CMPE 260 Laboratory Exercise 3
Instruction Fetch and Decode Stages

Glenn Vodra
Performed: February 16, 2023
Submitted: March 2, 2023

Lab Section: 3
Instructor Sutradhar
TA: Ian Chasse
Cole Johnson
Robbie Riviere
Jonathan Russo
Colin Vo

Lecture Section: 1
Professor Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further, acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Abstract

The purpose of this exercise was to design the first two stages of a pipelined MIPS processor. These are the instruction fetch and instruction decode stages. The first of these, fetch, reads from instruction memory by concatenating four addresses, and updating the program counter. Each instruction is thirty-two bits, and each memory address addresses one byte. The second stage, decode, takes this instruction and generates control signals, as well as reads from any registers that are needed. In addition, there is also a sign extend in this stage, that extends a potential immediate value to thirty-two bits. Register data retrieved from one of the thirty-two registers, during this stage, is also thirty-two bits, however, it does not need to be sign extended. All of these signals are then passed out of this stage, to an execute stage. These two stages, fetch, and decode were implemented and tested with VHDL, using simulation software. These two designed stages produced correct results when tested with a test bench, resulting in a successful exercise.

Design Methodology

A key aspect of the fetch stage is the memory design. The design created consisted of a two-dimensional array, the first holding address space, and the second, holding address data. The memory was byte addressable and held a total of one thousand twenty-four bytes. Since each instruction in MIPS is thirty-two bits, the address being read needs to be concatenated with the following three addresses. When checking the address being read, the design will ensure that none of the following four addresses are out of bounds. If the address signal points to a word of memory that is fully, or partly out of bounds, the design will return a word of zeros. The address signal is twenty-eight bits, however during testing, as will be explained in the results and analysis section, the amount of memory was significantly reduced to ensure the proper functionality of the over-read protection.

This memory design is only one aspect of the instruction fetch stage. The other aspect is the program counter (PC), updater. Each rising clock edge, the PC will get the value of four added to it. This is because every instruction's starting memory address is four away from the last instruction. The program counter updater also has a reset that can be asynchronously triggered to restart the program from the first instruction. In this part of this design, the memory's instruction signal was also passed through, so it can be accessed by the next stage of the processor.

The next stage of the processor is the decode stage. This stage sets most of the control signals used to determine what hardware aspects of the execute and memory/writeback stages will be used. The control unit creates the following six control signals. These are, RegWrite, MemToReg, MemWrite, ALUControl, ALUSrc, and RegDst. Each of these signals are assigned based on the Opcode, found in bits thirty-one through twenty-six of the instruction, and sometimes the function field, in the case of a register type(R-type) instruction, using bits five through zero. Table one shows how some of the control signals are assigned, based on the type of data read from the two instruction fields.

Table 1: *Control Signal Assignment for MIPS Instructions*

	RegDest	ALUSrc	MemWrite	RegWrite	MemToReg	ALUControl
R-Type	1	0	0	1	0	Depends
I-Type	0	1	0	1	0	Depends
SW	X	1	1	0	0	0100
LW	0	1	0	1	1	0100

In table one, many of the important cases are covered, and a case that shows the purpose of each control signal is also present. In the VHDL design, each of these signals was assigned in unique processes to make debugging easier, and avoid multi-driven signal errors. For any signals in this table that are, do not care, they are set to zero when implemented. The ALU control signal assignment was assigned based on the function field for R-type instructions, but based on the Opcode for the I-type. This is because the area that would hold a "function" field in an I-type is already reserved for use by the immediate. The immediate value, in this case, could be any sixteen-bit number.

On the subject of the immediate value, bits fifteen through zero, from the instruction are assumed to be the immediate value. If an immediate is not needed for the current instruction, it can be discarded by the ALU source control signal as part of the execute stage. In order to convert the sixteen-bit number to thirty-two bits, it is signed extended. This sign extension is done by taking the last bit of the number and using it to fill the leading sixteen bits.

The final aspect of the decode stage is the register file. The MIPS design has thirty-two registers, two of which can be read asynchronously from, and one that is written to on the falling clock edge. During this stage of the processor, no data is written to the register file. During testing, data was written to ensure the proper functionality of the read. The address of the two registers that are read, are decoded from the instruction. Bits twenty-five through twenty-one are used as the first address and bits twenty through sixteen are used for the second. In MIPS these two addresses are RS and RT. A copy of RT and RD, found in bits fifteen through eleven of the instruction, are passed through the decode stage. These two signals will be mux selected during the execute stage, to pick the register that data will be written to.

