

A Modified RISC Processor

Alessandro

Daniel
William

Harshdeep

Problem statement and motivation

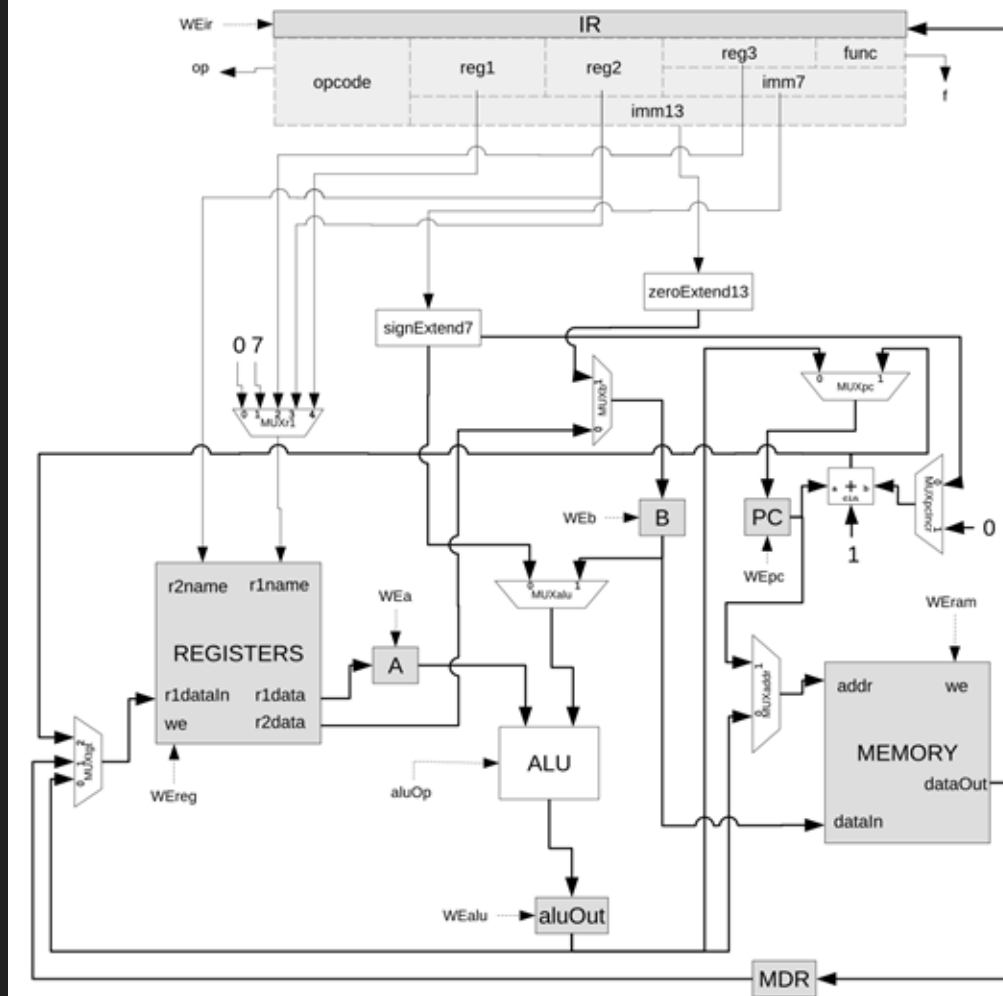
- Make a modified RISC architecture learned from our Comp Org/Arch class
- Slightly different from common RISC architectures commonly taught in a typical class - historically these have been MIPS, and as of late, RISC-V
- Motivation: use this hackathon as an opportunity to learn to use a new GenAI-based EDA tool to iterate upon a chip architecture for educational purposes, evaluate feasibility of making such a chip

Design methodology

- Previously created a simulator for our class in C++
- Use this as our golden model
- Lean towards behavioral implementation rather than structural - prioritize readability for the designer
- Iteratively test, prompt Cognichip, check results
- Use automation to run testbench (shell scripts)
- Compare expected (as previously manually determined) vs actual results
- Revise - incremental/iterative development, debugging

