Elec3608 assignment report

The single-cycle RISCV processor for a Xorshift linear feedback shift register

Introduction

The processor is an integrated electronic circuit that can have some calculations by running a computer. The processor can perform several functions and have different capabilities:

A) At a given time to perform instructions.

B) Measuring the maximum numbers of bits or instructions.

C) relative clock speed.

The single-cycle RISCV processor is a processor that base on RISCV. The RISCV is a hardware instruction set architecture (ISA) and based on Reduced instruction Set Computer (RISC) principles and the processor performs one instruction in a clock cycle. The processor prepares for a Xorshift linear feedback shift register (LFSR) that generates pseudo-random numbers by an exclusive-or (Xor) and a linear function. The LFSR is popular in hardware and software implementations.

Part one Datapath of processor and different instructions

Datapath of processor

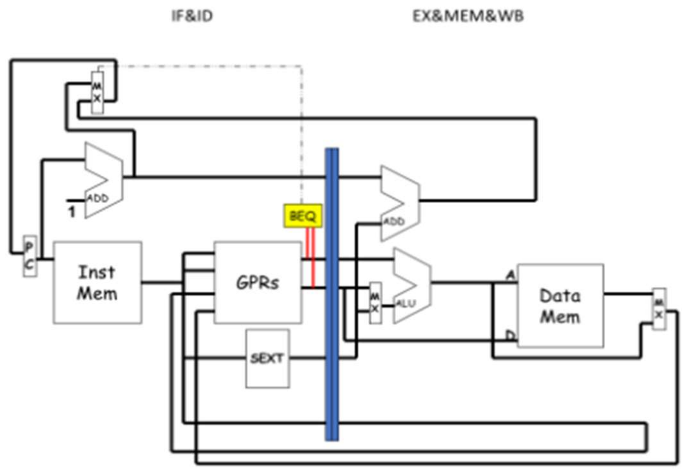


Figure 1

The Figure1 is showing the Datapath of the processor. In this processor, there are three 4 Register instructions (register file), three Arithmetic Logic Units (ALU) and several multiplexers. The instruction memory outputs one instruction according to the number of pc, then generate different signals (address of Rs1, Rs2, Rd, the number of imm, opcode function3, function7, depends on the instruction). Then the imm have a sign extension. Some unsigned number need zero extension(lui,lhu) to 32 bits in the "sext" part. The address of Rs1, Rs2 and Rd is the input of "GPRs", the get the data of Rs1 and Rs2, the Rs1 is one input of ALU, the Rs2 is one input of op2sel mux, then compare will imm, select one signal as the other input ALU.

The ALU generates an answer according to function 3 or the ALU function to have a calculation. The output of ALU is also the input of data memory (Dmem). Dmem will output data that stores in the register of the input address. When the signal of write enable of Dmem turns on, the data of rs2 will store in the register of rs2. Then the wbsel mux selects a data between Dmem output, alu output, rs2, lui_imm(the imm for lui instruction) and (pc+4)(that is for jalr) to write back to Rd(destination register), when the signal of the write enables of "GPRs" turns on. Another mux is for pc increasement. The input of pcsel mux is "pc+4" (normal session), "pc+branch_imm" (branch instruction) and "rs1+jalr_imm" (jalr instruction), and the output is the number of PC. The above is one cycle of the processor.

Register file

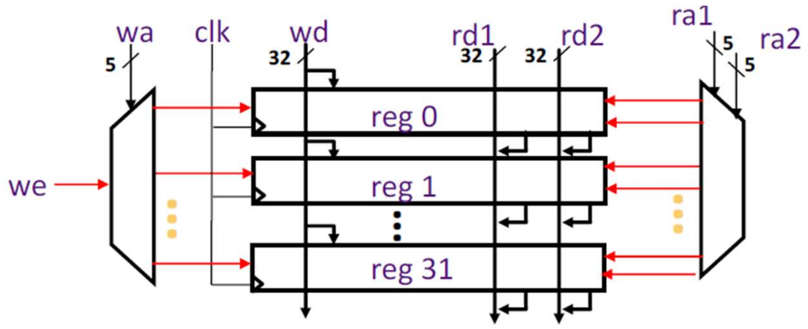
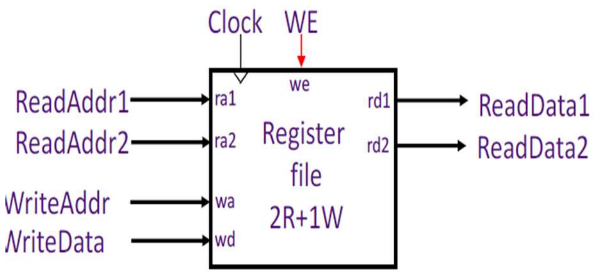
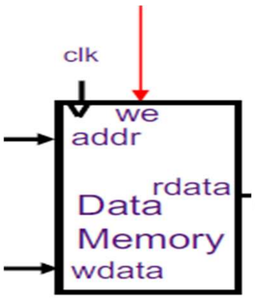


Figure 2

Figure2 is showing the Register file. Register file is an array of processor registers in a processor. In the datapath, the is an instruction memory, a GPRs, a pipeline regfile, data memory and program counter (PC). The PC is also called instruction pointer (IP) that is a register to indicate the program sequence of the computer. The instruction memory (IMEM) is a register file that stores the instruction set. When the IMEM receives the signal from PC, the IMEM push the next instruction according to the number of PC. The GPRS is a register file to store the data of destination register (rd) and load the data of two resource registers (rs1, rs2) and there are 64 registers (for each resource register is 32 registers).

According to Figure 3, there are 8 ports for the whole regfile. The ra1 and ra2 receive the address of two resource registers and the rd1 and rd2 are output which are the data of the resource registers. The writeAddr (wa) is an input port that is the address of destination register and when we turn on, the writedata(wd) store the data of register. The data memory works for the instruction of store and load and there are 32 registers in DMEM. According to the Figure 4, there are 5 ports, by the addr to get the address of the register, the rdata output the data of the register. When the WE turns on, the wdata put the data in the register.





