数字部件设计Project 第一部分——功能元件

实现单周期CPU的功能元件可分为三类：

1. ALU类

   这类元件都是组合逻辑电路，根据输入进行一定的运算或逻辑操作，将结果输出，有以下元件：

   - alu：算术逻辑单元
   - adder：32位加法器
   - mux2：二选一复用器
   - sl2：两位左移器

2. Register类

   这类元件都是时序电路，实现了CPU中的寄存器的功能，可在一个周期内进行read和write，有以下元件：

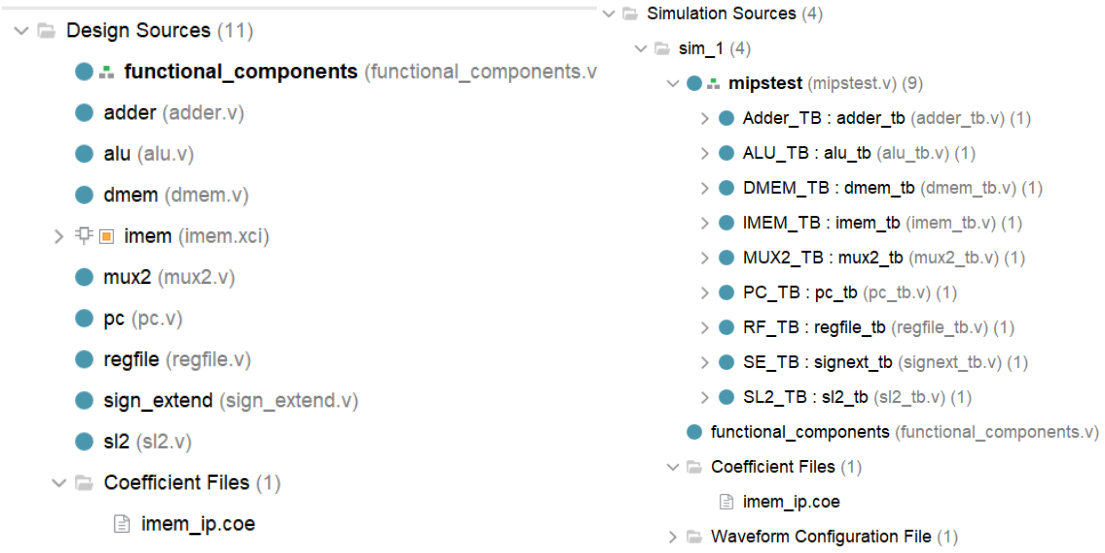
   - regfile：寄存器堆
   - pc：程序计数器

3. Memory类

   这类元件都是时序电路，实现了计算机中的RAM和ROM，可在一个周期内根据字节地址或字地址访问内存，进行load或save。

   - imem：指令内存，ROM
   - dmem：数据内存，RAM

design和simulation的目录结构如下



目录结构

仿真的scope层次如下图所示：

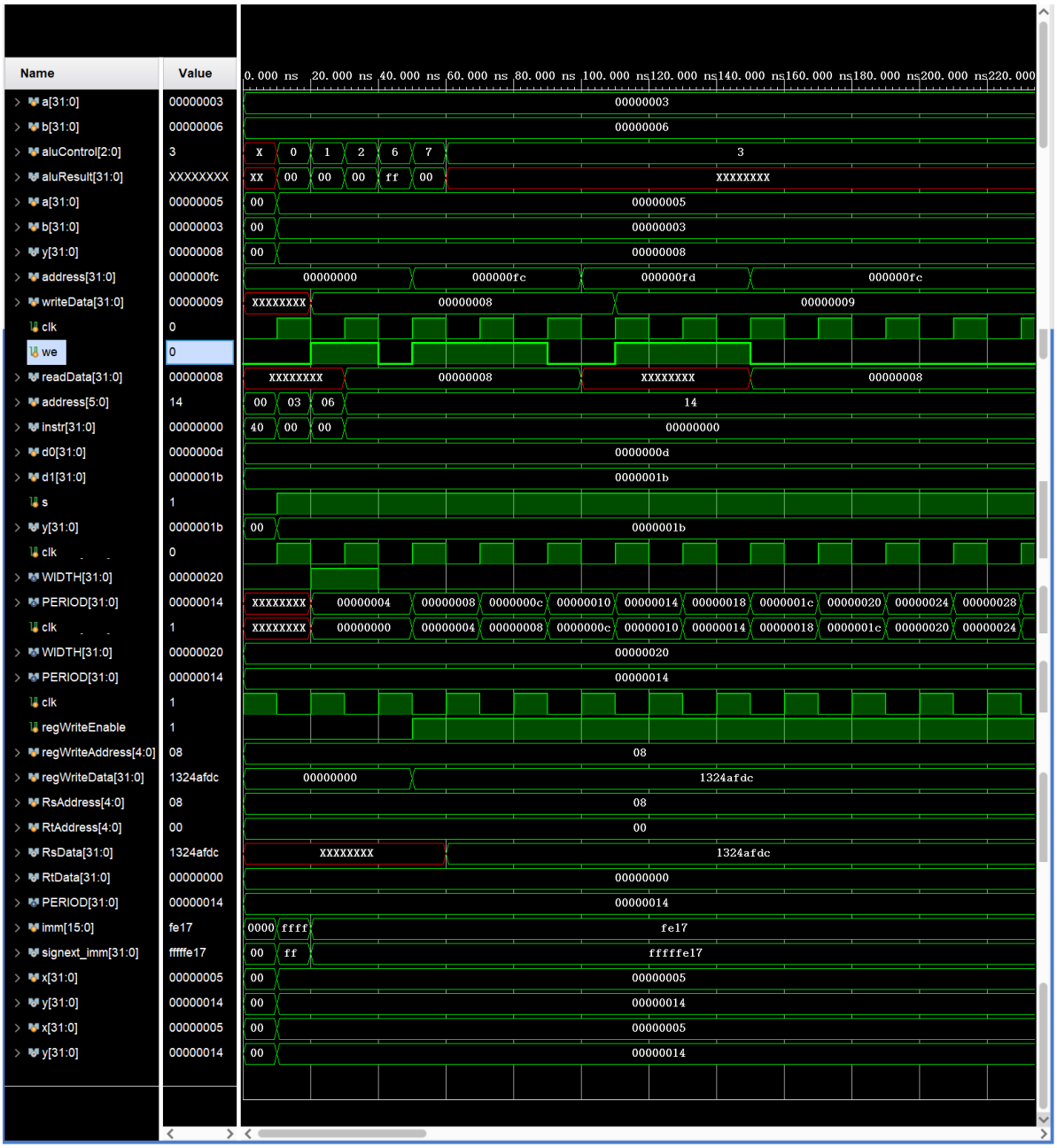
Scope × Sources

Name	Design Unit	Block Type
mipstest	mipstest	Verilog Module
DMEM_TB	dmem_tb	Verilog Module
DMEM	dmem	Verilog Module
IMEM_TB	imem_tb	Verilog Module
IMEM	imem	Verilog Module
[illegible]	[illegible]	[illegible]
IMEM	imem	Verilog Module
inst	dist_mem_gen_v8_0_13	Verilog Module
MUX2_TB	mux2_tb	Verilog Module
MUX2	mux2	Verilog Module
PC_TB	pc_tb	Verilog Module
PC	pc	Verilog Module
RF_TB	regfile_tb	Verilog Module
regFile	regfile	Verilog Module
SE_TB	signext_tb	Verilog Module
SignExt	sign_extend	Verilog Module
SL2_TB	sl2_tb	Verilog Module
SL2	sl2	Verilog Module
glbl	glbl	Verilog Module

