Simple Approach of Designing a Two-stage Pipelined RV32I Micro-architecture

Bosheng Peng,
J03 Building, Electrical Engineering
Darlington NSW 2008, Australia
bpen8455@uni.sydney.edu.au

***Abstract*--RISC-V is an open source ISA which was designed by The University of California, Berkeley in 2010. It is based on Reduced Instruction Set Computer design principles, and also to be designed with simplicity, flexibility, and extendibility. These mean that, RISC-V is designed with 4 Base Integer Instruction Sets, and 13 Standard Extension Sets. For implementations, the Base Integer Instruction Sets are essential, Standard Extension Sets are optional. The four Base Integer Instruction Sets are RV32I, RV32E, RV64I, and RV128I. The sizes of instructions can be told from the names, that RV32I has 32 bits in instructions, RV64I is for 64 bits systems, and RV128I is for 128 bits systems. The exception is RV32E, which is a size reduced (16 bits instructions) version of RV32I. In this simple approach, the RV32I is chosen to be implemented, and the pipeline has two stages which are IF/ID, and EX/MEM/WB.**

I. Introduction

In general, for a RV32I design, only the 32 general purpose registers are required. Due to the flexibility, the architecture can be implemented as either a von Neumann Architecture or a Harvard Architecture. Also, the endian system is not specified by the standards, so an endian system should also be chosen by the implanters (can a mixed endian system). For this simple approach, the Little-Endian Harvard Architecture with 32 bits memory address size, and 32 registers with 32 bits each are chosen to be implemented. For the conventional date sizes, a machine word is defined to be 32 bits, a machine half-word is defined to be 16 bits, and a machine byte is 8 bits. In this design, the SYSTEM, FENCE, or exception related functions are not mentioned or implemented.

II. RV32I ISA [1]

There are 47 different instructions in RV32I Base Instruction Set, and are categorised in six groups.

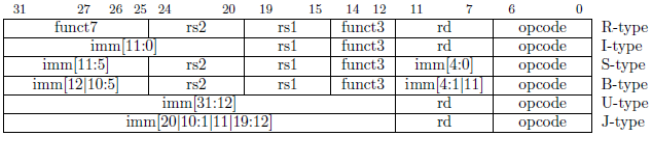


Fig.1. Five Groups of ISA. [1]

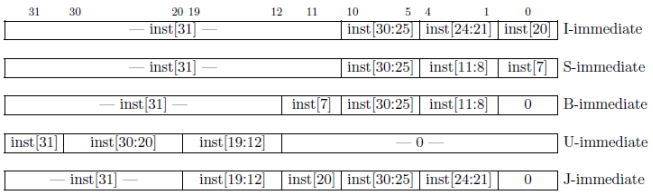


Fig.2. Five Groups of Immediate. [1]

In general, RV32I instructions may have the following format:

Operation Rd, Rs1, Rs2
Or
Operation Rd, Rs1, IMME

Where *Operation* is the function that performs on the operands. Rd is the destination register, Rs1 and Rs2 are the register sources. IMME is the immediate. These two formats are not be true all the time, since S and B types have no Rds, and U, J types have only Rds, and IMMEs, but they show that there are maximum of 3 operands in a single RV32I instruction.

From Fig.1., it shows that, the six groups of instructions are, R-type, I-type, S-type, B-type, U-type, and J-type. They are in common that, funct3 and funct7 are used to determine different functions in each type, and imm is used to do addressing or arithmetic operations.

R-type always have two source registers, and a destination register encoded. They also share the same opcode, hence, for distinguishing different R-type operations, funct3 and funct7 fields are used. For the R-type operations, they are all arithmetic functions that need to be done in the ALU.

There are three different opcodes for I-type instructions. They are load-related I-type, arithmetic I-type, and JALR. Load-related I-types are the operations that performs loading data from data memory to registers. They can load signed/unsigned byte, signed/unsigned half-word, and word to the registers. Arithmetic I-type operations are almost the same as R-type operations, the major difference is that R-type works on two source registers, I -type works on one source register and an immediate. JALR is a jump operation which does PC<=rs1+sign_extend(imm), and let Rd<=PC+4.

S-type has only store-related operations, which means it can only store a byte/half-word/word into data

memory which is also a similar process that is performed by Load-related I-types.

B-type has all the branching instructions defined. Which are BEQ (Equal), BNE (Not Equal), BLT (Signed/Unsigned, Less Than), and BGE (Signed/Unsigned Greater or Equal).

U-type has only two different operations. They are AUIPC (Add Upper Immediate PC) and LUI (Load Upper Immediate).

J-type only refers to JAL, which is the unconditional jump, PC<=PC+sign_extend(imm), Rd<=PC+4. There are also fence/system instructions, but they are not mentioned or implemented in this approach. For doing the sign extension to the immediate fields, fig.2. is the guideline to be followed.

III. Single Cycle Processor

Refer to Appendix 1, it shows the full data-path that is implemented in this approach. For the large components, we have Program Counter (PC), Arithmetic Logic Unit (ALU), Register File (RF), Instruction Memory (IMEM), Data Memory (DMEM), and Decoder (DEC). For the small components, they are mostly multiplexers. There are 4 standalone multiplexers in this design. Which are responsible for selecting different new PC values (pc_sel), operand 1/2 (Op1Sel, Op2Sel) for the ALU inputs, and Write Back Data (wb_sel). There are more Selection Signals, which are AluFun (for selecting the ALU Functions based on the instructions), rf_wen (Enable/Disable RF input), and dmem_wen (Enable/Disable writing new data into DMEM).

All possible choices for pc_sel:

Type	Meaning
PC+4	No Branch/JALR/JAL, only pointing to the next instruction
JALR	Jump and Link Register, PC gets a new value which is Register Source 1 value plus offset, and stores PC+4 in Rd.
Branch	PC gets a new value which is PC+offset
Jump/JAL	PC gets a new value of PC+offset, and stores PC+4 in Rd
Exception (Not implemented)	When exception happens, PC values should be changed relatively.

All possible choices for Op1Sel:

Rs1	Select the value of operand one of ALU. Which is the value store in register source one
U type IMME	Select sign extended (based on fig.2.) U type Immediate as ALU operand one input.

All possible choices for Op2Sel:

PC	Select PC as ALU operand two input.
I Type IMME	Select sign extended (based on fig.2.) I type Immediate as ALU operand two input.
B Type IMME	Select sign extended (based on fig.2.) B type Immediate as ALU operand two input.
Rs2	Select the value of operand two of ALU. Which is the value store in register source two.

All possible choices for web_sel:

PC+4	Store PC+4 in Rd.
ALU	Store the ALU ouput in Rd (for all the arithmetic related instructions).
dmem output	Store the data comes from DMEM in Rd. (for S-type instructions only)
CSRs (Not implemented)	Which is related to CSR registers. (System related type instructions)

There is a largest and the most important multiplexer in this design, which is the Decoder. It's responsible for generating all the control signals (includes the Branch Condition Generator). With this Decoder, the rest of work would be linking the wires between different components, and form the final RV32I processor.

IV. Performance

There is a sample program used in this design to test the performance of the Single Cycle Processor. In Appendix 2, the simulation shows that the program runs around 513500 ps, which is also, 5135 ns, since the simulation runs with 10 ns per cycle, the total cycles is 513.5. If the Reset cycle is not counted, this should be 512.5 (513 rounds up) cycles. Since this is a Single Cycle Processer, the CPI of this single stage design is one, which is also the ideal CPI.

V. 2 Stage Pipelined RV32I

Refer to Appendix 3, that is the full data-path of a 2 Stage Pipelined RV32I. The two stages are 1. IF/ID and 2. EX/MEM/WB. In order to achieve this, some modifications need to be done to the Single Cycle Processor. Firstly, we need 5 more 32-bit registers to

buffer the second stage data and signals. Which are named IR, op1_buffer, op1_buffer, rs1_buffer, and rs2_buffer. After buffering for Stage 2 data, the hazards come up, they are Structural, Data, and Control Hazards. For each one them, we need different strategy to deal with. Secondly, for resolving the Structural Hazards, a second Decoder is introduced, so that, there are now two Decoders. The Stage 1 Decoder is for decoding Op2Sel and Op1Sel, and the Stage 2 Decoder is for decoding web_sel, AluFun, pc_sel, dmem_wen, rf_wen. Thirdly, for dissolving the Data Hazards (only RAW hazards are need to be considered in this design), the forwarding/bypass paths are created. The first forwarding path is from ALU output to op1_buffer input, the second path is from ALU output to op2_buffer, the third path is from ALU output to rs2_buffer input, and two more paths are from DMEM output to both op1/op2_buffer inputs. As we doing this, there are 3 more multiplexers are introduced, hence a much larger multiplexer (Pipeline Controller) is introduced in order to generate and control all these three (fwd_op1_sel, fwd_op2_sel, fwd_rs2) new MUX signals. Finally, for dealing with the Control Hazards, a bubble MUX is created shown in the graph. Its control signal is called stall_sig, which is also controlled by Pipeline Controller. There is one more thing needs to be done for Control Hazards, which is minus PC by 4 when storing the return addresses while doing Branch/JALR/JAL. Otherwise, there will be one instruction being missed executed when the program counter goes back from a subroutine.

VI. Performance

The sample program in instruction memory is also being executed for measure the performance (appendix 4), the total cycle is around 604 (15% performance loss) which is larger than the Single Cycle Processor. This is reasonable, because that every time there is branching condition, a stall will be introduced, which wastes a clock cycle, but since there is no latency in the simulation, so it is hard to say if this is a decision or not.

VII. Conclusion and future improvements

From the simulations, it is hard to tell if a pipelined processor is better than a Signal Cycle Processor or not, this is because there is no latency in the simulations, but still, more improvements could be done to deal with the Control Hazards, such as, branch predictions, and branch delay slot. These methods can definitely improve the simulation speed, so as in the real word CPUs.

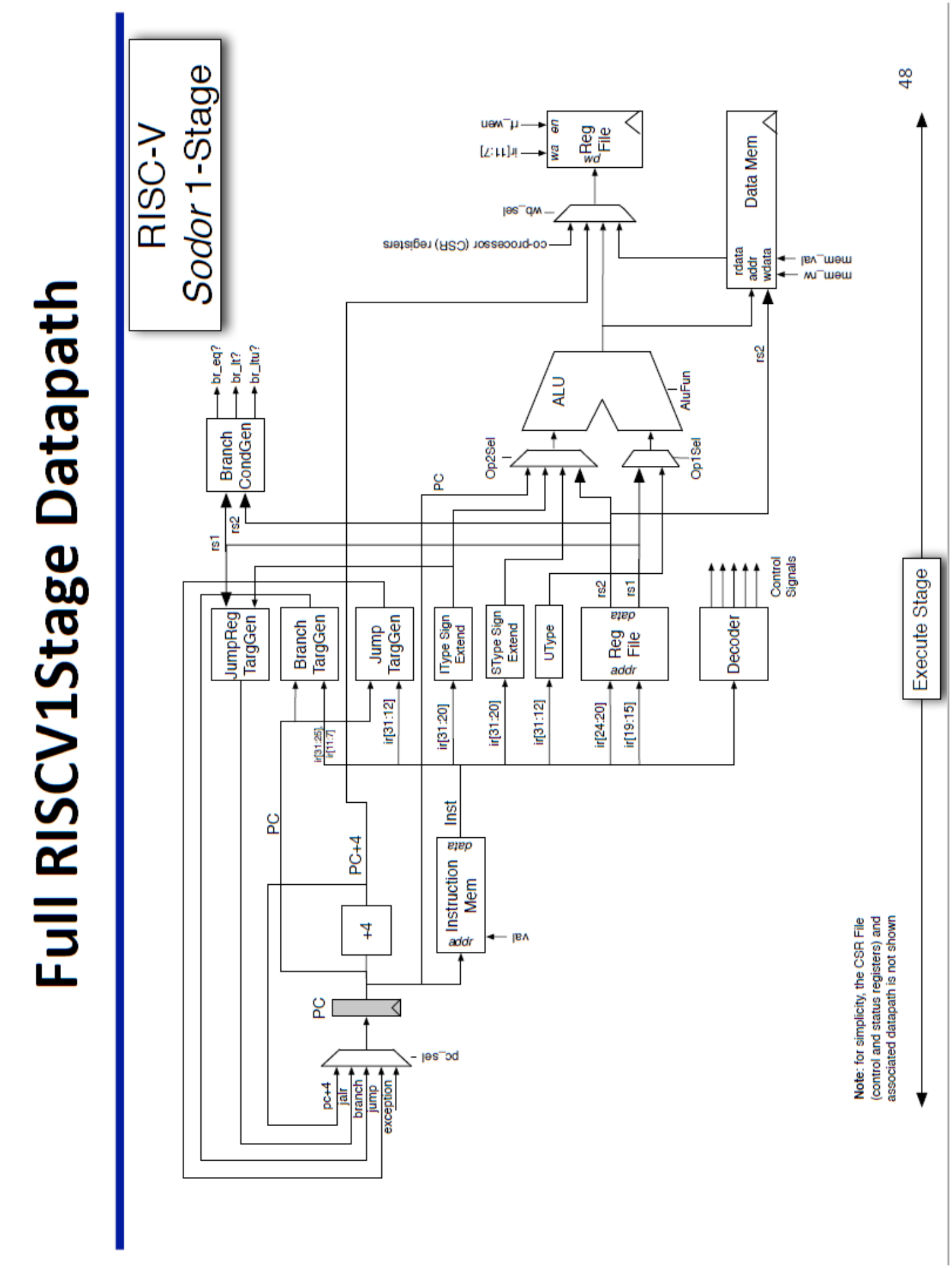
References:

[1]: riscv-spec-v2.2, https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

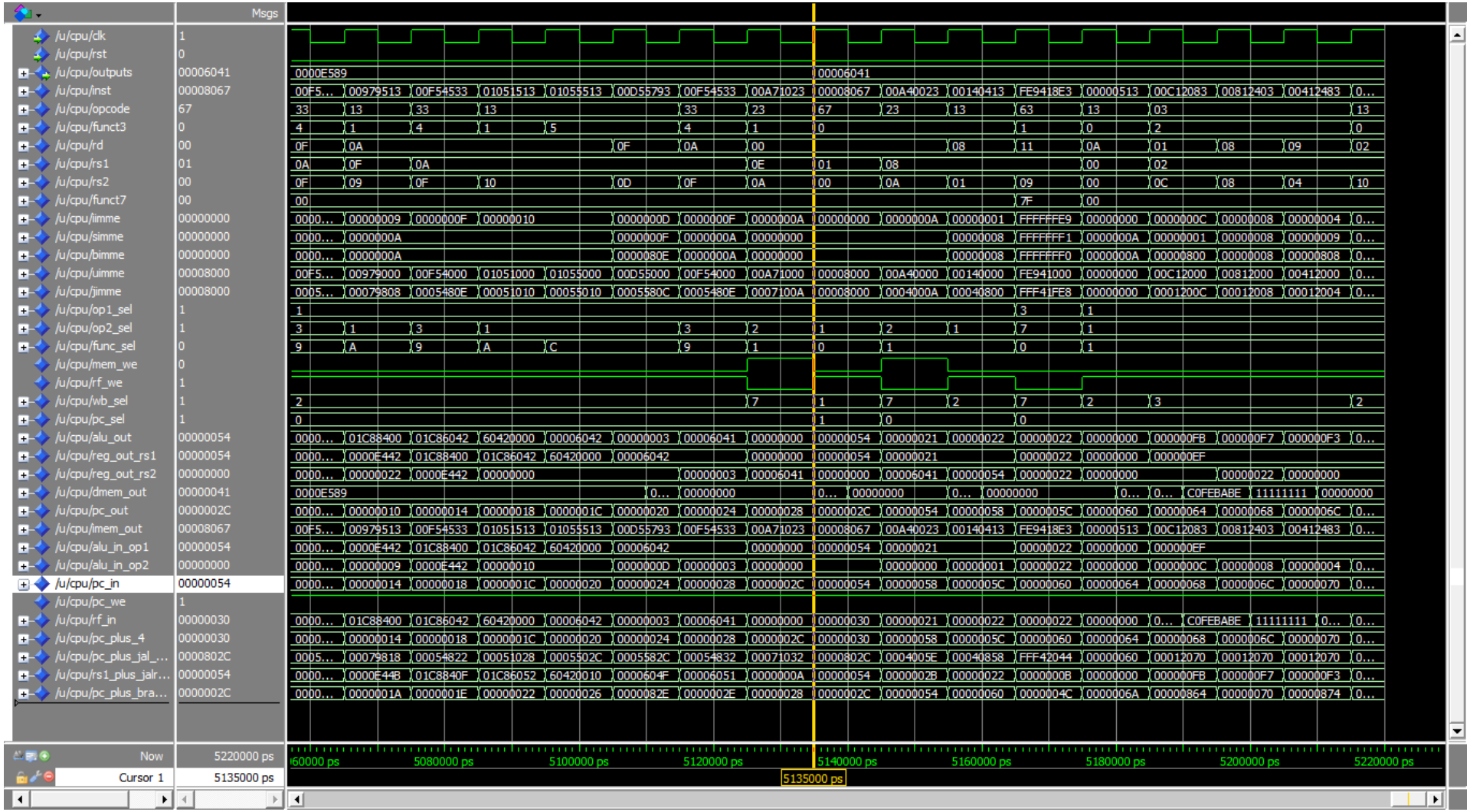
[2]: RISC-V Single Cycle Processor Implementation https://passlab.github.io/CSE564/notes/lecture08_RISCV_Impl.pdf

[3]: RISC-V Pipeline https://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf

Appendix 1 [2]:

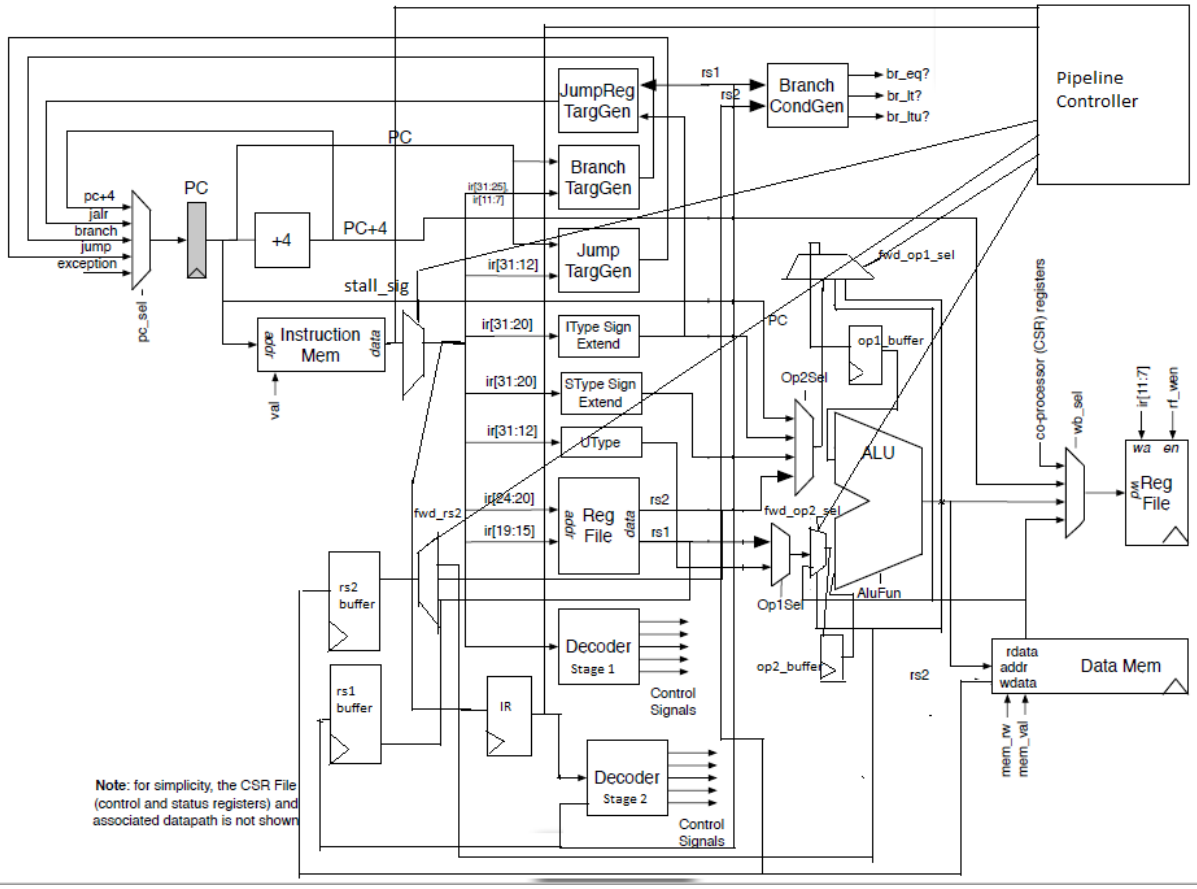

Full RISC-V1Stage Datapath
RISC-V
Sodor 1-Stage
Execute Stage
Note: for simplicity, the CSR File (control and status registers) and associated datapath is not shown
48
Data Mem
rdata
addr
wdata
mem_val
mem_rw
Reg File
wa
wd
en
rf_wen
ir[11:7]
wb_sel
co-processor (CSR) registers
ALU
AluFun
Op2Sel
Op1Sel
br_eq?
br_lt?
br_ltu?
Branch CondGen
rs1
rs2
PC
JumpReg TargGen
Branch TargGen
Jump TargGen
IType Sign Extend
SType Sign Extend
UType
Reg File
data
addr
Decoder
Control Signals
ir[31:25]
ir[11:7]
ir[31:12]
ir[31:20]
ir[24:20]
ir[19:15]
Inst
Instruction Mem
+4
PC+4
val
pc_sel
pc+4
jalr
branch
jump
exception

Appendix 2:

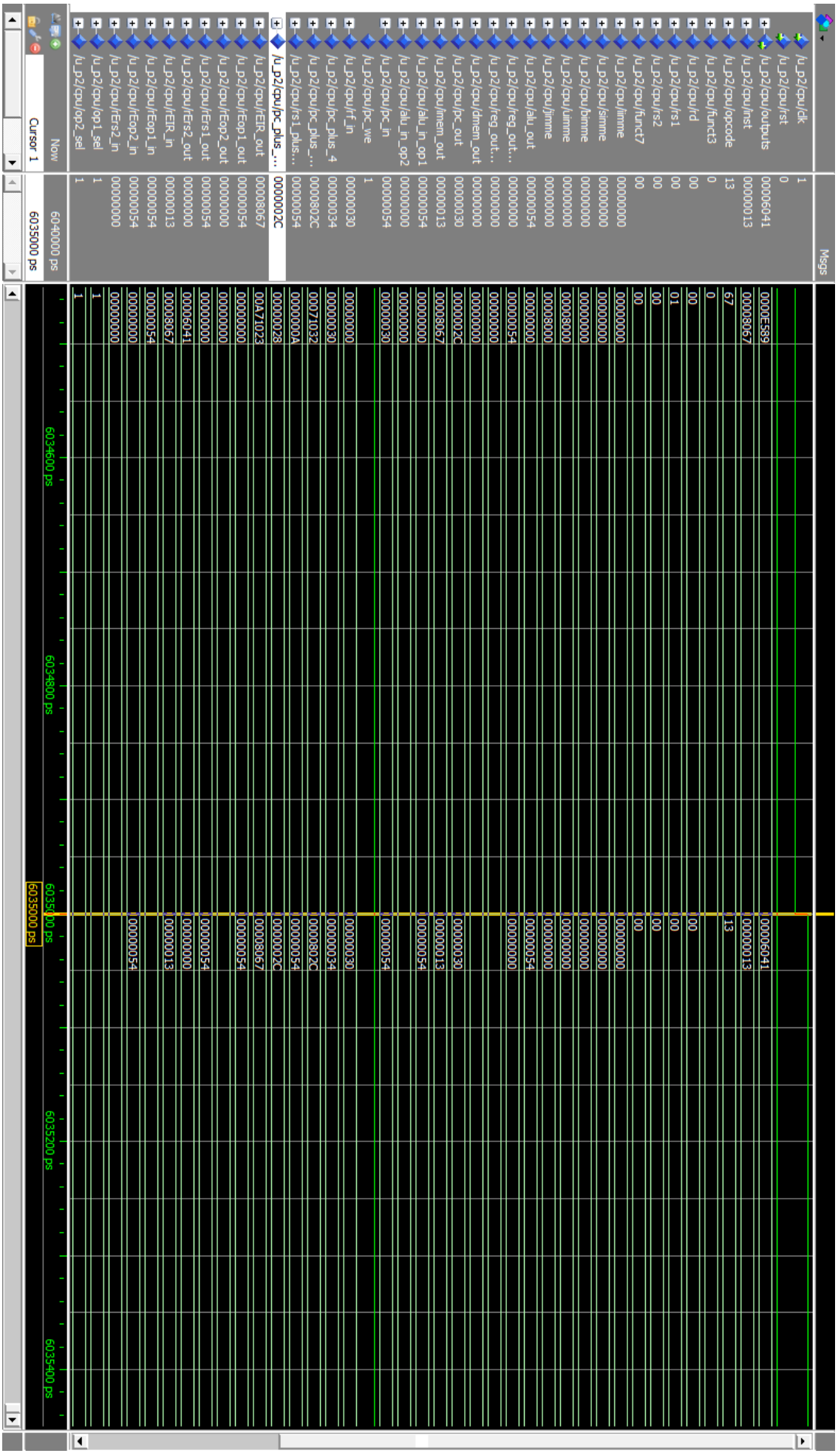

Msgs
/u/cpu/clk
/u/cpu/rst
/u/cpu/outputs
/u/cpu/inst
/u/cpu/opcode
/u/cpu/funct3
/u/cpu/rd
/u/cpu/rs1
/u/cpu/rs2
/u/cpu/funct7
/u/cpu/iimme
/u/cpu/simme
/u/cpu/bimme
/u/cpu/uimme
/u/cpu/jimme
/u/cpu/op1_sel
/u/cpu/op2_sel
/u/cpu/func_sel
/u/cpu/mem_we
/u/cpu/rf_we
/u/cpu/wb_sel
/u/cpu/pc_sel
/u/cpu/alu_out
/u/cpu/reg_out_rs1
/u/cpu/reg_out_rs2
/u/cpu/dmem_out
/u/cpu/pc_out
/u/cpu/imem_out
/u/cpu/alu_in_op1
/u/cpu/alu_in_op2
/u/cpu/pc_in
/u/cpu/pc_we
/u/cpu/rf_in
/u/cpu/pc_plus_4
/u/cpu/pc_plus_jal_...
/u/cpu/rs1_plus_jalr...
/u/cpu/pc_plus_bra...
Now
Cursor 1
5220000 ps
5135000 ps
5080000 ps
5100000 ps
5120000 ps
5140000 ps
5160000 ps
5180000 ps
5200000 ps
5135000 ps

Appendix 3:

2 Stage Pipeline



Appendix 4:


Msgs
/u_p2/cpu/clk
/u_p2/cpu/rst
/u_p2/cpu/outputs
/u_p2/cpu/inst
/u_p2/cpu/opcode
/u_p2/cpu/funct3
/u_p2/cpu/rd
/u_p2/cpu/rs1
/u_p2/cpu/rs2
/u_p2/cpu/funct7
/u_p2/cpu/iimme
/u_p2/cpu/simme
/u_p2/cpu/bimme
/u_p2/cpu/uimme
/u_p2/cpu/jimme
/u_p2/cpu/alu_out
/u_p2/cpu/reg_out...
/u_p2/cpu/reg_out...
/u_p2/cpu/dmem_out
/u_p2/cpu/pc_out
/u_p2/cpu/imem_out
/u_p2/cpu/alu_in_op1
/u_p2/cpu/alu_in_op2
/u_p2/cpu/pc_in
/u_p2/cpu/pc_we
/u_p2/cpu/f_in
/u_p2/cpu/pc_plus_4
/u_p2/cpu/pc_plus_...
/u_p2/cpu/rs1_plus...
/u_p2/cpu/pc_plus_...
/u_p2/cpu/eIR_out
/u_p2/cpu/fEop1_out
/u_p2/cpu/fEop2_out
/u_p2/cpu/fEr1_out
/u_p2/cpu/fEr2_out
/u_p2/cpu/eIR_in
/u_p2/cpu/fEop1_in
/u_p2/cpu/fEop2_in
/u_p2/cpu/fEr2_in
/u_p2/cpu/op1_sel
/u_p2/cpu/op2_sel
Now
Cursor 1
6040000 ps
6035000 ps
6034600 ps
6034800 ps
6035000 ps
6035200 ps
6035400 ps

Appendix Code:

Alu.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

--DEFINE alu START
entity alu is
    port (alu_func : in  FuncSel;--Selection signal of ALU functions
          op1      : in  word;--Oprand 1 input to ALU
          op2      : in  word;--Oprand 2 input to ALU
          result   : out word);--The output of the given ALU function
end entity alu;--DEFINE alu END

--IMPLEMENT alu START
architecture behavioral of alu is

begin--architecture alu

    alu_proc : process (alu_func, op1, op2) is--alu_proc, do the arithmetic calculations
        variable so1, so2 : signed(31 downto 0);--
local variables that holds signed oprands
        variable uo1, uo2 : unsigned(31 downto 0);--
local variables that holds unsigned oprands
    begin  -- process alu_proc
        so1 := signed(op1);--set so1 = signed oprand 1
        so2 := signed(op2);--set so2 = signed oprand 2
        uo1 := unsigned(op1);--set uo1 = unsigned oprand 1
        uo2 := unsigned(op2);--set uo2 = unsigned oprand 2

        case (alu_func) is--switch on alu_func
            when ALU_ADD  => result <= std_logic_vector(so1 + so2);--Signed ADD
            when ALU_ADDU => result <= std_logic_vector(uo1 + uo2);--Unsigned ADD
            when ALU_SUB  => result <= std_logic_vector(so1 - so2);--Signed SUBTRACT
            when ALU_SUBU => result <= std_logic_vector(uo1 - uo2);--Unsigned SUBTRACT
            when ALU_SLT =>--SET LESS THAN (SIGNED)
                if so1 < so2 then --when signed(op1) is less than signed(op2)
                    result <= "00000000000000000000000000000001";--alu ouputs 1
                else--otherwise
                    result <= (others => '0');--alu outputs 0
                end if;--END
            when ALU_SLTU =>--SET LESS THAN (UNSIGNED)
                if uo1 < uo2 then--when unsigned(op1) is less than unsigned(op2)
                    result <= "00000000000000000000000000000001";--alu ouputs 1
                else--otherwise
                    result <= (others => '0');--alu outputs 0
                end if;--END
```

```
            when ALU_AND => result <= op1 and op2;--
alu outputs the result of arithmetic AND between op1 and op2
            when ALU_OR  => result <= op1 or op2;--
alu outputs the result of arithmetic OR between op1 and op2
            when ALU_XOR => result <= op1 xor op2;--
alu outputs the result of arithmetic XOR between op1 and op2
            when ALU_SLL => result <= std_logic_vector(shift_left(uo1, to_integer(uo2(4 do
wnto 0))));--
alu outputs the result of SHIFT LEFT LOGICAL between unsigned(op1) and unsigned(op2)
            when ALU_SRA => result <= std_logic_vector(shift_right(so1, to_integer(uo2(4 d
ownto 0))));--
alu outputs the result of SHIFT LEFT ARTHMETIC between signed(op1) and unsigned(op2)
            when ALU_SRL => result <= std_logic_vector(shift_right(uo1, to_integer(uo2(4 d
ownto 0))));--
alu outputs the result of SHIFT RIGHT LOGICAL between unsigned(op1) and unsigned(op2)
            when others  => result <= op1;--
the rest of outputs are set to op1 by default (U type immediate, op1_sel)
        end case;--end of switch
    end process alu_proc;--END of the process

