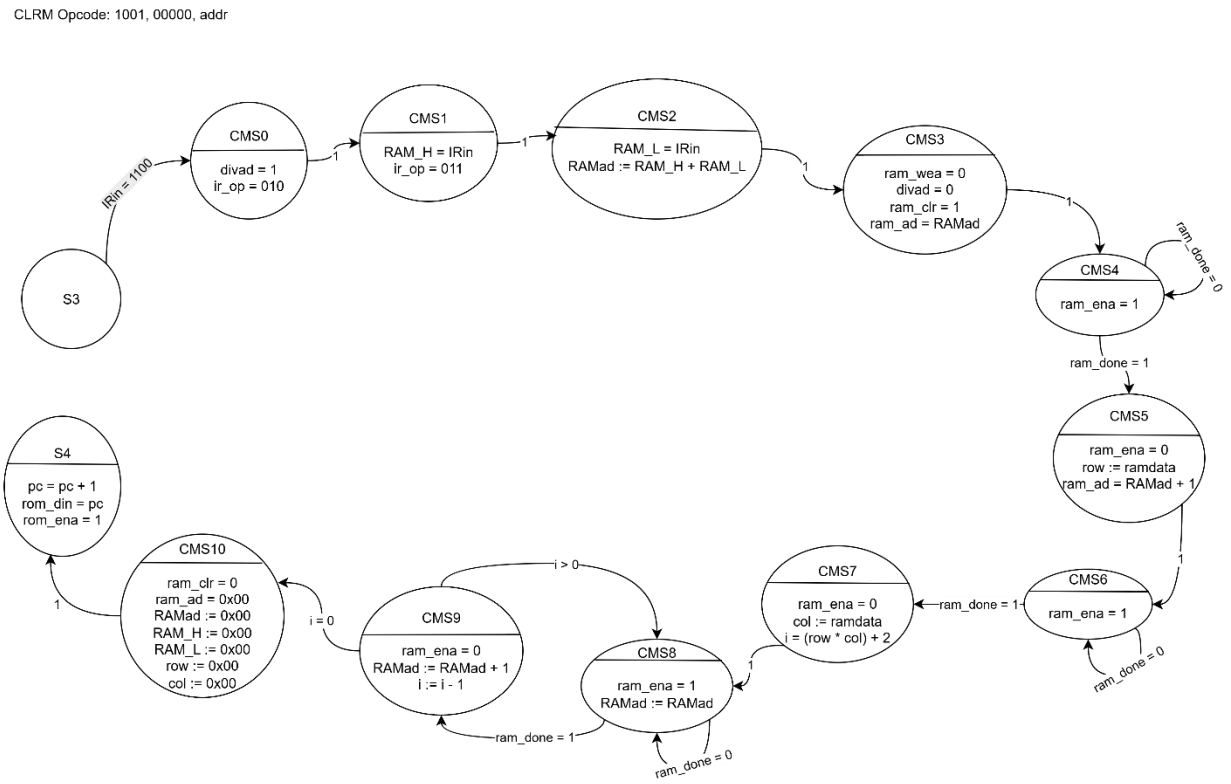
According to the opcode fuzzy matrix operations are performed on the matrices stored in A Register and matrix pointed by the Temp Register and write back to the A register.

CLR Opcode: 1000, A / R



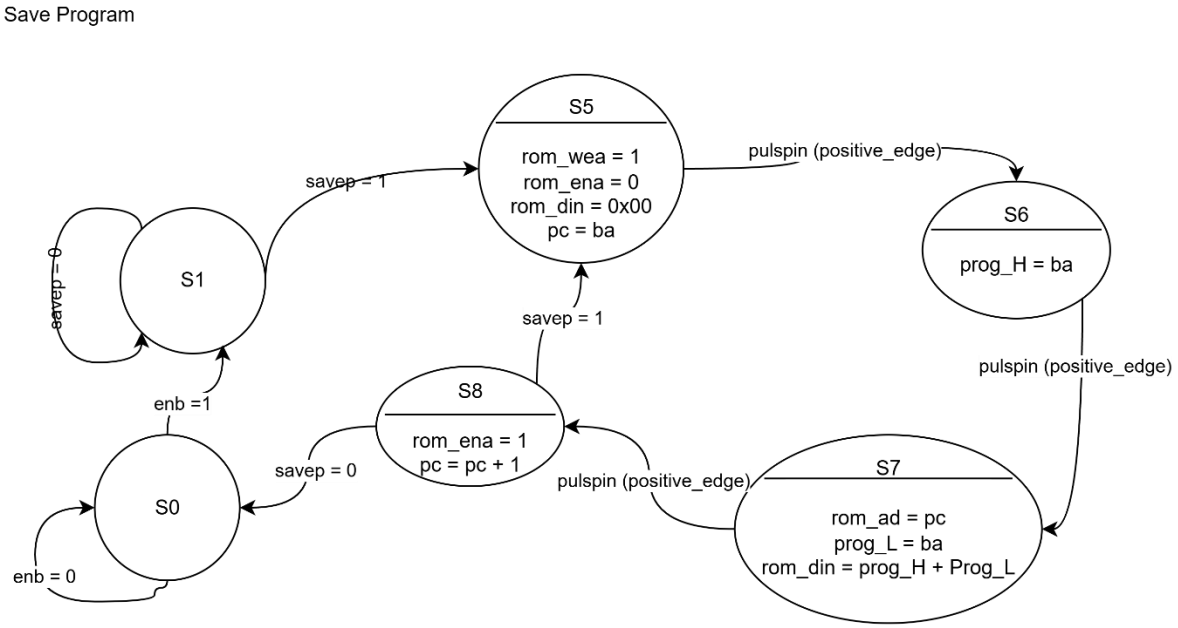
Opcode 1000:

Clear the registers pointed in the instruction.



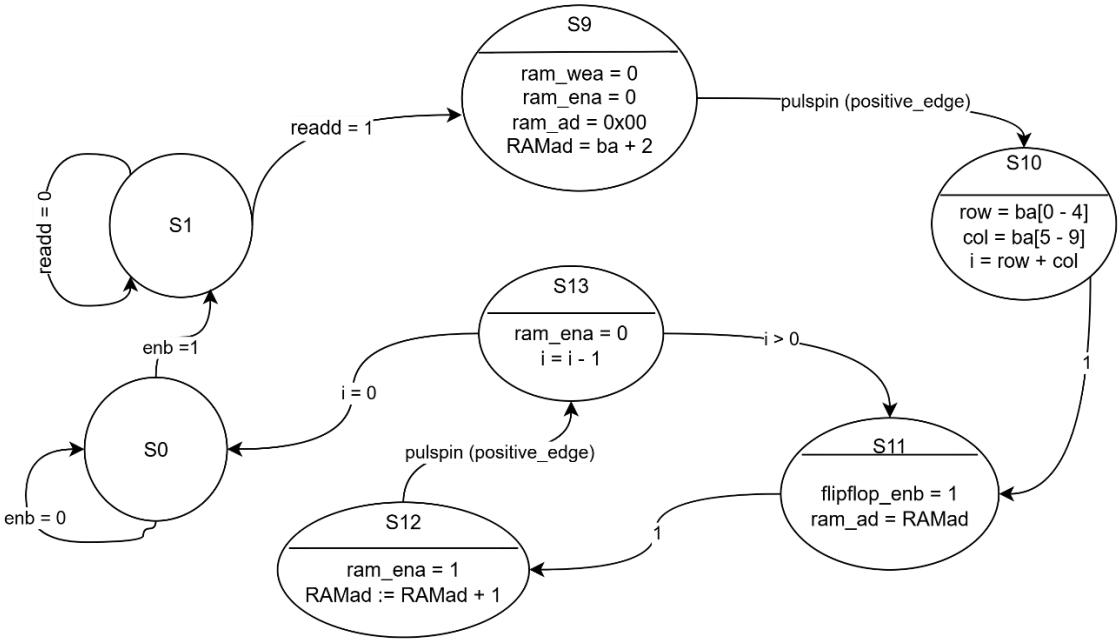
Opcode 1001:

Clear the matrix stored in the memory location pointed in the instruction.



If the savep instruction is high processor goes in to save program mode and user can write instructions to the rom using ba signal vector.

Read data



If the readd signal is high processor goes in to read data mode and outputs 8bit matrix data to flipflop starting from the address provided at the ba signal vector. In this elements outputs one by one when providing a rising edge to pulse signal.

Signals and Their Functions:

Inputs

regdata → Output of the Temp register

irdata → Output of the Instruction Register

ramdata → Output of the RAM

adata → Output of the A Register

ram_done → This signal goes high when RAM finishes it's read or write operations

rom_done → This signal goes high when ROM finishes it's read or write operations

reg_done → This signal goes high when Temp Register finishes its read or writes operations

a_done → This signal goes high when A Register finishes its read or writes operations

ba → This 16bit input signal is used to input RAM/ROM addresses and Instructions

runp → Use to send processor in to Run Instruction mode

savep → Use to send processor in to Save Instruction mode

readd → Use to send processor in to read matrix data mode

pulse → Used to give rising edge pulse when reading data and writing instructions

enb → Enable signal of Control Unit

rst → Reset signal of Control Unit

clk → Clock of Control Unit

Outputs

reg_ad → Temp Register address (4 bits)

reg_wea → Temp register read / write mode selection

reg_ena → Temp Register enable signal

reg_clr → Temp Register clear signal

a_ad → A Register address (11 bits)

a_wea → A Register read / write mode selection

a_ena → A Register enable signal

a_clr → A Register clear signal

rom_din → ROM data in (32 bits)

rom_ad → ROM address (16 bits)

rom_wea → ROM read / write mode selection

rom_ena → ROM enable signal

rom_clr → ROM clear signal

ram_ad → RAM address (16 bits)

ram_wea → RAM read / write mode selection

ram_ena → Ram enable signal

ram_clr → RAM clear signal

opcode → opcode input to the Inference Unit (4 bits)

inf_ena → Inference Unit enable signal

ir_enb → Instruction Register enable signal

ir_op → Instruction Register opcode (3 bits)

divad → Instruction Register RAM address dividing logic signal

ir_sel → Instruction Register Output mux selection signal (1 bits)

mux_en → Instruction Register Output mux enable signal

flipflop_enb → RAM data output flipflop enable signal

err → Error trap state indication signal

Control Unit VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity control_unit is
  Port (
        --Inputs
        regdata      : in STD_LOGIC_VECTOR(7 downto 0);
        irdata       : in STD_LOGIC_VECTOR(7 downto 0);
        ramdata      : in STD_LOGIC_VECTOR(7 downto 0);
        adata        : in STD_LOGIC_VECTOR(7 downto 0);
        ram_done     : in STD_LOGIC;
        rom_done     : in STD_LOGIC;
        reg_done     : in STD_LOGIC;
        a_done       : in STD_LOGIC;
        ba           : in STD_LOGIC_VECTOR(15 downto 0);
        runp         : in STD_LOGIC;
        savep        : in STD_LOGIC;
        readd        : in STD_LOGIC;
        pulse        : in STD_LOGIC;
        enb          : in STD_LOGIC;
        rst          : in STD_LOGIC;
        clk          : in  STD_LOGIC;

        --Outputs
        reg_ad       : out STD_LOGIC_VECTOR(3 downto 0);
        reg_wea      : out STD_LOGIC;
        reg_ena      : out STD_LOGIC;
        reg_clr      : out STD_LOGIC;
        a_ad         : out STD_LOGIC_VECTOR(10 downto 0);
        a_wea        : out STD_LOGIC;
        a_ena        : out STD_LOGIC;
        a_clr        : out STD_LOGIC;
        rom_din      :out STD_LOGIC_VECTOR(31 downto 0);
        rom_ad       : out STD_LOGIC_VECTOR(15 downto 0);
        rom_wea      : out STD_LOGIC;
        rom_ena      : out STD_LOGIC;
        rom_clr      : out STD_LOGIC;
        ram_ad       : out STD_LOGIC_VECTOR(15 downto 0);
        ram_wea      : out STD_LOGIC;
        ram_ena      : out STD_LOGIC;
        ram_clr      : out STD_LOGIC;
        opcode       : out STD_LOGIC_VECTOR(3 downto 0);
        inf_ena      : out STD_LOGIC;
        ir_enb       : out STD_LOGIC;
        ir_op        : out STD_LOGIC_VECTOR(2 downto 0);
        divad        : out STD_LOGIC;
        ir_sel       : out STD_LOGIC;
        mux_en       : out STD_LOGIC;
```

```
        flipflop_enb : out STD_LOGIC;
        prog_done    : out STD_LOGIC;
        err          : out STD_LOGIC
    );
end control_unit;

architecture Behavioral of control_unit is
  type state_type is ( S0, S1, S2, S2_1, S3, S3_1, S4, S4_1, S5, S5_1, S5_2, S6, S6_1,
S6_2, S7, S7_1, S8, S9, S9_1, S10, S10_1, S10_2, S10_3, S11, S12, S12_1, S12_2, S13,
ERROR, DONE,
                      MV0, MV1, MV1_1, MV2, MV3, MV4, MV5, MV6, MV7, MV8,
                      LD0, LD1, LD2, LD3, LD4, LD5, LD5_1, LD6, LD7, LD8, LD9, LD10,
LD11, LD12, LD13, LD13_1, LD14, LD15, LD16, LD16_1, LD17, LD17_1, LD18, LD18_1, LD19,
LD19_1, LD_STR_F,
                      CM0, CM1, CM2, CM3, CM4, CM5, CM6, CM7, CM8, CM9, CM10,
                      CR0, CR1, CR2, CR3, CR4, CR5, CR6, CR7, CR8, CR9, CR10, CR11,
CR12,
                      INF0, INF1, INF1_1, INF1_2, INF2, INF2_1, INF3, INF3_1, INF4,
INF4_1, INF5, INF6, INF7, INF8, INF9, INF9_1, INF10, INF11, INF12,
                      STR0, STR1, STR2, STR3, STR3_1, STR4, STR5, STR5_1, STR6, STR7,
STR7_1, STR8, STR8_1,
                      SET0, SET1, SET2, SET3, SET4, SET5, SET6, SET7, SET8
                     );
  signal state, next_state : state_type;
  -- signal out_reg : STD_LOGIC_VECTOR(7 downto 0);
--  variable i : std_logic_vector(10 downto 0);
--  signal i : STD_LOGIC_VECTOR(9 downto 0);
  --Signals to detect and synchronyze 'pulse' pin rising edge
  signal pulse_sync1, pulse_sync2 : std_logic; -- sync registers
  signal pulse_prev               : std_logic; -- store previous state
  signal pulse_edge               : std_logic; -- one-clock pulse

begin

  process(clk, rst)
  begin
      if rst = '1' then
        state <= S0;
        pulse_sync1 <= '0';
        pulse_sync2 <= '0';
        pulse_prev  <= '0';
      elsif rising_edge(clk) then
        state <= next_state;
        pulse_sync1 <= pulse;       -- first sync
        pulse_sync2 <= pulse_sync1; -- second sync
        pulse_prev  <= pulse_sync2; -- store previous value
      end if;
  end process;

  -- Rising edge detection
  pulse_edge <= '1' when (pulse_sync2 = '1' and pulse_prev = '0') else '0';
```

```
  -- Next state logic
  process(state, enb, ba, irdata, pulse, readd, savep, runp, a_done, reg_done,
rom_done, ram_done, adata, regdata)
    variable row : unsigned(4 downto 0);
    variable col : unsigned(4 downto 0);
    variable RAM_H : unsigned(7 downto 0);
    variable RAM_L : unsigned(7 downto 0);
    variable regad : unsigned(3 downto 0);
    variable basead : unsigned(7 downto 0);
    variable i : unsigned(9 downto 0);
    variable Aad : unsigned(10 downto 0);
    variable RAMad : unsigned(15 downto 0);
    variable prog_H : unsigned(15 downto 0);
    variable prog_L : unsigned(15 downto 0);
    variable pc : unsigned(15 downto 0);
    variable ROMdin : unsigned(31 downto 0);
  begin
    next_state <= state;
    case state is
      when S0 =>
        if enb = '1' then
          next_state <= S1;
        else
          next_state <= S0;
        end if;

      when S1 =>
        if runp = '1' then
          next_state <= S2;
        elsif savep = '1' then
          next_state <= S5;
        elsif readd = '1' then
          next_state <= S9;
        else
          next_state <= S1;
        end if;

      --Opcode Process (Run program)
      when S2 =>
        ir_enb <= '1';
        rom_wea <= '0';
        rom_ena <= '1';
        mux_en <= '0';
        rom_ad <= ba;
        pc := unsigned(ba);
        next_state <= S3;

      when S2_1 =>
        if rom_done = '1' then
          next_state <= S3;
        else
          next_state <= S2_1;
```

```
        end if;

      when S3 =>
        rom_ena <= '0';
        ir_op <= "001";
        next_state <= S3_1;

      when S3_1 =>
        opcode <= irdata(3 downto 0);
        case irdata is
          when "00000000" => next_state <= S3;
          when "00000001" => next_state <= SET0; --Set matrix meta data
          when "00000010" => next_state <= MV0; --Move values to matrix elements
          when "00000011" => next_state <= LD0; --Load matrix in to registers
          when "00000100" => next_state <= STR0; --Store matrix in A in to RAM
          when "00000101" | "00000110" | "00000111" => --Fuzzy addition, substraction,
multiplication
            next_state <= INF0;
          when "00001000" => next_state <= CR0; --Clear registers
          when "00001001" => next_state <= CM0; --Clear RAM
          when "00001111" => next_state <= DONE;
          when others => next_state <= ERROR;
        end case;

      when S4 =>
        if runp = '1' then
          next_state <= S4_1;
        else
          next_state <= S1;
        end if;

      when S4_1 =>
        pc := pc + 1;
        rom_ad <= std_logic_vector(pc);
        rom_ena <= '1';
        next_state <= S2_1;

