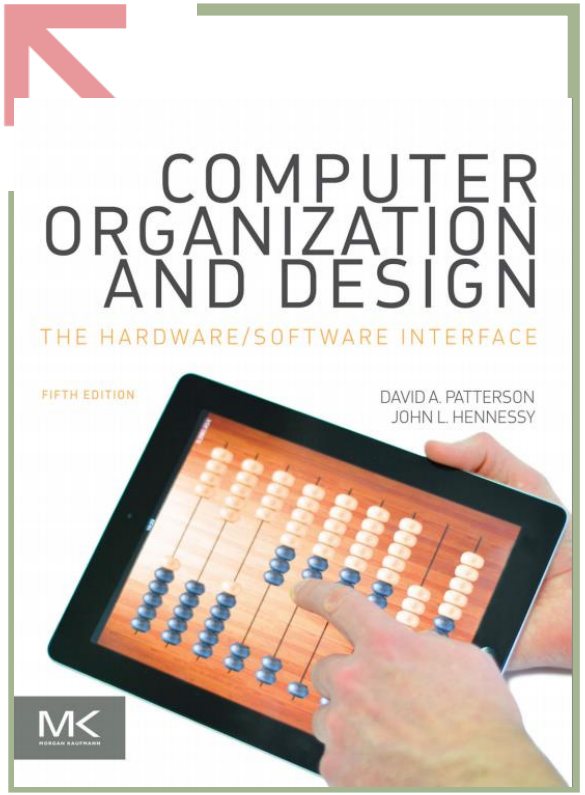
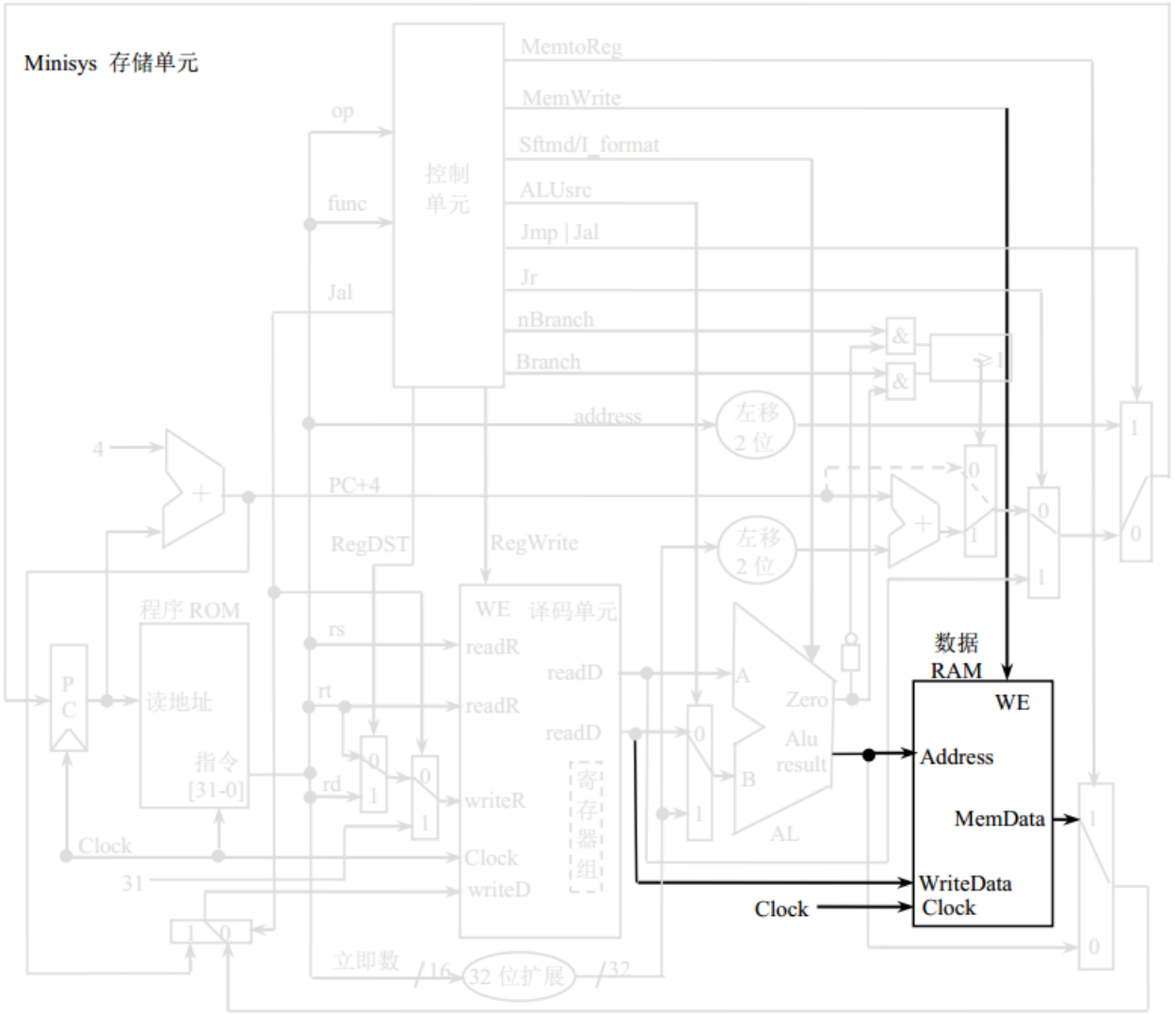
存储单元及I/O单元


