Homework 4 Report

Course:

IL2234 HT25 Digital Systems Design and Verification using Hardware Description Languages

Student Name:

Jinye Gong

GitHub Repository:

https://github.com/kth-ees/il2234ht25-homework-4-jinye-gong

Submission Date:

2025-10-15

Question 1:

Calculation of the inverse of a 2 by 2 matrix is shown below.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, A^{-1} = \frac{1}{|A|}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} aOut & bOut \\ cOut & dOut \end{bmatrix}$$

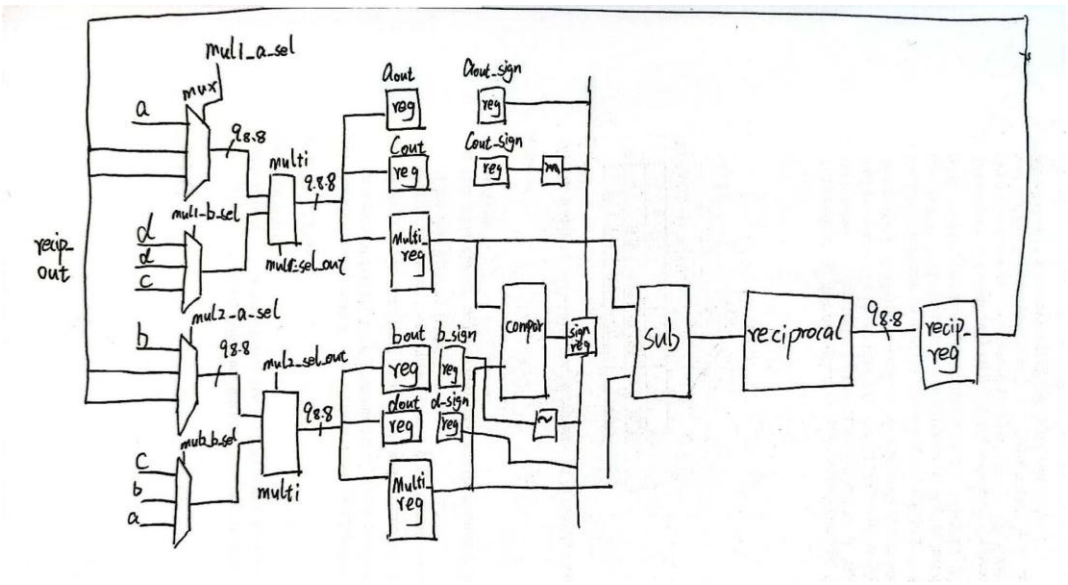
You are to design an RTL circuit for this calculation. When *start* is asserted, *aIn*, *bIn*, *cIn*, and *dIn* 16-bit busses will contain the four elements of the matrix in upper left to lower right order. When the inverse calculation is completed, the **IMC** (**I**nverse **M**atrix **C**alculator) generates a 1 on *ready* and keeps this value until a new round of calculation begins. When calculation is completed, the output data becomes available on *aOut*, *bOut*, *cOut*, and *dOut* output buses. Input and output data formats are 16-bit fixed point with eight integer bits. The inputs have only integer parts, and the outputs are 16-bit data with integer and fractional parts. The input integer part should be less than or equal to 15.

A. Create behavioural models for the multipliers and adder/subtractor.

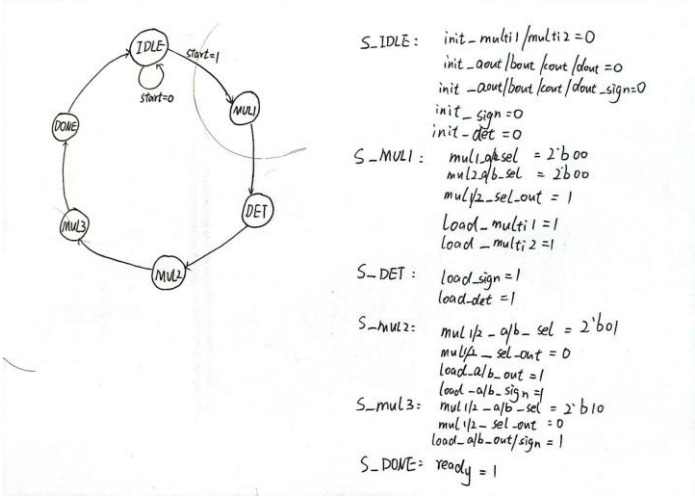
```
// 16x16 unsigned multiplier with selectable 16-bit slice
module multi (
    input  logic [15:0] a,
    input  logic [15:0] b,
    input  logic        select_output, // 1: [31:16], 0: [23:8]
    output logic [15:0] y
);
    logic [31:0] p;

  // behavioural multiply
    assign p = a * b;


    always_comb begin
    y = select_output ? p[31:16] : p[23:8];
end
endmodule
```

```
module add_sub (
    input  logic [15:0] a,
    input  logic [15:0] b,
    input  logic        sel, // 0: add, 1: sub
    output logic [15:0] y
);
    always_comb begin
        if (sel == 1'b0) begin
            y = a + b; // addition
        end else begin
            y = a - b; // subtraction
        end
    end
endmodule
```