      --Set metrix meta data
      when SET0 =>
        ir_op <= "010";
        reg_wea <= '1';
        next_state <= SET1;

      when SET1 =>
        basead := unsigned(irdata);
        divad <= '1';
        ir_sel <= '0';
        mux_en <= '1';
        reg_ad <= std_logic_vector(basead (3 downto 0));
        reg_ena <= '1';
        if reg_done = '1' then
          next_state <= SET2;
```

```
      else
        next_state <= SET1;
      end if;

    when SET2 =>
      reg_ena <= '0';
      ir_op <= "011";
      basead := basead + 1;
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= SET3;

    when SET3 =>
      reg_ena <= '1';
      if reg_done = '1' then
        next_state <= SET4;
      else
        next_state <= SET3;
      end if;

    when SET4 =>
      reg_ena <= '0';
      divad <= '0';
      ir_op <= "100";
      basead := basead + 1;
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= SET5;

    when SET5 =>
      reg_ena <= '1';
      if reg_done = '1' then
        next_state <= SET6;
      else
        next_state <= SET5;
      end if;

    when SET6 =>
      reg_ena <= '0';
      ir_op <= "101";
      basead := basead + 1;
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= SET7;

    when SET7 =>
      reg_ena <= '1';
      if reg_done = '1' then
        next_state <= SET8;
      else
        next_state <= SET7;
      end if;

    when SET8 =>
      reg_ena <= '0';
```

```
      reg_wea <= '0';
      mux_en <= '0';
      ir_op <= "000";
      basead := "00000000";
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= S4;

    --Move values to matrix elements
    when MV0 =>
      ir_op <= "010";
      reg_wea <= '0';
      reg_ena <= '1';
      next_state <= MV1;

    when MV1 =>
      reg_ad <= irdata(3 downto 0);
      if reg_done = '1' then
        next_state <= MV1_1;
      else
        next_state <= MV1;
      end if;

    when MV1_1 =>
      reg_ad <= std_logic_vector(unsigned(irdata(3 downto 0)) + 1);
      next_state <= MV2;

    when MV2 =>
      RAM_H := unsigned(regdata);
      if reg_done = '1' then
        next_state <= MV3;
      else
        next_state <= MV2;
      end if;

    when MV3 =>
      RAM_L := unsigned(regdata);
      RAMad := RAM_H & RAM_L;
      reg_ena <= '0';
      ir_op  <= "011";
      next_state <= MV4;

    when MV4 =>
      row := unsigned(irdata(4 downto 0));
      ir_op <= "100";
      next_state <= MV5;

    when MV5 =>
      col := unsigned(irdata(4 downto 0));
      ir_op <= "101";
      next_state <= MV6;

    when MV6 =>
```

```
      ram_ad <= std_logic_vector(RAMad + (row * col));
      ram_wea <= '1';
      ir_op <= "011";
      ir_sel <= '1';
      mux_en <= '1';
      next_state <= MV7;

    when MV7 =>
      ram_ena <= '1';
      if ram_done = '1' then
        next_state <= MV8;
      else
        next_state <= MV7;
      end if;

    when MV8 =>
      ram_ena <= '0';
      ram_ad <= x"0000";
      RAMad := x"0000";
      row := "00000";
      col := "00000";
      RAM_H := x"00";
      RAM_L := x"00";
      ram_wea <= '0';
      ir_op <= "000";
      mux_en <= '0';
      ir_sel <= '0';
      next_state <= S4;

    --Load matrix in to registers
    when LD0 =>
      ir_op <= "010";
      case irdata is
        when "00000000" | "00000100" | "00001000" | "00001100"  =>
          next_state <= LD1;
        when"00010000" =>
          next_state <= LD10;
        when others => next_state <= ERROR;
      end case;
    when LD1 =>
      basead := unsigned(irdata);
      reg_wea <= '1';
      divad <= '1';
      ir_sel <= '0';
      mux_en <= '1';
      reg_ad <= std_logic_vector(basead(3 downto 0));
      next_state <= LD2;

    when LD2 =>
      reg_ena <= '1';
      RAM_H := unsigned(irdata);
      if reg_done = '1' then
```

```
        next_state <= LD3;
      else
        next_state <= LD2;
      end if;

    when LD3 =>
      reg_ena <= '0';
      ir_op <= "011";
      basead := basead + 1;
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= LD4;

    when LD4 =>
      reg_ena <= '1';
      RAM_L := unsigned(irdata);
      RAMad := RAM_H & RAM_L;
      if reg_done = '1' then
        next_state <= LD5;
      else
        next_state <= LD4;
      end if;

    when LD5 =>
      ram_ad <= std_logic_vector(RAMad);
      reg_ena <= '0';
      divad <= '0';
      ram_wea <= '0';
      ram_ena <= '1';
      if ram_done = '1' then
        next_state <= LD5_1;
      else
        next_state <= LD5;
      end if;

    when LD5_1 =>
      basead := basead + 1;
      reg_ad <= std_logic_vector(basead (3 downto 0));
      next_state <= LD6;

    when LD6 =>
      reg_ena <= '1';
      if reg_done = '1' then
        next_state <= LD7;
      else
        next_state <= LD6;
      end if;

    when LD7 =>
      reg_ena <= '1';
      basead := basead + 1;
      ram_ad <= std_logic_vector(RAMad + 1);
      reg_ad <= std_logic_vector(basead (3 downto 0));
```

```
        next_state <= LD8;

      when LD8 =>
        reg_ena <= '1';
        if reg_done = '1' then
          next_state <= LD9;
        else
          next_state <= LD8;
        end if;

      when LD9 =>
        reg_ena <= '0';
        ram_ena <= '0';
        basead := "00000000";
        ram_ad <= "0000000000000000";
        reg_ad <= "0000";
        RAMad := x"0000";
        RAM_H := x"00";
        RAM_L := x"00";
        next_state <= S4;

      when LD10 =>
        Aad := "00000000000";
        a_ad <= std_logic_vector(Aad);
        a_wea  <= '1';
        divad  <= '1';
        ir_sel <= '1';
        mux_en <= '1';
        opcode <= "1011";
        next_state <= LD11;

      when LD11 =>
        a_ena <= '1';
        RAM_H := unsigned(irdata);
        if a_done = '1' then
          next_state <= LD12;
        else
          next_state <= LD11;
        end if;

      when LD12 =>
        a_ena <= '0';
        ir_op <= "011";
        Aad := Aad + 1;
        a_ad <= std_logic_vector(Aad);
        next_state <= LD13;

      when LD13 =>
        a_ena <= '1';
        RAM_L := unsigned(irdata);
        RAMad := RAM_H & RAM_L;
        if a_done = '1' then
```

```
        next_state <= LD13_1;
      else
        next_state <= LD13;
      end if;

    when LD13_1 =>
      ram_ad <= std_logic_vector(RAMad);
      Aad := Aad + 1;
      a_ad <= std_logic_vector(Aad);
      a_ena <= '0';
      divad <= '0';
      ram_wea <= '0';
      mux_en <= '0';
      next_state <= LD14;


    when LD14 =>
      ram_ena <= '1';
      inf_ena <= '1';
      if ram_done = '1' then
        next_state <= LD15;
      else
        next_state <= LD14;
      end if;

    when LD15 =>
      ram_ena <= '0';
      row := unsigned(ramdata(4 downto 0));
      a_ena <= '1';
      if a_done = '1' then
        next_state <= LD16;
      else
        next_state <= LD15;
      end if;

    when LD16 =>
      a_ena <= '0';
      Aad := Aad + 1;
      RAMad := RAMad + 1;
      ram_ad <= std_logic_vector(RAMad);
      a_ad <= std_logic_vector(Aad);
      next_state <= LD16_1;

      when LD16_1  =>
        ram_ena <= '1';
        if ram_done = '1' then
          next_state <= LD17;
        else
          next_state <= LD16_1;
        end if;

    when LD17 =>
```

```
      ram_ena <= '0';
      col := unsigned(ramdata(4 downto 0));
      a_ena <= '1';
      if a_done = '1' then
        next_state <= LD17_1;
      else
        next_state <= LD17;
      end if;

    when LD17_1  =>
      i := row * col;
      next_state <= LD18;

    when LD18  =>
      a_ena <= '0';
      Aad := Aad + 1;
      RAMad := RAMad + 1;
      ram_ad <= std_logic_vector(RAMad);
      a_ad <= std_logic_vector(Aad);
      next_state <= LD18_1;

    when LD18_1  =>
      ram_ena <= '1';
      if ram_done = '1' then
        next_state <= LD19;
      else
        next_state <= LD18_1;
      end if;

    when LD19 =>
      ram_ena <= '0';
      a_ena <= '1';
      if i > "0000000000" then
        if a_done = '1' then
          next_state <= LD19_1;
        else
          next_state <= LD19;
        end if;
      else
        if a_done = '1' then
          next_state <= LD_STR_F;
        else
          next_state <= LD19;
        end if;
      end if;

    when LD19_1 =>
      i := (i - 1);
      next_state <= LD18;

    when LD_STR_F =>
      a_ena <= '0';
```

```
      ram_ena <= '0';
      inf_ena <= '0';
      Aad := "00000000000";
      ram_ad <= x"0000";
      a_ad <= "00000000000";
      RAMad := "0000000000000000";
      RAM_H := "00000000";
      RAM_L := "00000000";
      row := "00000";
      col := "00000";
      opcode <= "0000";
      next_state <= S4;

    --Store matrix in A in to RAM
    when STR0 =>
      divad <= '1';
      ir_op <= "010";
      next_state <= STR1;

    when STR1 =>
      RAM_H := unsigned(irdata);
      ir_op <= "011";
      Aad := "00000000010";
      a_wea <= '0';
      a_ena <= '0';
      ram_wea <= '1';
      ram_ena <= '0';
      opcode <= "1010";
      next_state <= STR2;

    when STR2 =>
      RAM_L := unsigned(irdata);
      divad <= '0';
      ir_enb <= '0';
      ir_op <= "000";
      RAMad := RAM_H & RAM_L;
      inf_ena <= '1';
      a_ad <= std_logic_vector(Aad);
      ram_ad <= std_logic_vector(RAMad);
      next_state <= STR3;

    when STR3 =>
      a_ena <= '1';
      if a_done = '1' then
        next_state <= STR3_1;
      else
        next_state <= STR3;
      end if;

    when STR3_1 =>
      a_ena <= '0';
      row := unsigned(adata(4 downto 0));
```

```
        Aad := Aad + 1;
        RAMad := RAMad + 1;
        a_ad <= std_logic_vector(Aad);
        next_state <= STR4;

      when STR4 =>
        ram_ena <= '1';
        if ram_done = '1' then
          next_state <= STR5;
        else
          next_state <= STR4;
        end if;

      when STR5 =>
        ram_ena <= '0';
        a_ena <= '1';
        ram_ad <= std_logic_vector(RAMad);
        if a_done = '1' then
          next_state <= STR5_1;
        else
          next_state <= STR5;
        end if;

      when STR5_1 =>
        a_ena <= '0';
        col := unsigned(adata(4 downto 0));
        i := row * col;
        next_state <= STR6;

      when STR6 =>
        ram_ena <= '1';
        if ram_done = '1' then
          next_state <= STR7;
        else
          next_state <= STR6;
        end if;

      when STR7 =>
        ram_ena <= '0';
        Aad := Aad + 1;
        RAMad := RAMad + 1;
        a_ad <= std_logic_vector(Aad);
        ram_ad <= std_logic_vector(RAMad);
        next_state <= STR7_1;

      when STR7_1 =>
        a_ena <= '1';
        if a_done = '1' then
          next_state <= STR8;
        else
          next_state <= STR7_1;
        end if;
```

```
when STR8 =>
  a_ena <= '0';
  ram_ena <= '1';
  if ram_done = '1' then
    next_state <= STR8_1;
  else
    next_state <= STR8;
  end if;

when STR8_1 =>
  i := (i - 1);
  if i > "0000000000" then
    next_state <= STR7;
  else
    next_state <= LD_STR_F;
  end if;

--Fuzzy addition, substraction, multiplication
when INF0 =>
  opcode <= irdata(3 downto 0);
  ir_op <= "011";
  reg_wea <= '0';
  reg_ena <= '0';
  a_wea <= '0';
  a_ena <= '0';
  next_state <= INF1;

when INF1 =>
  regad := unsigned(irdata(3 downto 0));
  Aad := "00000000000";
  reg_ad <= std_logic_vector(regad + 2);
  a_ad <= std_logic_vector(Aad + 2);
  next_state <= INF1_1;

when INF1_1 =>
  reg_ena <= '1';
  if reg_done = '1' then
    next_state <= INF1_2;
  else
    next_state <= INF1_1;
  end if;

when INF1_2 =>
  reg_ena <= '0';
  a_ena <= '1';
  if a_done = '1' then
    next_state <= INF2;
  else
    next_state <= INF1_2;
  end if;
```

```
      when INF2 =>
        a_ena <= '0';
        if not(regdata = adata) then
          next_state <= ERROR;
        else
          row := unsigned(regdata(4 downto 0));
          next_state <= INF2_1;
        end if;

      when INF2_1 =>
        reg_ad <= std_logic_vector(regad + 3);
        a_ad <= std_logic_vector(Aad + 3);
        ir_op <= "000";
        next_state <= INF3;

      when INF3 =>
        reg_ena <= '1';
        if reg_done = '1'then
          next_state <= INF3_1;
        else
          next_state <= INF3;
        end if;

      when INF3_1 =>
        reg_ena <= '0';
        a_ena <= '1';
        if a_done = '1' then
          next_state <= INF4;
        else
          next_state <= INF3_1;
        end if;

      when INF4 =>
        a_ena <= '0';
        if not(regdata = adata) then
          next_state <= ERROR;
        else
          col := unsigned(regdata(4 downto 0));
          next_state <= INF4_1;
        end if;

      when INF4_1 =>
        i := row * col;
        reg_ad <= std_logic_vector(regad);
        next_state <= INF5;

      when INF5 =>
        reg_ena <= '1';
        if reg_done = '1' then
          next_state <= INF6;
        else
          next_state <= INF5;
```

```
      end if;

    when INF6 =>
      reg_ena <= '0';
      reg_ad <= std_logic_vector(regad + 1);
      RAM_H := unsigned(regdata);
      next_state <= INF7;

    when INF7 =>
      reg_ena <= '1';
      if reg_done = '1' then
        next_state <= INF8;
        else
        next_state <= INF7;
        end if;

    when INF8 =>
      reg_ena <= '0';
      RAM_L := unsigned(regdata);
      RAMad := RAM_H & RAM_L;
      reg_ad <= "0000";
      inf_ena <= '0';
      Aad := "00000000010";
      RAMad := RAMad + 2;
      ram_wea <= '0';
      next_state <= INF9;

    when INF9 =>
      inf_ena <= '1';
      a_ad <= std_logic_vector(Aad);
      a_ena <= '1';
      if a_done = '1' then
        next_state <= INF9_1;
      else
        next_state <= INF9;
      end if;

    when INF9_1 =>
      a_ena <= '0';
      ram_ad <= std_logic_vector(RAMad);
      ram_ena <= '1';
      if ram_done = '1'then
        next_state <= INF10;
      else
        next_state <= INF9_1;
      end if;

    when INF10 =>
      ram_ena <= '0';
      inf_ena <= '0';
      a_wea <= '1';
      next_state <= INF11;
```

```
when INF11 =>
  a_ena <= '1';
  if a_done = '1' then
    next_state <= INF12;
  else
    next_state <= INF11;
  end if;

when INF12 =>
  a_ena <= '0';
  a_wea <= '0';
  i := i - 1;
  Aad := Aad + 1;
  RAMad := RAMad + 1;
  if i > "0000000000" then
    next_state <= INF9;
  else
    next_state <= LD_STR_F;
  end if;

--Clear registers
when CR0 =>
  ir_op <= "010";
  reg_ena <= '0';
  a_ena <= '0';
  case irdata is
    when "00000000" | "00000100" | "00001000" | "00001100"  =>
      next_state <= CR1;
    when"00010000" =>
      next_state <= CR4;
    when others => next_state <= ERROR;
  end case;

when CR1 =>
  regad := unsigned(irdata(3 downto 0));
  reg_ad <= std_logic_vector(regad);
  i := "0000000100";
  reg_clr <= '1';
  next_state <= CR2;

when CR2 =>
  reg_ena <= '1';
  if reg_done = '1' then
    next_state <= CR3;
  else
    next_state <= CR2;
  end if;

when CR3 =>
  reg_ena <= '0';
  i := i - 1;
```

```
        regad := regad + 1;
        reg_ad <= std_logic_vector(regad(3 downto 0) + 1);
        if i < "0000000000" then
          next_state <= CR2;
        else
          next_state <= CR11;
        end if;

      when CR4 =>
        i := "0000000000";
        Aad := "00000000000";
        a_ad <= std_logic_vector(Aad + 2);
        next_state <= CR5;

      when CR5 =>
        a_ena <= '1';
        if a_done = '1' then
          next_state <= CR6;
        else
          next_state <= CR5;
        end if;

      when CR6 =>
        a_ena <= '0';
        row := unsigned(adata(4 downto 0));
        a_ad <= std_logic_vector(Aad + 3);
        next_state <= CR7;

      when CR7 =>
        a_ena <= '1';
        if a_done = '1' then
          next_state <= CR8;
        else
          next_state <= CR7;
        end if;

      when CR8 =>
        a_ena <= '0';
        a_clr <= '1';
        col := unsigned(adata(4 downto 0));
        i := (row * col) + 4;
        a_ad <= std_logic_vector(Aad);
        next_state <= CR9;

      when CR9 =>
        a_ena <= '1';
        if a_done = '1' then
          next_state <= CR10;
        else
          next_state <= CR9;
        end if;
```

