Project Proposal for VLSI LAB, EEE, BUET

Table of Contents

Objective

The main objective of this lab project is to give the students a flavor of digital design, design verification, physical design of a chip, and introduce them with processor design using Verilog.

Students will design (write RTL) and simulate a 32-bit RISC-V (rv32i compatible) core. This core will be used to execute instructions written in C. The code written in C needs to be converted into 32-bit hexadecimal format using C-compiler and then will be loaded into the instruction memory of the processor during verification. Further, they will synthesize and perform the physical design of Hardware Description of their RISC-V processor. For verification, synthesis and physical design students may follow the examples in the lab sheet.

Specification

1. Core must be compatible with rv32i instruction set.
2. Implement base ISA spec. **Reference:** RISC-V 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, I2C, APB, UART, Wishbone, etc. interface**) to load the instruction data to the memory.

Group	Interface	Spec
1, 6	SPI	https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
2, 7	I2C	https://en.wikipedia.org/wiki/I%C2%B2C
3, 8	APB	https://verificationforall.wordpress.com/apb-protocol/
4, 9	UART	https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
5, 10	Wishbone	https://en.wikipedia.org/wiki/Wishbone_(computer_bus)

Architecture & HDL of Memory and Memory wrapper is given below.

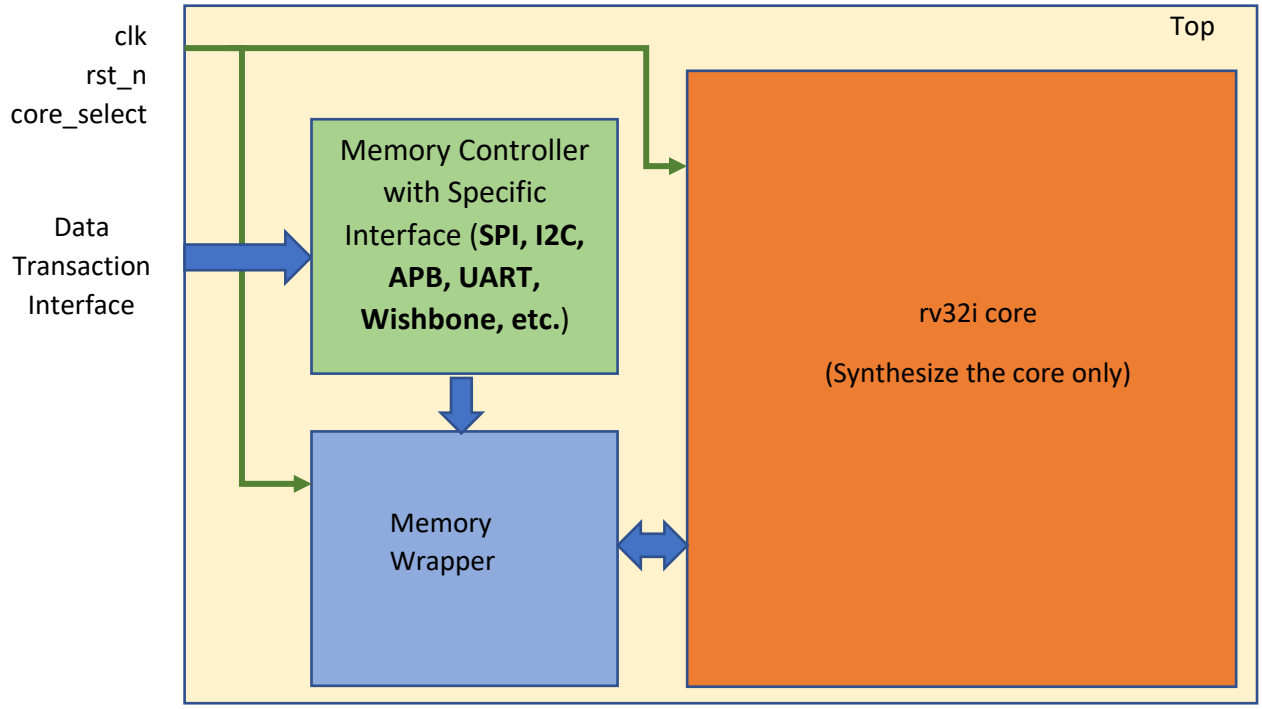
5. Functionality of **core_select** pin

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. **Introducing pipelining stages is a plus. Implementation of Harvard Architecture for separate instruction and data memory is a plus.**