COMPUTER
ORGANIZATION
AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
FIFTH EDITION
DAVID A. PATTERSON
JOHN L. HENNESSY
MK

存储单元的设计

存储单元的设计需要完成：

1. 定义数据存储单元RAM
2. 实现对数据存储器的读写操作



存储单元的端口

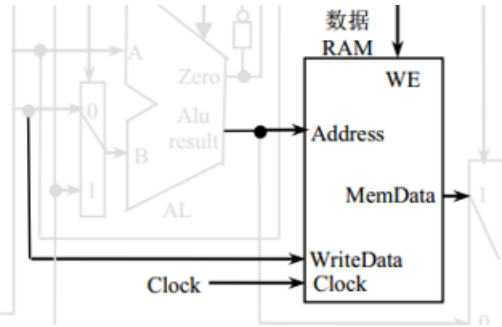
module **dmemory32**(read_data,address,write_data,Memwrite,clock);
// 从存储器读出的数据
output[31:0] **read_data**;

//来自 memorio 模块，源头是来自执行单元的输出 alu_result
input[31:0] **address**;

//来自译码单元的 read_data2
input[31:0] **write_data**;

//来自控制单元
input **Memwrite**;

//时钟信号
input **clock**;

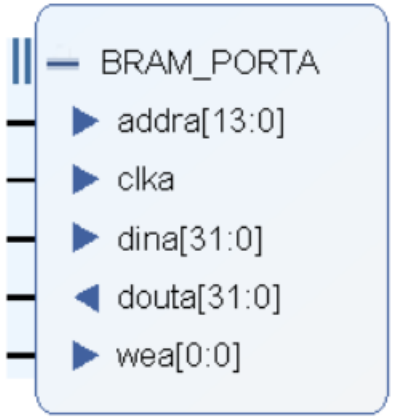


定义数据存储单元RAM

数据 RAM 使用 Xilinx 公司提供的 IP 核 Block Memory Generator实现。

设置数据存储单元：
名称：RAM，类型： Native 单口 RAM，无 ECC 校验，最小面积算法。

I/O 引脚如图所示：
数据读写位宽： 32 位
size：64KB， 16384 个基本数据。



Vivado中定义RAM的步骤可参照取址单元定义ROM的操作完成。

定义数据存储单元RAM

指定RAM的初始化文件时请注意：
刚创建的项目是没有初始文件和相应的路径的，所以在创建 RAM 的时候可以先设置成没有初始文件。

创建好 RAM 后，将初 始 化 文 件 dmem32.coe 拷贝到工程文件夹目录下的/sources_1/ip/ram/中。

双击刚建立的 RAM IP 核，重新设置其为有初始化文件，并选择已经拷贝好的 dmem32.coe 文件。

实验中RAM使用的初始化文件请使用 dmem32.coe

存储单元例化

```
//生成clk信号，该信号是clock信号的反相时钟
assign clk = !clock;

RAM ram (
      .clka(clk),              // input wire clka
      .wea(Memwrite),  // input wire [0 : 0] wea
      .addra(address[15:2]), // input wire [13 : 0] addra
      .dina(write_data), // input wire [31 : 0] dina
      .douta(read_data)            // output wire [31 : 0] douta
 );
```

说明：因为使用芯片的固有延迟， RAM 的地址线来不及在时钟上升沿准备好, 从而造成在时钟上升沿数据读出有误，考虑采用反相时钟的方式，使得读出数据比地址准备好要晚大约半个时钟，从而得到正确地址。

模块仿真

```
`timescale 1ns / 1ps
module ram_sim(  );
    // input
    reg[31:0] address = 32'h00000010;     //来自执行单元算出的 alu_result
    reg[31:0] write_data = 32'ha0000000;  //来自译码单元的 read_data2
    reg  Memwrite = 1'b0;          //来自控制单元
    reg  clock = 1'b0;
    // output
    wire[31:0] read_data;

    dmemory32 Uram(read_data,address,write_data,Memwrite,clock);

    initial begin
      #200 begin write_data = 32'hA00000F5;Memwrite = 1'b1; end
      #200 Memwrite = 1'b0;
    end
    always #50 clock = ~clock;
endmodule
```

