ECE585

ADVANCED COMPUTER ARCHITECTURE

PROJECT II

SIMULATION OF CPU, CACHE, BUS AND MEMORY DATAPATH

Ashutosh Jain- A20325782

Harshita Ravi Shankar- A20327797

ABSTRACT

The main objective of this project is the simulation of the datapath of CPU, cache, bus, memory for a 32-bit version of a MIPS processor to understand the functions of the architecture using VHDL. We have chosen 8 different instruction sets for the complete processing and simulated. The four major blocks along with accessing them are implemented as given. The Write Through and No Write Allocate strategy has been employed for the cache.

The CPU is first provided with a set of instructions which have a PC address each. Once the CPU receives the address, the cache is accessed to pull up and send the corresponding instruction back to the CPU. It will know the relevant information about the instruction, like the type, functions and the registers it uses. The instructions generally access other registers to read or change its contents which requires another cache access to in turn do the same. Whereas some need arithmetic operations performed by the ALU located in the CPU. The type of instructions that change the sequence of instructions with change in the PC do not access the cache.

When the cache is accessed, the data may or may not be there. If it isn't in the cache, the bus has to be accessed and then the memory to get the data and make the necessary changes depending on the instruction used.

A testbench is used to test the design and show the changes made by each of the instructions. It simulates each of the 8 instruction sets. The result indicates the changes in the registers made after each instruction is completed. The clock frequency of 100Hz per second along with a 10ms time period which is counted as a single clock cycle is used for the entire module.

1. INTRODUCTION

The CPU-Cache-Bus-Memory model is an integral design that performs operations as per the instructions designated. Since the cache is a smaller is size as compared to the memory which stores the data most likely to be used by the CPU, reducing the time required to access the memory. The modules are coded in VHDL language in Xilinx ISE framework. The optimizations to match the cache and memory unit access times is most important.

The next section explains all the steps taken to implement each of the major blocks in detail, presenting each important block of the design as a separate module. The number of cycles needed by every instruction set and the design challenges faced are tabulated in section 3. The VHDL code along with the screenshots of the test results for each of the modules designed have been attached in Appendix section.

2. DESIGN

Each of the modules are programmed along with their input output ports, their functional working described, and the assumptions made for designing the complete architecture is explained as follows.

A. CPU

The CPU has to only get the PC and gives the address of the destination register on the Bus as the output. The PC is the address of the instruction that has to be performed that is received from the I-cache (Instruction memory) which is then sent to the Instruction Register (IR) of the CPU in order to decode the instructions. The type of instruction and the operation to be performed and the values used for the operation are given by the Control Unit. For instructions like ADD and LUI, the arithmetic operation is performed and saved to the register directly and the cache is not accessed.

The five cycles of the CPU are implemented for this module. The instruction fetch (IF) cycle is done directly and the cache provides the CPU with instruction that is implemented as the instruction input. Hence the cycles used are ID, EXE, MEM, WB, each of which are described by different modules and integrated in the Top CPU Module.

The Block diagram of the CPU top module in Fig. 1, describes the sub modules and the internal signals. The type of instruction is decided by the number of clock cycles that the CPU takes. The load instruction uses a total of 4.5 cycles for execution, the store instruction 3 cycles. The R type instructions use 3.5 cycles, branch and Jump instruction use 2 cycles.

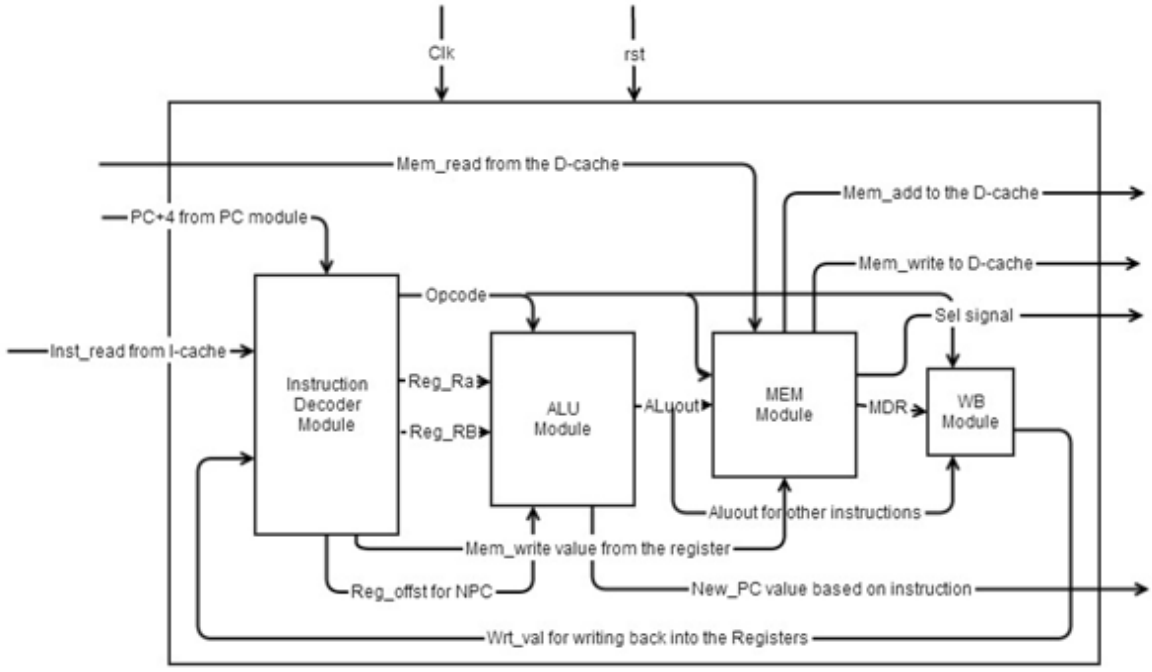

Fig. 1: CPU Module

- Instruction Decoder

The purpose of the Instruction decoder cycle is to decode instructions and the modules are given the respective inputs from the general purpose registers.

```
entity ID is
port(   clk,rst                     : in STD_LOGIC;
        IR,PC,Wrt_Val               : in STD_ULOGIC_VECTOR(31 downto 0);
        opcode                      : out STD_ULOGIC_VECTOR(5 downto 0);
        shft                        : out STD_ULOGIC_VECTOR(4 downto 0);
        RA,RB,Reg_Offset            : out STD_ULOGIC_VECTOR(31 downto 0);
        Mem_write                   : out STD_ULOGIC_VECTOR(31 downto 0)
    );
end ID;
```

The instruction decoder has been designed to complete within a single clock cycle. The data registers are first initialized in the reset stage. The instruction register is decoded at the rising edge of the clock cycle. The register write is done at the rising edge of the clock cycle and the register read at the falling edge.

```
entity IR is
port (  clk,rst      : in STD_LOGIC;
        IR_reg       : in STD_LOGIC_VECTOR(31 downto 0);
        PC           : in STD_LOGIC_VECTOR(31 downto 0);
        Aluout       : in STD_LOGIC_VECTOR(31 downto 0);
        Reg_A,Reg_B  : out STD_LOGIC_VECTOR(31 downto 0);
        Reg_offst    : out STD_LOGIC_VECTOR(31 downto 0);
        Mem_write    : out STD_LOGIC_VECTOR(31 downto 0);
        opcode       : out STD_LOGIC_VECTOR(5 downto 0);
     );
end IR;
```

The registers R_A and R_B will be the outputs from this module which will be the input to the EXE module along with the control signal opcode. Based on the instruction, the value of NPC is provided by the Reg_off.