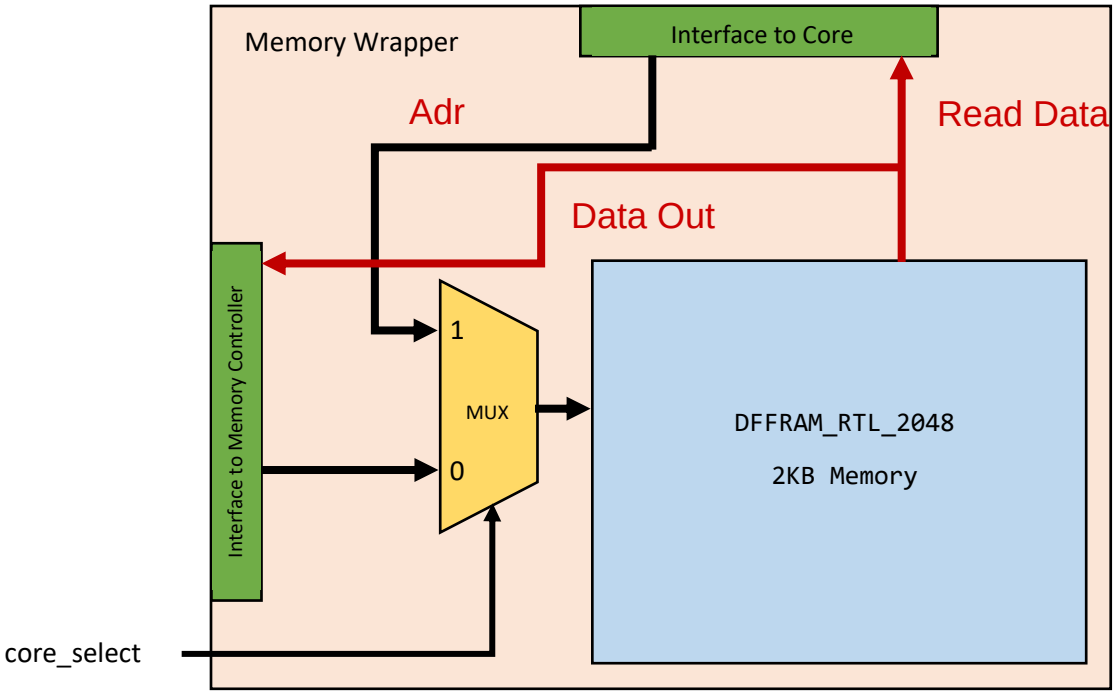
Architecture

RV32I Core



Memory Wrapper

Wrapper Architecture



Wrapper RTL

```
module data_memory_wrapper #( parameter DATA_LENGTH, ADDRESS_LENGTH) (clk, core_select,
from_core_mem_en, from_core_mem_wr_en, from_core_mem_rd_en, from_core_mem_address,
from_core_mem_data_in, from_core_mem_data_length, to_core_mem_data_out,
from_intf_mem_ctrl_mem_en, from_intf_mem_ctrl_mem_wr_en, from_intf_mem_ctrl_mem_rd_en,
from_intf_mem_ctrl_mem_address, from_intf_mem_ctrl_mem_data_in,
from_intf_mem_ctrl_mem_data_length, to_intf_mem_ctrl_mem_data_out);

    input clk;
    input core_select;
    //core
    input from_core_mem_en;
    input from_core_mem_wr_en;
    input from_core_mem_rd_en;
    input [DATA_LENGTH-1:0] from_core_mem_address;
    input [DATA_LENGTH-1:0] from_core_mem_data_in;
    input [1:0] from_core_mem_data_length;
    output wire [DATA_LENGTH-1:0] to_core_mem_data_out;
    //top
    input from_intf_mem_ctrl_mem_en;
    input from_intf_mem_ctrl_mem_wr_en;
    input from_intf_mem_ctrl_mem_rd_en;
    input [DATA_LENGTH-1:0] from_intf_mem_ctrl_mem_address;
    input [DATA_LENGTH-1:0] from_intf_mem_ctrl_mem_data_in;
    input [1:0] from_intf_mem_ctrl_mem_data_length;
    output wire [DATA_LENGTH-1:0] to_intf_mem_ctrl_mem_data_out;

    wire mem_wr_en, mem_en, mem_rd_en;
    wire [DATA_LENGTH-1:0] mem_data;
    wire [ADDRESS_LENGTH-1:0] mem_address;

    assign mem_en = core_select ? from_core_mem_en: from_intf_mem_ctrl_mem_en;
    assign mem_wr_en = core_select ? from_core_mem_wr_en: from_intf_mem_ctrl_mem_wr_en;
