CODH - 寄存器堆与存储器及其应用 实验报告

院系: 姓名: 学号:

2023 年 7 月 13 日

1 实验目的

- 掌握寄存器堆 (Register File) 和存储器的功能、时序及其应用
- 熟练掌握数据通路和控制器的设计和描述方法

2 实验环境

- vlab.ustc.edu.cn
- Vivado 2016.3
- Nexys 4 DDR 开发板

3 实验内容

3.1 完成 32x32 位的寄存器堆的功能仿真

- 端口：ra1, rd1, ra2, rd2, wa, wd, we, clk
- 寄存器堆的 0 号寄存器内容恒定为零
- 寄存器堆的写优先的读操作模式

3.2 完成排序模块 (SRT) 的逻辑设计和功能仿真

- 端口：clk, rstn, run, done, cycles, addr, dout, din, we, clk_ld
- 存储器 0 号位置存放的是待排序数组的长度
- 需要注意与 SDU 的整合端口

之后将其与串行调试单元模块（SDU_DM）整合后，下载至 FPGA 中测试。

4 逻辑设计 / 核心代码

4.1 32x32 位的寄存器堆

4.1.1 逻辑设计

只需更改 ppt 中提供的代码即可，添加时序逻辑使 0 号寄存器的内容恒定为零。

至于写优先的读操作模式，由于此处由组合逻辑实现（读的永远是寄存器的最新值），不需要特别设置。但为了之后的实验还是加入了相关判断语句。

4.1.2 核心代码

```
reg [DWidth - 1:0] rf [0: (1 << AWidth) - 1]; //寄存器堆

always@(ra1, ra2)
    rf[0] = 0;

always @(posedge clk)
begin
    if (we && wa != 0) rf[wa] <= wd;    //写操作
end

assign rd1 = (ra1 == wa && we == 1 && wa != 0) ? wd : rf[ra1];   //写优先的读操作1
assign rd2 = (ra2 == wa && we == 1 && wa != 0) ? wd : rf[ra2];   //写优先的读操作2
```

reg.v

4.1.3 模块仿真

直接对默认的 32x32 REG 进行仿真，编写了相应的 test bench 进行测试，在 Vivado 仿真后得到波形图：

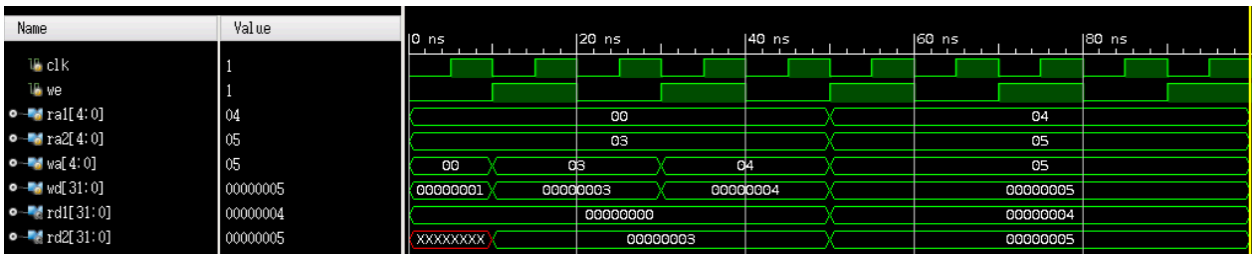

图 1: REG SIM 波形图

可以看到 0 号寄存器的内容恒定为零，写优先的读操作模式也正确。

4.2 排序（SRT）

4.2.1 逻辑设计

一开始不考虑与 SDU 的整合，只考虑排序模块的逻辑设计，并将使用 SDU 的调试读写阶段预先留出。

我们采用冒泡排序，先用高级语言写出后参照结构更改为 FSM：

```
    for (int i = 1; i <= n - 1; i++)
        for (int j = 1; j <= n - i + 1; j++)
        if(M[j] > M[j+1])
            { temp = M[j];  M[j] = M[j+1]; M[j+1] = temp; }
```

