CS230 Project Report

IITB-RISC-22 (pipelined processor)

Dhvanit Beniwal, 200050035
Margav Mukeshbhai Savsani 200050072
Pasala Poorna Teja, 200050101
Sartaj Islam, 200050128

DESCRIPTION:

We have designed a pipelined processor, IITB-RISC-22, with the given instruction set architecture. There are 8 (16 bit each) registers. There is a memory block of 2^{16} memory elements of 16 bit each, one for every memory address implemented as an entity of its own.

The top level entity, **Top_entity**, has a clock and reset input, and it assembles all the components of the processor in its architecture. Each individual component has its own vhdl file including one for the memory, the ALU, register file etc. There are specific components for each of the five stages in our pipeline, (fetch, decode, execute, memory-access, write-back), and some more for implementing forwarding and branch prediction techniques.

BASIC WORKING:

The instructions are executed in a 5-stage pipelined manner, each stage 'mostly' implemented in a vhdl-component of appropriate name:

- **InstructionFetch.vhd**
- **InstructionDecode.vhd**
- **InstructionExecute.vhd**
- **MemoryAccess.vhd**
- **WriteBack.vhd**

The branch prediction table block is maintained and implemented in **Next_instruction**.vhd. As the name suggests, it handles the next instruction determination.

Throughout the processor, **stall** signals have been implemented when the instruction is to be stalled and this is handled from **Stall_Control.vhd**. There are stall signals for fetch, decode and execute (IF/ID/EX).

InstructionFetch.vhd looks at the instruction and incase we have an LM or SM instruction, it passes it on to **FetchEncoder.vhd** so that it can be converted to an equivalent series of **LW** or **SW** instructions. Also branch prediction is used if possible.

InstructionDecode.vhd decodes the type of instruction (ADD/NDU/SW/BEQ etc) and also reads the registers. The comments in the vhd file above a particular block of code (like "--BEQ") indicate which instruction the following read/write signals refer to.

InstructionExecute.vhd starts the execution, invoking the ALU when necessary. Similar comments in the vhd file (like "--NDZ") mark the block of code doing the necessary calculation.

(All of this is with taking care of the aforementioned **stall** signals)

The file **Execute_Control.vhd** makes sure that for registers any read operation does not overlap with a write operation, i.e. they are carried out separately. **Execute_Control_cz.vhd** does this for the carry and zero bits.

MemoryAccess.vhd is the 4th stage of the pipeline which prepares data that is to be read/written in the memory. The actual read/write operation happens in **Memory.vhd** itself as described later.

WriteBack.vhd is the final stage where we update the program counter, carry bit, zero bit, and write back (if necessary) the required data into the respective register.

Other major components:

Memory.vhd implements a memory as an array of 16 bit signals (2^{16} of them). It receives a memory address, Data and **M_write** bit as inputs. It outputs the data in that memory address as **M_out** (should it be needed). If the **M_write** bit is **1** it writes the value **Data** into the memory at the given address. This way there are no separate processes for read and write.

ALU.vhd needs a 2 bit **ALU_control** that selects what it has to do. **00** means addition, **01** means subtraction, **10** means bitwise NAND.

Some other minor components (file description)

ADD.vhd

This file adds two 16 bit inputs A and B along with a single bit carry input cin and stores the sum in S and the carry output in cout. This uses ripple carry adder to add the signals with the help of ADD_4bit which adds 4 bit numbers in the same fashion.

ALU.vhd

This file performs the alu operation on inputs ALU_A and ALU_B based on the input signal from ALU_control and stores the result in ALU_C and also sets the carry flag 'c' and zero flag 'z'.

left_shift.vhd

This entity does the left shift operation on the 16 bit input RC and stores the output in the 16 bit output port B.

carry_Reg.vhd

This file has 2 input ports, one is single bit ALU_carry and the other is carry_write. According to the signal carry_write the output signal is changed to ALU_carry.

zero_Reg.vhd

This file has 2 input ports, one is single bit ALU_zero and the other is zero_write. According to the signal zero_write the output signal is changed to ALU_zero.

Memory.vhd

This file has an array of Memory which can be accessed through this file only. Read and write actions are performed on memory through this file. It takes a 16 bit index A1 as input and returns the value in M_out. If M_write signal is set then it write the 16 bit Data input to the memory at index A1.

pc.vhd

This file has 2 input ports, one is 16 bit pcin and the other is pc_write. According to the signal pc_write the 16 bit output signal is changed to pcin.

Register_File.vhd

This file has an array of Registers of size 8. This takes 3 bit inputs A1 and A2 which are indices for the Register and the values at these indices are stored in 16 bits output ports D1 and D2 respectively. Also if RF_write signal is set then the value D3 is stored in the register at index A3, both taken as input.

Sign_extend.vhd

This entity performs the sign extension of the input port ins_8to0 and stores the result. Further based on the imm6 input it decides the number of bits by which sign has to be extended.

Top_entity.vhd

This file creates the components of all the entities and maps the signals defined here. This creates a connection between all the ports in the entities and at each cycle our program functions all the required actions defined.