Bangladesh University of Engineering and Technology

Department of Electrical and Electronic Engineering

Course Number: EEE 468
Course Title: VLSI Laboratory

Group No: 1 (G1)

Project Title: Designing a RV32i compatible core with SPI interface

Submitted To

Dr. A.B.M. Harun-Ur-Rashid
Professor
Department of EEE, BUET

Mumtahina Islam Sukanya
Lecturer
Department of EEE, BUET

Submitted By

Mrinmoy Kundu	-1706001
Md. Jawad Ul Islam	-1706003
Aroni Ghosh	-1706004
Swapnil Siddiky	-1706018
Saleh Ahmed Khan	-1706053
Adib Md. Ridwan	-1706055
Sadat Tahmeed Azad	-1706064

Introduction

An RISC-V 32i core is a type of processor core that uses the RISC-V instruction set architecture and supports 32-bit instructions. The "i" in 32i stands for "integer," indicating that the core is designed to perform integer-based operations. RISC-V is an open-source instruction set architecture, which means that the specifications for the instruction set are publicly available and can be freely used and modified by anyone. RISC-V 32i cores are often used in embedded systems, such as microcontrollers, as well as in applications that require low power consumption and efficient code execution.

An interface is a set of systems (Standardized hardware and software) that the core utilizes to communicate with the outside world. There are several types of interfaces but ours is Serial Peripheral Interface (SPI).

The SPI (Serial Peripheral Interface) is a synchronous serial communication interface used for connecting devices in embedded systems. It typically consists of four lines: a clock signal, a master-out/slave-in (MOSI) data line, a master-in/slave-out (MISO) data line, and a slave select (SS) line. SPI is used for high-speed communication between microcontrollers, sensors, flash memory, and other peripherals. It's a simple and low-cost interface that supports full duplex communication and can operate in a master/slave configuration. The SPI interface is widely used in applications where high data rates and low-latency communication are critical, such as in digital cameras, audio codecs, and LCD displays.

Objectives

- Design (with RTL) and simulate a 32-bit rv32i compatible core.
- Use core to execute instructions written in C that's converted into 32-bit hexadecimal format.
- Load disassembled instructions into the instruction memory of the processor during verification with the help of a chosen interface (SPI).
- Synthesize and perform the physical design of Hardware Description of their RISC-V processor.

Design Specifications

1. Core must be compatible with the rv32i instruction set.
2. Implement RISC-V ISA spec. Reference: RV32i-ISA
3. Use input global clock (positive edge triggered: clk), global reset (negative edge triggered: rst_n) and core_select signal.
4. Create a memory controller with an industry standard interface (SPI) to load the instruction data to the memory.
5. core_select functionality:

core_select	Functionality
0	1. Memory controller interface will be active to write data into the memory.
1	1. Core will start to work 2. Core side interface of the memory will be active to read and write data into the memory.

6. Core HDL descriptions need to be Synthesizable and optimized.
7. No pipelining stages are necessary.

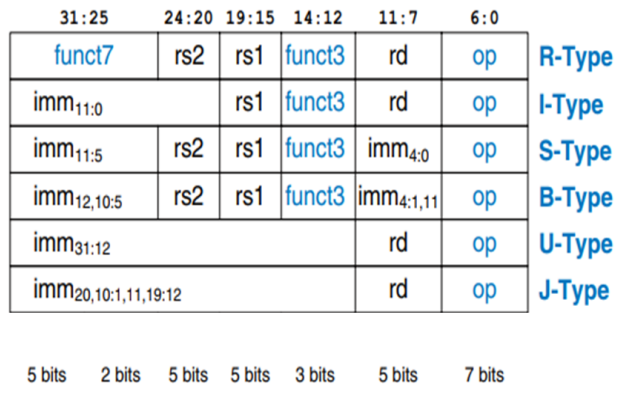
Design Software

ModelSim, Cadence NCSim, Cadence Genus, Cadence Innovus

RISC-V Core

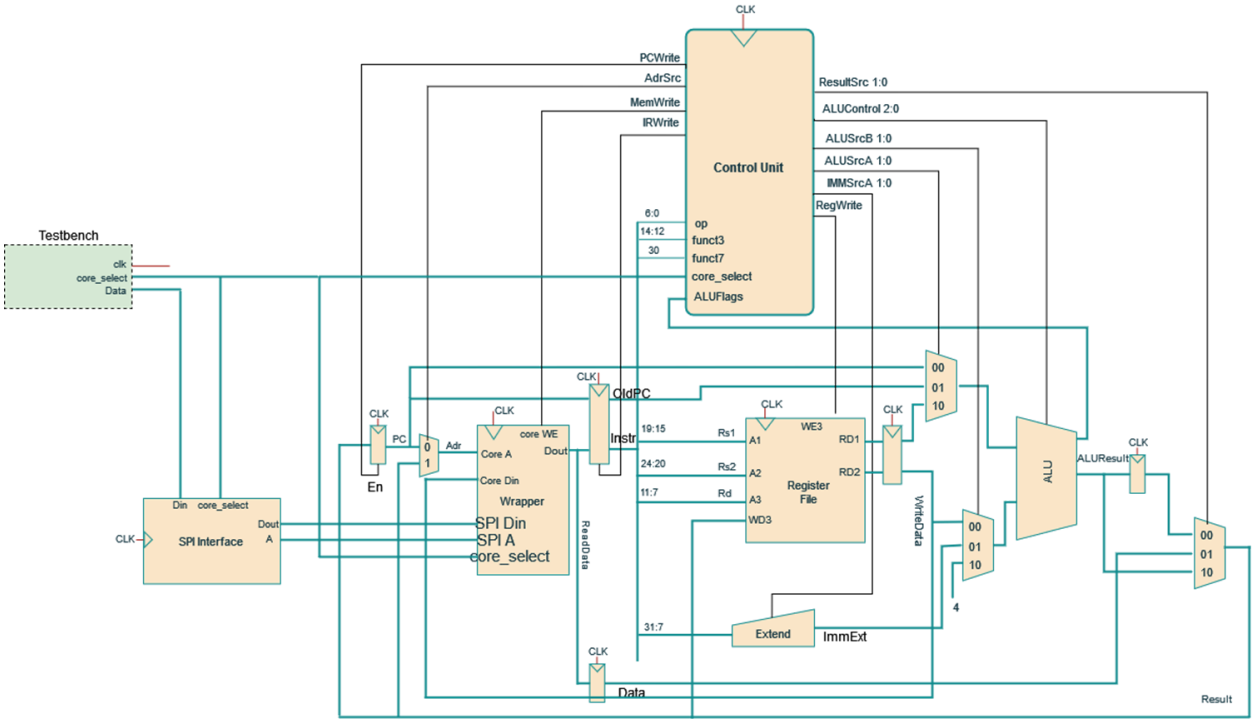
Instruction Set Architecture

- The instruction set architecture - the entire group of commands that the processor can perform to execute the program instructions.



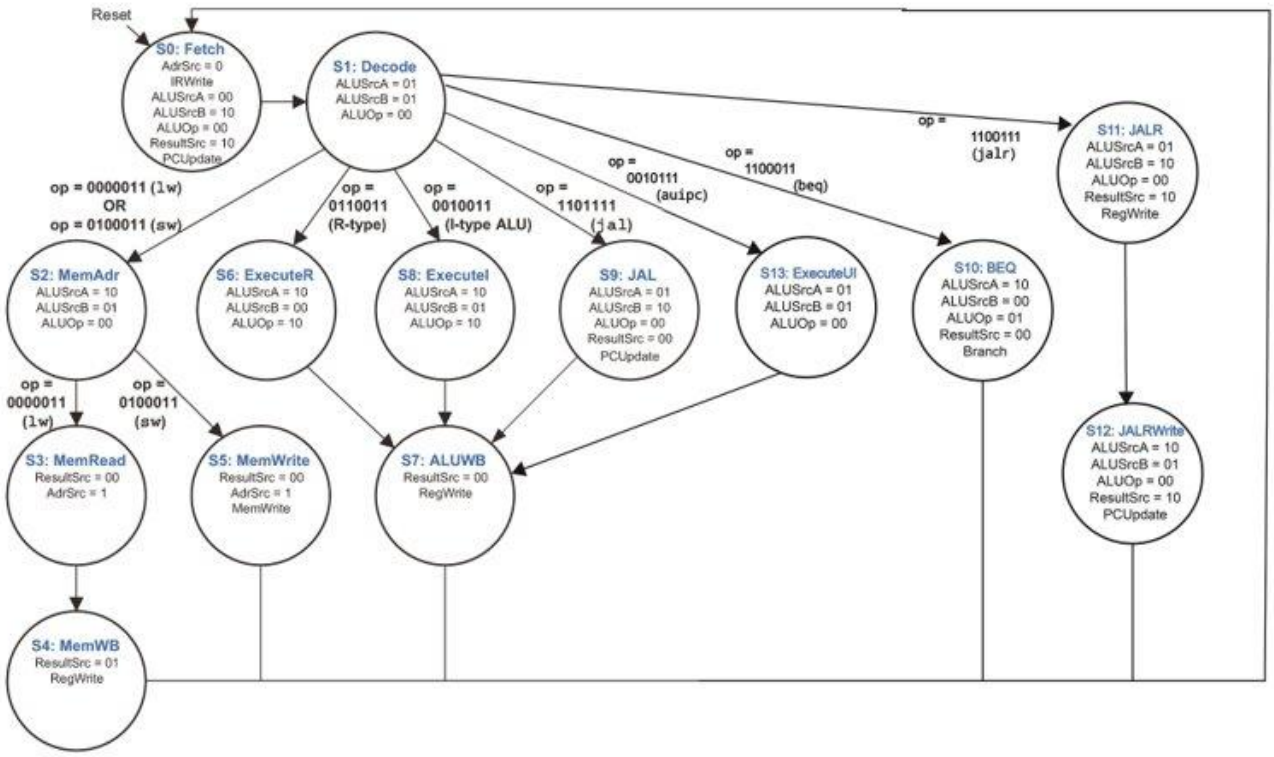
- RV32I ISA is a 32-bit version of the RISC-V ISA --> data and address buses are 32 bits wide.
- Includes 6 types of instructions.
- Consists of 37 unique instructions.

Block Diagram of the Core



This is the block diagram of our core including testbench, interface and wrapper. Data is sent to the serial peripheral interface & then to memory wrapper. The datapath of an RV32I multicycle core typically consists of several main components, including a register file, an arithmetic and logic unit (ALU), a program counter, and various multiplexers and control logic. The program counter is a register that holds the memory address of the next instruction to be fetched. ALU is responsible for performing arithmetic and logic operations on the data values based on ALUCOntrol signal from the control unit. The Main FSM produces multiplexer select, register enable, and memory write enable signals for the data path. Enable signals (RegWrite, MemWrite, IRWrite, PCUpdate, and Branch) are listed only when they are asserted; otherwise, they are 0. The control unit computes the control signals based on the op, funct3, and funct75 fields of the instruction (Instr6:0, Instr14:12, and Instr30).

Core Control Unit- FSM



For a multicycle riscv-32i processor, 6 types of instructions are implemented by 15 states including one idle state for core_select=0. As we can see from the state diagram, all types of instructions include fetching the instruction from memory at the address held in the program counter & then decoding the instruction by reading the register file. This step figures out what operation should be performed. After that, the remaining steps are different for different types of instructions. Overall, the multicycle processor requires 3 cycles for branch, 4 cycles for R-type, I-type, jump, store and 5 cycles for load instructions.

