Lab11--Design of Register File & Register Transfer Level

姓名：潘子曰 学号：3180105354 专业：计算机科学与技术
课程名称：逻辑与计算机设计基础实验 同组学生姓名：张佳文
试验时间：2019-12-05 实验地点：紫金港东4-509 指导老师：洪奇军

1. Objectives & Requirements

- Master the structure of a **register** and how to design it.
- Master the working principle of the **register file** and how to design it.
- Master **Register Transfer Level (RTL)** and how to design it.
- Master the design of register transfer level based on the **BUS**.
- Know the basic concept of register and register file in computer.

2. Contents & Principles

2.1 Tasks

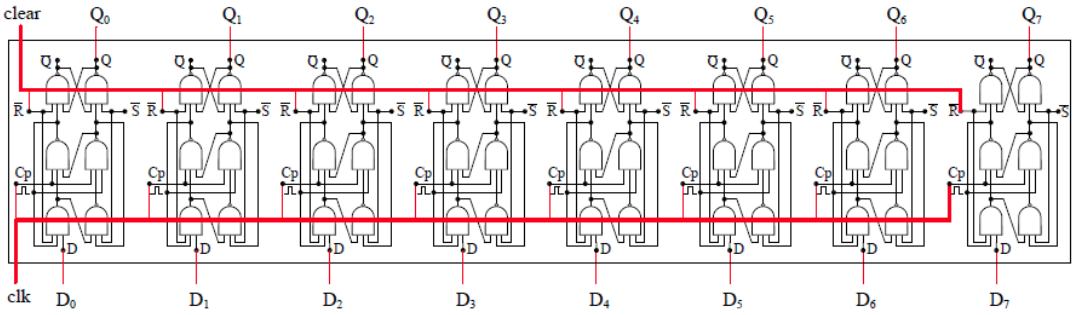
- Design **32-bit register** with clock input.
- Design **8×32-bit register file**.
- Integrate the design with experiment I/O environment and implement **ALU arithmetic based on RTL**.

2.2 Principles

Register

Registers are circuits typically composed of flip flops, often with characteristics such as:

- The ability to read or write multiple bits at a time, and
- Using an address to select a particular register in a manner similar to a memory address.



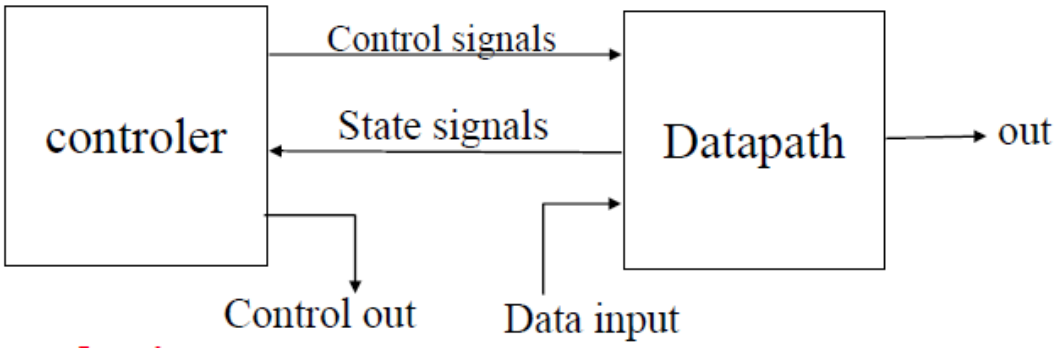
Register File

A **register file** is an array of processor registers in (CPU).

- **Write**: register address → variable decoder.
- **Read**: register address → multiplexer

Register Transfer Level

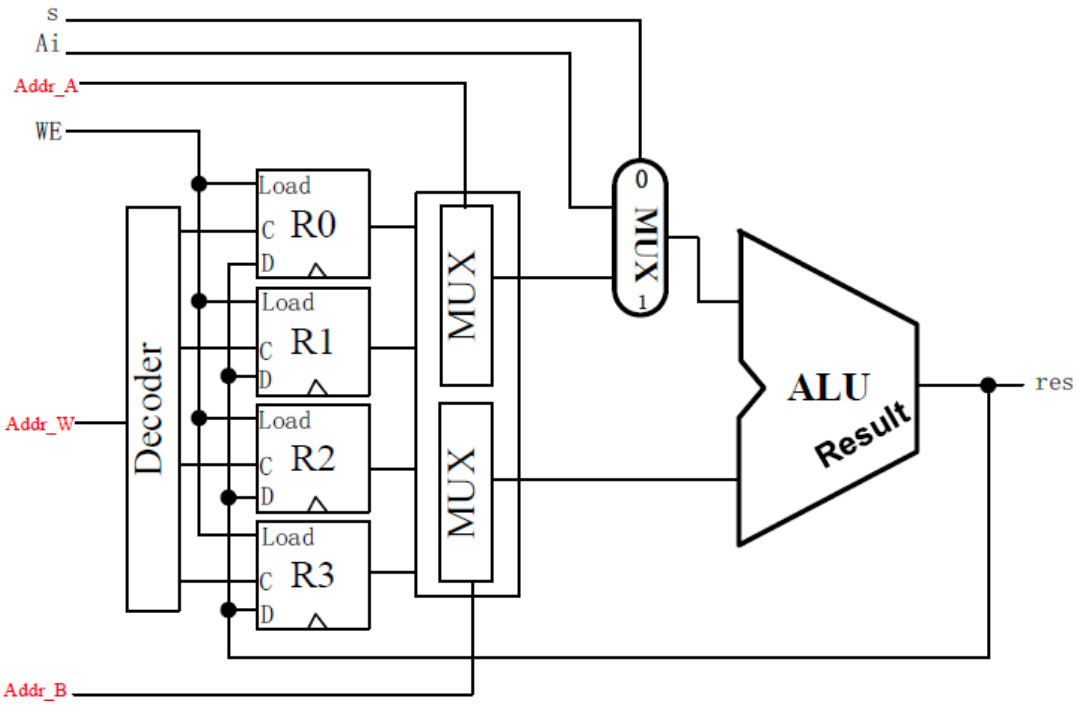
Register-transfer level (**RTL**) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.