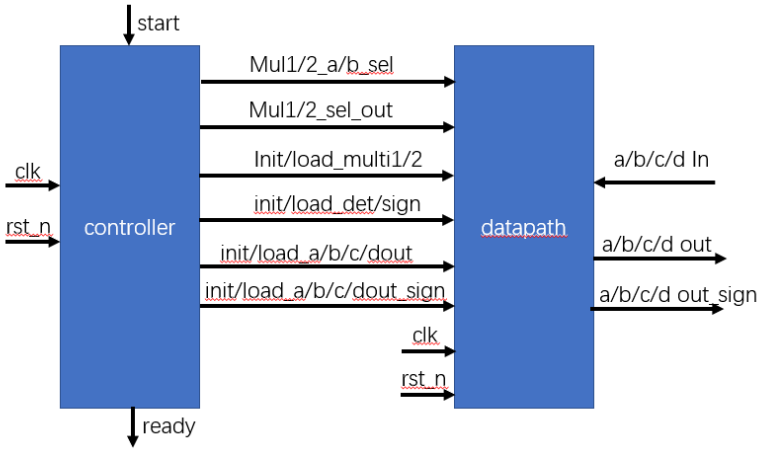
B. In your report, draw the complete datapath of **IMC**, including the components and necessary internal control signals.



C. Draw a state diagram showing your controller's behaviour in your report. In each state, show the control signals that are issued.



D. In your report, show wiring between the datapath and the controller

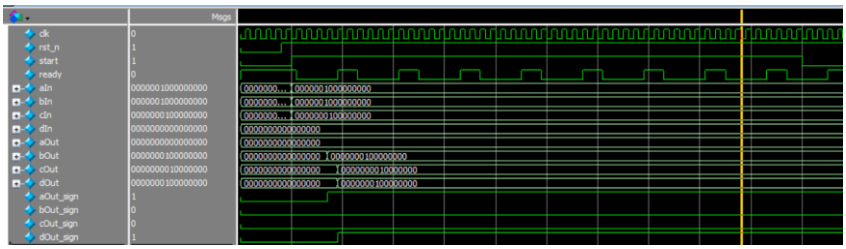


F. Verify your design with a Testbench

Test1:

```
initial begin
    start = 0;
    aIn = '0; bIn = '0; cIn = '0; dIn = '0;
    #50;
    start = 1;
    aIn = 16'b0000_0010_0000_0000; //2
    bIn = 16'b0000_0010_0000_0000; //2
    cIn = 16'b0000_0001_0000_0000; //1
    dIn = 16'b0000_0000_0000_0000; //0
    #500;
    start = 0;

end
```

Result1:

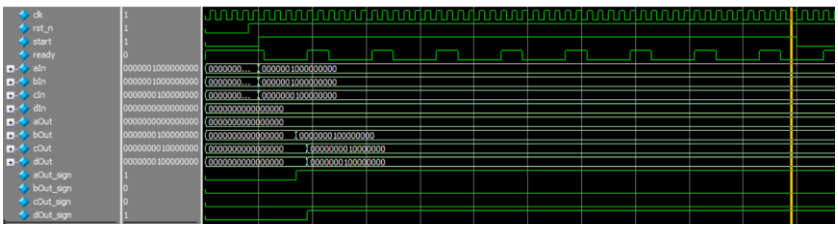


The matrix operation corrects!