Results and Analysis

To test the two designs, instruction fetch and decode, a self-checking test bench was designed and simulated for each. To test the fetch's full capabilities, a few things needed to change with the design. First, instead of initializing the instruction memory to all zeros, it needed to be filled with some "instruction" data. In this test, the data was set to random chunks of numbers. In a final implementation, it would hold real encoded MIPS instructions. The other design change that was implemented for testing was a smaller cap on the memory space. To ensure that accessing data part way, or fully past the memory limit returns null data, the memory size was set to fifty-three bytes. The following figure, figure one,

shows the results of a behavioral simulation, reading four bytes of instruction memory on each rising edge.

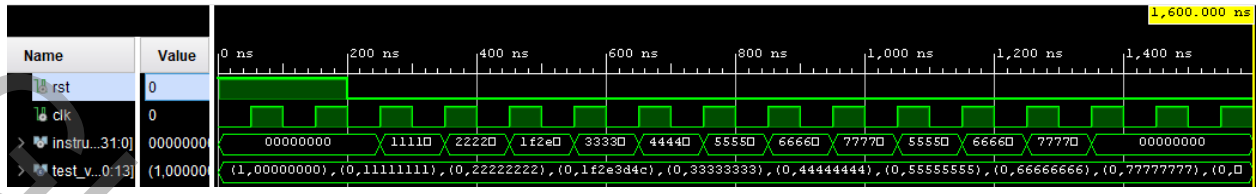

Figure 1: *Instruction Fetch Behavioral Simulation*

Figure one shows word-long instructions, being read on each rising edge. The program counter is also being updated properly as data from previous words is not being added to the next. When reading this data it is important that it is concatenated in the correct order, to build a full thirty-two-bit instruction.

The next figure, figure two, shows how the data is arranged in the intended order. MIPS uses big-endian for memory addressing, meaning that the lowest byte address holds the MSB of the word. This can be seen as the memory stored in the instruction memory is “0x1F”, “0x2E”, “0x3D”, and “0x4C”.

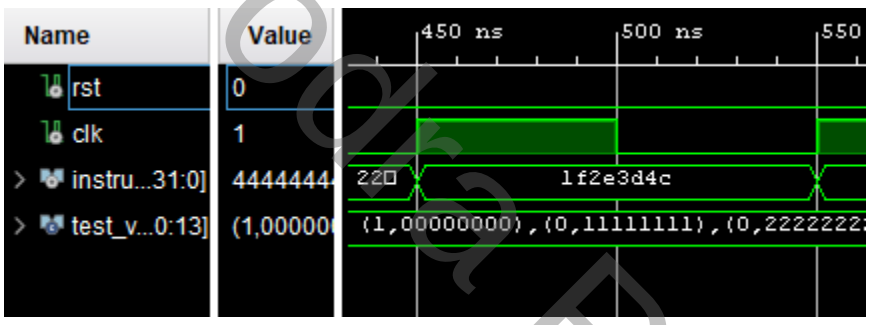

Figure 2: *Big-endian Memory*

This design was also tested using a post-synthesis timing simulation, where gate delay is added. The result of this simulation is found in figure three.

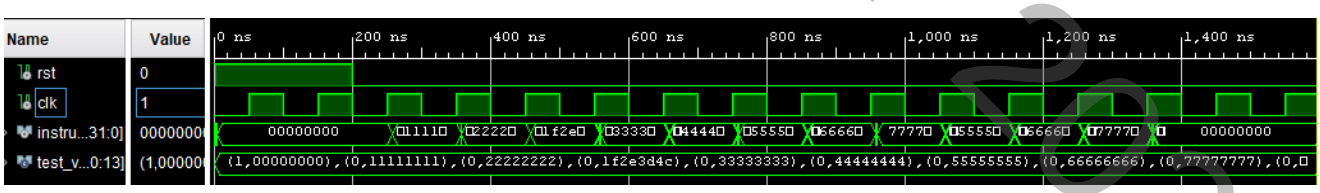

Figure 3: *Instruction Fetch Post-Implementation Simulation*

In figure three, the output signals behave as expected, reading a new instruction at every rising edge. To demonstrate the proper functionality of the over-read protection, the next figure, figure four, shows the instructions being set to zero, as there is no more memory to read.