RTL is basically composed by the following:

- Set of registers
- Operations
- Control of operations

This is an RTL example:



There are two output port from the register file. The value fetched from the register file through the output port B will be then re-transferred into the register file and participates in the next round of calculation.

3. Major Experiment Instruments

Equipment
Computer (Intel Core i7-9750H, 16GB memory)
Sword circuit design box
Xilinx ISE 14.7

4. Experiment Procedure

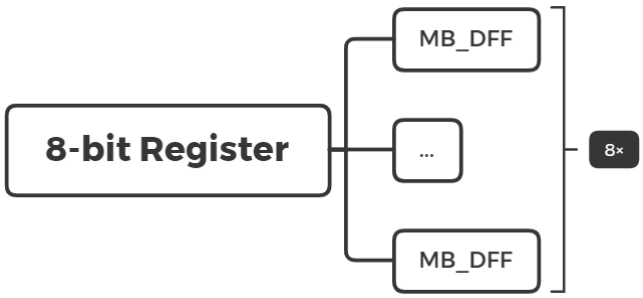
Task1: Exp110-REGS

The following steps will be included in this task:

- Design and implement 32-bit register
  - Implement based on maintain-block D flip-flop
  - By structuralized description
- Design and implement 8×32 register file
  - Implement by structuralized description
    - One input port: **WE** (write enable), **Addr_W** (write-in address)
    - Two output ports: **Addr_A** & **Addr_B**
- Integrate register file to the **IO framework**
  - Rename the top module **Exp11-RTL**
  - Implement other functions:
    - Implement **register transfer control structure** with **register file** and **ALU** module.
    - Assign $\mathbf{A_i}$ as one input port and $\mathbf{B_i}$ as the control code.
    - Assign the display channel 4 & 5 as the output port.

8-bit register

- Implement by calling 8 **MB_DFF** modules

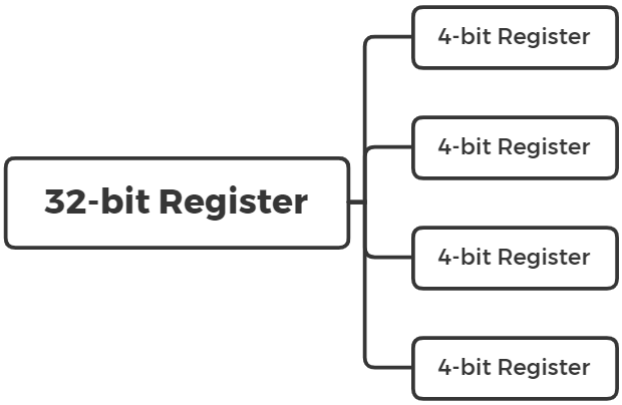



```
module Reg_8bit(
    input clk,
    input [7:0]D,
    input clear,
```

```
    output [7:0]Q
   );

    wire [7:0] Qbar;
    assign cr=~clear;

MB_DFF  T0(.Cp(clk), .D(D[0]), .Rn(cr), .Sn(1'b1), .Q(Q[0]),
.Qn(Qbar[0])),
        T1(.Cp(clk), .D(D[1]), .Rn(cr), .Sn(1'b1), .Q(Q[1]),
.Qn(Qbar[1])),
        T2(.Cp(clk), .D(D[2]), .Rn(cr), .Sn(1'b1), .Q(Q[2]),
.Qn(Qbar[2])),
        T3(.Cp(clk), .D(D[3]), .Rn(cr), .Sn(1'b1), .Q(Q[3]),
.Qn(Qbar[3])),
        T4(.Cp(clk), .D(D[4]), .Rn(cr), .Sn(1'b1), .Q(Q[4]),
.Qn(Qbar[4])),
        T5(.Cp(clk), .D(D[5]), .Rn(cr), .Sn(1'b1), .Q(Q[5]),
.Qn(Qbar[5])),
        T6(.Cp(clk), .D(D[6]), .Rn(cr), .Sn(1'b1), .Q(Q[6]),
.Qn(Qbar[6])),
        T7(.Cp(clk), .D(D[7]), .Rn(cr), .Sn(1'b1), .Q(Q[7]),
.Qn(Qbar[7]));

endmodule
```