Test2:

```
initial begin
    start = 0;
    aIn = '0; bIn = '0; cIn = '0; dIn = '0;
    #50;
    start = 1;
    aIn = 16'b0000_0100_0000_0000; //4
    bIn = 16'b0000_0010_0000_0000; //2
    cIn = 16'b0000_0011_0000_0000; //3
    dIn = 16'b0000_0001_0000_0000; //1
    #500;
    start = 0;

end
```

Result2:



The matrix operation corrects!

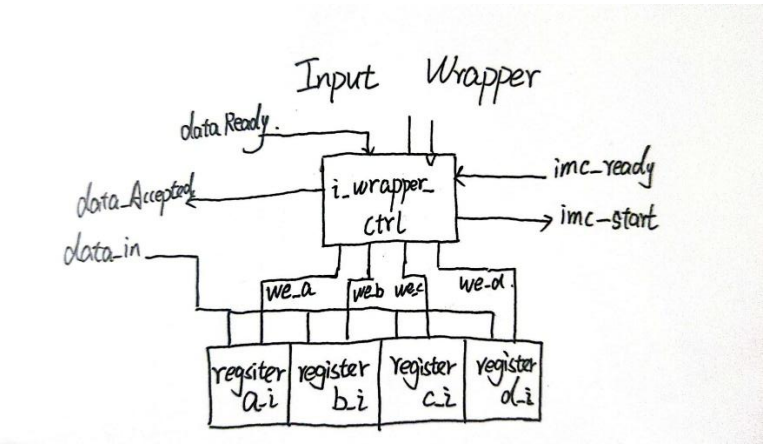
Question 2:

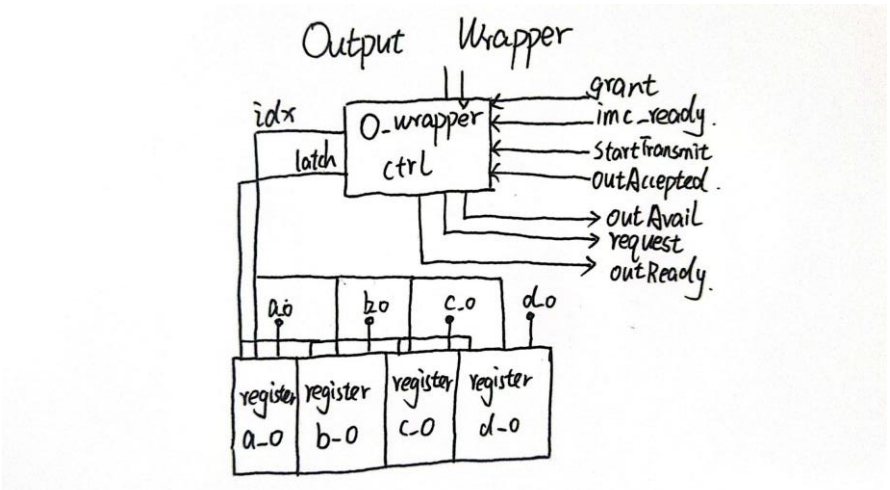
In this problem, you are to design an **input wrapper** and an **output wrapper** that connect the IMC circuit of the previous problem to a 16-bit arbitrated bus. (If you couldn't design the IMC you can generate the input-output of IMC which are connected to the input and output wrapper in your testbench.)

The **input wrapper** waits for four 16-bit data that will be handshaked to it using a two-line *dataReady, dataAccept* fully responsive handshaking scheme. When such is received, it checks if the IMC circuit is ready to receive its four 16-bit inputs by monitoring the IMC's *ready* output. If so, it issues a *start* signal and allows the IMC to start its operation.

The **output wrapper** works independently of the input wrapper. The output wrapper receives the four outputs of the IMC circuit when IMC issues the *ready* signal. The output wrapper has an *outAvail* output that informs an external device of the availability of four 16-bit elements of the transposed matrix. When this signal is asserted, the external device may issue a start signal, *startTransmit*, to tell the output wrapper to start sending the results. The output wrapper uses *request* to get permission to use the bus, that will be responded by *grant* when the output wrapper is allowed to drive the bus. The output wrapper uses fully responsive two-line handshaking using an *outReady* output and an *outAccepted* input. The four 16-bit data outputs will be sent over the bus in a burst fashion.