    assign mem_rd_en = core_select ? from_core_mem_rd_en: from_intf_mem_ctrl_mem_rd_en;
    assign mem_data = core_select ? from_core_mem_data_in : from_intf_mem_ctrl_mem_data_in;
    assign mem_address = core_select ? from_core_mem_address[ADDRESS_LENGTH-1:2]:
from_intf_mem_ctrl_mem_address[ADDRESS_LENGTH-1:0];

    wire [3:0] mem_we_out;
    wire [3:0] spi_mem_we_out;
    wire [3:0] core_mem_we_out;
    wire [DATA_LENGTH-1:0] mem_data_out_wire;

    assign spi_mem_we_out = (from_intf_mem_ctrl_mem_data_length==2'b00) ? 4'b0000 :
(from_intf_mem_ctrl_mem_data_length==2'b01) ? 4'b0001 :
(from_intf_mem_ctrl_mem_data_length==2'b10) ? 4'b0011 : 4'b1111; //data_length signal is stuck
2'b11 from spi interface
    assign core_mem_we_out = (from_core_mem_data_length==2'b00) ? 4'b1111 :
(from_core_mem_data_length==2'b01) ? 4'b0011 : (from_core_mem_data_length==2'b10) ? 4'b0001 :
4'b0000; // data_length 2'b00 means full word write;
    assign mem_we_out = mem_wr_en ? core_select ? core_mem_we_out : spi_mem_we_out : 4'b0000;

    assign to_core_mem_data_out = (core_select & mem_rd_en) ? mem_data_out_wire :
to_core_mem_data_out;
    assign to_intf_mem_ctrl_mem_data_out = (~core_select & mem_rd_en) ? mem_data_out_wire :
to_intf_mem_ctrl_mem_data_out;
```

```
    DFFRAM_RTL_2048 #(ADDRESS_LENGTH,DATA_LENGTH) memory_from_DFFRAM (
        .CLK(clk),
        .WE(mem_we_out),
        .EN(mem_en),
        .Di(mem_data),
        .Do(mem_data_out_wire),
        .A(mem_address)
    );

endmodule
```