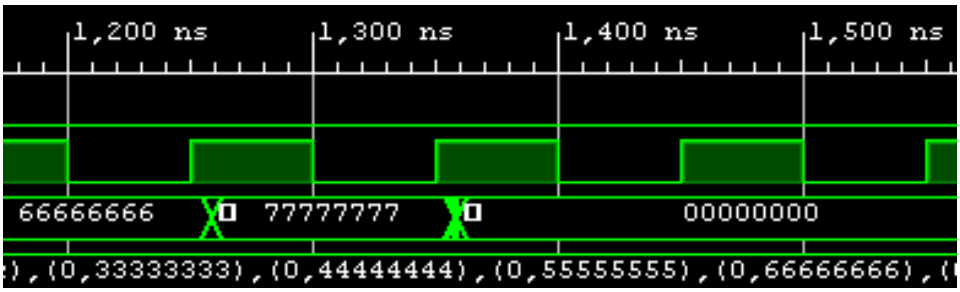

Figure 4: *End of Memory*

At the end of memory, instructions fetched will only return zero. To ensure that this will happen even if there are a few, in this case, three out of the four bytes for a final word present, the memory will limit the last valid readable address to four below the memory limit. In figure four, the clock is also run a few extra cycles, to show that the instructions will all be zero.

In addition to the simulations run in Vivado, this design was also tested using ModelSim. The following figure, figure five, shows the ModelSim simulation.

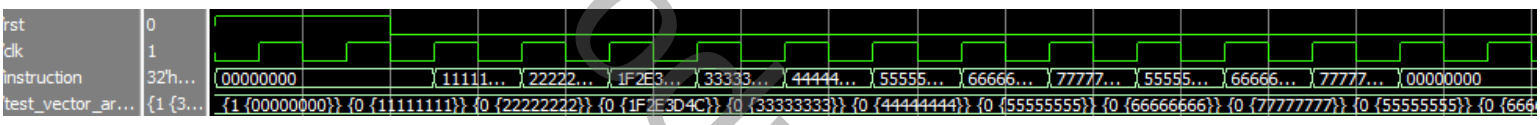

Figure 5: *Modelsim Simulation*

This simulation matches the behavioral simulation from Vivado.

The next design that was tested, was the decode stage. This stage consists of the control unit, registers, and sign extension. In addition, groups of bits are also taken directly from the instruction and passed along to the next stage. There are six control signals generated by the instruction decode. Instructions are assigned to the instruction signal on the rising edge, similar to how instructions were read on the rising edge of the clock in the instruction fetch. The only other synchronized event in this design is the register write, which writes a “random” test value on the falling edge. Once a full processor is designed, this data will be replaced with a value that is actually generated by the execute stage.

To test this design five test cases were added to a self-checking test bench. Since these are “real” MIPS instructions, the correct control signals should be generated. The following figure, figure six, shows the results of a behavioral simulation, running these five instructions.

