RISC-E (reduced instruction set computer eric)

NOTE: This is a revised version of the proposal with additional documentation on how to test the final design. Project checkpoints are described in this document.

Overview:

- This processor will be von Neumann architecture (unified instruction and data memory).
- This processor will be around the complexity of the intel 4004 or 4008 but with a reduced instruction set. It is a simple processor.
- The opcode is 4 bits and the ISA has 16 instructions.
    - This includes LOAD/STORE/ALU Ops and control flow
- For testing: I will load instructions into the instruction memory for basic arithmetic. The processor will then run the instructions. The result will be visible in the data memory (off chip).
- The primary engineering challenge will be to interface memory with the CPU since we only have 12 data inputs and 12 data outputs.
- 12-bit data word & instruction word. 10-bit memory address.
- Sixteen 12-bit registers
- The off-chip memory will have 2^10 locations. Each location will be 12 bits.
- Memory for instruction and data will be unified. There will be no caches.
- The processor will be multi-cycle (3 stage). This will allow access to the memory for instruction fetch in the first stage and data read/write access in the later stages.
    - For the first RTL version, only one instruction will flow through the pipeline at a time.
    - If developer time/transistor budget allows, multiple instructions will flow through the pipeline simultaneously.
- There will be flags that are set by the ALU operations. These can subsequently be used in the branch operations

ISA Specification:

Format	Assembly syntax	Semantics
0000 xxxx xxxx	HALT	Ends execution
0001 dddd ssss	LOAD Rd, Rs	Rd ← MEM[Rs[9:0]]
0110 dddd ssss	STOREL Rd, Rs	MEM[Rs[9:0]][5:0] ← Rd[5:0]
0111 dddd ssss	STOREU Rd, Rs	MEM[Rs[9:0]][11:6] ← Rd[11:6]
1000 dddd ssss	ADD Rd, Rs	Rd ← Rs + Rd
1001 dddd ssss	SUB Rd, Rs	Rd ← Rs - Rd
1010 dddd ssss	AND Rd, Rs	Rd ← Rs & Rd
1011 dddd ssss	OR Rd, Rs	Rd ← Rs | Rd
1100 dddd ssss	XOR Rd, Rs	Rd ← Rs ^ Rd
1101 dddd ssss	SL Rd, Rs	Rd ← Rd << Rs
1110 dddd ssss	SRL Rd, Rs	Rd ← Rd >> Rs [Logical]
1111 dddd ssss	SRA Rd, Rs	Rd ← Rd >> Rs [Arithmetic]
0101 dddd ssss	NOT Rd, Rs	Rd ← ~Rs
0100 dddd iiii	LI Rd, IMM	Rd ← {8'h0,IMM}
0010 dddd xxxx	JUMP Rd	PC ← PC + 1 + Rd
0011 o iii iiii	o=0: BRANCHz IMM	If z flag set: PC ← PC + 1 + SEXT{IMM}
	o=1: BRANCHp IMM	If p flag set: PC ← PC + 1 + SEXT{IMM}

Flag	Function
z	Previous ALU op result was zero
p	Previous ALU op result was positive

An ALU op is if the MSB of opcode is 1 or if it is a NOT instruction.

Register File:

Sixteen 12-bit general purpose registers. Dual port read (needed to read both operands for ALU ops). Single port write.

I/O Interface:

All 12 inputs and outputs from the chip will be connected the memory (which will be on the FPGA).

External Memory Interface:

Name	Dir	Width	Purpose for Read	Purpose for Write
addr_data	in	10	Addr	Cycle1:addr, C2: data
read_write	in	1	Read when == 1	Write when == 0
write_commit	in	1	Must be zero	Signifies 2nd cyc of write
mem_result	out	12	Read Result	X

*Direction is from memory's perspective

If read_write and write_commit are both one, this indicates a HALT condition (HALT instruction reached).

Memory Read Protocol:

> The CPU will place the desired address to read on the addr_data bus. The CPU will set the read_write line to 1. The write_commit is a don't care. The mem_result will have the data stored at that memory address within the same clock cycle.

Memory Write Protocol:

> First Cycle: The CPU will place the desired address to write on the addr_data bus. The CPU will set the read_write line to zero. The CPU will set the write_commit to 0.
>
> Second cycle: The CPU will place the data to write on addr_data[5:0]. addr_data[6] will be set to 1 if we are writing the top half of the data word and zero for the bottom half. addr_data[9:7] are don't cares. Write commit will be set to 1. The data is now written to the memory.