sort.c

由此可以画出以下状态转换图。

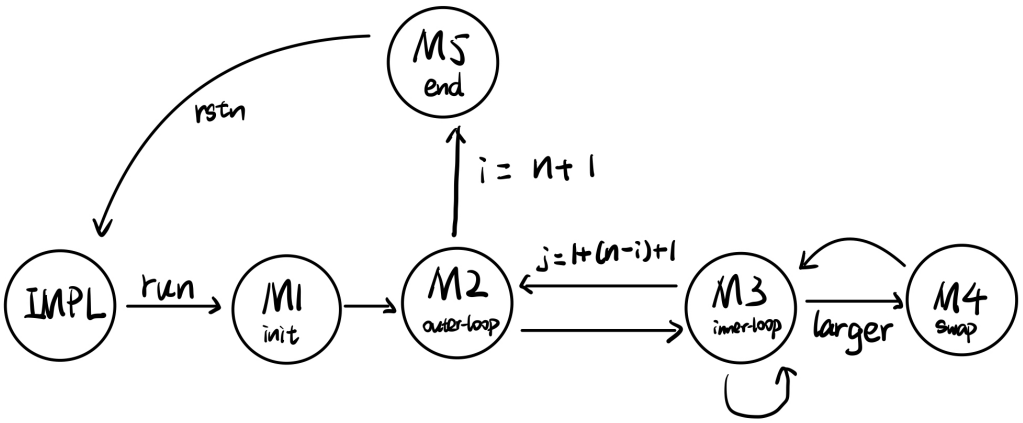

图 2: 状态转换图

IMPL 状态为默认状态，当复位信号有效时恢复到此状态，此时存储器的各信号切换为 SDU 输入的信号，可以使用 SDU 查看或设置存储器中的内容。

当按下按钮，run 信号有效，状态机切换至状态 M1 状态，在此状态做一些进入循环前的参数准备，例如将存储器的各型号切换为 SRT 内部信号。

接着切换至 M2 状态，此状态负责检查外层循环是否达到终止条件以及内层循环的参数设置（若 i = n + 1，下一步进入结束状态 M5）。

否则 M2 切换到 M3，进行内层循环，此状态负责检查内层循环是否达到终止条件以及比较相邻两个数的大小并交换（若 j = n - i + 2，下一步进入 M2，进行下一循环；若 M[j] > M[j+1]，设置写入信号将 M[j] 设为 M[j+1]，temp 存为 M[j]，然后切换到 M4 状态进行另一半切换）。

对于 M4 状态，设置写入信号将 M[j+1] 设为 temp（M[j]），然后切换回 M3 状态进行下一次内层循环。

对于 M5 结束状态，将存储器的各信号切换为 SDU 输入的信号，并且保持 cycle 信号不变显示状态数。若要进行下一次排序，通过按下 rstn 切换为 IMPL 再按下 run 即可。

4.2.2 核心代码

flag 信号用于切换来自 SDU 的信号还是 SRT 内部的信号。由于每次读相邻两个存储器的内容，因此将 dpra 端口链接为 a+1。由于在下一个时钟边沿到来时才会更新存储器的内容，为了实现写优先，设置了 real_spo 来标识写优先模式下读出的存储器内容。larger 信号则是由组合逻辑实现的比较相邻两个数的大小，若 M[j] > M[j+1]，则 larger 为 1，否则为 0。

