

## 1. 任务

**任务：**在模块**dec\_gpr\_ctl**中实现寄存器文件，并在模块**dec**中将其实例化（参见图7）。分析模块**dec\_gpr\_ctl**的Verilog代码及主要信号的仿真（位于文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv`中），以了解其工作方式。请注意，SweRV EH1处理器允许包含多个寄存器文件，但RVfpga系统中使用的配置仅使用一个寄存器文件（参见文件**dec.sv**的第402行：  
`localparam GPR_BANKS = 1;`）。

模块**dec**中的实例化：

```
525     dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526                  .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.*,
527                  // inputs
528                  .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529                  .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530                  .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531                  .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533                  .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534                  .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535                  .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537                  // outputs
538                  .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539                  .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540                  );
```

模块**dec\_gpr\_ctl**中32个寄存器的实现：

```
66     // GPR Write Enables for power savings
67     assign gpr_wr_en[31:1] = (w0v[31:1] | w1v[31:1] | w2v[31:1]);
68     for (genvar i=0; i<GPR_BANKS; i++) begin: gpr_banks
69         assign gpr_bank_wr_en[i][31:1] = gpr_wr_en[31:1] & {31{gpr_bank_id[GPR_BANKS_LOG2-1:0] == i}};
70         for (genvar j=1; j<32; j++) begin: gpr
71             rvdffe #(32) gprff (.*, .en(gpr_bank_wr_en[i][j]), .din(gpr_in[j][31:0]), .dout(gpr_out[i][j][31:0]));
72         end: gpr
73     end: gpr_banks
74
```

本例中仅实现了1个存储区。这一存储区通过将模块**rvdffe**（位于文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/lib/beh_lib.sv`中）实例化31次实现31个寄存器。注意每个**rvdffe**寄存器的宽度使用参数选择，本例中为32位→`rvdffe #(32)`。寄存器0不是必需的，因为RISC-V架构强制其始终为0。

寄存器读操作：

```
86     // GPR Read logic
87     for (int i=0; i<GPR_BANKS; i++) begin
88         for (int j=1; j<32; j++) begin
89             rd0[31:0] |= ((32{rden0 & (raddr0[4:0] == 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}) & gpr_out[i][j][31:0]);
90             rd1[31:0] |= ((32{rden1 & (raddr1[4:0] == 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}) & gpr_out[i][j][31:0]);
91             rd2[31:0] |= ((32{rden2 & (raddr2[4:0] == 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}) & gpr_out[i][j][31:0]);
92             rd3[31:0] |= ((32{rden3 & (raddr3[4:0] == 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}) & gpr_out[i][j][31:0]);
93         end
94     end
95
```

实现了4个读端口。每一个读端口均分配有寄存器的值（用**raddr0**/**raddr1**/**raddr2**/**raddr3**信号表示）。**rden0**/**rden1**/**rden2**/**rden3**信号允许/禁止读操作。注意**j**的初始值为1，因此寄存器0的读操作始终返回值0。

寄存器写操作：

```

96 // GPR Write logic
97 for (int j=1; j<32; j++ ) begin
98     w0v[j] = wen0 & (waddr0[4:0] == 5'(j) );
99     w1v[j] = wen1 & (waddr1[4:0] == 5'(j) );
100    w2v[j] = wen2 & (waddr2[4:0] == 5'(j) );
101    gpr_in[j] = ({32{w0v[j]}} & wd0[31:0]) |
102                ({32{w1v[j]}} & wd1[31:0]) |
103                ({32{w2v[j]}} & wd2[31:0]);
104 end
105 end // always_comb begin

```

实现了3个写端口。每个寄存器写入信号wd0/wd1/wd2中提供的值，具体取决于寄存器地址waddr0/waddr1/waddr2。wen0/wen1/wen2信号使能/禁止写操作。注意j的初始值为1，因此没有写入寄存器0。

