Description:

Omicron is a general purpose, pipelined CPU written in the Verilog hardware description language. 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, Write Back. By splitting the CPU into pipelined stages, the clock speed can be increased causing an increased throughput compared to a non-pipelined CPU.

OpCode Types:

There are three types of encoding styles. Each one will tell the data path how to read the data. The sources and destinations are addresses to the registers, shown below.

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. Some commonly used commands are ADD, SUB, AND, OR, etc.

Load/Store and Branch Type

OpCode [15-12] Source 1 [11-9] Destination [8-6] Immediate [5-0]

This type is used for loading, storing, and branching. Commonly used commands are load, store, beq, and bne.

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 for the next address.

Registers:

There will be $2^3 = 8$ 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). 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

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

Instructions:

The following table contains the 16 instructions implemented in Omicron.

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	cu_reg_dest
NOOP	Math	0000	00000000001	0	0	00	0	0	1
CPY	Math	0001	00000000010	1	0	00	0	0	1
ADD	Math	0010	00000000100	1	0	00	0	0	1
SUB	Math	0011	00000001000	1	0	00	0	0	1
MUL	Math	1000	00000010000	1	0	00	0	0	1
AND	Math	0100	00000100000	1	0	00	0	0	1
OR	Math	0101	00001000000	1	0	00	0	0	1
NOT	Math	0110	00010000000	1	0	00	0	0	1
XOR	Math	0111	00100000000	1	0	00	0	0	1
LS	Math	1001	01000000000	1	0	00	0	0	1
RS	Math	1010	10000000000	1	0	00	0	0	1
BEQ	Ld/St/Br	1011	00000001000	0	0	01	0	0	1
BNE	Ld/St/Br	1100	00000001000	0	0	10	0	0	1
LD	Ld/St/Br	1101	00000000010	1	1	00	0	1	1
STR	Ld/St/Br	1110	00000000010	0	0	00	1	1	1
JMP	Jump	1111	XXXXXXXXXXX	0	0	11	0	0	1

Control Unit:

opcode [15:12] (4 bits) is the input to the control unit.

cu_alu_opcode[10:0] (11bits) is used for math types and SUB for branches, otherwise don't care (use NOOP).

cu_reg_load should be 0 for branches, jumps, store, and no op (since these don't write to registers), 1 otherwise.

cu_reg_data_loc should be 1 for loads (since you're reading from memory, not the ALU), 0 otherwise.

cu_branch[1:0] (2 bits) is 00 for math types as well as load and store, 01 for BEQ, 10 for BNE, and 11 for jumps.

cu_alu_sel_b should be 1 for loads and stores, 0 otherwise.

cu_reg_dest is unneeded due to the format of our instructions. Oops!

Simulating:

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 8 register in the register file and were then read back to assure it held data correctly.