Architecture / RTL description

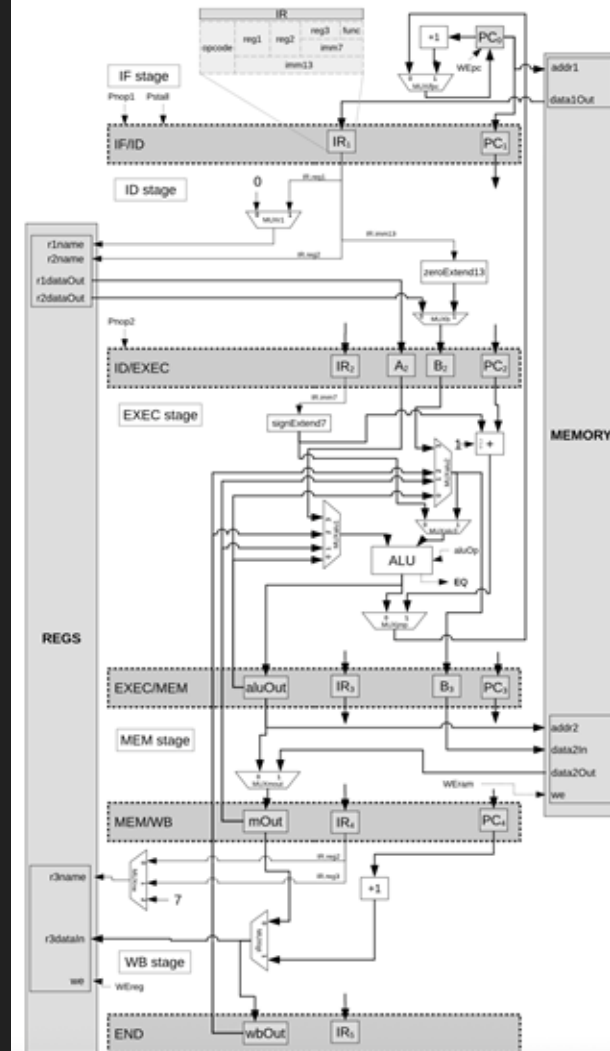
Single-cycle implementation



Architecture / RTL description

Standard 5-stage pipelined processor

IF, ID, EX, MEM, WB



Courtesy of E20 Manual

Simulation results

- Testbenches scan through assembly programs to run
 - Tested with basic programs to see if instructions work together
 - Fibonacci, array sum, basic tests
-
- basic arithmetic and logic operations
 - loops
 - control flow
 - memory access
 - subroutines - makes recursion possible!

Performance discussion

- Single-cycle vs pipelined implementation
- In practice:
 - Certain instructions will run slower based on how the hardware is wired
 - Single-cycle has a bottleneck on cycle time
 - Pipelined instructions rely on microinstructions - can be 2 to 3 times faster: effective CPI can substantially increase
 - This matters when programs scale

Challenges

- Overall processor design seemed to be a relatively trivial task for Cognichip, but occasional debugging still required
- Imperfections of AI, sometimes requires active feedback even when it auto-runs commands in terminal and corrects mistakes in feedback loop
- Pipelined version of processor was difficult to get right - flushing instructions
- Halting instructions, “two-tiered” halt detection to prevent infinite loops - two consecutive halts as pseudo-instructions equate to a true halt

Lessons learned

- Importance of knowing what the requirements/specifications are (instruction set, control signals, architecture description)
- Knowing how to verify and interpret results
- Can do a better job matching architecture diagram (emulating hardware better), using control signals, making the actual hardware-related behavior more obvious

Lessons learned

- Helps make a more “bona fide” / synthesizable processor
- Consider using Register Transfer Level (RTL) to better (clearly) describe data flow between registers
 - Also cycle-accurate: relevant for a processor

Future work

- Verilog synthesis - revise architecture to make it synthesizable
- Tapeout?
- Architecture types
 - Single, shared memory and bus for both data and instructions, leading to simpler, lower-cost designs but potential speed bottlenecks (von Neumann)
 - Physically separate memories and signal pathways for data and instructions, allowing simultaneous access, higher performance, and faster execution speed, making it ideal for microcontrollers (Harvard)
- Adding more instructions, slightly increasing in complexity, more supporting immediates, optimizing for more complex arithmetic instructions like multiplication and division & remainder

References, Links

- CS-UY 2214 E20 Manual
- <https://github.com/wl2491/modified-risc-processor>