```
      when CR10 =>
        a_ena <= '0';
        i := i - 1;
        Aad := Aad + 1;
        a_ad <= std_logic_vector(Aad);
        if i > "0000000000" then
          next_state <= CR10;
        else
          next_state <= CR11;
        end if;

      when CR11 =>
        a_ena <= '0';
        reg_ena <= '0';
        a_clr <= '0';
        reg_clr <= '0';
        Aad := "00000000000";
        reg_ad <= "0000";
        a_ad <= "00000000000";
        row := "00000";
        col := "00000";
        next_state <= S4;

      --Clear RAM
      when CM0 =>
        divad <= '1';
        ir_op <= "010";
        next_state <= CM1;

      when CM1 =>
        RAM_H := unsigned(irdata);
        ir_op <= "011";
        next_state <= CM2;

      when CM2 =>
        RAM_L := unsigned(irdata);
        RAMad := RAM_H & RAM_L;
        next_state <= CM3;

      when CM3 =>
        divad <= '0';
        ram_clr <= '1';
        ram_ad <= std_logic_vector(RAMad);
        next_state <= CM4;

      when CM4 =>
        ram_ena <= '1';
        if ram_done = '1' then
          next_state <= CM5;
        else
          next_state <= CM4;
        end if;
```

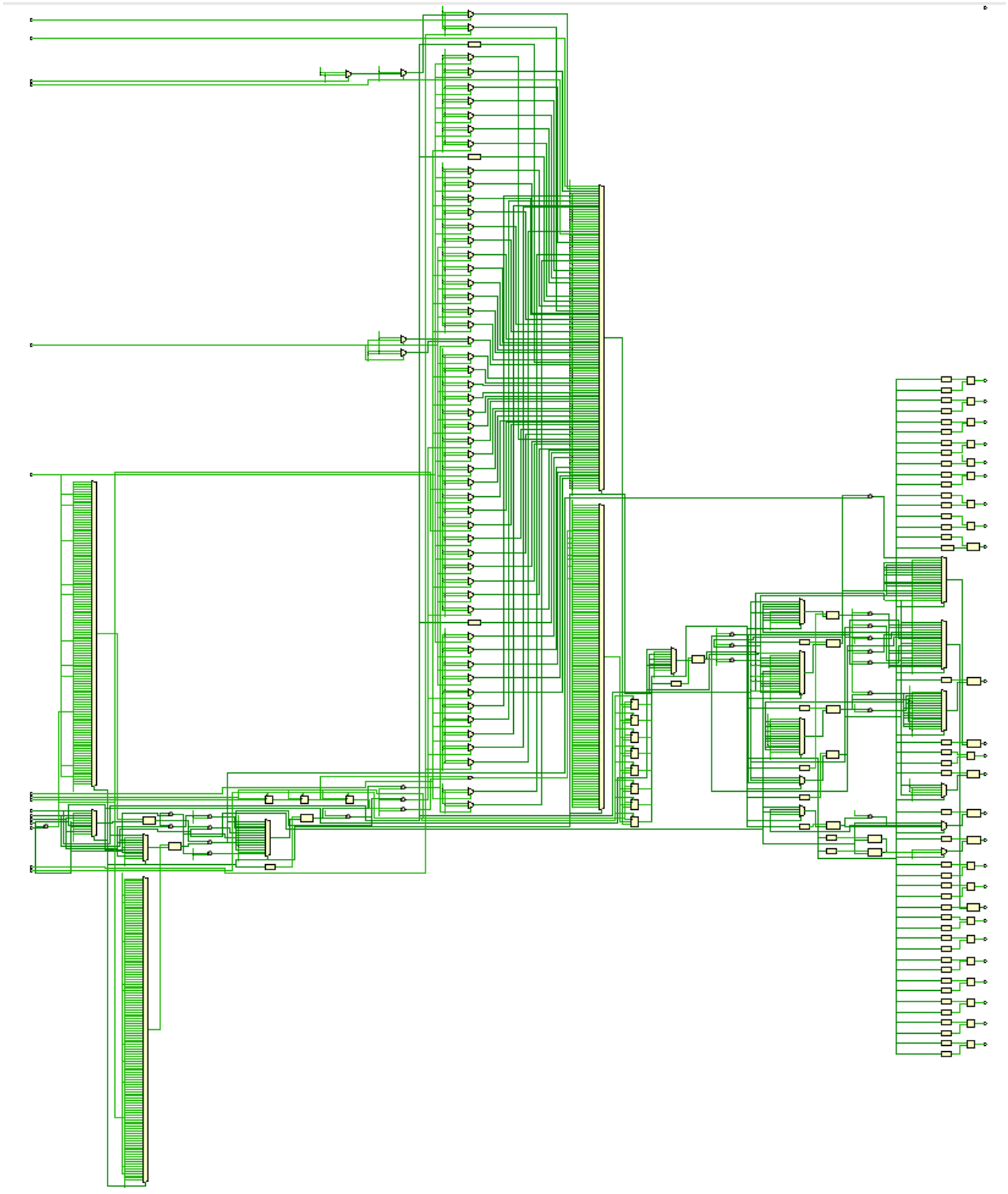
```
    when CM5 =>
      ram_ena <= '0';
      row := unsigned(ramdata(4 downto 0));
      ram_ad <= std_logic_vector(RAMad + 1);
      next_state <= CM6;

    when CM6 =>
      ram_ena <= '1';
      if ram_done = '1' then
        next_state <= CM7;
      else
        next_state <= CM6;
      end if;

    when CM7 =>
      ram_ena <= '0';
      col := unsigned(ramdata(4 downto 0));
      i := (row * col) + 2;
      next_state <= CM8;

    when CM8 =>
      ram_ad <= std_logic_vector(RAMad);
      ram_ena <= '1';
      if ram_done = '1' then
        next_state <= CM9;
      else
        next_state <= CM8;
      end if;

    when CM9 =>
      ram_ena <= '0';
      RAMad := RAMad + 1;
      i := i - 1;
      if i > "0000000000" then
        next_state <= CM8;
      else
        next_state <= CM10;
      end if;

    when CM10 =>
      ram_ena <= '0';
      ram_clr <= '0';
      ram_ad <= x"0000";
      RAMad := "0000000000000000";
      RAM_H := "00000000";
      RAM_L := "00000000";
      row := "00000";
      col := "00000";
      next_state <= S4;

    --Save program process
```

```
      when S5 =>
        rom_wea <= '1';
        rom_ena <= '0';
        rom_din <= x"00000000";
        if pulse_edge = '1' then
          next_state <= S5_1;
        else
          next_state <= S5;
        end if;

      when S5_1 =>
        pc := unsigned(ba);
        next_state <= S5_2;

      when S5_2 =>
        if pulse_edge = '1' then
          next_state <= S6;
        else
          next_state <= S5_2;
        end if;

      when S6 =>
        prog_H := unsigned(ba);
        next_state <= S6_1;

      when S6_1 =>
        if pulse_edge = '1' then
          next_state <= S6_2;
        else
          next_state <= S6_1;
        end if;

      when S6_2 =>
        prog_L := unsigned(ba);
        next_state <= S7;

      when S7 =>
        rom_din <= std_logic_vector(prog_H & prog_L);
        rom_ad <= std_logic_vector(pc);
        if pulse_edge = '1' then
          next_state <= S7_1;
        else
          next_state <= S7;
        end if;

      when S7_1 =>
        rom_ena <= '1';
        if rom_done = '1' then
          next_state <= S8;
        else
          next_state <= S7_1;
        end if;
```

```
when S8 =>
  rom_ena <= '0';
  pc := pc + 1;
  if savep = '1' then
    next_state <= S5;
  else
    next_state <= S0;
  end if;

--Read RAM process
when S9 =>
  ram_wea <= '0';
  ram_ena <= '0';
  ram_ad <= "0000000000000000";
  if pulse_edge = '1' then
    next_state <= S9_1;
  else
    next_state <= S9;
  end if;

when S9_1 =>
  RAMad := unsigned(ba);
  next_state <= S10;

when S10 =>
  if pulse_edge = '1' then
    next_state <= S10_1;
  else
    next_state <= S10;
  end if;

when S10_1 =>
  row := unsigned(ba(4 downto 0));
  next_state <= S10_2;

when S10_2 =>
  if pulse_edge = '1' then
    next_state <= S10_3;
  else
    next_state <= S10_2;
  end if;

when S10_3 =>
  col := unsigned(ba(4 downto 0));
  i := row * col;
  next_state <= S11;

when S11 =>
  flipflop_enb <= '1';
  ram_ad <= std_logic_vector(RAMad);
  next_state <= S12;
```

```
      when S12 =>
        ram_ena <= '1';
        if ram_done = '1' then
          next_state <= S12_1;
        else
          next_state <= S12;
        end if;

      when S12_1 =>
        ram_ena <= '0';
        RAMad := RAMad + 1;
        i := i -1;
        next_state <= S12_2;

      when S12_2 =>
        if pulse_edge = '1' then
          next_state <= S13;
        end if;

      when S13 =>
        flipflop_enb <= '0';
        if i > "0000000000" then
          next_state <= S11;
        else
          next_state <= S0;
        end if;

      when DONE =>
        prog_done <= '1';
        if pulse_edge = '1' then
          next_state <= S0;
        end if;

      when ERROR =>
        err <= '1';
        next_state <= ERROR;

      when others =>
          next_state <= S0;

      end case;
  end process;

end Behavioral;
```

Control Unit RTL Schematic



Control Unit Test Bench Code

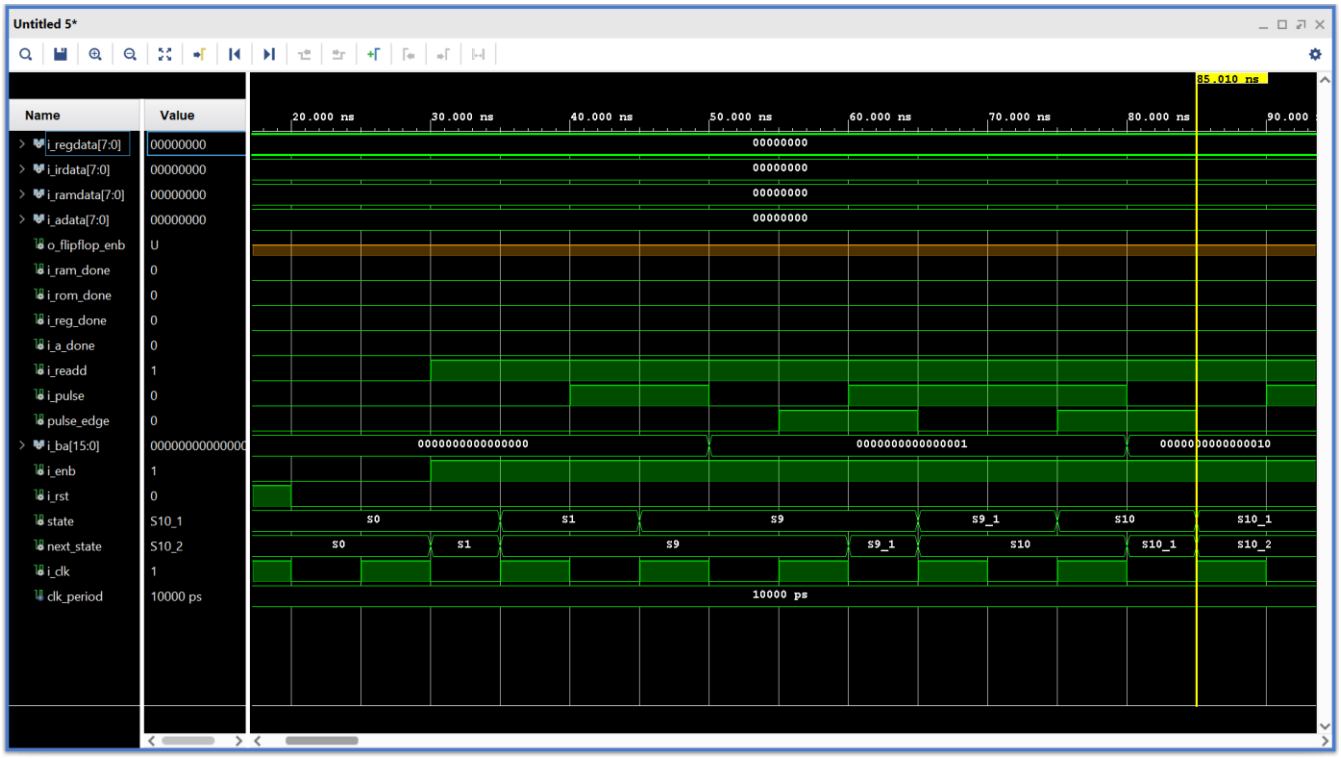
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity tb_control_unit is
end tb_control_unit;

architecture Behavioral of tb_control_unit is
    -- Component Declaration for the Unit Under Test (UUT)
    component control_unit
        Port (
            --Inputs
            regdata      : in STD_LOGIC_VECTOR(7 downto 0);
            irdata       : in STD_LOGIC_VECTOR(7 downto 0);
            ramdata      : in STD_LOGIC_VECTOR(7 downto 0);
            adata        : in STD_LOGIC_VECTOR(7 downto 0);
            ram_done     : in STD_LOGIC;
            rom_done     : in STD_LOGIC;
            reg_done     : in STD_LOGIC;
            a_done       : in STD_LOGIC;
            ba           : in STD_LOGIC_VECTOR(15 downto 0);
            runp         : in STD_LOGIC;
            savep        : in STD_LOGIC;
            readd        : in STD_LOGIC;
            pulse        : in STD_LOGIC;
            enb          : in STD_LOGIC;
            rst          : in STD_LOGIC;
            clk          : in STD_LOGIC;
            --Outputs
            reg_ad       : out STD_LOGIC_VECTOR(3 downto 0);
            reg_wea      : out STD_LOGIC;
            reg_ena      : out STD_LOGIC;
            reg_clr      : out STD_LOGIC;
            a_ad         : out STD_LOGIC_VECTOR(10 downto 0);
            a_wea        : out STD_LOGIC;
            a_ena        : out STD_LOGIC;
            a_clr        : out STD_LOGIC;
            rom_din      : out STD_LOGIC_VECTOR(31 downto 0);
            rom_ad       : out STD_LOGIC_VECTOR(15 downto 0);
            rom_wea      : out STD_LOGIC;
            rom_ena      : out STD_LOGIC;
            rom_clr      : out STD_LOGIC;
            ram_ad       : out STD_LOGIC_VECTOR(15 downto 0);
            ram_wea      : out STD_LOGIC;
            ram_ena      : out STD_LOGIC;
            ram_clr      : out STD_LOGIC;
            opcode       : out STD_LOGIC_VECTOR(3 downto 0);
            inf_ena      : out STD_LOGIC;
            ir_enb       : out STD_LOGIC;
            ir_op        : out STD_LOGIC_VECTOR(2 downto 0);
```

