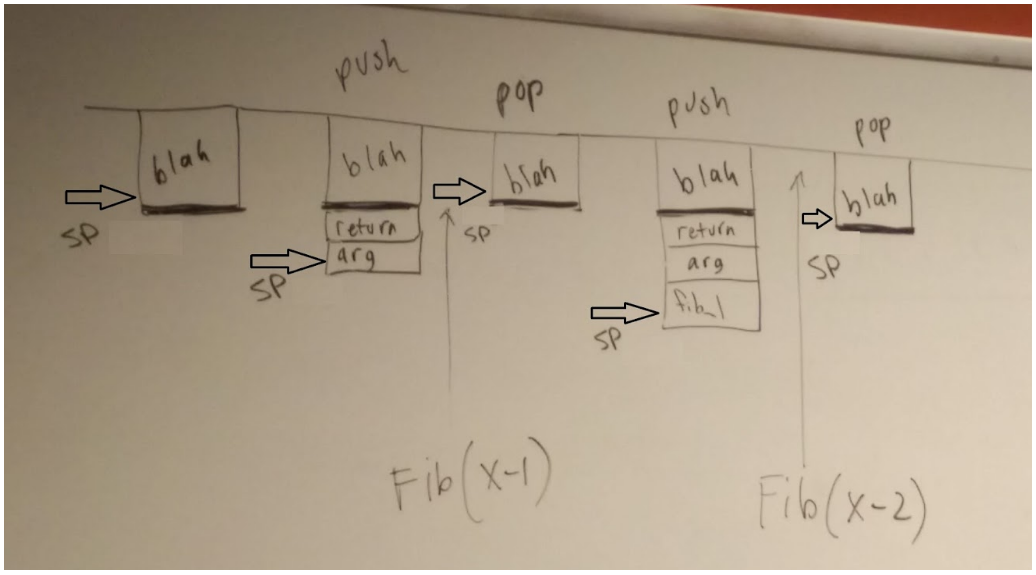
MP3 Writeup

Jacob Kingery, Ryan Louie, Kyle Mayer, Ankeet Mutha

Stack Diagram:



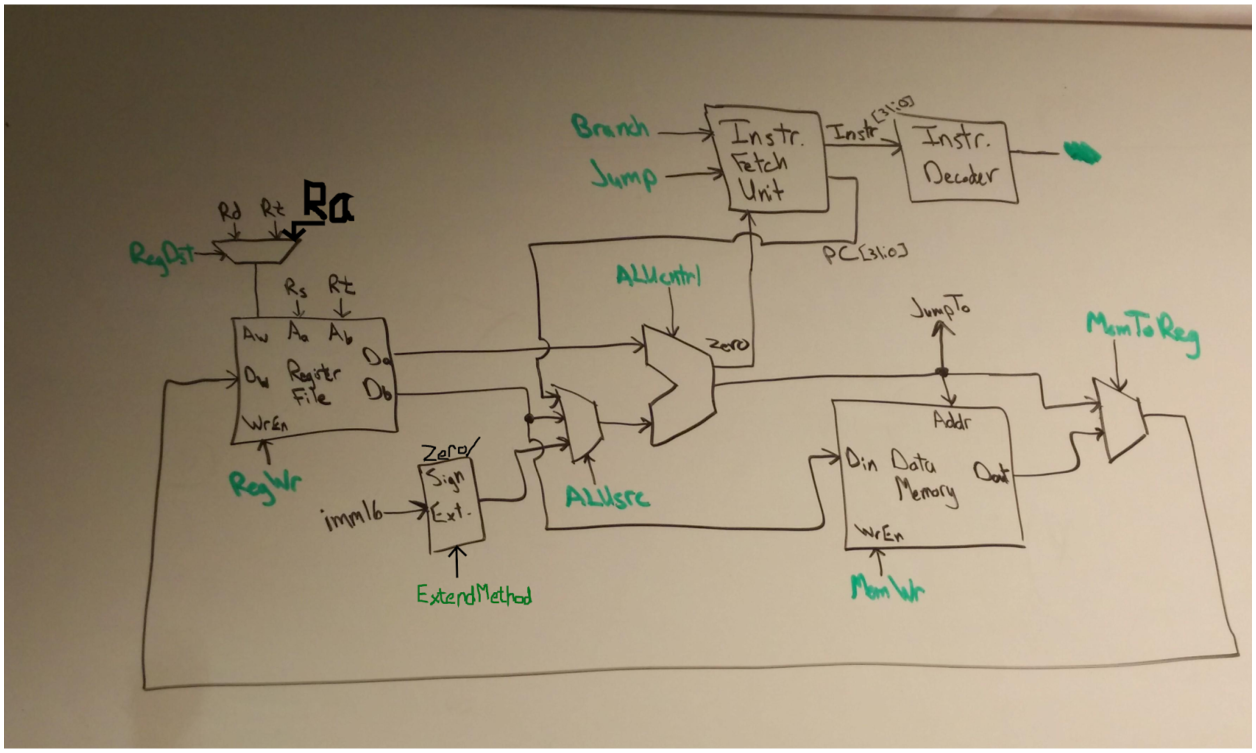
Instructions Implemented:

addiu, jal, addu, add, addi, slt, bne, beq, jr, sw, lw

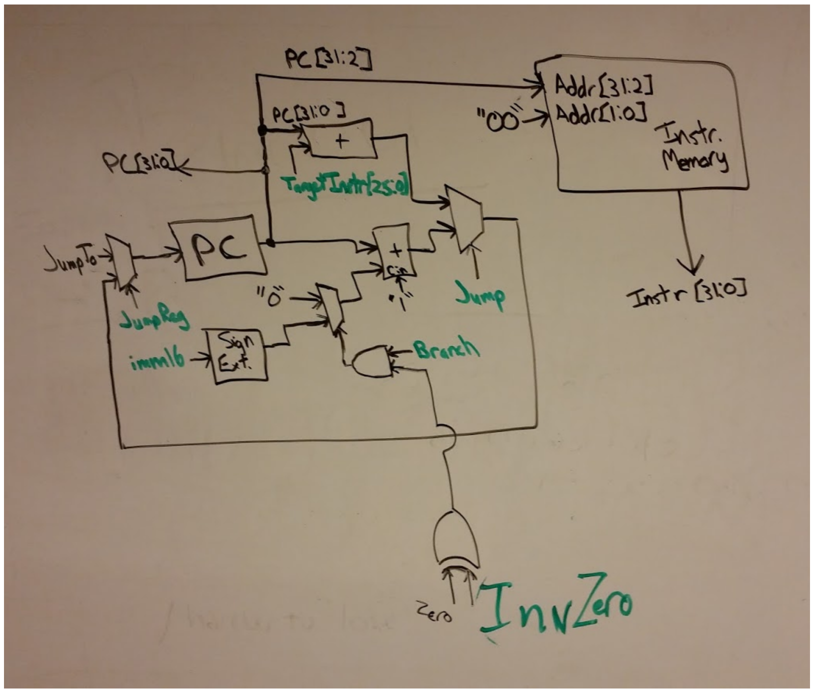
CPU Design Description:

We decided to create a single cycle CPU for design simplicity. This means that a single instruction is completed on each clock signal, although it may be limited to slower clock speeds than similarly scoped multi-cycle CPUs.
Our CPU includes a register file, which is the working memory of the processor. This is the fastest access memory, although it is very small. It stores values used for computation and fast access. The register file feeds into the ALU (arithmetic logic unit), which contains the logic for mathematical computation. It can perform basic math on two values, including addition, subtraction, and boolean operations. the output of the ALU is routed to two places; back to the regfile if continued computations are necessary, or to the data memory for longer term storage. The data memory is slower to access and can't feed directly into the ALU, but is a much larger data storage method. It is where the stack lives, and is where data is stored when the regfile isn't large enough. Then there is the Instruction fetch unit, which keeps track of the program counter and grabs the appropriate instruction from the program memory. Since this instruction is a 32 bit word, the instruction decoder is a lookup table that translates this binary code into all the flags that control the operation of the rest of the CPU.