- Execution Module

The Execution module is also called the ALU module, where all the ALU operations are done on the control signals, is one of the main modules.

```
entity EX is
port(   clk,rst                 : in STD_LOGIC;
        PC,RA,RB,Reg_Offset     : in STD_ULOGIC_VECTOR(31 downto 0);
        opcode                  : in STD_ULOGIC_VECTOR(5 downto 0);
        shft                    : in STD_ULOGIC_VECTOR(4 downto 0);
        NPC, Aluout             : out STD_ULOGIC_VECTOR(31 downto 0)
    );
end EX;
```

This module depends on the control signal which decides the type of operation, for its execution. The two outputs of this module are NPC which holds the new PC value and ALuout which holds the output of any operation. A case statement is used to define the operation.

The NPC for the instructions LW, SW, ADD, ADDI are sequential PC value or the destination address depending on the Branch or Jump instructions. This module is implemented with the clock and reset signals and hence, the module utilizes the entire clock cycle for its execution.

- Memory Module

The Load and Store instructions are taken care of by this memory module. It accesses the memory based on the address for the respective data inputs.

```
entity MM is
port    (
        clk,rst     : in STD_LOGIC;
        opcode      : in STD_ULOGIC_VECTOR(5 downto 0);
        RA, Aluout  : in STD_ULOGIC_VECTOR(31 downto 0);
        Mem_read    : in STD_ULOGIC_VECTOR(31 downto 0);
        Mem_write   : out STD_ULOGIC_VECTOR(31 downto 0);
        MDR         : out STD_ULOGIC_VECTOR(31 downto 0);
        Mem_add     : out STD_ULOGIC_VECTOR(31 downto 0);
        sel         : out STD_LOGIC
        );
end MM;
```

This behaves as the interface between the CPU and the memory. The opcode that is the control signal and the ALuout are the inputs to this module. Memory read and write are the other port specifications. The MDR Port is the value retrieved from the memory to write in to the register.

The MEM Module is in use only during the Load and store instructions. The execution is done in one cycle. The MEM module writes into the register at the rising edge and reads value at the falling edge of the clock cycle. The Sel which is a single bit select signal differentiates between the load and store instructions.

- Write Back Module

This module will write back the ALuout or the memory data into the registers.

```
entity WB is
port(   clk,rst     : in STD_LOGIC;
        MDR,Aluout  : in STD_ULOGIC_VECTOR(31 downto 0);
        opcode      : in STD_ULOGIC_VECTOR(5 downto 0);
        Wrt_Val     : out STD_ULOGIC_VECTOR(31 downto 0)
    );
end WB;
```

The port structure of this module includes the two inputs Aluout and the MDR from the EXE and the MEM module. The opcode which is the control signal controls this module. It selects the write back value into the registers depending on the control signal. The WB module, like the other modules are executed in a single clock cycle.

B. CACHE

The cache module is implemented as a split cache with an Instruction cache block of 512Bytes and a Data cache block of 256 Bytes. Two concurrently running processes are defined separately for I-cache and D-cache. The operating clock frequency is 100Hz (for simplicity).The read operations are done at the falling edge and write operations at the rising edge of the clock cycle. The cache access takes place in a single clock cycle (10ms) by default. It consists of 128 blocks and each of them are 16 Bytes of data/instruction. Hence, the total memory size is 4096 Bytes. Direct memory mapping is used to map the blocks to the cache. The cache follows Write through

and No-Write allocate for Write Hit and Write miss respectively. Hence, the concept of dirty bit has not been implemented here.
The PC provides the address of the instruction and the I-cache checks for a hit/miss. Since direct mapping is used, first the block in I-cache to which the instruction would be mapped from the memory is given from decoding the address. It is a hit if the block contents are not null and the instruction will be programmed into it before the execution is started. If the data is not loaded in cache (i.e., commented out), there will be an instruction miss.

Similarly, the data hit/miss is checked after the address of load/store operations are calculated making use of the address from the CPU. Since we are using the Write Through strategy, the block will not be checked for replacement and a dirty bit is not needed here. When there is a read miss on the D-cache, the data comes from the memory into the cache and utilized. When there is a write miss on the D-cache, the data will be written to the memory only and not to the cache. Hence, even in the following clock cycles, it will never be a hit. In the case of a hit/miss scenario corresponding signals are generated to tell the bus to request instruction or data from the local memory.

```
entity Cache is
    Port (      clk,rst                     : in STD_LOGIC;
                I_cache_in,D_cache_in       : in STD_ULOGIC_VECTOR(255 downto 0);
                npc_in                      : in STD_ULOGIC_VECTOR(31 downto 0);
                D_addr                      : in STD_ULOGIC_VECTOR(31 downto 0);
                ldsw                        : in STD_LOGIC; --'0'=>Load, '1'=>Store
                rdy_inst,rdy_data           : in STD_LOGIC;
                data_in                     : in STD_ULOGIC_VECTOR(31 downto 0);
                I_cache_out,D_cache_out     : out STD_ULOGIC_VECTOR(31 downto 0);
                I_hit,rd_inst               : out STD_LOGIC;
                iaddr_out                   : out STD_ULOGIC_VECTOR(31 downto 0);
                D_hit                       : out STD_LOGIC;
                rd_data,wt_data             : out STD_LOGIC;
                daddr_out                   : out STD_ULOGIC_VECTOR(31 downto 0));
end Cache;
```

The bus module takes care of the cycles to access memory in case of a miss. Finally the Instruction or data obtained from the memory are wired to Instruction decoder or the
ALU respectively for further operation. The ALU and branch operations do not make use of the D-cache and therefore are completed much faster than the Load and Store operations.

C. BUS MODULE

The Bus module employs all the delays in the access time of local memory for read/write miss scenarios. The bus is modeled to include explicit delays since there is no hardware.