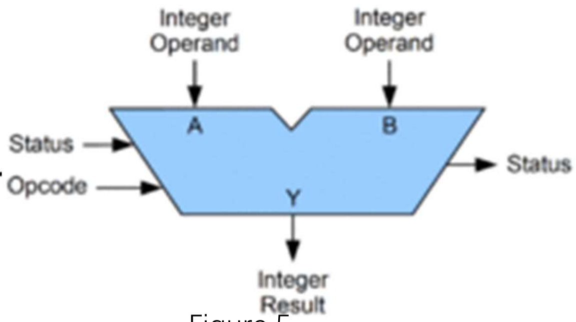


Figure 3 Figure 4 Figure 5

The last one is the pipeline register file. The array length of the pipeline register file is 4 and those stores the data of rs1 and rs2, the number of PC and the instruction number. More detail about the pipeline will be discussed in the next part.

The Arithmetic Logic Units (ALU)

There are 3 ALU in the datapath. The main one is for the instruction set, and others are for pc increasement.

The Multiplexer (MUX)

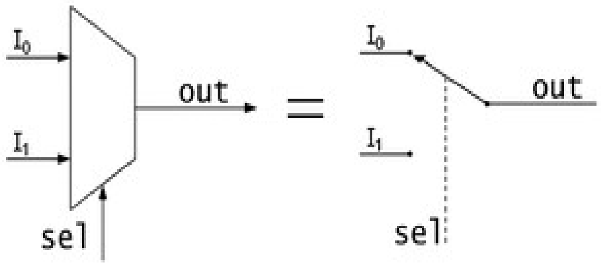
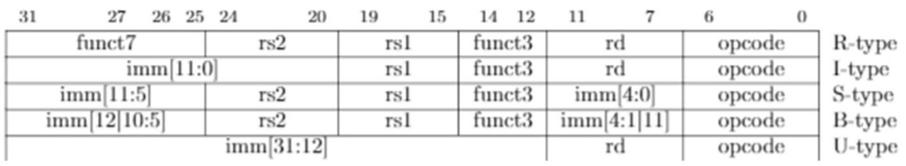


Figure 6

The multiplexer (MUX) is a device that selects between several input signals and forwards it to a single output line. In the Datapath, there are 3 muxes. The first one is op2sel that selects data for the second alu input(alu_b). The second one is wbsel that selects data for writing back to the destination register. The third one is PCsel that selects data for pc.

Result of question A

According to the instruction set, the question one is that it generates a random number by xor, shift left logical and shift right logical and repeat 30 times to get 30 outputs. The instruction set includes 4 types that are I-type, R-type, SB-type and U-type.

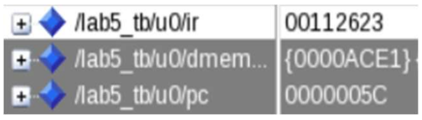


31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type

Figure 7

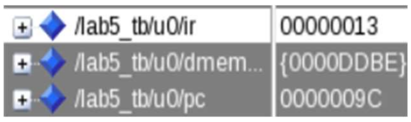
I-type and R type are for addi, slli, srli, load and jalr (jr), SB-type is for BNE and store and U-type is for AUIPC and LUI. The first data is x" ACE1", then generates the random numbers by the Xorshift LFSR. Figure 8 is showing the starting of instruction that starts with the primary function of the instruction.

Then the instruction set keeps running to function L3, then jump to lfsr3 that is the calculation of Xorshift LFSR. After the first calculation, the processor generates the first output x“DDBE” (showing in Figure 9). Then go back to L3, there is a loop for 30 times by the instruction “BNE”. After the 30 times, the processor generates the last output x”6041”, That is the result of question A (shown in Figure 10). The time of the end of the simulation is 5225000 ps. (Show in Figure 11).



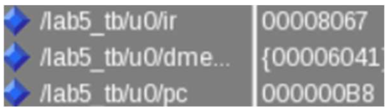
```
/lab5_tb/u0/ir       00112623
/lab5_tb/u0/dmem...  {0000ACE1}
/lab5_tb/u0/pc       0000005C
```

Figure 8



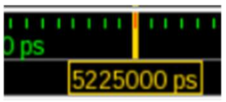
```
/lab5_tb/u0/ir       00000013
/lab5_tb/u0/dmem...  {0000DDBE}
/lab5_tb/u0/pc       0000009C
```

Figure 9



```
/lab5_tb/u0/ir       00008067
/lab5_tb/u0/dme...   {00006041}
/lab5_tb/u0/pc       000000B8
```

Figure 10

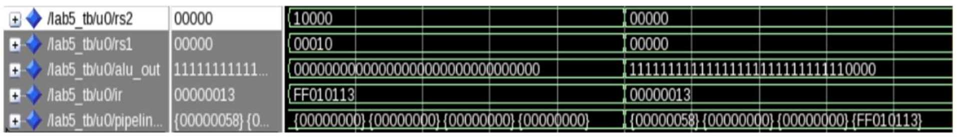


```
0 ps
5225000 ps
```

Figure 11

Part two improve performance and two-stage pipeline

When the processor executes one instruction in RISC-V, there are 5 steps (Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory operation (MEM) and Write back (WB)). In this processor, the processor divided into two parts which are IF&ID and EX&MEM&WB, so there is a pipeline register file between the two parts. The pipeline regfile stores 4 signals which are the data of rs1, rs2, pc and Ir (instruction code). The pc stored in the pipeline is after pc increasement(pc+4), and Ir is preparing for an opcode, Imm and function3. The working process for the pipeline is that Imem outputs the instruction then decodes the instruction when the PC generates a number to Imem. After decoding, the regfile have two outputs, rs1 and rs2, store the outputs into the pipeline



```
/lab5_tb/u0/rs2       00000           10000                                    00000
/lab5_tb/u0/rs1       00000           00010                                    00000
/lab5_tb/u0/alu_out   11111111111...  00000000000000000000000000000000         11111111111111111111111111110000
/lab5_tb/u0/ir        00000013        FF010113                                 00000013
/lab5_tb/u0/pipelin...{00000058} {0...{00000000} {00000000} {00000000} {00000000} {00000058} {00000000} {00000000} {FF010113}
```

regfile, in this cycle. In the next cycle, the pc has added 4 and the Imem will generate next instruction. At the same time, the pipeline outputs the signals from the last cycle and continue running the EX&MEM&WB of last instruction. The data write back to the rd of regfile. The simulation shows in Figure 12.

Part 3 hazard of pipeline

In this instruction set, if there is a pipeline in the processor, the processor has the data hazard. There are two ways to solve hazard, the stalling and Data Forwarding. The stalling is waiting for the result to be available by freezing earlier pipeline stages that are adding nop after the instruction. The data forwarding is that checks the source register with the destination register from last instruction, if they are same, directly change the data of source register to ALU out. There is two way to control a hazard.

