MIPS Pipelined Processor

Haoran Du
260776911
McGill University
Montreal, Canada

Carlo D'Angelo
260803454
McGill University
Montreal, Canada

Edem Nuviadenu
260779440
McGill University
Montreal, Canada

Nathanael Lemma
260779759
McGill University
Montreal, Canada

***Abstract*—This paper details our project group's design process and implementation of a standard pipelined 32-bit MIPS processor in VHDL. It explains our implementation of the major components in detail, including how we handled data detection and forwarding. This design was split into the following 5 stages: Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back. Our testing and integration of these stages are described in depth in this report with an evaluation of such an optimized processor.**

I. Introduction

The purpose of this project, as described in the provided project specification file, was to implement a standard five-stage pipelined 32-bit MIPS processor in VHDL. This processor is implemented using a basic 5-stage pipeline, namely the IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access), and WB (Write Back) stages, to execute a subset of the MIPS instruction set. The block diagram for these five stages can be seen in Fig. 1. An additional objective was to implement various optimizations and assess their effects on the performance of the pipelined processor. This report describes our design, implementation and evaluation of such a processor, and a brief overview of each stage will be provided in the following sections.

II. Methodology

A. IF (Instruction Fetch) Stage

The IF stage consists of the program counter (PC), the instruction memory, an adder, and a multiplexer (MUX). Program execution begins in this stage with the instruction memory first reading from an ASCII text file that contains lines of 32-bit words encoded in binary. This text file is the output of an assembler that has converted MIPS assembly into machine code. The 32-bit words each represent the different instructions of the program and, thus, VHDL code initializes instruction memory with these instructions. Since memory is byte-addressable in this context, these instructions are divided into four parts before being put in instruction memory. Based on the address in the PC, an instruction is loaded from instruction memory and passed to the ID stage for decoding in the next clock cycle. Next, the adder and the MUX work together to update the PC to the address of the next correct instruction and maintain program flow. The 2:1 MUX selects between the adder output (i.e., PC + 4) and a potential decoded branch address. Note that each instruction is 32-bits in this byte addressing MIPS, so the adder adds four to the PC value to fetch the next instruction when a branch is not taken.

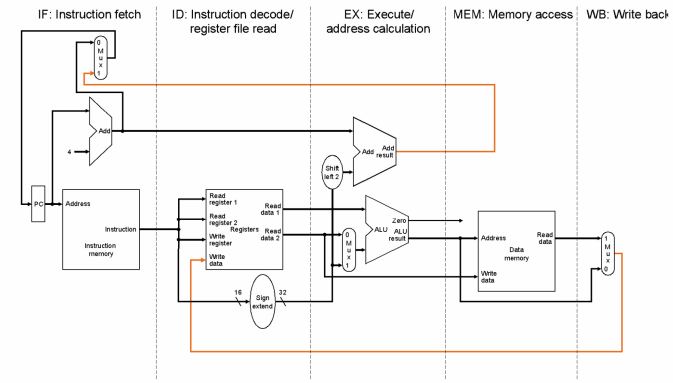


Fig. 1: Block diagram showing the 5 stages of the MIPS pipelined processor. Datapaths for write back are highlighted in orange.

B. ID (Instruction Decode) Stage

The ID stage is mainly composed of the instruction decoder and the registers module. The decoder would interpret the fetched instruction from IF and produce the register address for the purpose of register referencing and data outputs.

If branches exist in the program, branch instructions are also resolved in this stage. The decoder sends the target address of branch instructions to decide if the program must branch. If the program is required to

branch, the target address is then added to the current instruction address. Next, as stated in the IF stage, PC will also be updated accordingly. There is also a sign extender to extend the immediate values from 16 bits to 32 bits in this stage. Hazard detection was also handled here but will be discussed later in section G of the report.