In this project, the datapath has been designed in such a way that the bus receives all the control signals from the cache in either of the hit or miss conditions. During the miss condition is when the signals access data from memory. Therefore, the bus module always delays the read/write requests which are forwarded from the bus to cache from reaching the local memory. The delays

that are observed from the results will be much higher because of the internal dependencies at the top module.

Similarly, the same is repeated for Instruction Read miss, Data Read miss, Data Write hit and Data Write miss. Memory is accessed during the hit and miss scenarios for a store operation which results in a delayed execution time from the memory.

D. MEMORY MODULE

Along with the other modules, this memory module also works in synchronous with the CPU, cache and bus modules with a clock frequency of 100Hz. The main function of this module is give the required data to the cache or CPU when requested during read operations and during store operations, to write data into itself. All the instruction sets are by default loaded into their designated addresses in the local memory.

The instructions are loaded as mentioned in the project. The custom set instruction is selected as per the CWID, #2 which include NOR, SLL, JR. The data is loaded from the memory for read instruction and the address in which the data has to be stored is calculated by the ALU for store instruction. It doesn't follow the same logic as for the cache module during read and store operations. It has been optimized such that the read and write are performed at any clock edge in order to make it faster.

```
entity Memory is
Port(   clk,rst      : in STD_LOGIC;
        r_inst       : in STD_LOGIC;
        inst_addr    : in STD_ULOGIC_VECTOR(31 downto 0);
        r_data       : in STD_LOGIC;
        w_data       : in STD_LOGIC;
        data_addr    : in STD_ULOGIC_VECTOR(31 downto 0);
        mem_in       : in STD_ULOGIC_VECTOR(31 downto 0);
        ins_rdy      : out STD_LOGIC;
        data_rdy     : out STD_LOGIC;
        mem_out      : out STD_ULOGIC_VECTOR(255 downto 0)
    );
end Memory;
```

For read requests, the operation is similar but the data from the local memory is brought outside using a line. All the read/write operations are turned on only on receiving a logic '1' at the control signals transferred from the bus module after introducing deliberate delays to account for memory access timings.

E. TOP MODULE

The Top Module is the overall module which includes all the previously described modules as components. They are the CPU, cache, bus and memory modules. The block diagram is as shown below.

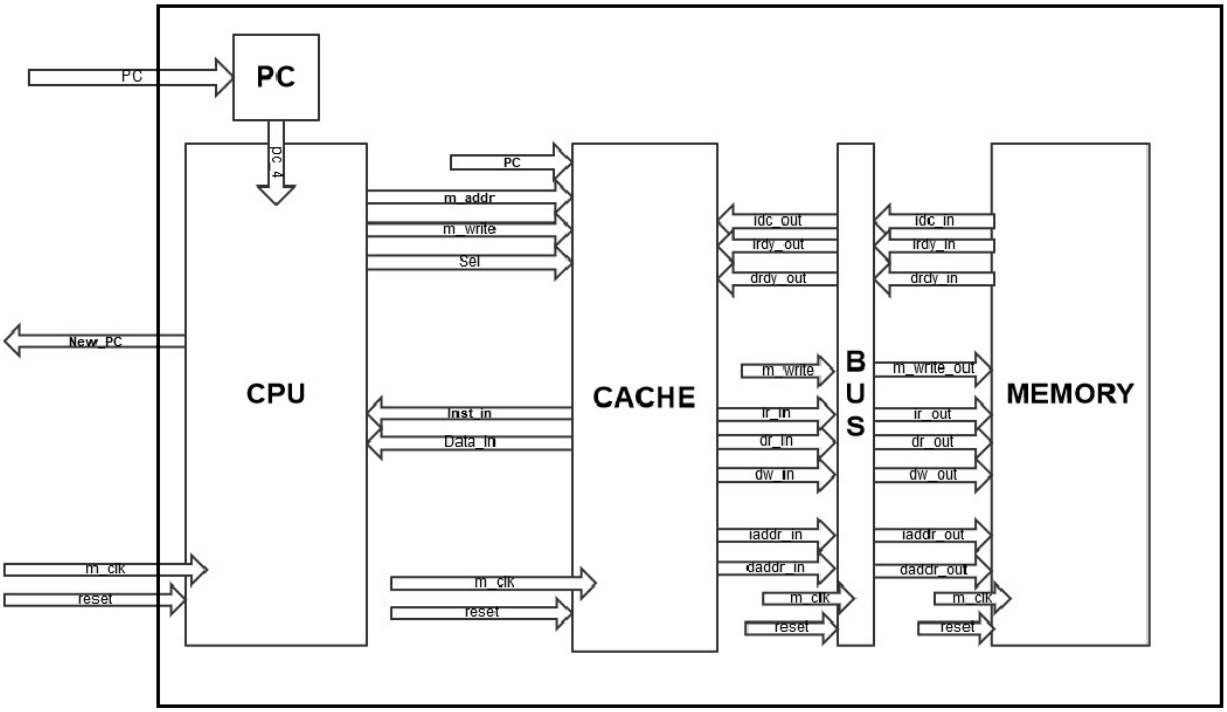

Fig. 2: Block Diagram

The block diagram shows all the modules and the interconnections with the corresponding names. The main clock of the top module is the same as the one given to each of the other modules, i.e., 100Hz. It is initially in the reset state for 20ms before the processing. Since the processor waits for 20ms before processing the pc address, all the memory initializations are done in this reset state.
When the pc address is given as input, the PC calculates the NPC value and passes the previous pc to cache module. In the cache, the address is checked in the I-cache for a hit or miss scenario. In case of a hit, the instruction at the direct mapped address is searched and brought to the CPU. If it is a miss, the appropriate control signal is sent to the bus which delays the signal from reaching the memory and retrieves the data at the appropriate moment.

The CPU on receiving the instruction from either the cache or memory, will decode it and determine the type of instruction that is being processed. Depending on the instruction type, the ALU calculates and gives the address or value that has to be stored in a register inside the CPU. In case the instruction is a load or store operation, the D-cache is checked for reading or writing the data and the hit or miss cases will be followed. The end of execution is thus denoted by the time the final data is written into the general purpose register.

3. IMPLEMENTATION

In our design we will be performing a set of instructions and spot the changes in the registers according to each of these instructions. Each instruction have a corresponding sequence that describes it and specify the registers used. The instructions used are as follows:
There are three types of instructions: I-type, R-type and J-type instructions.

The last column in the above table gives the actual number of cycles needed by each instruction depending on whether it is a cache hit or miss. For example, consider the first load instruction, it loads data into the memory [100 + Reg T2] and puts it in Reg S1. Both the address and data are a hit in the cache. The number of cycles is calculated as follows.
# of Cycles =
Icache access + Dcache access + 5 Cycles for pipeline stages (IF+ID+EX+MEM+WB)

Thus for Load Instruction, # of Cycles = 1 + 1 + 5 = 7 cycles. The observed number of cycles is 8 because of the write delay that is in the cache. This is calculated for all the rest of the instructions in a similar manner and verified against the tabulated values.

