**Implementation details**

**Control:** Our processor has two control units, a main control unit that is the source of many one or two bit multiplexer control signals, and an ALU controller that is the source of a 4 bit ALU op code. Both controllers take a 4 bit op code and a 4 bit function code from the instruction, however, the function code is ignored unless the op code equals 0.

The main control unit is a state machine implemented in behavioral Verilog. When the state machine’s current state is Fetch (state 0), the controller will read the op code to determine its next state. If the op code is 0, then the controller will select the next state based on the func code. If the control unit is not in the “fetch” state, it’s next state is based on its current state and the op and func codes are ignored. This means the controller will continue to execute the given instruction, whether the op code changes or not.

The ALU control unit is much simpler. It simply takes in an op and function code, ignoring the func code unless the op code is zero, ALU uses the same arithmetic fuctions for different commands, the ALU controller acts as a interpreter for the op codes fed into the ALU. The ALU control is implemented in behavioral Verilog.

**Arithmetic Logic Unit**: Our ALU is implemented in behavioral Verilog. The ALU takes a 4-bit control signal, two 16-bit arithmetic inputs (A and B), one 16-bit arithmetic output, and one single-bit “isZero” output used in comparisons. The ALU can perform 12 operations, as follows: Add, or, xor, and, nor, sll (shift left logical), srl (shift right logical), sub (subtraction), nand, mult (multiplication), eq0 (set if equals 0), slt (set if less than). The ALU is used in any arithmetic or logic based instructions.

**Register File**: Our Register File was built in behavioral Verilog. It contains eight 16-bit registers. Two registers can be read at once, however only one address can be written to at once. In addition to the 2 read out signals, there is a main register out signal that always outputs the data from register [0], the accumulator. This data is wired straight to dataIn on our memory, since the only way to store data on the stack is directly from this register using the store word command.

**Adder**: This adder is implemented in behavioral Verilog. Theadder isusedat one points in the datapath, to add 1 to PC every instruction to simplify the datapath by saving wires and muxes going into the ALU.

**Sign Extender:** The sign extender is written in behavioral Verilog, and is used to sign extend 9 and 12 bit immediates into 16 bit immediates that can be used in the ALU.

**Memory**: We used the IP(Core Generator & Architecture Wizard) to generate a block memory for our project, exactly as described in Lab 7. This memory will be double clocked, to prevent any delays in data access. There will be one memory, which will store instructions and a stack for use by programs. The memory has an access depth of 16 and an access width of 1024.

**Registers and Multiplexers**: Registers and multiplexers are built into the syntax of behavioral level Verilog, and we have not implemented our own versions.

**Datapath:** We have not implemented a datapath design in Xilinx yet, however we plan to use behavioral level Verilog code to link the modules we have designed and tested.