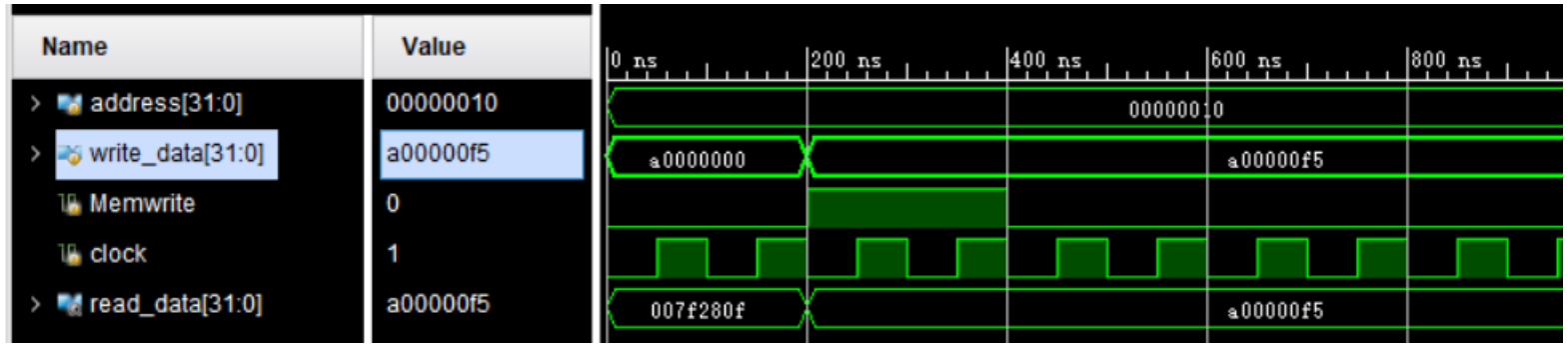
地址为 0x00000010
1）设置MemWrite为低电平，读取RAM中该地址的数据

2）设置 MemWrite 为高电平,write_data 为 0xa00000f5

3）设置MenWrite为低电平，读取RAM中该地址的数据

仿真参考波形

```
memory_initialization_radix = 16;
memory_initialization_vector =
007f2812,
007f2811,
007f2810,
007f2810,
007f280f,
00000001,
00000002,
00000003,
00000005,
00000006,
00000007,
0000ffff,
00000000,
00000000,
```



请思考：

1）对RAM的读操作和写操作分别是在clock信号的什么时刻发生

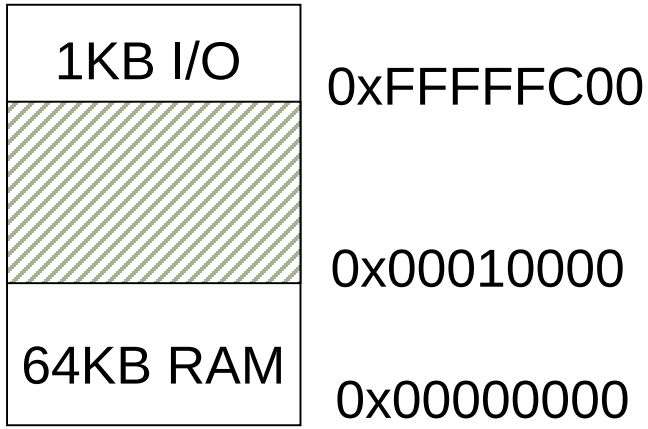
2）分别从address为0和为1的位置读取的数据是什么

CPU简单接口设计

这一部分主要完成以下工作：

1. 添加 I/O 功能
2. 16 位 LED 的设计
3. 16 位拨码开关的设计

添加 I/O 功能



I/O 端口与存储器统一编址

将 32 位 RAM 地址分区：高 1024 字节分配给 I/O。

指定10 根 IO 端口线（32 位地址线的低 10 位，高 22 位为全 1）来实现片选和寻址。

1. LED灯 的片选信号为ledCtrl
    1. Minisys开发板上有24个led灯：首地址为0xFFFFFC60，基于该地址可做16bit数据的写入操作，剩余的8bit数据（24-16）基于0xFFFFFC62进行写入。

2. 拨码开关的片选信号为switchCtrl
    1. Minisys开发板上有24个拨码开关：首地址为0xFFFFFC70，基于该地址可做16bit数据的读出操作，剩余的8bit数据（24-16）基于0xFFFFFC72进行读出。

*说明：*在计算机领域，对于 I/O 地址空间的设计通常采用两种方案：I/O 与存储器统一编址 或I/O 独立编址.*当前 Minisys-1 的指令系统中没有设置专用的I/O 指令，借用 LW 和 SW 两条指令来进行 RAM 访问和 I/O 访问，从而决定了 Minisys-1 目前只能使用 I/O 统一编址方式。*