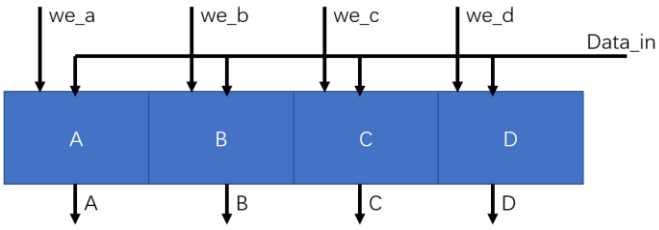
A. In your report, draw block diagrams of the **input wrapper**, **output wrapper**, and the interfacing bus, and how they wrap around the IMC circuit.



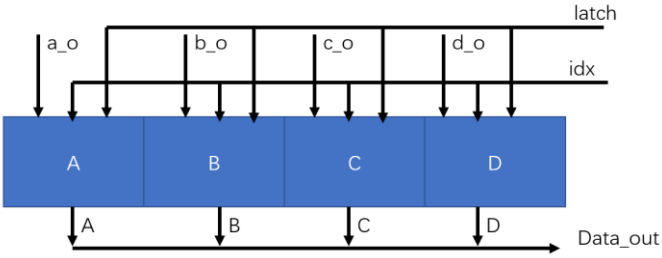


B. In your report, draw the datapath of the interface circuits, identifying control signals that are issued by the controller.

