Lab03 存储器RAM

PB15000102 王嵩超

实验内容

- 学习使用ISE的IP核
- 学习使用Xilink FPGA内的RAM资源
    - 例化一个简单双端口的RAM (32bit x 64)
    - 使用coe文件对RAM进行初始化

实验功能要求

综合利用三次实验的结果，完成以下功能：

- 从ram中0地址和1地址读取两个数， 分别赋给reg0和reg1
- 利用第二次实验的结果(ALU+Regfile)进行斐波拉契运算，运算结果保存在对应的寄存器
- 运算结果同时保存在对应的ram地址中，即ram[0]<----->reg0, ram[1]<----->reg1, ram[2]<----->reg2,……

实验设计

- 实现一个control模块，完成整个运算的控制。

  control应包含RAM的输出、ALU的输出、寄存器堆的输出三者作为输入信号；输出信号作为RAM的控制、输入信号，ALU的输入信号、寄存器输入、使能信号。也可把内部的状态寄存器作为输出，方便调试。

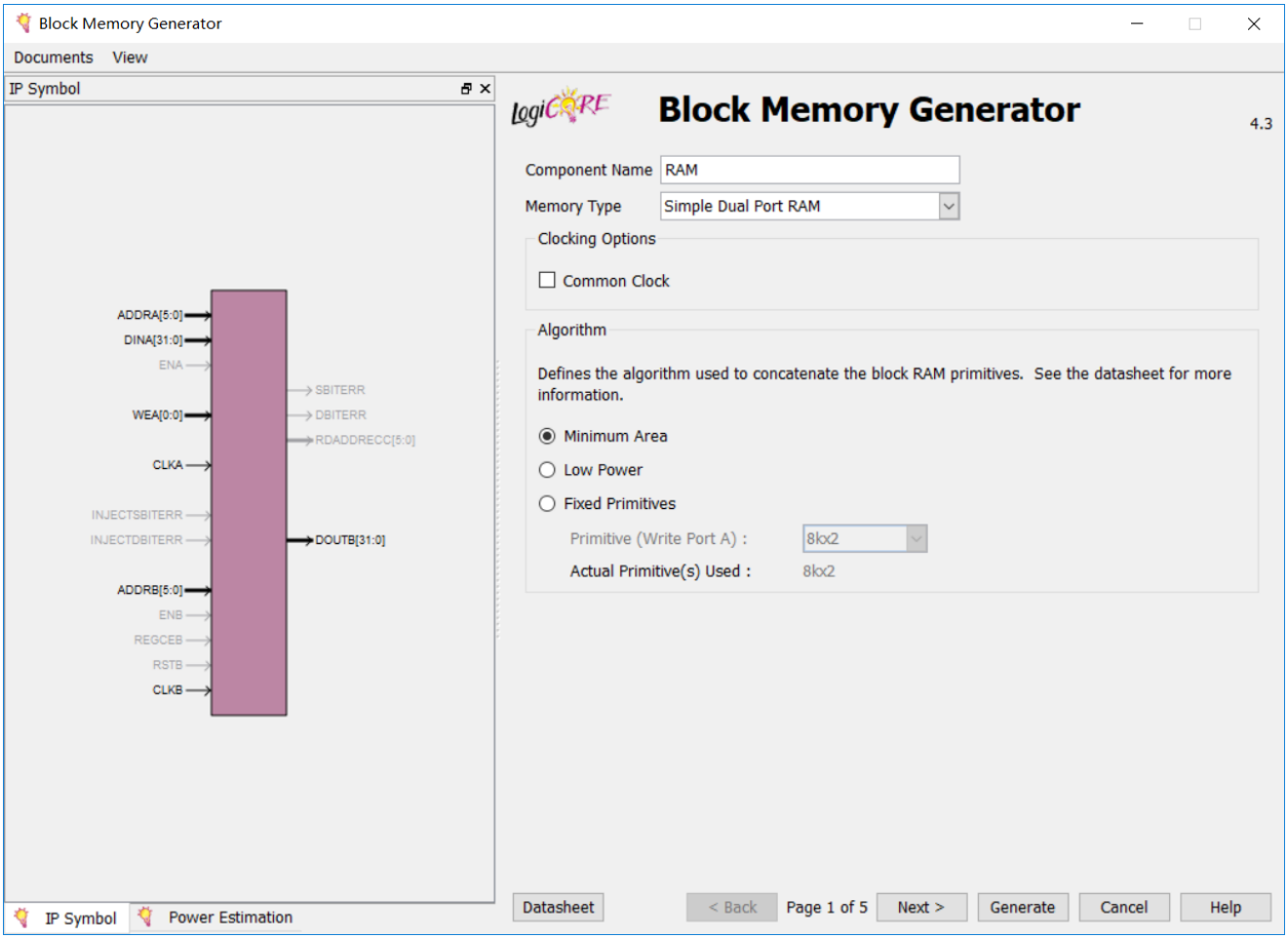
- 实现一个顶层模块Top
    - 调用Ram模块
    - 调用RegFile
    - 调用ALU完成加法运算
    - 调用control模块，完成运算控制