end architecture behavioral;--IMPLEMENT alu END
```

common.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd

package common is--START

    subtype word is std_logic_vector(31 downto 0);--Define a machine word
    subtype half is std_logic_vector(15 downto 0);--Define a machine half-word
    subtype byte is std_logic_vector(7 downto 0);--Define a machine byte

    subtype reg_addr_type is std_logic_vector(4 downto 0);--32 registers in total
    subtype mem_addr_type is std_logic_vector(31 downto 0);--32 bits memory address

    --instruction fields
    subtype opcode_type is std_logic_vector(6 downto 0);--opcode has 7 bits
    subtype funct3_type is std_logic_vector(2 downto 0);--funct3 has 3 bits
    subtype rd_type is std_logic_vector(4 downto 0);--rd has 5 bits
    subtype rs1_type is std_logic_vector(4 downto 0);--rs1 has 5 bits
    subtype rs2_type is std_logic_vector(4 downto 0);--rs2 has 5 bits
    subtype funct7_type is std_logic_vector(6 downto 0);--funct7 has 7 bits

    --Decoder control signals
    subtype Op1Sel is std_logic_vector(1 downto 0); --Select for OP1
    subtype Op2Sel is std_logic_vector(2 downto 0);--Select for OP2
    subtype FuncSel is std_logic_vector(3 downto 0);--Select for ALU Function
    subtype MemWr is std_logic;--Select for Mem Write Enable
```

```
    subtype RFWen is std_logic;--Select for RF Write Enable
    subtype WBSel is std_logic_vector(2 downto 0);--Select for RF Write Back Data
    subtype PCSel is std_logic_vector(2 downto 0);--Select for PC Write Back Data

    --Enable mem/reg write or not, must choose one
    constant enable : std_logic:='1';-->ENABLED
    constant disable: std_logic:='0';-->DISABLED

    --next PC value, must choose one
    constant pc_pc4: PCSel:="000";--PC=PC+4
    constant pc_jalr: PCSel:="001";--PC=RS1+OFFSET, RD=PC+4
    constant pc_branch: PCSel:="010";--PC=PC+OFFSET, if branch taken
    constant pc_jump: PCSel:="011";--PC=PC+OFFSET, RD=PC+4
    constant pc_e: PCSel:="100";--exception, not implemented

    --WBSels, can be undefined
    constant wb_csr: WBSel:="000";--CSR, not implemented
    constant wb_pc4: WBSel:="001";--write PC+4 back to RD
    constant wb_alu: WBSel:="010";--write ALU ouput back to RD
    constant wb_dmem: WBSel:="011";--write DMEM output back to RD
    constant wb_none: WBSel:="111";--N/A

    --Op1Sels, can be undefiend
    constant op1_uimme:Op1Sel:="00";--OP1=U type immediate
    constant op1_rs1:Op1Sel:="01";--OP1=rs1's value
    constant op1_none:Op1Sel:="11";--N/A

    --Op2Sels, can be undefined
    constant op2_pc:Op2Sel:="000";--OP2=PC
    constant op2_iimme:Op2Sel:="001";--OP2=I type immediate
    constant op2_simme:Op2Sel:="010";--OP2=S type immediate
    constant op2_rs2:Op2Sel:="011";--OP2=rs2's value
    constant op2_none:Op2Sel:="111";--N/A


    --ALU Functions, use within ALU
    constant ALU_NONE : FuncSel := "0000";--N/A
    constant ALU_ADD  : FuncSel := "0001";--Add
    constant ALU_ADDU : FuncSel := "0010";--Add unsigned
    constant ALU_SUB  : FuncSel := "0011";--subtract
    constant ALU_SUBU : FuncSel := "0100";--subtract unsigned
    constant ALU_SLT  : FuncSel := "0101";--set less than
    constant ALU_SLTU : FuncSel := "0110";--set less than unsigned
    constant ALU_AND  : FuncSel := "0111";--arithmetic and
    constant ALU_OR   : FuncSel := "1000";--arithmetic or
    constant ALU_XOR  : FuncSel := "1001";--arithmetic xor
    constant ALU_SLL  : FuncSel := "1010";--shift left logical
    constant ALU_SRA  : FuncSel := "1011";--shift right arithmetic
    constant ALU_SRL  : FuncSel := "1100";--shift right logical

    --Branch Types, Defined by Funct3
    constant FUNCT3_BEQ         : funct3_type := "000";--funct3 field of BEQ
    constant FUNCT3_BNE         : funct3_type := "001";--funct3 field of BNE
    constant FUNCT3_BLT         : funct3_type := "100";--funct3 field of BLT
```

```
    constant FUNCT3_BGE         : funct3_type := "101";--funct3 field of BGE
    constant FUNCT3_BLTU        : funct3_type := "110";--funct3 field of BLTU
    constant FUNCT3_BGEU        : funct3_type := "111";--funct3 field of BGEU

    --Load Types, Defined by Funct3
    constant FUNCT3_LB        : funct3_type := "000";--funct3 field of LB
    constant FUNCT3_LH        : funct3_type := "001";--funct3 field of LH
    constant FUNCT3_LW        : funct3_type := "010";--funct3 field of LW
    constant FUNCT3_LBU       : funct3_type := "100";--funct3 field of LBU
    constant FUNCT3_LHU       : funct3_type := "101";--funct3 field of LHU
    --Store Types, Defined by Funct3
    constant FUNCT3_SB         : funct3_type := "000";--funct3 field of SB
    constant FUNCT3_SH         : funct3_type := "001";--funct3 field of SH
    constant FUNCT3_SW         : funct3_type := "010";--funct3 field of SW

    --FUNCT3 of JALR
    constant FUNCT3_JALR:funct3_type:="000";--funct3 field of JALR

    --FUNCT3 of I Type Arithmetic functions
    constant FUNCT3_ADDI:funct3_type:="000";--funct3 field of ADDI
    constant FUNCT3_SLTI:funct3_type:="010";--funct3 field of SLTI
    constant FUNCT3_SLTIU:funct3_type:="011";--funct3 field of SLTIU
    constant FUNCT3_XORI:funct3_type:="100";--funct3 field of XORI
    constant FUNCT3_ORI:funct3_type:="110";--funct3 field of ORI
    constant FUNCT3_ANDI:funct3_type:="111";--funct3 field of ANDI
    constant FUNCT3_SLLI:funct3_type:="001";--funct3 field of SLLI
    constant FUNCT3_SRLI:funct3_type:="101";--funct3 field of SRLI
    constant FUNCT3_SRAI:funct3_type:="101";--funct3 field of SRAI

    --FUNCT3 of R Type Arithmetic functions
    constant FUNCT3_ADD:funct3_type:="000";--funct3 field of ADD
    constant FUNCT3_SUB:funct3_type:="000";--funct3 field of SUB
    constant FUNCT3_SLL:funct3_type:="001";--funct3 field of SLL
    constant FUNCT3_SLT:funct3_type:="010";--funct3 field of SLT
    constant FUNCT3_SLTU:funct3_type:="011";--funct3 field of SLTU
    constant FUNCT3_XOR:funct3_type:="100";--funct3 field of XOR
    constant FUNCT3_SRL:funct3_type:="101";--funct3 field of SRL
    constant FUNCT3_SRA:funct3_type:="101";--funct3 field of SRA
    constant FUNCT3_OR:funct3_type:="110";--funct3 field of OR
    constant FUNCT3_AND:funct3_type:="111";--funct3 field of AND

    ----FUNCT3 of Fence functions
    constant FUNCT3_FENCE:funct3_type:="000";--funct3 field of FENCE
    constant FUNCT3_FENCEI:funct3_type:="001";--funct3 field of FENCEI

    ----FUNCT3 of System functions
    constant FUNCT3_ECALL  : funct3_type := "000";--funct3 field of ECALL
    constant FUNCT3_EBREAK : funct3_type := "000";--funct3 field of EBREAK
    constant FUNCT3_CSRRW  : funct3_type := "001";--funct3 field of CSRRW
    constant FUNCT3_CSRRS  : funct3_type := "010";--funct3 field of CSRRS
    constant FUNCT3_CSRRC  : funct3_type := "011";--funct3 field of CSRRC
    constant FUNCT3_CSRRWI : funct3_type := "101";--funct3 field of CSRRWI
    constant FUNCT3_CSRRSI : funct3_type := "110";--funct3 field of CSRRSI
    constant FUNCT3_CSRRCI : funct3_type := "111";--funct3 field of CSRRCI
```

```
    --FUNCT7 of I Type OP Code
    constant FUNCT7_SLLI :funct7_type:="0000000";--funct7 field of SLLI
    constant FUNCT7_SRLI :funct7_type:="0000000";--funct7 field of SRLI
    constant FUNCT7_SRAI :funct7_type:="0100000";--funct7 field of SRAI
    --FUNCT7 of R Type OP Code
    constant FUNCT7_ADD :funct7_type:="0000000";--funct7 field of ADD
    constant FUNCT7_SUB :funct7_type:="0100000";--funct7 field of SUB
    constant FUNCT7_SLL :funct7_type:="0000000";--funct7 field of SLL
    constant FUNCT7_SLT :funct7_type:="0000000";--funct7 field of SLT
    constant FUNCT7_SLTU :funct7_type:="0000000";--funct7 field of SLTU
    constant FUNCT7_XOR :funct7_type:="0000000";--funct7 field of XOR
    constant FUNCT7_SRL :funct7_type:="0000000";--funct7 field of SRL
    constant FUNCT7_SRA :funct7_type:="0100000";--funct7 field of SRA
    constant FUNCT7_OR :funct7_type:="0000000";--funct7 field of OR
    constant FUNCT7_AND :funct7_type:="0000000";--funct7 field of AND



    --U Type OP Code
    constant OP_UTYPE_AUIPC: opcode_type:="0010111";--opcode of AUIPC
    constant OP_UTYPE_LUI: opcode_type:="0110111";--opcode of LUI
    --J Type OP Code
    constant OP_JTYPE_JAL: opcode_type:="1101111";--opcode of JAL
    --B Type OP Code
    constant OP_BTYPE_COMMON : opcode_type:="1100011";--opcode of B type instructions
    --I Type OP Code
    constant OP_ITYPE_LOAD_COMMON  : opcode_type := "0000011";--
opcode of load type instructions
    constant OP_ITYPE_ARITH_COMMON  : opcode_type := "0010011";--
opcode of I type arthmetic instructions
    constant OP_ITYPE_JALR  : opcode_type := "1100111";--opcode of JALR
    --S Type OP Code
    constant OP_STYPE_COMMON: opcode_type:="0100011";--opcode of S type instructions
    --R Type OP Code
    constant OP_RTYPE_COMMON:opcode_type:="0110011";--opcode of R type instructions
    --Other Types
    constant OP_FENCE_COMMON:opcode_type:="0001111";--opcode of FENCE type instructions
    constant OP_SYSTEM_COMMON:opcode_type:="1110011";--
opcode of SYSTEM related instructions

    --Functions to get immediate part of a given instruction
    function get_signed_Iimme(ins:word) return word;--DEFINE get_signed_Iimme
    function get_signed_Simme(ins:word) return word;--DEFINE get_signed_Simme
    function get_signed_Bimme(ins:word) return word;--DEFINE get_signed_Bimme
    function get_signed_Uimme(ins:word) return word;--DEFINE get_signed_Uimme
    function get_signed_Jimme(ins:word) return word;--DEFINE get_signed_Jimme

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

end package common;--END

package body common is--START
```

```
    function get_signed_Iimme(ins:word) return word is--
given an instruction ins, return I type immediate
        variable immediate:word:=(others=>'0');--local variable
    begin--START
        immediate(31 downto 11):=(others=>ins(31));--Sign Extend
        immediate(10 downto 5):=ins(30 downto 25);--Actual Value
        immediate(4 downto 1):=ins(24 downto 21);--Actual Value
        immediate(0):=ins(20);--Actual Value
        return immediate;--return the value
    end get_signed_Iimme;--END

    function get_signed_Simme(ins:word) return word is--
given an instruction ins, return S type immediate
        variable immediate:word:=(others=>'0');--local variable
    begin--START
        immediate(31 downto 11):=(others=>ins(31));--Sign Extend
        immediate(10 downto 5):=ins(30 downto 25);--Actual Value
        immediate(4 downto 1):=ins(11 downto 8);--Actual Value
        immediate(0):=ins(7);--Actual Value
        return immediate;--return the value
    end get_signed_Simme;--END

    function get_signed_Bimme(ins:word) return word is--
given an instruction ins, return B type immediate
        variable immediate:word:=(others=>'0');--local variable
    begin--START
        immediate(31 downto 12):=(others=>ins(31));--Sign Extend
        immediate(11):=ins(7);--Actual Value
        immediate(10 downto 5):=ins(30 downto 25);--Actual Value
        immediate(4 downto 1):=ins(11 downto 8);--Actual Value
        immediate(0):='0';--Actual Value
        return immediate;   --return the value
    end get_signed_Bimme;--END

    function get_signed_Uimme(ins:word) return word is--
given an instruction ins, return U type immediate
        variable immediate:word:=(others=>'0');--local variable
    begin--START
        immediate(31):=ins(31);--Sign Extend
        immediate(30 downto 20):=ins(30 downto 20);--Actual Value
        immediate(19 downto 12):=ins(19 downto 12);--Actual Value
        immediate(11 downto 0):=(others=>'0');--Actual Value
        return immediate;   --return the value
    end get_signed_Uimme;--END

    function get_signed_Jimme(ins:word) return word is--
given an instruction ins, return J type immediate
        variable immediate:word:=(others=>'0');--local variable
    begin--START
        immediate(31 downto 20):=(others=>ins(31));--Sign Extend
        immediate(19 downto 12):=ins(19 downto 12);--Actual Value
        immediate(11):=ins(20);--Actual Value
        immediate(10 downto 5):=ins(30 downto 25);--Actual Value
```

```
        immediate(4 downto 1):=ins(24 downto 21);--Actual Value
        immediate(0):='0';--Actual Value
        return immediate;--return the value
    end get_signed_Jimme;--END

end package body common;--END
```

decoder.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity decoder is--START
    port(inst:in word;--instruction in
        rs1_val:in word;--rs1's value in
        rs2_val:in word;--rs2's value in
        op1_sel : out Op1Sel; --Select for RS1
        op2_sel : out Op2Sel;--Select for RS2
        func_sel : out FuncSel;--Select for ALU Function
        mem_we : out MemWr;--Select for Mem Write Enable
        rf_we : out RFWen;--Select for RF Write Enable
        wb_sel : out WBSel;--Select for RF Write Back Data
        pc_sel : out PCSel);--Select for PC Write Back Data
end entity decoder;--END

architecture rtl of decoder is--START

signal opcode :opcode_type;--the opcode field of the given instruction
signal funct3:funct3_type;--the funct3 field of given instruction
signal funct7:funct7_type;--the funct7 field of given instruction
begin--START

opcode <=inst(6 downto 0);--extract the opcode field
funct3<=inst(14 downto 12);--extract the funct3 field
funct7<=inst(31 downto 25);--extract the funct7 field

    sel:process(inst,opcode,funct3,funct7,rs1_val,rs2_val) is--do the decoding
    begin--START

        op1_sel <= op1_rs1;--set the default value of op1_sel
        op2_sel<= op2_none;--set the default value of op2_sel
        func_sel<= ALU_NONE;--set the default value of func_sel
        mem_we<=disable;--set the default value of mem_we
        rf_we<=disable;--set the default value of rf_we
        wb_sel<=wb_none;--set the default value of wb_sel
        pc_sel<=pc_pc4;--set the default value of pc_sel

        if (opcode=OP_UTYPE_AUIPC) or --if -->
```

```
        (opcode=OP_UTYPE_LUI) then --U type

            op1_sel<=op1_uimme;-->op1 is set to be u type immediate

            if opcode=OP_UTYPE_AUIPC then-- but if AUIPC presents,
                op2_sel<=op2_pc;--op2 has the value of PC
                func_sel<=ALU_ADD;--alu function is ADD
            end if;    --END