Primarily, 3 types of instructions were decoded. R-type instructions, J-Type instructions and I-type instructions. The R-type instructions have a the basic format of the opcode, rs (first source address), rt (second source address), rd (destination register), shamt (for shift instructions), funct. I type instructions have opcode, rs, rt and immediate value. Finally J type instructions have opcode, and address. We parse these instruction types accordingly in the decode phase. An additional component to this phase is the register file write and read. Given that the decode component sends register data to the execute phase, a register block component was created to handle all data reads and writes to and from register addresses. In decoding R-type instructions, the respective data form the source registers are obtained and passed on to the execute phase. The Decode entity can be seen in the Appendix. The core logic of decode uses a switch case statements to assign the respective unique ALU opcodes once instructions have been deciphered via the MIPS reference card; an example of this is shown below.

```
  if op_temp = "000000" then --Rtype
      --NB: Instruction is in descending order
        rt_temp := to_integer(unsigned(
    instruction(20 downto 16)));
        rd_temp := to_integer(unsigned(
    instruction(15 downto 11)));
        shamt_temp := instruction(10 downto 6)
    ;
        funct_temp := instruction(5 downto 0);

        rd_delay1 <= rd_temp;
        rd_delay2 <= rd_delay1;
        case(funct_temp) is
          when "100000" =>
            -- add
            alu_opcode <= "00000";
...
```

Listing 1: Instruction Decode

The register block component in our design was designed to perform reads from register file, in lieu of rising edge, and writes on the falling edge. There are 32 registers with 2^5 addressing depth for the registers. Writing data to a register location only occurs when the register write signal is high and write address is not equal to zero. This is because R0 is always set to zero. Write to "register_file.txt" occurs once the program has completed. The corresponding code snippet is shown below.

```
...
IF(now > 9999999 ps)THEN
      file_open(write_file, "register_file.txt
    ", write_mode);
      For i in 0 to reg_size-1 LOOP
        write_data := reg_block(i);
        write(write_line, write_data);
        writeline(write_file, write_line);
      END LOOP;
      file_close(write_file);
    end if;
  end process;
  reg_block(write_address) <= write_data when
    (reg_write = '1' and write_address /=0);
  data_out1 <= reg_block(read_address1); --
    read data in rs register location
  data_out2 <= reg_block(read_address2); --
    read data in rt register location
...
```

Listing 2: Register Block Core Logic

C. EX (Execute) Stage

The EX stage consists of three major components a 2:1 MUX, an adder, and the ALU. The ALU is of the greatest significance at this stage due to the fact that it performs the arithmetic operations necessary to execute the instructions. The ALU op-code determines whether the I type or the R type instructions would be completed. In general this component has a capability of executing arithmetic, Logical, transfer, shift, memory and control flow instructions. In addition, at this stage of the pipeline, in order to do branch resolution, the next value of the program counter is calculated if the branch is taken.

The 2:1 MUX is used to feed the ALU register an immediate value based on the decoded instruction. It is controlled by the decoder. On the other hand, the adder is used to shift left an input from the instruction fetch (IF) stage by two. Finally, the ALU is connected to the MEM and WB stage. Below we discuss how the ALU was implemented. The corresponding op-code values for each instruction can also be found in the table below.

The arithmetic operations our ALU supports with their mnemonics are: add, sub, addi, mult, div, slt, and slti. For mult and div operations we used the hi and lo signal that we got from the MIPS reference. During multiplication, we used 64 instead of 32 for the product vector size. After the product was obtained hi is assigned the the last 32 bits of the product vector (63 down to 32) and the lo is assigned the first 32 bits of the product vector

(31 down to 0). Similarly for division, after the quotient and remainder are obtained, hi is assigned the value of the remainder whereas lo is assigned the value of the quotient.

The logical operations our ALU supports with their mnemonics are: and, or, nor, xor, andi, ori, and xori. And for transfer operations we have: mfhi, mflo, and lui. For mfhi we basically assigned the value of of hi to the output of the ALU. Similarly, in the case of mflo the value of lo is assigned to the output. For shift operations the instructions our ALU supports are: sll, srl and sra. As well as, lw and sw for memory operations and finally, beq, bne, j, jr, and jal for control-flow instructions.