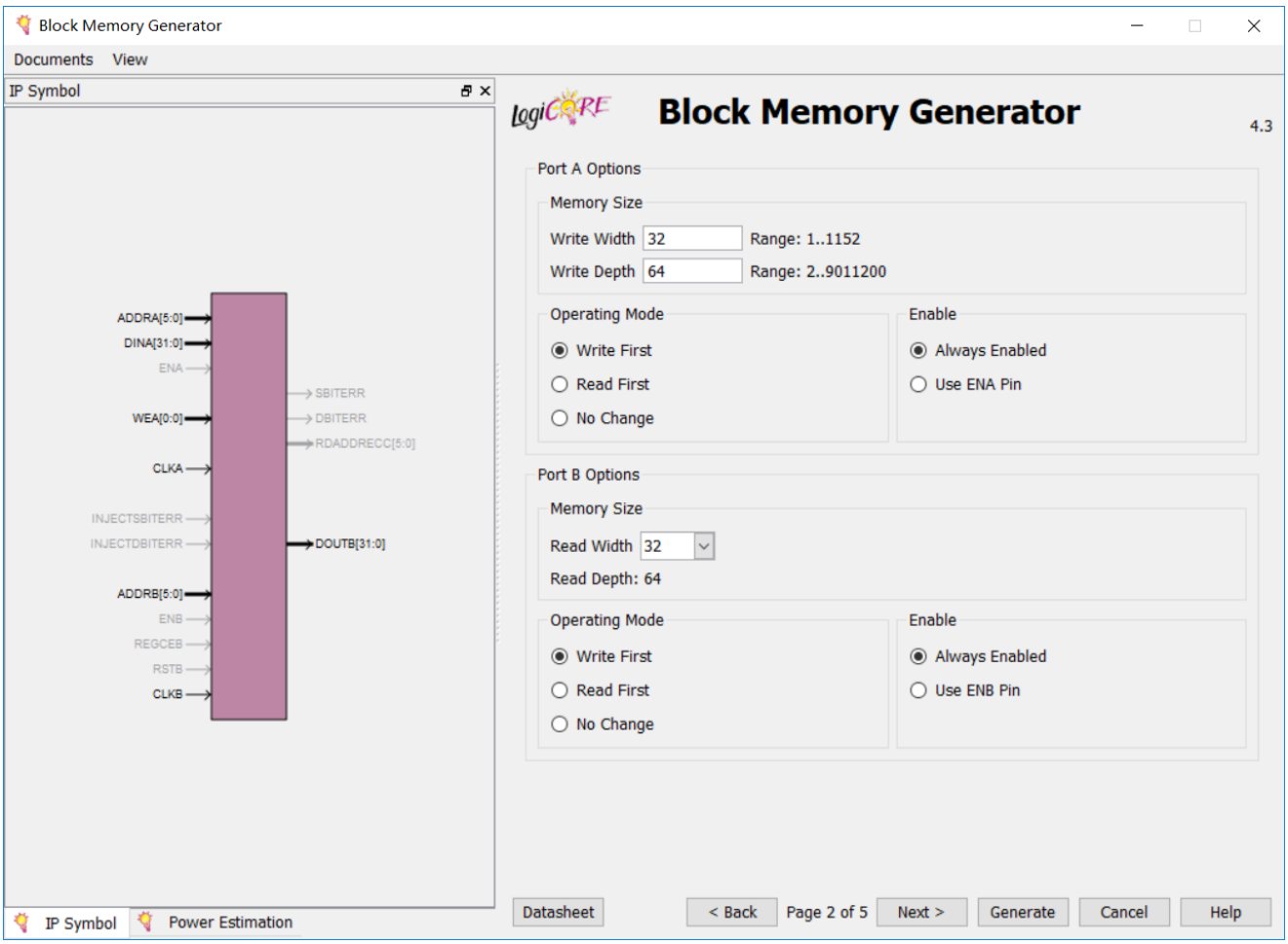
    > 本实验接线较多，调用繁琐。在写top和control模块的调用时应格外仔细。因为漏写接口并不会导致报错！