32-bit Register

- Implement by calling 4 **8-bit register** module




```
module Reg_32bit(
    input clk,
    input clear,
    input Load,
    input [31:0] Di,
    output [31: 0] Dot
    );

    wire [31:0] Dbar;
    assign Dbar = Load? Di:Dot;
    BUFG cc(clk1, clk);

Reg_8bit    T0(.clk(clk), .D(Dbar[7:0]), .clear(clear),
.Q(Dot[7:0])),
```

```
            T1(.clk(clk), .D(Dbar[15:8]), .clear(clear),
.Q(Dot[15:8])),
            T2(.clk(clk), .D(Dbar[23:16]), .clear(clear),
.Q(Dot[23:16])),
            T3(.clk(clk), .D(Dbar[31:24]), .clear(clear),
.Q(Dot[31:24]));

endmodule

//This is the behavioral description
//module Reg_32bit(
//  input clk,
//  input [31:0] Di,
//  input clear,
//  input Load,
//  output reg [31:0] Dot
//  );
//
//  always@(posedge clk or posedge clear)
//  if(clear) Dot <= 0;
//  else if(Load) Dot <= Di;
//  else Dot <= Dot;
//endmodule
```

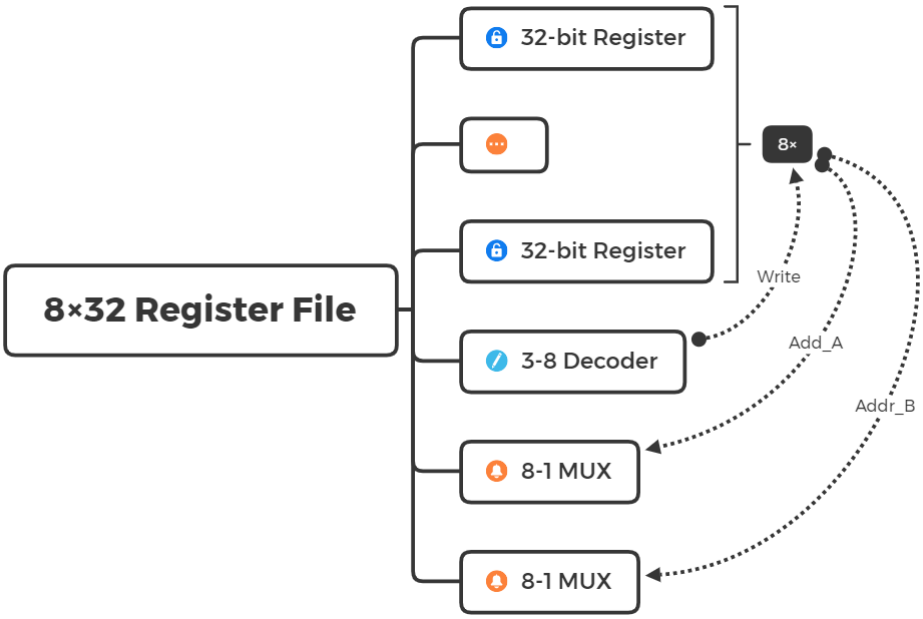
Simulation code for 32-bit register

```
module Reg_32bit_test;

    // Inputs
    reg clk;
    reg clear;
    reg Load;
    reg [31:0] Di;

    // Outputs
    wire [31:0] Dot;

    // Instantiate the Unit Under Test (UUT)
    Reg_32bit uut (
        .clk(clk),
        .clear(clear),
        .Load(Load),
        .Di(Di),
        .Dot(Dot)
    );

    initial begin
    Load = 0;
    clk = 0;
    fork
        forever #20 clk<=~clk;
        #20; clear = 0;
        begin
            Di = 32'hAAAAAAAA;
            #50; Load <= 1;
```

```
            #40; Load <= 0;
            Di = 32'h55555555;
            #20; Load <= 1;
            #40; Load <= 0;
            Di = 32'hA5A5A5A5;
            #70; Load <= 1;
            #30; Load <= 0;
            #40; clear = 1;
            #30; clear = 0;
            #25; Load <= 1;
            #45; Load <= 0;
            Di = 32'h5A5A5A5A;
            #65; Load <= 1;
            #45; Load <= 0;
        end
    join
    end

endmodule
```

8×32 register file

- Implement by calling 8 **32-bit register** modules
- An **HCT_138** decoder is used as the register **DeMUX** to write value into a specified register
- 2 **MUX**s are used to read the value stored in the registers which are specified by their address code.