Core: ISA support

Required Instructions for “Non Prime” testbench instruction

Instruction	Type	Comment
auipc rd, upimm	U	
addi rd, rs1, imm	I	add imm to reg
add rd, rs1, rs2	R	add regs
jal rd, label	J	Jump and link, PC = JTA, rd = PC + 4
sw rs2, imm(rs1)	S	Store word
lw rd, imm(rs1)	I	Load Word
beq rs1, rs2, label	B	Branch if (rs1 == rs2), PC = BTA
bne rs1, rs2, label	B	Branch if (rs1 ≠ rs2), PC = BTA
sub rd, rs1, rs2	R	substract reg
bge rs1, rs2, label	B	if (rs1 ≥ rs2) PC = BTA
jalr rd, rs1, imm	I	jump and link register, PC = rs1 + SignExt(imm), rd = PC + 4

Additional Instructions Implemented-

ori rd, rs1, imm	I	
andi rd, rs1, imm	I	
xori rd, rs1, imm	I	
slli rd, rs1, umm	I	rd = rs1 << uimm (a 5 bit unsigned int)
srli rd, rs1, umm	I	rd = rs1 >> uimm (a 5 bit unsigned int)
srai rd, rs1, uimm	I	rd = rs1 >>> uimm (a 5 bit unsigned int)
slti rd, rs1, imm	I	rd = rs1 < SingExt(Imm) ; signed comparison
sltiu rd, rs1, imm	I	rd = rs1 < SingExt(Imm) ; unsigned comparison
or rd, rs1, rs2	R	rd = rs1 | rs2
and rd, rs1, rs2	R	rd = rs1 & rs2
xor rd, rs1, rs2	R	rd = rs1 ^ rs2
sll rd, rs1, rs2	R	rd = rs1 << rs2[4:0]
srl rd, rs1, rs2	R	rd = rs1 >> rs2[4:0]
sra rd, rs1, rs2	R	rd = rs1 >>> rs2[4:0]
slt rd, rs1, rs2	R	rd = rs1 < rs2; signed comparison
sltu rd, rs1, rs2	R	rd = rs1 < rs2; unsigned comparison
blt rs1, rs2, label	B	Branch if (rs1 < rs2) PC = BTA
bltu rs1, rs2, label	B	branch if < unsigned
bgeu rs1, rs2, label	B	branch if ≥ unsigned

Instructions Not Implemented-

sll rd, rs1, rs2	R	rd = rs1 << rs2[4:0]
srl rd, rs1, rs2	R	rd = rs1 >> rs2[4:0]
sra rd, rs1, rs2	R	rd = rs1 >>> rs2[4:0]
slt rd, rs1, rs2	R	rd = rs1 < rs2; signed comparison
sltu rd, rs1, rs2	R	rd = rs1 < rs2; unsigned comparison
blt rs1, rs2, label	B	Branch if (rs1 < rs2) PC = BTA
bltu rs1, rs2, label	B	branch if < unsigned
bgeu rs1, rs2, label	B	branch if ≥ unsigned

Some instructions have not been implemented mainly due to **byte addressability.** Thus, in total, **30 out of 37** rv32i instructions are functional in our design.

RTL Codes

A.v:

```
//////////////////////////////////////////////////////////////////////////////////
// RV32i RTL
// Create Date: 09/01/2023
// Module Name: RISCV_ALU
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Diganta & Co.
//////////////////////////////////////////////////////////////////////////////////
module A(
input rst,clk,
input [31:0]read_data_1,read_data_2,
output reg [31:0]Aread_data_1,Aread_data_2
    );

    always @ (posedge clk or negedge rst) begin
     if (!rst)
                        begin
                                Aread_data_1 <= 0;
                                Aread_data_2 <= 0;
                        end
      else if (clk)
                        begin
                                Aread_data_1 <= read_data_1;
                                Aread_data_2 <= read_data_2;
                        end
    end

endmodule
```

ALU.v:

```
//////////////////////////////////////////////////////////////////////////////////
// RV32i RTL
// Create Date: 09/01/2023
// Module Name: RISCV_ALU
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Diganta & Co.
//////////////////////////////////////////////////////////////////////////////////


// ALU is incomplete, must add more instructions later.
module ALU(input rst,
                        input reg[31:0]a,b,
          input reg[3:0]ALUControl,
          output reg[3:0] flags,
          output reg[31:0]ALUResult
         );

  reg [31:0] b_not;

  always@(*)begin
            if(!rst) ALUResult <= 0;
            else
                      case(ALUControl)
                        4'b0000:            {flags[1],ALUResult} <= a+b;

                        4'b0001:            begin
                                            b_not = ~b + 1;
                                            {flags[1],ALUResult} <= a+b_not;
                                            end

                        4'b0101:  ALUResult <= (a[31]^b[31])? a[31] : (a < b);
                                                                  //slt, slti, signed comparison
                        4'b1101:            ALUResult <= a<b; //sltu, sltiu, unsigned comparison
                        4'b0011:            ALUResult <= a|b; //or
                        4'b0010:            ALUResult <= a&b; //and
                        4'b0100:            ALUResult <= a^b; //xor
                        4'b0110:  ALUResult <= a << b[4:0]; //sll, slli
                        4'b0111:  ALUResult <= a >> b[4:0]; //srl, srli
                        4'b1111:  ALUResult <= $signed(a) >>> b[4:0]; //sra, srai
                      default:   ALUResult <= ALUResult;
                      endcase

    flags[2] = !(|ALUResult);
    flags[3] = ALUResult[31];
    if(ALUControl == 1)       flags[0] = a[31]&(!b[31])&(!ALUResult[31])|
                                                        (!a[31])&(b[31])&ALUResult[31]; ///for negative numbers, b
sign is inverted beforehand
                      else flags[0] = a[31]&b[31]&(!ALUResult[31])|
                      (!a[31])&(!b[31])&ALUResult[31];
  end
endmodule
```

ALU_out_reg.v:

```
//////////////////////////////////////////////////////////////////////////////////
// RV32i RTL
// Create Date: 09/01/2023
// Module Name: RISCV_ALU
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Diganta & Co.
//////////////////////////////////////////////////////////////////////////////////

module ALU_out_reg(
input rst,clk,
input [31:0]ALUResult,
output reg [31:0]ALUOut
    );

    always @ (posedge clk or negedge rst) begin
     if (!rst)
      ALUOut <= 32'h00000000;
      else
      ALUOut <= ALUResult;
    end

endmodule
```

Data_register.v:

```
//////////////////////////////////////////////////////////////////////////////////
// RV32i RTL
// Create Date: 09/01/2023
// Module Name: RISCV_ALU
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Diganta & Co.
//////////////////////////////////////////////////////////////////////////////////
module Data_register(
input rst,clk,[31:0]ReadData,
output reg [31:0]Data
    );

    always @ (posedge clk or negedge rst) begin
     if (!rst)
      Data <= 32'h00000000;
      else
      Data <= ReadData;
    end
endmodule
```

Datapath.v:

```
//////////////////////////////////////////////////////////////////////////////////
// RV32i RTL
// Create Date: 09/01/2023
// Module Name: Datapath
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Diganta & Co.
```

```
////////////////////////////////////////////////////////////////////////////////
module Datapath(
input clk,rst,PCWrite,AdrSrc,MemWrite,IRWrite,RegWrite,core_select,
input [1:0]ALUSrcA,ALUSrcB,ResultSrc,
input [2:0]ImmSrc,
input [3:0]ALUControl,
input [31:0] ReadData,
output [3:0]flags,
output [31:0]Instr,Adr,WriteData
);

parameter DATA_LENGTH = 32;
parameter ADDRESS_LENGTH = 32;
wire [31:0]PC,OldPC,read_data_2,read_data_1,ALUOut,Data,ALUResult,SrcA,SrcB,Result; //Areaddata1,2Immext
declared below
////1
program_counter pc(
.clk(clk),.rst(rst),.PCWrite(PCWrite),
.PCNext(Result),
.PC(PC)
    );

mux_2X1 mux1(
.A(PC),.B(Result),
.Sel(AdrSrc),
.mux(Adr)
    );

/*  Memory mem(
.clk(clk),.rst(rst),.MemWrite(MemWrite),
.address(Adr),.data_in(WriteData),
.data_out(ReadData)
);
  data_memory_wrapper  # (DATA_LENGTH,ADDRESS_LENGTH) wrapper(.clk(clk),
.core_select(core_select),.from_core_mem_en(core_select),
 .from_core_mem_wr_en(MemWrite), .from_core_mem_rd_en(core_select), .from_core_mem_address(Adr),
 .from_core_mem_data_in(WriteData), .from_core_mem_data_length(2'b00), .to_core_mem_data_out(ReadData)
);*/


 Instruction_register IR(
.rst(rst),.clk(clk),.IRWrite(IRWrite),.mem_data(ReadData),.PC(PC),
.OldPC(OldPC),.Instr(Instr)
    );

Data_register Data_register(
.rst(rst),.clk(clk),.ReadData(ReadData),
.Data(Data)
 );

mux_4X1 mux2(
.A(ALUOut),.B(Data),.C(ALUResult),.D(32'h00000000),
.Sel(ResultSrc),
.mux(Result)
    );

 wire [31:0] Aread_data_1,ImmExt; //Aread_data_2 not required, its same as WriteData
Registers R(
.rst(rst),.clk(clk),.RegWrite(RegWrite),
.read_reg1(Instr[19:15]),.read_reg2(Instr[24:20]),.write_address(Instr[11:7]),
.write_data(Result),
```

```
.read_data_1(read_data_1),.read_data_2(read_data_2)
);

Extend Extend(
.rst(rst),.Instr(Instr),.ImmSrc(ImmSrc),
.ImmExt(ImmExt)
);

A A(
.rst(rst),.clk(clk),.read_data_1(read_data_1),.read_data_2(read_data_2),
.Aread_data_1(Aread_data_1),.Aread_data_2(WriteData)
    );


mux_4X1 mux3(
.A(PC),.B(OldPC),.C(Aread_data_1),.D(32'h00000000),
.Sel(ALUSrcA),
.mux(SrcA)
    );

mux_4X1 mux4(
.A(WriteData),.B(ImmExt),.C(32'h00000004),.D(32'h00000000),
.Sel(ALUSrcB),
.mux(SrcB)
    );

ALU ALU(
.rst(rst),.a(SrcA),.b(SrcB),.ALUControl(ALUControl),
.flags(flags),
.ALUResult(ALUResult)
    );


ALU_out_reg Alu_out_reg(
.rst(rst),
.clk(clk),
.ALUResult(ALUResult),
.ALUOut(ALUOut)
    );

endmodule
```

Extend.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 08/31/2022 01:42:40 AM
// Design Name:
// Module Name: Imm_gen
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Extend(input rst,
            input reg [31:0]Instr,
            input wire [2:0]ImmSrc,
            output reg[31:0]ImmExt);
  always@(*)
    begin
      if(!rst)
        ImmExt <= 32'h00000000;
      else
        case(ImmSrc)
          3'b000:       ImmExt <= {{20{Instr[31]}}, Instr[31:20]};
          3'b001:       ImmExt <={{20{Instr[31]}}, Instr[31:25],Instr[11:7]};
          3'b010:       ImmExt <={{20{Instr[31]}}, Instr[7],
                         Instr[30:25], Instr[11:8], 1'b0};
          3'b011:       ImmExt <={{12{Instr[31]}}, Instr[19:12],
                         Instr[20], Instr[30:21], 1'b0};
                          3'b100: ImmExt <= {Instr[31:12],12'h000};
        endcase
    end
endmodule
```

Instruction_register.v:

```
module Instruction_register(
input rst,clk,IRWrite,
input [31:0]mem_data,PC,
output reg [31:0]OldPC,Instr
    );

    always @ (posedge clk or negedge rst) begin
     if (!rst)
                        begin
                                Instr <= 0;
                                OldPC <= 0;
                        end
      else if (IRWrite)
                        begin
                                OldPC <= PC;
                                Instr <= mem_data;
                        end
    end

endmodule
```

RV32i_core.v:

```
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/02/2022 03:24:53 AM
// Design Name:
// Module Name: RV32i_multi
// Project Name:
// Target Devices:
// Tool Versions:
```

```
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module RV32i_core(
input clk,rst,core_select,
input [31:0]ReadData,
output [31:0]Adr, WriteData,
output MemWrite
);

wire AdrSrc,IRWrite,RegWrite, PCWrite;
wire [1:0]ALUSrcA,ALUSrcB, ResultSrc;
wire [2:0]ImmSrc;
wire [3:0] ALUControl;
wire [31:0]Instr;
wire [3:0]flags;

control control_unit(
.clk(clk),.rst(rst),.core_select(core_select),.flags(flags),.op(Instr[6:0]),.funct7(Instr[31:25]),.funct3(Instr[14:12]),.AdrS
rc(AdrSrc),.IRWrite(IRWrite),
.MemWrite(MemWrite),      .RegWrite(RegWrite),      .PCWrite(PCWrite),.ALUSrcA(ALUSrcA),.ALUSrcB(ALUSrcB),
.ResultSrc(ResultSrc),.ImmSrc(ImmSrc),.ALUControl(ALUControl)
);

Datapath datapath(
 .clk(clk),.rst(rst),.PCWrite(PCWrite),.AdrSrc(AdrSrc),.MemWrite(MemWrite),
 .IRWrite(IRWrite),.RegWrite(RegWrite),.core_select(core_select),.ALUSrcA(ALUSrcA),.ALUSrcB(ALUSrcB),
 .ResultSrc(ResultSrc),.ImmSrc(ImmSrc),.ALUControl(ALUControl),.ReadData(ReadData),.flags(flags),
 .Instr(Instr),.Adr(Adr),.WriteData(WriteData)
);

endmodule
```

Registers.v:

```
module Registers(
input rst,clk,RegWrite,
input [4:0]read_reg1,read_reg2,write_address,
input [31:0]write_data,
output [31:0]read_data_1,read_data_2
);
reg [31:0]registers[0:31];

assign read_data_1 = registers[read_reg1];
assign read_data_2 = registers[read_reg2];

integer i;
always @ (posedge clk  or negedge rst)begin
        if (!rst) begin
                for (i=0;i<32;i=i+1)
                  registers[i] <= 0;
                end
```

```
        else if (RegWrite)
                registers[write_address] <= write_data;
        registers[0] <= 32'h00000000;
end
endmodule
```

Control.v:

```
module control(
input clk,rst,core_select,
input [3:0]flags,
input [6:0]op,funct7,
input [2:0]funct3,
output reg AdrSrc,IRWrite,MemWrite, RegWrite,
output PCWrite,
output reg [1:0]ALUSrcA,ALUSrcB,ResultSrc,
output reg[2:0]ImmSrc,
output reg[3:0]ALUControl
   );

reg [1:0]ALUOp;
////////////////////////////////////// FSM START ////////////////////////////////////////////////
reg PCUpdate,Branch;
reg [4:0]present_state;
reg [4:0]next_state;
parameter idle = 4'b1111,state_0 = 4'b0000,state_1 = 4'b0001,state_2 = 4'b0010,state_3 = 4'b0011,
     state_4 = 4'b0100,state_5 = 4'b0101,state_6 = 4'b0110,state_7 = 4'b0111,state_8 = 4'b1000,
          state_9 = 4'b1001,state_10 = 4'b1010,state_11 = 4'b1011,state_12 = 4'b1100, state_13 = 4'b1101;


always @ (*)
begin
        if (!rst)
                next_state <= idle;
        else if(!core_select)
                        next_state <= present_state;
        else
        begin
                case (present_state)
                        idle :    next_state <= state_0;
                        state_0 :  next_state <= state_1;
                                state_1 :  case(op)
                                                7'b0100011 : next_state <= state_2;
        //sw
                                                7'b0000011 : next_state <= state_2;
        //lw
                                                7'b0110011  :  next_state  <=  state_6;
        //R
                                                7'b0010011 : next_state <= state_8;
        //itypeALU
                                                7'b1101111  :  next_state  <=  state_9;
        //jal
                                                7'b1100011  :  next_state  <=  state_10;
        //B - type
                                                7'b1100111  :  next_state  <=  state_11;
        //jalr
                                                7'b0010111        :next_state<=  state_13;
        //uimm ops
                                                default : next_state <= next_state;
                                        endcase
```

```
                                    state_2 :  case(op)
                                                    7'b0000011 : next_state <= state_3;// load
                                                    7'b0100011 : next_state <= state_5;// store
                                                    default : next_state <= next_state;
                                            endcase
                                    state_3 :  next_state <= state_4;
                                    state_4 :  next_state <= state_0;
                                    state_5 :  next_state <= state_0;
                                    state_6 :  next_state <= state_7;
                                    state_8 :  next_state <= state_7;
                                    state_9 :  next_state <= state_7;
                                    state_7 :  next_state <= state_0;
                                    state_10 :  next_state <= state_0;
                                    state_11:           next_state <= state_12;
                                    state_12:           next_state <= state_0;
                                    state_13:           next_state <= state_7;

                        endcase
                end

end

 always @ (posedge clk or negedge rst) begin  //state register
  if (!rst)
     present_state <= idle;
  else
     present_state <= next_state;
 end

 always @ (present_state,core_select) //output block
  begin
            if (!core_select) begin IRWrite <= 0; PCUpdate <= 0; Branch <=0; RegWrite <= 0; end
            else
               case (present_state)
                        state_0 : begin AdrSrc <= 0; IRWrite <= 1; ALUSrcA <= 2'b00; ALUSrcB <= 2'b10; ALUOp <=
2'b00; ResultSrc <= 2'b10; PCUpdate <= 1; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_1 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b01; ALUSrcB <= 2'b01; ALUOp <=
2'b00; ResultSrc <= 2'b10; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_2 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b01; ALUOp <=
2'b00; ResultSrc <= 2'b10; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_3 : begin AdrSrc <= 1; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b00; ALUOp <=
2'b00; ResultSrc <= 2'b00; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_4 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b00; ALUSrcB <= 2'b00; ALUOp <=
2'b00; ResultSrc <= 2'b01; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 1; end
                        state_5 : begin AdrSrc <= 1; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b00; ALUOp <=
2'b00; ResultSrc <= 2'b00; PCUpdate <= 0; Branch <=0; MemWrite <= 1; RegWrite <= 0; end
                        state_6 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b00; ALUOp <=
2'b10; ResultSrc <= 2'b10; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_7 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b00; ALUOp <=
2'b00; ResultSrc <= 2'b00; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 1; end
                        state_8 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b01; ALUOp <=
2'b10; ResultSrc <= 2'b10; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_9 : begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b01; ALUSrcB <= 2'b10; ALUOp <=
2'b00; ResultSrc <= 2'b00; PCUpdate <= 1; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_10: begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b00; ALUOp <=
2'b01; ResultSrc <= 2'b00; PCUpdate <= 0; Branch <=1; MemWrite <= 0; RegWrite <= 0; end
                        state_11: begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b01; ALUSrcB <= 2'b10; ALUOp <=
2'b00; ResultSrc <= 2'b10; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 1; end
                        state_12: begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b10; ALUSrcB <= 2'b01; ALUOp <=
2'b00; ResultSrc <= 2'b10; PCUpdate <= 1; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
                        state_13: begin AdrSrc <= 0; IRWrite <= 0; ALUSrcA <= 2'b01; ALUSrcB <= 2'b01; ALUOp <=
```

```
2'b00; ResultSrc <= 2'b00; PCUpdate <= 0; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
            endcase
  end
//////////////////////////////////////// FSM END ////////////////////////////////////////////////


//////////////////////////////////////// Branch Control ////////////////////////////////////////
reg BranchCond;

always@(*)
        case(funct3)
                3'b000: BranchCond <= flags[2];                         //beq, Z
                3'b001: BranchCond <= !flags[2];                        //bne, ~Z
                3'b100: BranchCond <= flags[3]^flags[0];                //blt, N^V
                3'b101: BranchCond <= !(flags[3]^flags[0]);    //bge, ~(N^V)
                3'b110: BranchCond <= !(flags[1] | flags[2]);  //bltu, ~(Z or C)
                3'b111: BranchCond <= (flags[1] | flags[2]);   //bgeu, Z or C
        default:  BranchCond <=0;
        endcase
assign PCWrite = PCUpdate|(Branch&BranchCond);
//////////////////////////////////////// Branch Control End ////////////////////////////////////

//////////////////////////////////////// ALU Decoder ////////////////////////////////////////////
always@(*)
        case(ALUOp)
                2'b00:    ALUControl <= 4'b0000;
                2'b01:    ALUControl <= 4'b0001; //B - type
                2'b10:    case(funct3)
                                        3'b000:  case({op[5],funct7[5]})
                                                                2'b00:    ALUControl <= 4'b0000; //addi
                                                                2'b01:    ALUControl <= 4'b0000; //addi
                                                                2'b10:    ALUControl <= 4'b0000; //add
                                                                2'b11:    ALUControl <= 4'b0001; //sub
                                                        endcase

                                        3'b001:    ALUControl <= 4'b0110;  //sll: shift left log.
                                                                                                //I and R not
seperated
                                        3'b010:  ALUControl <= 4'b0101;    //slt: set less than
                                                                                                //signed
                                        3'b011:  ALUControl <= 4'b1101;    //sltu: set less than
                                                                                                //unsigned
                                        3'b100:    ALUControl <= 4'b0100;  //xor

                                        3'b101:    case(funct7[5])
                                                                1'b0: ALUControl <= 4'b0111;    //srl: shift
                                                                                                        //right
log.
                                                                1'b1: ALUControl <= 4'b1111;    //sra: shift
                                                                                                        //right
arith.
                                                        endcase
                                        3'b110:  ALUControl <= 4'b0011;    //or
                                        3'b111:  ALUControl <= 4'b0010;    //and
                                endcase
        endcase
//////////////////////////////////////// ALU Decoder End ////////////////////////////////////////

//////////////////////////////////////// Extend Control ////////////////////////////////////////
always@(*)
        case(op)
```