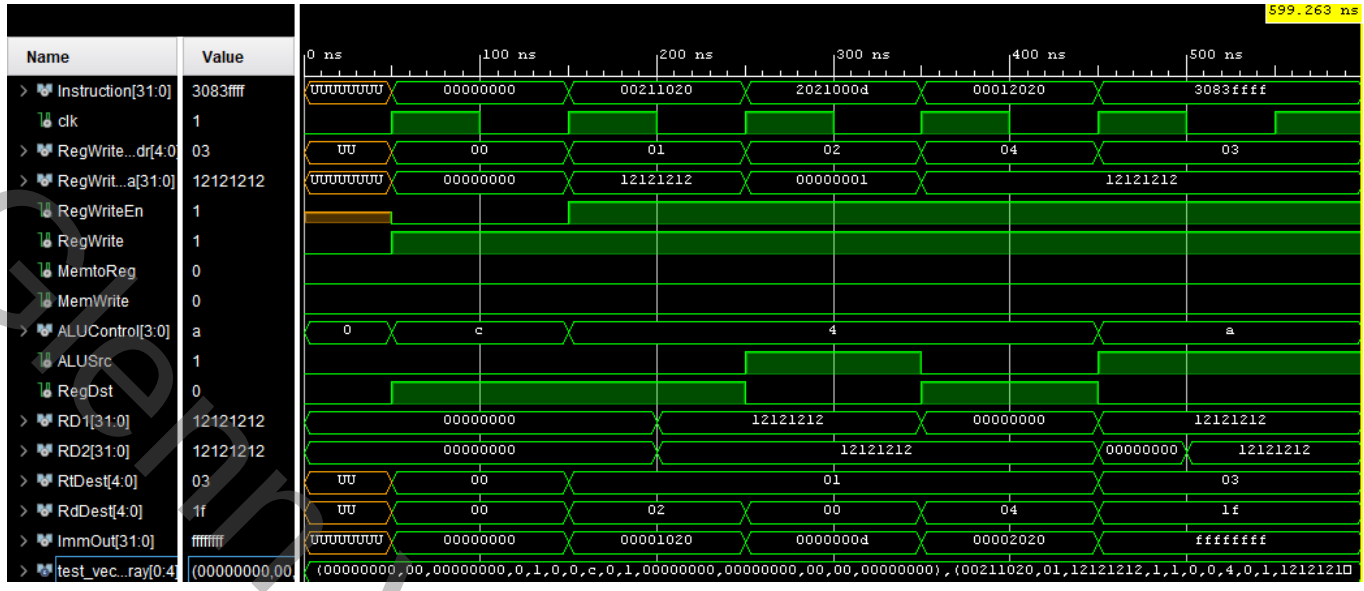

Figure 6: *Instruction Decode Behavioral Simulation*

To understand if the signals in figure six are correct, it is critical to examine the instructions that were tested. The following sequence of instructions were used for this simulation:

NOP	*Instruction 1*
ADD R1, R1, R2	*Instruction 2*
ADDI R2, R1, 13	*Instruction 3*
ADD R4, R0, R1	*Instruction 4*
ANDI R3, R4, FFFF	*Instruction 5*

Something to note about these instructions, is that none of them access memory. In figure six, MemToReg and MemWrite both stay low for the entire duration of the simulation. This is good, as it is not expected that any of these operations would need to write data to memory or store memory data in a register. To look more thoroughly at a test case, and all of its signals, consider figure seven, in which the result of a post-implementation timing simulation is presented.

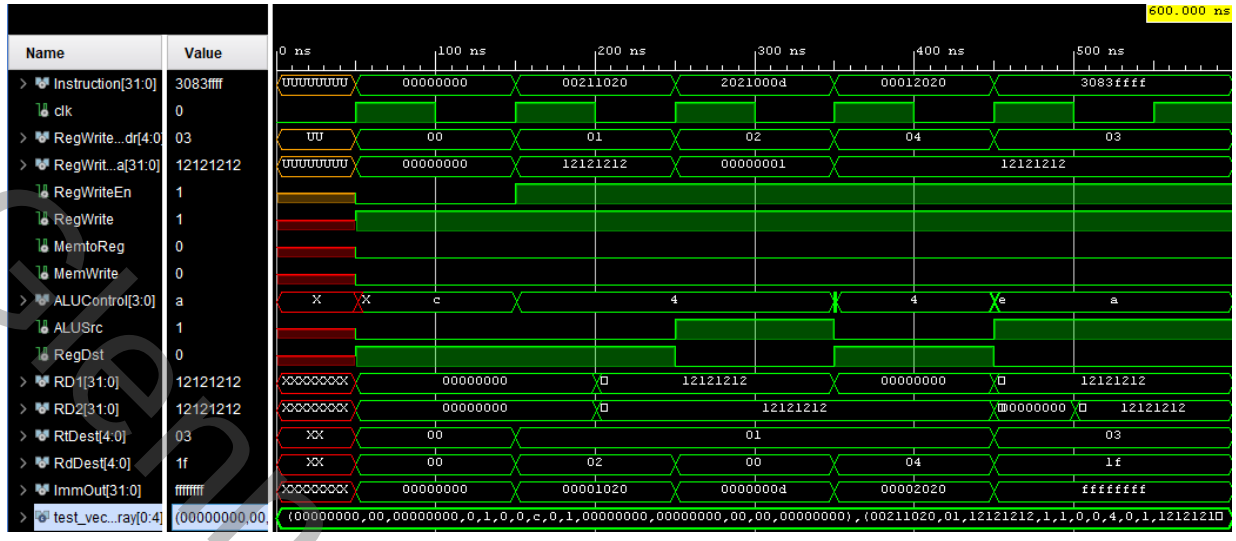

Figure 7: *Instruction Decode Post-Implementation Simulation*

It is worth noting that in figures six and seven many of the signals start as undefined or do not cares. This is expected behavior for this test bench, as the signals are not being initiated to a known value. Usually, this is a bad idea, however in this case it appeared to have no impact on the results. For this design, it is expected that the decode stage performs as intended, regardless of what data might have been present during the last clock cycle.