具体信号数据设计参见代码及注释。

```
    ...
    // 次态切换
    always@(*) begin
    ns = cs;
    case (cs)
        IDLE: begin
            if (run) begin
                ns = M1;
            end
        end
        M1:
            ns = M2;
        M2: begin
            if (i == n + 1 | ~swaped)
                ns = M5;
            else
                ns = M3;
        end
        M3: begin
            if (a == n - i + 2)
                ns = M2;
            else if (larger)
                ns = M4;
            else
                ns = M3;
        end
        M4: begin
            ns = M3;
        end
        M5: begin    //finish, display cycles
            ns = M5;
        end
    endcase
    end

    assign dpra = a + 1; //dpra = j+1, a = j, spo=M[j], dpo=M[j+1]
    assign real_spo = we_srt ? d : spo;  //write first
    assign larger = (real_spo <= dpo) ? 0 : 1; //larger -> swap
    //组合逻辑设置信号
    always@(posedge clk or negedge rstn)
    begin
    if(~rstn)
    begin // default state for sdu
        a <= 0; //For MEM[0]
        finish <= 1;
        swaped <= 0;
        flag <= 0;
        cnt <= 0;
        n <= 0;
        i <= 1;
        cnt <= 0;
        we_srt <= 0;
        temp <= 0;
        d <= 0;
    end
    else
    case (cs)
    IDLE: begin // default state for sdu
```

```
        a <= 0; //For MEM[0]
        finish <= 1;
        swaped <= 0;
        flag <= 0;
        cnt <= 0;
        n <= 0;
        i <= 1;
        cnt <= 0;
        we_srt <= 0;
        temp <= 0;
        d <= 0;
    end

    M1: begin // init
        finish <= 0;
        n <= spo;
        i <= 1;
        swaped <= 1;
        flag <= 1;
        cnt <= cnt + 1;
    end

    M2: begin // outer-loop
        we_srt <= 0;
        a <= 1;
        swaped <= 0;
        cnt <= cnt + 1;
    end

    M3: begin // inner-loop
        if(a != n - i + 2 && larger) begin //-> M4 -> M3 to swap, M[j] = M[j+1], temp = M[j]
            swaped <= 1;
            d <= dpo;
            we_srt <= 1;
            temp <= real_spo;
        end
        else if(a == n - i + 2)begin // -> M2
            a <= 1; //a = j =1
            we_srt <= 0;
            i <= i + 1; //i = i + 1
            swaped <= 1;
        end
        else begin //-> M4, a = j = j + 1
            we_srt <= 0;
            a <= a + 1;
        end
        cnt <= cnt + 1;
    end

    M4: begin //swap
        a <= a + 1; // M[j+1] = temp = M[j], j = j + 1
        d <= temp;
        we_srt <= 1;
        cnt <= cnt + 1;
    end

    M5: begin //end
        finish <= 1;
        flag <= 0;
```

```
118     end
119
120     endcase
121     end
```

srt.v

4.2.3 模块仿真

将 ip 核使用 coe 文件初始化内部数据为 x10, xf, xe, xd, xc, xb, xa, x9, x8, x7, x6, x5, x4, x3, x2, x1, x0. 然后编写调用 srt 模块的 testbench，在 Vivado 进行仿真，测试结果如下（最终存储器内数据）

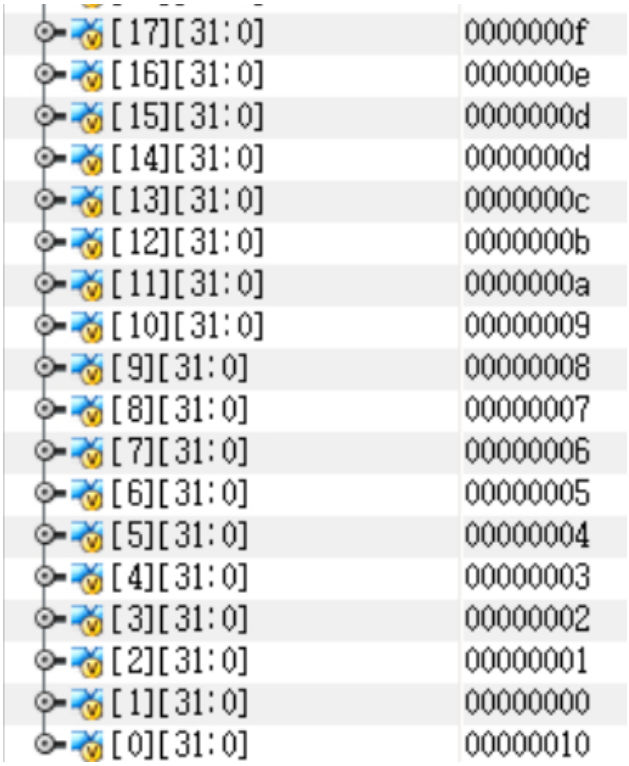

图 3: SRT SIM 结果图

可以发现 SRT 模块很好地完成了任务，将 16 个数字排序成了升序。

4.2.4 下载测试

首先需要编写取边沿，去抖动的模块，对 run 进行处理：

```
1   ...
2   always@(posedge clk)
3     begin
4         if (en == 0)
5             cnt <= 0;
6         else if (cnt < 16'h8000)
7             cnt <= cnt + 1;
8         en1 <= cnt[15];
9         en2 <= en1;
10    end
11  assign out = en1 & ~en2;
```

