**Executive Summary:**

Our group created a multicycle, load-store processor that is capable of running Euclid’s algorithm. Taking tips from how accumulator-based processors dictate assembly, every instruction that doesn’t use immediates takes only a maximum of two registers as parameters, instead of three. This extra space is used for a condition code, which dictates whether the instruction should be executed or not. With load-store, our assembly language is very concise and powerful, and programs can be written in much fewer lines than other architecture types. This processor is capable of function calls, recursion, and uses two stacks to keep track of variables and data. It is powerful, fast, and we are proud of it.

**Introduction:**

Our processor uses a multicycle datapath. It combines a program counter, block memory, register file, ALU, and many, many muxes and small registers to process instructions and programs. The strength of this processor lies in its ability to skip instructions. Every instruction that doesn’t use immediates takes a condition code, and only if this code is valid does the instruction execute. This allows for easy multi-line if statements, loops, and large if/else-if/else statements. Our original objective was to make the processor very easy on the hardware side to design, but we ended up trying to create a very powerful and efficient instruction set, both in cycle time and number of lines of code. Our processor does not have any major weaknesses, but there are many places where it could be improved, like making the stack pointer accessible to the user.

**Body:**

(Instruction Set Design)

After deciding on a load-store architecture, coming up with how to write instructions was a lot simpler. Though our processor is not an accumulator, our instruction set was designed so that it could be used on an accumulator architecture. Every operation instruction takes the first register passed in and uses it also as the register that is written to afterward. In programming terms, this makes every “+” act like a “+=”—every time an addition is made, the result is stored back in the first register. This saves a lot of space in the instructions and allows for use of the flag register. The flag register stores a three bit value that shows whether one number is less than, equal to, or greater than another number. These two numbers are supplied by the programmer and must be kept track of manually, but the value in the flag register is never zero. The flag register can be updated with the condition code from compare instruction (cmp). For every instruction that doesn’t take an immediate (any non-I-type instruction), the three of the four saved bits are reserved for the condition code. If the condition code anded with the flag register isn’t zero, then the instruction is executed. This means that if statements can be performed without using branches, saving many lines of code when writing long problems. However, because our jumps are not I-type instructions, they also take a condition code. In this processor, jumps and branches are the same thing. After a very long debate we chose to scrap all branches and get rid of unconditional jumps, instead having two different types of jumps—jump label and jump register. This allows for both the functionality of jumps and branches. In addition, with conditional jumps and the compare command, programmers for our processor have no need for a zero register or a one register, though this was a much-debated topic.

This processor does not do shifts. Though our group left the opportunity for a shift command open in the case that we finished our other designs early, it was decided that it would be too difficult to write one in.

(Implementation)

The CPU has two components--the datapath and the control. The datapath processes data and manipulates data flow, while the control instructs the datapath what job to do. In our multi cycle processor, the datapath is designed to be synchronous. The Control Unit is a mealy state machine, able to output different control signals in the same state based on the datapath status flag.

Memory and the register file are clocked so they only change their output synchronously on the clock edge. In addition, block memory only takes one cycle to use. Most inputs to these components are controlled by control signals and muxes. There are a few special registers--PC keeps track of program counter, which increments by one unless there is a jump instruction; SP keeps track of the stack pointer and only push and pop can change it by one; and the flag keeps track of the last comparison result and enables our processor to skip instructions and jump conditionally. The datapath has three status flags to inform the control of its status--Op specifies which instruction is executing, perform decided whether this instruction will be finished, and LMC controls whether to load from memory for that instruction.

The control unit has eight total states. All instructions execute in 2 to 4 cycles. For most instructions, there is instruction fetch, decode, memory load, and execution. The briliand design enables our control unit to branch out at the second cycle and save one cycle. For simple instructions like cpi and lui, control manages to skip the decode cycle and write the immediate value into the register in the second cycle. Those two instructions only takes two cycles, then. For non-I-type instructions, when Perform is low, the control changes to state 1 and skips the instruction. Skipping takes only two cycles. For a non-I-type instruction when LMC is high, the control goes to a special load memory cycle and continues normally afterwards. We manage to combine the write back cycle with execution, so the executed result is written into a register or memory in the same cycle.