```
                    7'b0000011:     ImmSrc <= 3'b000;       //lw
                    7'b0100011:     ImmSrc <= 3'b001;       //sw
                    7'b0110011:     ImmSrc <= 3'b000;       //R type
                    7'b1100011:     ImmSrc <= 3'b010;       //beq
                    7'b0010011:     ImmSrc <= 3'b000;       //I-ALU
                    7'b1101111:     ImmSrc <= 3'b011;       //jal
                    7'b1100111:     ImmSrc <= 3'b000;       //jalr
                    7'b0010111:     ImmSrc <= 3'b100;       //upper immediate
            endcase
////////////////////////////////////// Extend Control End //////////////////////////////////////////
 endmodule
//begin AdrSrc <= 0; IRWrite <= 1; ALUSrcA <= 2'b00; ALUSrcB <= 2'b10; ALUOp <= 2'b00; ResultSrc <= 2'b10;
PCUpdate <= 1; Branch <=0; MemWrite <= 0; RegWrite <= 0; end
```

mux_2X1.v:

```
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 08/31/2022 06:55:18 AM
// Design Name:
// Module Name: mux2X1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mux_2X1(
input [31:0]A,B,
input Sel,
output [31:0]mux
    );

   assign mux = Sel ? B : A;
endmodule
```

mux_4X1.v:

```
module mux_4X1 ( input [31:0] A,
                 input [31:0] B,
                 input [31:0] C,
                                                  input [31:0] D,
                 input [1:0] Sel,            // input sel used to select between a,b,c,d
                 output [31:0] mux);          // 4-bit output based on input sel
  assign mux = Sel[1] ? (Sel[0] ? D : C) : (Sel[0] ? B : A);
endmodule
```

program_counter.v:

```
module program_counter(
input clk,rst,PCWrite,
input [31:0]PCNext,
output reg [31:0]PC
    );
     always @ (posedge clk  or negedge rst) //next instruction
    begin
    if(!rst)
                        PC <= 0;
    else if (PCWrite)
                        PC <= PCNext;
    else
                        PC <= PC; end
endmodule
```

Top level Test bench

testbench.v:

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Description:       SPI_Slave_Controller Testbench
////////////////////////////////////////////////////////////////////////////////

`include "peripheral_spi_master.v"
`include "peripheral_top.v"
`include "DFFRAM_RTL_2048.v"
`include "data_memory_wrapper.v"

module SPI_Memory_Controller_TB ();

  //=========================Top =======================//

  // parameter related to SPI interface
  parameter SPI_MODE = 0;          // CPOL = 1, CPHA = 1
  parameter CLKS_PER_HALF_BIT = 1;  // 50 MHz
  parameter MAIN_CLK_DELAY = 5;     // 100 MHz
  parameter CS_INACTIVE_CLKS = 3;   // Adds 4 SPI_clk amount of deay between words


  // Parameter related to Wrapper
  parameter DATA_LENGTH = 32;
  parameter ADDRESS_LENGTH = 32;


  // Core Specific
  logic rst_n     = 1'b0;
  logic clk       = 1'b0;
  logic core_select = 1'b1;
```

```
// Peripheral Specific Signals, used only for testbench purpose
logic [31:0] r_Master_TX_Word = 0;
logic r_Master_TX_DV = 1'b0;
logic w_Master_TX_Ready;
//logic r_Master_TX_Count = 1'b1;


// SPI signals fed into our system
logic SPI_Clk;
logic SPI_MOSI;
logic SPI_CS_n;


// SPI_Slave_Controller outputs
logic from_spi_mem_en;
logic from_spi_mem_wr_en;
logic from_spi_mem_rd_en;
logic [ADDRESS_LENGTH-1:0] from_spi_mem_address;
logic [DATA_LENGTH-1:0] from_spi_mem_data_in;
logic [1:0] from_spi_mem_data_length;

logic RX_DV;     // Data Valid pulse (1 clock cycle)
logic RX_Word;               // Word received on MOSI


// Wrapper Specific Signals
logic [DATA_LENGTH-1:0] to_spi_mem_data_out;


// Clock Generators:
always #(MAIN_CLK_DELAY) clk = ~clk;

// Instantiate Peripheral
peripheral_top
#(.SPI_MODE(SPI_MODE),
  .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT),
  .CS_INACTIVE_CLKS(CS_INACTIVE_CLKS),
  .DATA_LENGTH(DATA_LENGTH),
  .ADDRESS_LENGTH(ADDRESS_LENGTH)
 ) peripheral_top_Inst
(
 // Control/Data Signals,
 .i_Rst_L(rst_n),     // Reset
 .i_Clk(clk),         // Clock

 // TX (MOSI) Signals
 .i_TX_Word(r_Master_TX_Word),     // Byte to transmit on MOSI
 .i_TX_DV(r_Master_TX_DV),         // Data Valid Pulse with i_TX_Byte
 .o_TX_Ready(w_Master_TX_Ready),   // Transmit Ready for Byte

 // Core select
  .core_select(core_select),

 // Output Wires for SPI Interface
  .o_SPI_Clk(SPI_Clk),
  .o_SPI_MOSI(SPI_MOSI),
  .o_SPI_CS_n(SPI_CS_n)
 );


// Slave will receive the SPI data, its output will connect to
```

```
// data_memory_wrapper
// Instantiate SPI_Slave_Controller
SPI_Slave_Controller
#(.SPI_MODE(SPI_MODE),
  .DATA_LENGTH(DATA_LENGTH),
  .ADDRESS_LENGTH(ADDRESS_LENGTH)
 ) SPI_Slave_Controller_Inst
(
 // Control/Data Signals,
  .i_rst_n(rst_n),      // Reset
  .i_clk(!clk),         // Clock

  .o_RX_DV(o_RX_DV),  // Data Valid pulse (1 clock cycle)
  .o_RX_Word(o_RX_Word),  // Word received on MOSI

 // SPI Interface
  .i_SPI_Clk(SPI_Clk),
  .i_SPI_MOSI(SPI_MOSI),
  .i_SPI_CS_n(SPI_CS_n),

  // Core select
  .core_select(core_select),

  //Outputs for wrapper
  .o_from_spi_mem_en(from_spi_mem_en),
  .o_from_spi_mem_wr_en(from_spi_mem_wr_en),
  .o_from_spi_mem_rd_en(from_spi_mem_rd_en),
  .o_from_spi_mem_address(from_spi_mem_address),
  .o_from_spi_mem_data_in(from_spi_mem_data_in),
  .o_from_spi_mem_data_length(from_spi_mem_data_length)
 );

          wire [31:0]Adr, ReadData, WriteData;
          wire MemWrite;

// Instantiate Core
          RV32i_core core(
          clk,rst_n,core_select, ReadData, Adr, WriteData, MemWrite
          );



// Instantiate data_memory_wrapper_Ints
data_memory_wrapper
#(.DATA_LENGTH(DATA_LENGTH),
  .ADDRESS_LENGTH(ADDRESS_LENGTH)
 ) data_memory_wrapper_Ints
(.clk(!clk),
 .core_select(core_select),
 //mem2core
 .from_core_mem_en(core_select),
 .from_core_mem_wr_en(MemWrite),
 .from_core_mem_rd_en(core_select),
 .from_core_mem_address(Adr),
 .from_core_mem_data_in(WriteData),
 .from_core_mem_data_length(2'b00),
 .to_core_mem_data_out(ReadData),
 // mem 2 spi
 .from_intf_mem_ctrl_mem_en(from_spi_mem_en),
 .from_intf_mem_ctrl_mem_wr_en(from_spi_mem_wr_en),
 .from_intf_mem_ctrl_mem_rd_en(from_spi_mem_rd_en),
```

```
  .from_intf_mem_ctrl_mem_address(from_spi_mem_address),
  .from_intf_mem_ctrl_mem_data_in(from_spi_mem_data_in),
  .from_intf_mem_ctrl_mem_data_length(from_spi_mem_data_length),
  .to_intf_mem_ctrl_mem_data_out(to_spi_mem_data_out)
 );


 // Sends a single Word from master.  Will drive CS on its own.
 task SendSingleWord(input [31:0] data);
   @(posedge clk);
   r_Master_TX_Word <= data;
   r_Master_TX_DV   <= 1'b1;
   @(posedge clk);
   r_Master_TX_DV <= 1'b0;
   @(posedge clk);
   @(posedge w_Master_TX_Ready);
 endtask // SendSingleByte

 reg [31:0] Mem [0:127];
 initial $readmemh("ins2.mem", Mem);
 integer k;

 initial
   begin
     // Required for EDA Playground
     $dumpfile("dump.vcd");
     $dumpvars;

     rst_n  = 1'b0;
     core_select = 1'b0;
     repeat(2) @(posedge clk);
     rst_n  = 1'b1;
     core_select = 1'b0;

     // Sending the data from file to RAM through SPI
     begin
       for (k=0; k<68; k=k+1) begin
                         SendSingleWord(Mem[k]);
                         $display("Inst No %d : Sent out %h, Received %h", k,Mem[k],to_spi_mem_data_out);
                         #10;
          //core_select = 1'b1;
       end
     end


     //$finish();
   end // initial begin

initial begin
             #49403   core_select = 1;
             end

endmodule // SPI_Slave
```

Implemented SPI Interface

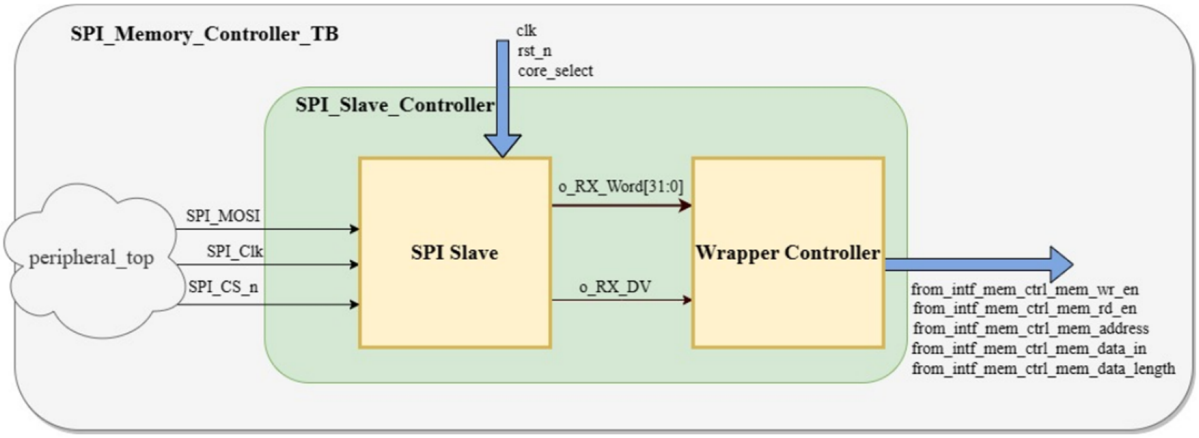
To load instruction programs onto the data memory, we need a data communication interface and associated protocol which will carry out the heavy duty of receiving the instruction data stream from an external peripheral and storing them at the appropriate location. For our project, we were to implement a SPI slave protocol which has 3 input signals

1. SPI_MOSI : 1-bit serial data stream for program instruction
2. SPI_Clk : SPI clock signal to sample the bit from the SPI_MOSI line at appropriate time
3. SPI_CS_n : Chip select pin to control the flow of data

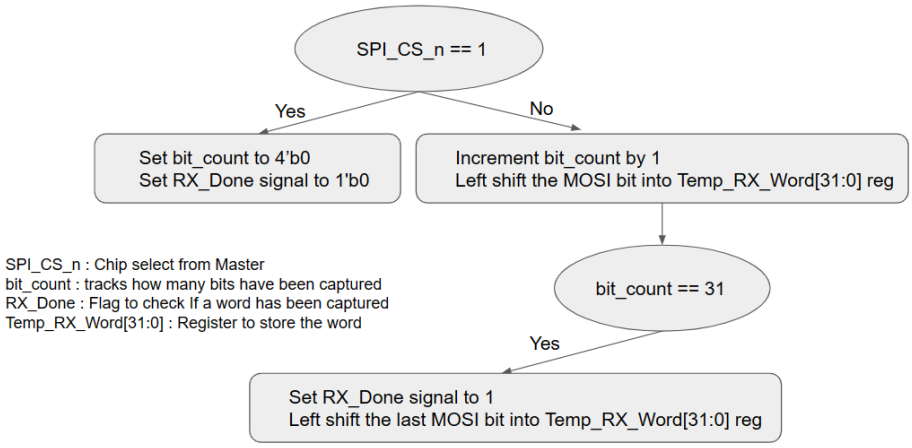
The SPI Slave also takes core clk, rst_n and core_select signals as input for higher level functionality to be controlled and synchronized from the core itself.

