Lab6 综合实验

王舒 PB19071472

添加指令

本次实验参照所给的快排汇编代码，在原来的基础上实现了一些新的指令。其中，在 `control` 中分析的时候需要用到 `funct3` 的部分，对原先的 `control` 和 `ALUcontrol` 部分代码进行了一些修改。
`ALUop` 控制信号对应的操作如下

ALUop	对应运算
2'b00	N/A
2'b01	-
2'b10	+
2'b11	or，xor，depends on funct3

- **xor，xori，or**
  这三条都是纯粹的算术指令，在原来的代码上改动不大，改 `control` 和 `ALUcontrol` 即可
- **lui，slli**
  需要左移的指令，在 `ALU_result_EXMEM` 前加上一个 `mux` ， `LeftShift` 作为控制线，控制信号如下

LeftShift	来路
2'b00	ALU运算
2'b01	对应slli
2'10	对应lui

  其中，lui的左移在立即数生成的时候就已经顺便实现了。
  代码实现如下

```
//mux for leftshift(after alu)
case(LeftShift_IDEX)
2'b00:
    ALU_result_EXMEM<=ALU_result_EXMEM_wire;
2'b01:
    ALU_result_EXMEM<=ALU_A_IDEX<<imm_IDEX[4:0];
2'b10:
    ALU_result_EXMEM<=imm_IDEX;
default:
    ALU_result_EXMEM<=ALU_result_EXMEM_wire;
endcase
```

- **blt，bge**
  这两条指令和 beq 指令几乎一样，不同的是，在判断时使用了 ALU_result 的最高位来比较大小。
- **jalr**
  这条指令需要在 pc 处进行主要的修改。由于多了一种 pc 的选择，要在 pc 前面加一个 mux ，并通过控制信号实现。具体代码如下，其中 PCSrc 的两位各有含义。

```
//PC
wire [1:0] PCSrc;
reg [31:0] pc_next;

//assign PCSrc = (ALU_Z_EXMEM_wire & BEQ_IDEX) | jal_IDEX;<-这是原来的代码，以供对比
assign PCSrc[0] = (ALU_Z_EXMEM_wire & BEQ_IDEX)|(~ALU_result_EXMEM_wire[31] & BGE_IDEX) |
(ALU_result_EXMEM_wire[31] & BLT_IDEX ) | jal_IDEX;
assign PCSrc[1]=jalr_IDEX;

always@(*)
begin
    if( PCSrc[1]==1'b1)//JALR
    begin
        pc_next<=ALU_A_IDEX+imm_IDEX;

    end
    else if(PCSrc[0]==1'b1)//BRANCH,JAL
    begin
        pc_next<=pc_IDEX+imm_IDEX;
    end
    else
    begin
        pc_next<=pc+4;
    end
end
```

直接相联cache实现

cache参数：

- lines=16，each line one word
- tag_len=7
- index_len=4

cache实现：

在 cache 内部调用 data_mem 。

- read
  输出的数据用组合逻辑连到对应位置的 cache 上。

  ```
  assign data_out=cache_mem[addr[3:0]];
  ```

  如果 miss 了， miss 信号会传出去，连到 hazard 里面，让整个流水线停顿。

  ```
  //hazard中新加的部分代码
  if(miss==1'b1)
    begin
        enable_IF=1'b0;
        enable_ID=1'b0;
        enable_EXMEM=1'b0;
        enable_MEMWB=1'b0;

        flush_IF=1'b0;
        flush_ID=1'b0;
    end
  ```

