Abstract

Embedded cores can be found on many if not all digital electronics today. Though some serve a specific task, more generalized processors such as microcontrollers, are cheaper to produce and can be adapted to most projects by use of their instructions. For this project, an embedded pipelined processing core was created to be run on a Field Programmable Gate Array (FPGA). This core was codenamed “Omicron.”

Omicron is a general purpose, pipelined processor written in the Verilog Hardware Description Language (HDL). It is designed to be compiled and run on a Xilinx Spartan-3E FPGA evaluation board. The Omicron contains five stages: Instruction Fetch, Instruction Decode, Execute, Memory, and Write Back. By splitting the processor into pipelined stages, the clock speed can be increased causing an increased throughput compared to a non-pipelined CPU.

A lot of work was done in creating all of the code and performing simulations, but more time was needed to finish a project of this magnitude and scale.

Design Methodology

The design and capabilities of the Omicron processor have evolved over the course of its development. It started based on a reference slide of a simple MIPS pipelined architecture, as seen in Figure 1. Because work of this nature had been performed in 0306-550 (Computer Organization) as a project, much of the work was adapted and furthered in this project. An example of an element that was reused is the Instruction Set Architecture (ISA). There are three types of encoding styles in the ISA. Each one tells the data path how to read the data. The sources and destinations are addresses to the registers, shown below.

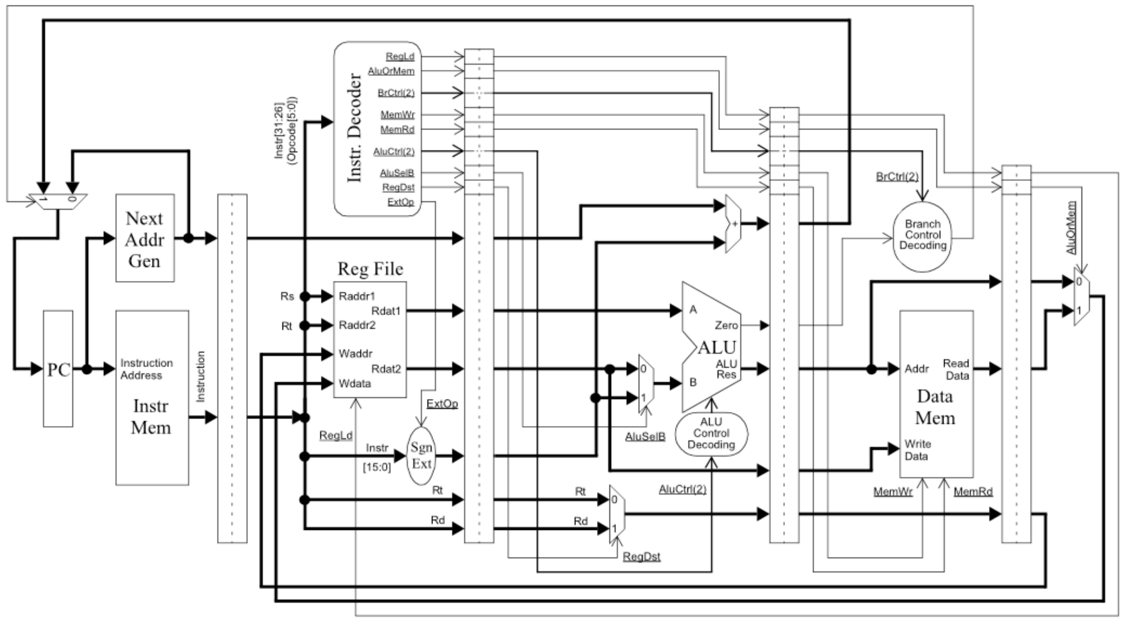


Figure 1: Pipelined MIPS Datapath with Control Signals

Math Type

OpCode [15-12]	Source 1 [11-9]	Destination [8-6]	Source 2 [5-3]	Not Used [2-0]

This is commonly used for adding together two registers and placing the result in the destination. The commands implemented under this type include noop, cpy, add, sub, mul, and, or, not, xor, ls, and rs. Some of these commands only require one source. In this case, the second source address is ignored by the processor. Also, the last two bits of the instruction are not used. In a future revision, these may be used to implement additional math type instructions, such as nor or nand.

Load/Store and Branch Type

OpCode [15-12]	Source [11-9]	Destination [8-6]	Immediate [5-0]

This type is used for loading, storing, and branching. The commands for this type include load, store, beq, and bne. The beq and bne compare the source register data with the destination register. If the branch is taken, the immediate value is added to the current value to determine the next PC value. For the load, the destination is the location that the load will be placed and the immediate is the location in memory to load it from. For the store, the source is where the value to be written is taken from and the immediate is the location in memory for it to be stored. There is no branch prediction in this revision of Omicron. Instead, a smart compiler is assumed that would insert NO-OPs for enough cycles after a branch so that no data hazards occur.