Simulation code for 8×32 register file

```
module Reg_8_32_test;
    // Inputs
    reg clk;
    reg cr;
    reg WE;
    reg [31:0] Di;
    reg [2:0] Addr_A;
```

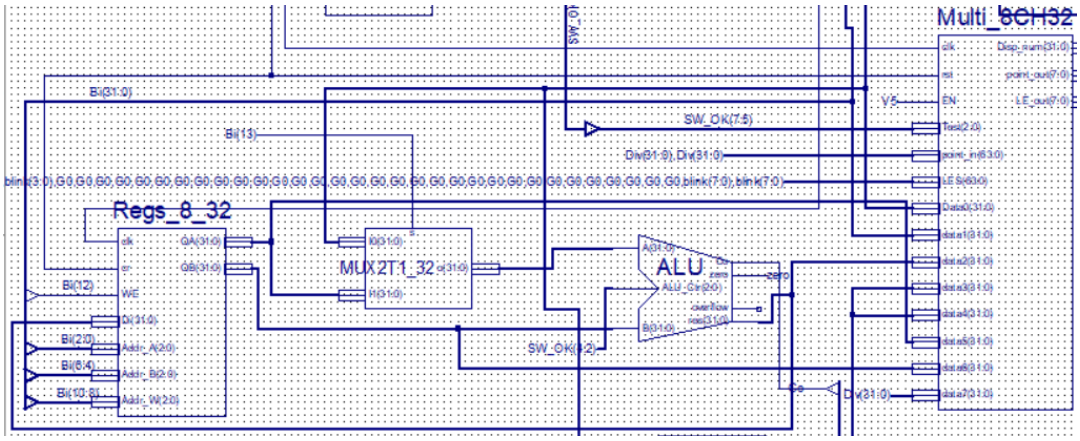
```
    reg [2:0] Addr_B;
    reg [2:0] Addr_W;

    // Outputs
    wire [31:0] QA;
    wire [31:0] QB;

    // Instantiate the Unit Under Test (UUT)
    Regs_8_32 uut (
        .clk(clk),
        .cr(cr),
        .WE(WE),
        .Di(Di),
        .Addr_A(Addr_A),
        .Addr_B(Addr_B),
        .Addr_W(Addr_W),
        .QA(QA),
        .QB(QB)
    );

    integer i=0;
    initial begin
    clk = 0;
    cr = 1;
    WE = 0;
    fork
        forever #20 clk<=~clk;
        #10 cr = 0;
        begin
        for(i=0; i<8; i=i+2)begin
            Addr_W <= i;
            Addr_A <= i;
            Addr_B <= i;
            Di <= 32'hAAAAAAA0 + i;
            #10; WE <= 1;
            #15; WE <= 0;
            #5;
            Addr_W <= i+1;
            Addr_A <= i+1;
            Addr_B <= i+1;
            Di <= 32'h55555551+i;
            #20; WE <= 1;
            #15; WE <= 0;
            #15;
        end

        WE = 0;
        for(i=0; i<8; i=i+1)begin
            #30 Addr_W <= i;
            Addr_A <= i;
            Addr_B <= i;
        end
        end
    join
    end

endmodule
```

Accessory Module

A **2-to-1 MUX** to select the data input into the A port of ALU module.

```
`timescale 1ns / 1ps
module MUX2T1_32(
    input s,
    input [31:0] I0,
    input [31:0] I1,
    output [31:0] o
    );
    assign o = s?I1:I0;

endmodule
```

Physical Test



Integrate with ALU module:

- We assign **BTN_OK(3)** to the clock signal to manually operate the register file
- **WE** signal is assigned with **Bi(12)**, **Addr_W** is assigned with **Bi(10:8)**, **Addr_B** is assigned with **Bi(6:4)**, **Addr_A** is assigned with **Bi(2:0)**.

Thus the **Bi** instruction can be illustrated as following:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x		s	WE	x	Addr_W			x	Addr_B			x	Addr_A		