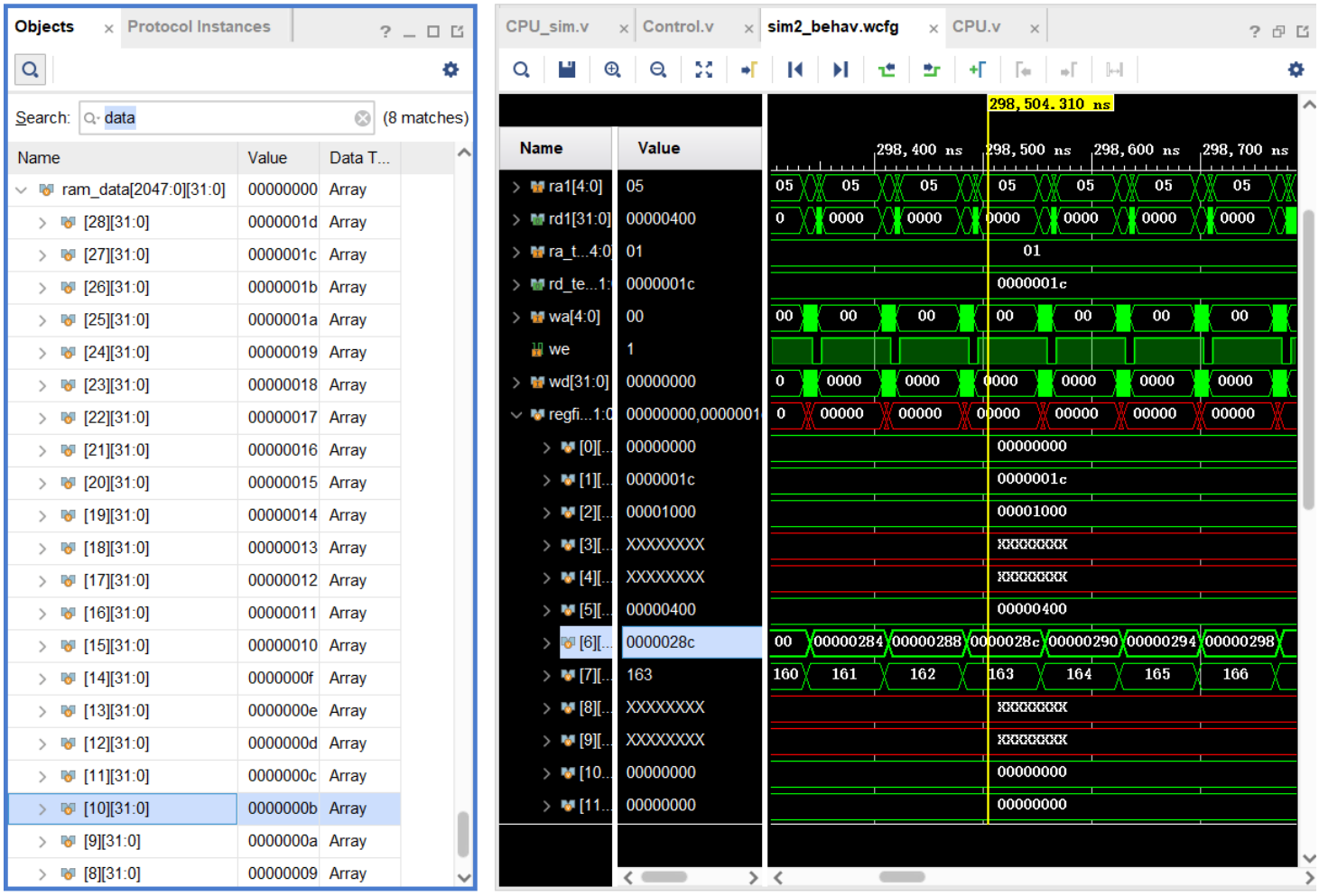
  同时，进入状态机来延迟十几个周期。当快结束的时候，从主存中调出数据，并且更改 cache 中对应位置的内容。
  这里在 miss 的时候虽然有组合逻辑连着 data_out ，但这个时候整个流水线都停机了，所以其实是没有影响的，相当于白写了一个错误的数出去。
  添加 read 信号的意义在于，如果遇到不需要读内存的指令，就不会进入 cache ，否则会一直产生没有意义的错误。
- write
  直接写入主存里。如果在 cache 中命中了，就更改 cache 中的对应数据。
  cache 部分代码整体如下

```
module Cache (
  input clk,
  input rst,
  input wire [10:0] addr,//depth=2048=2^11
  input wire [31:0] data,//data width=32
  input we,
  input read,//add this wire to know which instruction really reads the DM
  output [31:0] data_out,
  output miss
      );

  reg [15:0] valid;
  reg [31:0] cache_mem[0:15];
  reg [6:0] tags[0:15];

  wire [31:0] mem_data_out;

  //read when hit
  assign data_out=cache_mem[addr[3:0]];

  reg [3:0] status;
  assign miss=(read==1'b1) && !((valid[addr[3:0]]==1'b1) && (tags[addr[3:0]]==addr[10:4]));
  always@(posedge clk,posedge rst)
  begin
      if(rst==1'b1)
      begin
          valid<=16'b0000000000000000;
          status<=4'b0;
      end
      else //read when miss
      begin
          if(miss==1'b1)
          begin
              if(status==4'b0)
              begin
                  status<=4'd12;
              end
              else
              begin
                  status<=status-4'd1; //counting
                  if(status==4'd1) //counts to the last number,read from DM
                  begin
                      cache_mem[addr[3:0]]<=mem_data_out;
                      valid[addr[3:0]]<=1'b1;
                      tags[addr[3:0]]<=addr[10:4];
                  end
              end
          end
          //write when hit,need to change cache
          if((we==1'b1) && (tags[addr[3:0]]==addr[10:4]) && (valid[addr[3:0]]==1))
              cache_mem[addr[3:0]]<=data;
```

```
    end
end


data_mem data_mem(
.a(addr),
.d(data),
.dpra(),
.clk(clk),
.we(we),
.spo(mem_data_out),
.dpo()
);
endmodule
```

运行结果

采用快排的汇编，将256个杂乱的数排序，得到的仿真结果如下



可以看到7号寄存器的读数和主存中的数字都是有序的

总结

本次实验中，主要是对 cache 的结构有了更深入的了解，化理论为实践。实际上的代码并不复杂，但是具体的思路还是需要思考一下。
加指令的时候由于需要改动的地方比较多，写的时候经常会漏掉一些内容，导致出现意想不到的 bug 。
由于提供的数据范围和栈的大小比原来的要大一些，在仿真的时候需要更长的模拟时间，还需要修改 data_mem 的大小。