- 加入仿真模块simu，对top模块进行仿真

RAM的例化

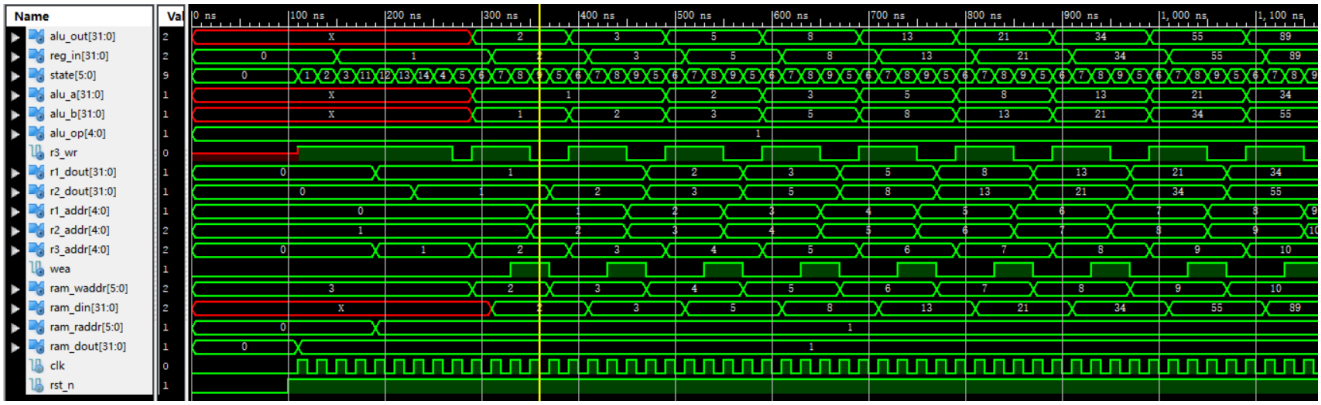
主要参数如下，其他保持默认即可：


Block Memory Generator
Documents View
IP Symbol
ADDRA[5:0]
DINA[31:0]
ENA
WEA[0:0]
CLKA
INJECTSBITERR
INJECTDBITERR
ADDRB[5:0]
ENB
REGCEB
RSTB
CLKB
SBITERR
DBITERR
RDADDRECC[5:0]
DOUTB[31:0]
IP Symbol
Power Estimation
LogiCORE
Block Memory Generator
4.3
Component Name RAM
Memory Type Simple Dual Port RAM
Clocking Options
Common Clock
Algorithm
Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.
Minimum Area
Low Power
Fixed Primitives
Primitive (Write Port A) : 8kx2
Actual Primitive(s) Used : 8kx2
Datasheet
< Back
Page 1 of 5
Next >
Generate
Cancel
Help


Block Memory Generator
Documents View
IP Symbol
ADDRA[5:0]
DINA[31:0]
ENA
WEA[0:0]
CLKA
INJECTSBITERR
INJECTDBITERR
ADDRB[5:0]
ENB
REGCEB
RSTB
CLKB
SBITERR
DBITERR
RDADDRECC[5:0]
DOUTB[31:0]
IP Symbol
Power Estimation
LogiCORE
Block Memory Generator
4.3
Port A Options
Memory Size
Write Width 32 Range: 1..1152
Write Depth 64 Range: 2..9011200
Operating Mode
Write First
Read First
No Change
Enable
Always Enabled
Use ENA Pin
Port B Options
Memory Size
Read Width 32
Read Depth: 64
Operating Mode
Write First
Read First
No Change
Enable
Always Enabled
Use ENB Pin
Datasheet
< Back
Page 2 of 5
Next >
Generate
Cancel
Help