4. CONCLUSION

The entire datapath of CPU-Cache-Bus-Memory for a 32-bit MIPS processor has been implemented using VHDL. The written code is simulated on the Xilinx framework for a set of instructions. The number of cycles required per instruction is observed and compared to the calculated theoretical values. The code and simulation has therefore been verified.

The VHDL code is given in the Appendix section of the report. The testbench is auto generated.

APPENDIX

1. CPU Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CPU is
port(       clk,rst                     : in STD_LOGIC;
            PC,IR,Mem_read                  : in STD_ULOGIC_VECTOR(31 downto
0);
            Mem_add,Mem_write,NPC   : out STD_ULOGIC_VECTOR(31 downto 0);
            Sel                             : out STD_LOGIC
     );
end CPU;

architecture Behavioral of CPU is

component PC4
port(
            PC                                              : in
STD_ULOGIC_VECTOR(31 downto 0);
            PCplus4                                         : out
STD_ULOGIC_VECTOR(31 downto 0)
   );
end component;

component ID
port(
            clk,rst                                 : in STD_LOGIC;
            IR,PC,Wrt_Val                           : in STD_ULOGIC_VECTOR(31
downto 0);

            opcode                                          : out
STD_ULOGIC_VECTOR(5 downto 0);
            shft                                            : out
STD_ULOGIC_VECTOR(4 downto 0);
            RA,RB,Reg_Offset                : out STD_ULOGIC_VECTOR(31 downto
0);
            Mem_write                               : out STD_ULOGIC_VECTOR(31
downto 0)
     );
end component;

component EX
port(       clk,rst                     : in STD_LOGIC;
```

```
        PC,RA,RB,Reg_Offset             : in STD_ULOGIC_VECTOR(31 downto
0);
        opcode                          : in STD_ULOGIC_VECTOR(5
downto 0);
        shft                            : in STD_ULOGIC_VECTOR(4 downto 0);
        NPC, Aluout                 : out STD_ULOGIC_VECTOR(31 downto 0)
    );
end component;

component MM
port(
        clk,rst                         : in STD_LOGIC;
        opcode                              : in
STD_ULOGIC_VECTOR(5 downto 0);
        RA, Aluout                      : in STD_ULOGIC_VECTOR(31
downto 0);
        Mem_read                        : in STD_ULOGIC_VECTOR(31
downto 0);

        Mem_write                       : out STD_ULOGIC_VECTOR(31
downto 0);
        MDR                                 : out
STD_ULOGIC_VECTOR(31 downto 0);
        Mem_add                             : out
STD_ULOGIC_VECTOR(31 downto 0);
        sel                                 : out STD_LOGIC
    );
end component;

component WB
port(
        clk,rst                         : in STD_LOGIC;
        MDR,Aluout                      : in STD_ULOGIC_VECTOR(31
downto 0);
        opcode                              : in
STD_ULOGIC_VECTOR(5 downto 0);

        Wrt_Val                             : out
STD_ULOGIC_VECTOR(31 downto 0)
  );
end component;

signal RA,RB,Reg_RT,Reg_Offset,Aluout,Wrt_Val,MDR,PCplus4,NFC :
STD_ULOGIC_VECTOR(31 downto 0);
signal opcode : STD_ULOGIC_VECTOR(5 downto 0);
signal shft   : STD_ULOGIC_VECTOR(4 downto 0);
```

```
    begin

    A1 : PC4 port map(PC,PCplus4);
    A2 : ID port map(clk,rst,IR,PCplus4,Wrt_Val,opcode,shft,RA,RB,Reg_Offset,Reg_RT);
    A3 : EX port map(clk,rst,PCplus4,RA,RB,Reg_Offset,opcode,shft,NFC,Aluout);
    A4 : MM port
    map(clk,rst,opcode,Reg_RT,Aluout,Mem_read,Mem_write,MDR,Mem_add,sel);
    A5 : WB port map(clk,rst,MDR,Aluout,opcode,Wrt_val);

    end Behavioral;
```

A. IR Register Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity IR is
port (   clk,rst  : in STD_LOGIC;
                  IR_reg                : in STD_LOGIC_VECTOR(31 downto 0);
                  PC                    : in STD_LOGIC_VECTOR(31 downto 0);
                  Aluout                : in STD_LOGIC_VECTOR(31 downto 0);
                  Reg_A,Reg_B : out STD_LOGIC_VECTOR(31 downto 0);
                  Reg_offst     : out STD_LOGIC_VECTOR(31 downto 0);
                  Mem_write     : out STD_LOGIC_VECTOR(31 downto 0);
                  opcode                : out STD_LOGIC_VECTOR(5 downto 0);
         );
end IR;

architecture Behavioral of IR is

begin


end Behavioral;
```