Instruction	Type	ALU opcode
add	R	00000
and	R	00001
div	R	00010
nor	R	00011
or	R	00100
slt	R	00101
sub	R	00110
xor	R	00111
mult	R	01000
mfhi	R	01001
mflo	R	01010
sra	R	01011
sll	R	01100
srl	R	01101
jr	R	01110
j	J	01111
jal	J	10000
addi	I	10001
andi	I	10010
ori	I	10011
xori	I	10100
lw	I	10101
lui	I	10110
sw	I	10111
slti	I	11000
beq	I	11001
bne	I	11010

D. MEM (Memory Access) Stage

The MEM stage is made up of the main memory block, which can either retrieve data after receiving a load instruction or write the referenced memory address on store instructions. Furthermore, the 2:1 MUX output in the WB stage, controlled by the ALU, controls the output of the MEM stage.

In implementing the memory stage, we designed 3 separate components, namely Instruction Memory, Data Memory, and 'Memory.vhd'. Studying the instructions from the project specification, the team wrote the code based on the memory model provided for the Cache project in this course, which is Adapted from Example 12-15 of Quartus Design and Synthesis handbook.

The data memory is sized at 32768 bytes as requested, and we initialize the data memory to all zeros by default.

In addition, after defining the initialization of the SRAM in simulation, we designed our read and write request to not be happening at the time. Moreover, our 'waitrequest' signal is used to vary the response time in simulation.

```
waitreq_w_proc: PROCESS (memwrite)
BEGIN
  IF(memwrite'event AND memwrite = '1')THEN
    write_waitreq_reg <= '0' after mem_delay
  , '1' after mem_delay + clock_period;

  END IF;
END PROCESS;
```

Listing 3: waitrequest signal for memory write

As stated above, the team separate memories for instructions and data into two VHDL files. In our 'Data_Memory.vhd', the handling of the SRAM in simulation and read/write wait request are very similar to the 'Memory.vhd'. In addition, if it is a 'memread', then the data from the 'ram_block(address+N)' will be assigned to the respective address in 'fetch_instr'. Similarly, if it is a 'memwrite', the respective block of 'writedata' will be assigned to the 'ram_block(address+N)'. A snippet of our implementation is shown below in Listing. 4.

```
      IF (memread = '1') THEN
        fetch_instr(7 downto 0) <= ram_block(
   address);
        fetch_instr(15 downto 8) <= ram_block(
   address + 1);
...
      ELSIF (memwrite = '1') THEN
        ram_block(address) <= writedata(7
   DOWNTO 0);
        ram_block(address+1) <= writedata(15
   downto 8);
...
```

Listing 4: read/write handling for Data Memory

E. Write_Back

The last stage of the processor is the WB stage. Here we forward the resulting data of an instruction to the required location such as the register file and

the EX stage (for data forwarding). For instance, if a load instruction is issued, data from the correct memory location would be selected and sent to the register file for write-back.

F. Pipelined_Processor

In this stage, we first initialised all the components we created ie Fetch, Decode, Execute, Instruction Memory, Data Memory, Memory, and Write Back. After, we specified corresponding signals in the architecture of the pipelined processor and connected these signals appropriately to the various components. Since our Register_Block component was connected to the Decode phase and the Data memory component handled memory writes, the register values were written to 'register_file.txt' and data memory written to 'memory.txt' as required. We used the avalon interface

G. Hazard Detection & Forwarding

1) Hazard Detection: This pipelined processor supports hazard detection and it is implemented in the ID stage. The presence of potential data hazards and control hazards requires the ID stage to have different behaviours depending on the situation. To account for this change in behaviour, we use 5 states to describe its functionality as can be seen in Listing. 5. State *operating* deals with the normal operation of the ID stage when there are no hazards. It is also in this state that hazards are actually detected.