控制单元的修改

控制单元的新端口定义

module **control32**
(Opcode,Jrn,Function_opcode,**Alu_resultHigh**,RegDST,ALUSrc,**MemorIOtoReg**,RegWrite,MemRead,MemWrite,
IORead, **IOWrite**,Branch,nBranch,Jmp,Jal,I_format,Sftmd,ALUOp);

```
…
input[21:0] Alu_resultHigh;        // 来自执行单元 Alu_Result[31..10]
output MemorIOtoReg;               // 为 1 表明需要从存储器或 I/O 读数据到寄存器
output RegWrite;                   // 为 1 表明该指令需要写寄存器
output MemRead;                    // 为 1 表明是存储器读
output MemWrite;                   // 为 1 表明该指令需要写存储器
output IORead;                     // 为 1 表明是 I/O 读
output IOWrite;                    // 为 1 表明是 I/O 写
…
```

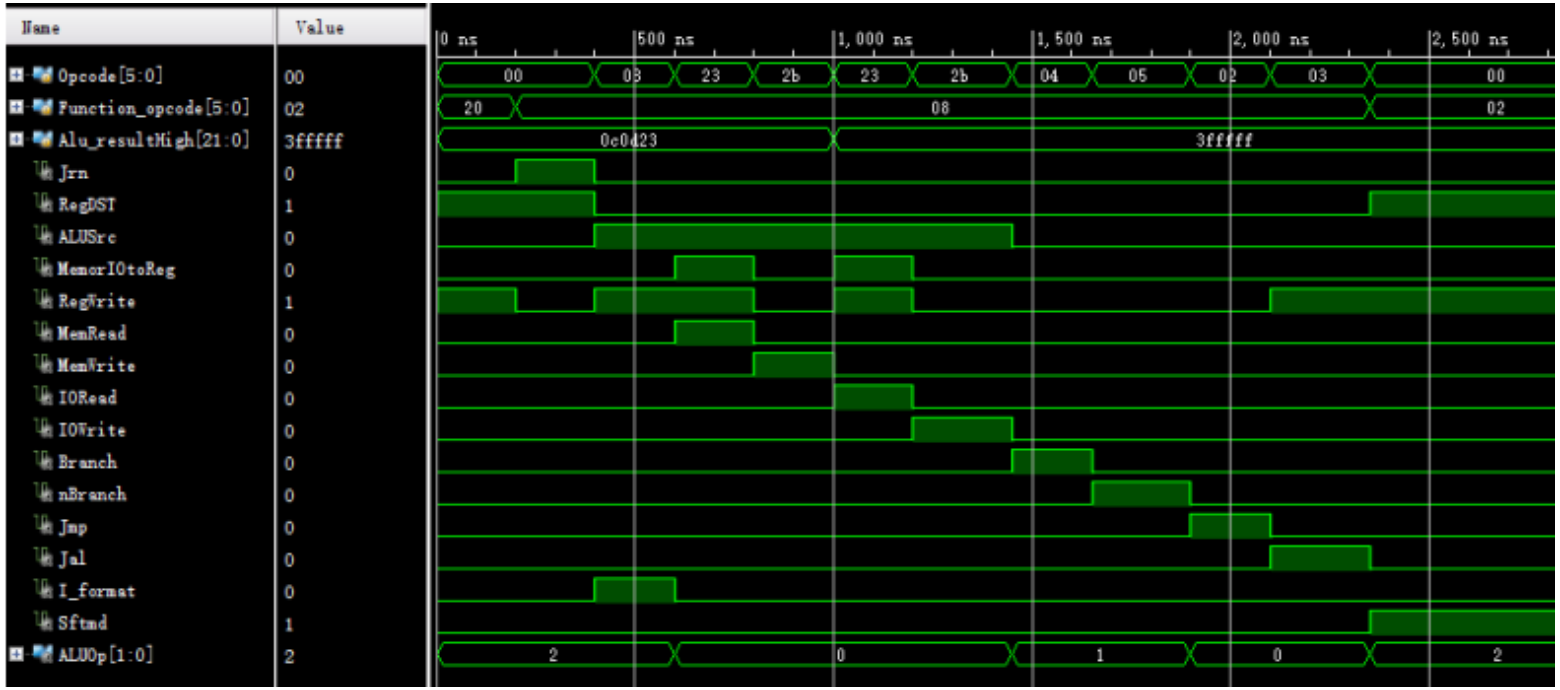
控制单元的修改

修改MemWrite信号的逻辑，增加MemRead, IORead和IOWrite信号，改MemtoReg为MemorIOtoReg