edge.v

然后编写 top 模块，并将 SDU 整合进去（只需将 SDU 相应的输出信号接入 SRT 对应的输入信号即可）。同时为了更直观的看到结果，用 16 位 LED 作为 cycles 的输出。其余显示并不必要，因为 SDU 可以方便的查看及改变存储器的内存数据

```
    sort_with_sdu srt_inst(
        .clk(clk),
        .rstn(rstn),
        .run(out),
        .done(done),
        .cycles(cycles),
        .addr(addr),
        .dout(dout),
        .din(din),
        .we(we),
        .clk_ld(clk_ld)
    );

    sdu_dm sdu_inst(
        .clk(clk),
        .rstn(rstn),
        .rxd(rxd),
        .txd(txd),
        .addr(addr),
        .dout(dout),
        .din(din),
        .we(we),
        .clk_ld(clk_ld)
    );
```

top.v

然后生成 bit 流进行烧写测试。一开始由于生成速度过慢导致难以发现某些错误，因此降低了时钟频率。最终在助教的帮助下发现了 top 模块内实例化的 srt 未加入 din 端口，这导致了大部分错误，将其改正后程序正确运行且通过了检查。

随机输入数据进行测试，结果如下图所示：

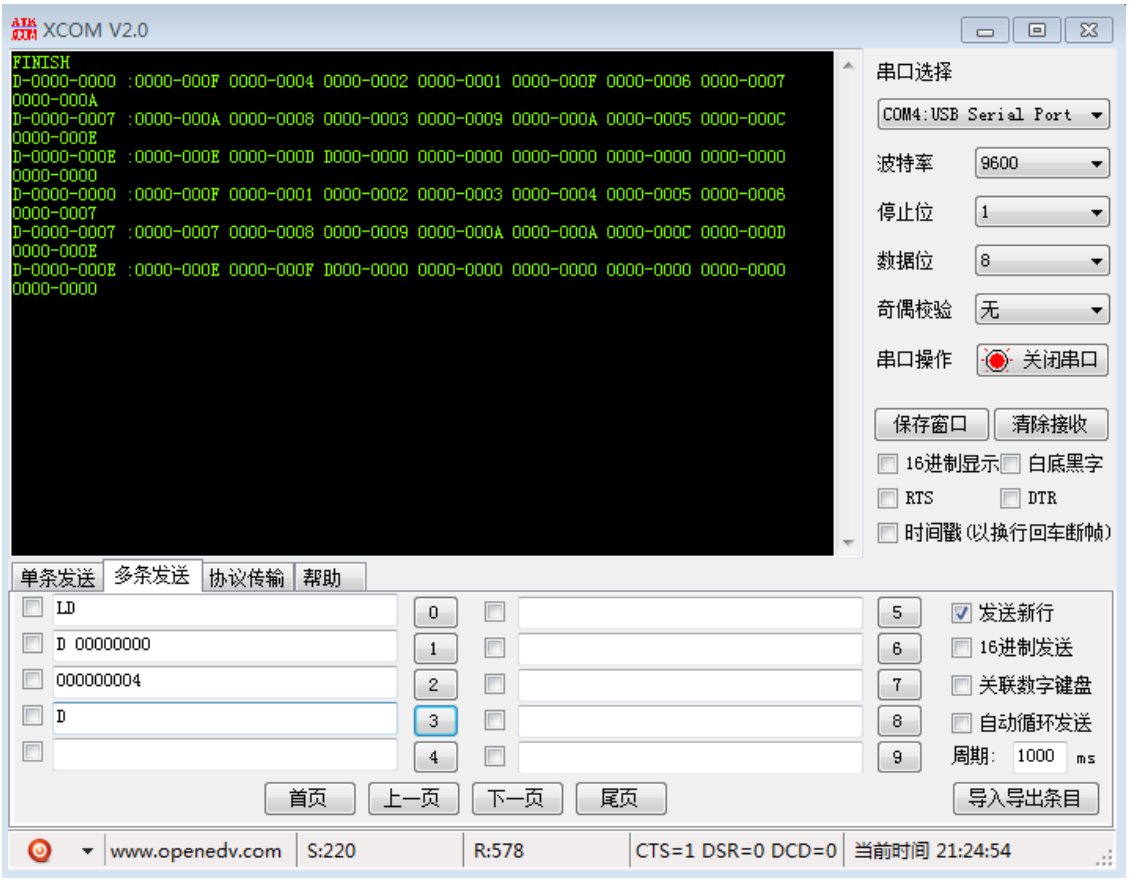

图 4: 实际结果图

4.2.5 结果分析

RTL 电路:

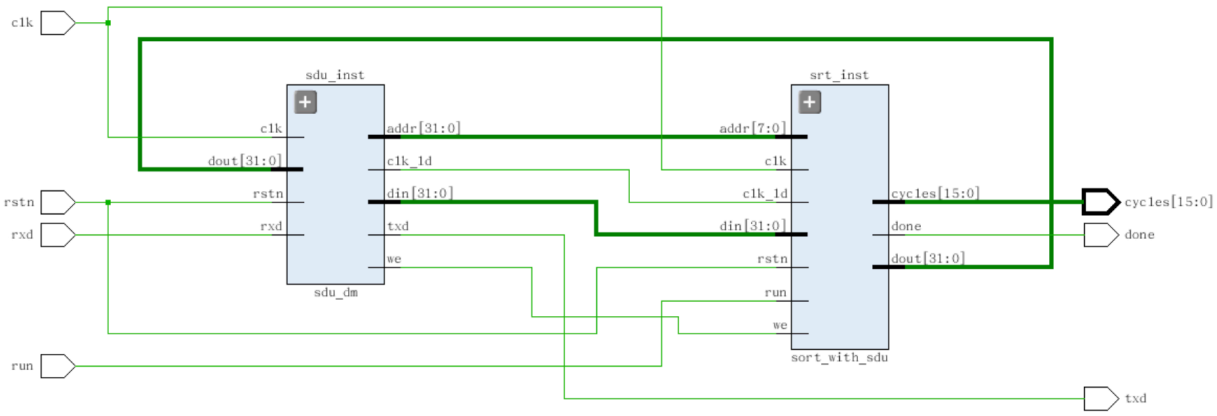

图 5: RTL

电路资源使用情况:

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Bonded IOB (210)	BUFGCTRL (32)
top	1924	852	164	22	2
sdu_inst (sdu_dm)	1311	629	36	0	0
SDU_wyl (SDU)	1311	629	36	0	0
inst1 (DIV_RX_CLK)	12	11	0	0	0
inst2 (RX)	38	30	0	0	0
inst3 (TX)	20	22	0	0	0
inst4 (DCP)	1232	566	36	0	0
srt_inst (sort_with_sdu)	613	223	128	0	0
mem_inst (dist_mem...	290	64	128	0	0
U0 (dist_mem_gen...	290	64	128	0	0

图 6: Util

时间性能报告:

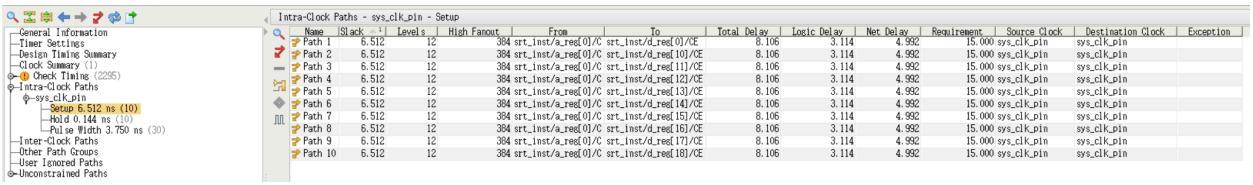

图 7: time

5 实验总结

1. 通过本次实验，我学习了如何使用串口与开发板进行通信，了解了用 SDU 的调试方法。
2. 对存储器的设计、结构、读写顺序有了更深刻的认识。
3. 我学会了如何从高级语言中的循环出发去设计合理的有限状态机。
4. 学习了查看错误信息来排查 bug 的方法，提高了调试程序效率。

6 意见/建议

大部分调试时间耗费在编写 bitstream 上，希望可以增加机房开放时间以供调试。