In particular, data hazards are detected with the help of the two signals *rd_delay1* and *rd_delay2*. Since the values written to the register file in the WB stage are available to read in the ID stage in the same clock cycle, the maximum delay caused by a potential data hazard is two clock cycles. Therefore, we must keep track of the previous two destination registers to ensure that no source registers match and cause a data hazard. As for control hazards, they are detected when an instruction is decoded and found to be a control-flow related instruction such as Branch On Equal and Jump. Since branches are resolved in the EX stage, there are always 2 stalls when a branch instruction is encountered.

States *one_stall* and *two_stall* describe the behaviour of ID when a data hazard causes one stall or two stalls respectively. Likewise, states *control_stall_first* and *control_stall_second* deal with the first and second stall caused by a control hazard.

```
...
-- hazard detection
signal rd_delay1: INTEGER RANGE 0 TO reg_size
    -1;
signal rd_delay2: INTEGER RANGE 0 TO reg_size
    -1;
type states is (operating,one_stall,two_stall,
    control_stall_first,control_stall_second);
signal current_state : states := operating;
...
```

Listing 5: Hazard Detection

2) Data Forwarding: Data forwarding was not completed. We specified our entity declarations and described our initial design logic. We decided we would implement a forwarding unit as a component attached in the Execute stage. As inputs to the forwarding unit, we would have a destination register from the EX/MEM phase, the destination register from the MEM/WB phase, the rs (src1 register), rt (src2 register) from decode phase and outputs from the forwarding unit would be signals "forwarded_inputX" and "forwarded_inputY" as forwarded outputs to the ALU. The idea was that if EX/MEM destination register is equal to either rs or rt then forwarding will be required. We would then use the value that the ALU just produced in the last cycle as the input to the ALU in the current cycle. Alternatively if the MEM/WB destination register equal to rs or rt to the forwarding unit, we would use the value from the MUX in the WB phase as one input to the ALU.

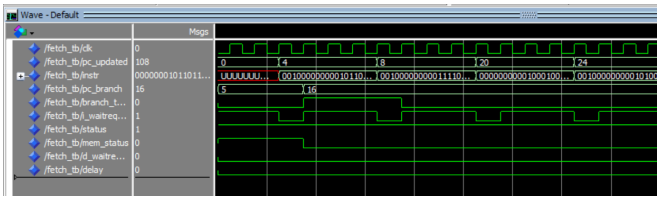
III. Results and Discussion

A. Fetch Intermediate results

To prove that the Fetch component delivers accurate intermediate results, a testbench is written for it. In the testbench, there are four test cases in total and code that reads from a sample program.txt file. These test cases make sure that the PC updates properly, whether there is a branch or not, and that the correct instruction is fetched from the instruction memory. Fig.2 shows that all these test cases pass. More details can found in the comments of the Fetch_tb.vhd file.

B. Decode Intermediate results

A test bench was written as part of a validation of the decode component. The I-type (addi $30, $0, 4) instruction was tested in our Decode_tb file. This test begins by assigning a fetch waitrequest signal as low, then waiting for a clock period after which we set the fetch waitrequest to high in between clock period waits. As a validation that decode component functions, our intermediate results asserts that the alu_opcode is obtained and verified to be "10001", the read data output to the Execute stage should be 0, and our extended immediate