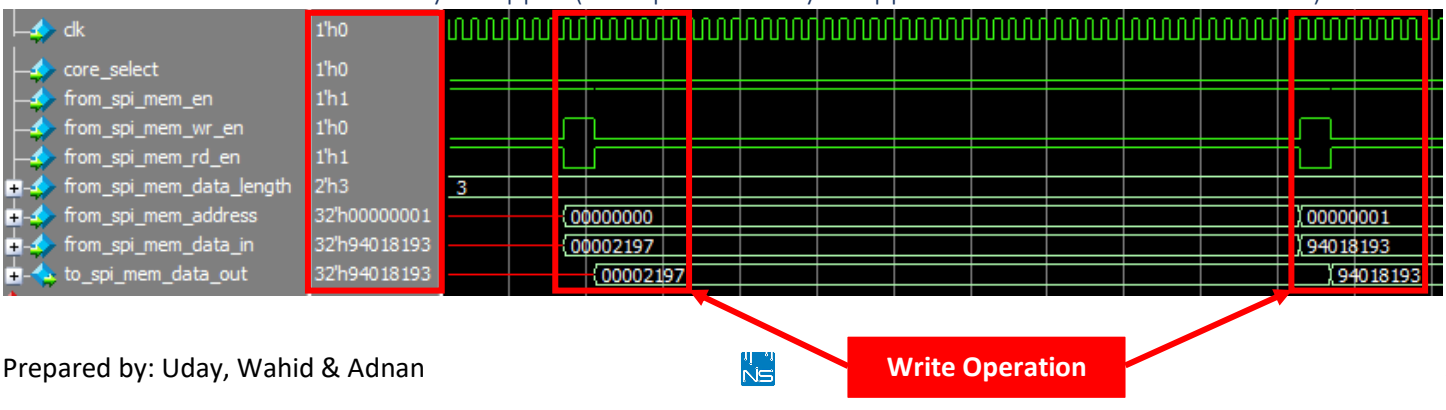
Memory RTL

```
module DFFRAM_RTL_2048 #(parameter ADDRESS_LENGTH, parameter DATA_LENGTH) //${size}
 (
     CLK,
     WE,
     EN,
     Di,
     Do,
     A
 );

    input   wire            CLK;
    input   wire    [3:0]   WE;
    input   wire            EN;
    input   wire    [(DATA_LENGTH) -1:0]  Di;
    output  reg     [(DATA_LENGTH) -1:0]  Do;
    input   wire    [(ADDRESS_LENGTH - 1): 0]   A;
    reg [(DATA_LENGTH) -1:0] RAM[2047 : 0];

    always @(posedge CLK)
        if(EN) begin
            Do <= RAM[A];
            if(WE[0]) RAM[A][ 7: 0] <= Di[7:0];
            if(WE[1]) RAM[A][15: 8] <= Di[15:8];
            if(WE[2]) RAM[A][23:16] <= Di[23:16];
            if(WE[3]) RAM[A][31:24] <= Di[31:24];
        end
        else
            Do <= 32'b0;
 endmodule
```

Simulation Waveform for Memory Wrapper (Example: Memory wrapper connected to a SPI controller)



Physical Design Specification of RV32I Core

Description	Remarks
Base Clock Frequency	100 MHz
Base Area	8323.254 um^2
Base Power	0.36872112 mW
Top power routing layer	Metal 9
Bottom power routing layer	Metal 8
Starting density at Floorplan Stage	50%
Percentage of Power Rail Area	10%
Top Routing Layer	Metal 7
Bottom Routing Layer	Metal 1
Pin Layer	Metal 3 & 4
PDK	gpdk045

Deliverables

Report will include the following:

1. Block Diagram of the design.
2. All RTL Codes, input SDC Files, I/O File. **Introducing pipelining stages is a plus. Implementation of Harvard Architecture for separate instruction and data memory is a plus.**
3. Top level testbench script and all simulation scripts (snapshot if needed).
4. Time taken to run the benchmark instructions. **Bonus mark for running additional instructions.**
5. Waveforms showing **register [12] = 1** for the benchmarking instructions.
6. No need to synthesize the communication interface, memory controller, memory itself and memory wrapper; those are for simulation purpose only. **Just synthesize and physical design the RV32I Core only.**
7. Output SDC and reports consists of **area, time and power.**
8. Comparison of area, time and power at every stage of APR. **Best PPA (Power Performance Area) improvement will get bonus marks.**
9. Snapshots of your design after floorplan and final stage of APR.