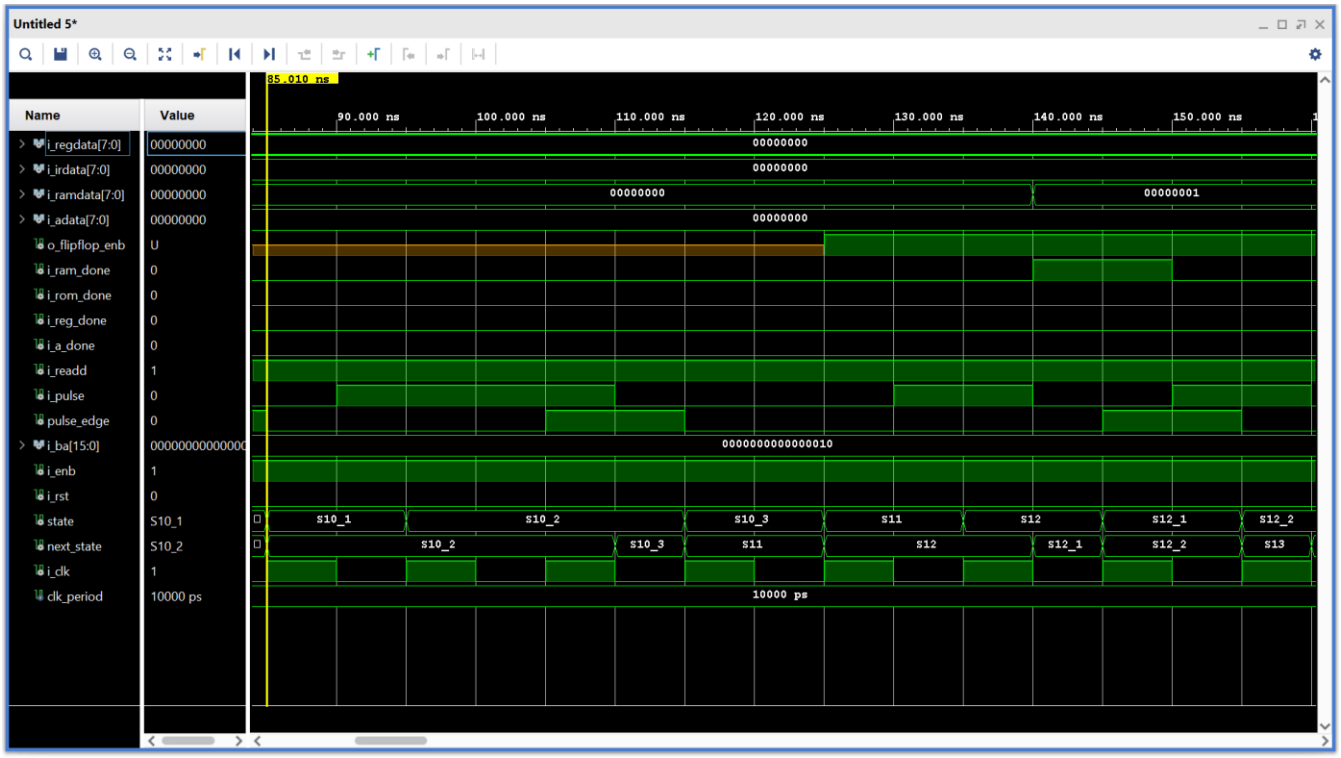
```
            divad        : out STD_LOGIC;
            ir_sel       : out STD_LOGIC;
            mux_en       : out STD_LOGIC;
            flipflop_enb : out STD_LOGIC;
            prog_done    : out STD_LOGIC;
            err          : out STD_LOGIC
        );
    end component;
    -- UUT Inputs
    signal i_regdata      : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal i_irdata       : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal i_ramdata      : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal i_adata        : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal i_ram_done     : STD_LOGIC := '0';
    signal i_rom_done     : STD_LOGIC := '0';
    signal i_reg_done     : STD_LOGIC := '0';
    signal i_a_done       : STD_LOGIC := '0';
    signal i_ba           : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
    signal i_runp         : STD_LOGIC := '0';
    signal i_savep        : STD_LOGIC := '0';
    signal i_readd        : STD_LOGIC := '0';
    signal i_pulse        : STD_LOGIC := '0';
    signal i_enb          : STD_LOGIC := '0';
    signal i_rst          : STD_LOGIC := '1';
    signal i_clk          : STD_LOGIC := '0';
    -- UUT Outputs (can be monitored)
    signal o_rom_din      : STD_LOGIC_VECTOR(31 downto 0);
    signal o_rom_ad       : STD_LOGIC_VECTOR(15 downto 0);
    signal o_rom_wea      : STD_LOGIC;
    signal o_rom_ena      : STD_LOGIC;
    signal o_ram_ena      : STD_LOGIC;
    signal o_reg_ad       : STD_LOGIC_VECTOR(3 downto 0);
    signal o_reg_wea      : STD_LOGIC;
    signal o_reg_ena      : STD_LOGIC;
    signal o_prog_done    : STD_LOGIC;
    signal o_err          : STD_LOGIC;
    signal o_flipflop_enb : std_logic;
    constant clk_period : time := 10 ns;
begin

    -- Instantiate the Unit Under Test (UUT)
    uut: control_unit port map (
        regdata => i_regdata, irdata => i_irdata, ramdata => i_ramdata, adata => i_adata,
        ram_done => i_ram_done, rom_done => i_rom_done, reg_done => i_reg_done, a_done =>
i_a_done,
        ba => i_ba, runp => i_runp, savep => i_savep, readd => i_readd,
        pulse => i_pulse, enb => i_enb, rst => i_rst, clk => i_clk,
        -- Connect outputs
        rom_din => o_rom_din, rom_ad => o_rom_ad, rom_wea => o_rom_wea, rom_ena =>
o_rom_ena, ram_ena => o_ram_ena,
        prog_done => o_prog_done, err => o_err, flipflop_enb => o_flipflop_enb
    );
```

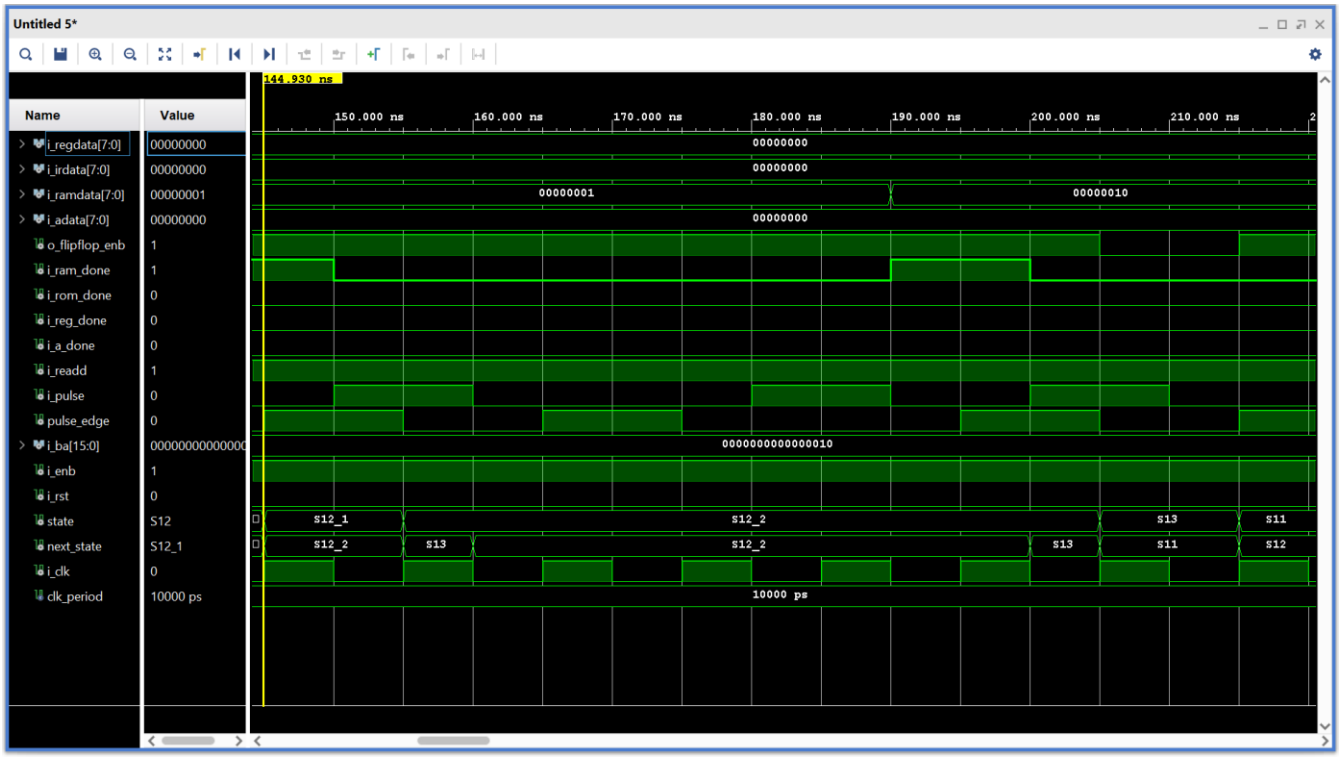
```
    -- Clock process
    clk_process: process
    begin
        i_clk <= '0'; wait for clk_period/2;
        i_clk <= '1'; wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- 1. Reset
        i_rst <= '1';
        wait for 20 ns;
        i_rst <= '0';
        wait for clk_period;
        -- 2. Enable the unit and start save mode
        i_enb <= '1';
        i_readd <= '1';
        wait for 10 ns;
        -- 3. Provide read parameters
        -- State S9 waits for pulse
        i_pulse <= '1'; wait for 10 ns; i_pulse <= '0'; -- Enter S9_1
        -- S9_1: provide RAM base address
        i_ba <= x"0001";
        wait for 10 ns;
        i_pulse <= '1'; wait for 10 ns; i_pulse <= '0';
        -- S10_3: provide col count
        i_pulse <= '1'; wait for 10 ns; i_pulse <= '0';
        i_ba <= x"0002"; -- 2 cols
        wait for 10 ns;
        i_pulse <= '1'; wait for 20 ns; i_pulse <= '0';

        -- 4. Simulate the read loop (2x2 = 4 reads)
        for i in 1 to 4 loop
            wait for 20 ns;
            i_pulse <= '1'; wait for 10 ns; i_pulse <= '0';
            report "Reading data item " & integer'image(i);
            i_ramdata <= std_logic_vector(to_unsigned(i, 8)); -- Provide some dummy data
            i_ram_done <= '1'; wait for 10 ns; i_ram_done <= '0';
            -- After the read, the CU will wait for the next pulse in S12_2
            i_pulse <= '1'; wait for 10 ns; i_pulse <= '0';
        end loop;
        -- 5. End read process
        i_readd <= '0';
        wait;
    end process;
end Behavioral;
```