Our processor can handle I/O and exceptions. Whenever there is an I/O interrupt, we can RESET our CPU to a designated PC address, and guide it to execute instructions there. In the meantime, the stack pointer is reset back to the top of stack and register $t0 contains an argument input from the user. Whenever there is an exception, the exception bits go high, RESET the CPU and abort the process.

(Xilinx Model Explanation)

Most of our Xilinx models are implemented with Verilog. WIth Verilog, building a CPU is really easy and straightforward.

Muxes can be built with ternary operator in Verilog.

The instruction register is just a 16-bit latch outputting different portions of its values, which can be specified with the { , } syntax. Writing into a latch can be implemented with an always @(\*) block.

The register file is just a group of 16 16-bit registers. To implement this with Verilog, we can directly declare an array of 16 registers, and update outputs and the clock edge. Always @(posedge CLK) is very handy. One tricky part is to use non-blocking assignment <= to shrink the clock cycle.

The ALU was implemented with Verilog with cases, which generate a mux.

Memory is a bit tricky. We used the block memory provided by Xilinx IP generator. In order to make the fastest processor, the block memory would need a whole clock cycle to read and write. It only outputs data at the clock edge. We made a few modification late in the term to accommodate this memory design. A delayMux is added and instruction register becomes a latch. Those design decision and changes are discussed in details in design document and design journal.

The Flag was implemented with Verilog. It has three flip flops, combinational logic to determine Perform, its output. Perform is asynchronous, so it updates whenever a new condition code arrives.

Our Control Unit is implemented in Verilog with both always @(posedge CLK) blocks to control state transition and continuously assign the control signal output. Since the datapath is much slower than the control, and the state transition is not part of the critical path of our CPU, there is no need for optimization. However, our datapath needs to wait for the control signal to start performing its work, so control signals are part of the critical path. Control signals are related to both state and input. K-Maps and utilizing the ability to ignore a signal simplified the control signal logic greatly.

As for I/O and exception handling, our processor has the ability to generate exceptions, like segment fault, overflow, or I/O interrupts by putting arguments into the register upon pushing a button. We did not build exception handlers for them yet, though, and all exceptions just abort the process. I/O is working with the board. We can send an argument to the $t0 register and display value in $v0 register with LCD.

(Testing)

For the testing of our processor, we wrote several different levels of testing—unit testing, two levels of integration testing, and system tests.

Our unit tests are fairly straightforward. For every component that we wrote, we made a list of desired inputs and expected outputs that this component could provide while it was isolated from the rest of the datapath. In addition to a base case that covered general instructions, each component would have all of its corner and edge cases tested so that our group would know that it could take strange instructions and wouldn’t break seemingly randomly in the middle of a program.

The unit test for control unit requires a lot of different cases. To make our life easier, we create a ControlUnitTester, which is built with all cases. Regarding each state and datapath flag, it provides necessary information, including whether certain control bit is 1, 0 or don’t care, what the next state is supposed to be, whether combination of state and status flag are unreachable. ControlUnitTester copies specifications directly from our design and cover all the cases. It is very slow, but guaranteed to match the design. Then we can run our control unit with all combination of possible inputs several cycles and compare outcome with ControlUnitTester. A lot of buts are caught in this process, and there is no more bugs discovered in Control Unit after unit test.

Integration tests were more difficult. First, we had to come up with meaningful combinations of the units of our datapath. As a group, we sat down and picked useful combinations and voted which ones to keep and which to discard. After that, we came up with useful combinations of those integration tests to be additional integration tests. Then, for each combination, we came up with a list of inputs and outputs to test base cases and all edge and corner cases. These tests were more difficult to write, as we had to test components that we had not written. Memory and the register file were difficult because they first had to be loaded with values, then have the values read. In addition, the clock had to be cycled manually during these tests, and components like multiplexers had to be tested in a certain order.