Submit a report on the project in PDF format and also prepare PowerPoint presentation for the final submission.

Prepare & submit all the necessary things before the submission date.

Project Schedule

Content	Review Date	Mode
Project Discussion	TBD	Offline
Architecture Design & Diagram	TBD	Online
RTL & Verification	TBD	Online
Physical Design	TBD	Online
Project Submission & Final Presentation	TBD	Offline

Benchmark Instructions

C Code

At first, we need to write the C code for detecting a prime/non-prime number.

```
#include <stdio.h>

int prime_number() {

  int n=50, i, flag = 0;

  // 0 and 1 are not prime numbers
  // change flag to 1 for non-prime number
  if (n == 0 || n == 1)
    flag = 1;

  for (i = 2; i <= n; ++i) {

    // if n is divisible by i, then n is not prime
    // change flag to 1 for non-prime number
    for(;;)
    {
    if (n - i == 0) {
      flag = 1;
      break;
    }
    else if (n-i<0)
    {
       flag = 0;
       break;
    }
    else
    {
       n=n-i;
    }
    }
    if(flag==1)
    {
        break;
    }
    }

  // flag is 0 for prime numbers

  return flag;
}

int main() {

   int m;
   m=prime_number();
   __asm__("mv a2, %0\n" :: "r" (m));

}
```

Assembly code

a.out: file format elf32-littleriscv

```
Disassembly of section .init:

00000000 <_start>: # Start-up sequence for 2KB Memory
   0:  00002197            auipc  gp,0x2
   4:  94018193            addi   gp,gp,-1728 # 1940 <__global_pointer$>
   8:  00000117            auipc  sp,0x0
   c:  7f810113            addi   sp,sp,2040 # 800 <__stack_top>
  10:  00010433            add    s0,sp,zero
  14:  0c80006f            j      dc <main>

Disassembly of section .text:

00000018 <prime_number>:
  18:  fe010113            addi   sp,sp,-32
  1c:  00812e23            sw     s0,28(sp)
  20:  02010413            addi   s0,sp,32
  24:  03200793            li     a5,50
  28:  fef42623                   sw     a5,-20(s0)
  2c:  fe042223            sw     zero,-28(s0)
  30:  fec42783                   lw     a5,-20(s0)
  34:  00078863            beqz   a5,44 <prime_number+0x2c>    not taken
  38:  fec42703                   lw     a4,-20(s0)
  3c:  00100793            li     a5,1
  40:  00f71663            bne    a4,a5,4c <prime_number+0x34>    not taken
  44:  00100793            li     a5,1
  48:  fef42223                   sw     a5,-28(s0)
  4c:  00200793            li     a5,2
  50:  fef42423                   sw     a5,-24(s0)
  54:  0600006f            j      b4 <prime_number+0x9c>    taken
  58:  fec42703                   lw     a4,-20(s0)
  5c:  fe842783            lw     a5,-24(s0)
  60:  00f71863            bne    a4,a5,70 <prime_number+0x58>
  64:  00100793            li     a5,1
  68:  fef42223                   sw     a5,-28(s0)
  6c:  0300006f            j      9c <prime_number+0x84>
  70:  fec42703                   lw     a4,-20(s0)
  74:  fe842783            lw     a5,-24(s0)
  78:  40f707b3            sub    a5,a4,a5
  7c:  0007d663            bgez   a5,88 <prime_number+0x70>
  80:  fe042223            sw     zero,-28(s0)
  84:  0180006f            j      9c <prime_number+0x84>
  88:  fec42703                   lw     a4,-20(s0)
  8c:  fe842783            lw     a5,-24(s0)
  90:  40f707b3            sub    a5,a4,a5
  94:  fef42623                   sw     a5,-20(s0)
  98:  fc1ff06f                   j      58 <prime_number+0x40>
  9c:  fe442703            lw     a4,-28(s0)
  a0:  00100793            li     a5,1
  a4:  02f70063            beq    a4,a5,c4 <prime_number+0xac>
  a8:  fe842783            lw     a5,-24(s0)
  ac:  00178793            addi   a5,a5,1
  b0:  fef42423                   sw     a5,-24(s0)
```

```
  b4:  fe842703            lw     a4,-24(s0)
  b8:  fec42783                   lw     a5,-20(s0)
  bc:  f8e7dee3                   bge    a5,a4,58 <prime_number+0x40>
  c0:  0080006f            j      c8 <prime_number+0xb0>
  c4:  00000013            nop
  c8:  fe442783            lw     a5,-28(s0)
  cc:  00078513            mv     a0,a5
  d0:  01c12403            lw     s0,28(sp)
  d4:  02010113            addi   sp,sp,32
  d8:  00008067            ret

000000dc <main>:
  dc:  fe010113            addi   sp,sp,-32
  e0:  00112e23            sw     ra,28(sp)
  e4:  00812c23            sw     s0,24(sp)
  e8:  02010413            addi   s0,sp,32
  ec:  f2dff0ef                   jal    ra,18 <prime_number>
  f0:  fea42623            sw     a0,-20(s0)
  f4:  fec42783                   lw     a5,-20(s0)
  f8:  00078613            mv     a2,a5
  fc:  00000013            nop
100:   00078513            mv     a0,a5
104:   01c12083            lw     ra,28(sp)
108:   01812403            lw     s0,24(sp)
10c:   02010113            addi   sp,sp,32
110:   00008067            ret
```