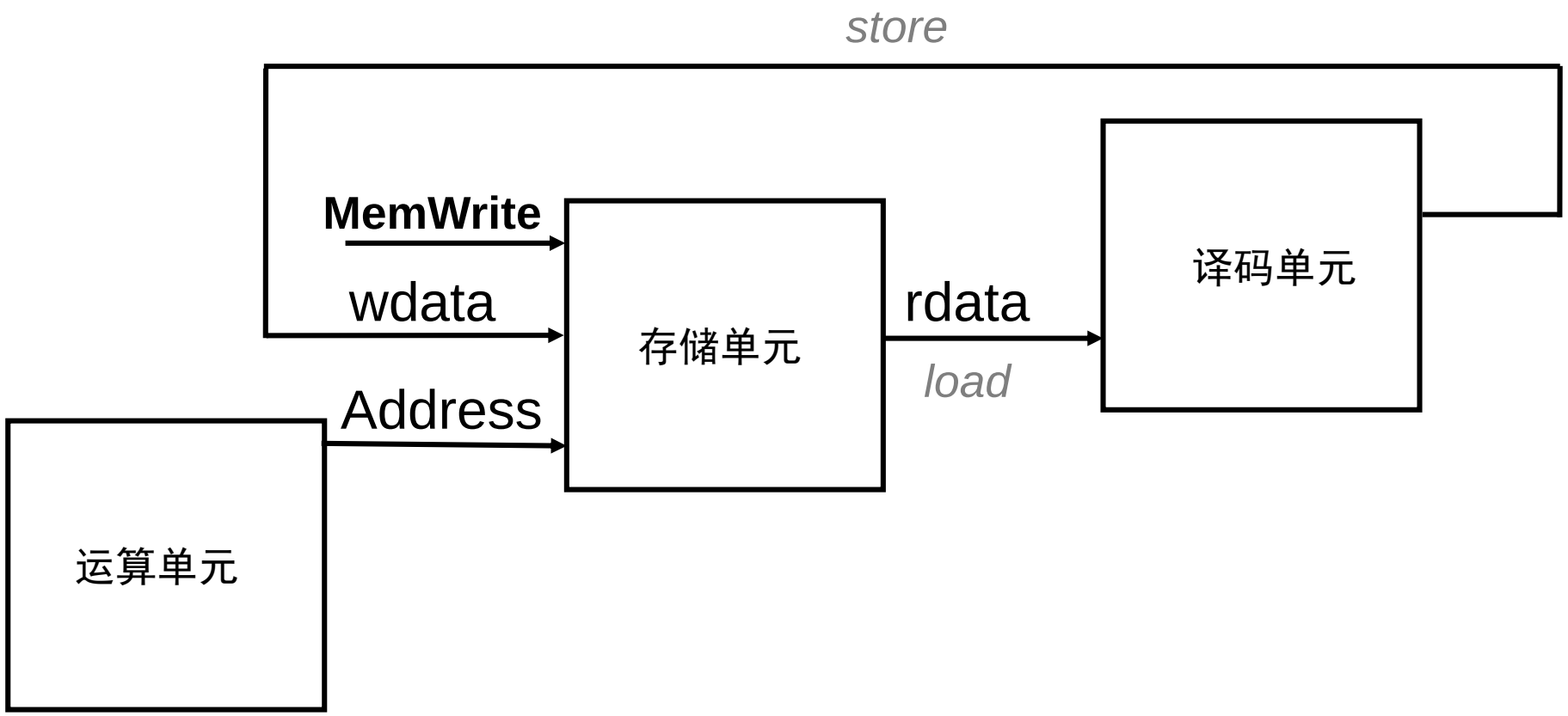
```
//LW和SW的真正地址是Alu_Result,该信号来自执行单元
 input[21:0]  Alu_resultHigh;    // 来自执行单元Alu_Result[31..10]，用于协助判断是对Mem还是IO进行处理

 output       MemorIOtoReg;     // 为1表明需要从存储器或I/O读数据到寄存器
 output       MemRead;          // 为1表明是存储器读
 output       IORead;           // 为1表明是I/O读
 output       IOWrite;          // 为1表明是I/O写

 assign RegWrite = (R_format || Lw || Jal || I_format) && !(Jrn) ;  // 需要写寄存器的指令
// 写存储器或者写IO
 assign MemWrite = ((Sw==1) && (Alu_resultHigh[21:0] != 22'b1111111111111111111111)) ? 1'b1:1'b0;
 assign MemRead = ？ ？ ？    // 读存储器
 assign IORead  = ？ ？ ？  // 读端口
 assign IOWrite = ？ ？ ？   // 写端口

 assign MemorIOtoReg = IORead || MemRead;      //读操作需要从端口或存储器读数据到寄存器
```