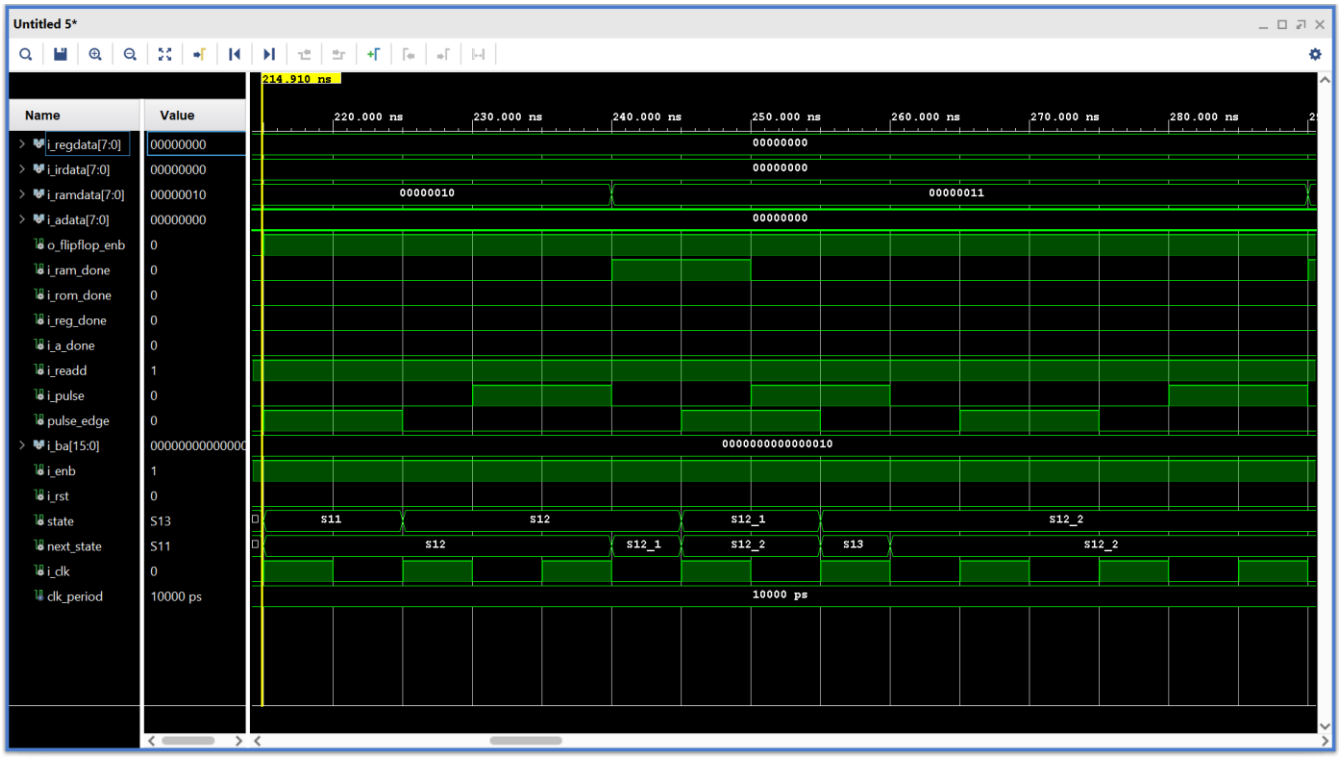
Control Unit Timing Diagram

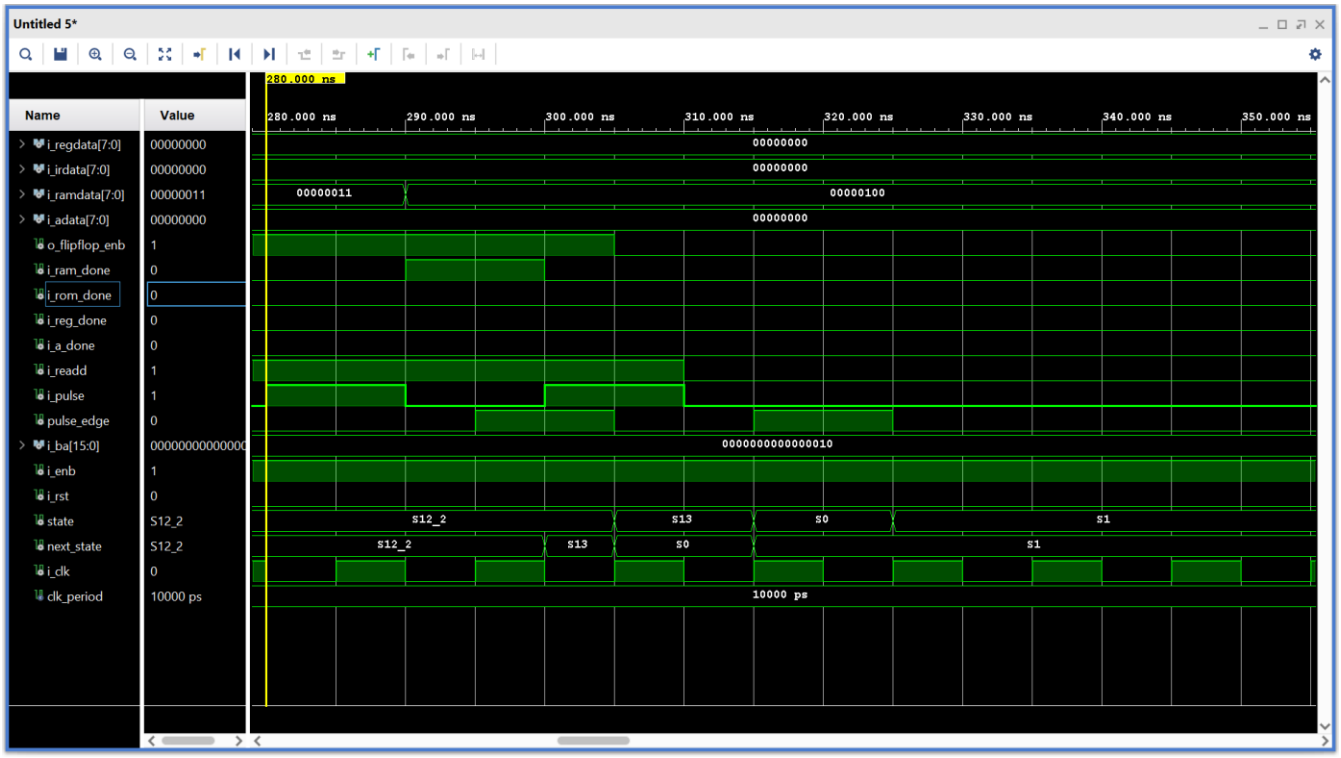
Read Data From RAM



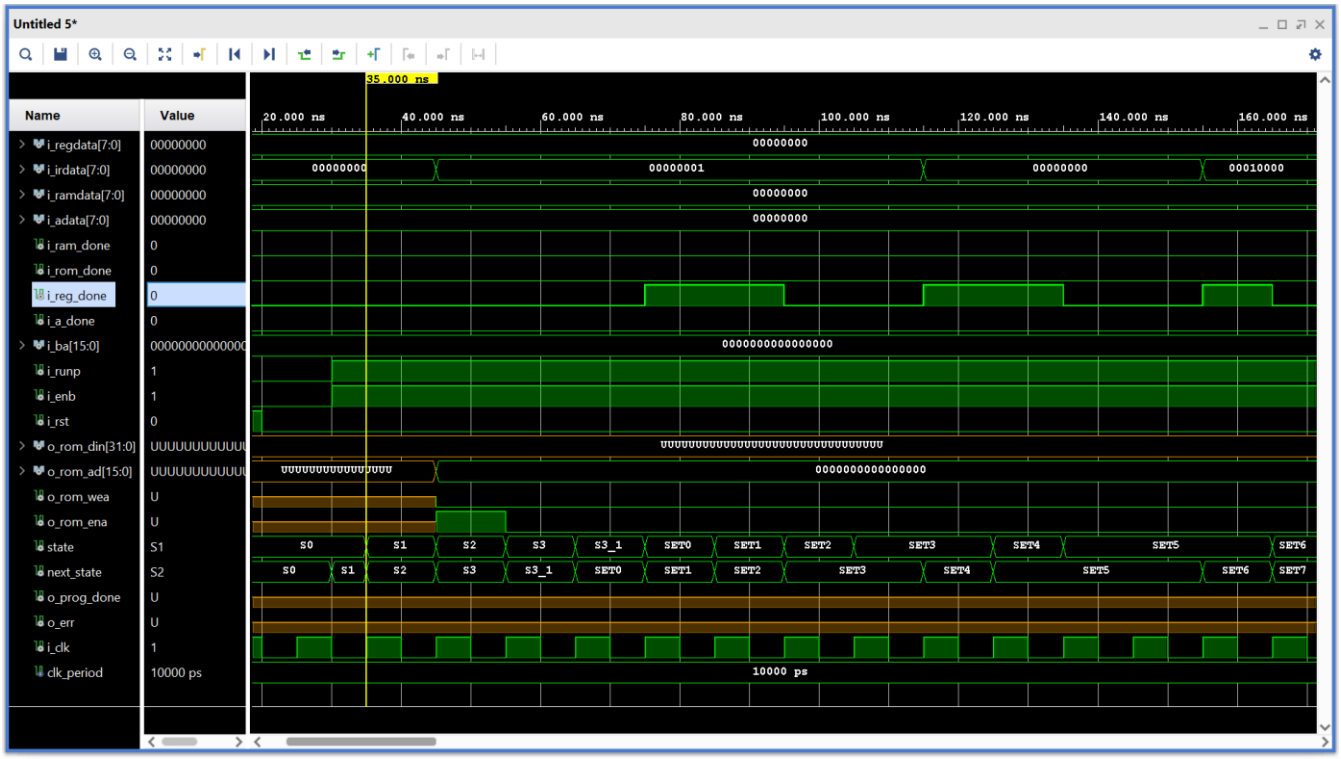


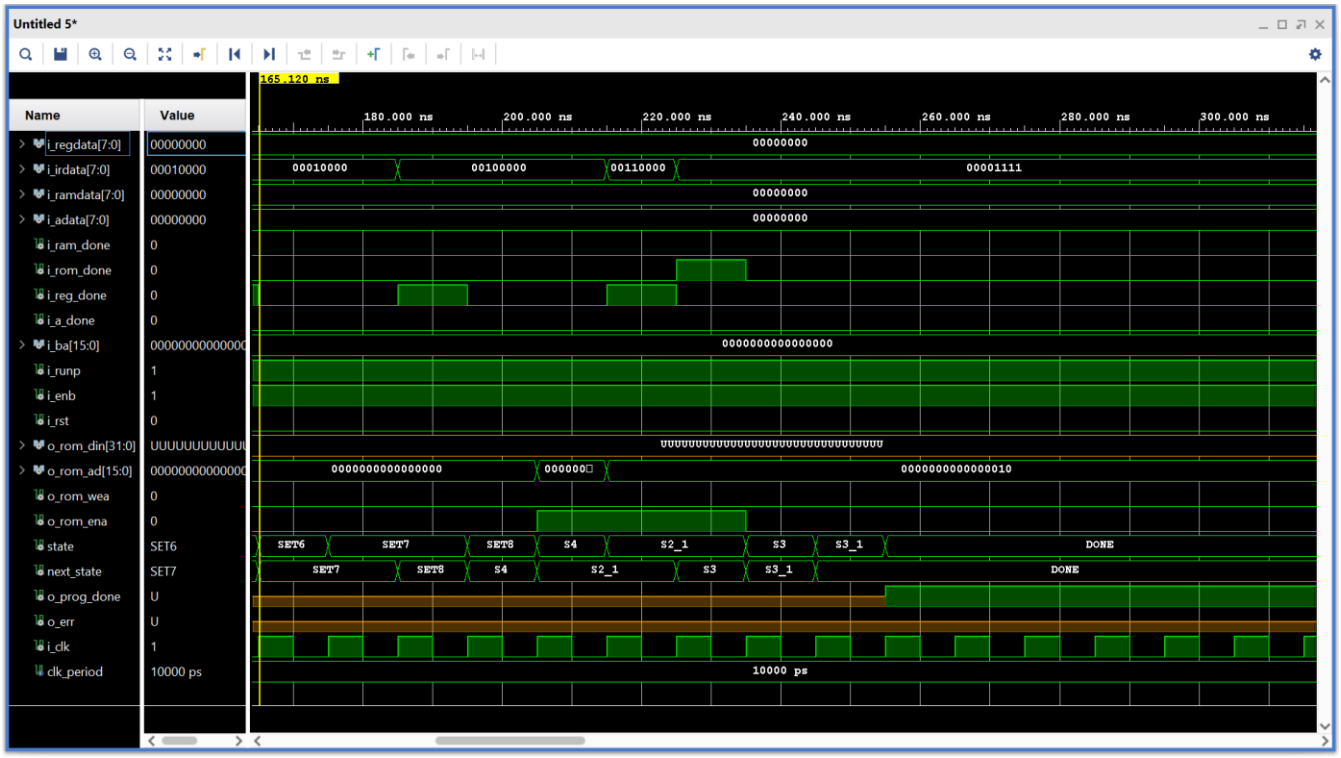

Untitled 5*
144.930 ns
Name
Value
i_regdata[7:0]
00000000
i_irdata[7:0]
00000000
i_ramdata[7:0]
00000001
i_adata[7:0]
00000000
o_flipflop_enb
1
i_ram_done
1
i_rom_done
0
i_reg_done
0
i_a_done
0
i_readd
1
i_pulse
0
pulse_edge
0
i_ba[15:0]
0000000000000
i_enb
1
i_rst
0
state
S12
next_state
S12_1
i_clk
0
clk_period
10000 ps
150.000 ns
160.000 ns
170.000 ns
180.000 ns
190.000 ns
200.000 ns
210.000 ns
00000010
0000000000000010
S12_1
S12_2
S13
S11
S12


Untitled 5*
214.910 ns
Name
Value
i_regdata[7:0]
00000000
i_irdata[7:0]
00000000
i_ramdata[7:0]
00000010
i_adata[7:0]
00000000
o_flipflop_enb
0
i_ram_done
0
i_rom_done
0
i_reg_done
0
i_a_done
0
i_readd
1
i_pulse
0
pulse_edge
0
i_ba[15:0]
0000000000000
i_enb
1
i_rst
0
state
S13
next_state
S11
i_clk
0
clk_period
10000 ps
220.000 ns
230.000 ns
240.000 ns
250.000 ns
260.000 ns
270.000 ns
280.000 ns
00000011
0000000000000010
S11
S12
S12_1
S12_2
S13

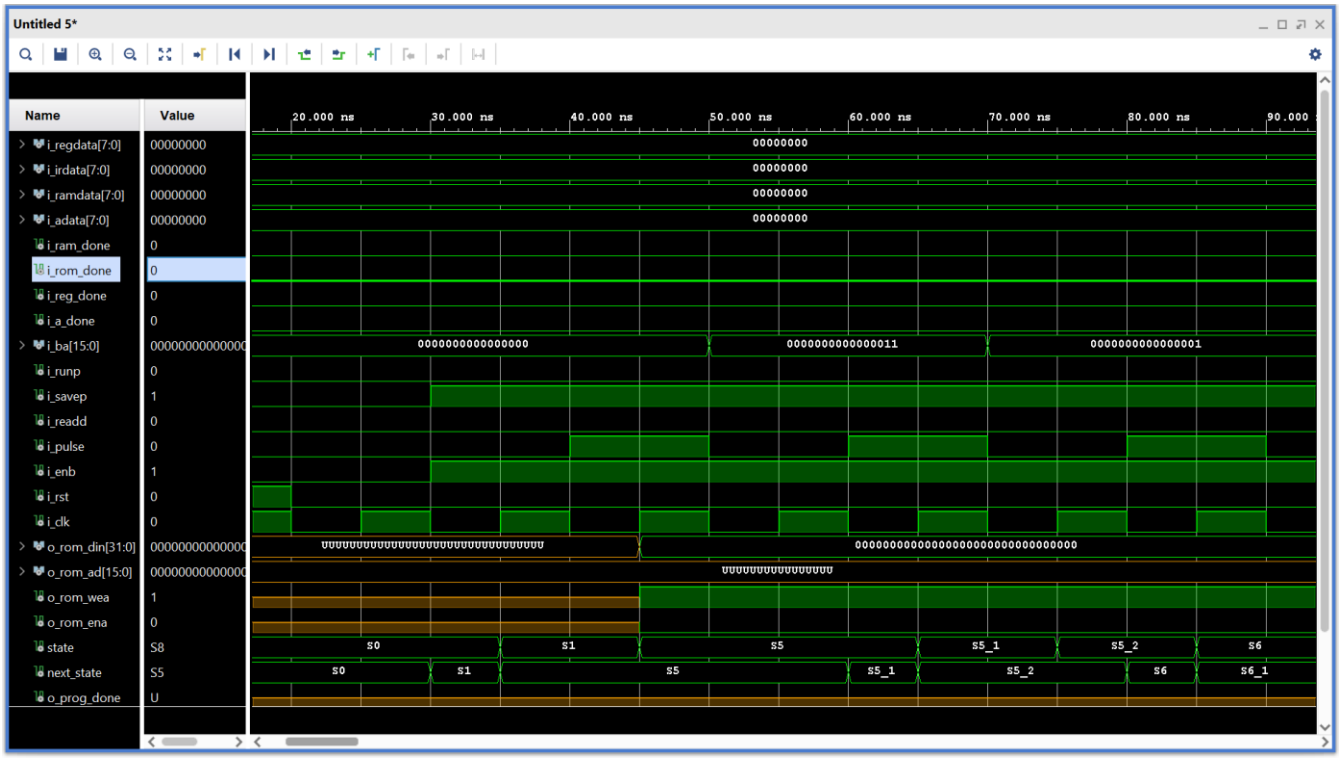

Untitled 5*
Name
Value
i_regdata[7:0]
00000000
i_irdata[7:0]
00000000
i_ramdata[7:0]
00000011
i_adata[7:0]
00000000
o_flipflop_enb
1
i_ram_done
0
i_rom_done
0
i_reg_done
0
i_a_done
0
i_readd
1
i_pulse
1
pulse_edge
0
i_ba[15:0]
i_enb
1
i_rst
0
state
S12_2
next_state
S12_2
i_clk
0
clk_period
10000 ps
280.000 ns
290.000 ns
300.000 ns
310.000 ns
320.000 ns
330.000 ns
340.000 ns
350.000 ns
00000100
0000000000000010
S13
S0
S1
10000 ps

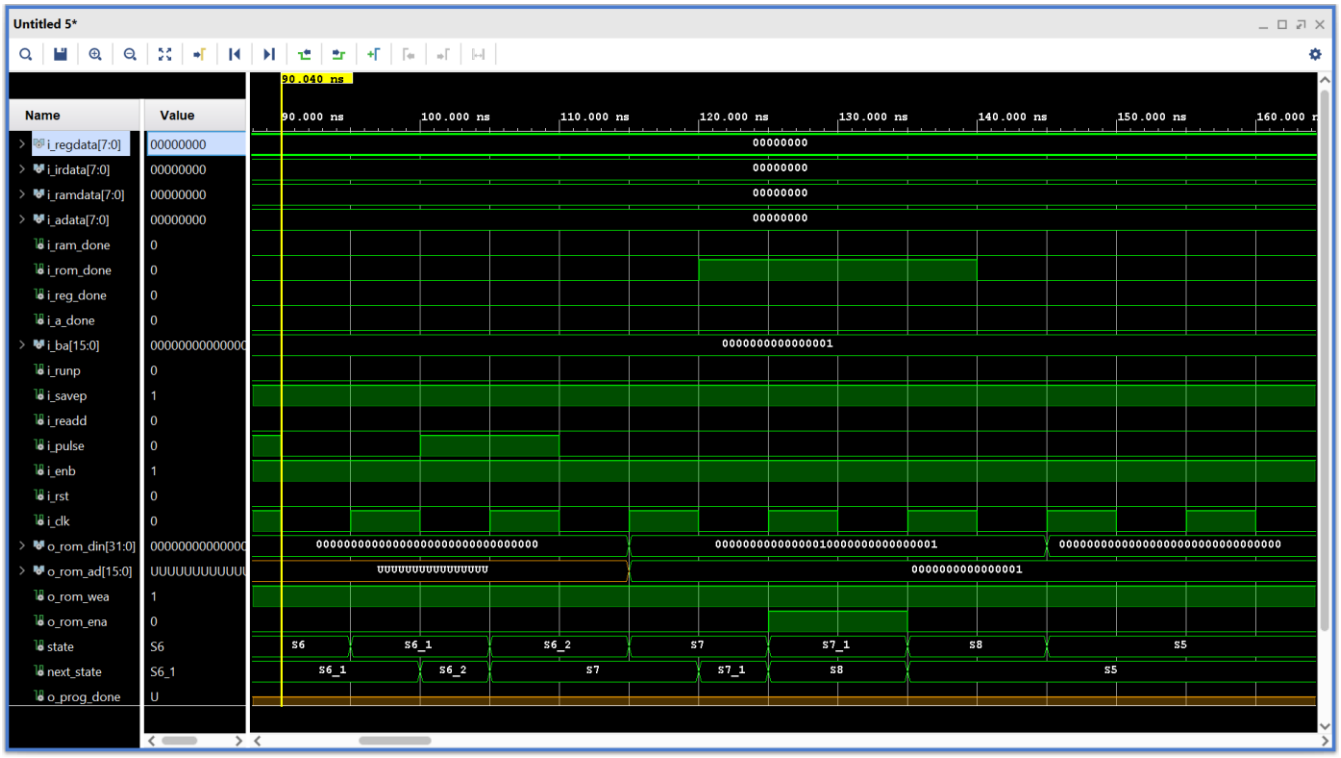
Run Program


Untitled 5*
Name
Value
i_regdata[7:0]
00000000
i_irdata[7:0]
00000000
i_ramdata[7:0]
00000000
i_adata[7:0]
00000000
i_ram_done
0
i_rom_done
0
i_reg_done
0
i_a_done
0
i_ba[15:0]
i_runp
1
i_enb
1
i_rst
0
o_rom_din[31:0]
o_rom_ad[15:0]
o_rom_wea
U
o_rom_ena
U
state
S1
next_state
S2
o_prog_done
U
o_err
U
i_clk
1
clk_period
10000 ps
35.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
00000001
00010000
0000000000000000
S0
S1
S2
S3
S3_1
SET0
SET1
SET2
SET3
SET4
SET5
SET6
SET7
10000 ps



Save Instruction To ROM




Untitled 5*
Name
Value
i_regdata[7:0]
00000000
i_irdata[7:0]
00000000
i_ramdata[7:0]
00000000
i_adata[7:0]
00000000
i_ram_done
0
i_rom_done
0
i_reg_done
0
i_a_done
0
i_ba[15:0]
i_runp
0
i_savep
1
i_readd
0
i_pulse
0
i_enb
1
i_rst
0
i_clk
0
o_rom_din[31:0]
o_rom_ad[15:0]
UUUUUUUUUUUU
o_rom_wea
1
o_rom_ena
0
state
S6
next_state
S6_1
o_prog_done
U
90.040 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
140.000 ns
150.000 ns
160.000 n
00000000
00000000
00000000
00000000
0000000000000001
00000000000000000000000000000000
00000000000000010000000000000001
00000000000000000000000000000000
UUUUUUUUUUUUUUUU
0000000000000001
S6
S6_1
S6_2
S7
S7_1
S8
S5
S6_1
S6_2
S7
S7_1
S8
S5

Top Model Structural Design of FMU

Signals and Their Functions:

Inputs

ext_data → This 16bit input signal is used to input RAM/ROM addresses and Instructions

ext_runp → Use to send processor in to Run Instruction mode

ext_savep → Use to send processor in to Save Instruction mode

ext_readd → Use to send processor in to read matrix data mode

ext_pulse → Used to give rising edge pulse when reading data and writing instructions

ext_enb → Enable signal of FMU

ext_rst → Reset signal of FMU

ext_clk → Clock of FMU

Outputs

ext_do → This 8bit output signal is used to output data

ext_err → Error trap state indication signal

ext_prog_done → Program execution completion signal

FMU VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FMU is
    Port (
            ext_data         : in STD_LOGIC_VECTOR(15 downto 0);
            ext_runp         : in STD_LOGIC;
            ext_savep        : in STD_LOGIC;
            ext_readd        : in STD_LOGIC;
            ext_pulse        : in STD_LOGIC;
            ext_enb          : in STD_LOGIC;
            ext_rst          : in STD_LOGIC;
            ext_clk          : in  STD_LOGIC;
            ext_do           : out STD_LOGIC_VECTOR(7 downto 0);
            ext_prog_done    : out STD_LOGIC;
            ext_err          : out STD_LOGIC
         );
end FMU;

architecture arch_FMU of FMU is
    component control_unit
        Port (
                --Inputs
                regdata      : in STD_LOGIC_VECTOR(7 downto 0);
                irdata       : in STD_LOGIC_VECTOR(7 downto 0);
                ramdata      : in STD_LOGIC_VECTOR(7 downto 0);
                adata        : in STD_LOGIC_VECTOR(7 downto 0);
                ram_done     : in STD_LOGIC;
                rom_done     : in STD_LOGIC;
                reg_done     : in STD_LOGIC;
                a_done       : in STD_LOGIC;
                ba           : in STD_LOGIC_VECTOR(15 downto 0);
                runp         : in STD_LOGIC;
                savep        : in STD_LOGIC;
                readd        : in STD_LOGIC;
                pulse        : in STD_LOGIC;
                enb          : in STD_LOGIC;
                rst          : in STD_LOGIC;
                clk          : in  STD_LOGIC;

                --Outputs
                reg_ad       : out STD_LOGIC_VECTOR(3 downto 0);
                reg_wea      : out STD_LOGIC;
                reg_ena      : out STD_LOGIC;
                reg_clr      : out STD_LOGIC;
                a_ad         : out STD_LOGIC_VECTOR(10 downto 0);
                a_wea        : out STD_LOGIC;
                a_ena        : out STD_LOGIC;
                a_clr        : out STD_LOGIC;
                rom_din      :out STD_LOGIC_VECTOR(31 downto 0);
```

```
            rom_ad       : out STD_LOGIC_VECTOR(15 downto 0);
            rom_wea      : out STD_LOGIC;
            rom_ena      : out STD_LOGIC;
            rom_clr      : out STD_LOGIC;
            ram_ad       : out STD_LOGIC_VECTOR(15 downto 0);
            ram_wea      : out STD_LOGIC;
            ram_ena      : out STD_LOGIC;
            ram_clr      : out STD_LOGIC;
            opcode       : out STD_LOGIC_VECTOR(3 downto 0);
            inf_ena      : out STD_LOGIC;
            ir_enb       : out STD_LOGIC;
            ir_op        : out STD_LOGIC_VECTOR(2 downto 0);
            divad        : out STD_LOGIC;
            ir_sel       : out STD_LOGIC;
            mux_en       : out STD_LOGIC;
            flipflop_enb : out STD_LOGIC;
            prog_done    : out STD_LOGIC;
            err          : out STD_LOGIC
        );
end component;

component rom_top
    Port (
            din : in  STD_LOGIC_VECTOR(31 downto 0);
            addr : in  STD_LOGIC_VECTOR(15 downto 0);
            nrst : in std_logic;
            enb : in std_logic;
            op : in std_logic;
            clr : in std_logic;
            done : out std_logic;
            clk_100MHz : in std_logic;
            dout : out STD_LOGIC_VECTOR(31 downto 0)
        );
end component;

component ram_top
    Port (
            din : in  STD_LOGIC_VECTOR(7 downto 0);
            addr : in  STD_LOGIC_VECTOR(15 downto 0);
            nrst : in std_logic;
            enb : in std_logic;
            op : in std_logic;
            clr : in std_logic;
            done : out std_logic;
            clk_100MHz : in std_logic;
            dout : out STD_LOGIC_VECTOR(7 downto 0)
        );
end component;

component reg_a_top
    Port (
            din : in  STD_LOGIC_VECTOR(7 downto 0);
```