mipstest为顶层测试文件，我用它实例化所有单独的仿真测试模块，并在仿真的时候将子模块的内部信号也加到Wave Window中进行显示。代码如下：

```
`timescale 1ns / 1ps

module mipstest();
    adder_tb Adder_TB();
    alu_tb ALU_TB();
    dmem_tb DMEM_TB();
    imem_tb IMEM_TB();
    mux2_tb MUX2_TB();
    pc_tb PC_TB();
    regfile_tb RF_TB();
    signext_tb SE_TB();
    sl2_tb SL2_TB();
endmodule
```

总的波形图如下：



下面是每个元件的介绍，以及对每个模块分别进行仿真的结果。

alu

功能：

32位的AND、OR、ADD、SUB、SLT、RETURN 0等操作

用途：

CPU中的ALU，用于R_type指令的运算结果的计算、lw和sw等指令的基址偏移量寻址、beq等指令的条件结果的计算。

实现思路：

根据下表，通过verilog的case语法实现。

aluControl	Meaning
000	And
001	Or
010	Add
011	Not Used
100	Return 0(NOP)
101	Not Used
110	Subtract
111	SLT

实现代码：

```
`timescale 1ns / 1ps

module alu(
input       [31:0]  a, b,
input       [2:0]   aluControl,
output  reg [31:0]  aluResult
    );
    always @(*) begin
        case(aluControl)
            3'b000: aluResult = a & b;  //AND
            3'b001: aluResult = a | b;  //OR
            3'b010: aluResult <= a + b; //ADD
            3'b110: aluResult <= a - b; //SUBTRACT
            3'b111: aluResult = (a < b) ? 1 : 0; //SLT
            3'b100: aluResult = 0; //RETURN 0
            default:aluResult = 32'bx;
        endcase
    end
endmodule
```