Conclusion

Following the whole assignment, the processor could simulate different type instruction set, and successfully compiles and generates the correct answer for Xor shift LFSR. Processor has improved the performance by adding a pipeline register file between two stages. Furthermore, it becomes a two-staged pipeline processor and controls the hazard by stalling and data forwarding.

Appendix

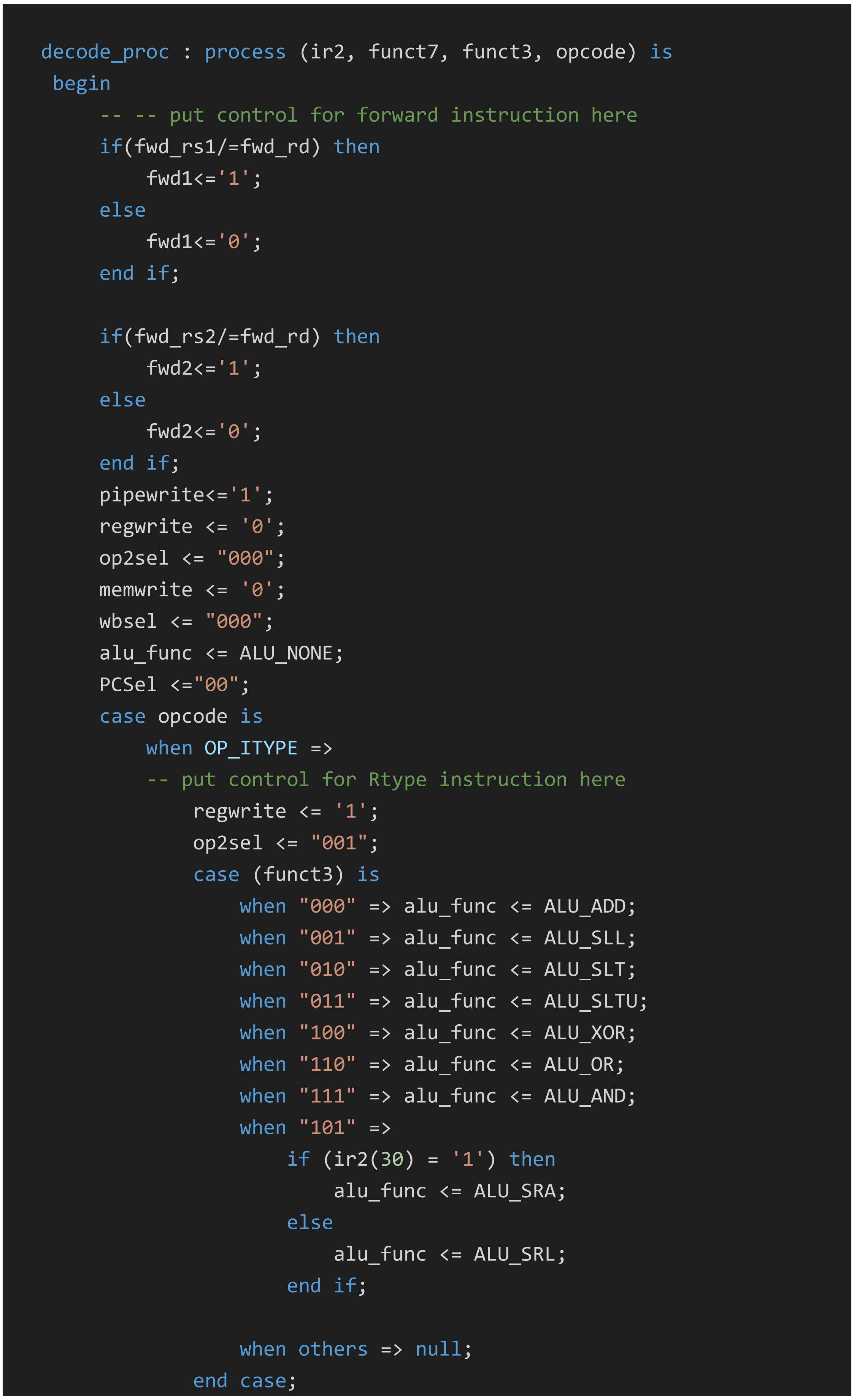
Code

Assignment.vhd

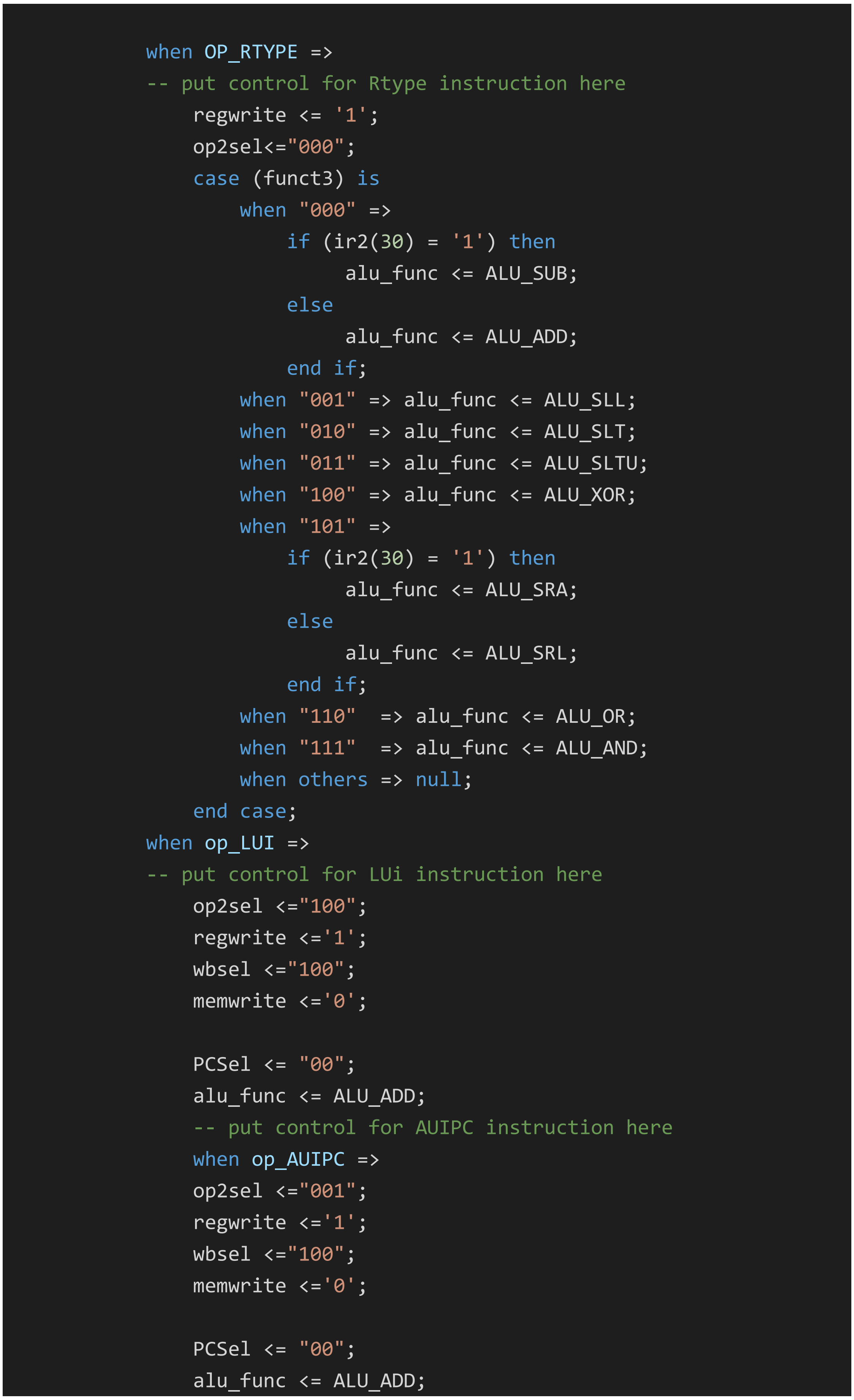
```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


library work;
use work.common.all;

entity assignment is
    port (reset : in  std_logic;
          clk   : in  std_logic;
             y : out word);
end assignment;

architecture behavioral of assignment is

signal alu_func : alu_func_t := ALU_NONE;
signal load_type : load_type_t :=LOAD_NONE;
signal alu_A : word := x"00000000"; -- one input of alu
signal alu_B : word := x"00000000"; -- one input of alu
signal alu_out : word := x"00000000"; -- output of alu

signal reg_B : word := x"00000000"; -- the data of register rs2
signal regb_out : word := x"00000000"; -
- the output of pipeline for the data of rs2
signal reg_A : word := x"00000000";   --the data of register rs1
signal fwd_A : word := x"00000000";   --the output of mux for forward
signal fwd_B : word := x"00000000";  --the output of mux for forward
signal imm : word := x"00000000";    -- the imm for I type

signal imm_rd : word := x"00000000"; -- the imm for R type

signal ir : word := x"00000000";  -- the instruction
signal ir2 : word := x"00000000"; -- the instruction put into pipeline
signal dmem_out : word := x"00000000"; -- the output of dmem
signal rf_wdata : word := x"00000000"; -- the data of write back
signal lu_imm : word := x"00000000";
signal fwd_rs1 : word := x"00000000";
signal fwd_rs2 : word := x"00000000";
signal fwd_rd : word := x"00000000";
```

```vhdl
signal branch_imm : unsigned(word'range) := x"00000000";

signal jalr_imm : unsigned(word'range) :=x"00000000";

signal lhu_imm : unsigned(word'range) :=x"00000000";

signal pc2 : unsigned(word'range) := x"00000000";
signal pc3 : unsigned(word'range) := x"00000000";

-- instruction fields
signal opcode : opcode_t;
--signal opcode_out1 : opcode_t;
signal funct3 : std_logic_vector(2 downto 0);
--signal func3 : std_logic_vector(2 downto 0);
signal funct7 : std_logic_vector(6 downto 0);
signal rs1 : std_logic_vector(4 downto 0);
signal rs2 : std_logic_vector(4 downto 0);
signal rd : std_logic_vector(4 downto 0);
--signal fwd_rd : std_logic_vector(4 downto 0);
signal pc : unsigned(word'range) := x"00000000";

-- control signals
signal regwrite : std_logic;
signal pipewrite : std_logic;
signal fwd1 : std_logic;
signal fwd2 : std_logic;
signal wbsel : std_logic_vector(2 downto 0);
signal memwrite : std_logic;
signal op2sel : std_logic_vector(2 downto 0);
signal PCSel : std_logic_vector(1 downto 0);



component alu is
port (alu_func : in  alu_func_t;
        op1      : in  word;
        op2      : in  word;
        result   : out word);
end component alu;

component pipeline is
port (reset : in  std_logic;
    clk   : in  std_logic;
    pc_in  : in word;
    rd1_in : in word;
    rd2_in : in word;
    ir_in : in word;
    pc_out  : out word;
    rd1_out : out word;
    rd2_out : out word;
    ir_out : out word;
    we    : in  std_logic);
    end component pipeline;
component imem is
port(
    addr : in std_logic_vector(5 downto 0);
    dout : out word);
end component imem;

component dmem is
port (reset : in  std_logic;
      clk   : in  std_logic;
      raddr : in  std_logic_vector(5 downto 0);
      dout  : out word;
      waddr : in  std_logic_vector(5 downto 0);
      din : in  word;
      we    : in  std_logic);
end component dmem;

component regfile is
port (reset : in  std_logic;
      clk   : in  std_logic;
      addra : in  std_logic_vector(4 downto 0);