Binary Dump

This is the instruction set that we will feed our RISC-V Core Design.

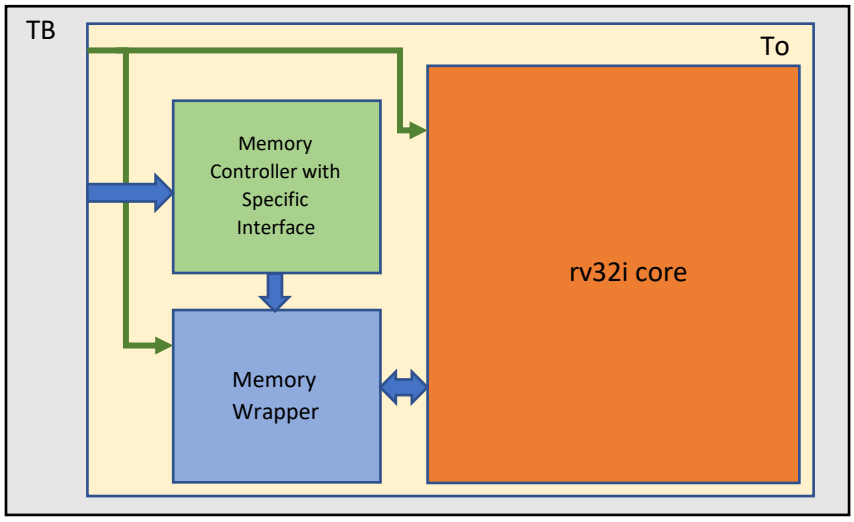
```
00002197
94018193
00000117
7f810113
00010433
0c80006f
fe010113
00812e23
02010413
03200793
fef42623
fe042223
fec42783
00078863
fec42703
00100793
00f71663
00100793
fef42223
00200793
fef42423
0600006f
fec42703
fe842783
00f71863
00100793
fef42223
```

```
0300006f
fec42703
fe842783
40f707b3
0007d663
fe042223
0180006f
fec42703
fe842783
40f707b3
fef42623
fc1ff06f
fe442703
00100793
02f70063
fe842783
00178793
fef42423
fe842703
fec42783
f8e7dee3
0080006f
00000013
fe442783
00078513
01c12403
02010113
00008067
fe010113
00112e23
00812c23
02010413
f2dff0ef
fea42623
fec42783
00078613
00000013
00078513
01c12083
01812403
02010113
00008067
```