            rf_we <= enable;--for all u type, rf is enabled
            wb_sel<=wb_alu;--for all u type, alu outupt is written back

        elsif opcode = OP_JTYPE_JAL then --J type
            op1_sel<=op1_none;--for all j type, op1 is N/A
            rf_we<=enable;--for all j type, rf is enabled
            wb_sel<=wb_pc4;--for all j type, PC+4 writes back
            pc_sel<=pc_jump;--for all j type, PC =PC+OFFSET

        elsif opcode=OP_BTYPE_COMMON then --B type
        op1_sel <=op1_none;--for all B type, op1 is N/A

            case funct3 is--switch on functs
                when FUNCT3_BEQ=>----------when branch equal
                if (rs1_val=rs2_val) then--if that is the real case
                    pc_sel<=pc_branch;-----pc =pc+branch_OFFSET
                end if;--------------------otherwise
                when FUNCT3_BNE=>-----------when branch not equal
                if (rs1_val/=rs2_val) then--if true
                    pc_sel<=pc_branch;------pc =pc+branch_OFFSET
                end if;----------------------otherwise
                when FUNCT3_BLT=>-----------branch less than
                if (to_integer(signed(rs1_val))<to_integer(signed(rs2_val))) then--if true
                    pc_sel<=pc_branch;-----------------------------------------------
pc =pc+branch_OFFSET
                end if;-----------------------------------------------------------
otherwise
                when FUNCT3_BGE=>------------------------------------------------
greater than or equal to
                if (to_integer(signed(rs1_val))>=to_integer(signed(rs2_val))) then--
if true
                    pc_sel<=pc_branch;-----------------------------------------------
pc =pc+branch_OFFSET
                end if;-----------------------------------------------------------
otherwise
                when FUNCT3_BLTU=>-----------------------------------------------
less than
                if (to_integer(unsigned(rs1_val))<to_integer(unsigned(rs2_val))) then--
if true
                    pc_sel<=pc_branch;-----------------------------------------------
pc =pc+branch_OFFSET
                end if;-----------------------------------------------------------
otherwise
                when FUNCT3_BGEU=>-----------------------------------------------
greater than or equal to (unsigned)
```

```
                if (to_integer(unsigned(rs1_val))>=to_integer(unsigned(rs2_val))) then--
if true
                    pc_sel<=pc_branch;----------------------------------------------------
pc =pc+branch_OFFSET
                end if;---------------------------------------------------------------------
otherwise
                when others=>null;----------------------------------------------------------
N/A
            end case;-------------------------------------------------------------------
END

        elsif (opcode =OP_ITYPE_LOAD_COMMON) or --I type
        (opcode=OP_ITYPE_ARITH_COMMON) or --I type
        (opcode = OP_ITYPE_JALR) then--I type

        op2_sel<= op2_iimme;--for all I type, op2=i type immediate
        rf_we<=enable;--for all I type, rf is enabled
        case opcode is--switch on opcode
            when OP_ITYPE_LOAD_COMMON=>--load type
            func_sel<=ALU_ADD;--alu func is add
            wb_sel<=wb_dmem;--write back is dmem
            when OP_ITYPE_ARITH_COMMON=>--i type arithmetic

            case funct3 is--switch on funct3
                when FUNCT3_ADDI=>func_sel<=ALU_ADD;--addi->alu func=add
                when FUNCT3_SLTI=>func_sel<=ALU_SLT;--SLTI->alu func=slt
                when FUNCT3_SLTIU=>func_sel<=ALU_SLTU;--SLTIU->alu func=sltu
                when FUNCT3_XORI=>func_sel<=ALU_XOR;--XORI->alu func=xor
                when FUNCT3_ORI=>func_sel<=ALU_OR;--ORI->alu func=or
                when FUNCT3_ANDI=>func_sel<=ALU_AND;--ANDI->alu func=and
                when FUNCT3_SLLI=>func_sel<=ALU_SLL;--SLLI->alu func=sll
                when FUNCT3_SRLI=>--srli and srai have the same opcode

                if (funct7=FUNCT7_SRLI) then--if srli determined by funct7
                    func_sel<=ALU_SRL;--alu func = SRL
                else--otherwise
                    func_sel<=ALU_SRA;--alu func = sra
                end if;--end

                when others=>null;--others=>N/A

            end case;--end
            wb_sel<=wb_alu;--for all I type, write back = alu output
            when OP_ITYPE_JALR=>--jalr
            func_sel<=ALU_NONE;--no need for alu func
            pc_sel<=pc_jalr;--pc=rs1+I immediate
            wb_sel<=wb_pc4;--write back is pc+4
            when others=>null;--others=>N/A
        end case;--end


        elsif opcode = OP_STYPE_COMMON then --S type

        op2_sel<= op2_simme;--for all s type, op2 is s type immediate
```

```
        func_sel<= ALU_ADD;--for all s type, alu func = add
        mem_we<=enable;--for all s type, need to dmem write back

        elsif opcode = OP_RTYPE_COMMON then --R type

        op2_sel<= op2_rs2;--for all r type, op2 is rs2's value
        rf_we<=enable;--for all r type, rf enabled
        wb_sel<=wb_alu;--for all r type, write back is alu output

        case funct3 is--switch on funct3
            when FUNCT3_ADD=>--add

                if funct7=FUNCT7_ADD then--add and sub have the same funct3
                    func_sel<=ALU_ADD;--add
                else--or
                    func_sel<=ALU_SUB;--sub
                end if;--end

            when FUNCT3_SLL=>func_sel<=ALU_SLL;--alu func =sll
            when FUNCT3_SLT=>func_sel<=ALU_SLT;--alu func =slt
            when FUNCT3_SLTU=>func_sel<=ALU_SLTU;--alu func =sltu
            when FUNCT3_XOR=>func_sel<=ALU_XOR;--alu func =xor
            when FUNCT3_SRL=>--srl and sra have the same funct3
                if funct7=FUNCT7_SRL then--when srl funct7
                    func_sel<=ALU_SRL;--alu func=srl
                else--otherwise
                    func_sel<=ALU_SRA;--alu func = sra
                end if;--end
            when FUNCT3_OR=>func_sel<=ALU_OR;--alu func = or
            when FUNCT3_AND=>func_sel<=ALU_AND;--alu func = and
            when others=>null;-- others=>N/A
        end case; --end

        else --Others
        --SYSTEM FUNCTIONS ARE NOT IMPLEMENTED
        end if;--END
    end process sel;--END

end architecture rtl;--END
```

dmem.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity dmem is--START
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          raddr : in  mem_addr_type;--32 bits address
```

```
          dout  : out word;--OUTPUT a WORD
          dsize:in funct3_type;--Data size, if we = '0' then read size, else write size
          waddr : in  mem_addr_type;--32 bits address
          din : in  word;--INPUT a WORD
          we    : in  std_logic;--write enable
          debug_output:out word--debug output, regbank0(0)
          );
end entity dmem;--END

--
-- 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--START
    type regbank_t is array (0 to 255) of byte;--Basic cell is byte, 256 bytes in total

    signal regbank0 : regbank_t := (--init mem array
        x"E1",x"AC",x"00",x"00",--0000 E1AC
        others=>x"00");--zero memory
begin  -- architecture Behavioral

    registers_proc : process (clk,reset,dsize,we) is--mem proc
    variable tmp_byte:byte:=x"00";--local variable
    begin  -- process registers_proc
        if reset='1' then--reset presents
            dout<=x"00000000";--output all zeros
        elsif rising_edge(clk) then--when clock edge
            if (we = '1') and (unsigned(waddr)<=255) then--
if the read address is not too large, and write enable is one,
                case dsize is--for different sizes
                    when FUNCT3_SB=>regbank0(to_integer(unsigned(waddr))) <= din(7 downto
0);--byte, writes one byte only
                    when FUNCT3_SH=>--Little Endian, half word, 2 bytes written
                    regbank0(to_integer(unsigned(waddr))) <= din(7 downto 0);--frist byte
                    regbank0(to_integer(unsigned(waddr))+1) <= din(15 downto 8);--
second byte
                    when FUNCT3_SW=>--Little Endian, 4 bytes word
                    regbank0(to_integer(unsigned(waddr))) <= din(7 downto 0);--first
                    regbank0(to_integer(unsigned(waddr))+1) <= din(15 downto 8);--second
                    regbank0(to_integer(unsigned(waddr))+2) <= din(23 downto 16);--third
                    regbank0(to_integer(unsigned(waddr))+3) <= din(31 downto 24);--fourth
                    when others=>null;--others=>N/A
                end case;--end
            end if;--end
        elsif (we='0') and (unsigned(raddr)<=255) and ((unsigned(raddr)+1)<=255)  then--
when reading
            case dsize is--different sizes
                when FUNCT3_LB=>--byte size,
                tmp_byte := regbank0(to_integer(unsigned(raddr)));--
temp variable = one byte data
```

```
                dout(7 downto 0)<=tmp_byte;--output it
                dout(31 downto 8)<=(others=>tmp_byte(7));--Sign Extended Output
                when FUNCT3_LH=>--Little Endian, half word
                dout(7 downto 0) <= regbank0(to_integer(unsigned(raddr)));--first byte
                tmp_byte := regbank0(to_integer(unsigned(raddr))+1);--second byte
                dout(15 downto 8)<=tmp_byte;--output it
                dout(31 downto 16)<=(others=>tmp_byte(7));--Sign Extended Output
                when FUNCT3_LW=>--Little Endian, 4 bytes word
                dout(7 downto 0) <= regbank0(to_integer(unsigned(raddr)));--first
                dout(15 downto 8)<= regbank0(to_integer(unsigned(raddr))+1);--second
                dout(23 downto 16)<=regbank0(to_integer(unsigned(raddr))+2);--third
                dout(31 downto 24)<=regbank0(to_integer(unsigned(raddr))+3);--fourth
                when FUNCT3_LBU=>--unsigned byte
                dout(7 downto 0)<= regbank0(to_integer(unsigned(raddr)));--actual data
                dout(31 downto 8)<=(others=>'0');--zero extended
                when FUNCT3_LHU=>--half word
                dout(7 downto 0)<= regbank0(to_integer(unsigned(raddr)));--frist byte
                dout(15 downto 8)<=regbank0(to_integer(unsigned(raddr))+1);--second byte
                dout(31 downto 16)<=(others=>'0');--zero extended
                when others=>dout<=x"C0FEBABE";--others=>magic number
            end case;--end
        end if;--end
    end process registers_proc;--end

    -- asynchronous read
    --dout <= regbank0(to_integer(unsigned(raddr)));
    debug_output(7 downto 0)<= regbank0(0);--debug output first byte
    debug_output(15 downto 8)<=regbank0(1);--debug output second byte
    debug_output(31 downto 16)<=(others=>'0');--zero extended
end architecture rtl;--end
```

imem.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity imem is--START
    port(addr : in mem_addr_type;--input address
        dout : out word);--output instruction
end imem;--END

architecture behavioral of imem is--START
type rom_arr is array(0 to 255) of byte;--maximum 64 insts, 256 Bytes in total

signal mem:rom_arr:=--Little Endian
    (   --
        --                  lfsr3:
```