After accumulating a 32-bit instruction on the o_RX_Word[31:0] register, SPI Slave module generates an o_RX_DV signal which indicates that a valid word has been received and available on the o_RX_Word register. Then, the received word is sent to the “Wrapper Controller” unit, and consequent signals to load the instruction in the data memory are generated.

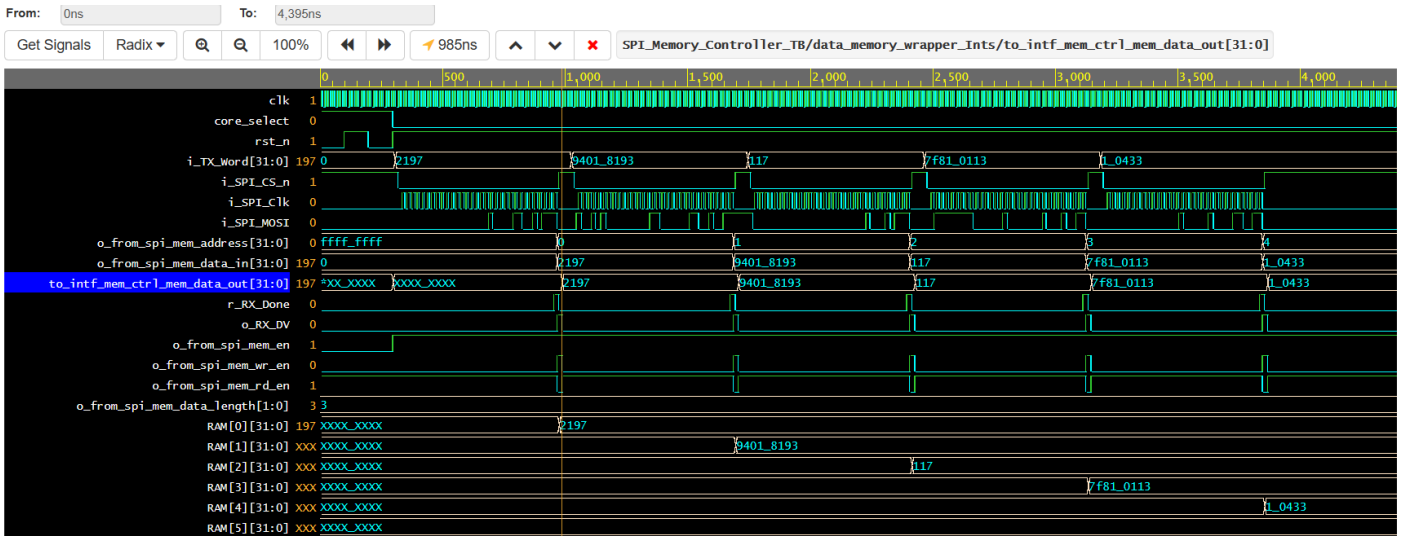
Block Diagram of the Overall System:



At each posedge of SPI_Clk or SPI_CS_n, the following behavioral logic will be implemented inside the “SPI Slave” module to correctly capture a word.



A sample timing diagram to load a few 32-bit instructions in the data memory a.k.a RAM is shown below. It takes around 3600 ns to load 5 instructions in the memory with a core and SPI frequency 100MHz and 50MHz respectively. Thus, on average, it takes 36 clock cycles per instruction.



RTL Codes

peripheral_spi_master.v:

```
////////////////////////////////////////////////////////////////////////////////
// Description: SPI (Serial Peripheral Interface) Master
//              Creates master based on input configuration.
//              Sends a 8-byet word one bit at a time on MOSI
//              Will also receive byte data one bit at a time on MISO.
//              Any data on input byte will be shipped out on MOSI.
//
//              To kick-off a transaction, the user must pulse i_TX_DV.
//              This module supports multi-byte transmissions by pulsing
//              i_TX_DV and loading up i_TX_Byte when o_TX_Ready is high.
//
```

```
//              This module is only responsible for controlling Clk, MOSI,
//              and MISO.  If the SPI peripheral requires a chip-select,
//              this must be done at a higher level.
//
// Note:        i_Clk must be at least 2x faster than i_SPI_Clk
//
// Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
//              Can be configured in one of 4 modes:
//              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (CPHA)
//               0   |             0             |        0
//               1   |             0             |        1
//               2   |             1             |        0
//               3   |             1             |        1
//              More: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Mode_numbers
//              CLKS_PER_HALF_BIT - Sets frequency of o_SPI_Clk.  o_SPI_Clk is
//              derived from i_Clk.  Set to integer number of clocks for each
//              half-bit of SPI data.  E.g. 100 MHz i_Clk, CLKS_PER_HALF_BIT = 2
//              would create o_SPI_CLK of 25 MHz.  Must be >= 2
//
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
// Module Name: peripheral_spi_master
// Project Name: EEE 468
// Dept of EEE, BUET
// Team 01
// Developed by: Mrinmoy & Co.
////////////////////////////////////////////////////////////////////////////////

module SPI_Master
  #(parameter SPI_MODE = 0,
    parameter CLKS_PER_HALF_BIT = 2)
  (
   // Control/Data Signals,
   input        i_Rst_L,     // FPGA Reset
   input        i_Clk,       // FPGA Clock

   // TX (MOSI) Signals
   input [31:0]   i_TX_Word,        // Byte to transmit on MOSI
   input          i_TX_DV,          // Data Valid Pulse with i_TX_Byte
   output reg     o_TX_Ready,       // Transmit Ready for next byte


   // SPI Interface
   output reg o_SPI_Clk,
   output reg o_SPI_MOSI
   );

  // SPI Interface (All Runs at SPI Clock Domain)
  wire w_CPOL;     // Clock polarity
  wire w_CPHA;     // Clock phase

  reg [$clog2(CLKS_PER_HALF_BIT*2)-1:0] r_SPI_Clk_Count;
  reg r_SPI_Clk;


  reg [19:0] r_SPI_Clk_Edges;
  reg r_Leading_Edge;
  reg r_Trailing_Edge;

  reg       r_TX_DV;
  reg [31:0] r_TX_Word;
```

```
reg [4:0] r_TX_Bit_Count;

// CPOL: Clock Polarity
// CPOL=0 means clock idles at 0, leading edge is rising edge.
// CPOL=1 means clock idles at 1, leading edge is falling edge.
assign w_CPOL  = (SPI_MODE == 2) | (SPI_MODE == 3);

// CPHA: Clock Phase
// CPHA=0 means the "out" side changes the data on trailing edge of clock
//              the "in" side captures data on leading edge of clock
// CPHA=1 means the "out" side changes the data on leading edge of clock
//              the "in" side captures data on the trailing edge of clock
assign w_CPHA  = (SPI_MODE == 1) | (SPI_MODE == 3);



// Purpose: Generate SPI Clock correct number of times when DV pulse comes
always @(posedge i_Clk or negedge i_Rst_L)
begin
  if (~i_Rst_L)
  begin
    o_TX_Ready      <= 1'b0;
    r_SPI_Clk_Edges <= 0;
    r_Leading_Edge  <= 1'b0;
    r_Trailing_Edge <= 1'b0;
    r_SPI_Clk       <= w_CPOL; // assign default state to idle state
    r_SPI_Clk_Count <= 0;
  end
  else
  begin

    // Default assignments
    r_Leading_Edge  <= 1'b0;
    r_Trailing_Edge <= 1'b0;

    if (i_TX_DV)
    begin
      o_TX_Ready      <= 1'b0;
      r_SPI_Clk_Edges = 64;  // Total # edges in one word ALWAYS 64
    end
    else if (r_SPI_Clk_Edges > 0)
    begin
      o_TX_Ready <= 1'b0;

      if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT*2-1)
      begin
        r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
        r_Trailing_Edge <= 1'b1;
        r_SPI_Clk_Count <= 0;
        r_SPI_Clk       <= ~r_SPI_Clk;
      end
      else if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT-1)
      begin
        r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
        r_Leading_Edge  <= 1'b1;
        r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
        r_SPI_Clk       <= ~r_SPI_Clk;
      end
      else
      begin
```

```
        r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
      end
    end
    else
    begin
      o_TX_Ready <= 1'b1;
    end


  end // else: !if(~i_Rst_L)
end // always @ (posedge i_Clk or negedge i_Rst_L)




// Purpose: Register i_TX_Word when Data Valid is pulsed.
// Keeps local storage of word in case higher level module changes the data
always @(posedge i_Clk or negedge i_Rst_L)
begin
  if (~i_Rst_L)
  begin
    r_TX_Word <= 32'h00;
    r_TX_DV   <= 1'b0;
  end
  else
    begin
      r_TX_DV <= i_TX_DV; // 1 clock cycle delay
      if (i_TX_DV)
      begin
        r_TX_Word <= i_TX_Word;
      end
    end // else: !if(~i_Rst_L)
end // always @ (posedge i_Clk or negedge i_Rst_L)




// Purpose: Generate MOSI data
// Works with both CPHA=0 and CPHA=1
always @(posedge i_Clk or negedge i_Rst_L)
begin
  if (~i_Rst_L)
  begin
    o_SPI_MOSI     <= 1'b0;
    r_TX_Bit_Count <= 5'b11111; // send MSb first
  end
  else
  begin
    // If ready is high, reset bit counts to default
    if (o_TX_Ready)
    begin
      r_TX_Bit_Count <= 5'b11111;
    end
    // Catch the case where we start transaction and CPHA = 0
    else if (r_TX_DV & ~w_CPHA)
    begin
      o_SPI_MOSI     <= i_TX_Word[5'b11111];
      r_TX_Bit_Count <= 5'b11110;
    end
```

```
    else if ((r_Leading_Edge & w_CPHA) | (r_Trailing_Edge & ~w_CPHA))
    begin
      r_TX_Bit_Count <= r_TX_Bit_Count - 1;
      o_SPI_MOSI     <= i_TX_Word[r_TX_Bit_Count];
    end
  end
end




 // Purpose: Add clock delay to signals for alignment.
 always @(posedge i_Clk or negedge i_Rst_L)
 begin
   if (~i_Rst_L)
   begin
     o_SPI_Clk  <= w_CPOL;
   end
   else
     begin
       o_SPI_Clk <= r_SPI_Clk;
     end // else: !if(~i_Rst_L)
 end // always @ (posedge i_Clk or negedge i_Rst_L)


endmodule // SPI_Master
```