Jump Type

OpCode [15-12]	Address [11-0]

This type is used for jumping using the jump command. The address stored is the instruction address is the next Program Counter (PC). Jumps, unlike Branches, will always be taken.

The following table shows the 16 instructions implemented in Omicron and their corresponding control unit signals.

Instruction	Type	opcode [15:12]	cu_alu_opcode[10:0]	cu_reg_load	cu_reg_data_loc	cu_branch[1:0]	cu_dm_wea	cu_alu_sel_b
NOOP	Math	0000	00000000001	0	0	00	0	0
CPY	Math	0001	00000000010	1	0	00	0	0
ADD	Math	0010	00000000100	1	0	00	0	0
SUB	Math	0011	00000001000	1	0	00	0	0
MUL	Math	1000	00000010000	1	0	00	0	0
AND	Math	0100	00000100000	1	0	00	0	0
OR	Math	0101	00001000000	1	0	00	0	0
NOT	Math	0110	00010000000	1	0	00	0	0
XOR	Math	0111	00100000000	1	0	00	0	0
LS	Math	1001	01000000000	1	0	00	0	0
RS	Math	1010	10000000000	1	0	00	0	0
BEQ	Ld/St/Br	1011	00000001000	0	0	01	0	0
BNE	Ld/St/Br	1100	00000001000	0	0	10	0	0
LD	Ld/St/Br	1101	00000000010	1	1	00	0	1
STR	Ld/St/Br	1110	00000000010	0	0	00	1	1
JMP	Jump	1111	XXXXXXXXXXX	0	0	11	0	0

Table 1: Omicron Instruction Set Architecture

The *instruction* field in Table 1 shows which instruction is to be executed. The *type* is which of the three types the instruction belongs. The *opcode* is for the instruction; it is different depending on the instruction. The *cu _alu_opcode* is an 11-bit signal that tells the ALU which operation it should execute. This field is encoded in a one-hot encoding style as to improve performance – the ALU will not need any addition decoding when the signal arrives. The *cu_reg_load* signal is used to tell the register file when to perform a write; if it is a 1, write, if it is a 0, don't write. The *cu_reg_data_loc* signal selects between writing the ALU result (0) or the Data Memory result (1) to the register file. The 2-bit *cu_branch* signal is used to select between a jump (11), a branch on equal (01), or a branch on not equal (10). The *cu_dm_wea* signal is used to write (1) to the data memory, or to read (2) from it. The *cu_alu_sel_b* signal is used to select which data is fed into the b input in the ALU; if it is a 1, the data will come from the "Immediate" value; else, it will come from the "Source 2" register.

There are $2^3 = 8$ registers because the encoding for each register is three bits long. The registers are named $A, $B, $C, $D, $F, $G, and $0 (which is the zero register). The zero register will always contain the value 0x0000. Each register has the size of 16 bits, allowing for 2^{16} bytes (65536 B or 64 KB) to be accessed in memory.

Register Name	Register Number
$0	000
$A	001
$B	010
$C	011
$D	100
$E	101
$F	110
$G	111

Table 2: Omicron Registers

The program counter is 2^7 bits so the maximum amount of instructions one can have in a program is $2^7 = 128$ instructions.

Since most of the design work had been completed in a previous class, most of the work on Omicron was done in writing the Verilog HDL code, simulation, synthesizing, mapping, and routing onto the Xilinx Spartan-3E FPGA Evaluation Kit.

Procedure

References from the Rochester Institute of Technology (RIT) Wallace Library demonstrated portions of pipelined HDL code. In order to pipeline a processor, the multiple stages are split up and synchronized by vectors of D-Flip Flops (DFFs). This causes instructions to take more clock cycles to go through the processor, but you can significantly speed up your clock, and push more instructions through at the same time. The benefits from pipelining are observed in all modern processors.

The Arithmetic Logic Unit (ALU) was designed first. A one hot encoding was chosen for speed. Then the stages were created individually. A decision was made to synchronize the DFFs on the rising edge, and execute ALU and memory operations on the falling edge; that way execution between stages was still synchronous. Xilinx Design Suite has tools to generate memory cells and a clock block, which were used in this project. Each of the five stages were designed as separate modules.

Once the data path was fully laid out, the control unit was implemented. A few changes were made to the ISA as well. The data path and control unit were separate modules. By modularizing the code, debugging and testing became easier.