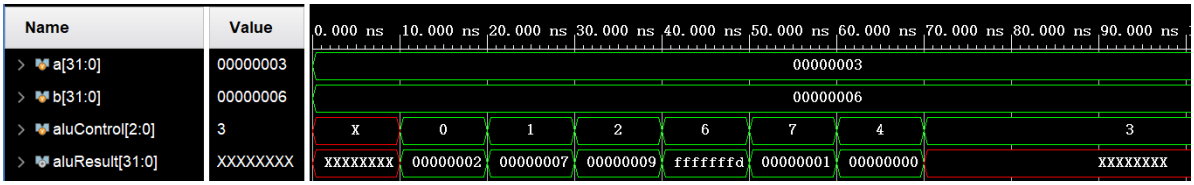
仿真代码：

```
`timescale 1ns / 1ps

module alu_tb();
    reg  [31:0]  a, b;
    reg  [2:0]   aluControl;
    wire [31:0]  aluResult;
    //实例化
    alu ALU(.a(a), .b(b), .aluControl(aluControl), .aluResult(aluResult));

    initial begin
        a = 3;
        b = 6;
        #10;
        aluControl = 3'b000;
        #10;
        aluControl = 3'b001;
        #10;
        aluControl = 3'b010;
        #10;
        aluControl = 3'b110;
        #10;
        aluControl = 3'b111;
        #10;
        aluControl = 3'b100;
        #10;
        aluControl = 3'b011;
    end
endmodule
```

测试功能：

- AND
- OR
- ADD
- SUB
- SLT
- RETURN 0
- 定义之外的值返回x

仿真结果：



波形图说明：

刚开始aluControl没有初始化，所以结果为x。然后给aluControl、a、b分别赋值，a = 0100，b = 1010，故a & b = 0010 = 2，a | b = 1011 = 7，a + b = 9，a - b = -3 = fffffffd，a < b所以结果被置为1，aluControl = 4时返回0，最后给aluControl赋了一个没有对应操作的值，结果为X。

adder

功能：

32位加法

用途：

用于PC加4、PC相对寻址、伪直接寻址。

实现代码：

```
`timescale 1ns / 1ps

module adder(
input   [31:0] a, b,
output  [31:0] y
    );
    assign y = a + b;
endmodule
```

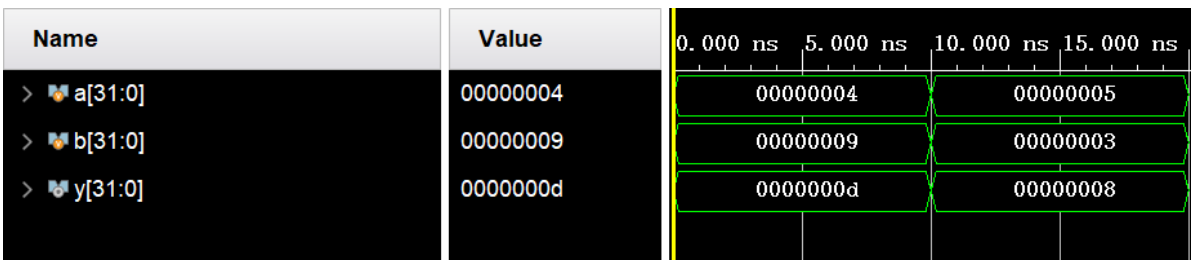
仿真代码：

```
`timescale 1ns / 1ps

module adder_tb();
    reg   [31:0] a, b;
    wire  [31:0] y ;
    adder Adder(.a(a), .b(b), .y(y));
    initial begin
        a = 4;
        b = 9;
        #10;
        a = 5;
        b = 3;
        #10;
    end
endmodule
```

测试功能：

- 加法

仿真结果：



波形图说明：

4 + 9 = 13 = d

5 + 3 = 8

sign_extend

功能：

符号扩展立即数

用途：

将I_type的16位立即数根据符号位扩展到32位

实现代码：

```
module sign_extend(
input   [15:0]  x,
output  [31:0]  y
    );
    assign y = {{16{x[15]}},x};
endmodule
```

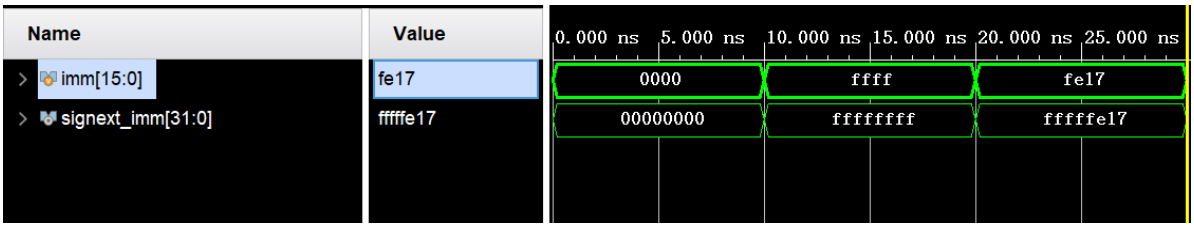
仿真代码：

```
`timescale 1ns / 1ps

module signext_tb();
    reg     [15:0]  imm;
    wire    [31:0]  signext_imm;
    //实例化
    sign_extend SignExt(.x(imm), .y(signext_imm));
    initial begin
        imm = 0;
        #10;
        imm = 16'hffff;
        #10;
        imm = 16'hfe17;
    end
endmodule
```