**任务：**从图8分析多路开关的控制位。请注意，控制位在信号e3d中，该信号由信号dd经流水线处理得到，后一个信号由控制单元在译码阶段生成（有关控制位的说明，请参见SweRVref.docx）。


- 如果DC3阶段的指令有效（e3d.i0v == 1）且为load指令（e3d.i0load == 1），则选择来自LSU管道的值：i0\_result\_e3\_final = lsu\_result\_dc3。
- 如果EX3阶段的指令有效（e3d.i0v == 1）且为mul指令（e3d.i0mul == 1），则选择来自乘法器的值：i0\_result\_e3\_final = exu\_mul\_result\_e3。
- 否则，将选择来自I0管道的值：i0\_result\_e3\_final = i0\_result\_e3。

**任务：**从图9分析多路开关的控制位，这些控制位位于模块dec\_decode\_ctl中。

- 如果EX4阶段的结果必须从I0辅助ALU中选择（e4d.i0secondary == 1），则选择来自I0辅助ALU的值：i0\_result\_e4\_final = exu\_i0\_result\_e4。我们将在实验15中分析辅助ALU操作。
- 如果DC4阶段的指令有效（e4d.i0v == 1）且为load指令（e4d.i0load == 1），则选择来自LSU管道的值：i0\_result\_e4\_final = lsu\_result\_corr\_dc4。
- 否则，将选择来自I0管道的值：i0\_result\_e4\_final = i0\_result\_e4。

**任务：**按照以下步骤（如GSG第7部分中详述）在自己的计算机上重复图11和图12中的仿真过程：

- 必要时生成仿真二进制文件（Vrvfpgasim）。
- 在PlatformIO中，打开在以下位置提供的项目：  
[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram。

- 在文件`platformio.ini`中建立到RVfpga仿真二进制文件（`Vrvfpgasim`）的正确路径。
- 使用Verilator生成仿真轨迹（生成轨迹）。
- 使用GTKWave打开轨迹。
- 使用文件`test_1.tcl`和`test_2.tcl`（在`[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram`中提供）打开与图11和图12所示信号相同的信号。为此，在GTKWave上，单击“**File** → **Read Tcl Script File**”（文件 → 读取Tcl脚本文件），然后选择`test_1.tcl`或`test_2.tcl`文件。
- 单击几次“**Zoom In**”（放大）（）移动至48500 ps（或循环的任何其他迭代，第一次迭代除外）。

解答请参见实验11的主文档。

**任务：**按照GSG所述在Nexys A7板上执行图13中的程序。对于测量的四个事件，应获得图14所示的结果。解释并证明结果。

```
lui t2, 0xF4
add t2, t2, 0x240
nop

REPEAT:
    add t0, t0, 1
    add t3, t3, t1
    sub t4, t4, t1
    or t5, t5, t1
    xor t6, t6, t1
    bne t0, t2, REPEAT
```

程序由1000000次迭代的循环构成，该循环包含5条算术逻辑指令和一个条件分支。未发生由于冒险引起的暂停，因此：

- 执行6 \* 1000000条指令
- 每个周期执行2条指令，因此： $(6/2) * 1000000$ 个周期
- 执行1000000个分支，据预测几乎所有分支均命中。

**任务：**在图13所示程序的硬件计数器中测量其他事件。为此，必须使用`pspPerformanceCounterSet`函数在`Test.c`文件中更改待测量事件的配置。请注意，可以使用WD PSP文件中定义的宏引用不同的事件（如图1所示）：

`.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h`。

例如，如果要测量I\$未命中数而不是分支未命中数，则必须在文件中将`Test.c`行：

`pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);`

替换为行：`pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);`

不提供解答。

**任务：**在Test\_Assembly函数中提供其他程序并检查不同的事件是否提供了预期的结果。可以尝试其他指令，例如装载、存储、乘法、除法...以及引发流水线暂停的冒险。

不提供解答。