```
            addr : in  STD_LOGIC_VECTOR(10 downto 0);
            nrst : in std_logic;
            enb : in std_logic;
            op : in std_logic;
            clr : in std_logic;
            done : out std_logic;
            clk_100MHz : in std_logic;
            dout : out STD_LOGIC_VECTOR(7 downto 0)
        );
end component;

component registry_bank_top
    Port (
            din : in  STD_LOGIC_VECTOR(7 downto 0);
            addr : in  STD_LOGIC_VECTOR(3 downto 0);
            nrst : in std_logic;
            enb : in std_logic;
            op : in std_logic;
            clr : in std_logic;
            done : out std_logic;
            clk_100MHz : in std_logic;
            dout : out STD_LOGIC_VECTOR(7 downto 0)
        );
end component;

component inference_unit
    Port (
            Da : in STD_LOGIC_VECTOR (7 downto 0);
            Db : in STD_LOGIC_VECTOR (7 downto 0);
            opcode : in STD_LOGIC_VECTOR (3 downto 0);
            ena : in STD_LOGIC;
            sig : out STD_LOGIC;
            Do : out STD_LOGIC_VECTOR (7 downto 0)
        );
end component;

component instruction_register
    Port (
            clk   : in  STD_LOGIC;
            nrst  : in  STD_LOGIC;
            en    : in  STD_LOGIC;
            din   : in  STD_LOGIC_VECTOR(31 downto 0);
            op    : in  STD_LOGIC_VECTOR(2 downto 0);
            divad : in  STD_LOGIC;
            iout  : out STD_LOGIC_VECTOR(7 downto 0)
        );
end component;

component demux
    port (
            din  : in  std_logic_vector(7 downto 0);
            sel  : in  std_logic;
```

```
            ena  : in  std_logic;
            y0   : out std_logic_vector(7 downto 0);
            y1   : out std_logic_vector(7 downto 0)
        );
end component;

component dflipflop
    Port (
            clk  : in  std_logic;
            rst  : in  std_logic;
            enb  : in  std_logic;
            Din  : in  std_logic_vector(7 downto 0);
            Do   : out std_logic_vector(7 downto 0)
        );
end component;

component customMux
    Port (
            I0 : in  STD_LOGIC_VECTOR(7 downto 0);
            I1 : in  STD_LOGIC_VECTOR(7 downto 0);
            Y  : out STD_LOGIC_VECTOR(7 downto 0);
            S0 : in  STD_LOGIC;
            S1 : in  STD_LOGIC
        );
end component;

-- Internal signals
signal clksig, rstsig   : STD_LOGIC;

-- Instruction register signals
signal inr_enb          : STD_LOGIC;
signal inr_op           : STD_LOGIC_VECTOR(2 downto 0);
signal inr_div          : STD_LOGIC;
signal inr_out          : STD_LOGIC_VECTOR(7 downto 0);
signal ro_dout          : STD_LOGIC_VECTOR(31 downto 0);

-- Demux signals
signal dm_sel, dm_enb   : STD_LOGIC;

signal datasig_dmux0    : STD_LOGIC_VECTOR(7 downto 0);
signal datasig_dmux1    : STD_LOGIC_VECTOR(7 downto 0);

-- Flip flop
signal dff_enb          : STD_LOGIC;

-- Inference unit
signal inu_op           : STD_LOGIC_VECTOR(3 downto 0);
signal inu_enb          : STD_LOGIC;

-- A Register
signal ar_addr          : STD_LOGIC_VECTOR(10 downto 0);
signal ar_enb, ar_op, ar_clr, ar_d : STD_LOGIC;
```

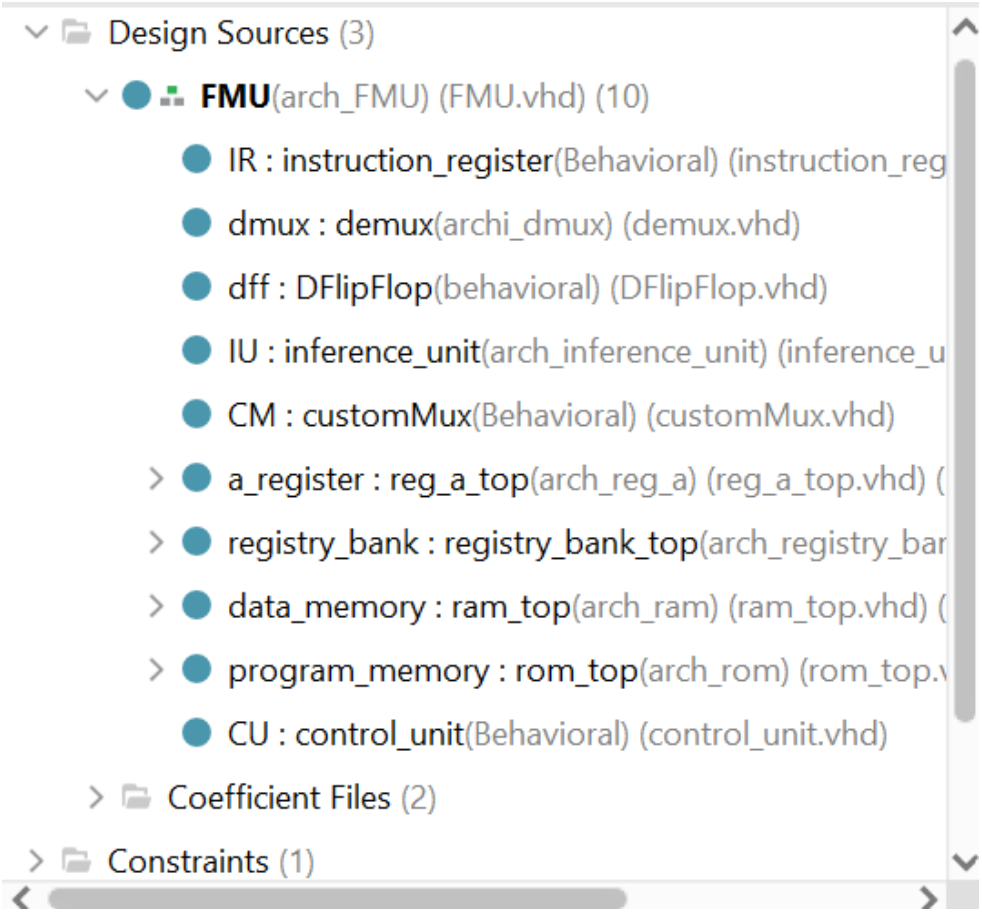
```
    signal ar_do            : STD_LOGIC_VECTOR(7 downto 0);

    -- Registry bank
    signal re_addr          : STD_LOGIC_VECTOR(3 downto 0);
    signal re_enb, re_op, re_clr, re_d : STD_LOGIC;
    signal re_do            : STD_LOGIC_VECTOR(7 downto 0);

    -- RAM
    signal ra_addr          : STD_LOGIC_VECTOR(15 downto 0);
    signal ra_enb, ra_op, ra_clr, ra_d : STD_LOGIC;
    signal ra_dout          : STD_LOGIC_VECTOR(7 downto 0);

    -- ROM
    signal ro_din           : STD_LOGIC_VECTOR(31 downto 0);
    signal ro_addr          : STD_LOGIC_VECTOR(15 downto 0);
    signal ro_wea, ro_op, ro_enb, ro_clr, ro_d : STD_LOGIC;

    --Custom Mux
    signal cmI0, cmI1, cmY  : STD_LOGIC_VECTOR(7 downto 0);
    signal cmS0, cmS1       : STD_LOGIC;


begin

    -- clksig <= ext_clk;
    -- rstsig <= ext_rst;

    -- Instruction Register
    IR: instruction_register
        port map (
            clk    => ext_clk,
            nrst   => ext_rst,
            en     => inr_enb,
            din    => ro_dout,
            op     => inr_op,
            divad  => inr_div,
            iout   => inr_out
        );

    -- Demux
    dmux: demux
        port map (
            din  => inr_out,
            sel  => dm_sel,
            ena  => dm_enb,
            y0   => datasig_dmux0,
            y1   => cmI0
        );

    -- Flip Flop
    dff: dflipflop
        port map (
```

```
            clk   => ext_clk,
            rst   => ext_rst,
            enb   => dff_enb,
            Din   => datasig_dmux0,
            Do    => ext_do
        );

    -- Inference Unit
    IU: inference_unit
        port map (
            Da     => ar_do,
            Db     => re_do,
            opcode => inu_op,
            ena    => inu_enb,
            sig    => cmS1,
            Do     => cmI1
        );

    CM: customMux
        port map (
            I0 => cmI0,
            I1 => cmI1,
            Y => datasig_dmux1,
            S0 => '0',
            S1 => cmS1
        );

    -- A Register
    a_register: reg_a_top
        port map (
            din        => datasig_dmux1,
            addr       => ar_addr,
            nrst       => ext_rst,
            enb        => ar_enb,
            op         => ar_op,
            clr        => ar_clr,
            done       => ar_d,
            clk_100MHz => ext_clk,
            dout       => ar_do
        );

    -- Registry Bank
    registry_bank: registry_bank_top
        port map (
            din        => datasig_dmux0,
            addr       => re_addr,
            nrst       => ext_rst,
            enb        => re_enb,
            op         => re_op,
            clr        => re_clr,
            done       => re_d,
            clk_100MHz => ext_clk,
```

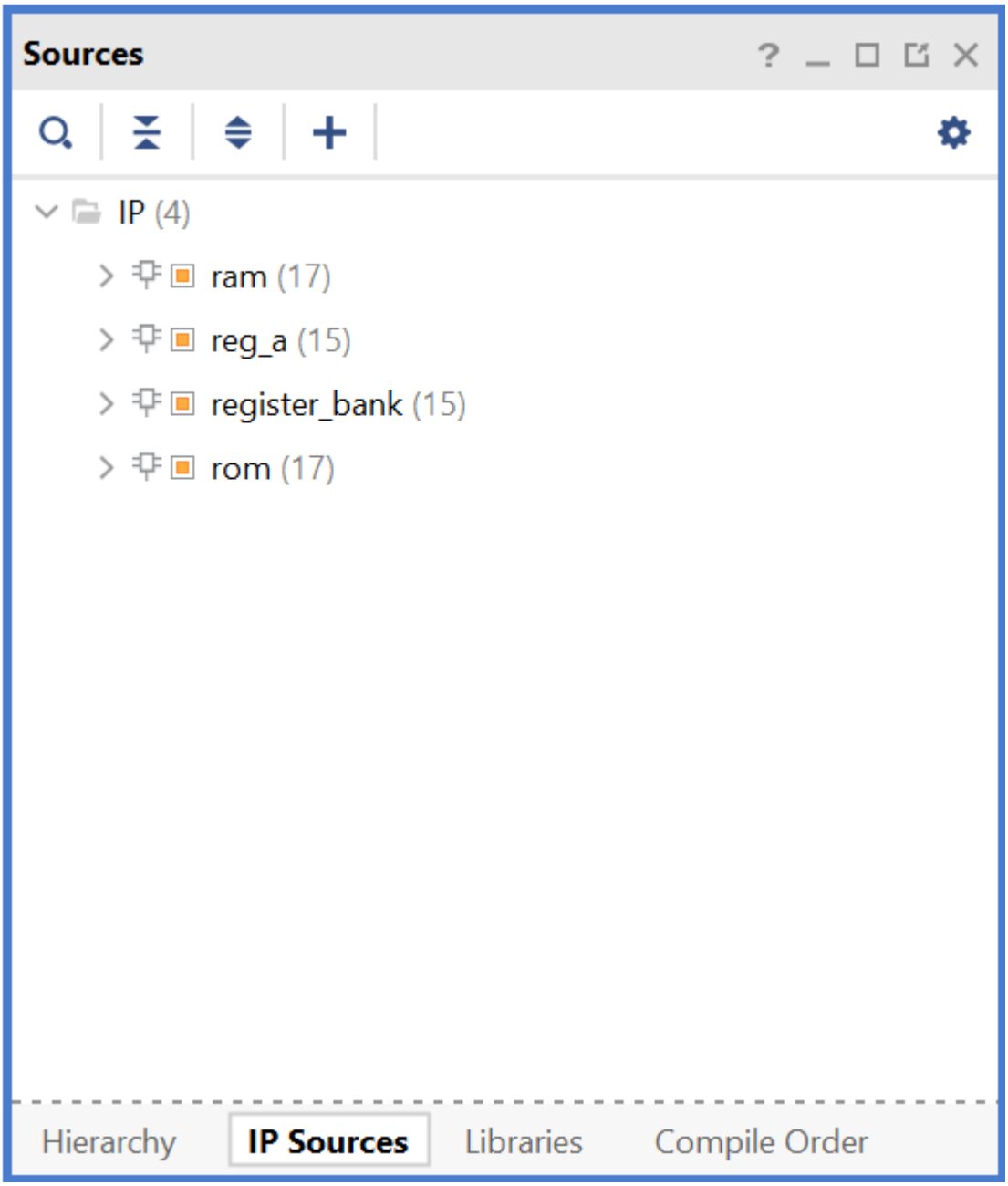
```
        dout       => re_do
    );

-- RAM
data_memory: ram_top
    port map (
        din        => datasig_dmux1,
        addr       => ra_addr,
        nrst       => ext_rst,
        enb        => ra_enb,
        op         => ra_op,
        clr        => ra_clr,
        done       => ra_d,
        clk_100MHz => ext_clk,
        dout       => ra_dout
    );

-- ROM
program_memory: rom_top
    port map (
        din        => ro_din,
        addr       => ro_addr,
        nrst       => ext_rst,
        enb        => ro_enb,
        op         => ro_op,
        clr        => ro_clr,
        done       => ro_d,
        clk_100MHz => ext_clk,
        dout       => ro_dout
    );

-- Control Unit
CU: control_unit
    port map (
        -- Inputs
        regdata   => re_do,
        irdata    => inr_out,
        ramdata   => ra_dout,
        adata     => ar_do,
        ram_done  => ra_d,
        rom_done  => ro_d,
        reg_done  => re_d,
        a_done    => ar_d,
        ba        => ext_data,
        runp      => ext_runp,
        savep     => ext_savep,
        readd     => ext_readd,
        pulse     => ext_pulse,
        enb       => ext_enb,
        rst       => ext_rst,
        clk       => ext_clk,
```

```
            -- Outputs
            reg_ad    => re_addr,
            reg_wea   => re_op,
            reg_ena   => re_enb,
            reg_clr   => re_clr,

            a_ad      => ar_addr,
            a_wea     => ar_op,
            a_ena     => ar_enb,
            a_clr     => ar_clr,

            rom_din   => ro_din,
            rom_ad    => ro_addr,
            rom_wea   => ro_wea,
            rom_ena   => ro_enb,
            rom_clr   => ro_clr,

            ram_ad    => ra_addr,
            ram_wea   => ra_op,
            ram_ena   => ra_enb,
            ram_clr   => ra_clr,

            opcode    => inu_op,
            inf_ena   => inu_enb,
            ir_enb    => inr_enb,
            ir_op     => inr_op,
            divad     => inr_div,
            ir_sel    => dm_sel,
            mux_en    => dm_enb,
            flipflop_enb => dff_enb,
            prog_done => ext_prog_done,
            err       => ext_err
        );