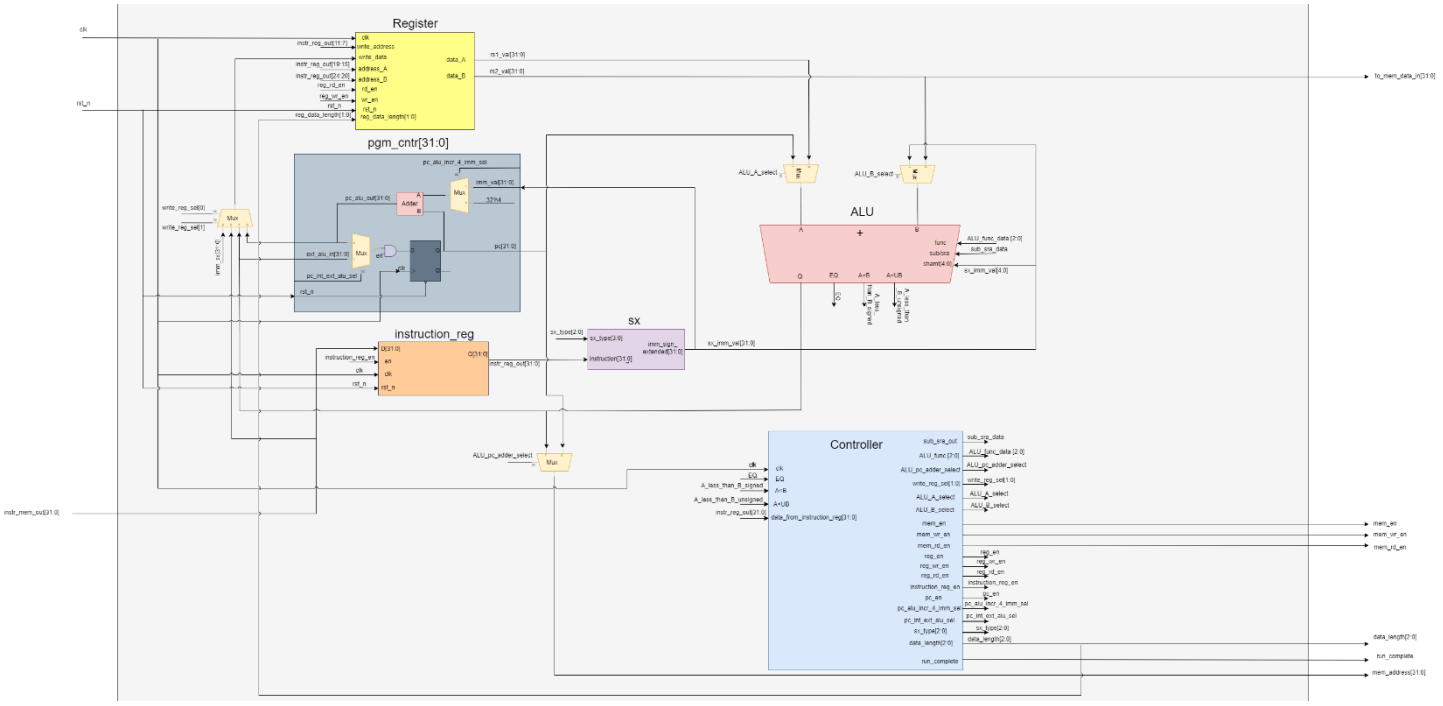
If all the instructions are executed correctly, then it will give an output of 1 at register [12].

References

Testbench Structure



Core Architecture Reference



Example Design

https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html

Video Lecture

Playlist (Video 1-4): https://www.youtube.com/watch?v=LKB5I12LctU&list=PL3by7evD3F53Dz2RiB47Ztp9l_piGVuus&index=1

Reference Compiler Usage

Playlist (Video 1-3): https://www.youtube.com/watch?v=qWqIkCLmZoE&list=PLERTijJOmYrDiiWd10iRHY0VRHdJwUH4g