测试功能：

- 将16位立即数符号扩展为32位

仿真结果：



波形图说明：

首先输入0，输出为0，然后输入ffff以及fe17，最高位为1，扩展后高位全部为1。

mux2

功能：

二选一

用途：

用于选择ALU操作数的来源、写回寄存器的地址和数据、PC'的来源。

实现代码：

```
module mux2
#(parameter WIDTH = 8)
(
input   [WIDTH-1:0] d0, d1,
input               s,
output  [WIDTH-1:0] y
    );
    assign y = s ? d1 : d0;
endmodule
```

仿真代码：

```
`timescale 1ns / 1ps

module mux2_tb();
    reg   [32-1:0] d0, d1;
    reg            s;
    wire  [32-1:0] y;
    mux2 #(32) MUX2(.d0(d0), .d1(d1), .s(s), .y(y));
    initial begin
        d0 = 13;
        d1 = 27;
        s = 0;
        #10;
        s = 1;
    end
endmodule
```

仿真结果：

波形图说明：

s先为0，后为1，所以先选了d0，后选了d1。

sl2

功能：

左移两位

用途：

用于PC相对寻址和伪直接寻址中将字的地址转化为字节地址。

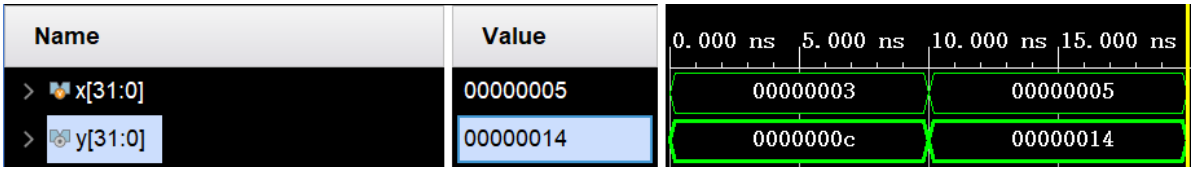
实现代码：

```
module sl2(
input   [31:0]  x,
output  [31:0]  y
    );
    assign y = {x[29:0],2'b00};
endmodule
```

仿真代码：

```
`timescale 1ns / 1ps

module sl2_tb();
    reg   [31:0]  x;
    wire  [31:0]  y;
    sl2 SL2(.x(x), .y(y));
    initial begin
        x = 3;
        #10;
        x = 5;
    end
endmodule
```

仿真结果：



波形图说明：

y = 4*x

pc

功能：

每个时钟上升沿将PC'加载到内部寄存器并作为PC在下一个时钟周期输出，程序开始执行时要先用reset将PC初始化为0。

用途：

CPU中的程序计数器，输出当前执行指令的地址，并在时钟上升沿加载下一条指令的地址，进而在下一个周期输出。

实现代码：

```
module pc
#(parameter WIDTH = 32)  //指令地址的宽度，默认为32位
(
input       clk, reset,
input       [WIDTH-1:0] nextPC,
output  reg [WIDTH-1:0] currPC
    );
    always @(posedge clk, posedge reset)
        if(reset)   currPC <= 0;
        else        currPC <= nextPC;
endmodule
```

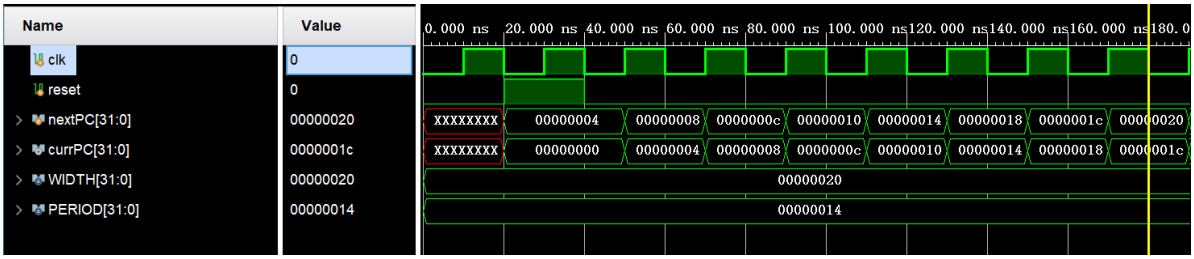
仿真代码：

```
`timescale 1ns / 1ps
module pc_tb();
    parameter WIDTH = 32;
    reg     clk, reset;
    reg     [WIDTH-1:0] nextPC;
    wire    [WIDTH-1:0] currPC;
    //实例化
    pc #(WIDTH) PC(.clk(clk), .reset(reset), .nextPC(nextPC), .currPC(currPC));
    //初始化
    initial begin
        reset = 0;
        clk = 0;
        #20;
        reset = 1;
        #20;
        reset = 0;
    end
    parameter PERIOD = 20;
    //产生时钟周期
    always begin
        #(PERIOD/2);
        clk = ~clk;
    end
    //nextPC = currPC + 4
    always @(currPC)begin
        nextPC <= currPC + 4;
    end
endmodule
```