Peripheral_top.v:

```
////////////////////////////////////////////////////////////////////////////////
// Description: Top level module for SPI (Serial Peripheral Interface) Memory controller
//                                      With single chip-select (AKA Slave Select) capability
//
//              Supports arbitrary length byte transfers.
//
//              Instantiates a SPI Master and adds single CS.
//              If multiple CS signals are needed, will need to use different
//              module, OR multiplex the CS from this at a higher level.
//
//              Instantiates a SPI Slave which will reveive the signals from Master and drive the
//                                      data_memory_wrapper declared in the testbench
//
//              If multiple CS signals are needed, will need to use different
//              module, OR multiplex the CS from this at a higher level.
//
// Note:        i_Clk must be at least 2x faster than i_SPI_Clk
//
// Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
//              Can be configured in one of 4 modes:
//              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (CPHA)
//               0   |             0             |       0
//               1   |             0             |       1
//               2   |             1             |       0
//               3   |             1             |       1
//                                      Mode 0 is selected for our purpose
//
//              CLKS_PER_HALF_BIT - Sets frequency of o_SPI_Clk.  o_SPI_Clk is
//              derived from i_Clk.  Set to integer number of clocks for each
//              half-bit of SPI data.  E.g. 100 MHz i_Clk, CLKS_PER_HALF_BIT = 2
//              would create o_SPI_CLK of 25 MHz.  Must be >= 2
```

```
//
//
//             CS_INACTIVE_CLKS - Sets the amount of time in clock cycles to
//             hold the state of Chip-Selct high (inactive) before next
//             command is allowed on the line.  Useful if chip requires some
//             time when CS is high between trasnfers.
////////////////////////////////////////////////////////////////////////////////

module peripheral_top
  #(parameter SPI_MODE = 0,
    parameter CLKS_PER_HALF_BIT = 2,
    parameter CS_INACTIVE_CLKS = 1,
    parameter DATA_LENGTH = 32,
    parameter ADDRESS_LENGTH = 32)
  (
   // Control/Data Signals,
   input        i_Rst_L,     // Reset
   input        i_Clk,       // Clock

   // TX (MOSI) Signals
   input [31:0]  i_TX_Word,       // Byte to transmit on MOSI
   input         i_TX_DV,         // Data Valid Pulse with i_TX_Byte
   output        o_TX_Ready,      // Transmit Ready for next byte


   // Core select
   input core_select,

   // Output Wires for SPI Interface
   output wire o_SPI_Clk,
   output wire o_SPI_MOSI,
   output wire o_SPI_CS_n
   );



  /////////////// Master definition and its control////////////////////////
  // Output wires from master which will be connected to Slave
  //
  //


  localparam IDLE        = 2'b00;
  localparam TRANSFER    = 2'b01;
  localparam CS_INACTIVE = 2'b10;

  reg [1:0] r_SM_CS;
  reg r_CS_n;
  reg [$clog2(CS_INACTIVE_CLKS)-1:0] r_CS_Inactive_Count;
  wire w_Master_Ready;

  // Instantiate Master
  SPI_Master
    #(.SPI_MODE(SPI_MODE),
      .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT)
      ) SPI_Master_Inst
   (
   // Control/Data Signals,
   .i_Rst_L(i_Rst_L),     // Reset
   .i_Clk(i_Clk),         // Clock
```

```
 // TX (MOSI) Signals
 .i_TX_Word(i_TX_Word),         // Byte to transmit
 .i_TX_DV(i_TX_DV),             // Data Valid Pulse
 .o_TX_Ready(w_Master_Ready),   // Transmit Ready for Byte

 // SPI Interface
 .o_SPI_Clk(o_SPI_Clk),
 .o_SPI_MOSI(o_SPI_MOSI)
 );

// Purpose: Control CS line using State Machine
always @(posedge i_Clk or negedge i_Rst_L)
begin
  if (~i_Rst_L)
  begin
    r_SM_CS <= IDLE;
    r_CS_n  <= 1'b1;   // Resets to high
    r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
  end
  else
  begin

    case (r_SM_CS)
    IDLE:
      begin
        if (r_CS_n & i_TX_DV) // Start of transmission
        begin
          r_CS_n     <= 1'b0;       // Drive CS low
          r_SM_CS    <= TRANSFER;   // Transfer bytes
        end
      end

    TRANSFER:
      begin
        // Wait until SPI is done transferring do next thing
        if (w_Master_Ready)
          begin
            r_CS_n  <= 1'b1; // we done, so set CS high
            r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
            r_SM_CS             <= CS_INACTIVE;
           end // if (w_Master_Ready)
      end // case: TRANSFER

    CS_INACTIVE:
      begin
        if (r_CS_Inactive_Count > 0)
        begin
          r_CS_Inactive_Count <= r_CS_Inactive_Count - 1'b1;
        end
        else
        begin
          r_SM_CS <= IDLE;
        end
      end

    default:
      begin
        r_CS_n  <= 1'b1; // we done, so set CS high
        r_SM_CS <= IDLE;
      end
    endcase // case (r_SM_CS)
```

```
   end
  end // always @ (posedge i_Clk or negedge i_Rst_L)

  assign o_SPI_CS_n = r_CS_n;
  assign o_TX_Ready  = (r_SM_CS == IDLE) & ~i_TX_DV;


endmodule // SPI_Master_With_Single_CS
```

Spi_slave_controller.v:

```
//////////////////////////////////////////////////////////////////////////////////
// Description: SPI (Serial Peripheral Interface) Slave
//              Creates slave based on input configuration.
//              Receives a 8-byte word one bit at a time on MOSI

//                                          MISO is deactivated**

//              Supports multiple bytes per transaction when CS_n is kept
//              low during the transaction.
//
// Note:        i_Clk must be at least 4x faster than i_SPI_Clk
//              MISO is tri-stated when not communicating.  Allows for multiple
//              SPI Slaves on the same interface.
//
// Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
//              Can be configured in one of 4 modes:
//              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (CPHA)
//               0   |             0             |       0
//               1   |             0             |       1
//               2   |             1             |       0
//               3   |             1             |       1
//              More info: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Mode_numbers
//////////////////////////////////////////////////////////////////////////////////

module SPI_Slave_Controller
  #(parameter SPI_MODE = 0,
    parameter DATA_LENGTH = 32,
    parameter ADDRESS_LENGTH = 32)
  (
   // Control/Data Signals,
   input            i_rst_n,    // Reset, active low
   input            i_clk,      // Clock
   output reg       o_RX_DV,    // Data Valid pulse (1 clock cycle)
   output reg [31:0] o_RX_Word,  // Word received on MOSI

   // SPI Interface
   input      i_SPI_Clk,
   input      i_SPI_MOSI,
   input      i_SPI_CS_n,       // active low

   // Core select
   input core_select,

   // Output signals for wrapper
   output wire o_from_spi_mem_en,
   output wire o_from_spi_mem_wr_en, // EDITED BY DIGANTA, (REG > wire)
   output wire o_from_spi_mem_rd_en, // EDITED BY DIGANTA, (REG > wire)
   output reg [ADDRESS_LENGTH-1:0] o_from_spi_mem_address,
   output wire [DATA_LENGTH-1:0] o_from_spi_mem_data_in,
```

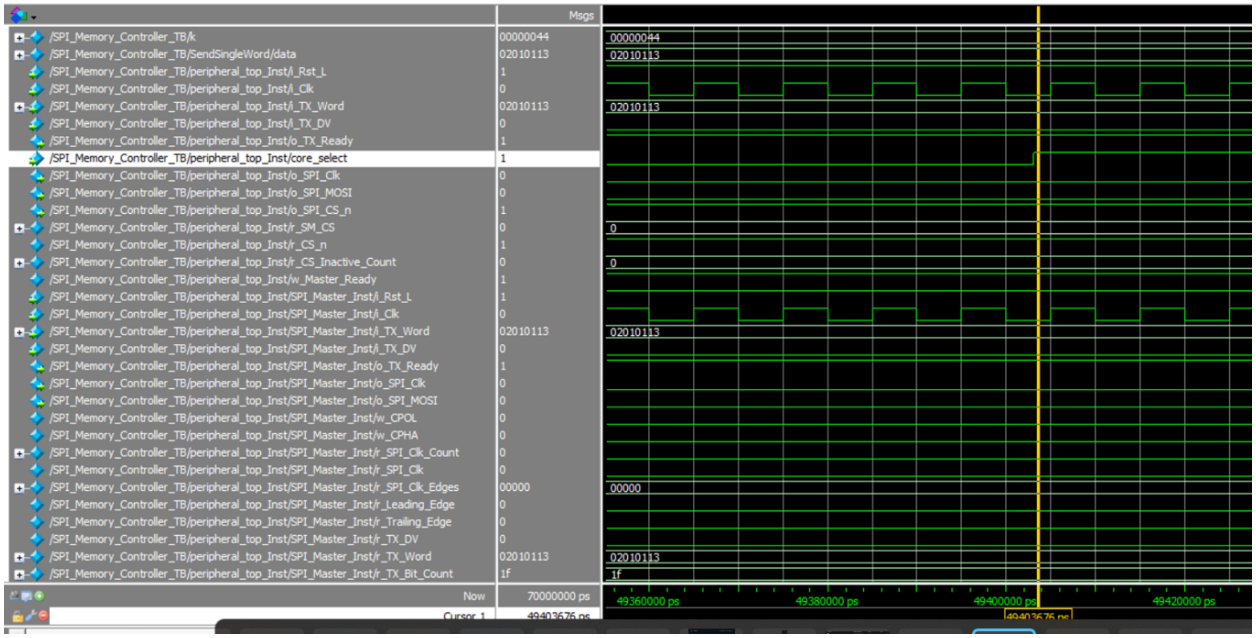
```
 output wire [1:0] o_from_spi_mem_data_length
 );


// SPI Interface (All Runs at SPI Clock Domain)
wire w_CPOL;     // Clock polarity
wire w_CPHA;     // Clock phase
wire w_SPI_Clk;  // Inverted/non-inverted depending on settings

reg [4:0] r_RX_Bit_Count;
reg [31:0] r_RX_Word;
reg [31:0] r_Temp_RX_Word;
reg r_RX_Done, r2_RX_Done, r3_RX_Done;

// CPOL: Clock Polarity
// CPOL=0 means clock idles at 0, leading edge is rising edge.
// CPOL=1 means clock idles at 1, leading edge is falling edge.
assign w_CPOL  = (SPI_MODE == 2) | (SPI_MODE == 3);

// CPHA: Clock Phase
// CPHA=0 means the "out" side changes the data on trailing edge of clock
//             the "in" side captures data on leading edge of clock
// CPHA=1 means the "out" side changes the data on leading edge of clock
//             the "in" side captures data on the trailing edge of clock
assign w_CPHA  = (SPI_MODE == 1) | (SPI_MODE == 3);
assign w_SPI_Clk = w_CPHA ? ~i_SPI_Clk : i_SPI_Clk;


// Purpose: Recover SPI Word in SPI Clock Domain
// Samples MOSI line on correct edge of SPI Clock
always @(posedge w_SPI_Clk or posedge i_SPI_CS_n)
begin
  if (i_SPI_CS_n)
  begin
    r_RX_Bit_Count <= 0;
    r_RX_Done      <= 1'b0;
  end
  else
  begin
    r_RX_Bit_Count <= r_RX_Bit_Count + 1;

    // Receive in LSB, shift up to MSB
    r_Temp_RX_Word <= {r_Temp_RX_Word[30:0], i_SPI_MOSI};

    if (r_RX_Bit_Count == 5'b11111)
    begin
      r_RX_Done <= 1'b1;
      r_RX_Word <= {r_Temp_RX_Word[30:0], i_SPI_MOSI};
    end
    else if (r_RX_Bit_Count == 5'b00000)
    begin
      r_RX_Done <= 1'b0;
    end

  end // else: !if(i_SPI_CS_n)
end // always @ (posedge w_SPI_Clk or posedge i_SPI_CS_n)



// Purpose: Cross from SPI Clock Domain to main core clock domain
// Assert o_RX_DV for 2 clock cycle when o_RX_Word has valid data.
```

```
always @(posedge i_clk or negedge i_rst_n)
begin
  if (~i_rst_n)
  begin
    r2_RX_Done <= 1'b0;
    r3_RX_Done <= 1'b0;
    o_RX_DV    <= 1'b0;
    o_RX_Word  <= 32'h00;
  end
  else
  begin
    // Here is where clock domains are crossed.
    // This will require timing constraint created, can set up long path.
    r2_RX_Done <= r_RX_Done;
    r3_RX_Done <= r2_RX_Done;

    // setting o_RX_DV to 1'b1 for 2 Clock cycle
    if ((r2_RX_Done == 1'b0 && r_RX_Done == 1'b1) || (r3_RX_Done == 1'b0 && r2_RX_Done == 1'b1))
    begin
      o_RX_DV   <= 1'b1;  // Pulse Data Valid 2 clock cycle
      o_RX_Word <= r_RX_Word;
    end
    else
    begin
      o_RX_DV <= 1'b0;
    end
  end // else: !if(~i_Rst_L)
end // always @ (posedge i_Bus_Clk)



//=============== Wrapper Control and data signals ============//

 //write and read enable follows o_RX_DV
 assign o_from_spi_mem_en = ~core_select;
 assign o_from_spi_mem_wr_en = core_select ? 1'b0 : o_RX_DV;
 assign o_from_spi_mem_rd_en = core_select ? 1'b1 : ~o_RX_DV;

 // Receiving data from SPI
 assign o_from_spi_mem_data_in = o_RX_Word;
 assign o_from_spi_mem_data_length = 2'b11;


// address will reset only when i_rst_n is imposed
always @(negedge i_rst_n)
  begin
    if (~i_rst_n)
    begin
      o_from_spi_mem_address <= 32'hffffffff;
    end
  end

// If core_select is 1'b1, address will retain its formal value, otherwise will increment
// after receiving a word
always @(posedge o_RX_DV)
  begin
    if(~core_select)
      begin
            o_from_spi_mem_address <= o_from_spi_mem_address + 1;
      end
```