External Hardware:

Laptop with python program ⇔FPGA(UART+RAM)⇔CPU Chip

On the host FPGA, there will be both a unified instruction/data memory as specified above. There will need to be a basic state machine to handle the multi-cycle write operation. The host FPGA will also have a basic UART interface to a host PC to read and write to the data memory from the PC. A basic python program will run on the host PC to make programming and viewing the results of the instruction memory easier.

Diagram of CPU:

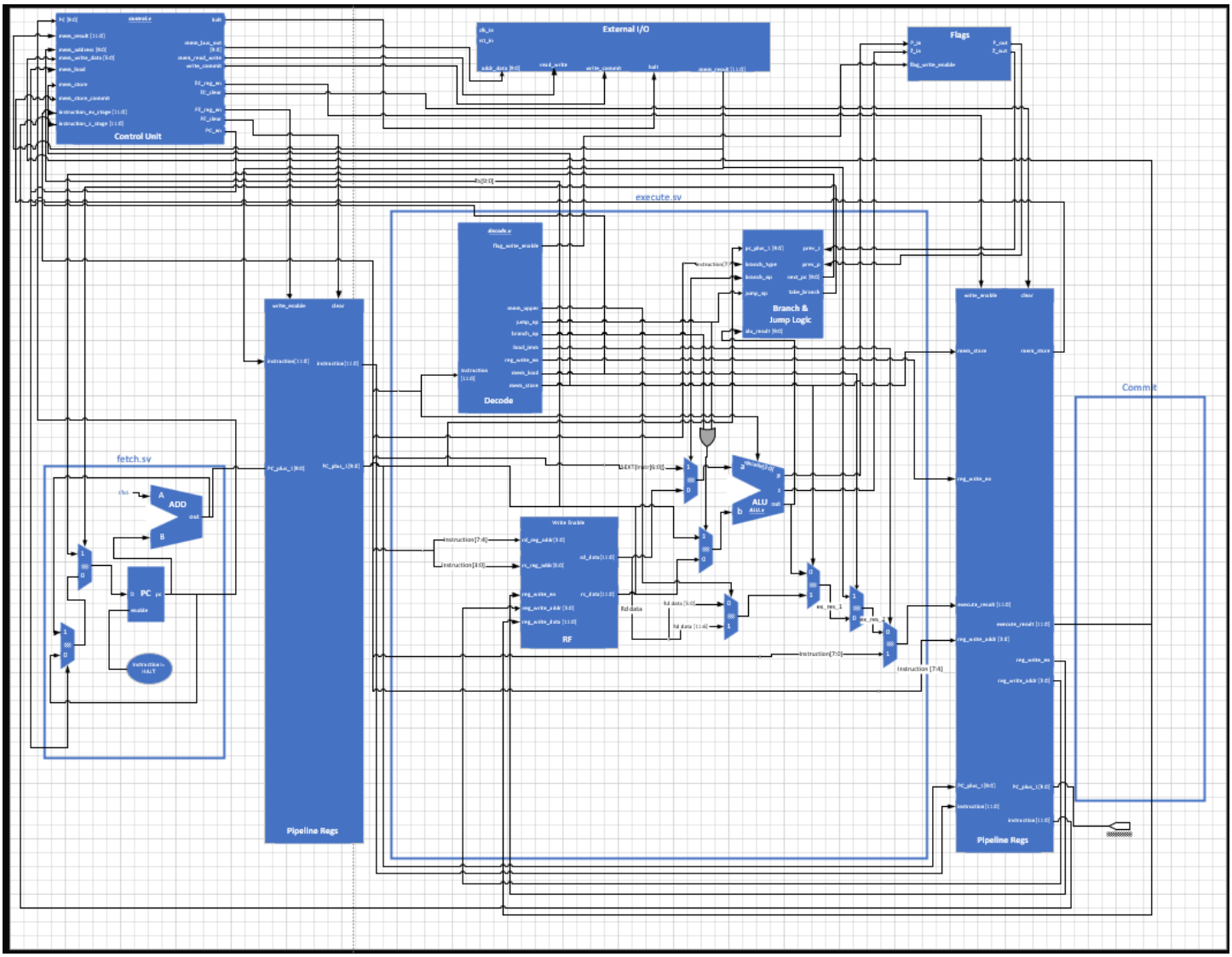
See copy at end of document to zoom in better. This is a proposed diagram and is subject to change.