ModelSim was used to simulate Omicron. Testing was done first on individual modules using force vectors. Then, the entire project was simulated at once using test code. Before testing could begin, the Xilinx HDL simulation libraries needed to be compiled. To accomplish this, the "*Compile HDL Simulation Libraries*" process was run inside of the *ISE Project Navigator* under *Simulation*. Once this process was completed, ModelSim was able to open up any of the stages and use the modules created by the *Xilinx Core Generator*, such as the memories.

For the execute stage, a force file was used to assure that each opcode performed the correct arithmetic function. That same force file also tested that the control unit signals, such as cu_alu_sel_b and cu_reg_dest, worked as expected. The input ports, id_next_addr and id_registerX_addr, were also tested to assure correctness. The force file is entitled, force_execute.txt.

The instruction decode stage was tested with a force file called force_instruction_decode.txt. This force file sent write commands to the register file using wb_reg_waddr, wb_reg_wea, and wb_reg_wdata. Writes were sent to each of the eight register in the register file and were then read back to assure it held data correctly.

Both the instruction memory and data memory had to be preloaded with instructions and data. The instructions were stored as machine code in hexadecimal.

Results

A lot of time was spent writing the 12+ Verilog HDL files. Reusing previous designs really helped, but many edits and additions were also performed. For each module created, many extensive tests had to be written, performed, and understood which was very time consuming. There were very many signals to keep track of.

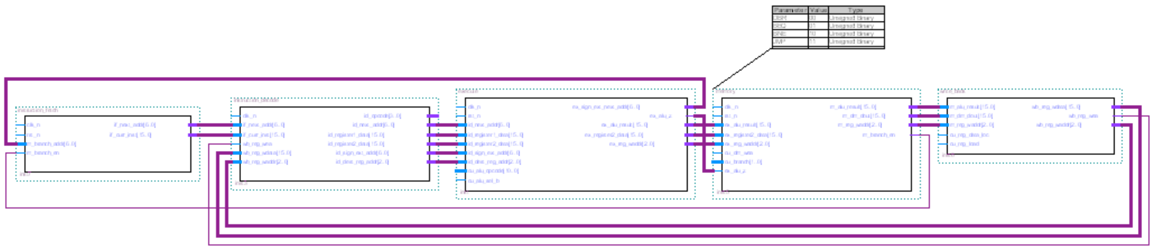
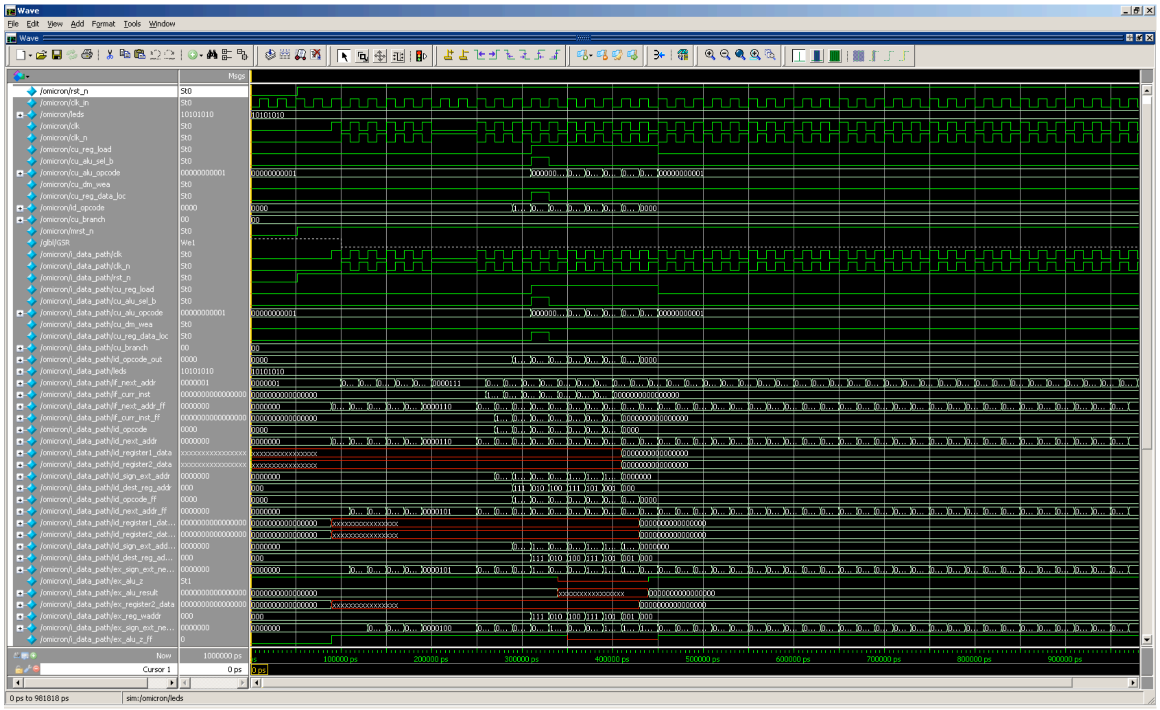