      addrb : in  std_logic_vector(4 downto 0);
      rega  : out word;
      regb  : out word;
      addrw : in  std_logic_vector(4 downto 0);
      dataw : in  word;
      we    : in  std_logic);
end component regfile;

begin
    -- datapath
    alu0: alu port map(
        alu_func => alu_func,
            op1 => alu_A,
            op2 => alu_B,
            result => alu_out);
    imem0: imem port map(
            addr => std_logic_vector(pc(7 downto 2)),
            dout => ir);

    dmem0: dmem port map(
    reset => reset,
        clk => clk,
        raddr => alu_out(7 downto 2),
        dout => dmem_out,
        waddr => alu_out(7 downto 2),
        din => regb_out,
        we => memwrite);

    rf0: regfile port map(
    reset => reset,
        clk => clk,
        addra => rs1,
        addrb => rs2,
        rega => reg_A,
        regb => reg_B,
        addrw => rd,
        dataw => rf_wdata,
        we => regwrite);
    pipeline0: pipeline port map(
        reset=> reset,
        clk  => clk,
        pc_in=>word(pc2),
        rd1_in=>fwd_A,
        rd2_in=>fwd_B,
        ir_in=> ir,
        unsigned(pc_out) =>pc3,
        rd1_out=>alu_A,
        rd2_out=>regb_out,
        ir_out=> ir2,
        we  => pipewrite);

    alu_B <= regb_out when op2sel = "000" else
            imm when op2sel = "001" else
            imm_rd when op2sel ="010" else
            word(lhu_imm) when op2sel ="011";
    -- the mux for alu_b
    rf_wdata <= alu_out when wbsel = "000" else
        word(lu_imm) when wbsel ="100" else
        regb_out when wbsel = "001" else
```

```vhdl
        dmem_out when wbsel ="010" else
        word(pc3+4) when wbsel="011";
    -- the mux for write data back
    pc2 <=pc+4;-- that is the input for pipeline
    fwd_A <= alu_out when fwd1 ='0' else
        reg_A;
    --The mux for the forward
    fwd_B <= alu_out when fwd2 ='0' else
        reg_B;
    --The mux for the forward

    -- instruction fields
    imm(31 downto 12) <= (others => ir2(31)); --I type sign extend
    imm(11 downto 0) <= ir2(31 downto 20);
    imm_rd(31 downto 12) <= (others => funct7(6));-- R type
    imm_rd(11 downto 5) <= funct7;
    imm_rd(4 downto 0) <= rd;
    rs1 <= ir(19 downto 15); -- the address of rs1
    rs2 <= ir(24 downto 20); -- the address of rs2
    rd <= ir2(11 downto 7);  -- the address of rd

    funct3 <= ir2(14 downto 12);
    funct7 <= ir2(31 downto 25);
    opcode <= ir2(6 downto 0);
    --the sign extend and zero extend
    branch_imm(31 downto 13) <= (others => ir2(31));
    branch_imm(12 downto 0) <= unsigned(ir2(31) & ir2(7) & ir2(30 downto 25) & ir2(11 downto 8) & '0');
    --The B type for branch eq
    jalr_imm(31 downto 12) <= (others => ir2(31));
    jalr_imm(11 downto 0) <= unsigned(ir2(31 downto 20));
    --The I type for jalr
    lu_imm(11 downto 0) <= (others => '0');
    lu_imm(31 downto 12) <= ir2(31 downto 12);
    --The U type for lui
    lhu_imm(15 downto 0)<= unsigned(imm(15 downto 0));
    lhu_imm(31 downto 16)<=(others =>'0');
    --the lhu imm convert to unsigned number
    fwd_rs1(4 downto 0)<= ir(19 downto 15);
    fwd_rs1(31 downto 5)<=(others =>'0');
    fwd_rs2(4 downto 0)<= ir(24 downto 20);
    fwd_rs2(31 downto 5)<=(others =>'0');
    fwd_rd(4 downto 0)<= ir2(11 downto 7);
    fwd_rd(31 downto 5)<=(others =>'0');
```



```
decode_proc : process (ir2, funct7, funct3, opcode) is
 begin
     -- -- put control for forward instruction here
     if(fwd_rs1/=fwd_rd) then
         fwd1<='1';
     else
         fwd1<='0';
     end if;

     if(fwd_rs2/=fwd_rd) then
         fwd2<='1';
     else
         fwd2<='0';
     end if;
     pipewrite<='1';
     regwrite <= '0';
     op2sel <= "000";
     memwrite <= '0';
     wbsel <= "000";
     alu_func <= ALU_NONE;
     PCSel <="00";
     case opcode is
         when OP_ITYPE =>
         -- put control for Rtype instruction here
             regwrite <= '1';
             op2sel <= "001";