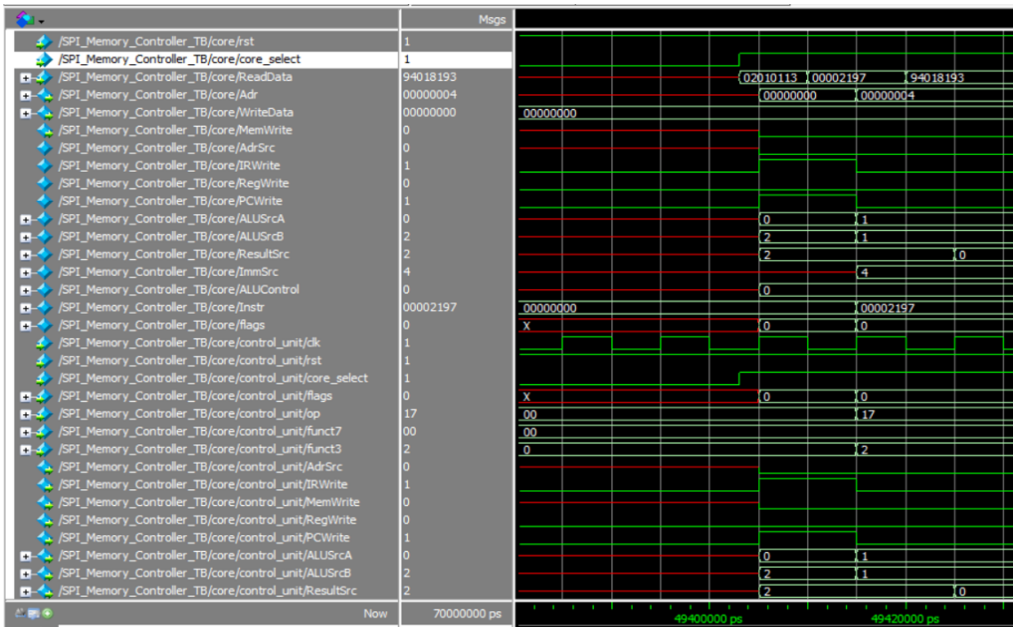
```
    end

endmodule // SPI_Slave
```

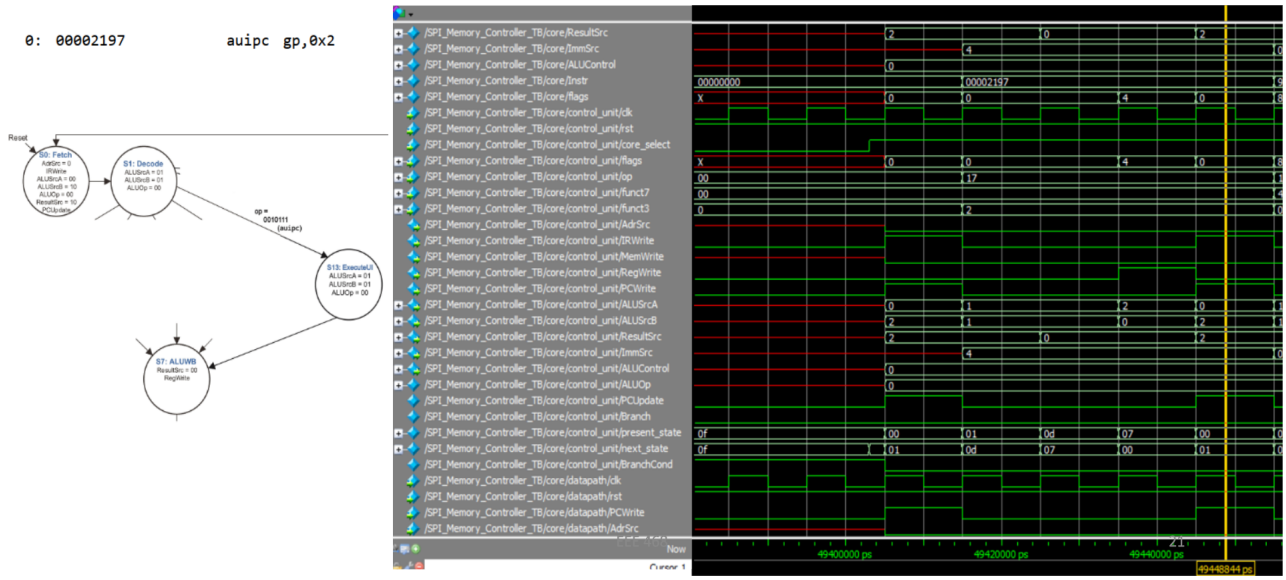
Final Result: Output Waveforms

In the following pictures, some simulation results of our code is shown. In our first result, we can see the signal core select getting HIGH from LOW near the selected time. This denotes the end of instruction loading via the SPI module and activation of the core, starting our program execution stage with the next positive clock edge.



In the next slide, it has been demonstrated that the core is inactive when core select signal is LOW. Note that it takes almost **49.4 microseconds** for the SPI module to load the instructions and 4us to execute.

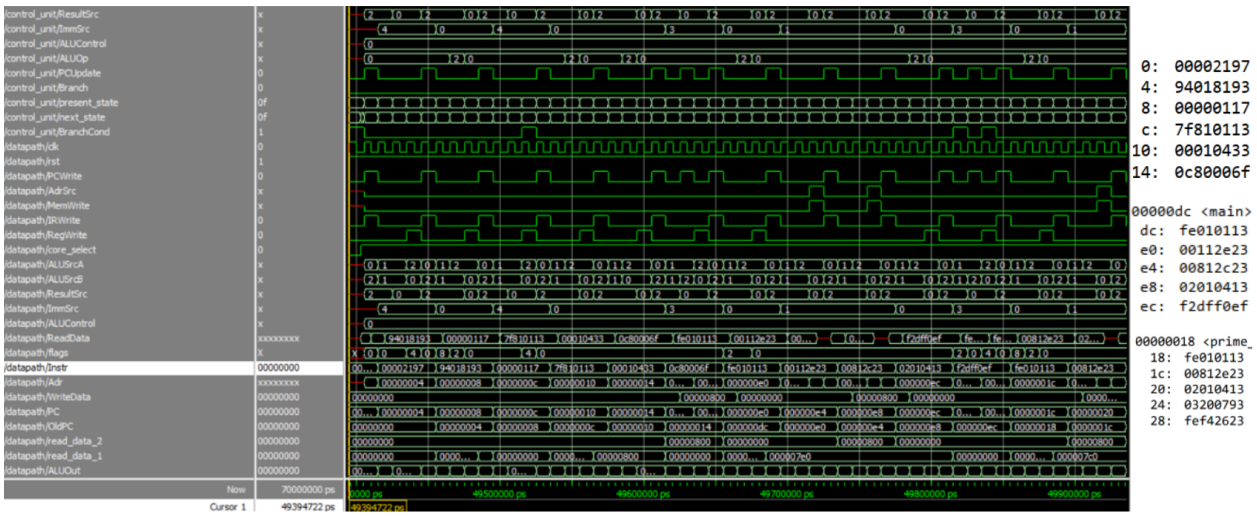




Here, we can see the states of our control unit FSM for executing the first instruction, which is **auipc gp, 2**. It is a U type instruction which attends 12 0 bits after the immediate 0x2 and adds this value to the PC,, and then stores the value in the register gp. In our FSM diagram, we can see there are 4 stages for this instruction, including the fetch stage.