其中Write Width代表每个寄存器的宽度、Write Depth代表有多少个寄存器。

仿真结果



存储器：

	0	1	2	3
0x0	1	1	2	3
0x4	5	8	13	21
0x8	34	55	89	144
0xC	233	377	610	987
0x10	1597	2584	4181	6765
0x14	10946	17711	28657	46368
0x18	75025	121393	196418	317811
0x1C	514229	832040	1346269	2178309
0x20	0	0	0	0
0x24	0	0	0	0
0x28	0	0	0	0
0x2C	0	0	0	0
0x30	0	0	0	0
0x34	0	0	0	0
0x38	0	0	0	0
0x3C	0	0	0	0

寄存器：

	0	1	2	3
0x1F	2178309	1346269	832040	514229
0x1B	317811	196418	121393	75025
0x17	46368	28657	17711	10946
0x13	6765	4181	2584	1597
0xF	987	610	377	233
0xB	144	89	55	34
0x7	21	13	8	5
0x3	3	2	1	1

总结

本实验状态数较多，主要是未多考虑存储器取数存数的性质，用大量周期来等待存储器的写入和读取，减慢了执行速度。

由于存储器在读地址变化后，会在时钟上升沿后的晚些时候才使输出变化，得在下一周期才能取得输出。

为了减少周期数，可采用以下几种办法：

1. 在状态机中精心安排信号的控制时间，尽量提前在操作前发出控制信号，做到在取数周期时，RAM的output端口已经有结果了。
2. 多采用组合逻辑，组合逻辑响应较快，不用等到下一个时钟周期才开始行动。

   > 本来按CPU的设计，很多接线（比如ALU与Reg、ALU与MEM两者的选择）都是组合逻辑，只需让状态机产生控制信号。但这几次实验为了方便，包括我在内很多人都把REG和MEM的输入端口设为寄存器，每次用时就手动将ALU_input赋给MEM或Reg，这样显然就迟钝很多。

3. 采用分频时钟。一种方式是让RAM响应时钟的上升沿和下降沿。上一周期改变地址，在上周期的后半时间就能使RAM读取，在下一周期就能取数。

源代码

myreg.v

```
//r1 r2 read; r3 write
module REG_FILE(
input                        clk,
input                        rst_n,
input        [4:0]           r1_addr,
input        [4:0]        r2_addr,
input        [4:0]        r3_addr,
input        [31:0]       r3_din,
input                        r3_wr,
output reg [31:0]         r1_dout,
output reg [31:0]         r2_dout
);

reg [31:0] R[31:0];
integer i;
always@(posedge clk,negedge rst_n)
begin
    if(~rst_n)
    begin
        for ( i=0; i<32; i=i+1 )  R[i] <= 0;
        r1_dout <=0;
        r2_dout <=0;
    end
    else
    begin
        r1_dout = R[r1_addr];
        r2_dout = R[r2_addr];
           if(r3_wr) //write enabled
        R[r3_addr] = r3_din;
    end
end
endmodule

```

ALU.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:53:21 03/20/2017
// Design Name:
// Module Name:    ALU
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ALU(
    input signed [31:0] alu_a,
    input signed [31:0] alu_b,
    input [4:0] alu_op,
    output reg [31:0] alu_out
    );
parameter A_NOP = 5'h00; //空运算
parameter A_ADD = 5'h01; //符号加
parameter A_SUB = 5'h02; //符号减
parameter A_AND = 5'h03; //与
parameter A_OR  = 5'h04; //或
parameter A_XOR = 5'h05; //异或
parameter A_NOR = 5'h06; //或非

always@(*)
begin
    case(alu_op)
     A_NOP:
     begin

     end
     A_ADD:
     begin
         alu_out <= alu_a + alu_b;
     end
     A_SUB:
     begin
         alu_out <= alu_a - alu_b;
     end
     A_AND:
     begin
         alu_out <= alu_a & alu_b;
     end
```