             case (funct3) is
                 when "000" => alu_func <= ALU_ADD;
                 when "001" => alu_func <= ALU_SLL;
                 when "010" => alu_func <= ALU_SLT;
                 when "011" => alu_func <= ALU_SLTU;
                 when "100" => alu_func <= ALU_XOR;
                 when "110" => alu_func <= ALU_OR;
                 when "111" => alu_func <= ALU_AND;
                 when "101" =>
                     if (ir2(30) = '1') then
                         alu_func <= ALU_SRA;
                     else
                         alu_func <= ALU_SRL;
                     end if;

                 when others => null;
             end case;
```



```vhdl
when OP_RTYPE =>
-- put control for Rtype instruction here
    regwrite <= '1';
    op2sel<="000";
    case (funct3) is
        when "000" =>
            if (ir2(30) = '1') then
                 alu_func <= ALU_SUB;
            else
                 alu_func <= ALU_ADD;
            end if;
        when "001" => alu_func <= ALU_SLL;
        when "010" => alu_func <= ALU_SLT;
        when "011" => alu_func <= ALU_SLTU;
        when "100" => alu_func <= ALU_XOR;
        when "101" =>
            if (ir2(30) = '1') then
                 alu_func <= ALU_SRA;
            else
                 alu_func <= ALU_SRL;
            end if;
        when "110"  => alu_func <= ALU_OR;
        when "111"  => alu_func <= ALU_AND;
        when others => null;
    end case;
when op_LUI =>
-- put control for LUi instruction here
    op2sel <="100";
    regwrite <='1';
    wbsel <="100";
    memwrite <='0';

    PCSel <= "00";
    alu_func <= ALU_ADD;
    -- put control for AUIPC instruction here
    when op_AUIPC =>
    op2sel <="001";
    regwrite <='1';
    wbsel <="100";
    memwrite <='0';

    PCSel <= "00";
    alu_func <= ALU_ADD;
```

```
            when OP_JALR =>
            -- put control for jalr instruction here
                op2sel <="000";
                regwrite <='1';
                wbsel <="011";
                memwrite <='0';

                PCSel <= "10";
                alu_func <= ALU_ADD;

            when OP_LOAD =>
            -- put control for load instruction here
                regwrite <= '1';

                memwrite <= '1';
                wbsel <= "010";
                PCSel <="00";
                alu_func <= ALU_ADD;
                case (funct3) is
                    when "010" => load_type <= LW;
                        op2sel <= "001";

                    when "101" => load_type <= LHU;
                        op2sel <= "011";
                    when others => null;
                end case;



            when OP_STORE =>
                -- put control for store instruction here
                regwrite <= '1';
                op2sel <= "010";
                memwrite <= '1';
                wbsel <= "001";
                PCSel <="00";

                alu_func <= ALU_ADD;



            when OP_BRANCH =>
                -- put control for branch instruction here
```

```vhdl
                regwrite <= '0';
                wbsel<="000";
                memwrite <= '0';
                --muxsel<="00";
                op2sel <="000";
                alu_func<= ALU_NONE;
                if (alu_A /=regb_out) then
                    PCSel <= "01";

                else
                    PCSel <="00";
                end if;

            when others => null;
        end case;
    end process;

    y <= alu_out;

    acc: process(reset, clk)
    begin
        if (reset = '1') then

            pc <= x"00000054";

        elsif rising_edge(clk) then

            if (PCSel ="01") then
                pc<= pc3 + branch_imm;
            elsif(PCSel ="00") then
                pc <= pc + 4;
            elsif(PCSel="10") then
                pc<=(unsigned(alu_A)+jalr_imm);
            end if;
        end if;
    end process;
end architecture;
```