测试功能：

- reset
- load nextPC at posedge

仿真结果：



波形图说明：

刚开始currPC和nextPC都没有值，故都是X，20ns时令reset=1初始化currPC，然后仿真代码将currPC+4赋给nextPC作为输入，在下一个时钟周期被load进PC中的寄存器，并作为currPC输出，循环往复，currPC和nextPC都不断加4，并且nextPC始终比currPC大4。

regfile

功能：

可根据0-31的寄存器号进行read和write，其中寄存器0的内容总是0。

用途：

CPU中的0~31共32个通用寄存器，统称寄存器堆（register file）。

实现代码：

```
module regfile(
input           clk,
input           regWriteEnable, //写入使能
input   [4:0]   regWriteAddress,//写入地址，深度为32故只要5位
input   [31:0]  regWriteData,
input   [4:0]   RsAddress,      //Rs寄存器，源操作数寄存器号
input   [4:0]   RtAddress,      // Rt寄存器
output  [31:0]  RsData,
output  [31:0]  RtData
    );
    reg [31:0] rf[31:0];//32个32位寄存器

    //write if enabled
    always @(posedge clk)
        if(regWriteEnable) rf[regWriteAddress] <= regWriteData;

    //read if address != 0, else output 0
    assign RsData = (RsAddress != 0) ? rf[RsAddress] : 0;
    assign RtData = (RtAddress != 0) ? rf[RtAddress] : 0;
endmodule
```

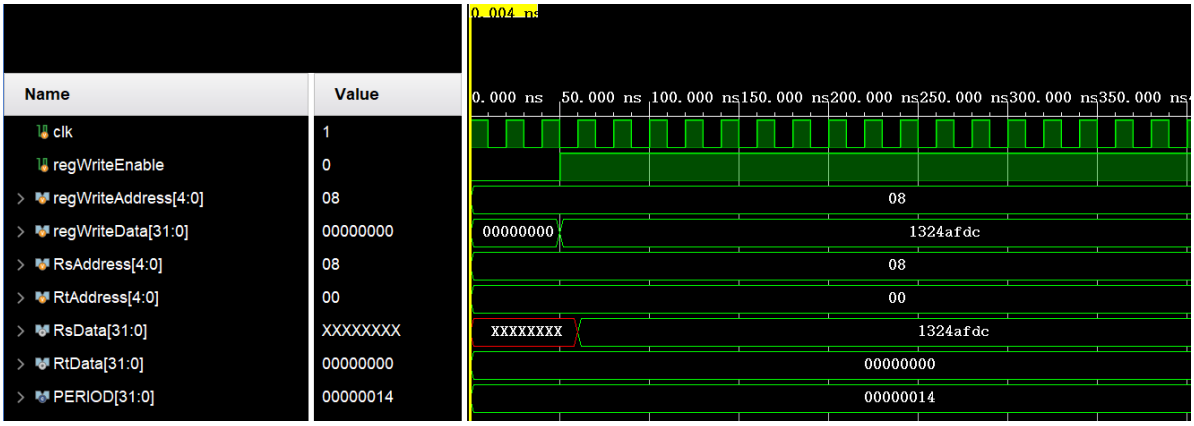
仿真代码：

```
`timescale 1ns / 1ps

module regfile_tb();
    reg           clk;
    reg           regWriteEnable; //写入使能
    reg   [4:0]   regWriteAddress;//写入地址，深度为32故只要5位
    reg   [31:0]  regWriteData;
    reg   [4:0]   RsAddress;      //Rs寄存器，源操作数寄存器号
    reg   [4:0]   RtAddress;      // Rt寄存器
    wire  [31:0]  RsData;
    wire  [31:0]  RtData;
    //实例化
    regfile regFile(.clk(clk), .regWriteEnable(regWriteEnable),
    .regWriteAddress(regWriteAddress), .regWriteData(regWriteData),
    .RsAddress(RsAddress), .RsData(RsData),
    .RtAddress(RtAddress), .RtData(RtData));
    //初始化
    initial begin
        clk = 0;
        regWriteEnable = 0;
        regWriteAddress = 0;
        regWriteData = 0;
        RsAddress = 0;
        RtAddress = 0;
        #50;
        regWriteEnable = 1;
        regWriteData = 32'h1324afdc;
    end
    //产生时钟信号
    parameter PERIOD = 20;
    always begin
        clk = ~clk;
        #(PERIOD/2);
    end
    //与时钟同步的激励信号
    always begin
        regWriteAddress = 8;
        RsAddress = 8;
        #PERIOD;
    end
endmodule
```