```
     A_OR:
     begin
         alu_out <= alu_a | alu_b;
     end
     A_XOR:
     begin
         alu_out <= alu_a ^| alu_b;
    end
     A_NOR:
     begin
         alu_out <= ~(alu_a | alu_b);
     end
     endcase
end
endmodule

```

control.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    13:32:24 03/28/2017
// Design Name:
// Module Name:    control
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module control(
input clk,
input rst_n,
input [31:0] ram_dout, //from RAM
input [31:0] alu_out,
input [31:0]   r1_dout,
input [31:0]    r2_dout,

output reg [5:0] state, //no need to output, just made for convenience when debugging
output reg  [4:0]                r1_addr,
output reg  [4:0]               r2_addr,
output reg  [4:0]               r3_addr,
output reg [31:0] alu_a,
output reg [31:0] alu_b,
output reg [4:0] alu_op,
output reg   r3_wr,

//for RAM
output reg wea,
output reg [5:0] ram_waddr,
output reg [31:0] ram_din,
output reg [5:0] ram_raddr,
output reg [31:0] reg_in
    );
//reg   [31:0]            r3_din; //substituted by alu_out.
//registers can only be set values via ALU.
    always@(posedge clk,negedge rst_n)
    begin
        if(~rst_n)
          begin
        state <= 0;
          r1_addr <= 0;
          r2_addr <= 1;
```

```
54            r3_addr <= 0;
55            alu_op <= 5'h01;
56            //ram
57            ram_raddr <= 0;
58            reg_in <= 0;
59            ram_waddr <= 3;
60            wea <= 0;
61            end
62          else
63          begin
64              case(state)
65              0://initialize I R[0]=RAM[0]
66              begin
67                          r3_wr <= 1;
68                          state <= 1;
69              end
70                    1://waiting for MEM fetch
71                    begin
72                        state <= 2;
73                    end
74                    2://assign to reg_in register
75                    begin
76                        reg_in <= ram_dout;
77                         state <= 3;
78                    end
79                    3://waiting for writing back
80                    begin
81                        state <= 11;
82                    end
83                    11://initialize II R[1]=RAM[1]
84                    begin
85                        r3_addr <= 1;
86                         ram_raddr <= 1;
87                         state <= 12;
88                    end
89                    12://waiting for MEM fetch
90                    begin
91                        state <= 13;
92                    end
93                    13://assign to reg_in register
94                    begin
95                        reg_in <= ram_dout;
96                         state <= 14;
97                    end
98                    14://waiting for writing back
99                    begin
100                       state <= 4;
101                   end
102                   4://turn off writing
103                   begin
104                       r3_wr <= 0;
105                        state <= 5;

106                   end
```

```
                5://before calculate
                begin
                    r3_addr <= r2_addr + 1; //adjust the writing port
                     ram_waddr <= r2_addr + 1;
                     r3_wr <= 1;//set write enable

                     alu_a <= r1_dout; //R[n-1]
                     alu_b <= r2_dout; //R[n-2]
                     state <= 6;
                end
                6://calculating
                begin
                    reg_in <= alu_out;//this need to be delayed
                     ram_din <= alu_out;
                    state <= 7;
                end
                7://assign and judge
                begin

                     wea <= 1; //set MEM write enable
                     if(r3_addr ==31) //reach the end
                     state <= 10;
                     else state <= 8;
                end
                8://move and wait when storing to MEM
                begin
                    r3_wr <= 0;
                    r1_addr <= r1_addr + 1;
                     r2_addr <= r2_addr + 1;
                     state <= 9;
                end
                9://waiting This is important, because r1_out and r2_out take time to
output.
                begin
                    wea <= 0;
                    state <= 5;
                     ram_waddr <= ram_waddr + 1;
                end
                10://finish
                begin
                    state <= 10;
                end
                default:
                begin
                    state <= 0;
                end
            endcase
        end
    end
endmodule

```