alu.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.common.all;

```

```vhdl
entity alu is
    port (alu_func : in  alu_func_t;
          op1      : in  word;
          op2      : in  word;
          result   : out word);
end entity alu;

architecture behavioral of alu is

begin  -- architecture behavioral

    -- purpose: arithmetic and logic
    -- type   : combinational
    -- inputs : alu_func, op1, op2
    -- outputs: result
    alu_proc : process (alu_func, op1, op2) is
        variable so1, so2 : signed(31 downto 0);
        variable uo1, uo2 : unsigned(31 downto 0);
    begin  -- process alu_proc
        so1 := signed(op1);
        so2 := signed(op2);
        uo1 := unsigned(op1);
        uo2 := unsigned(op2);

        case (alu_func) is
            when ALU_ADD  => result <= std_logic_vector(so1 + so2);
            when ALU_ADDU => result <= std_logic_vector(uo1 + uo2);
            when ALU_SUB  => result <= std_logic_vector(so1 - so2);
            when ALU_SUBU => result <= std_logic_vector(uo1 - uo2);
            when ALU_SLT =>
                if so1 < so2 then
                    result <= "00000000000000000000000000000001";
                else
                    result <= (others => '0');
                end if;
            when ALU_SLTU =>
                if uo1 < uo2 then
                    result <= "00000000000000000000000000000001";
                else
                    result <= (others => '0');
                end if;

            when ALU_AND => result <= op1 and op2;
            when ALU_OR  => result <= op1 or op2;
            when ALU_XOR => result <= op1 xor op2;
            when ALU_SLL => result <= std_logic_vector(shift_left(uo1,
to_integer(uo2(4 downto 0))));
            when ALU_SRA => result <= std_logic_vector(shift_right(so1,
 to_integer(uo2(4 downto 0))));
            when ALU_SRL => result <= std_logic_vector(shift_right(uo1,
 to_integer(uo2(4 downto 0))));
            when others  => result <= op1;
        end case;
    end process alu_proc;

end architecture behavioral;
```

the common.vhd

```vhdl
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

package common is
    -- definition for a machine word
    subtype word is std_logic_vector(31 downto 0);
    subtype reg_addr_t is std_logic_vector(4 downto 0);

    subtype alu_func_t is std_logic_vector(3 downto 0);
    constant ALU_NONE : alu_func_t := "0000";
    constant ALU_ADD  : alu_func_t := "0001";
    constant ALU_ADDU : alu_func_t := "0010";
    constant ALU_SUB  : alu_func_t := "0011";
    constant ALU_SUBU : alu_func_t := "0100";
    constant ALU_SLT  : alu_func_t := "0101";
    constant ALU_SLTU : alu_func_t := "0110";
    constant ALU_AND  : alu_func_t := "0111";
    constant ALU_OR   : alu_func_t := "1000";
    constant ALU_XOR  : alu_func_t := "1001";
    constant ALU_SLL  : alu_func_t := "1010";
    constant ALU_SRA  : alu_func_t := "1011";
    constant ALU_SRL  : alu_func_t := "1100";

    subtype branch_type_t is std_logic_vector(2 downto 0);
    constant BEQ          : branch_type_t := "000";
    constant BNE          : branch_type_t := "001";
    constant BLT          : branch_type_t := "100";
    constant BGE          : branch_type_t := "101";
    constant BLTU        : branch_type_t := "110";
    constant BGEU        : branch_type_t := "111";

    subtype load_type_t is std_logic_vector(2 downto 0);
    constant LOAD_NONE : load_type_t := "000";
    constant LB        : load_type_t := "001";
    constant LH        : load_type_t := "010";
    constant LW        : load_type_t := "011";
    constant LBU       : load_type_t := "100";
    constant LHU       : load_type_t := "101";

    subtype store_type_t is std_logic_vector(1 downto 0);
    constant STORE_NONE : store_type_t := "00";
    constant SB         : store_type_t := "01";
    constant SH         : store_type_t := "10";
    constant SW         : store_type_t := "11";

    subtype system_type_t is std_logic_vector(2 downto 0);
    constant SYSTEM_ECALL  : system_type_t := "000";
    constant SYSTEM_EBREAK : system_type_t := "001";
    constant SYSTEM_CSRRW  : system_type_t := "010";
    constant SYSTEM_CSRRS  : system_type_t := "011";
    constant SYSTEM_CSRRC  : system_type_t := "100";
    constant SYSTEM_CSRRWI : system_type_t := "101";
    constant SYSTEM_CSRRSI : system_type_t := "110";
    constant SYSTEM_CSRRCI : system_type_t := "111";

    subtype opcode_t is std_logic_vector(6 downto 0);
    constant OP_ITYPE  : opcode_t := "0010011";
    constant OP_RTYPE : opcode_t := "0110011";
    constant OP_JTYPE : opcode_t := "1101111";
    constant OP_LOAD : opcode_t :="0000011";
    constant OP_STORE : opcode_t := "0100011";
    constant OP_BRANCH : opcode_t := "1100011";
    constant OP_JALR : opcode_t :="1100111";
    constant OP_LUI : opcode_t :="0110111";
    constant OP_AUIPC : opcode_t :="0010111";

    -- print a string with a newline
    procedure println (str : in    string);
    procedure print (slv   : in    std_logic_vector);
    procedure write(l      : inout line; slv : in std_logic_vector);
    function hstr(slv      :       std_logic_vector) return string;
```