CPU Block Diagram:



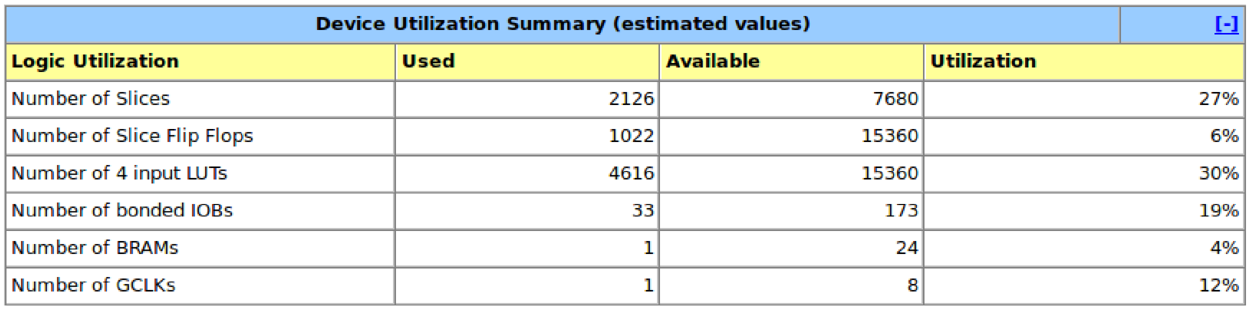
Instruction Fetch Unit Block Diagram:



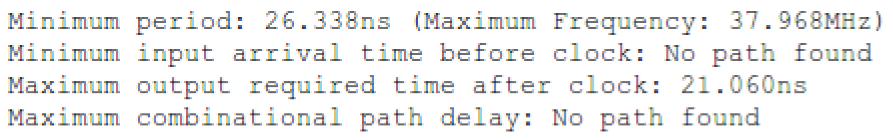
Processor Functionality Verification:

After running “doFibonacci.do”, move the cursor to the end of the wave form and view the “answer” waveform to read the result of our program. The value for Fib(4) + Fib(10) is 0x3a. Notice that the “sp” waveform ends at 0x3ffc, meaning that the stack is balanced.

Processor Statistics:



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2126	7680	27%
Number of Slice Flip Flops	1022	15360	6%
Number of 4 input LUTs	4616	15360	30%
Number of bonded IOBs	33	173	19%
Number of BRAMs	1	24	4%
Number of GCLKs	1	8	12%



```
Minimum period: 26.338ns (Maximum Frequency: 37.968MHz)
Minimum input arrival time before clock: No path found
Maximum output required time after clock: 21.060ns
Maximum combinational path delay: No path found
```

The total MIPS for our system is 37.968, the same as the maximum clock frequency because of our single-cycle architecture.

Test Strategy:

Our test strategy involves testing each sub-module via a dedicated test bench and do file. After the operation of each individual module was verified, we tested all the modules by running simpler assembled code. We would watch the propagation of the code by watching where the program counter moved and by looking at the contents of the registers to verify that the code was working and to diagnose errors. After we verified that our processor was working correctly, we ran our assembled fibonacci code and verified that the results of the register file were correct.
The modules we reused from previous work were the ALU and Register File which we had tested rigorously when they were created.
We also performed two more elaborate, well documented tests.

Instruction Decoder Test Bench:

a. You can activate the test by running doInstructionDecoder.do. You can verify that the processor has passed this test by cross referencing all of the control flag values (seen in waveforms) with the values found in the control flag spreadsheet.

b. This test works by sending in a slew of instruction words that use every supported op code. This will cause every possible permutation of control flags.

c. This will catch an error with the LUT not working, because the output flags will be independant from the input instructions, and therefore won't match the table. It can also catch errors with an incorrectly populated LUT, which would cause errors later with only the instructions that were not correct, and therefore be very difficult to catch otherwise. This will be seen as a values that doesn't match the spreadsheet. However, this test will not catch flaws in our logic when decided how to set the flags which was an issue we ran into a few times that was difficult to diagnose.

Instruction Fetch Unit Test Bench:

a. Run doInstructionMemory.do, verify that the PC waveform matches the following sequence:
   PC: 0, 1, 2, 0, 1, 9, 10, 1, 2, 2, 3, 4, 5, 9, 10, 14, 15

b. The test tries out all the different ways the program counter can change. First it allows it to increment by one command at a time (the default behavior). It also does this in between each of the following. It does Jumps with two different immediates. Then it does JumpRegisters with two different values in the $ra register. Then it does Branches with all of the different arrangements of Zero and InvZero.

c. This test could catch a design error where it Branches when the branch check evaluates to false. It also could catch off-by-one(or more) errors with the Jump, JumpRegister, and Branch operations.