测试功能：

- 读取地址为0时确保读到的数据为0
- 时钟上升沿写入
- 任何时候的读取

仿真结果：



波形图说明：

RtAddress一直为0所以输出为0，RsAddress为8，刚开始regfile中没有数据，所以输出为X，50ns时将1324afdc赋给regWriteData，在下一个时钟上升沿（60ns）被写入地址为8的register，与此同时RsData读出这个数据。

imem

功能：

32(width)x64(depth) distributed ROM

使用模板如下：

```
//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
imem your_instance_name (
  .a(a),      // input wire [5 : 0] a
  .spo(spo)  // output wire [31 : 0] spo
);
```

用途：

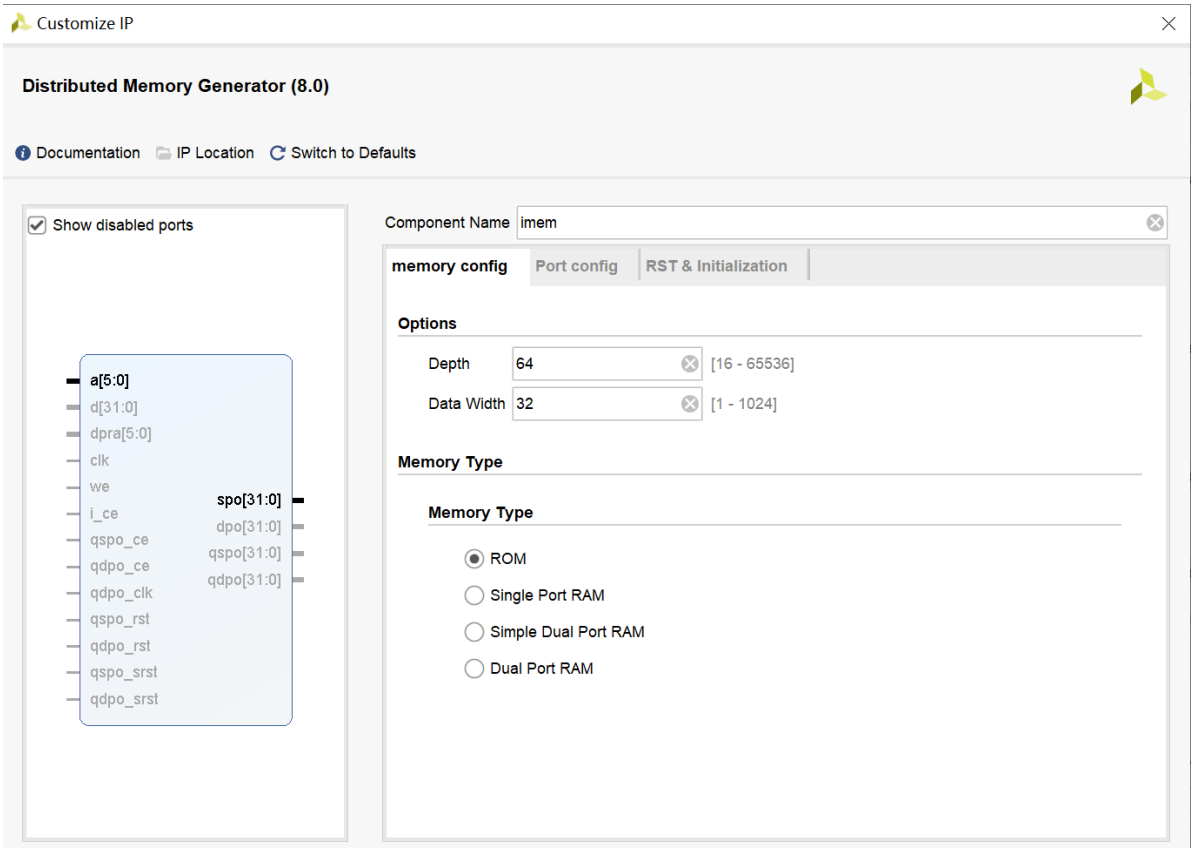
用作程序执行时的instruction memory。

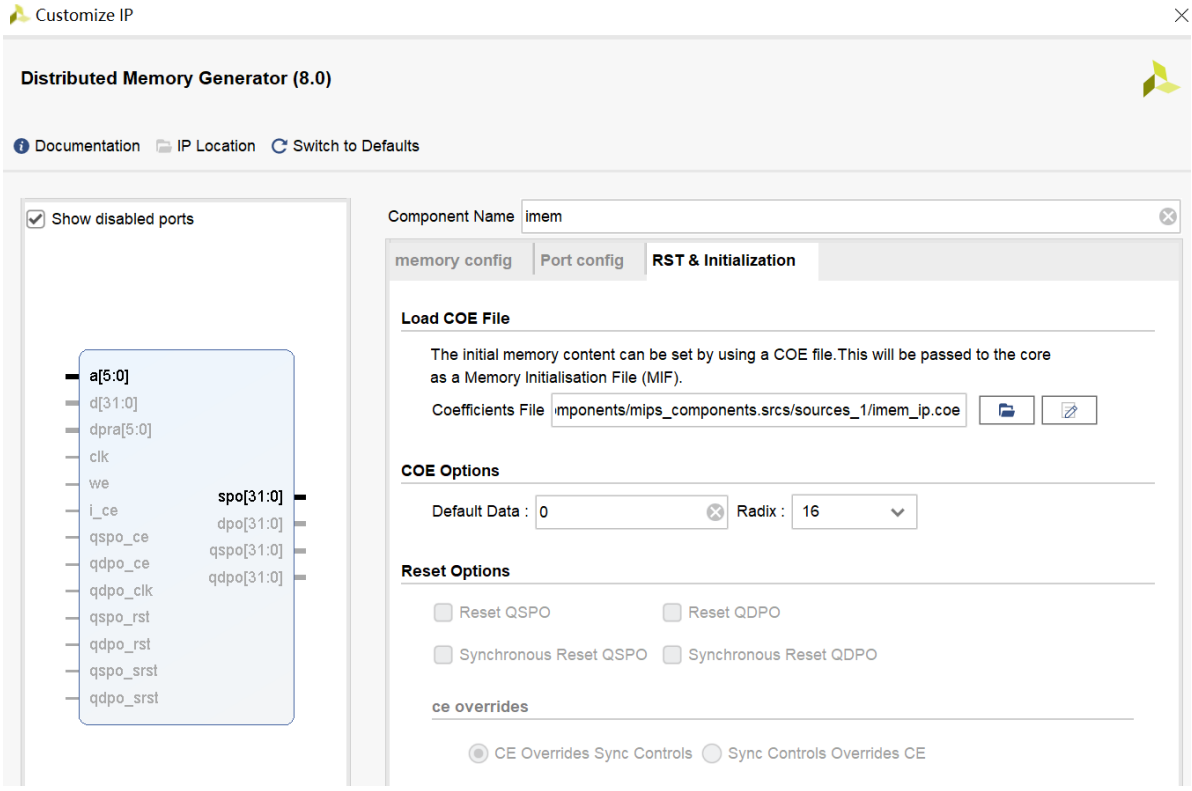
实现思路：

利用vivado提供的IP核——Distributed Memory Generator来实现，可以将程序指令写在一个COE文件中对内存进行初始化，方便测试和修改。