Updates for milestone 1:

1. I accidentally pipelined the branch taken & next_pc through the second set of pipeline registers in the first version of my schematic. This would add an extra cycle of delay on branches without reducing the critical path. This issue is now fixed.
2. I added a synchronous clear to each set of pipeline registers. This signal will insert a NOP (OR R1, R1) on the next clock edge when set to 1.

Updates for final project submission: various bug fixes including updating the branch/jump logic, pipelining pc through the final pipeline register for debugging in simulation.

See the schematic file in the repo to see this file in more detail.



Pipelining Specifics:

In the first cycle, the processor will fetch the instruction from memory

In the second cycle, the processor will compute the ALU operation or read/write memory.

In the third cycle, the processor will complete the operation and fully store the result into memory or the register file. This is when the “write-commit” step will happen for memory write.

In the first version of the RTL, the processor will not fetch the next instruction until the previous instruction has completed all stages of the pipeline. The pipeline stages do not improve throughput but simply allow for interfacing with the multi-cycle memory in the planned version.

If gate count allows, I will add more optimized pipelining. For example, if it is an operation with no memory access (after fetching the instruction), the pipeline does not need to stall at the memory stage to commit the memory op.

Details about this extension:

- Add logic to accomplish this inside the “control” module. More inputs need to be added to the control module to detect when there are read-after-write or control hazards.
- The two sets of pipeline registers would need a “flush” functionality added (to insert NOPs)
- For non-memory operations, we could see throughput improvements from implementing true pipelining (as we could fetch the next instruction from memory while computing the previous)
- Register File Bypass would also be added.

Discussion:

- Why do I have a write commit signal? In the case that the CPU and memory aren't clocked at the same frequency, we need a way for the memory (which will be on the FPGA) to know that the data to write at the requested location is valid. The data is valid when write_commit goes from 0->1.

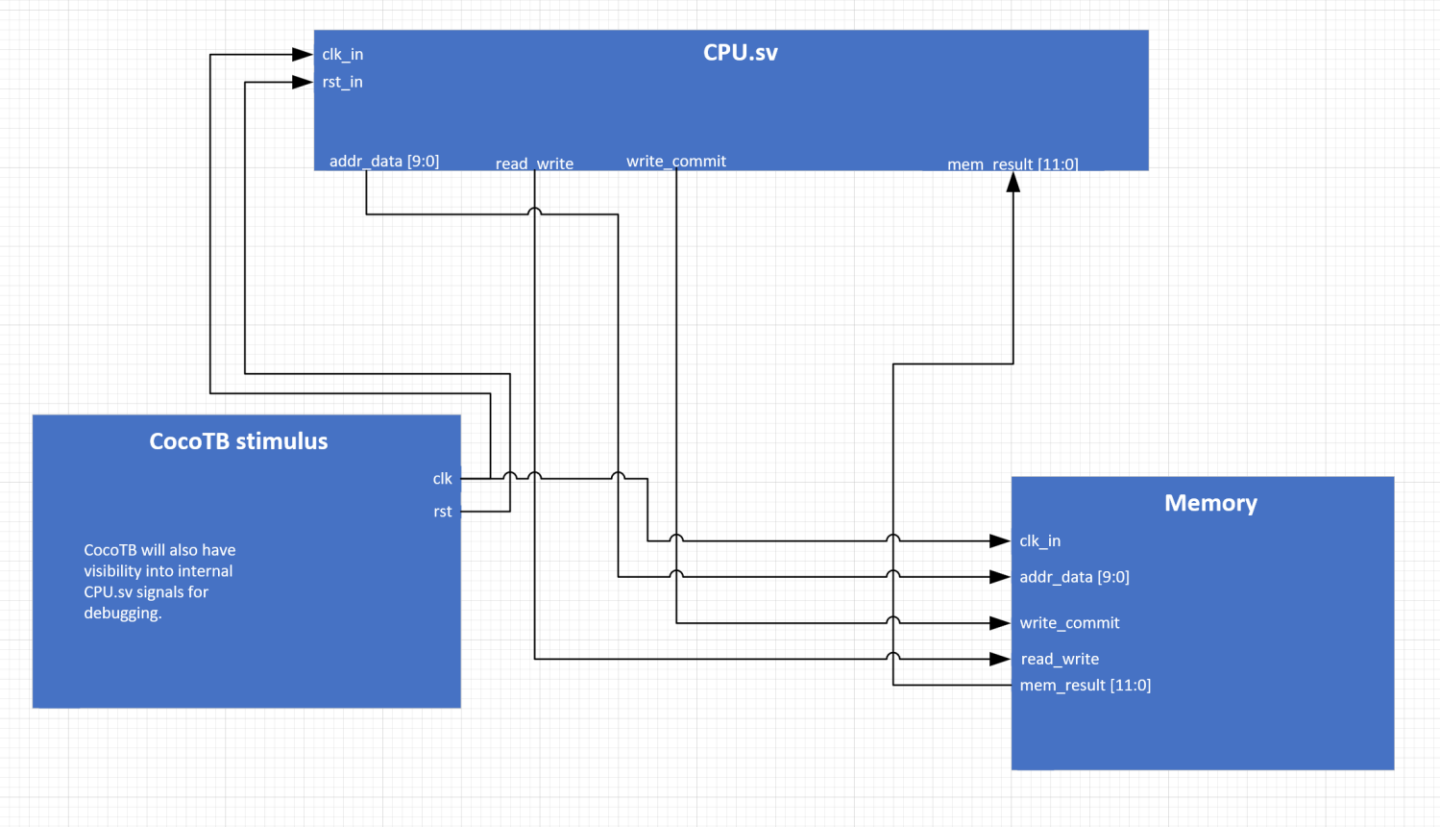
- 12-bits is an unusual word size, but I made this choice because then the entire word can be loaded from memory in one cycle (since for the tape out we can only have a max of 12 input pins to the CPU). This is crucial because otherwise each instruction fetch would take multiple cycles.