end arch_FMU;
```

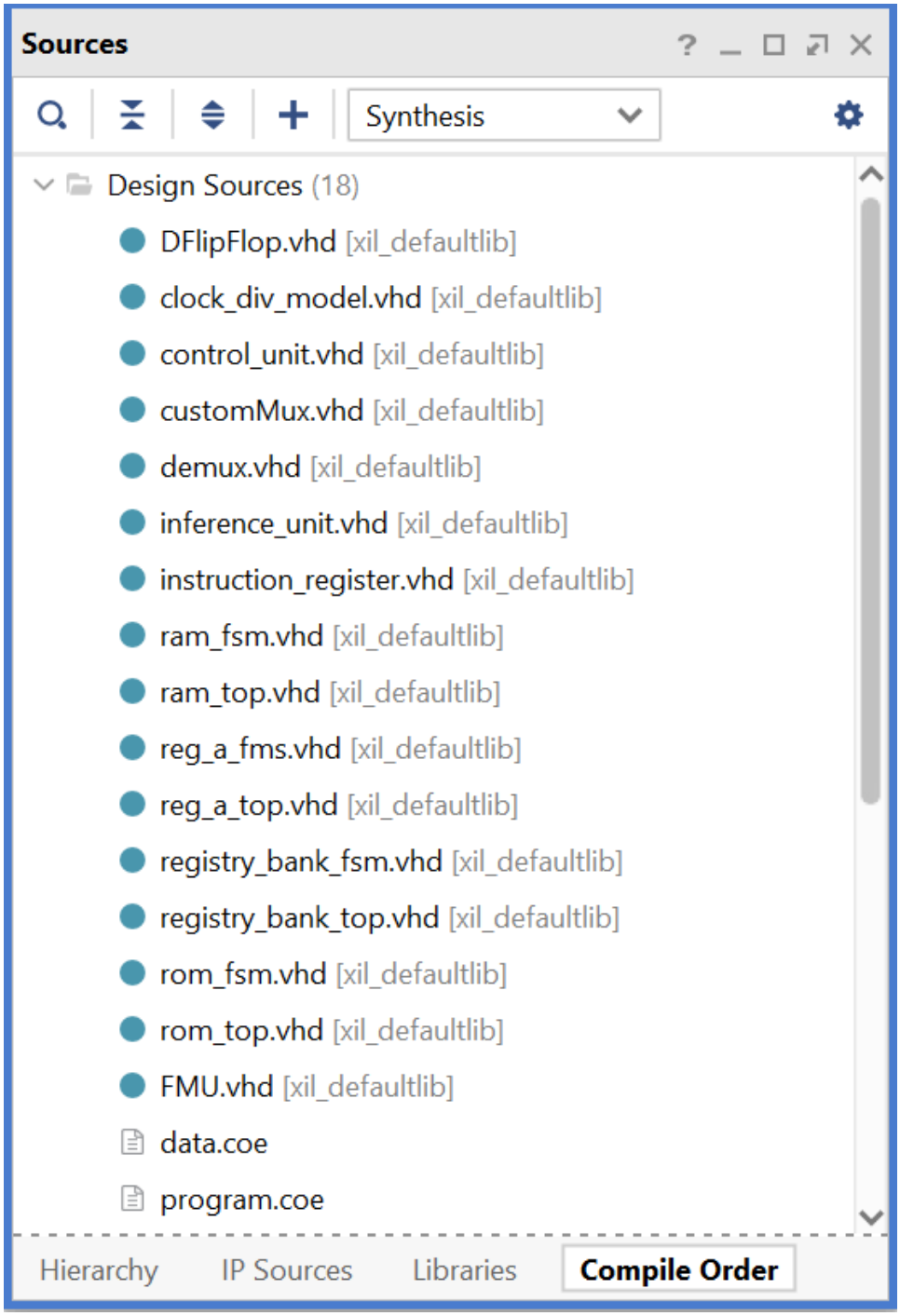
FMU Component Hierarchy



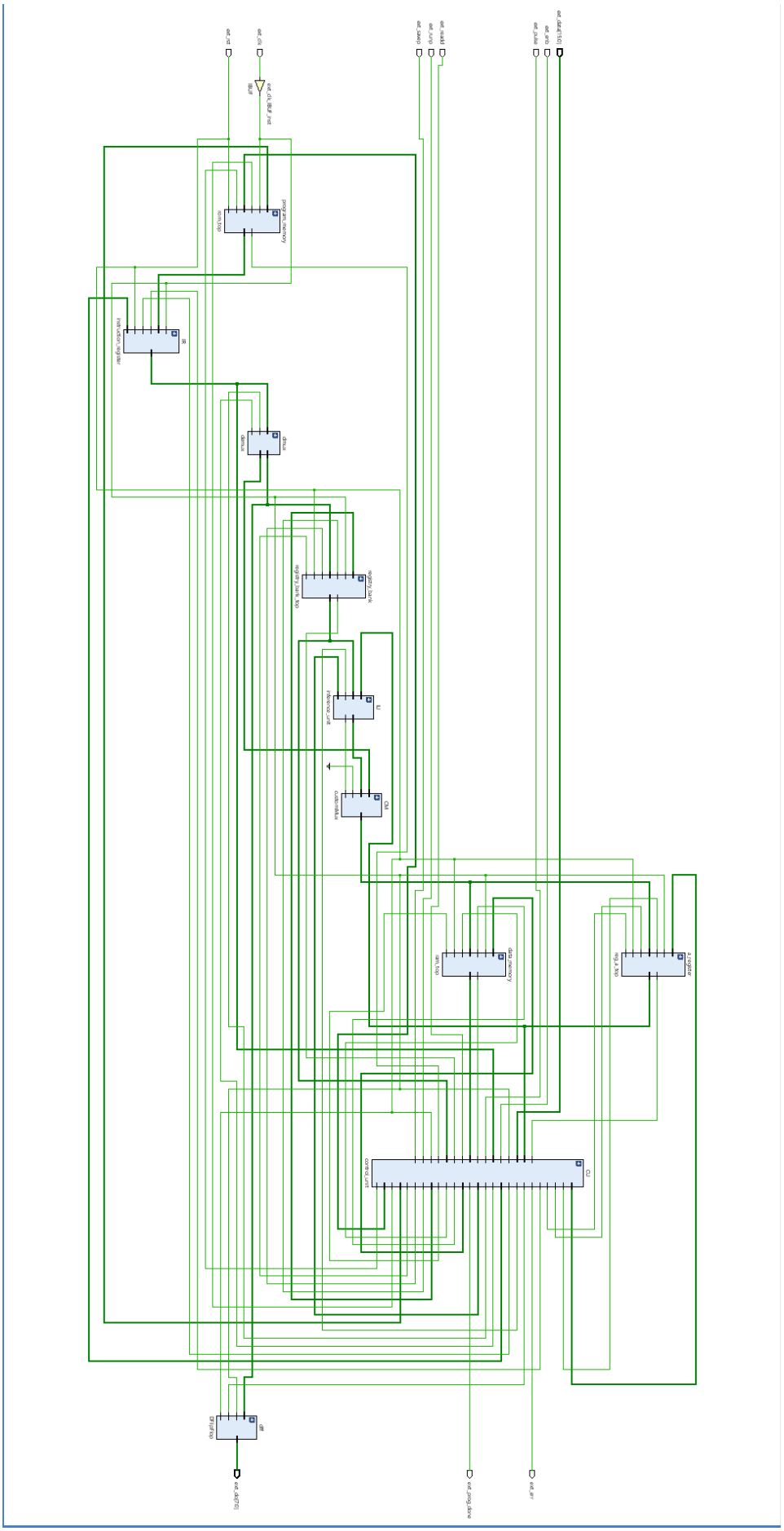
FMU IP Sources



FMU Compile Order



FMU RTL Schematic



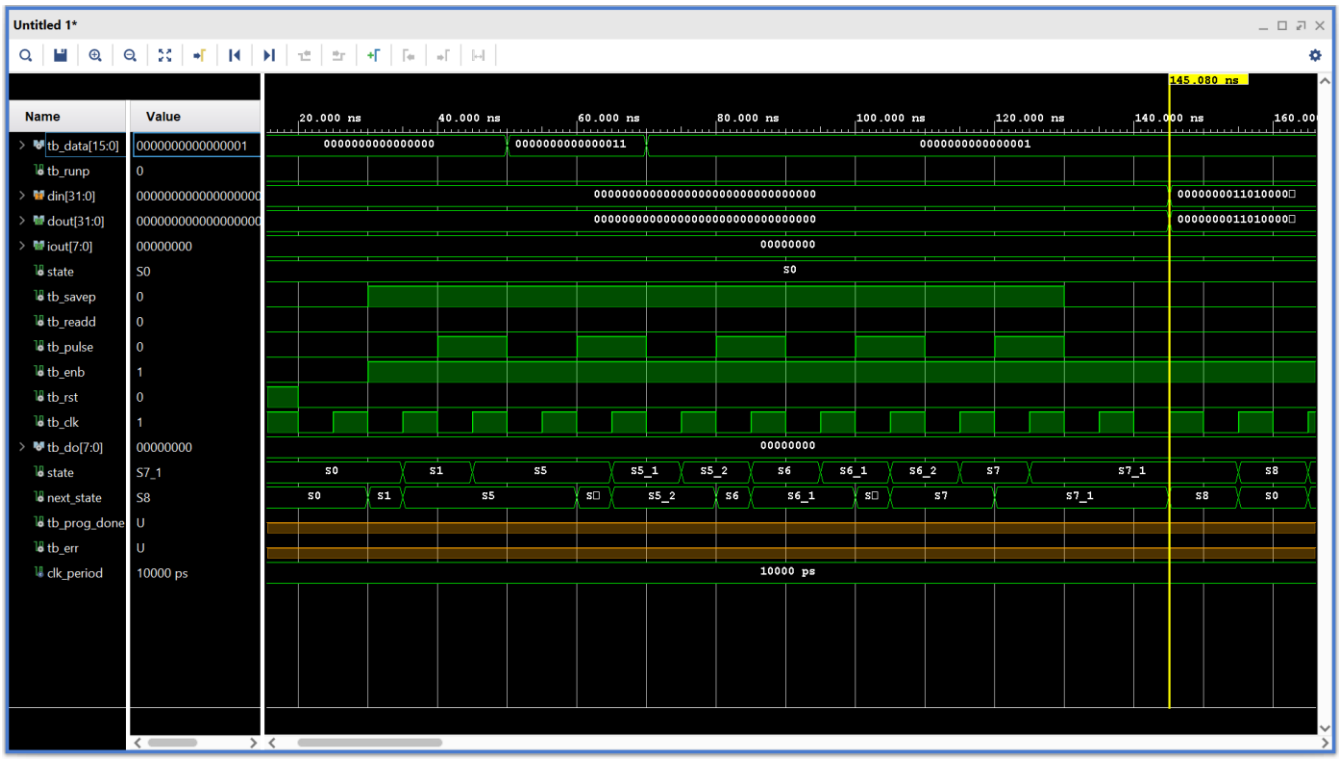
FMU Test Bench Code

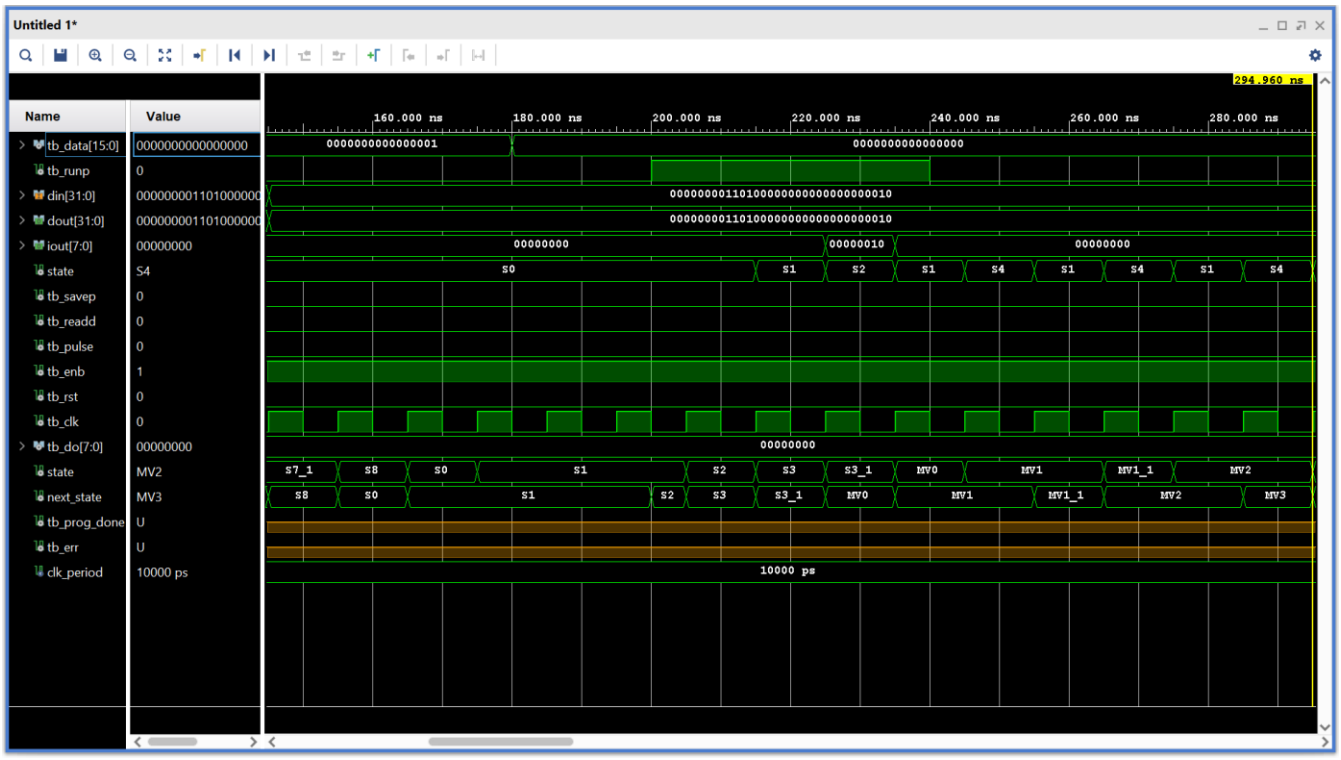
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tb_FMU is
end tb_FMU;

architecture Behavioral of tb_FMU is

    -- Component Declaration for the Unit Under Test (UUT)
    component FMU
        Port (
            ext_data         : in STD_LOGIC_VECTOR(15 downto 0);
            ext_runp         : in STD_LOGIC;
            ext_savep        : in STD_LOGIC;
            ext_readd        : in STD_LOGIC;
            ext_pulse        : in STD_LOGIC;
            ext_enb          : in STD_LOGIC;
            ext_rst          : in STD_LOGIC;
            ext_clk          : in  STD_LOGIC;
            ext_do           : out STD_LOGIC_VECTOR(7 downto 0);
            ext_prog_done    : out STD_LOGIC;
            ext_err          : out STD_LOGIC
        );
    end component;

    --Inputs
    signal tb_data         : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
    signal tb_runp         : STD_LOGIC := '0';
    signal tb_savep        : STD_LOGIC := '0';
    signal tb_readd        : STD_LOGIC := '0';
    signal tb_pulse        : STD_LOGIC := '0';
    signal tb_enb          : STD_LOGIC := '0';
    signal tb_rst          : STD_LOGIC := '0';
    signal tb_clk          : STD_LOGIC := '0';

    --Outputs
    signal tb_do           : STD_LOGIC_VECTOR(7 downto 0);
    signal tb_prog_done    : STD_LOGIC;
    signal tb_err          : STD_LOGIC;

    -- Clock period definitions
    constant clk_period : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: FMU
        port map (
            ext_data        => tb_data,
```

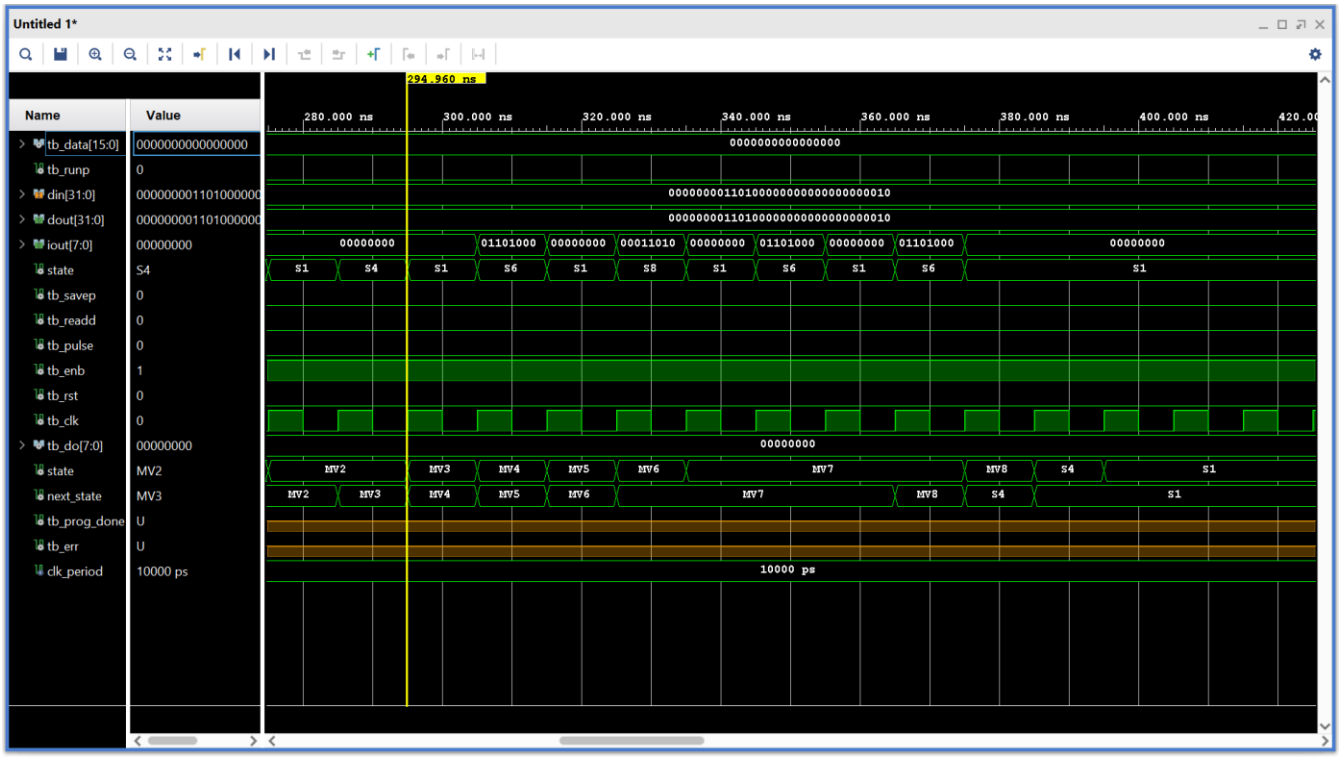
```
            ext_runp        => tb_runp,
            ext_savep       => tb_savep,
            ext_readd       => tb_readd,
            ext_pulse       => tb_pulse,
            ext_enb         => tb_enb,
            ext_rst         => tb_rst,
            ext_clk         => tb_clk,
            ext_do          => tb_do,
            ext_prog_done   => tb_prog_done,
            ext_err         => tb_err
        );

    -- Clock process definition
    clk_process :process
    begin
        tb_clk <= '0';
        wait for clk_period/2;
        tb_clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Reset the system
        tb_rst <= '1';
        wait for 20 ns;
        tb_rst <= '0';
        wait for 10 ns;

        -- Enable the FMU
        tb_enb <= '1';
        -- Start the save process
         tb_savep <= '1';
         wait for 10 ns;

         tb_pulse <= '1';
         wait for 10 ns;
         tb_pulse <= '0'; -- Enter S5_1
         -- In S5_1, set the program counter base address (pc)
         tb_data <= x"0003";
         wait for 10 ns;
         tb_pulse <= '1'; wait for clk_period; tb_pulse <= '0'; -- Enter S6
         -- In S6, set the high part of the instruction
         tb_data <= x"0001"; -- prog_H
         wait for 10 ns;
         tb_pulse <= '1'; wait for clk_period; tb_pulse <= '0'; -- Enter S6_2
         -- In S6_2, set the low part of the instruction
         tb_data <= x"0001"; -- prog_L
         wait for 10 ns;
         tb_pulse <= '1';
         wait for 10 ns;
```

```
         tb_pulse <= '0'; -- Enter S7
         wait for 10 ns;
         tb_pulse <= '1';
         wait for 10 ns;
         tb_pulse <= '0'; -- Enter S7_1
         tb_savep <= '0';
         wait for 50 ns;

        tb_data <= x"0000";
        wait for 20 ns;
        tb_runp <= '1';
        wait for 40 ns;
        tb_runp <= '0';

        -- -- Instruction 2: Base Address for Registers
        -- tb_data <= x"0000";
        -- tb_pulse <= '1';
        -- wait for clk_period;
        -- tb_pulse <= '0';
        -- wait for clk_period;

        -- -- Instruction 3: Value to store in register
        -- tb_data <= x"1234";
        -- tb_pulse <= '1';
        -- wait for clk_period;
        -- tb_pulse <= '0';
        -- wait for clk_period;

        -- -- Instruction 4: MV0
        -- tb_data <= x"0002";
        -- tb_pulse <= '1';
        -- wait for clk_period;
        -- tb_pulse <= '0';
        -- wait for clk_period;

        -- -- End the save process
        -- tb_savep <= '0';
        -- tb_enb <= '0';

        wait;
    end process;

end Behavioral;
```

