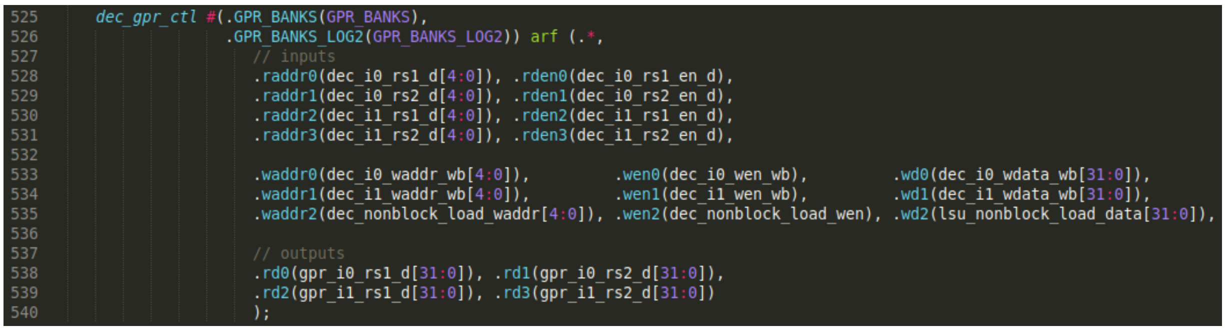
1. TASKS

TASK: The Register File is implemented in module **dec_gpr_ctl** and it is instantiated in module **dec** (see Figure 7). Analyse both the Verilog code and the simulation of the main signals of module **dec_gpr_ctl** (available in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv*), in order to understand how it works. Note that the SweRV EH1 processor allows the inclusion of several Register Files, but the configuration used in the RVfpga System only uses one Register File (see line 402 of file *dec.sv*: `localparam GPR_BANKS = 1;`).

Instantiation in module dec:

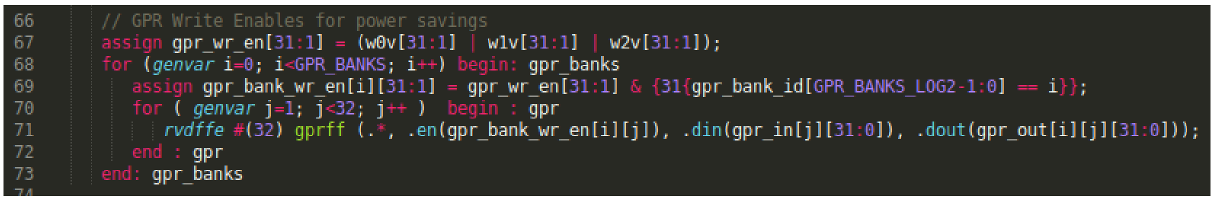


```
    dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
                  .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.*,
                    // inputs
                    .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
                    .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
                    .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
                    .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),

                    .waddr0(dec_i0_waddr_wb[4:0]),         .wen0(dec_i0_wen_wb),         .wd0(dec_i0_wdata_wb[31:0]),
                    .waddr1(dec_i1_waddr_wb[4:0]),         .wen1(dec_i1_wen_wb),         .wd1(dec_i1_wdata_wb[31:0]),
                    .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),

                    // outputs
                    .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
                    .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
                    );
```

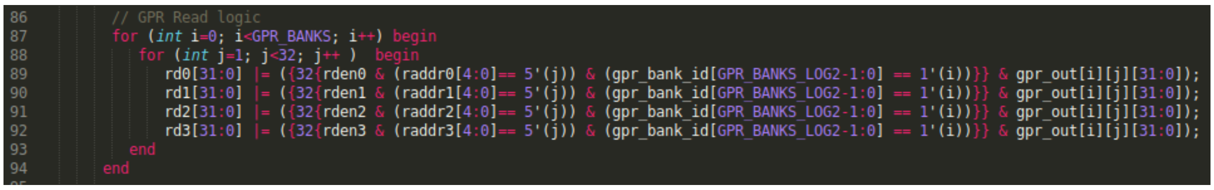
Implementation of the 32 registers in module dec_gpr_ctl:



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

In our case, only 1 bank is implemented. For this single bank, 31 registers are implemented by instantiating 31 times module **rvdffe** (which you can find in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/lib/beh_lib.sv*). Note that the width of each **rvdffe** register is selected using a parameter, which in our case is 32 bits → rvdffe #(32). Register 0 is not necessary as RISC-V architecture forces it to be always 0.

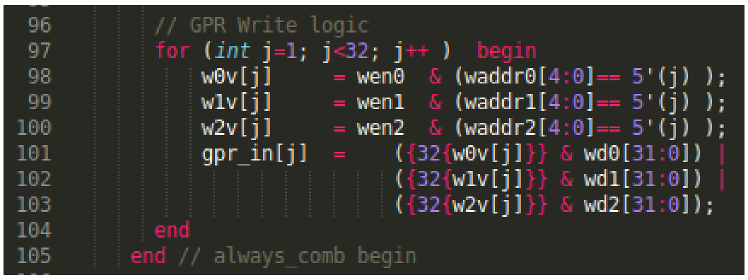
Register reading:



```
      // GPR Read logic
      for (int i=0; i<GPR_BANKS; i++) begin
         for (int j=1; j<32; j++ )  begin
            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]);
            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]);
            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]);
            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]);
         end
      end
```

4 read ports are implemented. Each one is assigned with the value of the register indicated by the `raddr0/raddr1/raddr2/raddr3` signals. The `rden0/rden1/rden2/rden3` signals enable/disable the reading. Note that the initial value of *j* is 1, thus the reading of register 0 always returns the value 0.