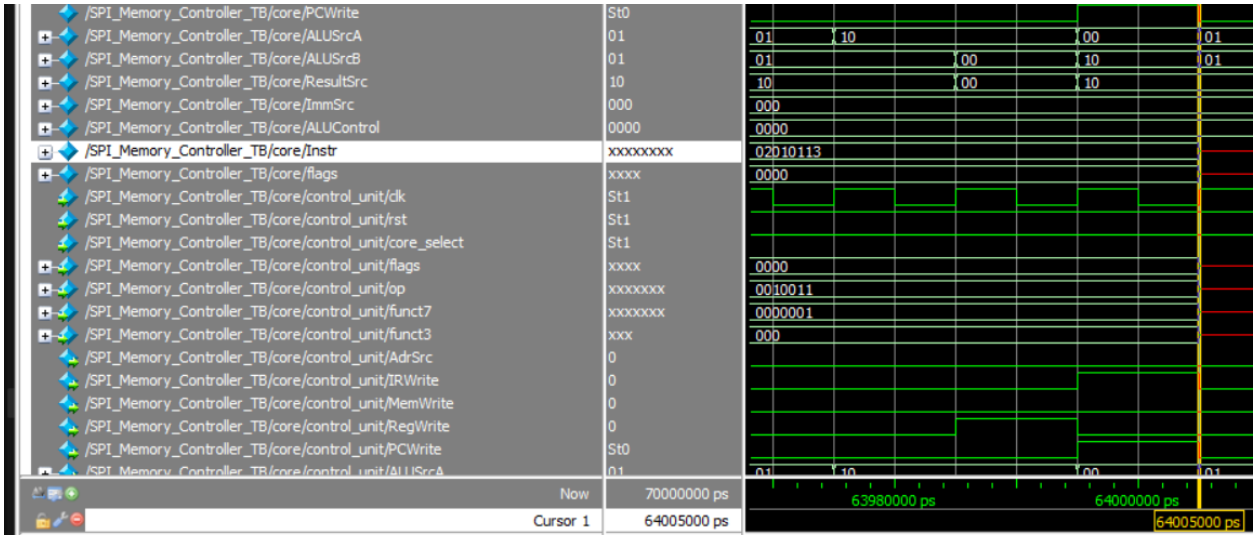
We may verify the state 13 or ExecuteUI stage, where ALUSrcA signal is supposed to be 1, meaning that the previous value of PC is being taken as input operand 1; and ALUSrcB is also 1, which means the second operand in the ALU is being taken from the Immediate Extender, which attends the 12 0 bits after 2. ALUop is 0 for addition in the ALU module.

Including the fetch stage, it takes 4 clock cycles for this instruction to execute, and the FSM returns to state 0 to fetch the next instruction.



Here,noting the instr signal, we can see that the instructions are being executed sequentially. There are some branching taking place, the first jump being to the <main> section of the code,

which eventually jumps to the <prime_number> section. We have included the encoded instructions in the sequence they are executed.



We can see that the last instruction executed is 02010113 around the time 64 microseconds. Thus, it takes almost **14.6 micro-seconds** for the core to execute the whole program.

Output - Register File and Memory:

DFFRAM memory data after loading the instructions through SPI:

```
67  02010113 01812403 01c12083 00078513
63  00000013 00078613 fec42783 fea42623
59  f2dff0ef 02010413 00812c23 00112e23
55  fe010113 00008067 02010113 01c12403
51  00078513 fe442783 00000013 0080006f
47  f8e7dee3 fec42783 fe842703 fef42423
43  00178793 fe842783 02f70063 00100793
39  fe442703 fc1ff06f fef42623 40f707b3
35  fe842783 fec42703 0180006f fe042223
31  0007d663 40f707b3 fe842783 fec42703
27  0300006f fef42223 00100793 00f71863
23  fe842783 fec42703 0600006f fef42423
19  00200793 fef42223 00100793 00f71663
15  00100793 fec42703 00078863 fec42783
11  fe042223 fef42623 03200793 02010413
 7  00812e23 fe010113 0c80006f 00010433
 3  7f810113 00000117 94018193 00002197
```

(Memory address 0 is on the right in the last row. The first instruction is loaded here)

Verification from TB after SPI slave received data:

```
# Inst No            0 : Sent out 00002197, Received 00002197
# Inst No            1 : Sent out 94018193, Received 94018193
# Inst No            2 : Sent out 00000117, Received 00000117
# Inst No            3 : Sent out 7f810113, Received 7f810113
# Inst No            4 : Sent out 00010433, Received 00010433
# Inst No            5 : Sent out 0c80006f, Received 0c80006f
# Inst No            6 : Sent out fe010113, Received fe010113
# Inst No            7 : Sent out 00812e23, Received 00812e23
```

Below, Register File contents after the execution of the program is shown.

0	0	0
2	2048	6464
4	0	0
6	0	0
8	2048	0
10	1	0
12	1	0
14	1	1
16	0	0
18	0	0
20	0	0
22	0	0
24	0	0
26	0	0
28	0	0
30	0	0

Return value stored in R12

- 1 = non-prime
- 0 = prime

Test done on a = 50, Result shows non-prime, thus, R12 shows 1.

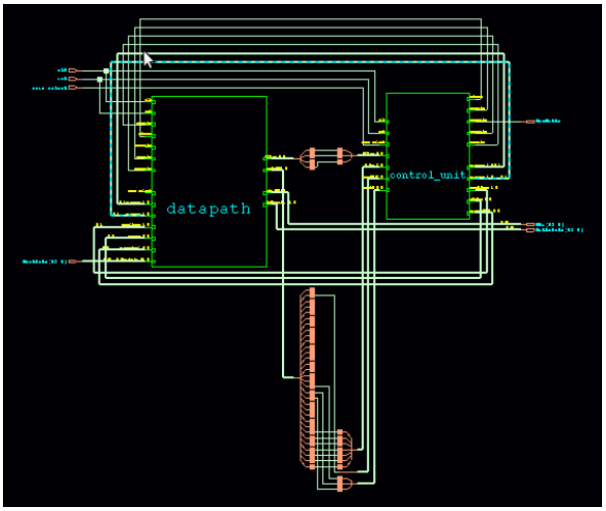
Final results: Synthesis and Physical Design

TCL file used in synthesis

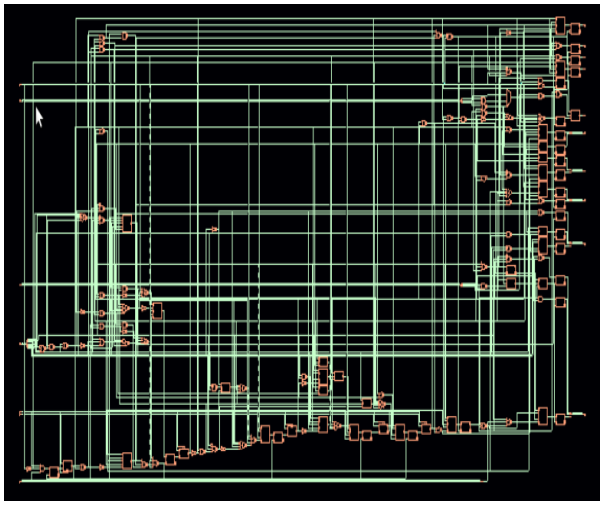
```
##rtl.tcl file adapted from http://ece.colorado.edu/~ecen5007/cadence/
##this tells the compiler where to look or the libraries

set_attribute lib_search_path /home/vlsi12/cds_digital/labtest_practice/exp_3/library

## This defines the libraries to use

set_attribute library {slow_vdd1v0_basicCells.lib fast_vdd1v0_basicCells.lib}

##This must point to your VHDL/verilog file
##read_hdl ../HDL/accu.v

read_hdl rtl_codes/vlsi_core/A.v
read_hdl rtl_codes/vlsi_core/ALU.v
read_hdl rtl_codes/vlsi_core/ALU_out_reg.v
read_hdl rtl_codes/vlsi_core/control.v
read_hdl rtl_codes/vlsi_core/Datapath.v
read_hdl -sv rtl_codes/vlsi_core/Data_register.v
read_hdl rtl_codes/vlsi_core/Extend.v
read_hdl rtl_codes/vlsi_core/Instruction_register.v
read_hdl rtl_codes/vlsi_core/mux_2X1.v
read_hdl rtl_codes/vlsi_core/mux_4X1.v
read_hdl rtl_codes/vlsi_core/program_counter.v
read_hdl rtl_codes/vlsi_core/Registers.v
read_hdl rtl_codes/vlsi_core/RV32i_core.v

## This builts the general block
elaborate

##this allows you to define a clock and the maximum allowable delays
## READ MORE ABOUT THIS SO THAT YOU CAN PROPERLY CREATE A TIMING FILE
#set clock [define_clock -period 300 -name clk]
#external delay -input 300 -edge rise clk
#external delay -output 2000 -edge rise p1

##This synthesizes your code
synthesize -to_mapped

## This writes all your files
## change the tst to the name of your top level verilog
## CHANGE THIS LINE: CHANGE THE "accu" PART REMEMBER THIS
## FILENAME YOU WILL NEED IT WHEN SETTING UP THE PLACE & ROUTE
write -mapped > synth_codes/core_synth.v

## THESE FILES ARE NOT REQUIRED, THE SDC FILE IS A TIMING FILE
write_script > script

##write sdc > tst.sdc
```

Synthesized Gate Level Diagrams

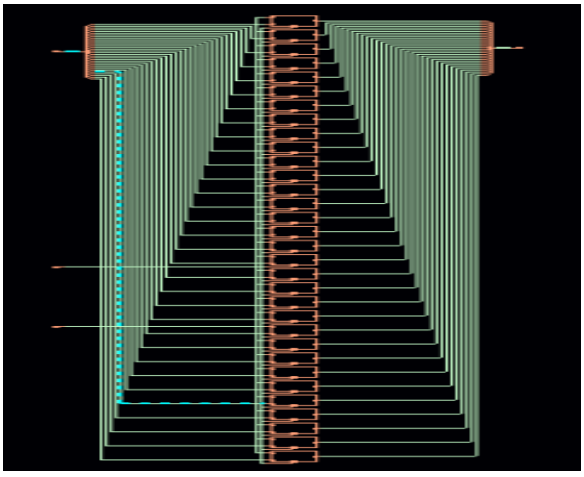
Out of the 17 modules, only 4 are shown here for convenience.
Synthesized Gate Level Verilog File is not attached because that code is over 180 pages long.



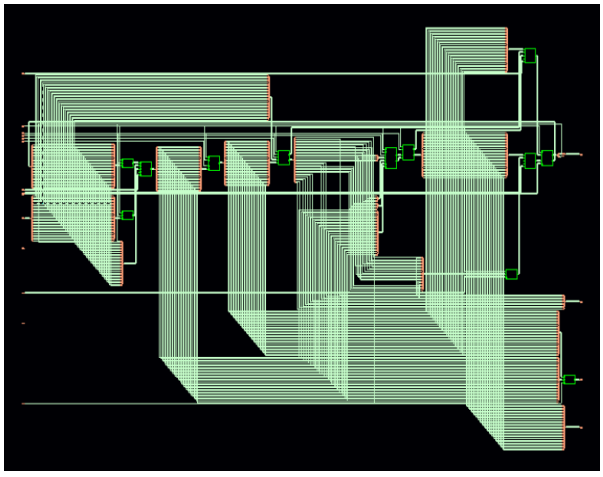
Core Top Module



Control Unit



Data register



Datapath

Physical Design

We could not perform the Physical Design Steps, Clock Tree Synthesis, Power Routing and the other steps due to server unavailability.

Acknowledgements

We would like to express our appreciation to Uday Hasan, Wahid Sadiq, and Adnan Osman of Neural Semiconductor Limited for their helpful suggestions and recommendations. They scheduled several meetings to discuss every detail and problem we encountered while implementing this project, and their intuitive and brilliant guidelines helped us to successfully complete this project. We would also like to express our gratitude to Dr. Prof. A.B.M. Harun-Ur-Rashid and Mumtahina Islam Sukanya for their constant guidance and support.

Conclusion

This project gave us a flavor of large scale circuit design where millions and billions of transistors work together to serve a purpose. From RTL design to physical design, the whole process has been introduced to us in an interesting manner.

References

1. Digital Design and Computer Architecture, RISC-V Edition,
   Sarah L. Harris, David Harris
2. The RISC-V Instruction Set Manual Volume I: User-Level ISA