Inwrapper_datapath

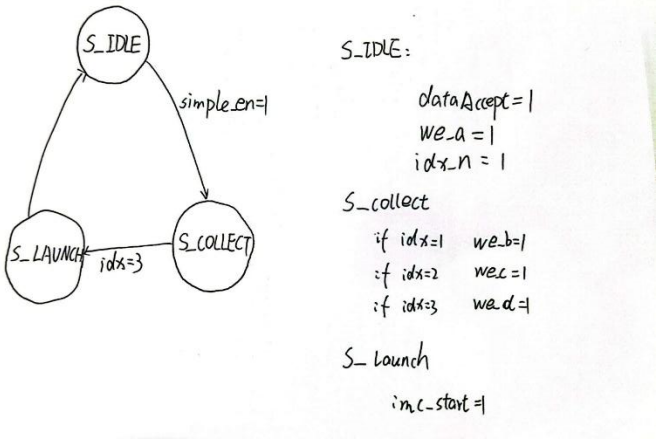


Outwrapper_datapath

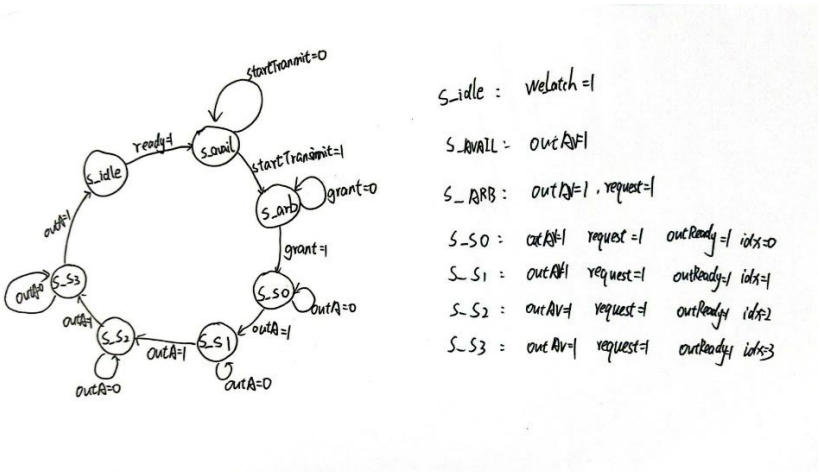


C. In your report, draw the state machines for the implementation of the interface circuits controller.

Inwrapper_state_machine

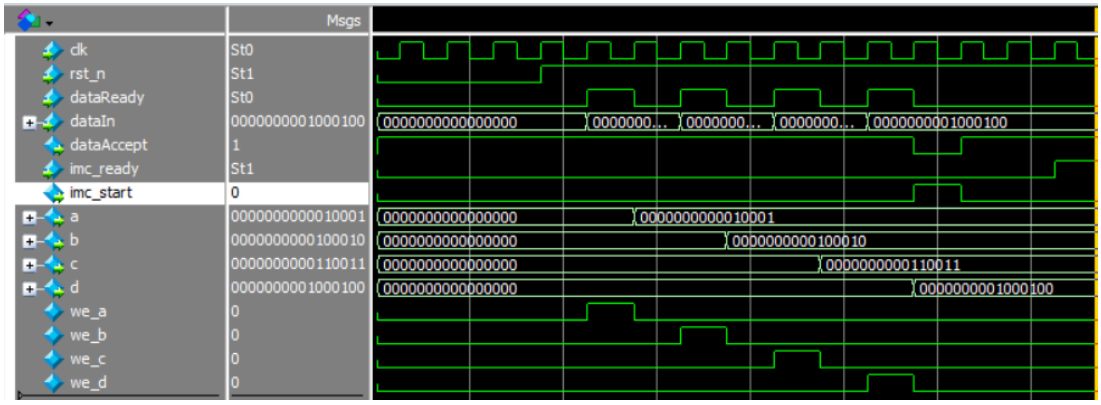


outwrapper_state_machine

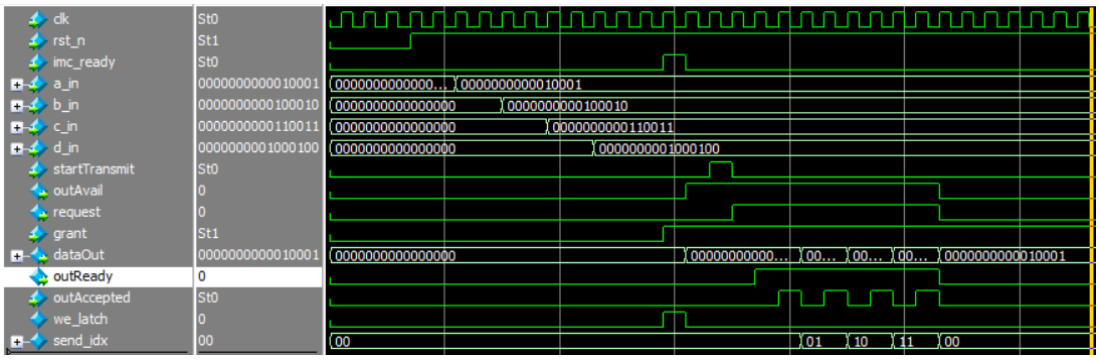


E. Verify your design

Inwrapper



outwrapper

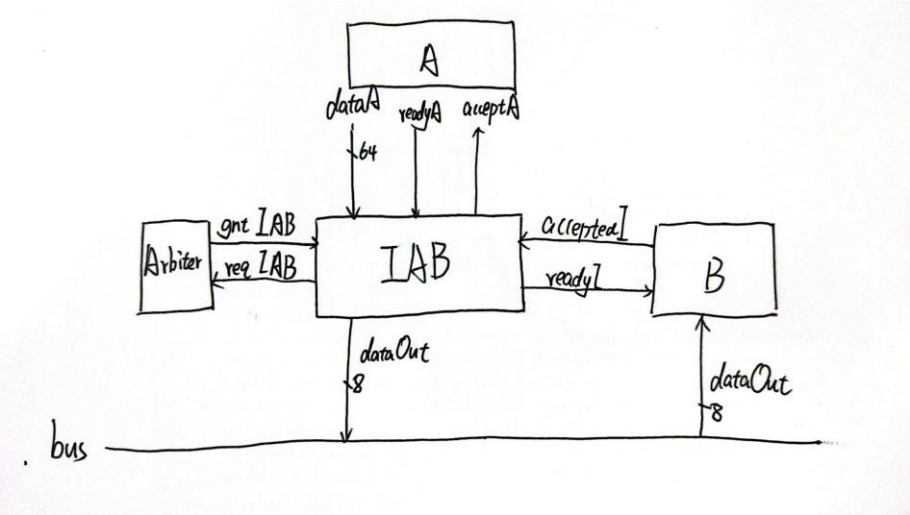


From the waveform, the input and output wrappers are verified to transfer data properly