B. Instruction Decode

```
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;

    entity ID is
    port(       clk,rst                                 : in STD_LOGIC;
```

```
        IR,PC,Wrt_Val                   : in STD_ULOGIC_VECTOR(31
downto 0);
        opcode                          : out
STD_ULOGIC_VECTOR(5 downto 0);
        shft                            : out STD_ULOGIC_VECTOR(4
downto 0);
        RA,RB,Reg_Offset                : out STD_ULOGIC_VECTOR(31 downto
0);
        Mem_write                       : out STD_ULOGIC_VECTOR(31
downto 0)
    );
end ID;

architecture Behavioral of ID is

type array_type is array (0 to 31) of STD_ULOGIC_VECTOR(31 downto 0);
signal GPREG                            :           array_type;
signal op                               :
STD_ULOGIC_VECTOR(5 downto 0);
signal RS,RT,RD                 :           STD_ULOGIC_VECTOR(4
downto 0);
signal Imm                              :
STD_ULOGIC_VECTOR(15 downto 0);
signal Reg_off                  :
STD_ULOGIC_VECTOR(25 downto 0);
signal PC_4                             :
STD_ULOGIC_VECTOR(3 downto 0);
signal test                     :           STD_LOGIC;

begin
process(clk,rst)
begin
op  <= IR(31 downto 26);
PC_4    <= PC(31 downto 28);        --4 bits from MSB of PC

-- at reset state registers are initialized
if(rst = '1') then
    GPREG(0)  <= "00000000000000000000000000000000";
--lw            $s1,100($t2)          --$s1=R17 $t2=R10
    GPREG(17) <= "00000000000000000000000000000000";
    GPREG(10) <= "00000000000000000000000001101100";
--sw            $t6,200($t2)          --$t6=R14 $t2=R10
    GPREG(14) <= "00010101011111000001010111111111";
    GPREG(10) <= "00000000000000000000000110111000";
--add           $s2,$t4,$s4                 --$s2=R18 $t4=R12 $s4=R20
    GPREG(18) <= "00000000000000000000000000011111";
```

```
    GPREG(12) <= "00000000000000000000000000000010";
    GPREG(20) <= "00000000000000000000000000000000";
            -- change for BRANCH NOT TAKEN
--beq       $t5,$t1,200                 --$t5=R13 $t1=R9
    GPREG(13) <= "00000000000000000000000000000000";
    GPREG(9)  <= "00000000000000000000000000000000";
--xori      $t3,$s4,100                 --$t3=R11 $s4=R20
    GPREG(11) <= "00000000000000000000000000000000";
    GPREG(20) <= "00000000000000000000000000000000";
--nor       $s4,$t1,$t3
    GPREG(20) <= "00000000000000000000000000000000";
    GPREG(9) <= "00000000000000000000000000000000";
    GPREG(11) <= "00000000000000000000000000000000";
--sll           $s5,$s5,10                  --$s5=R22
    GPREG(22) <= "00000000000000000000000110000000";
--jr            $s3                                 --$s3=R19
    GPREG(19) <= "00001100000000011110000000011111";

else

case op is

-- SW instruction :
when "101011" =>
    if(rising_edge(clk) and (rst ='0')) then
            opcode      <= IR(31 downto 26);
            RS          <= IR(25 downto 21);
            RT          <= IR(20 downto 16);
            Imm         <= IR(15 downto 0);
    end if;

    if(falling_edge(clk) and (rst ='0')) then
            RA          <= GPREG(to_integer(unsigned(RS)));
            RB          <= "0000000000000000" & Imm;
            Mem_write <= GPREG(to_integer(unsigned(RT)));
    end if;

--R-type : ADD, NOR and SLL Inst
when "000000" =>
    if(rising_edge(clk) and (rst ='0')) then
            opcode      <= IR(5 downto 0);
            RS          <= IR(25 downto 21);
            RT          <= IR(20 downto 16);
            RD          <= IR(15 downto 11);
    end if;
```

```
    if(falling_edge(clk) and (rst ='0')) then
            RA          <= GPREG(to_integer(unsigned(RS)));
            RB          <= GPREG(to_integer(unsigned(RT)));
            GPREG(to_integer(unsigned(RD))) <= Wrt_Val;
    end if;

-- BEQ instruction
when "000100" =>
    if(rising_edge(clk) and (rst ='0')) then
            opcode      <= IR(31 downto 26);
            RS          <= IR(25 downto 21);
            RT                  <= IR(20 downto 16);
            Imm         <= IR(15 downto 0);
    end if;

    if(falling_edge(clk) and (rst ='0')) then
            RA          <= GPREG(to_integer(unsigned(RS)));
            RB          <= GPREG(to_integer(unsigned(RT)));
            Reg_Offset<= "0000000000000000" & Imm;
    end if;

-- J instruction
when "000010" =>
    if(rising_edge(clk) and (rst ='0')) then
            opcode      <= IR(31 downto 26);
            Reg_off     <= IR(25 downto 0);
    end if;

    if(falling_edge(clk) and (rst ='0')) then
            Reg_Offset<= PC_4 & Reg_off & "00";                 --first 4 bits of current
PC & multiply by 4
    end if;

-- JR instruction
when "000101" =>
    if(rising_edge(clk) and (rst ='0')) then
            opcode      <= IR(31 downto 26);
            RS          <= IR(25 downto 21);
    end if;

    if(falling_edge(clk) and (rst ='0')) then
            Reg_Offset<= GPREG(to_integer(unsigned(RS)));
    end if;

-- I-type instructions : LW and XORI Instructions
when others =>
```

```
        if(rising_edge(clk) and (rst ='0')) then
                opcode          <= IR(31 downto 26);
                RS                      <= IR(25 downto 21);
                RT              <= IR(20 downto 16);
                Imm             <= IR(15 downto 0);
        end if;

        if(falling_edge(clk) and (rst ='0')) then
                RA              <= GPREG(to_integer(unsigned(RS)));
                RB              <= "0000000000000000" & Imm;
                GPREG(to_integer(unsigned(RT))) <= Wrt_Val;
        end if;
    end case;
    if(IR(5 downto 0) = "000000") then
                shft            <= IR(10 downto 6);
    end if;
    end if;
    end process;
    end Behavioral;
```

C. EXE Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE.NUMERIC_BIT.ALL;



entity EX is
port(   clk,rst                         : in STD_LOGIC;
                PC,RA,RB,Reg_Offset             : in STD_ULOGIC_VECTOR(31 downto
0);
                opcode                                  : in STD_ULOGIC_VECTOR(5
downto 0);
                shft                            : in STD_ULOGIC_VECTOR(4 downto 0);
                NPC, Aluout                 : out STD_ULOGIC_VECTOR(31 downto 0)
        );
end EX;

architecture Behavioral of EX is
signal RA_temp, RB_temp, Aluout_temp, shft_temp : integer;
begin

process(clk,rst)
begin
```

```
if(rst = '1') then
	null;                                                    --Nothing
happens during Reset
else
--EX takes place during Rising edge of clk
	if( rising_edge(clk) and rst ='0') then
		case opcode is

--LW,SW,ADD Instructions
		when "100011" | "101011" | "100000" =>
			RA_temp <= to_integer(unsigned(RA));
			RB_temp <= to_integer(unsigned(RB));
			Aluout_temp <= RA_temp + RB_temp;
			Aluout <= STD_ULOGIC_VECTOR(to_unsigned(Aluout_temp, 32));
			NPC <= PC;                                       --NPC is PC+4 (not a
Branch Inst)

--BEQ Instruction
		when "000100" =>
			if (RA = RB) then
				NPC <= Reg_Offset;          --Branch Taken
			else
				NPC <= PC;                       --NPC is PC+4 (Branch Not
Taken)
			end if;

--XORI Instruction
		when "001110" =>
			Aluout <= RA xor RB;
			NPC <= PC;                                       --NPC is PC+4 (not a
Branch Inst)

--NOR Instruction
		when "011011" =>
			Aluout <= RA nor RB;
			NPC <= PC;                                       --NPC is PC+4 (not a
Branch Inst)

--SLL Instruction
		when "000000" =>
			RA_temp <= to_integer(unsigned(RA));
			shft_temp <= to_integer(unsigned(shft));
			for i in 1 to shft_temp loop
				Aluout <= RA(31 downto 0) & "0";
			end loop;
```

```
                        Aluout <= STD_ULOGIC_VECTOR(to_unsigned(Aluout_temp, 32));
                        NPC <= PC;                                      --NPC is PC+4 (not a
Branch Inst)