```
        x"37",x"07",x"00",x"00",--0000 37070000         lui a4,%hi(lfsr.2565)
        x"03",x"55",x"07",x"00",--0004 03550700         lhu a0,%lo(lfsr.2565)(a4)
        x"93",x"57",x"75",x"00",--0008 93577500         srli    a5,a0,7
        x"B3",x"47",x"F5",x"00",--000c B347F500         xor a5,a0,a5
        x"13",x"95",x"97",x"00",--0010 13959700         slli    a0,a5,9
        x"33",x"45",x"F5",x"00",--0014 3345F500         xor a0,a0,a5
        --x"13",x"05",x"10",x"00",--Li a0,1
        x"13",x"15",x"05",x"01",--0018 13150501         slli    a0,a0,16
        x"13",x"55",x"05",x"01",--001c 13550501         srli    a0,a0,16
        x"93",x"57",x"D5",x"00",--0020 9357D500         srli    a5,a0,13
        x"33",x"45",x"F5",x"00",--0024 3345F500         xor a0,a0,a5
        x"23",x"10",x"A7",x"00",--0028 2310A700         sh  a0,%lo(lfsr.2565)(a4)
        x"67",x"80",x"00",x"00",--002c 67800000         ret
        --                  main:
        x"13",x"01",x"01",x"FF",-- 0030 130101FF        addi    sp,sp,-16
        x"23",x"26",x"11",x"00",-- 0034 23261100        sw  ra,12(sp)
        x"23",x"24",x"81",x"00",-- 0038 23248100        sw  s0,8(sp)
        x"23",x"22",x"91",x"00",-- 003c 23229100        sw  s1,4(sp)
        --x"37",x"04",x"00",x"00",-- 0040 37040000      lui s0,0
        --x"13",x"04",x"04",x"00",-- 0044 13040400      addi    s0,s0,0
        x"13",x"00",x"00",x"00",-- xxxx xxxxxxxx        NOP; for balancing the imem
        x"13",x"04",x"40",x"00",-
- xxxx xxxxxxxx        addi    s0,zero,4; Outputs address starts at 0x4
        x"93",x"04",x"E4",x"01",-- 0048 9304E401        addi    s1,s0,30
        --                  .L3:
        --x"97",x"00",x"00",x"00",-- 004c 97000000      call    lfsr3;auipc x1,0
        x"13",x"00",x"00",x"00",-- xxxx xxxxxxxx        NOP; for balancing the imem
        x"E7",x"00",x"00",x"00",--      E7000000        jalr ra, zero,0; Call lfsr3
        x"23",x"00",x"A4",x"00",-- 0054 2300A400        sb  a0,0(s0)
        x"13",x"04",x"14",x"00",-- 0058 13041400        addi    s0,s0,1
        x"E3",x"18",x"94",x"FE",-- 005c E31894FE        bne s0,s1,.L3
        x"13",x"05",x"00",x"00",-- 0060 13050000        li  a0,0
        x"83",x"20",x"C1",x"00",-- 0064 8320C100        lw  ra,12(sp)
        x"03",x"24",x"81",x"00",-- 0068 03248100        lw  s0,8(sp)
        x"83",x"24",x"41",x"00",-- 006c 83244100        lw  s1,4(sp)
        x"13",x"01",x"01",x"01",-- 0070 13010101        addi    sp,sp,16
        x"67",x"80",x"00",x"00",-- 0074 67800000        jr  ra
        others=>x"00"--zero memory
        );--END

begin--START
--little endian
    dout(7 downto 0)<=mem(conv_integer(addr));--output first byte of instruction
    dout(15 downto 8)<=mem(conv_integer(addr)+1);--output second byte of instruction
    dout(23 downto 16)<=mem(conv_integer(addr)+2);--output third byte of instruction
    dout(31 downto 24)<=mem(conv_integer(addr)+3);--output fourth byte of instruction
end behavioral;--end
```

pc.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
```

```
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity pc is--START
    port(clk: in std_logic;--clock
        rst: in std_logic;--reset
        ra: out mem_addr_type;--read address
        wa:in mem_addr_type;--write address
        we:in std_logic);--write enable
end pc;--END

architecture rtl of pc is --START
    signal pc_count: mem_addr_type;--stored counter
begin--START
    pc_register:process(clk,rst,we) is--the register
    begin--START
        if rst='1' then--if reset
            pc_count<=x"00000030";--output zeros
        elsif rising_edge(clk) then--clock edge
            if we='1' then--of write enabled
                pc_count<=wa;--write new data in
            else--otherwise
                pc_count<=pc_count;--stay the same
            end if;--end
        end if;--end
    end process pc_register;--end
    ra<=pc_count;--output the stored counter
end architecture rtl;--END
```

pplcontro.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity pplcontrol is--start
port(inst_s1:in word;--stage one instruction
    inst_s2:in word;--stage two instruction
    stall_disen:out std_logic;--output stall
    fwd_op1_sel:out std_logic_vector(1 downto 0);--forwarding for op1
    fwd_op2_sel:out std_logic_vector(1 downto 0);--forwarding for op2
    fwd_rs2_en:out std_logic);--forwarding for rs2's value
end entity pplcontrol;--end

architecture rtl of pplcontrol is--start

signal opcode1:opcode_type;--stage 1 relevant inst field
signal rs11:rs1_type;--stage 1 relevant inst field
```

```
signal rs21:rs2_type;--stage 1 relevant inst field

signal opcode2:opcode_type;--stage 2 relevant inst field
signal rd2:rd_type;--stage 2 relevant inst field


begin--start
opcode1<=inst_s1(6 downto 0);--stage 1 opcode

rs11<=inst_s1(19 downto 15);--stage 1 rs1
rs21<=inst_s1(24 downto 20);--stage 1 rs2

opcode2<=inst_s2(6 downto 0);--stage 2 opcode
rd2<=inst_s2(11 downto 7);--stage 2 rd


stall_gen:process(opcode2,opcode1)--mux for stall signal
begin--start
    if (opcode2=OP_JTYPE_JAL) or (opcode2=OP_ITYPE_JALR) or (opcode2=OP_BTYPE_COMMON) then
--if branch/jmp/jalr
        stall_disen<='0';--Stall when redirecting
    else--otherwise
        stall_disen<='1';--NO00000000000000000000
    end if;--end
end process stall_gen;--end

fwd_sel_gen:process(rd2,rs11,rs21,opcode2,opcode1)--mux for forwarding signals
begin--start
    if (rd2=rs11) and (opcode2=OP_ITYPE_LOAD_COMMON) then--Load RAW hazard on RS1
        fwd_op1_sel<="10";--forwarding dmem output
    elsif (rd2=rs11) and (opcode2/=OP_ITYPE_LOAD_COMMON) and (opcode1/=OP_STYPE_COMMON) th
en--alu arithmetic RAW hazard
        fwd_op1_sel<="01";--forwarding it
    else--otherwise
        fwd_op1_sel<="00";--do not
    end if;--end

    if (rd2=rs21) and (opcode2=OP_ITYPE_LOAD_COMMON) then--Load RAW hazard on RS2
        fwd_op2_sel<="10";--forwarding dmem output
    elsif (rd2=rs21) and (opcode2/=OP_ITYPE_LOAD_COMMON) and (opcode1/=OP_STYPE_COMMON) th
en--alu arithmetic RAW hazard
        fwd_op2_sel<="01";--forwarding it
    else--otherwise
        fwd_op2_sel<="00";--do not
    end if;--end

end process fwd_sel_gen;--end

fwd_rs2_gen:process(rd2,rs11,rs21,opcode2,opcode1)--STORE forwarding mux
begin--start

if (opcode1 = OP_STYPE_COMMON) and (rd2=rs21) then--
when stage 1 opcode is S type and stage 2 rd is equal to stage 1 rs2
    fwd_rs2_en<='1';--do it
```

```
else--otherwise
    fwd_rs2_en<='0';--don't do it
end if;--end


end process fwd_rs2_gen;--end

end architecture rtl;--end
```

reg32.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity reg32 is--start
    port(clk: in std_logic;--clock
        rst: in std_logic;--reset
        din: in word;--data in
        we : in std_logic;--write enable
        dout: out word);--data out
end entity reg32;--end

architecture rtl of reg32 is--start
signal data:word:=x"00000000";--local variable
begin--start
    reg_proc:process(clk,rst,we) is--32 bits general register
    begin--start
        if rst='1' then--if reset
            data<=x"00000000";--output zeros
        elsif rising_edge(clk) then--clock edge
            if we='1' then--if write enabled
                data<=din;          --write in new data
            else--otherwise
                data<=data;--keep data
            end if;--end
        end if;--end
    end process reg_proc;--end
    dout<=data;--output data
end architecture rtl;--end
```

regfile.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd
```

```
entity regfile is--start
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          addra : in  reg_addr_type;--rs1
          addrb : in  reg_addr_type;--rs2
          rega  : out word;--rs1's value
          regb  : out word;--rs2's value
          addrw : in  reg_addr_type;--write address
          dataw : in  word;--write data
          we    : in  std_logic);--write enable
end entity regfile;--end

--
-- 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--start
    type regbank_t is array (0 to 31) of word;--32 registers

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

    -- purpose: create registers
    -- type   : sequential
    -- inputs : clk
    -- outputs:
    registers_proc : process (clk) is--register array
    begin  -- process registers_proc
        if reset='1' then--if reset
            regbank0(1) <= x"C0FEBABE";--test data
            regbank1(1) <= x"C0FEBABE";--test data
            regbank0(8) <= x"11111111";--test data
            regbank1(8) <= x"11111111";--test data
            regbank0(2) <= x"000000FF";--test data
            regbank1(2) <= x"000000FF";--test data
        elsif rising_edge(clk) then--when clock edge
            if (we = '1') and (addrw/="00000") then--if write enabled, and write addr !=0
                regbank0(to_integer(unsigned(addrw))) <= dataw;--write in new data
                regbank1(to_integer(unsigned(addrw))) <= dataw;--write in new data
            end if;--end
        end if;--end
    end process registers_proc;--end

    -- asynchronous read
    rega <= regbank0(to_integer(unsigned(addra)));--output for rs1
    regb <= regbank1(to_integer(unsigned(addrb)));--output for rs2