The **UCF** file is presented below:

```
NET "clk_100mhz"    LOC=AC18    |   IOSTANDARD=LVCMOS18;
NET "clk_100mhz"    TNM_NET=TM_CLK;
TIMESPEC TS_CLK_100M = PERIOD "TM_CLK" 10 ns HIGH 50%;

NET "RSTN"  LOC=W13 |   IOSTANDARD=LVCMOS18;

NET "K_ROW[0]"  LOC=V17 |   IOSTANDARD=LVCMOS18;
NET "K_ROW[1]"  LOC=W18 |   IOSTANDARD=LVCMOS18;
NET "K_ROW[2]"  LOC=W19 |   IOSTANDARD=LVCMOS18;
NET "K_ROW[3]"  LOC=W15 |   IOSTANDARD=LVCMOS18;
NET "K_ROW[4]"  LOC=W16 |   IOSTANDARD=LVCMOS18;
```

```
12
13 NET "K_COL[0]"  LOC=V18 |   IOSTANDARD=LVCMOS18;
14 NET "K_COL[1]"  LOC=V19 |   IOSTANDARD=LVCMOS18;
15 NET "K_COL[2]"  LOC=V14 |   IOSTANDARD=LVCMOS18;
16 NET "K_COL[3]"  LOC=W14 |   IOSTANDARD=LVCMOS18;
17
18 NET "readn" LOC=U21 |   IOSTANDARD=LVCMOS33;
19 NET "RDY"   LOC=U22 |   IOSTANDARD=LVCMOS33;
20 NET "CR"    LOC=V22 |   IOSTANDARD=LVCMOS33;
21
22 NET "SEGCLK"    LOC=M24 |   IOSTANDARD=LVCMOS33;
23 NET "SEGCLR"    LOC=M20 |   IOSTANDARD=LVCMOS33;
24 NET "SEGDT" LOC=L24 |   IOSTANDARD=LVCMOS33;
25 NET "SEGEN" LOC=R18 |   IOSTANDARD=LVCMOS33;
26
27 NET "LEDCLK"    LOC=N26 |   IOSTANDARD=LVCMOS33;
28 NET "LEDCLR"    LOC=N24 |   IOSTANDARD=LVCMOS33;
29 NET "LEDDT" LOC=M26 |   IOSTANDARD=LVCMOS33;
30 NET "LEDEN" LOC=P18 |   IOSTANDARD=LVCMOS33;
31
32 NET "SW[0]" LOC=AA10    |   IOSTANDARD=LVCMOS15;
33 NET "SW[1]" LOC=AB10    |   IOSTANDARD=LVCMOS15;
34 NET "SW[2]" LOC=AA13    |   IOSTANDARD=LVCMOS15;
35 NET "SW[3]" LOC=AA12    |   IOSTANDARD=LVCMOS15;
36 NET "SW[4]" LOC=Y13 |   IOSTANDARD=LVCMOS15;
37 NET "SW[5]" LOC=Y12 |   IOSTANDARD=LVCMOS15;
38 NET "SW[6]" LOC=AD11    |   IOSTANDARD=LVCMOS15;
39 NET "SW[7]" LOC=AD10    |   IOSTANDARD=LVCMOS15;
40 NET "SW[8]" LOC=AE10    |   IOSTANDARD=LVCMOS15;
41 NET "SW[9]" LOC=AE12    |   IOSTANDARD=LVCMOS15;
42 NET "SW[10]"    LOC=AF12    |   IOSTANDARD=LVCMOS15;
43 NET "SW[11]"    LOC=AE8 |   IOSTANDARD=LVCMOS15;
44 NET "SW[12]"    LOC=AF8 |   IOSTANDARD=LVCMOS15;
45 NET "SW[13]"    LOC=AE13    |   IOSTANDARD=LVCMOS15;
46 NET "SW[14]"    LOC=AF13    |   IOSTANDARD=LVCMOS15;
47 NET "SW[15]"    LOC=AF10    |   IOSTANDARD=LVCMOS15;
48
49 NET "SEGMENT[0]"    LOC=AB22    |   IOSTANDARD=LVCMOS33;
50 NET "SEGMENT[1]"    LOC=AD24    |   IOSTANDARD=LVCMOS33;
51 NET "SEGMENT[2]"    LOC=AD23    |   IOSTANDARD=LVCMOS33;
52 NET "SEGMENT[3]"    LOC=Y21 |   IOSTANDARD=LVCMOS33;
53 NET "SEGMENT[4]"    LOC=W20 |   IOSTANDARD=LVCMOS33;
54 NET "SEGMENT[5]"    LOC=AC24    |   IOSTANDARD=LVCMOS33;
55 NET "SEGMENT[6]"    LOC=AC23    |   IOSTANDARD=LVCMOS33;
56 NET "SEGMENT[7]"    LOC=AA22    |   IOSTANDARD=LVCMOS33;
57
58 NET "AN[0]" LOC=AD21    |   IOSTANDARD=LVCMOS33;
59 NET "AN[1]" LOC=AC21    |   IOSTANDARD=LVCMOS33;
60 NET "AN[2]" LOC=AB21    |   IOSTANDARD=LVCMOS33;
61 NET "AN[3]" LOC=AC22    |   IOSTANDARD=LVCMOS33;
62
63 NET"LED[0]"LOC=W23  | IOSTANDARD=LVCMOS33 ;#D1
64 NET"LED[1]"LOC=AB26 | IOSTANDARD=LVCMOS33 ;#D2
65 NET"LED[2]"LOC=Y25  | IOSTANDARD=LVCMOS33 ;#D3
66 NET"LED[3]"LOC=AA23 | IOSTANDARD=LVCMOS33 ;#D4
67 NET"LED[4]"LOC=Y23  | IOSTANDARD=LVCMOS33 ;#D5
68 NET"LED[5]"LOC=Y22  | IOSTANDARD=LVCMOS33 ;#D6
69 NET"LED[6]"LOC=AE21 | IOSTANDARD=LVCMOS33 ;#D7
```