```vhdl
    -- instruction formats
    type r_insn_t is (R_ADD, R_SLT, R_SLTU, R_AND, R_OR, R_XOR, R_SLL,
R_SRL, R_SUB, R_SRA);
    type i_insn_t is (I_JALR, I_LB, I_LH, I_LW, I_LBU, I_LHU, I_ADDI, I
_SLTI, I_SLTIU, I_XORI, I_ORI, I_ANDI, I_SLLI, I_SRLI, I_SRAI);
    type s_insn_t is (S_SB, S_SH, S_SW);
    type sb_insn_t is (SB_BEQ, SB_BNE, SB_BLT, SB_BGE, SB_BLTU, SB_BGEU
);
    type u_insn_t is (U_LUI, U_AUIPC);
    type uj_insn_t is (UJ_JAL);

    -- ADDI r0, r0, r0
    constant NOP : word := "00000000000000000000000000010011";

end package common;

package body common is

    function hstr(slv : std_logic_vector) return string is
        variable hexlen  : integer;
        variable longslv : std_logic_vector(67 downto 0) := (others =>
'0');
        variable hex     : string(1 to 16);
        variable fourbit : std_logic_vector(3 downto 0);
    begin
        hexlen := (slv'left+1)/4;
        if (slv'left+1) mod 4 /= 0 then
            hexlen := hexlen + 1;
        end if;
        longslv(slv'left downto 0) := slv;
        for i in (hexlen -1) downto 0 loop
            fourbit := longslv(((i*4)+3) downto (i*4));
            case fourbit is
                when "0000" => hex(hexlen -I) := '0';
                when "0001" => hex(hexlen -I) := '1';
                when "0010" => hex(hexlen -I) := '2';
                when "0011" => hex(hexlen -I) := '3';
                when "0100" => hex(hexlen -I) := '4';
                when "0101" => hex(hexlen -I) := '5';
                when "0110" => hex(hexlen -I) := '6';
                when "0111" => hex(hexlen -I) := '7';
                when "1000" => hex(hexlen -I) := '8';
                when "1001" => hex(hexlen -I) := '9';
                when "1010" => hex(hexlen -I) := 'A';
                when "1011" => hex(hexlen -I) := 'B';
                when "1100" => hex(hexlen -I) := 'C';
                when "1101" => hex(hexlen -I) := 'D';
                when "1110" => hex(hexlen -I) := 'E';
                when "1111" => hex(hexlen -I) := 'F';
                when "ZZZZ" => hex(hexlen -I) := 'z';
                when "UUUU" => hex(hexlen -I) := 'u';
                when "XXXX" => hex(hexlen -I) := 'x';
                when others => hex(hexlen -I) := '?';
            end case;
        end loop;
        return hex(1 to hexlen);
    end hstr;

    -- print a string with a newline
    procedure println (str : in string) is
        variable l : line;
    begin  -- procedure println
        write(l, str);
        writeline(output, l);
    end procedure println;

    procedure write(l : inout line; slv : in std_logic_vector) is
    begin
        for i in slv'range loop
            if slv(i) = '0' then
                write(l, string'("0"));
            elsif slv(i) = '1' then
                write(l, string'("1"));
            elsif slv(i) = 'X' then
                write(l, string'("X"));
            elsif slv(i) = 'U' then
                write(l, string'("U"));
            end if;
        end loop;  -- i
    end procedure write;

    procedure print (slv : in std_logic_vector) is
        variable l : line;
    begin  -- procedure print
        write(l, slv);
        writeline(output, l);
    end procedure print;
```

```
end package body common;
```

dmem.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.common.all;

entity dmem is
    port (reset : in  std_logic;
          clk   : in  std_logic;
          raddr : in  std_logic_vector(5 downto 0);
          dout  : out word;
          waddr : in  std_logic_vector(5 downto 0);
          din : in  word;
          we    : in  std_logic);
end entity dmem;

--
-- Note: Because this core is FPGA-
targeted, the idea is that these registers
--   will get implemented as dual-
port Distributed RAM.  Because there is no
--   such thing as triple-
port memory in an FPGA (that I know of), and we
-
-   need 3 ports to support 2 reads and 1 write per cycle, the easiest
way
-
-   to implement that is to have two identical banks of registers that
contain
--   the same data.  Each uses 2 ports and everybody's happy.
--
architecture rtl of dmem is
    type regbank_t is array (0 to 63) of word;

    signal regbank0 : regbank_t := (others => (others => '0'));

begin  -- architecture Behavioral

    -- purpose: create registers
    -- type   : sequential
    -- inputs : clk
    -- outputs:
```

```vhdl
    registers_proc : process (clk) is
    begin  -- process registers_proc
        if reset='1' then
            regbank0(0) <= x"0000ACE1";
        elsif rising_edge(clk) then
            if (we = '1') then
                regbank0(to_integer(unsigned(waddr))) <= din;
            end if;
        end if;
    end process registers_proc;

    -- asynchronous read
    dout <= regbank0(to_integer(unsigned(raddr)));

end architecture rtl;
```

imem.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.common.all;

entity imem is
    port(
        addr : in std_logic_vector(5 downto 0);
        dout : out word);
end imem;

architecture behavioral of imem is
    type rom_arr is array(0 to 46) of word;

    constant mem:rom_arr:=
        (








              x"00000737" ,--
8 0000 37070000            lui a4,%hi(lfsr.2565)      0000 0000 0000 00
00 0000 0111 0011 0111

```

```
              x"00075503" ,--
9 0004 03550700            lhu a0,%lo(lfsr.2565)(a4)  0000 0000 0000 01
11 0101 0101 0000 0011
              x"00000013", --   0008    13000000                nop
              x"00755793" ,--
10 000c 93577500       srli    a5,a0,7            0000 0000 0111 0101 0
101 0111 1001 0011
              x"00000013", --    0010   13000000                nop
              x"00F547B3" ,--
11 0014 B347F500       xor a5,a0,a5               0000 0000 1111 0101 0
100 0111 1011 0011
              x"00000013", --   0018    13000000                nop
              x"00979513" ,--12 001c 13959700       slli    a0,a5,9
              x"00000013", --    0020   13000000                nop
              x"00F54533" ,--13 0024 3345F500       xor a0,a0,a5
              x"00000013", --   0028 13000000                nop
              x"01051513", --14 002c 13150501       slli    a0,a0,16
              x"00000013", --   0030    13000000                nop
              x"01055513" ,--15 0034 13550501       srli    a0,a0,16
              x"00000013", --   0038    13000000                nop
              x"00D55793" ,--16 003c 9357D500       srli    a5,a0,13
              x"00000013", --   0040 13000000                nop
              x"00F54533" ,--17 0044 3345F500       xor a0,a0,a5
              x"00000013", --   0048    13000000                nop
              x"00A71023" ,--
18 004c 2310A700       sh  a0,%lo(lfsr.2565)(a4) 0000 0000 1010 0111 00
01 0000 0010 0011

              x"00008067", --
19 0050 67800000       ret                       0000 0000 0000 0000 10
00 0000 0110 0111
      --20                      .size   lfsr3, .-lfsr3
      --21                      .align  2
      --22                      .globl  main
      --23                      .type   main, @function
      --24                  main:
              x"FF010113" ,--25 0054 130101FF       addi    sp,sp,-
16  1111 1111 0000 0001 0000 0001 0001 0011 sp=-16
              x"00000013", --    0060   13000000                nop
              x"00112623" ,--
26 0064 23261100       sw  ra,12(sp)      0000 0000 0001 0001 0010 0110
 0010 0011 ra= -4
```