We make modification to our datapath and write several tasks in Verilog to ease system task:

run( PCstart , num) -- runs num instructions from PCstart.

readMem( address , data) -- read Mem[address] and store data for later comparasion.

writeMem( address , data ) – write data into Mem[address]

readReg( address , data) -- read Rem[address] and store data for later comparasion.

writeReg( address , data ) – write data into Rem[address]

With the help of those tasks, we can simply put instruction into the memory, run them and examine the output, all in test bench, saving a lot trouble from modifying coe file.

System tests are composed of several different levels. The first level of system tests is for individual instructions. Once each instruction was verified to be working, small sets of instructions were tested together. These tests were composed of our short code snippets from the first milestone of the project and pseudoinstructions that compiled down into more than one regular assembly instruction. These did not need edge cases to be tested, though we were careful to use plenty of tests to help make us more certain of the abilities of our processor. Finally, we tested the Euclid’s algorithm program on our processor and debugged that to prove that long programs could run without problems. Most of the problems we ran into with making Euclid’s algorithm work properly were with Xilinx, and not with our processor. However, we were forced to make a small change to our instruction set to make the program run easier, which was adding cpi (copy immediate) to the standard instruction set instead of having it be a psuedoinstruction.

After these tests all ran and passed, we felt very confident in saying that our processor was functional.

(Performance Results)

Performance summary

1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.

Main function has 3 instructions, 10 Bytes, including one jl instruction at the end to create an infinite loop. RelPrime has 19 instruction, 42 Bytes. Gcd has 13 instructions, 30 Bytes. Altogether we have total 33 instructions, 82 Bytes program.

We only need 5 slots in the memory for backup register. The minimum memory run for E

euclid's algorithm is 82 + 5 \* 2 = 92 Bytes.

1. The total number instructions executed when relPrime is called with 0x13B0 (the result should be0x000B using the algorithm specified in the project specifications).

51090 instructions

1. The total number of cycles required to execute relPrime under the same conditions as Step [2](https://www.rose-hulman.edu/class/csse/csse232/www/csse232_project_milestone_6.html#step:inst).

143029 cycles

1. The average cycles per instruction based on the data collected in Steps [2](https://www.rose-hulman.edu/class/csse/csse232/www/csse232_project_milestone_6.html#step:inst) and [3](https://www.rose-hulman.edu/class/csse/csse232/www/csse232_project_milestone_6.html#step:cyc).

2.7995 CPI

1. The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary).

10.770ns

1. The total execution time for relPrime under the same conditions as Step [2](https://www.rose-hulman.edu/class/csse/csse232/www/csse232_project_milestone_6.html#step:inst).

143029 cycles \* 10.770ns/cycle = 1.5404ms

1. The gate count for your entire design (from the Xilinx Map report). This appears to have changed/is omitted in recent version. Extra credit for any group that finds a reasonable way to estimate the equivalent gate count from the data in the Xilinx reports.

1. The device utilization summary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Logic Utilization | Used | Available | Utilization | Note(s) | |
| Number of Slice Flip Flops | 21 | 9,312 | 1% |  | |
| Number of 4 input LUTs | 71 | 9,312 | 1% |  | |
| Number of occupied Slices | 36 | 4,656 | 1% |  | |
| Number of Slices containing only related logic | 36 | 36 | 100% |  | |
| Number of Slices containing unrelated logic | 0 | 36 | 0% |  | |
| Total Number of 4 input LUTs | 71 | 9,312 | 1% |  | |
| Number of bonded IOBs | 39 | 232 | 16% |  | |
| Number of BUFGMUXs | 1 | 24 | 4% |  | |
| Average Fanout of Non-Clock Nets | 3.20 |  |  |  | |

**Conclusion:**