--J and JR Instructions
                when others =>
                        NPC <= Reg_Offset;              --Jumps to Target Address
                end case;
        end if;
end if;
end process;
end Behavioral;
```

D. MEM Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MM is
port    (
                clk,rst         : in STD_LOGIC;
                opcode                  : in STD_ULOGIC_VECTOR(5 downto 0);
                RA, Aluout      : in STD_ULOGIC_VECTOR(31 downto 0);
                Mem_read        : in STD_ULOGIC_VECTOR(31 downto 0);
                Mem_write       : out STD_ULOGIC_VECTOR(31 downto 0);
                MDR             : out STD_ULOGIC_VECTOR(31 downto 0);
                Mem_add         : out STD_ULOGIC_VECTOR(31 downto 0);
                sel             : out STD_LOGIC
                );
end MM;

architecture Behavioral of MM is
signal mem : STD_ULOGIC_VECTOR(31 downto 0);
begin
process(clk,rst)
begin
if(rst = '1') then
        null;
else
        mem <= RA;
        case opcode is
                when "100011" =>                                --LW
                        if(falling_edge(clk) and rst ='0') then
                                Mem_add <= Aluout;
                                MDR <= Mem_read;
```

```
                        sel <= '0';
                    end if;

                when "101011" =>                                --SW
                    if(rising_edge(clk) and rst ='0') then
                        Mem_add <= Aluout;
                        sel <= '1';
                        Mem_write <= mem;
                    end if;

                when others =>
                    null;
        end case;
end if;
end process;
end Behavioral;
```

E. WB Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity WB is
port(   clk,rst  : in STD_LOGIC;
        MDR,Aluout   : in STD_ULOGIC_VECTOR(31 downto 0);
        opcode          : in STD_ULOGIC_VECTOR(5 downto 0);
        Wrt_Val      : out STD_ULOGIC_VECTOR(31 downto 0)
   );
end WB;

architecture Behavioral of WB is

begin
process(clk,rst)
begin

if(rst = '1') then
        null;
else
        if(rising_edge(clk) and rst ='0') then
            case opcode is
                when "100011" => Wrt_Val <= MDR;
                when "101011" => null;
                when others   => Wrt_Val <= Aluout;
            end case;
```

```
        end if;
end if;
end process;
end Behavioral;
```

2. CACHE MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Cache is
   Port (              clk,rst                              : in STD_LOGIC;
                       I_cache_in,D_cache_in                : in
STD_ULOGIC_VECTOR(255 downto 0);
                       npc_in                                   : in
STD_ULOGIC_VECTOR(31 downto 0);
                       D_addr                                       : in
STD_ULOGIC_VECTOR(31 downto 0);
                       ldsw                                     : in STD_LOGIC; --
'0'=>Load, '1'=>Store
                       rdy_inst,rdy_data                    : in STD_LOGIC;
                       data_in                                  : in
STD_ULOGIC_VECTOR(31 downto 0);
                       I_cache_out,D_cache_out      : out STD_ULOGIC_VECTOR(31
downto 0);
                       I_hit,rd_inst                        : out STD_LOGIC;
                       iaddr_out                                : out
STD_ULOGIC_VECTOR(31 downto 0);
                       D_hit                                    : out STD_LOGIC;
                       rd_data,wt_data                  : out STD_LOGIC;
                       daddr_out                                : out
STD_ULOGIC_VECTOR(31 downto 0));
end Cache;

architecture Behavioral of Cache is

type MEM_I is array (511 downto 0) of STD_ULOGIC_VECTOR(7 downto 0 );
type MEM_D is array (255 downto 0) of STD_ULOGIC_VECTOR(7 downto 0 );
signal IC_MEM                                   : MEM_I := (OTHERS => "00000000");
signal DC_MEM                                   : MEM_D := (OTHERS => "00000000");
signal npc_int                  : integer ;
signal daddr_int                        : integer;
signal I_blk,D_blk              : integer;
signal IC_blk,DC_blk            : integer;
signal IC_temp,DC_temp          : integer;
```

```
signal go_read                                  : STD_LOGIC;
begin
npc_int <= to_integer(unsigned(npc_in));
daddr_int <= to_integer(unsigned(D_addr));
        D_blk <= daddr_int / 16;
        DC_blk <= D_blk mod 16;
        DC_temp <= DC_blk * 16;
-------------------------I-Cache process-----------
I_cache : process (clk, rst)
begin

if rst = '1' then
I_hit <= '0';
I_cache_out <= (OTHERS=> '0');
I_blk <= npc_int / 16;
IC_blk <= I_blk mod 32;
IC_temp <= IC_blk * 16;

else if rising_edge(clk) and rst = '0' then
-------------------------Inst Write Cycle---------------
if (IC_MEM(IC_temp + 0) & IC_MEM(IC_temp + 1) & IC_MEM(IC_temp + 2) &
IC_MEM(IC_temp + 3)) /= "00000000000000000000000000000000" then
        I_hit <= '1';
        rd_inst <='0';
else
        I_hit <= '0';
        rd_inst <= '1';
        iaddr_out <= npc_in;
if rdy_inst = '1' then
if I_cache_in(7 downto 0) /= "UUUUUUUU" then
for I in 0 to 15 loop
        IC_MEM(IC_temp + I) <= I_cache_in((8*(I+1)-1)downto(8*I));
end loop;
end if;
end if;
end if;
end if;
-------------------------Inst Read cycle-----------------
if falling_edge(clk) and rst = '0' then
        I_cache_out <= IC_MEM(IC_temp + 0) & IC_MEM(IC_temp + 1) &
IC_MEM(IC_temp + 2) & IC_MEM(IC_temp + 3);
end if;
end if;
end process;
-------------------------D-Cache process-------------------
D_cache : process (clk, rst)
```

```
begin

if rst = '1' then
        D_hit <= '0';
        rd_data <= '0';
        wt_data <= '0';
        D_cache_out <= "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
        go_read <= '0';

else
if rising_edge(clk) and rst = '0' then

if D_addr /= "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU" then
-------------------------Write Cycle-------------------------
if (DC_MEM(DC_temp + 0) & DC_MEM(DC_temp + 1) & DC_MEM(DC_temp + 2) &
DC_MEM(DC_temp + 3)) /= "00000000000000000000000000000000" then
        D_hit <= '1';
        go_read <= '1';
        rd_data <= '0';
        -------------------------Store Hit Operation-------------
        if ldsw = '1' then
        for k in 0 to 3 loop
                DC_MEM(DC_temp + k) <= data_in((8*(k+1)-1)downto(8*k));
        end loop;
        wt_data <= '1';
        daddr_out <= D_addr;
        end if;
else
        -------------------------Load Operation--------------------
        D_hit <= '0';
        if ldsw = '0' then
        rd_data <= '1';
        daddr_out <= D_addr;
        if rdy_data = '1' then
        if D_cache_in(7 downto 0) /= "UUUUUUUU" then
        for J in 0 to 15 loop
                DC_MEM(DC_temp + J) <= D_cache_in((8*(J+1)-1)downto(8*J));
                go_read <= '1';
        end loop;
        end if;
        end if;
        else
        -------------------------Store Miss Operation-------------
        if ldsw = '1' then
        wt_data <= '1';
        daddr_out <= D_addr;
```

```
        end if;
        end if;
end if;
end if;
end if;
-------------------------Read cycle---------------------------
if falling_edge(clk) and rst = '0' then
        if D_addr /= "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU" then
        --------------------------Read only for Load operation------
        if ldsw = '0' then
                        if go_read = '1' then
                D_cache_out <= DC_MEM(DC_temp + 0) & DC_MEM(DC_temp + 1) &
DC_MEM(DC_temp + 2) & DC_MEM(DC_temp + 3);
                        end if;
        end if;
end if;
end if;
end if;

end process;
end Behavioral;
```