因为每个指令为32位，如果地址不是4的倍数读出来的数据没有意义，所以这个ROM是字寻址的，即address0实际上对应内存中0-3，address1对应4-7……而且考虑到实际情况，一次性运行的指令将不会大于16条，这里申请的内存空间为32×64字节，最多可以容纳64条指令，故指令地址的实际有效位应为低11（5+6）位，且最低2位始终为0。

所以该部件的输入信号是PC[7:2]，共6位，输出信号作为instruction，共32位。

配置图如下：





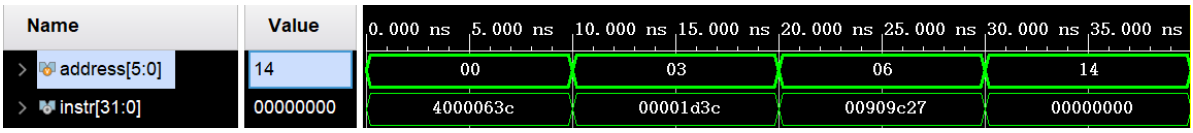
测试功能：

- 读取被.coe文件初始化过的指令
- 读取没有被.coe文件初始化过的地址的值

仿真代码：

```
`timescale 1ns / 1ps

module imem_tb();
    reg     [5:0]   address;
    wire    [31:0]  instr;
    imem your_instance_name (
      .a(address),      // input wire [5 : 0] a
      .spo(instr)  // output wire [31 : 0] spo
    );
    initial begin
        address = 0;
        #10;
        address = 3;
        #10;
        address = 6;
        #10;
        address = 20;
    end
endmodule
```