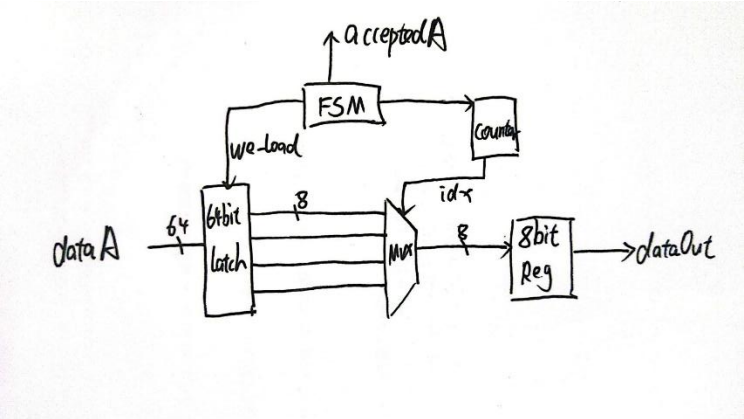
Question 3:

You are to design an interface circuit for device *A* with a 64-bit data bus (*dataA*) that is to write to device *B*, which receives 8-bit data through an arbitrated 8-bit shared bus (*sharedBus*). The interface circuit is called **IAB** and communicates with device *A* by fully responsive two-line handshaking initiated by device A (*readyA*, when *A* is to write, and *acceptedA* when the 64-bit data is received by **IAB**). The interface circuit requests the use of the *sharedBus* by issuing *reqIAB* and waits for *gntIAB* before it can use the bus. When **IAB** has access to the bus, it writes the 64-bit data it received from *A* in eight 8-bit chunks to device *B* in a burst. For each write to *B*, **IAB** communicates with device *B* by fully responsive two-line handshaking initiated by IAB (*readyI*, when **IAB** is to write, and *acceptedI* when the 8-bit data is received by device *B*).

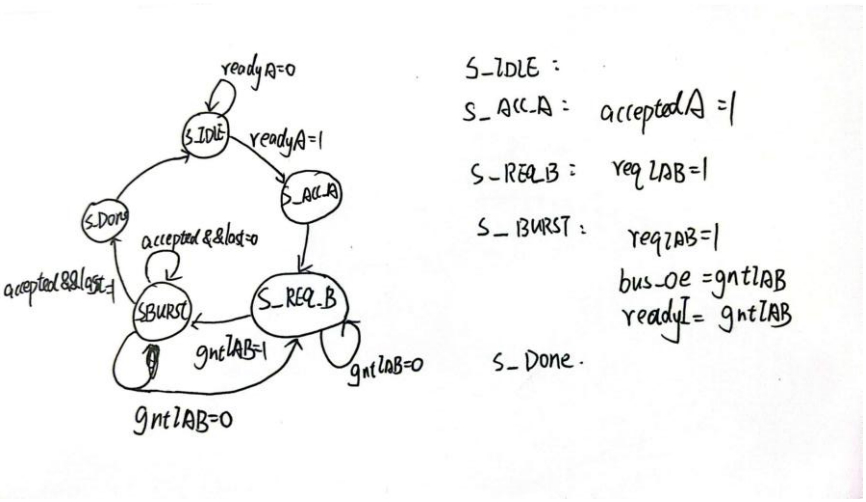
A. In your report, draw the block diagram of the complete system, including your circuits' input and output bus, shared bus, and the bus between the two devices and your interface circuit.



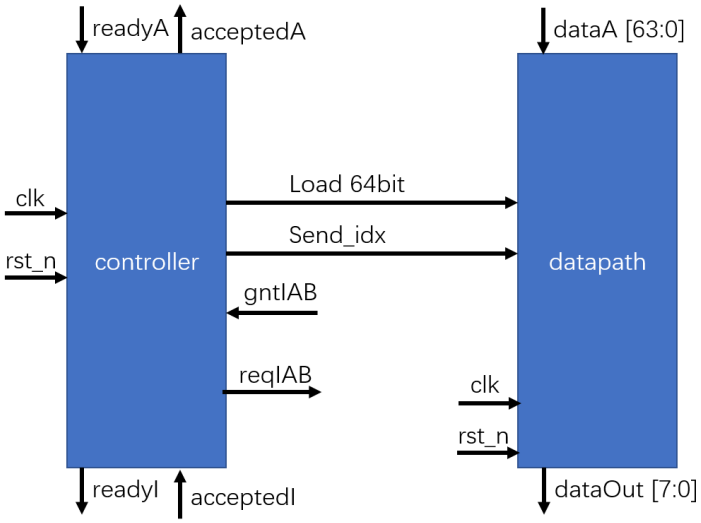
B. In your report, draw the complete datapath of **IAB**, including the components and necessary internal control signals.