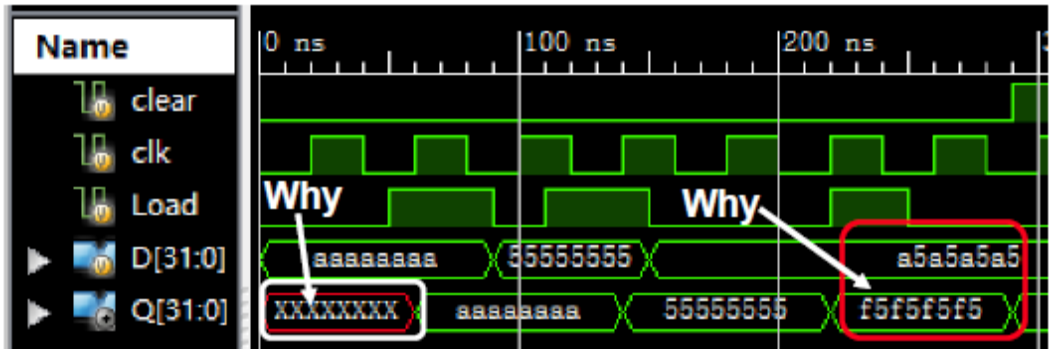
```
70 NET"LED[7]"LOC=AF24  | IOSTANDARD=LVCMOS33 ;#D8
```

5. Experiment Results and Analyses

Simulation Test

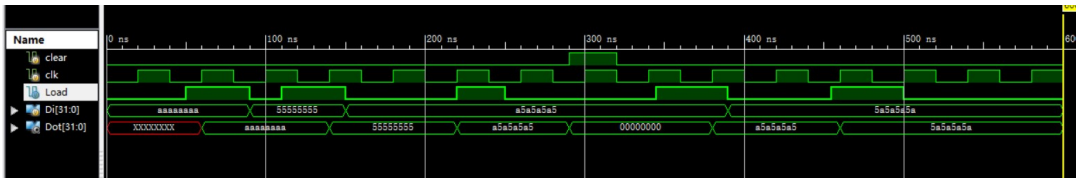
32-bit Register

This is the simulation result for 32-bit register implemented by structural description. Yet I found some difference between my simulation result and out professor+ Qingsong Shi's. It seems that he witnessed a confliction when both **clk** and **Load** signal are undergoing a rising edge, making **Dot** a pretty weird value:



weird value

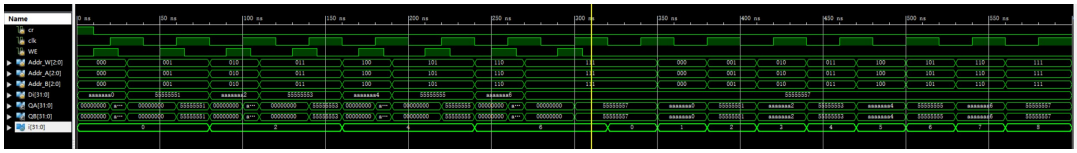
And I also found that the simulation result tends to be different when the registers are implemented by various description. Here's the result when the register is implemented by behavioral description. Clearly, we see the value of **Dot** is loaded by $a5a5a5a5$ at **220ns**:



simulation result for behavioral description

My conjecture: Since **Maintain Block D Flip-Flop** has been used as the basic architecture of the structural description of the register, it will treat time signal more strictly. Therefore, the register rejects the **load signal** when processing to the rising edge of the **clk** signal, even both signal is activated at exactly the same time. While the same situation is not treated so strictly in behavioral description, which caused the simulator to accept the load signal when processing the clk signal.

8×32 Register File



This time the register perfectly passed all the simulation test points. We can thus proceed to physical test.