To look at a specific test case, Instruction 5, the following figure, figure eight, must be considered. This instruction is “ANDI R3, R4, FFFF”.

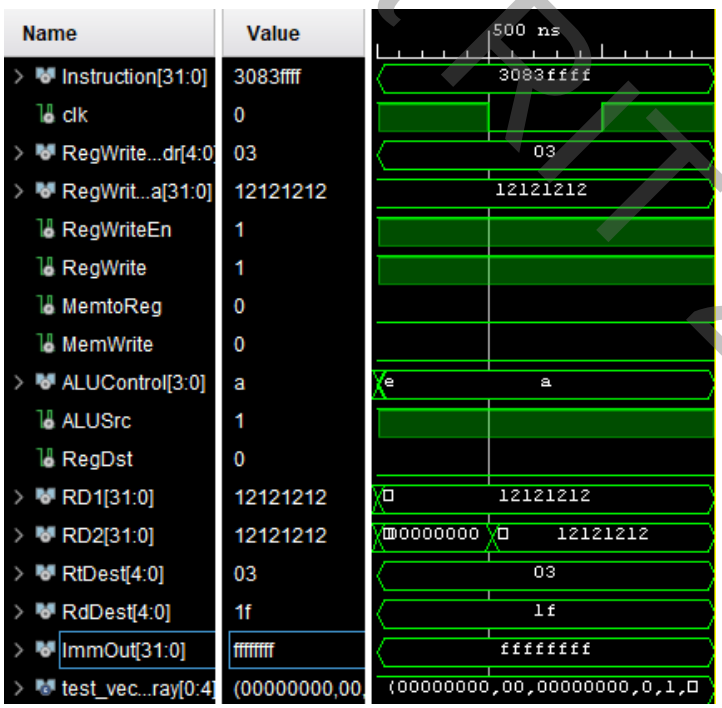

Figure 8: *ANDI Instruction*

Looking at the disassembly for this instruction there are the bits:

001100 00100 00011 11111 11111 111111 *Disassembly 1*

The first six are the Opcode, “001100”, which signifies the operation ANDI. ANDI is an immediate or I-type instruction, where all of the ALU functionality is stored in the OPcode. The ALUOp for the ANDI is the code “1010”, which can be seen as the“0xA” signal in figure eight. For the control signals, RegWrite should be ‘1’ as a register will receive the result of the AND operation. RegDest is ‘1’ as the register RT should be written to, not RD (RD does not exist for I-type instructions). Both memory signals are low, as none of the instructions being executed interact with memory. Lastly, the ALU source should be ‘1’ as the immediate should be used instead of the contents of RD2.

Addr1 and 2 are assigned to their respective bit ranges (giving R4 and R3), and those both return RD1 and RD2. The last piece of notable decoding that is happening is with immediate value. To test the sign extend functionality the value “0xFFFF” is used, which as seen in Disassenbaly 1, is all ones. To get the full thirty-two bit sign-extended immediate, the MSB is copied to fill out the rest of the number, in this case, it adds ones to the rest of the number resulting in “0xFFFFFFFF”. This result can be seen at the bottom of figure eight.

These results produce consistent expected signals throughout the presented figures, across each simulation.

Conclusion

This exercise was successful because two stages of a MIPS processor were successfully created and tested, using various simulation methods and test cases. The first stage, the instruction fetch, will be useful as programs are added to it. The instructions the processor should execute are stored here, and it is important they can be read consistently and accurately. The program counter should also update appropriately to reflect the data size of the instruction. This stage performed as expected and didn’t try to read data as the address moved out of bounds. The second stage, the decode stage, also performed as expected. It was able to generate the correct control signals, read the correct registers, found in the instruction, and sign extend an immediate value for use later. Many of the signals generated in this stage aren’t used here, at the moment. As the rest of the processor design is created and implemented, it is important that these signals are correct. If a single wrong control signal is generated it can cause the entire program to break.