中国科学院大学计算机组成原理实验课实验报告

prj2-单周期处理器设计

李国峰 2018K8009922027

移位器shifter.v

为支持MIPS32中的移位运算指令，移位器需要支持32位数的逻辑左移、逻辑右移和算术右移。

逻辑移位的实现很容易，Verilog自带逻辑移位算符`>>`和`<<`。

```
assign sll_result = B << A[4:0];
assign srl_result = B >> A[4:0];
```

算术右移需要在高位补符号位。考虑到移位位数由`A[4:0]`决定，因此移位位数上限是32位。那么对源操作数进行符号扩展到64位，再做逻辑右移即可。

```
assign sra_64 = {{32{B[31]}}, B} >> A[4:0];
assign sra_result = sra_64[31:0];
```

用并行数据选择器输出结果。

```
assign Result = {32{Shiftop == `SLL}} & sll_result |
                {32{Shiftop == `SRA}} & sra_result |
                {32{Shiftop == `SRL}} & srl_result ;
```

处理器核mips_cpu.v

取指

本实验用到的指令RAM是基于理想内存实现的，即输入`PC`就可以输出对应的指令码。在单周期处理器取指过程中，只需要确定`PC`即可。

`PC`的来源有两个，这取决于当前执行的指令。如果当前指令是跳转指令，需要根据执行结果决定是否跳转，以及跳转的目标地址。如果不跳转，则按原来的顺序取下一条指令，即`PC+4`。

```
wire [31:0] next_pc;
wire [31:0] br_target;
wire        br_go;

assign next_pc = br_go ? br_target : PC + 4; //br_go为跳转标志
always@(posedge clk) begin
    if(rst)
        PC <= 32'b00000000;
    else
        PC <= next_pc;
end
```

译码

这里采用的是逐条指令译码的译码逻辑，因为这样在需要对CPU添加支持的指令时较方便，同时代码可读性也较好。这样的译码逻辑可以准确识别且直观看出当前执行的指令是哪一条。以`addu`指令为例。

```
assign inst_addu  = opcode == 6'b0 && sa == 5'b0 && func == 6'b100001;
```

这样就获得了所有指令执行情况的信息。在这些`inst_xxx`信号中，同一时刻只能由一个为高电平。下一步就是根据指令执行情况生成Alu，Shifter，Register，数据选择器以及其他模块的控制信号。

仅以`addiu`指令为例进行分析。该指令为立即数加法指令，因此需要Alu的参与，并且运算类型是确定的，即加法。因此需要将`inst_addiu`添加到`aluop`的生成逻辑中。

```
assign alu_add  = inst_addiu | inst_addu | inst_lw  | inst_sw  |
                  inst_jal   | inst_jalr | inst_lb  | inst_lbu |
                  inst_lh    | inst_lhu  | inst_sb  | inst_sh  |
                  inst_lwl   | inst_lwr  | inst_swl | inst_swr ;
```

该指令做立即数加法，因此需要对立即数扩展，需要在0扩展和符号扩展的结果中选出一个送到Alu。进行扩展的依据是当前指令类型是算数运算指令还是逻辑运算指令。另一个源操作数来自寄存器读数。

```
assign alu_src2 = (src2_is_imm &&  is_logic) ? {16'b0, imm}         :
                  (src2_is_imm && !is_logic) ? {{16{imm[15]}}, imm} :
                   src2_is_8                 ? 32'd8                :
                                               rt_value             ;
```

译码器还需要输出运算结果写入的寄存器地址。`addiu`指令的目标地址由`rt`决定，因此直接将`rt`对应的数送到寄存器的`waddr`即可。

```
assign dest_is_rt = inst_addiu | inst_lui  | inst_lw  | inst_slti |
                    inst_sltiu | inst_andi | inst_ori | inst_xori |
                    inst_lb    | inst_lbu  | inst_lh  | inst_lhu  |
                    inst_lwl   | inst_lwr  ;

assign dest = dest_is_r31 ? 5'd31 :
              dest_is_rt  ? rt    :
                            rd    ;

assign RF_waddr = dest;
```

需要考虑写入寄存器的数据的来源：Alu，Shifter或访存数据。需要注意的是，访存指令数据来源于内存，但同时也用到了Alu，因此这个数据选择器必须包含优先级。

```
assign RF_wdata = move_to_reg  ? rs_value     :
                  res_from_mem ? mem_result   :
                  need_alu     ? alu_result   :
                  need_shift   ? shift_result :
                  inst_lui     ? lui_result   :
                                 32'b0        ;
```

要对寄存器进行写操作，还需要`wen`处于高电平。

```
assign RF_wen = inst_addiu | inst_addu  | inst_subu   | inst_and  |
                inst_andi  | inst_nor   | inst_or     | inst_ori  |
                inst_xor   | inst_xori  | inst_slt    | inst_slti |
                inst_sltu  | inst_sltiu | inst_sll    | inst_sllv |
                inst_sra   | inst_srav  | inst_srl    | inst_srlv |
                inst_jal   | inst_jalr  | inst_lb     | inst_lh   |
                inst_lw    | inst_lbu   | inst_lhu    | inst_lwl  |
                inst_lwr   | inst_lui   | move_to_reg ;
```