end architecture rtl;--end
```

rv32i.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity rv32i is--RV32I without pipeline entity
port(clk:in std_logic;--clock
    rst:in std_logic;--reset
    outputs: out word);--debug output
end entity rv32i;--END RV32I

architecture rtl of rv32i is--implement rv32i

--DEFINE alu START
component alu is
    port (alu_func : in  FuncSel;--Selection signal of ALU functions
          op1      : in  word;--Oprand 1 input to ALU
          op2      : in  word;--Oprand 2 input to ALU
          result   : out word);--The output of the given ALU function
end component alu;--DEFINE alu END

component decoder is--decoder
    port(inst:in word;--instruction in
        rs1_val:in word;--rs1's value in
        rs2_val:in word;--rs2's value in
        op1_sel : out Op1Sel; --Select for RS1
        op2_sel : out Op2Sel;--Select for RS2
        func_sel : out FuncSel;--Select for ALU Function
        mem_we : out MemWr;--Select for Mem Write Enable
        rf_we : out RFWen;--Select for RF Write Enable
        wb_sel : out WBSel;--Select for RF Write Back Data
        pc_sel : out PCSel);--Select for PC Write Back Data
end component decoder;--END

component dmem is--data memory
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          raddr : in  mem_addr_type;--32 bits address
          dout  : out word;--OUTPUT a WORD
          dsize:in funct3_type;--Data size, if we = '0' then read size, else write size
          waddr : in  mem_addr_type;--32 bits address
          din : in  word;--INPUT a WORD
          we    : in  std_logic;--write enable
          debug_output:out word--debug output, regbank0(0)
          );
end component dmem;--END

component imem is--inst memory
```

```
    port(addr : in mem_addr_type;--input address
        dout : out word);--output instruction
end component imem;--END

component pc is--program counter
    port(clk: in std_logic;--clock
        rst: in std_logic;--reset
        ra: out mem_addr_type;--read address
        wa:in mem_addr_type;--write address
        we:in std_logic);--write enable
end component pc;--END

component regfile is--register file
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          addra : in  reg_addr_type;--rs1
          addrb : in  reg_addr_type;--rs2
          rega  : out word;--rs1's value
          regb  : out word;--rs2's value
          addrw : in  reg_addr_type;--write address
          dataw : in  word;--write data
          we    : in  std_logic);--write enable
end component regfile;--end

--Signals
signal inst:word;--instruction signal
--Instruction Fields
signal opcode:opcode_type;--opcode field of inst
signal funct3:funct3_type;--funct3 field of inst
signal rd:rd_type;--rd field of inst
signal rs1:rs1_type;--rs1 field of inst
signal rs2:rs2_type;--rs2 field of rs2
signal funct7:funct7_type;--funct7 field of inst
--Immediates
signal iimme:word;--i type immediate
signal simme:word;--s type immediate
signal bimme:word;--b type immediate
signal uimme:word;--u type immediate
signal jimme:word;--j type immediate
--Control Signals
signal op1_sel :Op1Sel; --Select for RS1
signal op2_sel :Op2Sel;--Select for RS2
signal func_sel : FuncSel;--Select for ALU Function
signal mem_we : MemWr;--Select for Mem Write Enable
signal rf_we : RFWen;--Select for RF Write Enable
signal wb_sel :  WBSel;--Select for RF Write Back Data
signal pc_sel :  PCSel;--Select for PC Write Back Data

--Component Data Outputs
signal alu_out:word;--alu output
signal reg_out_rs1:word;--rs1's value
signal reg_out_rs2:word;--rs2's value
signal dmem_out:word;--data memory output
signal pc_out:word;--program counter output
```

```
signal imem_out:word;--inst memory output

--Component Data Inputs
signal alu_in_op1:word;--input for alu op1
signal alu_in_op2:word;--input for alu op2
signal pc_in:word;--new address for program counter
signal pc_we:std_logic;--pc write enable
signal rf_in:word;--new data register field

--Temp Signals
signal pc_plus_4: word;--holds pc+4
signal pc_plus_jal_offset :word;--holds pc+offset
signal rs1_plus_jalr_offset:word;--holds rs1+offset
signal pc_plus_branch_offset:word;--pc+branch_offset

begin--start

alu0:alu port map(--mapping for alu
    alu_func =>func_sel,--chose alu function
    op1      =>alu_in_op1,--chose alu op1 input
    op2      =>alu_in_op2,--chose alu op2 input
    result   =>alu_out);--output of alu

dec0:decoder port map(--mapping of decoder
    inst=>inst,--input instruction
    rs1_val=>reg_out_rs1,--input rs1's value
    rs2_val=>reg_out_rs2,--input rs2's value
    op1_sel =>op1_sel, --Select for RS1
    op2_sel =>op2_sel,--Select for RS2
    func_sel =>func_sel,--Select for ALU Function
    mem_we =>mem_we,--Select for Mem Write Enable
    rf_we =>rf_we,--Select for RF Write Enable
    wb_sel =>wb_sel,--Select for RF Write Back Data
    pc_sel =>pc_sel--Select for PC Write Back Data
);

dmem0:dmem port map(--mapping for data memory
    reset =>rst,--reset
    clk   =>clk,--clock
    raddr =>alu_out,--32 bits address
    dout  =>dmem_out,--OUTPUT a WORD
    dsize =>funct3,--Data size, if we = '0' then read size, else write size
    waddr =>alu_out,--32 bits address
    din   =>reg_out_rs2,--INPUT a WORD
    we    =>mem_we,--write enable
    debug_output=>outputs);--output regbank0(0)

imem0:imem port map(--mapping for inst memory
        addr =>pc_out,--inst address = pc value
        dout =>imem_out);--output of inst memory

pc0:pc port map(--mapping for pc
        clk=>clk,--clock
        rst=>rst,--reset
```

```
        ra=>pc_out,--read address
        wa=>pc_in,--new pc in
        we=>pc_we);--pc write enable

rf0:regfile port map(--regsiter field mapping
          reset =>rst,--reset
          clk   =>clk,--clock
          addra =>rs1,--rs1
          addrb =>rs2,--rs2
          rega  =>reg_out_rs1,--rs1's value
          regb  =>reg_out_rs2,--rs2's value
          addrw =>rd,--rd
          dataw =>rf_in,--data to be written
          we    =>rf_we);--writen enable


pc_we<='1';--always enable pc write

pc_plus_4<=pc_out+4;--pc=PC+4
pc_plus_jal_offset<=pc_out+jimme;--PC=PC+J type immediate
rs1_plus_jalr_offset<=reg_out_rs1+iimme;--PC=rs1+i type immediate
pc_plus_branch_offset<=pc_out+bimme;--PC=PC+b type immediate

inst<=imem_out;--inst gets the output from inst memory

opcode<=inst(6 downto 0);--opcode filed of instruction
rd<=inst(11 downto 7);--rd filed of instruction
funct3<=inst(14 downto 12);--funct3 filed of instruction
rs1<=inst(19 downto 15);--rs1 filed of instruction
rs2<=inst(24 downto 20);--rs2 filed of instruction
funct7<=inst(31 downto 25);--funct7 filed of instruction

iimme<=get_signed_Iimme(inst);--gets i type immediate
simme<=get_signed_Simme(inst);--gets s type immediate
bimme<=get_signed_Bimme(inst);--gets b type immediate
uimme<=get_signed_Uimme(inst);--gets u type immediate
jimme<=get_signed_Jimme(inst);--gets j type immediate

alu_in_op1<=uimme when op1_sel = op1_uimme else reg_out_rs1;--
MUX for selecting op1 of ALU, can be either u type immediate or rs1's value

with op2_sel select alu_in_op2<=--MUX for selecting op2 of ALU, can be
pc_out when op2_pc,--pc's value
iimme when op2_iimme,--i type immediate
simme when op2_simme,--s type immediate
reg_out_rs2 when others;--rs2's value

with wb_sel select rf_in<=--MUX for selecting register file input, can be
pc_plus_4 when wb_pc4,--PC+4
dmem_out when wb_dmem,--data memory output
alu_out when others;--alu output
```

```
with pc_sel select pc_in<=--MUX for selecting PC input, can be
pc_plus_4 when pc_pc4,--PC+4
rs1_plus_jalr_offset when pc_jalr,--rs1+I type immediate
pc_plus_branch_offset when pc_branch,--PC+b type immediate
pc_plus_jal_offset when others;--PC+j type immediate

end architecture rtl;--END
```

rv32ip2.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity rv32ip2 is--rv32i with 2 pipeline stages
port(clk:in std_logic;--clock
    rst:in std_logic;--reset
    outputs: out word);--debug output
end entity rv32ip2;--end

architecture rtl of rv32ip2 is--implement rs32i with 2 stage pipeline

--DEFINE alu START
component alu is
    port (alu_func : in  FuncSel;--Selection signal of ALU functions
          op1      : in  word;--Oprand 1 input to ALU
          op2      : in  word;--Oprand 2 input to ALU
          result   : out word);--The output of the given ALU function
end component alu;--DEFINE alu END

component decoder is--decoder
    port(inst:in word;--instruction in
        rs1_val:in word;--rs1's value in
        rs2_val:in word;--rs2's value in
        op1_sel : out Op1Sel; --Select for RS1
        op2_sel : out Op2Sel;--Select for RS2
        func_sel : out FuncSel;--Select for ALU Function
        mem_we : out MemWr;--Select for Mem Write Enable
        rf_we : out RFWen;--Select for RF Write Enable
        wb_sel : out WBSel;--Select for RF Write Back Data
        pc_sel : out PCSel);--Select for PC Write Back Data
end component decoder;--END

component dmem is--data memory
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          raddr : in  mem_addr_type;--32 bits address
          dout  : out word;--OUTPUT a WORD
          dsize:in funct3_type;--Data size, if we = '0' then read size, else write size
          waddr : in  mem_addr_type;--32 bits address
```

```
          din : in  word;--INPUT a WORD
          we    : in  std_logic;--write enable
          debug_output:out word--debug output, regbank0(0)
          );
end component dmem;--END

component imem is--inst memory
    port(addr : in mem_addr_type;--input address
        dout : out word);--output instruction
end component imem;--END

component pc is--program counter
    port(clk: in std_logic;--clock
        rst: in std_logic;--reset
        ra: out mem_addr_type;--read address
        wa:in mem_addr_type;--write address
        we:in std_logic);--write enable
end component pc;--END

component regfile is--register file
    port (reset : in  std_logic;--reset
          clk   : in  std_logic;--clock
          addra : in  reg_addr_type;--rs1
          addrb : in  reg_addr_type;--rs2
          rega  : out word;--rs1's value
          regb  : out word;--rs2's value
          addrw : in  reg_addr_type;--write address
          dataw : in  word;--write data
          we    : in  std_logic);--write enable
end component regfile;--end

component reg32 is--general purpose 32bit register
    port(clk: in std_logic;--clock
        rst: in std_logic;--reset
        din: in word;--new data in
        we : in std_logic;--write enable
        dout: out word);--data out
end component reg32;--end

component pplcontrol is--pipeline control signal generator
port(inst_s1:in word;--stage one instruction input
    inst_s2:in word;--stage two instruction input
    stall_disen:out std_logic;--output of stall signal
    fwd_op1_sel:out std_logic_vector(1 downto 0);--forwarding to op1 selection
    fwd_op2_sel:out std_logic_vector(1 downto 0);--forwarding to op2 selection
    fwd_rs2_en: out std_logic);--for resolving store RAW hazard
end component pplcontrol;--end

---Signals
signal inst:word;--instruction signal
--Instruction Fields
signal opcode:opcode_type;--opcode field of inst
signal funct3:funct3_type;--funct3 field of inst
signal rd:rd_type;--rd field of inst
```

```
signal rs1:rs1_type;--rs1 field of inst
signal rs2:rs2_type;--rs2 field of rs2
signal funct7:funct7_type;--funct7 field of inst
--Immediates
signal iimme:word;--i type immediate
signal simme:word;--s type immediate
signal bimme:word;--b type immediate
signal uimme:word;--u type immediate
signal jimme:word;--j type immediate

--Component Data Outputs
signal alu_out:word;--alu output
signal reg_out_rs1:word;--rs1's value
signal reg_out_rs2:word;--rs2's value
signal dmem_out:word;--data memory output
signal pc_out:word;--program counter output
signal imem_out:word;--inst memory output

--Component Data Inputs
signal alu_in_op1:word;--input for alu op1
signal alu_in_op2:word;--input for alu op2
signal pc_in:word;--new address for program counter
signal pc_we:std_logic;--pc write enable
signal rf_in:word;--new data register field

--Temp Signals
signal pc_plus_4: word;--holds pc+4
signal pc_plus_jal_offset :word;--holds pc+offset
signal rs1_plus_jalr_offset:word;--holds rs1+offset
signal pc_plus_branch_offset:word;--pc+branch_offset

--Pipline Signals
--outputs
signal rEIR_out:word;--Execute stage IR Output
signal rEop1_out:word;--Execute stage Op1 Buffer OUT
signal rEop2_out:word;--Execute stage Op2 Buffer OUT
signal rErs1_out:word;--Execute stage rs1 Buffer OUT
signal rErs2_out:word;--Execute stage rs1 Buffer OUT
--inputs
signal rEIR_in : word;----Execute stage IR input
signal rEop1_in:word;--Execute stage Op1 Buffer input
signal rEop2_in:word;--Execute stage Op2 Buffer input
signal rErs2_in:word;--Execute stage rs2 Buffer input
--Control Signals
signal op1_sel :Op1Sel; --Select for RS1
signal op2_sel :Op2Sel;--Select for RS2
signal func_sel : FuncSel;--Select for ALU Function
signal mem_we : MemWr;--Select for Mem Write Enable
signal rf_we : RFWen;--Select for RF Write Enable
signal wb_sel :  WBSel;--Select for RF Write Back Data
signal pc_sel :  PCSel;--Select for PC Write Back Data
--Stall Control Signal
signal stall_disen:std_logic;--'1' =>disabled, '0'=>enabled
--Forwarding Signal
```