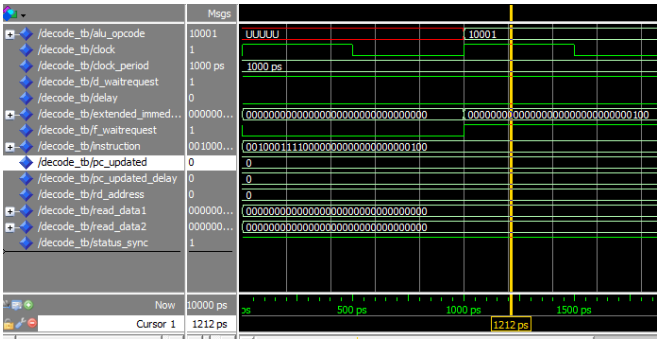
(a)

```
VSIM 46> run
# ** Note: Test case 1
#    Time: 0 ps  Iteration: 0  Instance: /fetch_tb
# ** Note: Test case 2
#    Time: 3500 ps  Iteration: 2  Instance: /fetch_tb
# ** Note: Test case 3
#    Time: 7500 ps  Iteration: 2  Instance: /fetch_tb
# ** Note: Test case 4
#    Time: 11500 ps  Iteration: 2  Instance: /fetch_tb
```

(b)

Fig. 2: (a)ModelSim test (Fetch) (b) ModelSim command line result (Fetch)

value should be 4. The corresponding testbench results, compiled in ModelSim are shown in Fig. 3.



(a)

```
VSIM 19> run
# ** Note: TEST 1: I-type instruction (addi $30, $0, 4)
#    Time: 0 ps  Iteration: 0  Instance: /decode_tb

VSIM 20>
```

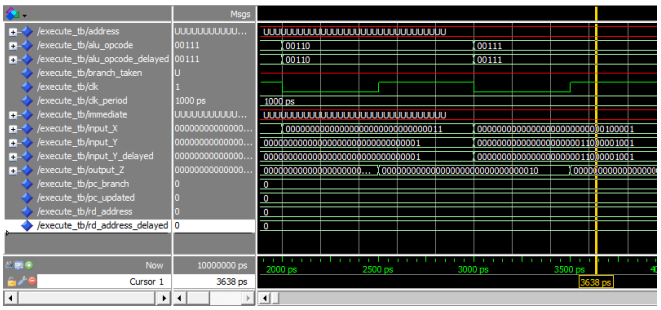
(b)

Fig. 3: (a)ModelSim test (Decode) (b) ModelSim command line result (Decode)

C. Execute Intermediate results

We have written an intermediary test bench to test the functionality of the execute stage of our pipelined processor. Out of the 27 supported instruction types by our ALU we have selected seven common operations for our testbench namely, add, sub, xor, or, nor, jr and sw. The test bench can be found in the Execute_tb file.

For each of the tests, we start with assigning values to the inputs of the ALU. The opcode is also passed as an input to specify what the ALU should do and what to be tested on. And then we assert if the output of the ALU corresponds to the expected output of the operation performed. The corresponding testbench results, compiled in ModelSim are shown in Fig. 4.



(a)

```
/SIM 39> run
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /execute_tb/dut
# ** Note: Test case 1
#    Time: 1 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 2
#    Time: 2 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 3
#    Time: 3 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 4
#    Time: 4 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 5
#    Time: 5 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 6
#    Time: 6 ns  Iteration: 0  Instance: /execute_tb
# ** Note: Test case 7
#    Time: 7 ns  Iteration: 0  Instance: /execute_tb
```

(b)

Fig. 4: (a)ModelSim test (Execute) (b) ModelSim command line result (Execute)

D. Memory Intermediate results

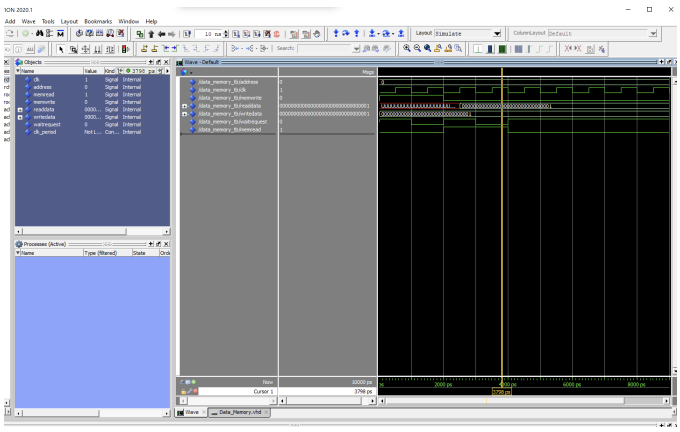


Fig. 5: ModelSim simulation result for the Data Memory testbench.