执行

主要由Alu或Shifter完成。

跳转类指令分为有条件跳转和无条件跳转，也分返回和不返回。无条件跳转指令一定跳转，有条件跳转指令需要进行数据的比较，符合要求才进行跳转。

```
assign rs_eq_rt     = rs_value == rt_value;
assign rs_neq_rt    = !rs_eq_rt;
assign rs_eq_zero   = rs_value == 32'b0;
assign rs_less_zero = rs_value[31];
assign rs_more_zero = !rs_value[31] && rs_value != 32'b0;
assign rt_eq_zero   = rt_value == 32'b0;

assign cond_br = inst_beq | inst_bne | inst_bgez | inst_blez | inst_bltz;

assign br_go = inst_bne  &  rs_neq_rt                  |
               inst_beq  &  rs_eq_rt                   |
               inst_bgez & !rs_less_zero               |
               inst_bgtz &  rs_more_zero               |
               inst_blez & (rs_eq_zero | rs_less_zero) |
               inst_bltz &  rs_less_zero               |
               inst_j                                  |
               inst_jal                                |
               inst_jr                                 |
               inst_jalr                               ;
```

跳转的目标地址的生成逻辑如下。

```
assign br_target = cond_br              ? (PC + {{14{imm[15]}}, imm, 2'b0} + 4)
  :
                   (inst_jr | inst_jalr) ? rs_value
   :
                   /*inst_jal | inst_j*/   {PC[31:28], jidx, 2'b0}
  ;
```

访存

只有访存指令有这一阶段。访存地址是Alu的结果最低两位抹零。

```
assign Address = {alu_result[31:2], 2'b0};
```

内存读写信号由指令执行情况直接生成。

```
assign MemWrite = inst_sw | inst_sb | inst_sh | inst_swl | inst_swr;

assign MemRead = inst_lw  | inst_lb  | inst_lbu | inst_lh |
                 inst_lhu | inst_lwl | inst_lwr ;
```

只要是LOAD类指令，不管是哪一条，地址确定的情况下，从内存中获取的数据都是确定的。因此，需要先根据各指令的需求对原始数据进行处理，再根据指令执行情况选择出需要的结果。

```
assign addr_low[0] = alu_result[1:0] == 2'b00;
assign addr_low[1] = alu_result[1:0] == 2'b01;
assign addr_low[2] = alu_result[1:0] == 2'b10;
assign addr_low[3] = alu_result[1:0] == 2'b11;

assign lb_lbu_origin = ({8{addr_low[0]}} & Read_data[ 7:0 ]) |
                       ({8{addr_low[1]}} & Read_data[15:8 ]) |
                       ({8{addr_low[2]}} & Read_data[23:16]) |
                       ({8{addr_low[3]}} & Read_data[31:24]) ;

assign lh_lhu_origin = ({16{addr_low[3] | addr_low[2]}} & Read_data[31:16]) |
                       ({16{addr_low[1] | addr_low[0]}} & Read_data[15:0 ]) ;

assign lb_result  = {{24{lb_lbu_origin[ 7]}}, lb_lbu_origin};

assign lbu_result = {24'b0, lb_lbu_origin};

assign lh_result  = {{16{lh_lhu_origin[15]}}, lh_lhu_origin};

assign lhu_result = {16'b0, lh_lhu_origin};

assign lw_result  = Read_data;

assign lwl_result = ({32{addr_low[0]}} & {Read_data[7:0 ], rt_value[23:0 ]}) |
                    ({32{addr_low[1]}} & {Read_data[15:0], rt_value[15:0 ]}) |
                    ({32{addr_low[2]}} & {Read_data[23:0], rt_value[ 7:0 ]}) |
                    ({32{addr_low[3]}} &  Read_data                         ) ;

assign lwr_result = ({32{addr_low[0]}} &  Read_data                         ) |
                    ({32{addr_low[1]}} & {rt_value[31:24], Read_data[31:8 ]}) |
```

```
                    ({32{addr_low[2]}} & {rt_value[31:16], Read_data[31:16]}) |
                    ({32{addr_low[3]}} & {rt_value[31:8 ], Read_data[31:24]}) ;


assign mem_result = {32{inst_lb}}  & lb_result  |
                    {32{inst_lbu}} & lbu_result |
                    {32{inst_lh}}  & lh_result  |
                    {32{inst_lhu}} & lhu_result |
                    {32{inst_lw}}  & lw_result  |
                    {32{inst_lwl}} & lwl_result |
                    {32{inst_lwr}} & lwr_result ;
```

同样地，对于STORE类指令，地址确定时从寄存器中获取的数据是唯一的，因此需要对数据预处理再进行选择。

写回

所有的控制信号和端口数据均已生成，由寄存器堆完成写回。