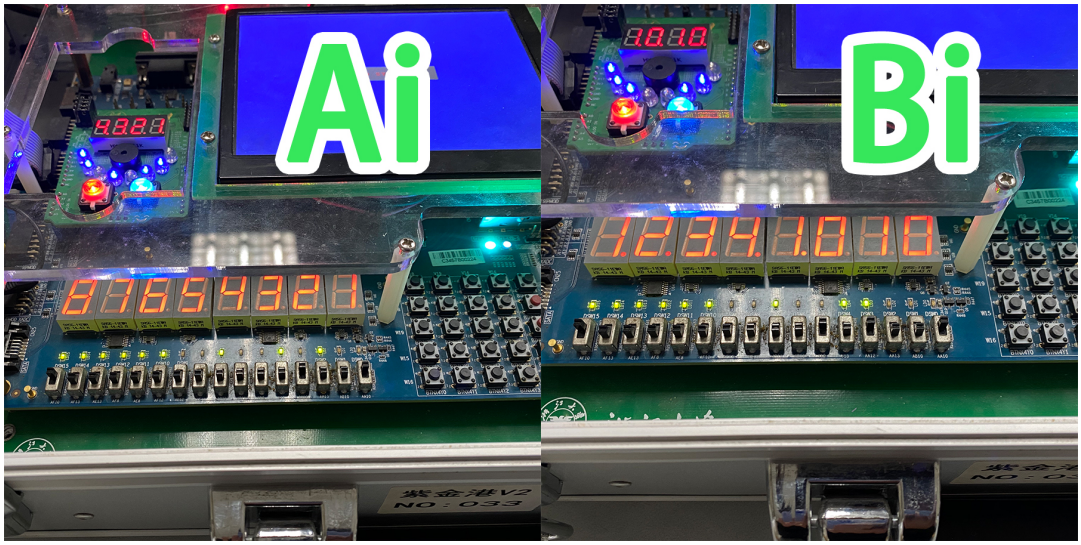
Physical Test

Step 1

First, check the value in all the 8 registers. The value in each register is 0.

Then adjust **Ai** to **32'h87654321** and **Bi** to 16'h1010. The behavior is that **add** the value of **Ai** with the value in **register[0]** and transfer the result to **register[0]**, while assign **register[0]** to **Port_A** and **register[1]** to **Port_B**.

Ai	Bi	SW[4:2]
32'h87654321	16'h1010	010



Ai & Bi

The expected performance is that the value in **Port_A** is **32'h87654321** and the value in **Port_B** is all 0.

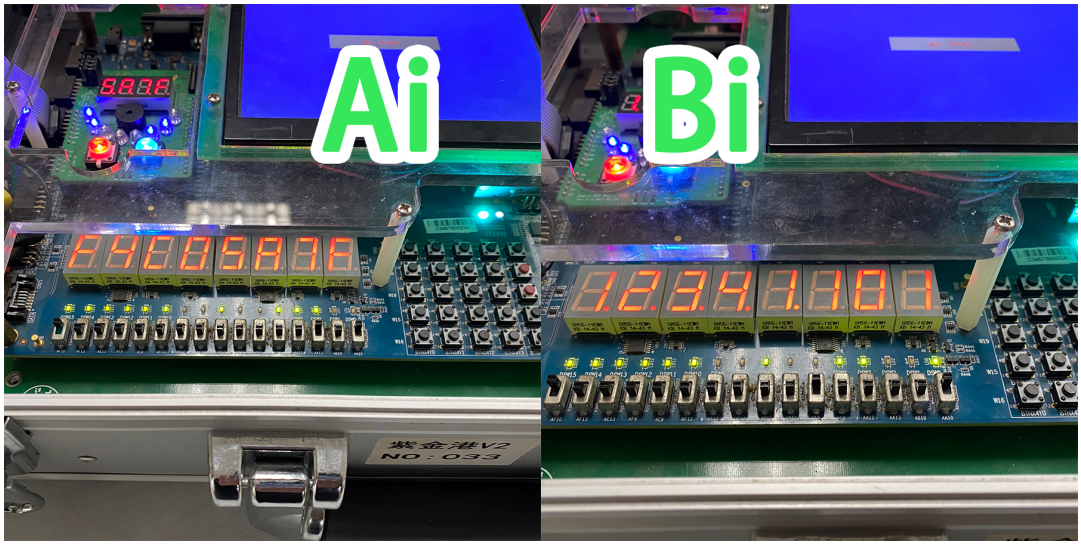
result

The result has been verified.

Step 2

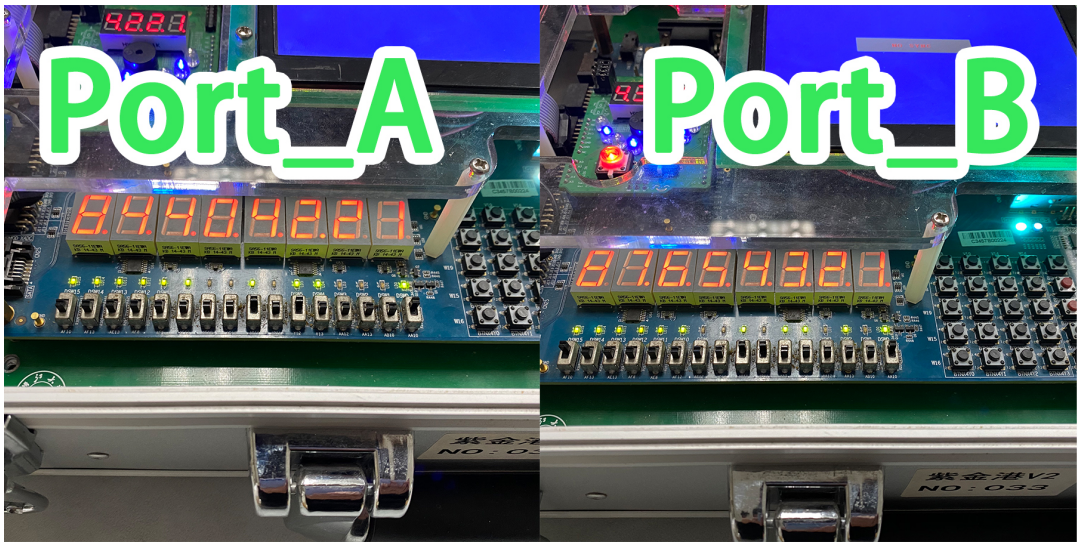
This time we **AND** the value from **Ai** with the value in **register[0]** and transfer the value to **register[1]**. The parameter is set as the following sheet.