仿真结果：



波形图说明：

随便找了一个.coe系数文件作为imem的初始化文件，然后取出其中一些地址的值，系数文件如下所示：

```
;Memory Width:32,Memory Depth:64
memory_initialization_radix=16;
memory_initialization_vector=
4000063c,00608640,00688040,00001d3c,
e01fbd27,01001c3c,00909c27,00000734,
1300e000,00001834,11000003,86000008,
21f0a003,0000c0af,01000224,0400c2af;
```

可看出imem模块可以正确读取内存中的值，超出coe文件初始化的范围（0-15）的地址被初始化为0，所以最后一个address=20时读出来的instr为0。

dmem

功能：

32(width)x64(depth) RAM

用途：

用作程序执行时的data memory。

实现思路：

刚开始本来也想用vivado的IP配置，但是用Distributed Memory Generator的RAM没有办法直接实现字节寻址，要么就只能像imem一样提供字寻址，要么就只能一次读取一个字节，而MIPS存储器模型是字节(8bits)寻址，而不是字(32bits)寻址，理论上可以访问任意一个位置的字节，同时MIPS指令集对内存的读写又都是以一个字（1word = 4bytes）为单位的，所以像PPT上的直接忽略地址的最低两位也不是很好。

既要字节寻址又要一次性输出一个字，就必须在外面再套一层模块，所以我最终选择自己用数组实现，而不用IP核。

每一个数据字节都有一个唯一的地址，所以width=8，由于memory的使用不会超过256字节，故depth=256。对当前地址以及后面三个地址进行读写操作，如果[address:address+3]有任何一个字节超出内存范围，就不允执行（write）或输出x（read）。

实现代码：

```
module dmem(
input   clk, we,
input   [31:0]  address,
input   [31:0]  writeData,
output  [31:0]  readData
    );
    reg [7:0] RAM[255:0];
    //read
    assign readData = (address + 3 < 256)? {RAM[address+3], RAM[address+2],
RAM[address+1], RAM[address]} : 32'bx;
    //write if we is 1
    always @(posedge clk)
        if(we & address + 3 < 256)begin
            RAM[address+3] <= writeData[31:24];
            RAM[address+2] <= writeData[23:16];
            RAM[address+1] <= writeData[15:8];
            RAM[address+0] <= writeData[7:0];
        end
endmodule
```

仿真代码：

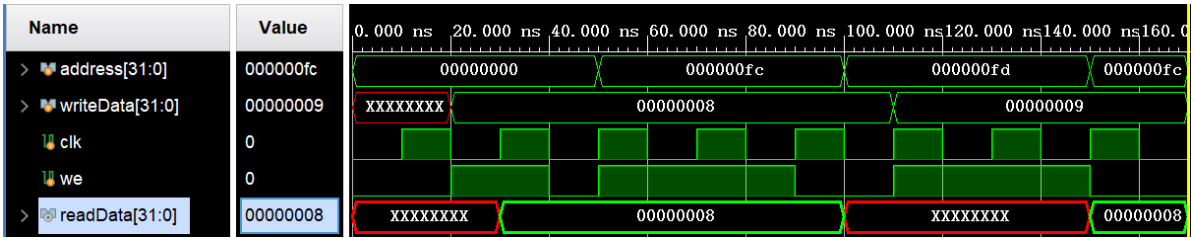
```
`timescale 1ns / 1ps

module dmem_tb();
    reg [31:0]   address;
    reg [31:0]  writeData;
    reg         clk, we;
    wire[31:0]  readData;
    dmem DMEM (
      .address(address),
      .writeData(writeData),
      .clk(clk),  // input wire clk
      .we(we),    // input wire we
```

```
        .readData(readData)
    );
    initial begin
        clk = 0;
        we = 0;
        //read data from address0
        address = 0;
        #20;
        //write 8 at address0
        writeData = 8;
        we = 1;
        #20;
        we = 0;
        #10;
        //write 8 at last word
        we = 1;
        address = 252;
        #40
        //read last word
        we = 0;
        #10
        //read from out scope
        address = 253;
        #10;
        //write 9 to out scope
        we = 1;
        writeData = 9;
        #40;
        //read last word, no change
        we = 0;
        address = 252;
    end

    always #10 clk = ~clk;
endmodule
```

测试功能：

- 任何时候都读取地址对应的数据
- 在时钟上升沿往指定地址写入数据
- address超过内存范围的异常处理

仿真结果：



波形图说明：

首先读取address0的数据，由于没有初始化所以是x。然后往address0写入8，在时钟上升沿即30ns时被写入，readData可以及时的反应这个变化。然后在50ns时令address=252，恰好是时钟上升沿，故被写入data memory，在100ns时令address=253，超出了内存字地址的范围，readData为x，在110ns尝试往这个地址写入9，在150ns时读取最后一个字还是8说明上一次write没有成功。