3. BUS MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Bus_module is
Port (
                clk,rst  : in STD_LOGIC;
                idc_in          : in STD_ULOGIC_VECTOR (255 downto 0);
                irdy_in         : in STD_LOGIC;
                drdy_in                 : in STD_LOGIC;
                ir_in           : in STD_LOGIC;
                dr_in           : in STD_LOGIC;
                dw_in           : in STD_LOGIC;
                sdata_in        : in STD_ULOGIC_VECTOR (31 downto 0);
                iaddr_in        : in STD_ULOGIC_VECTOR (31 downto 0);
                daddr_in        : in STD_ULOGIC_VECTOR (31 downto 0);
                sdata_out       : out STD_ULOGIC_VECTOR (31 downto 0);
                idc_out         : out STD_ULOGIC_VECTOR (255 downto 0);
                irdy_out        : out STD_LOGIC;
                drdy_out        : out STD_LOGIC;
                ir_out          : out STD_LOGIC;
                dr_out          : out STD_LOGIC;
```

```
                dw_out                  : out STD_LOGIC;
                iaddr_out       : out STD_ULOGIC_VECTOR (31 downto 0);
                daddr_out       : out STD_ULOGIC_VECTOR (31 downto 0)
        );
end Bus_module;

architecture Behavioral of Bus_module is

signal clk_counter: integer := 1;
signal clk_value : integer;
signal clk_temp1, clk_temp2: integer;
begin

process (clk, rst)
begin

if rst = '1' then
        irdy_out <= '0';
        drdy_out <= '0';
        ir_out <= '0';
        dr_out <= '0';
        dw_out <= '0';

else
        if rising_edge(clk)  then

                idc_out <= idc_in;
                iaddr_out <= iaddr_in;
                daddr_out <= daddr_in;
                sdata_out <= sdata_in;

                if ir_in = '1' then
                        clk_counter <= clk_counter +1;
                        if clk_counter = 13 then
                                ir_out <= '1';
                                irdy_out <= '1';
                        end if;
                        if clk_counter = 15 then
                                ir_out <= '0';
                                irdy_out <= '0';
                                clk_counter <= 1;
                        end if;
                end if;

                if dr_in = '1' then
                        clk_counter <= clk_counter +1;
```

```
                        if clk_counter = 13 then
                                dr_out <= '1';
                                drdy_out <= '1';
                        end if;
                        if clk_counter = 15 then
                                dr_out <= '0';
                                drdy_out <= '0';
                                clk_counter <= 1;
                        end if;
                end if;

                if dw_in = '1' then
                        clk_counter <= clk_counter +1;
                        if clk_counter = 16 then
                                dw_out <= '1';
                        end if;
                        if clk_counter = 18 then
                                dw_out <= '0';
                        end if;
                end if;
        end if;
end if;
end process;
end Behavioral;
```

4. MEMORY MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Memory is
Port(   clk,rst  : in STD_LOGIC;
                r_inst          : in STD_LOGIC;
                inst_addr       : in STD_ULOGIC_VECTOR(31 downto 0);
                r_data          : in STD_LOGIC;
                w_data                  : in STD_LOGIC;
                data_addr       : in STD_ULOGIC_VECTOR(31 downto 0);
                mem_in                  : in STD_ULOGIC_VECTOR(31 downto 0);
                ins_rdy         : out STD_LOGIC;
                data_rdy        : out STD_LOGIC;
                mem_out         : out STD_ULOGIC_VECTOR(255 downto 0)
   );
end Memory;

architecture Behavioral of Memory is
```

```
type MEM is array (4095 downto 0) of STD_ULOGIC_VECTOR(7 downto 0);
signal L_MEM                    : MEM := (OTHERS => "00000000");
signal iaddr_int          : integer;
signal daddr_int          : integer;

begin
iaddr_int         <= to_integer(unsigned(inst_addr));
daddr_int         <= to_integer(unsigned(data_addr));

Memory : Process(clk, rst)
begin

if rst = '1' then
--lw $s1,100($t2)
          L_MEM (576) <= "10001101";
          L_MEM (577) <= "01010001";
          L_MEM (578) <= "00000000";
          L_MEM (579) <= "01100100";
--1 byte of Data to be read from memory
          L_MEM (208) <= "01110010";
--sw $t6,200($t2)
          L_MEM (672) <= "10101101";
          L_MEM (673) <= "01001110";
          L_MEM (674) <= "00000000";
          L_MEM (675) <= "11001000";
--add $s2,$t4,$s4
          L_MEM (16)  <= "00000010";
          L_MEM (17)  <= "01001100";
          L_MEM (18)  <= "10100000";
          L_MEM (19)  <= "00100000";
--beq $t5,$t1,200
          L_MEM(96)   <= "00010001";
          L_MEM(97)   <= "10101001";
          L_MEM(98)   <= "00000000";
          L_MEM(99)   <= "11001000";
--xori $t3,$s4,100
          L_MEM(192) <= "00111010";
          L_MEM(193) <= "10001011";
          L_MEM(194) <= "00000000";
          L_MEM(195) <= "01100100";
--j 400
          L_MEM(128) <= "00001000";
          L_MEM(129) <= "00000000";
          L_MEM(130) <= "00000001";
          L_MEM(131) <= "10010000";
--nor $s4,$t1,$t3
```

```
        L_MEM(800) <= "00000001";
        L_MEM(801) <= "00101011";
        L_MEM(802) <= "10100000";
        L_MEM(803) <= "00011011";
--sll $s1,$s1,10
        L_MEM(448) <= "00000000";
        L_MEM(449) <= "00010001";
        L_MEM(450) <= "10001010";
        L_MEM(451) <= "10000000";
--jr $s3
        L_MEM(320) <= "00010110";
        L_MEM(321) <= "01100000";
        L_MEM(322) <= "00000000";
        L_MEM(323) <= "00000000";
end if;

if w_data = '1' then
        ins_rdy <= '0';
        data_rdy <= '0';
        for i in 0 to 3 loop
                L_MEM(daddr_int+i) <= mem_in((8*(i+1)-1)downto(8*i));
        end loop;

else
        if r_inst = '1' then
                ins_rdy <= '1';
                data_rdy <= '0';
                mem_out <= L_MEM(iaddr_int +15) & L_MEM(iaddr_int +14) &
L_MEM(iaddr_int +13) & L_MEM(iaddr_int +12) & L_MEM(iaddr_int +11) &
L_MEM(iaddr_int +10) & L_MEM(iaddr_int +9) & L_MEM(iaddr_int +8) &
L_MEM(iaddr_int +7) & L_MEM(iaddr_int +6) & L_MEM(iaddr_int +5) & L_MEM(iaddr_int
+4) & L_MEM(iaddr_int +3) & L_MEM(iaddr_int +2) & L_MEM(iaddr_int +1) &
L_MEM(iaddr_int +0) ;

        else
                if r_data = '1' then
                        ins_rdy <= '0';
                        data_rdy <= '1';
                        mem_out <= L_MEM(daddr_int +15) & L_MEM(daddr_int +14) &
L_MEM(daddr_int +13) & L_MEM(daddr_int +12) & L_MEM(daddr_int +11) &
L_MEM(daddr_int +10) & L_MEM(daddr_int +9) & L_MEM(daddr_int +8) &
L_MEM(daddr_int +7) & L_MEM(daddr_int +6) & L_MEM(daddr_int +5) &
L_MEM(daddr_int +4) & L_MEM(daddr_int +3) & L_MEM(daddr_int +2) &
L_MEM(daddr_int +1) & L_MEM(daddr_int +0) ;
                end if;
        end if;
```

```
end if;
end process;
end Behavioral;
```