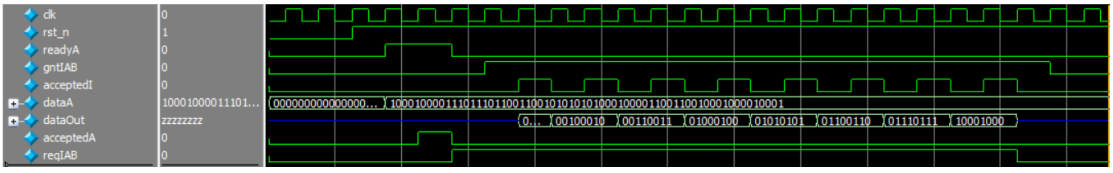
C. In your report, draw a state diagram showing the **IAB** controller's behaviour. In each state, show the control signals that are issued.



D. In your report, draw wiring between the datapath and controller.



F. Verify your design with a testbench.



From the waveform, the input and output are verified to transfer data properly

Question 4:

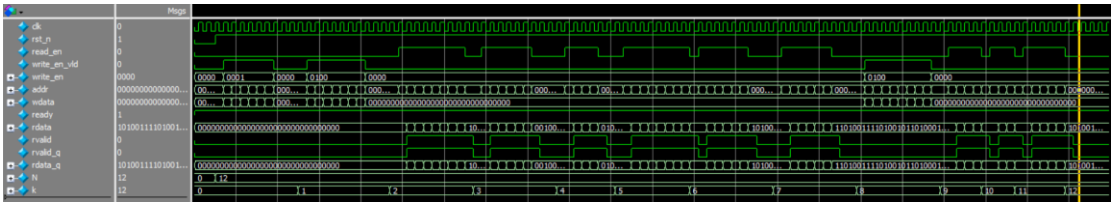
Constrained Random Test Generation: You are to generate constrained random test data for the CPU–Memory interface shown below, which you have previously seen in *Milestones 3 and 4*. In this exercise, the CPU performs **burst transfers** when reading from or writing to memory.

A burst access is a sequence of consecutive transfers. In this scenario, we do bursts without an explicit `burst` signals. We just handle bursts internally by generating one transaction with `burst_len=N`, driving `read_en` or `write_en` high for `N` consecutive cycles, and incrementing the address by `4` each cycle.

Define the following constraints to control randomization behavior.

- Burst Length Constraint: Limits the number of data items in a burst to between 2 and 8 transfers.
- Address Alignment Constraint: Ensures the address is word-aligned (4-byte aligned)
- Address Range Partitioning: Assigns different memory regions for read and write operations. Memory region for reading is `[32'h0000_0000:32'h0000_FFFF]` and for writing is `[32'h1000_0000:32'h1000_FFFF]`.
- Operation Probability (Read vs Write): Randomly selects read or write operations with weighted probability:
  - 80% chance to generate a read
  - 20% chance to generate a write
- Write Enable Constraint: Ensures valid `write_en` values only for write transfers. At least one byte must be enabled (non-zero pattern). For read transfers, `write_en` must be zero. The 4 bits of write enable (`write_en`) are used for the four bytes in the addressed word. Only the 8 values 0000, 1111, 1100, 0011, 1000, 0100, 0010, and 0001 are possible, i.e. no write, write 32 bits, write upper 16 bits, write lower 16, or write a single byte, respectively.

The code has uploaded in GitHub



Question 5:

You are provided with the design and SystemVerilog descriptions of a Moore FSM that detects the sequence 1011. Your task is to write SystemVerilog properties to verify the correct functionality of the sequence detector. Your assertion-based verification should at least include the following properties:

- Whenever the output `w` is high, the last four input bits equal `1011`
- The output `w` is asserted for only one clock cycle per detection
- Every property you write must include a reset condition using disable `iff(!rst_n)`

Hints (Optional):

You may use `$past(signal, n)` to refer to the value of a signal `n` clock cycles ago.

You may use `$rose(w)` if you want to detect the rising edge of `w`. This is useful for checking properties at the moment `w` is asserted.

Think of disable `iff (!rst_n)` as "turn off the property while the FSM is in reset."

Note: These hints are optional. You are encouraged to implement the properties in any way that correctly verifies the FSM.

The code has uploaded in GitHub