Register writing:



```
    // GPR Write logic
    for (int j=1; j<32; j++ )  begin
        w0v[j]     = wen0  & (waddr0[4:0]== 5'(j) );
        w1v[j]     = wen1  & (waddr1[4:0]== 5'(j) );
        w2v[j]     = wen2  & (waddr2[4:0]== 5'(j) );
        gpr_in[j]  =    ({32{w0v[j]}} & wd0[31:0]) |
                        ({32{w1v[j]}} & wd1[31:0]) |
                        ({32{w2v[j]}} & wd2[31:0]);
    end
  end // always_comb begin
```

3 write ports are implemented. Each register is written with the value provided in signals `wd0/wd1/wd2`, depending on the register address `waddr0/waddr1/waddr2`. The `wen0/wen1/wen2` signals enable/disable the writing. Note that the initial value of *j* is 1, thus there is no write of register 0.

TASK: Analyse the control bits of the multiplexer from Figure 8. Note that the control bits are in signal `e3d`, which was pipelined from signal `dd`, which was generated in the Decode stage by the Control Unit (see SweRVref.docx for descriptions of the control bits).

- If the instruction at DC3 is valid (`e3d.i0v` == 1) and it is a `load` instruction (`e3d.i0load` == 1), then the value coming from the LSU Pipe is selected: `i0_result_e3_final` = `lsu_result_dc3`.

- If the instruction at EX3 is valid (`e3d.i0v` == 1) and it is a `mul` instruction (`e3d.i0mul` == 1), then the value coming from the Multiplier is selected: `i0_result_e3_final` = `exu_mul_result_e3`.

- Otherwise, the value coming from the I0 Pipe is selected: `i0_result_e3_final` = `i0_result_e3`.

TASK: Analyse the control bits of the multiplexer from Figure 9, which you can find in module **dec_decode_ctl**.

- If the result at EX4 must be selected from the I0 Secondary ALU (`e4d.i0secondary` == 1), then the value coming from the I0 Secondary ALU is selected: `i0_result_e4_final` = `exu_i0_result_e4`. We analyse the Secondary ALU operation in Lab 15.

- If the instruction at DC4 is valid (`e4d.i0v` == 1) and it is a `load` instruction (`e4d.i0load` == 1), then the value coming from the LSU Pipe is selected: `i0_result_e4_final` = `lsu_result_corr_dc4`.

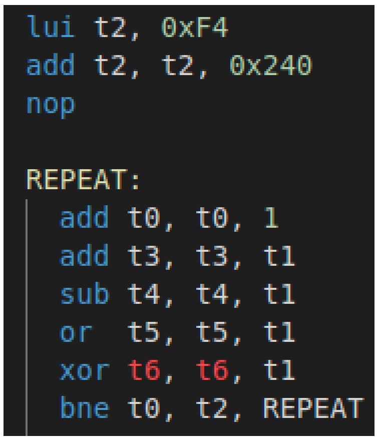
- Otherwise, the value coming from the I0 Pipe is selected: `i0_result_e4_final` = `i0_result_e4`.

TASK: Replicate the simulation from Figure 11 and Figure 12 on your own computer by following these steps (as described in detail in Section 7 of the GSG):
- If necessary, generate the simulation binary (*Vrvfpgasim*).
- In PlatformIO, open the project provided at:

[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram.
- Establish the correct path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace using GTKWave.
- Use files *test_1.tcl* and *test_2.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram*) for opening the same signals as the ones shown in Figure 11 and Figure 12. For that purpose, on GTKWave, click on *File → Read Tcl Script File* and select the *test_1.tcl* or *test_2.tcl* file.
- Click on *Zoom In* ( ) several times and move to 48500ps (or any other iteration of the loop, except the first one).

Solution provided in the main document of Lab 11.

TASK: Execute the program from Figure 13 on the Nexys A7 board as explained in the GSG. You should obtain the results shown in Figure 14 for the four measured events. Explain and justify the results.



```
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
```

The program is made by a 1000000 iterations loop that includes 5 Arithmetic-Logic instructions plus a conditional branch. There are no stalls due to hazards, thus:
- 6 * 1000000 instructions are executed
- 2 instructions are executed per cycle, thus: (6/2) * 1000000 cycles
- 1000000 branches are executed and almost all of them hit in the prediction.

TASK: Measure other events in the Hardware Counters for the program from Figure 13. For this purpose, you must change in file *Test.c* the configuration of the events to be measured with function `pspPerformanceCounterSet`. Note that the different events (shown in Table 1) can be referenced using the macros defined in WD's PSP file: *.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h*. For example, if you want to measure the number of I$ misses instead of the number of branch misses, you must substitute in file *Test.c* line: `pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);`
for line: `pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);`

Solution not provided.

<u>TASK</u>: Propose other programs in the the `Test_Assembly` function and check if the different events provide the expected results. You can try other instructions such as loads, stores, multiplications, divisions… as well as hazards that provoke pipeline stalls.

Solution not provided.