```
signal fwd_op1_sel:std_logic_vector(1 downto 0);--
Forwarding to op1, 00=>tmp_fwd_mux_op1,01=>alu_out,10=>dmem_out
signal fwd_op2_sel:std_logic_vector(1 downto 0);--
Forwarding to op2, 00=>tmp_fwd_mux_op2,01=>alu_out,10=>dmem_out
signal fwd_rs2_en:std_logic;--Forwarding to rs2 Buffer, 1=enable, 0=disable

signal tmp_fwd_mux_op1:word;--temp signal for holding the value comes from op1 MUX
signal tmp_fwd_mux_op2:word;--temp signal for holding the value comes from op2 MUX
signal tmp_imem_out:word;--holds the output from instruction memory

signal stage2_iimme:word;--stage 2 i type immediate
signal stage2_bimme:word;--stage 2 b type immediate
signal stage2_jimme:word;--stage 2 j type immediate

begin--START of process

alu0:alu port map(--mapping for alu in stage two
    alu_func =>func_sel,--chose alu function
    op1      =>alu_in_op1,--chose alu op1 input
    op2      =>alu_in_op2,--chose alu op2 input
    result   =>alu_out);--output of alu

dec_stage1:decoder port map(--
For Stage one decoder, it only chooeses for op1_sel and op2_sel.
    inst=>imem_out,--stage one instruction in
    rs1_val=>reg_out_rs1,--actually not in use
    rs2_val=>reg_out_rs2,--actually not in use
    op1_sel =>op1_sel, --Select for RS1
    op2_sel =>op2_sel);--Select for RS2



dec_stage2:decoder port map(--
For stage two decoder, there is no need for selecting op1/op2_sel
    inst=>rEIR_out,--input stage 2 instruction
    rs1_val=>rErs1_out,--input stage 2 rs1's value
    rs2_val=>rErs2_out,--input stage 2 rs2's value
    func_sel =>func_sel,--Select for ALU Function
    mem_we =>mem_we,--Select for Mem Write Enable
    rf_we =>rf_we,--Select for RF Write Enable
    wb_sel =>wb_sel,--Select for RF Write Back Data
    pc_sel =>pc_sel);--Select for PC Write Back Data


dmem0:dmem port map(--Stage 2 data memory
    reset =>rst,--reset
    clk   =>clk,--clock
    raddr =>alu_out,--32 bits address
    dout  =>dmem_out,--OUTPUT a WORD
    dsize =>rEIR_out(14 downto 12),--
Data size, if we = '0' then read size, else write size
    waddr =>alu_out,--32 bits address
    din   =>rErs2_out,--INPUT a WORD
    we    =>mem_we,--write enable
```

```
    debug_output=>outputs);--output regbank0(0)

imem0:imem port map(--Stage 1 instruction memory
        addr =>pc_out,--reads input from PC
        dout =>tmp_imem_out);--output the instruction

pc_stage1:pc port map(--Stage one program counter
        clk=>clk,--clock
        rst=>rst,--reset
        ra=>pc_out,--read address input
        wa=>pc_in,--write address input
        we=>pc_we);--write enable

rf0:regfile port map(--Stage one register field
          reset =>rst,--reset
          clk   =>clk,--clock
          addra =>rs1,--rs1
          addrb =>rs2,--rs2
          rega  =>reg_out_rs1,--rs1's value
          regb  =>reg_out_rs2,--rs2's value
          addrw =>rEIR_out(11 downto 7),--address write from stage 2 instruction
          dataw =>rf_in,--new data write
          we    =>rf_we);--write enable


---------------------------------------Pipeline
reg_E_IR: reg32 port map(--Execute Stage Inst Register
        clk=>clk,--clock
        rst=>rst,--reset
        din=>rEIR_in,--input can be bubble or stage 1 instruction
        we =>enable,--always enable
        dout=>rEIR_out);--output to stage 2
reg_E_op1: reg32 port map(--Execute Stage op1 buffer
        clk=>clk,--clock
        rst=>rst,--reset
        din=>rEop1_in,--input is multiplexed due to forwarding
        we =>enable,--write enable
        dout=>rEop1_out);--output to second stage
reg_E_op2: reg32 port map(--Execute Stage op2 buffer
        clk=>clk,--clock
        rst=>rst,--reset
        din=>rEop2_in,--input is multiplexed due to forwarding
        we =>enable,--write enable
        dout=>rEop2_out--output to second stage
        );
reg_E_rs1: reg32 port map(--Execute Stage rs1 buffer
        clk=>clk,--clock
        rst=>rst,--reset
        din=>reg_out_rs1,--buffer stage 1 rs1's value
        we =>enable,--write enable
        dout=>rErs1_out);--output to second stage
reg_E_rs2: reg32 port map(--Execute Stage rs2 buffer
        clk=>clk,--clock
        rst=>rst,--reset
```

```
        din=>rErs2_in,--input is multiplexed due to forwarding
        we =>enable,--write enable
        dout=>rErs2_out);--output to second stage

pcl:pplcontrol port map(--pipeline control signal gennerator
    inst_s1=>imem_out,--stage 1 instruction input
    inst_s2=>rEIR_out,--stage 2 instruction input
    stall_disen=>stall_disen,--stall signal
    fwd_op1_sel=>fwd_op1_sel,--forwarding to op1
    fwd_op2_sel=>fwd_op2_sel,--forwarding to op2
    fwd_rs2_en=>fwd_rs2_en);--forwarding to reg_E_rs2

----------------------------------------END Pipeline

--Pipeline--
--Stall MUX
rEIR_in<=imem_out; --stage 1 instruction goes to buffer
imem_out<=tmp_imem_out when stall_disen='1' else NOP;--when stall_disen='1' else NOP;

--Forwarding MUX for op1
with fwd_op1_sel select rEop1_in<=
tmp_fwd_mux_op1 when "00",--when chosen 00, it reads from normal op1 mux
alu_out when "01",--when 01, forwarding alu output to op1
dmem_out when others;--forwarding data memory output to op1

--Forwarding MUX for op2
with fwd_op2_sel select rEop2_in<=--when chosen 00, it reads from normal op2 mux
tmp_fwd_mux_op2 when "00",--when 01, forwarding alu output to op2
alu_out when "01",--when 01, forwarding alu output to op2
dmem_out when others;--forwarding data memory output to op2

alu_in_op1<=rEop1_out;--op1 buffer outputs value to alu op1 input
alu_in_op2<=rEop2_out;--op2 buffer outputs value to alu op2 input

--rs2 buffer MUX
rErs2_in<=reg_out_rs2 when fwd_rs2_en = '0' else alu_out;--
if 0 normal rs2's value, else buffer alu output

----------------------

stage2_iimme<=get_signed_Iimme(rEIR_out);--gets stage 2 i type immediate
stage2_bimme<=get_signed_Bimme(rEIR_out);--gets stage 2 b type immediate
stage2_jimme<=get_signed_Jimme(rEIR_out);--gets stage 2 j type immediate

pc_we<='1';--always enable pc write

pc_plus_4<=pc_out+4;--stage 1 PC=PC+4
pc_plus_jal_offset<=pc_out+stage2_jimme -4 ;--stage 1 PC=PC+stage 2 j type immediate -4
rs1_plus_jalr_offset<=rErs1_out+stage2_iimme;--
PC = stage two rs2's value + stage 2 i type immediate
pc_plus_branch_offset<=pc_out+stage2_bimme-4;--PC = PC + stage 2 b type immediate - 4
```

```
inst<=imem_out;-- instruction signal

opcode<=inst(6 downto 0);--opcode filed of instruction
rd<=inst(11 downto 7);--rd filed of instruction
funct3<=inst(14 downto 12);--funct3 filed of instruction
rs1<=inst(19 downto 15);--rs1 filed of instruction
rs2<=inst(24 downto 20);--rs2 filed of instruction
funct7<=inst(31 downto 25);--funct7 filed of instruction

iimme<=get_signed_Iimme(inst);--gets i type immediate
simme<=get_signed_Simme(inst);--gets s type immediate
bimme<=get_signed_Bimme(inst);--gets b type immediate
uimme<=get_signed_Uimme(inst);--gets u type immediate
jimme<=get_signed_Jimme(inst);--gets j type immediate

--MUX for selecting normal op1 of ALU, can be either u type immediate or rs1's value
tmp_fwd_mux_op1<=uimme when op1_sel = op1_uimme else reg_out_rs1;

with op2_sel select tmp_fwd_mux_op2<=--MUX for selecting normal op2 of ALU, can be
pc_out when op2_pc,--pc's value
iimme when op2_iimme,--i type immediate
simme when op2_simme,--s type immediate
reg_out_rs2 when others;--rs2's value

with wb_sel select rf_in<=--MUX for selecting register file input, can be
pc_out when wb_pc4,--PC
dmem_out when wb_dmem,--data memory output
alu_out when others;--alu output


with pc_sel select pc_in<=--MUX for selecting PC input, can be
pc_plus_4 when pc_pc4,--stage 1 PC+4
rs1_plus_jalr_offset when pc_jalr,--stage 2 rs1's value + stage2 I type immediate
pc_plus_branch_offset when pc_branch,--stage 1 PC+stage 2 b type immediate-4
pc_plus_jal_offset when others;--PC+ stage2 j type immediate-4

end architecture rtl;--END
```

test_rv32i.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd

entity u is--testbench
end entity u;--testbench

architecture test of u is--testbench for not pipelined rv32i

component rv32i is--testbench for not pipelined rv32i
```

```
port(clk:in std_logic;--clock
    rst:in std_logic;--rest
    outputs: out word);--data output regbank0(0)
end component rv32i;--testbench for not pipelined rv32i

constant period : time := 10 ns;--period is 10 ns
signal clk:std_logic:='0';--init clock 0
signal rst:std_logic:='1';--init reset 1
signal val:word:=x"00000000";--tmp variable

begin--testbench for not pipelined rv32i

cpu:rv32i port map(--testbench for not pipelined rv32i
    clk=>clk,--link clock
    rst=>rst,--link reset
    outputs=>val--link data output
);--testbench for not pipelined rv32i

    proc_clock: process--clock process
    begin--testbench for not pipelined rv32i
        clk <= '0';--0 clock
        wait for period/2;--after one half period
        clk <= '1';--1 clock
        wait for period/2;--repeat
    end process;--testbench for not pipelined rv32i

    proc_stimuli: process--testbench for not pipelined rv32i
    begin--testbench for not pipelined rv32i
        rst <= '1';--reset
        wait for period;--for 2 periods
        rst <= '0';--not reset
        wait for period*9;--run 9 periods
        wait for period*507;--Calculations
        wait for period;--run
        wait for period;--run
        wait for period;--run
        wait for period;--run
        wait for period;--run

        assert false report "success - end of simulation" severity failure;--
end of simulation
    end process;--testbench for not pipelined rv32i

end architecture test;--testbench for not pipelined rv32i
```

test_rv32ip2.vhd

```
--Use IEEE Standard Library
library ieee;
use ieee.std_logic_1164.all;--Use std_logic_1164.vhd
use ieee.std_logic_unsigned.all;--Use std_logic_unsigned.vhd
use ieee.numeric_std.all;--use numeric_std.vhd
--Use the work Library
library work;
use work.common.all;--Use the constants defined in common.vhd
```

```
entity u_p2 is--testbench
end entity u_p2;--testbench

architecture test of u_p2 is--testbench for 2 stage pipelined rv32i

component rv32ip2 is--testbench for 2 stage pipelined rv32i
port(clk:in std_logic;--clock
    rst:in std_logic;--rest
    outputs: out word);--data output regbank0(0)
end component rv32ip2;--testbench for 2 stage pipelined rv32i

constant period : time := 10 ns;--period is 10 ns
signal clk:std_logic:='0';--init clock 0
signal rst:std_logic:='1';--init reset 1
signal val:word:=x"00000000";--tmp variable

begin--testbench for 2 stage pipelined rv32i

cpu:rv32ip2 port map(--testbench for 2 stage pipelined rv32i
    clk=>clk,--link clock
    rst=>rst,--link reset
    outputs=>val--link data output
);--testbench for 2 stage pipelined rv32i

    proc_clock: process--testbench for 2 stage pipelined rv32i
    begin--testbench for 2 stage pipelined rv32i
        clk <= '0';--0 clock
        wait for period/2;--after one half period
        clk <= '1';--1 clock
        wait for period/2;--repeat
    end process;--testbench for 2 stage pipelined rv32i

    proc_stimuli: process--testbench for 2 stage pipelined rv32i

    begin--testbench for 2 stage pipelined rv32i
        rst <= '1';--1 reset
        wait for period * 2;--run for 2 periods
        rst <= '0';--0 reset
        while unsigned(val)/=x"00006041" loop--run unitl we have the result
        wait for period;--keep running
        end loop;--end loop
        assert false report "success - end of simulation" severity failure;--
end of simulation
    end process;--testbench for 2 stage pipelined rv32i

end architecture test;--testbench for 2 stage pipelined rv32i
```