对修改后的控制单元进行功能验证

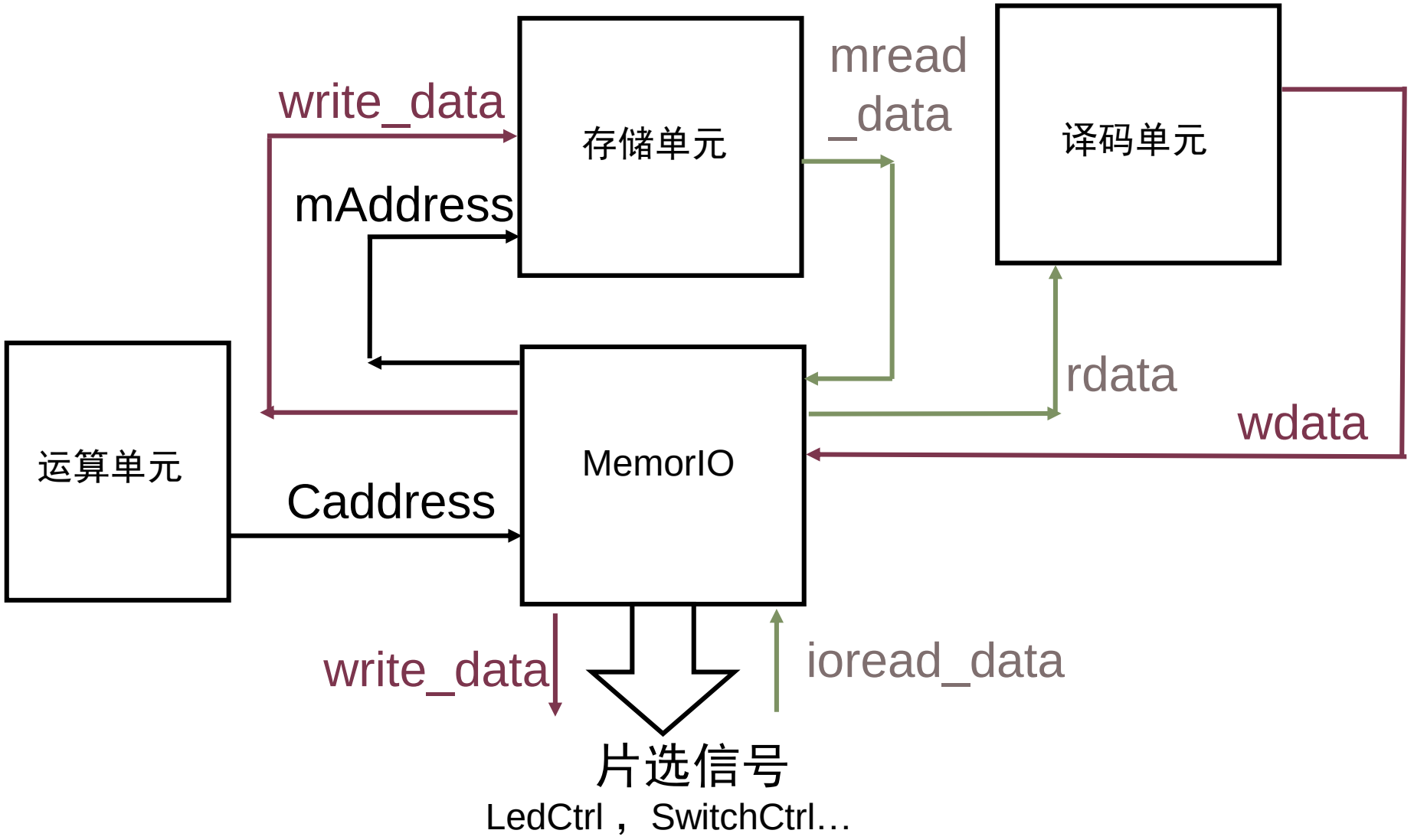
```
initial begin
    #200     Function_opcode  = 6'b001000;                 // JR
    #200    Opcode = 6'b001000;                            //  ADDI
    #200    Opcode = 6'b100011;                            //  LW
    #200    Opcode = 6'b101011;                            //  SW
    #200    begin Opcode = 6'b100011; Alu_resultHigh = 22'b1111111111111111111111; end;  //  LW  IO
    #200    begin Opcode = 6'b101011; Alu_resultHigh = 22'b1111111111111111111111; end;  //  SW  io
    #250    Opcode = 6'b000100;                            //  BEQ
    #200    Opcode = 6'b000101;                            //  BNE
    #250    Opcode = 6'b000010;                            //  JMP
    #200    Opcode = 6'b000011;                            //  JAL
    #250    begin Opcode = 6'b000000; Function_opcode  = 6'b000010; end;//  SRL
end
```



LW/SW 对应的操作



增加 memorio 模块



增加 memorio 模块

module **memorio**(caddress,address,memread,memwrite,ioread,
iowrite,mread_data,ioread_data,wdata,rdata,write_data,LEDCtrl,SwitchCtrl);

```
input[31:0] caddress;        // from alu_result in executs32

input memread;               // read memory, from control32
input memwrite;              // write memory, from control32
input ioread;                // read IO, from control32
input iowrite;               // write IO, from control32

input[31:0] mread_data;      // data from memory
input[15:0] ioread_data;     // data from io,16 bits
input[31:0] wdata;           // the data from idecode32
output[31:0] rdata;          // data from memory or IO
output[31:0] write_data;     // data to memory or I/O
output[31:0] address;        // address to memory
output LEDCtrl;              // LED CS
output SwitchCtrl;           // Switch CS
```