- This will be a very basic ISA, just enough to be Turing complete, yet also not too painful to program. The full ISA specification is in the table above. I want to make my own ISA because the 4004 has been done many times before. I think it will be more fun and educational to design my own from the ground up, even if it isn't quite as optimized as the 4004 or 8008.

RTL Milestone: For the RTL milestone, I had a fully implemented CPU. Verification was not complete at this time, though. Bug fixes to the RTL were required during the verification stage of the project.

Testing (new for project milestone 1):

The overall testing setup will be as follows:



To generate test program traces, I wrote a basic assembler in python. So, I am able to write basic program traces in my assembly language and get the output that will be loaded into memory.

For the final verification milestone, I also added automatic test generation (see automated_test.py)

Testing is primarily done with functional simulations (usually the setup would be a SystemVerilog testbench that is run in VCS/questa sim). For this class, I will do the functional simulations using cocoTB. The cocoTB testbench itself is relatively basic with supporting files to check correctness.

For **functional verification**, I completed it in two phases.

First phase: Used very simple test programs (10-20 instructions). Load the output from the assembler into memory. Start the CPU (release reset). Let the CPU run until halt is reached. Manually check the final state of the memory to make sure it matches what you expect. See the testbench/tests/handwritten_tests directory.

Second phase: (out of scope of original proposal, but I completed it): I coded up a golden model of my CPU in python (in the real world this is usually in c++). This golden model will not be cycle accurate, but

will result in the same final state in the memory. To check if an instruction trace ran successfully, we just need to check if the final state of the memory connected to the DUT is the same as the final state of the memory connected to the golden model.

Although I did not include it in my original specs, for debugging purposes, I also added memory and register read/write trace tracking for both the golden model and the cpu in simulation. For a test to pass, the final state of the memory and output traces must be the same. All 74 tests pass in simulation.

Last, I added primitive **emulation on the FPGA**. This level of testing wasn't my original intention, but I had extra time to add FPGA emulation.

The testbench will have the UART load up the memory with the machine code, the cpu will run, and then the memory data is read out over UART and checked against the golden model.

Note: I needed to use an off-the-shelf UART module (from a prior exercise) to communicate from the PC to the FPGA to set the initial state of the memory, start the CPU, and inspect the final state of the memory.

The memory System Verilog code on FPGA is different to assure that the memory properly synthesizes into DSPs within the FPGA instead of on the FPGA fabric.

The FPGA emulation is not comprehensive and is still a work in progress. To provide more meaningful results, a more robust debug solution (ex: UART prints for each register write) would need to be implemented. This was crucial for debugging in simulation. Unfortunately, this is out of scope for this project.

How to run tests:

To test the design, go to src/ and then run bash test_all.sh. You will need to be in an OSS-CAD environment (following 18-224 setup)

To view the waveforms, run bash view_waves.sh

You can write your own testcases in assembly and then run them using automated_test.py -asm <file_name>

To run a test on the fpga, program the fpga. Go to testbench directory, run build.sh, make sure the FPGA is programmed, and then run automated_test.py -asm tests/handwritten_tests/add_test.asm -e