An intermediary testbench was presented to test the functionality of the MEM stage of our pipelined processor, more precisely the data memory. Since the testing of the Memory.vhd is the same as Data_Memory.vhd, that

is a test case for write which includes read test as well, only the Data_Memory_tb.vhd is presented in our code. It's simulation result can be referenced in Fig. 5.

In Data_Memory_tb.vhd, after the ports, entities, and the signals are defined, a test case for memory write is written. In this test case, after a random address is assigned, the flag signal for memory write is set to 1, indicating that it is the write request. Then, a random 32 bit writedata is issued. To validate the result, we then revert the flags of read and write by reading the content in the above memory address. If the results do not match, a "WRITE RES ERROR" severity ERROR would be reported.

E. Pipelined_Processor Complete results

This test runs our entire pipeline. The corresponding results can be seen via running the Testbench.tcl file and looking at the output memory.txt and register_file.txt files. These results do not match with the benchmark files.

IV. Limitations

Overall, hazard detection and forwarding did not work correctly. As such our results do not match with the benchmark yet. All other components when tested on their own (unit tests) worked fine. Our intermediate results which show how the various components functioned are discussed in section (III).

A. Brief Discussion

Hazard detection, although implemented, could not be fully tested to ensure that the appropriate stalls were being set. Consequently, we could not fully implement the forwarding unit as there is a need to perform the respective validation checks of the register destination values from the EX/ME phase and the ME/WB phase with the source registers from the decode phase. Section II(G) of this report describes how we planned to approach forwarding.

V. Conclusion

This project allowed us to use knowledge learnt throughout the semester to design and implement a five-stage pipelined 32-bit MIPS processor in VHDL. In terms of key takeaways from this project, it was first important to breakdown the pipeline stages into different components, as indicated in the report. This allowed us to separate concerns in the pipeline, and facilitated unit testing of the various components. It was further important that we understood what the respective signal inputs and outputs were to the various component entities to ensure that all components of the pipeline were connected appropriately. Hazard detection also plays an important role in ensuring that instructions get executed correctly and the complexity of detecting hazards increases rapidly as the number of instructions increases. Through this project we have come to appreciate the importance of the pipelined architecture in increasing overall instruction throughput.

VI. Appendix

```
entity Decode is
  port(
    clock: in std_logic;
    -- synchronie with instruction memory (or
    data memory if hazard)
    f_waitrequest: in std_logic;
    d_waitrequest: in std_logic;
    -- instruction from IF stage
    instruction: in std_logic_vector (31
    downto 0);
    -- PC + 4 from IF stage
    pc_updated : in integer range 0 to
    instr_mem_size-1;
    -- ID sends PC + 4 to EX stage for branch
    resolution
    pc_updated_delay : out integer range 0 to
    instr_mem_size-1;
    -- decoded values from instruction to send
     to EX stage
    read_data1 : out std_logic_vector(31
    downto 0);
    read_data2 : out std_logic_vector(31
    downto 0);
    extended_immediate : out std_logic_vector
    (31 downto 0);
    address : out std_logic_vector (31 downto
    0);
    -- a code specifying what EX stage needs
    to do, based on the opcode of the
    instruction
    alu_opcode : out std_logic_vector (4
    downto 0);
    -- destination register address for WB
    stage
    rd_address: out INTEGER RANGE 0 TO
    reg_size -1;
    -- send to IF stage to stall (hazard
    detection)
    delay: out std_logic := '0'
    );
end Decode;
...
```

Listing 6: Decode Entity Declaration