增加memorio 模块

```
reg[31:0] write_data;
wire iorw;

assign address = caddress;
assign rdata = ？？？ // 可能是从memory读出，也可能自io读出，自io读取的数据是rdata的低16bit
assign iorw = (iowrite||ioread);

assign LEDCtrl= ？？？ // led 模块的片选信号，高电平有效;
assign SwitchCtrl= ？？？ //switch 模块的片选信号，高电平有效;

always @* begin
    if((memwrite==1)||(iowrite==1)) begin
        write_data = ？？？； //向io的写操作数据写入write_data的低16bit，高16bit为0
    end
    else begin
        write_data = 32'hZZZZZZZZ;
    end
end
endmodule
```

为多路 IO 输入增加一个多路选择器

考虑CPU外设的可扩展性(如可能有多个IO接口单元，比如拨码开关\4X4 键盘\按键等)， 通过ioread 模块实现多路 输入数据的选择。在文件 ioread.v 中实现该模块：

```
module ioread(reset,ior,switchctrl,ioread_data,ioread_data_switch);
    input reset;        // 复位信号
    input ior;          // 从控制器来的 I/O 读信号
    input switchctrl;                       // 从 memorio 经过地址高端线获得的拨码开关模块片选
    input[15:0] ioread_data_switch;         //从外设来的读数据，此处来自拨码开关
    output[15:0] ioread_data;               // 将外设来的数据送给 memorio
    reg[15:0] ioread_data;
    always @* begin
        if(reset == 1)
                ioread_data = 16'b0000000000000000;
        else if(ior == 1) begin
                        if(switchctrl == 1)
                                ioread_data = ioread_data_switch;
                        else
                                ioread_data = ioread_data;
        end
    end
endmodule
```

练习：对存储模块进行仿真验证

- 请从sakai直接导入写好的leds.v和switchs.v文件。
- 搭建一个顶层模块，例化并对接好存储单元, memorio, ioread, leds和switchs。
- 搭建仿真用的testbench并进行仿真验证，要求：
  - 在200ns时读取 switchs 的数据，将其写入reg $1的低24bit；
  - 在600ns时读取$1上的数据，将其写入memory 地址为1的存储单元；
  - 在800ns读取memory 地址为1的存储单元的值
  - 在1000ns时将其写入到 led 作为输出；

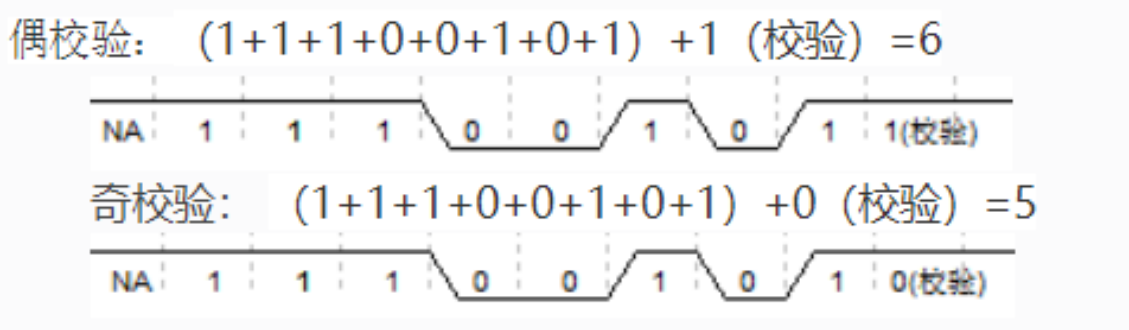
存储单元的比较

内存类型	静态RAM和动态RAM之间的区别			RAM和ROM的区别
	区别点	静态RAM	动态RAM	
随机存储器（RAM）	1	集成度低	集成度高	信息可以随时写出。写入时，原数据被冲掉。加电时信息完好，一旦断电，信息消失，无法恢复
	2	价格高	价格低	
	3	存取速度快	存取速度慢	
	4	不需要刷新	需要刷新	
只读存储器（ROM）	分类	可编程只读存储器（PROM）可擦除的可编程只读擦除器（EPROM）掩模型只读存储器（MROM）		信息时永久性的，即使关机也不会消失

ECC校验

ECC校验是一种内存纠错原理，它是比较先进的内存错误检查和更正的手段。ECC内存即纠错内存，简单的说，其具有发现错误，纠正错误的功能，这将使整个电脑系统在工作时更趋于安全稳定。