5. TOP MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity System is

end System;

architecture Behavioral of System is
component CPU is
port(    clk,rst                        : in STD_LOGIC;
         PC,IR,Mem_read                 : in STD_ULOGIC_VECTOR(31 downto
0);
         Mem_add,Mem_write,NPC   : out STD_ULOGIC_VECTOR(31 downto 0);
         Sel                            : out STD_LOGIC
     );
end Component;

component Cache is
   Port (        clk,rst                                  : in STD_LOGIC;
                 I_cache_in,D_cache_in        : in STD_ULOGIC_VECTOR(255
downto 0);
                 npc_in                                   : in
STD_ULOGIC_VECTOR(31 downto 0);
                 D_addr                                   : in
STD_ULOGIC_VECTOR(31 downto 0);
                 ldsw                                     : in
STD_LOGIC;                                       --'0'=>Load, '1'=>Store
                 rdy_inst,rdy_data            : in STD_LOGIC;
                 data_in                                  : in
STD_ULOGIC_VECTOR(31 downto 0);

                 I_cache_out,D_cache_out : out STD_ULOGIC_VECTOR(31
downto 0);
                 I_hit,rd_inst                : out STD_LOGIC;
                 iaddr_out                    : out
STD_ULOGIC_VECTOR(31 downto 0);
                 D_hit                        : out STD_LOGIC;
                 rd_data,wt_data              : out STD_LOGIC;
```

```
                                daddr_out                                : out
STD_ULOGIC_VECTOR(31 downto 0));
end component;

component Bus_module is
Port (
        clk,rst             : in STD_LOGIC;
        idc_in              : in STD_ULOGIC_VECTOR (255 downto 0);
    irdy_in             : in STD_LOGIC;
    drdy_in             : in STD_LOGIC;
    ir_in               : in STD_LOGIC;
    dr_in               : in STD_LOGIC;
    dw_in               : in STD_LOGIC;
        sdata_in        : in STD_ULOGIC_VECTOR (31 downto 0);
    iaddr_in    : in STD_ULOGIC_VECTOR (31 downto 0);
    daddr_in    : in STD_ULOGIC_VECTOR (31 downto 0);

        sdata_out       : out STD_ULOGIC_VECTOR (31 downto 0);
    idc_out             : out STD_ULOGIC_VECTOR (255 downto 0);
    irdy_out    : out STD_LOGIC;
    drdy_out    : out STD_LOGIC;
    ir_out              : out STD_LOGIC;
    dr_out              : out STD_LOGIC;
    dw_out              : out STD_LOGIC;
    iaddr_out   : out STD_ULOGIC_VECTOR (31 downto 0);
    daddr_out   : out STD_ULOGIC_VECTOR (31 downto 0)
        );
end Component;

component Memory is
Port(
                clk,rst         : in STD_LOGIC;
                r_inst          : in STD_LOGIC;
                inst_addr       : in STD_ULOGIC_VECTOR(31 downto 0);
                r_data          : in STD_LOGIC;
                w_data                  : in STD_LOGIC;
                data_addr       : in STD_ULOGIC_VECTOR(31 downto 0);
                mem_in                  : in STD_ULOGIC_VECTOR(31 downto 0);

                ins_rdy                 : out STD_LOGIC;
                data_rdy        : out STD_LOGIC;
                mem_out                 : out STD_ULOGIC_VECTOR(255 downto 0)
  );
end Component;
```

```
signal pc4, m_addr, m_write, m_write_out, inst_in, data_in : STD_ULOGIC_VECTOR(31
downto 0);
signal sel, ir_in, dr_in, dw_in, ir_out, dr_out, dw_out, irdy_in, drdy_in, irdy_out, drdy_out :
STD_LOGIC;
signal iaddr_in, iaddr_out, daddr_in, daddr_out : STD_ULOGIC_VECTOR(31 downto 0);
signal idc_in, idc_out : STD_ULOGIC_VECTOR(255 downto 0);

begin
CPU_Module : CPU port map( clk,rst,pc4, inst_in, data_in, m_addr, m_write, new_pc, sel);
CACHE_Module : Cache port map( clk, rst, idc_out, idc_out, pc, m_addr, sel, irdy_out,
drdy_out, m_write, inst_in, data_in, ihit, ir_in, iaddr_in, dhit, dr_in, dw_in, daddr_in);
BUS_Module : Bus_module port map (clk, rst, idc_in, irdy_in, drdy_in, ir_in, dr_in, dw_in,
m_write, m_write_out,iaddr_in, daddr_in, idc_out, irdy_out, drdy_out, ir_out, dr_out, dw_out,
iaddr_out, daddr_out);
MEMORY_Module : Memory port map (clk, rst, ir_out, iaddr_out, dr_out, dw_out, daddr_out,
m_write_out, irdy_in, drdy_in, idc_in);

end Behavioral;
```