FMU Timing Diagram






Untitled 1*
294.960 ns
280.000 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
400.000 ns
Name
Value
tb_data[15:0]
0000000000000000
tb_runp
0
din[31:0]
00000000110100000000000000000010
dout[31:0]
00000000110100000000000000000010
iout[7:0]
00000000
state
S4
tb_savep
0
tb_readd
0
tb_pulse
0
tb_enb
1
tb_rst
0
tb_clk
0
tb_do[7:0]
00000000
state
MV2
next_state
MV3
tb_prog_done
U
tb_err
U
clk_period
10000 ps
01101000
00011010
S1
S4
S6
S8
MV3
MV4
MV5
MV6
MV7
MV8

FMU.xdc Code For Pin Definitions

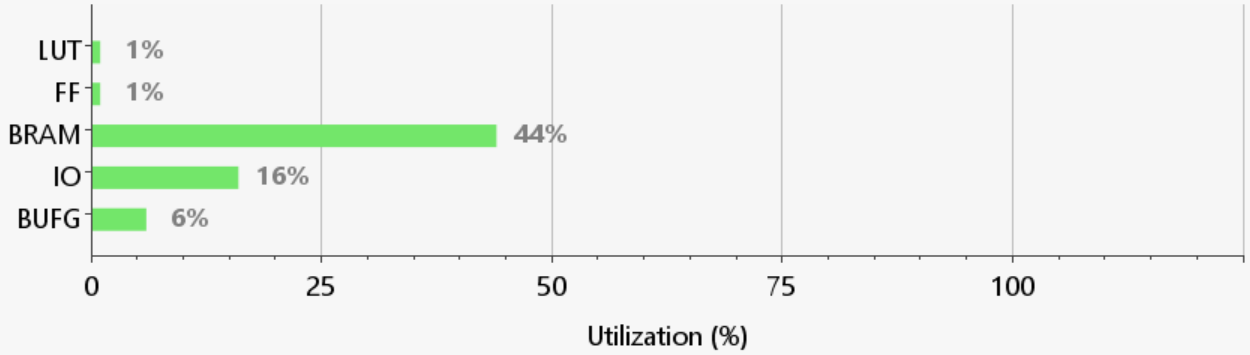
```
## Nexys A7-100T Constraint File for FMU Module

##

## This file maps the ports of the 'FMU' entity to the

## physical pins of the XC7A100T-1CSG324C FPGA.


## Clock

# 100MHz System Clock

set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { ext_clk }]

create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} [get_ports { ext_clk
}]


## Switches (ext_data[15:0])

# All switches use LVCMOS33. SW0-7 are pulled up to 3.3V .

# SW8-15 are pulled up to 1.8V , which is a valid Vih (HIGH)

# for the 3.3V FPGA bank (Bank 14) .

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { ext_data[0] }]  ;
# SW0

set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { ext_data[1] }]  ;
# SW1

set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { ext_data[2] }]  ;
# SW2

set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { ext_data[3] }]  ;
# SW3

set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { ext_data[4] }]  ;
# SW4

set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { ext_data[5] }]  ;
# SW5

set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { ext_data[6] }]  ;
# SW6

set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { ext_data[7] }]  ;
# SW7

set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS33 } [get_ports { ext_data[8] }]  ;
# SW8

set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports { ext_data[9] }]  ;
# SW9

set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { ext_data[10] }];
# SW10

set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { ext_data[11] }];
# SW11
```

```
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { ext_data[12] }];
# SW12

set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { ext_data[13] }];
# SW13

set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { ext_data[14] }];
# SW14

set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { ext_data[15] }];
# SW15


## Push-buttons

set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { ext_enb }]      ;
# BTNC

set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { ext_pulse }]    ;
# BTNU

set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { ext_runp }]     ;
# BTND

set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { ext_savep }]    ;
# BTNR

set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { ext_readd }]    ;
# BTNL


## LEDs (ext_do[7:0])

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { ext_do[0] }]   ;
# LED0

set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { ext_do[1] }]   ;
# LED1

set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { ext_do[2] }]   ;
# LED2

set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { ext_do[3] }]   ;
# LED3

set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { ext_do[4] }]   ;
# LED4

set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { ext_do[5] }]   ;
# LED5

set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { ext_do[6] }]   ;
# LED6

set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { ext_do[7] }]   ;
# LED7

## Status LEDs

set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { ext_prog_done }];
# LED17_G

set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { ext_err }]      ;
# LED17_R
```

FMU Resource Utilization

Utilization Post-Synthesis | **Post-Implementation**

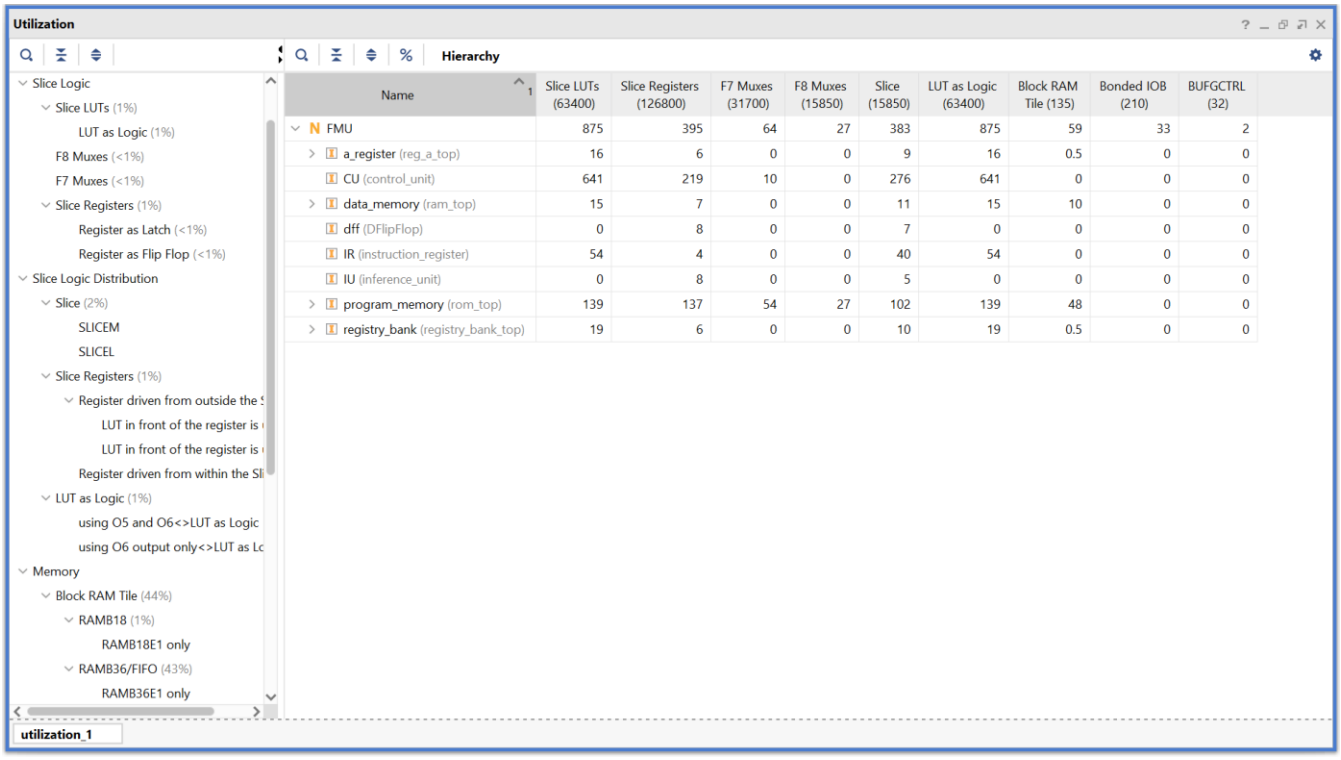
Graph | Table



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	875	63400	1.38
FF	174	126800	0.14
BRAM	59	135	43.70
IO	33	210	15.71
BUFG	2	32	6.25



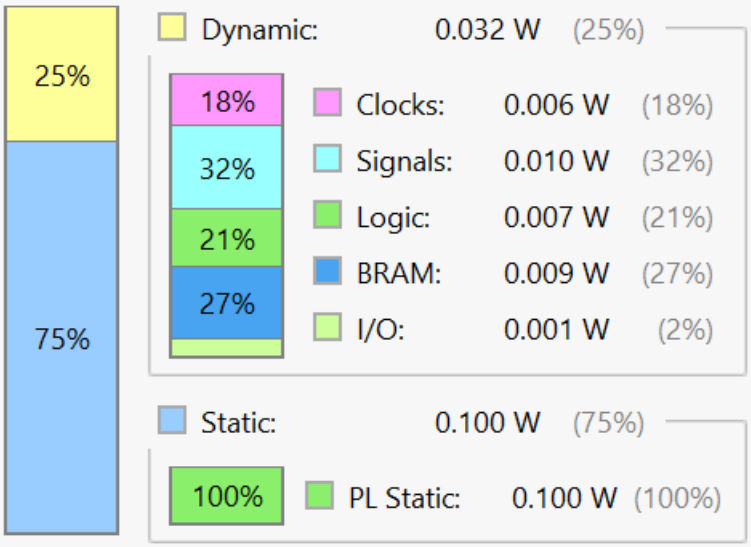
FMU Power Consumption Report

Power — **Summary** | On-Chip

Total On-Chip Power:	**0.132 W**
Junction Temperature:	**25.6 °C**
Thermal Margin:	59.4 °C (12.9 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Power — Summary | **On-Chip**



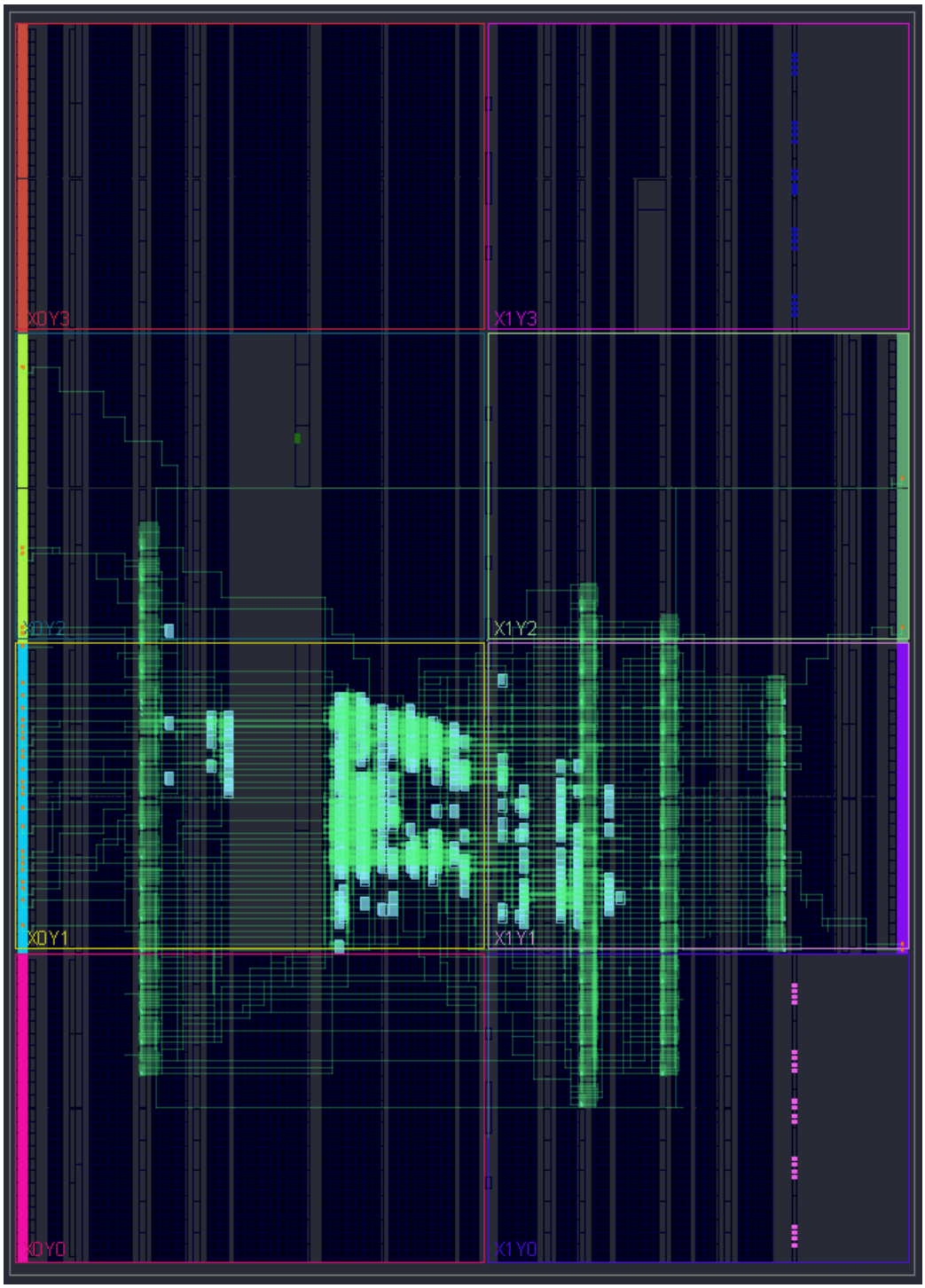
FMU Timing Summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.247 ns	Worst Hold Slack (WHS):	0.187 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	212	Total Number of Endpoints:	212	Total Number of Endpoints:	295

All user specified timing constraints are met.

FMU Implemented Design On xc7a100tcsg324-1



Environmental Issues

The manufacturing of advanced semiconductors like FPGAs is an extremely resource-intensive process.

- High Energy Consumption: Chip fabrication plants are one of the most energy-intensive factories in the world. They require large amounts of electricity to maintain ultra-clean, climate-controlled environments (cleanrooms) and power the complex fabrication equipment 24/7.
- High Water Usage: Manufacturing uses large quantities of ultrapure water, primarily for rinsing silicon wafers between the hundreds of chemical steps. This can impact local water supplies.
- Hazardous Materials: Chip production relies on a toxic chemicals, solvents, acids, and gases. Managing, treating, and disposing of this hazardous waste is a significant environmental challenge.
- Greenhouse Gas Emissions: In addition to the carbon footprint from energy use, the fabrication process uses potent greenhouse gases (like fluorinated compounds) for etching and cleaning.
- E-Waste: At the end of its life, the Nexys A7 board (and the FPGA on it) becomes electronic waste (e-waste). E-waste is a major global problem, as it contains heavy metals and toxic substances that can pollute soil and water if not properly recycled.

Social Impact

Positive Impacts

- Enabling Technology: FPGAs are a key enabling technology. Because they are reconfigurable, they are used to prototype new technologies, accelerate computing tasks, and power critical systems in:
    - Medicine: (e.g., in MRI machines and diagnostic equipment)
    - Telecommunications: (e.g., 5G and network infrastructure)
    - Scientific Research: (e.g., in data-acquisition systems)
- Education and Innovation: The entire purpose of the Nexys A7 board is education. It allows students, hobbyists, and researchers to learn digital design and create new devices, fostering innovation.

Negative Impacts and Challenges

- Supply Chain and Labor: Semiconductor manufacturing is a global, complex, and high-stress industry. It has faced criticism over labor practices, including long hours and intense working conditions, particularly in fabrication plants in Asia.
- Geopolitical Issues: The concentration of advanced chip manufacturing in a few specific locations (like Taiwan and South Korea) makes the supply chain vulnerable and a central point of global geopolitical tension, which can affect economic stability.
- Conflict Minerals: Like all modern electronics, FPGAs contain trace amounts of minerals such as tin, tungsten, tantalum, and gold. The sourcing of these minerals is sometimes linked to conflict zones, raising significant human rights and ethical concerns.