```
              x"00812423" ,--
27 006c 23248100       sw  s0,8(sp)
           s0=-8

              x"00912223" ,--
28 0074 23229100       sw  s1,4(sp)
            s1=-12

             x"00000437" ,--
29 007c 37040000        lui s0,%hi(outputs)0000 0000 0000 0000 0000 010
0 0011 0111
             x"00000013", --    0080   13000000                nop
              x"00040413" ,--
30 0084 13040400       addi    s0,s0,%lo(outputs)
              x"00000013", --    0088   13000000                nop
              x"01E40493" ,--31 008c 9304E401       addi    s1,s0,30
      --32                  .L3:
               x"00000013", --    0089   13000000                nop
              x"00000097" ,--
33 0094 97000000       call    lfsr3 0000 0000 1001 0111
              x"00000013", --    0098   13000000                nop
              x"000080E7" ,--
33 009c E7800000                      0000 0000 0000 0000 1000 0000 11
10 0111
              x"00000013", --   00a0    13000000                nop
             x"00A40023" ,--34 0094 2300A400        sb  a0,0(s0)
             x"00000013", --    0098   13000000                nop
              x"00140413" ,--35 009c 13041400       addi    s0,s0,1
              x"00000013", --    00a0   13000000                nop
              x"FC940EE3" ,--
36 00a4 E31894FE       bne s0,s1,.L3 11111100100101000000111011100011
                                    --11100
              x"00000513", --37 0060 13050000       li  a0,0

              x"00C12083" ,--38 0064 8320C100       lw  ra,12(sp)

              x"00812403" ,--39 0068 03248100       lw  s0,8(sp)

              x"00412483" ,--40 006c 83244100       lw  s1,4(sp)

              x"01010113" ,--41 0070 13010101       addi    sp,sp,16

              x"00008067");--42 0074 67800000       jr  ra -
-       13000000                nop
```

```
  --43                      .size   main, .-main
  --44                      .comm   outputs,30,4
  --45                      .section    .sdata,"aw"
  --46                      .align  1
 -- 47                      .type   lfsr.2565, @object
 -- 48                      .size   lfsr.2565, 2
  --49                  lfsr.2565:

         --50 0000 E1AC             .half   -21279
  --51                      .ident  "GCC: (GNU) 8.2.0"

begin
    dout<=mem(conv_integer(addr));
end behavioral;
```

pipeline.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.common.all;

entity pipeline is
    port (reset : in  std_logic;
          clk   : in  std_logic;
          pc_in  : in word;
          rd1_in : in word;
          rd2_in : in word;
          ir_in : in word;
          pc_out  : out word;
          rd1_out : out word;
          rd2_out : out word;
          ir_out : out word;
          we    : in  std_logic);
end entity pipeline;

--
-- Note: Because this core is FPGA-
targeted, the idea is that these registers
--   will get implemented as dual-
port Distributed RAM.  Because there is no
--   such thing as triple-
port memory in an FPGA (that I know of), and we
```

```vhdl
-
-   need 3 ports to support 2 reads and 1 write per cycle, the easiest
way
-
-   to implement that is to have two identical banks of registers that
contain
--   the same data.  Each uses 2 ports and everybody's happy.
--
architecture rtl of pipeline is
    type regbank_t is array (0 to 3) of word;

    signal regbank0 : regbank_t := (others => (others => '0'));

begin  -- architecture Behavioral

    -- purpose: create registers
    -- type   : sequential
    -- inputs : clk
    -- outputs:
    registers_proc : process (clk) is
    begin  -- process registers_proc
        if rising_edge(clk) then
            if (we = '1') then
                regbank0(0) <= pc_in;
                regbank0(1) <= rd1_in;
                regbank0(2) <= rd2_in;
                regbank0(3) <= ir_in;
            end if;
        end if;
    end process registers_proc;
    pc_out <= regbank0(0);
    rd1_out <= regbank0(1);
    rd2_out <= regbank0(2);
    ir_out <= regbank0(3);

    -- asynchronous read


end architecture rtl;
```

regfile.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.common.all;

entity regfile is
    port (reset : in  std_logic;
          clk   : in  std_logic;
          addra : in  std_logic_vector(4 downto 0);
          addrb : in  std_logic_vector(4 downto 0);
          rega  : out word;
          regb  : out word;
          addrw : in  std_logic_vector(4 downto 0);
          dataw : in  word;
          we    : in  std_logic);
end entity regfile;

--
-- Note: Because this core is FPGA-
targeted, the idea is that these registers
--   will get implemented as dual-
port Distributed RAM.  Because there is no
--   such thing as triple-
port memory in an FPGA (that I know of), and we
-
-   need 3 ports to support 2 reads and 1 write per cycle, the easiest
way
-
-   to implement that is to have two identical banks of registers that
contain
--   the same data.  Each uses 2 ports and everybody's happy.
--
architecture rtl of regfile is
    type regbank_t is array (0 to 31) of word;

    signal regbank0 : regbank_t := (others => (others => '0'));
    signal regbank1 : regbank_t := (others => (others => '0'));
begin  -- architecture Behavioral

    -- purpose: create registers
    -- type   : sequential
    -- inputs : clk
    -- outputs:
    registers_proc : process (clk) is
    begin  -- process registers_proc
        if rising_edge(clk) then
            if (we = '1') then
```

```vhdl
                regbank0(to_integer(unsigned(addrw))) <= dataw;
                regbank1(to_integer(unsigned(addrw))) <= dataw;
            end if;
        end if;
    end process registers_proc;

    -- asynchronous read
    rega <= regbank0(to_integer(unsigned(addra)));
    regb <= regbank1(to_integer(unsigned(addrb)));

end architecture rtl;
```