Ai	Bi	SW[4:2]
32'hF4C05A7F	16'h1101	000



The expected result is that the value in **Port_A** is 32'h84404221 and the value in **Port_B** is 32'h87654321.

- *F4C05A7F AND 87654321 = 84404221*



result

And the result is verified.

Step 3

We do subtraction between the value from **register[0]** and the value from **register[1]** and transfer the value to **register[2]**.

Bi	SW[4:2]
16'h3210	110

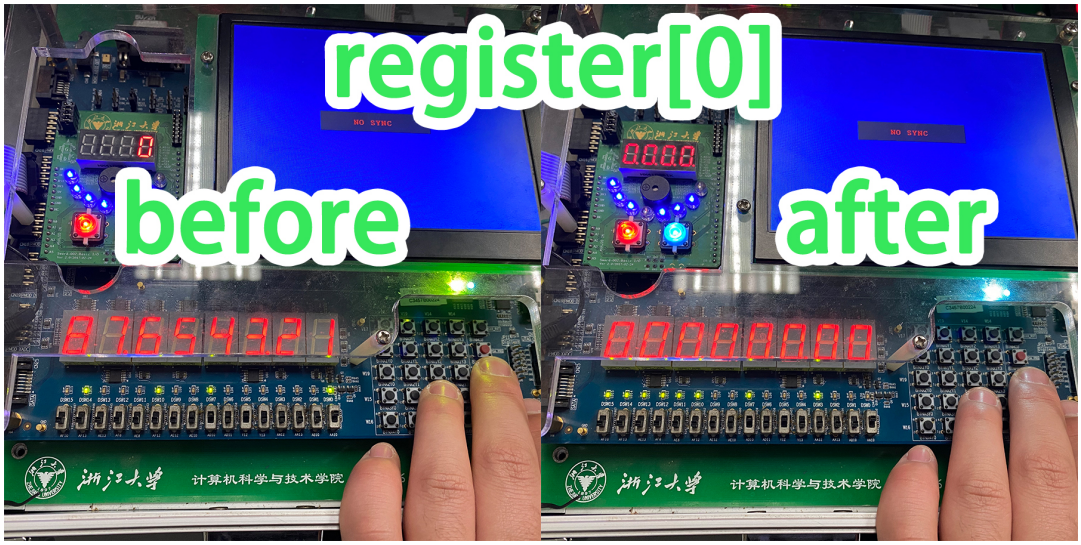
The expected result in **register[2]** is 32'h03250100

- 87654321 - 84404221 = 03250100

Thus the result is also verified.

Step 4

We verified the **clear** function. Keep pressing "rstn" button for several seconds, and the value in **register[0]** is cleared to 0.



All functions are verified perfectly. ✨

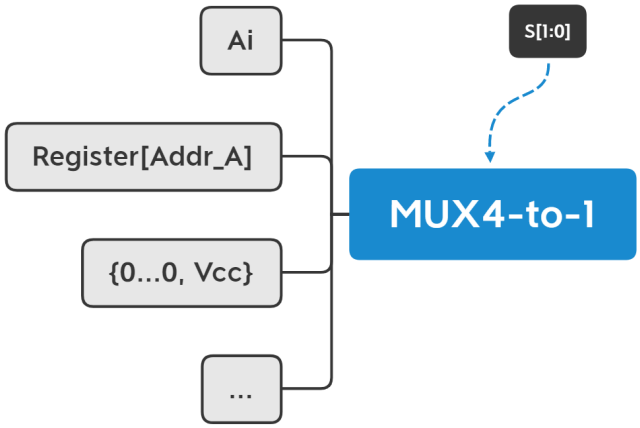
6. Discussion and Conclusion

Feelings

> Without doubt this experiment is super difficult. The hard core is to understand the working mechanism of RTL, which involves both register file and ALU arithmetic module.
>
> Much effort spent, I figure out a way of understanding the logic function in a natural way:
>
> - Perform a `ALU-mode` between `Ai` and `register[Addr_B]` and transfer the result into `register[Addr_W]` when `s=0` & `WE=1`
> - Perform a `ALU-mode` between `register[Addr_A]` and `register[Addr_B]` and transfer the result into `register[Addr_W]` when `s=1` & `WE=1`
> - Watch the value in `register[Addr_A]` and `register[Addr_B]` when `WE=0`

Thinking question

- How to add 1 cumulatively without entering 1 to **Ai**
  - A simple way: expand the **MUX2to1** to **4 input**, the lowest bit of one input is connected to **Vcc**, which is a constant integer 1.



  - We can also use the counter to add 1 cumulatively. Yet this will be more difficult to revise than the above way.