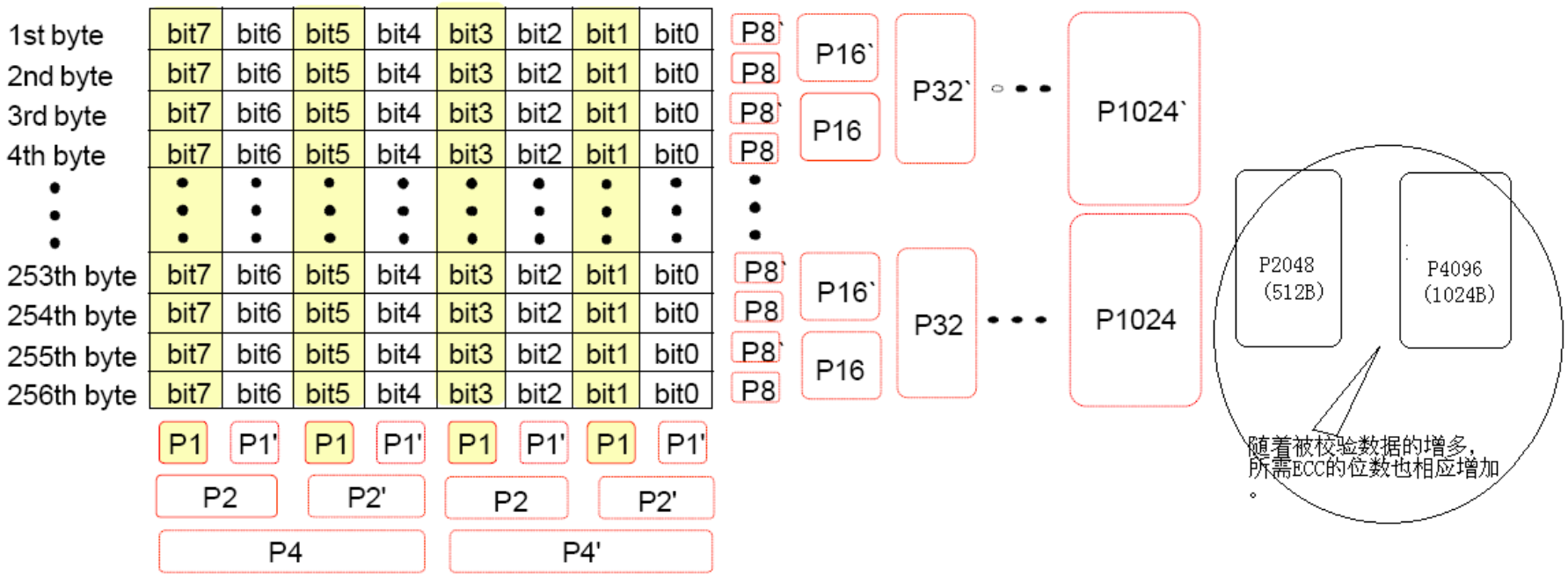
奇偶校验原理



对于偶校验，校验位就定义为1，反之则为0；
对于奇校验，则相反。
当CPU读取存储的数据时，它会再次把前8位中存储的数据相加，计算结果是否与校验位相一致。从而一定程度上能检测出内存错误，奇偶校验只能检测出错误而无法对其进行修正，同时虽然双位同时发生错误的概率相当低，奇偶校验却无法检测出双位错误。

ECC校验

- **Parity Generation (In case of 256 byte input)**



P1=bit7(+)bit5(+)bit3(+)bit1(+)P1
P2=bit7(+)bit6(+)bit3(+)bit2(+)P2
P4=bit7(+)bit6(+)bit5(+)bit4(+)P4

P8 = bit7(+)bit6(+)bit5(+)bit4(+)bit3(+)bit2(+)bit1(+)bit0(+)P8

ECC校验

表格 1　ECC 列校验规则示意图

Byte 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 2	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 3	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 252	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 253	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 254	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 255	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	CP1	CP0	CP1	CP0	CP1	CP0	CP1	CP0
	CP3		CP2		CP3		CP2	
	CP5				CP4			

ECC校验

表格 2　ECC 行校验规则示意图

Byte 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP2	RP4	....	RP14
Byte 1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 2	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP3			
Byte 3	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 252	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP2	RP5		RP15
Byte 253	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				
Byte 254	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP0	RP3			
Byte 255	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RP1				

ECC校验

综上所述，对256字节的数据共生成了6个Bit的列校验结果，16个Bit的行校验结果，共22个Bit。在Nand中使用3个字节存放校验结果，多余的两个Bit位置1。存放次序如下表所示：

表格 3　K9F1208 中 22Bit 校验码的排列规则

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	RP7	RP6	RP5	RP4	RP3	RP2	RP1	RP0
Byte 1	RP15	RP14	RP13	RP12	RP11	RP10	RP9	RP8
Byte 2	CP5	CP4	CP3	CP2	CP1	CP0	1	1

课外阅读：
ECC算法的详细说明
http://www.360doc.com/content/11/0523/17/496343_118837985.shtml

生成算法

最小面积算法：数据和校验位都有用。用最小数量的块ram原语组成。

最低功耗算法：组成和上面一样，就是在读写操作时才使能块RAM。

合适的原语算法：这个是由软件自动择中选择。