Figure 2: Omicron Datapath

Figure 2 shows the data path of Omicron. The first stage of the pipeline is the Instruction Fetch. Next is the Instruction Decode. Followed by that is the Execute stage. After that is the Memory stage. Finally is the Write back stage. By combining those five stages together, and placing the DFF vectors between them, a pipelined processor is created.

Each of the five stages were written and tested separately for correctness. In theory, all five stages should be able to come together to make a working processor. However, Murphy’s Law states that is something could go wrong, it will.



```
/omicron/rst_n                              St0
/omicron/clk_in                             St0
/omicron/leds                               10101010
/omicron/clk                                St0
/omicron/clk_n                              St0
/omicron/cu_reg_load                        St0
/omicron/cu_alu_sel_b                       St0
/omicron/cu_alu_opcode                      00000000001
/omicron/cu_dm_wea                          St0
/omicron/cu_reg_data_loc                    St0
/omicron/id_opcode                          0000
/omicron/cu_branch                          00
/omicron/mrst_n                             St0
/glbl/GSR                                   We1
/omicron/i_data_path/clk                    St0
/omicron/i_data_path/clk_n                  St0
/omicron/i_data_path/rst_n                  St0
/omicron/i_data_path/cu_reg_load            St0
/omicron/i_data_path/cu_alu_sel_b           St0
/omicron/i_data_path/cu_alu_opcode          00000000001
/omicron/i_data_path/cu_dm_wea              St0
/omicron/i_data_path/cu_reg_data_loc        St0
/omicron/i_data_path/cu_branch              00
/omicron/i_data_path/id_opcode_out          0000
/omicron/i_data_path/leds                   10101010
/omicron/i_data_path/if_next_addr           0000001
/omicron/i_data_path/if_curr_inst           0000000000000000
/omicron/i_data_path/if_next_addr_ff        0000000
/omicron/i_data_path/if_curr_inst_ff        0000000000000000
/omicron/i_data_path/id_opcode              0000
/omicron/i_data_path/id_next_addr           0000000
/omicron/i_data_path/id_register1_data      xxxxxxxxxxxxxxxx
/omicron/i_data_path/id_register2_data      xxxxxxxxxxxxxxxx
/omicron/i_data_path/id_sign_ext_addr       0000000
/omicron/i_data_path/id_dest_reg_addr       000
/omicron/i_data_path/id_opcode_ff           0000
/omicron/i_data_path/id_next_addr_ff        0000000
/omicron/i_data_path/id_register1_dat...    0000000000000000
/omicron/i_data_path/id_register2_dat...    0000000000000000
/omicron/i_data_path/id_sign_ext_add...     0000000
/omicron/i_data_path/id_dest_reg_ad...      000
/omicron/i_data_path/ex_sign_ext_ne...      0000000
/omicron/i_data_path/ex_alu_z               St1
/omicron/i_data_path/ex_alu_result          0000000000000000
/omicron/i_data_path/ex_register2_data      0000000000000000
/omicron/i_data_path/ex_reg_waddr           000
/omicron/i_data_path/ex_sign_ext_ne...      0000000
/omicron/i_data_path/ex_alu_z_ff            0
                                    Now     1000000 ps
                               Cursor 1     0 ps
0 ps to 981818 ps        sim:/omicron/leds
```

Figure 3: Omicron Simulation Testing

Figure 3 shows an example screenshot of testing inside of ModelSim. As can be seen by the figure, testing the processor for correctness was, and will be, a very time consuming process that would take much longer than six weeks to complete.

Conclusion

In the end, the entire design was created and coded in Verilog, as can be seen in the many appendixes attached. However, because of the size of the processor and the amount of time that it would have taken to test every feature, it cannot yet be called a success. Many bugs were fixed and problems were solved, such as how to test a Xilinx-created memory.

The project was a lot of fun and furthered our understanding of Verilog HDL, FPGAs, processor architecture, and digital system design. Additionally, it showed us that a project of this scale requires significantly more time to produce. A less intense project could have been chosen, but it would not have provided the learning experience that came along with designing and implementing a pipelined processor.