RISC - V SIMULATOR FINAL REPORT

Ria Mahajan, Mitch Montee, Bhavana Srinivasegowda, John Yang

Design Summary: -

The RISC-V simulator implementation in C includes core logic for executing instructions resides in instructions.c, which handles arithmetic, logical, memory access, and control flow instructions, effectively implementing the RISC-V instruction set. The decoding process is managed by decode.c, which extracts critical information such as opcodes, function codes (funct3 and funct7), register indices, and immediate values from binary instruction formats. This decoding ensures the correct interpretation of various RISC-V instruction types like I-type, R-type, B-type, U-type and S-type. Memory operations are centralized in memory.c, which provides functions for reading from and writing to memory, handling byte, half-word, and word data sizes while ensuring proper alignment and address calculation. The main simulation logic is contained in main.c, which acts as the entry point for the simulator. This file is responsible for parsing command-line arguments, reading input files, and managing the overall fetch, decode, execute cycle. It also controls the PC to ensure proper instruction flow. To support the implementation, defines.h and types.h define key constants and data types for improved code readability and maintainability.

For extra credit, we implemented the ecall instruction to support read, write, and exit system calls. The system call number in a7 determines the function: read (63) reads input into a buffer, write (64) outputs data from a buffer, and exit (94) terminates the program. Arguments are passed through a0 to a2, and results are returned in a0, with -1 indicating errors for read and write. This implementation enhances the simulator's I/O and termination handling. There are also optional compile flags that allow for register, instruction and memory monitoring and the ability to single step execution.

The RISC-V simulator was tested by writing assembly (.s) files that cover various instruction scenarios for each instruction in the RV32I ISA . These assembly files were assembled into .mem files, which were then provided as input to the simulator for execution. The output of the assembled test field is then examined to evaluate the against the expected result. Detailed verification procedures, covering corner cases are outlined in the verification plan. Unit tests are used to verify subsystems for memory access and parsing inputs files.

Commands: -

1. Set the RISC-V toolchain path and create aliases:
   export RISCV=/pkgs/riscv-gnu-toolchain/riscv-gnu-toolchain-elf
   alias rv32as=$RISCV/bin/riscv32-unknown-elf-as
   alias rv32objdump=$RISCV/bin/riscv32-unknown-elf-objdump
   alias rv32gcc=$RISCV/bin/riscv32-unknown-elf-gcc
   alias rv32ld=$RISCV/bin/riscv32-unknown-elf-ld

2. Assemble, link, and generate a .mem file with RISC-V toolchain:

```
rv32as test.s -o test.o
rv32ld -Ttext 0x0 test.o
rv32objdump -d | ~faustm/bin/rv32pp > test.mem
```

3. To build use: make all or make main

*Compile time argument flags: make main EXTRA_CFLAGS="-DSTEP -DDEBUG -DVERBOSE"

-DVERBOSE -- Enable verbose output.
-DSTEP -- Enable instruction stepping with key press.
-DDEBUG -- Enable debug output and define VERBOSE

4. Run simulator example: ./main -f ./input_files/Btype/BGE.mem -s 128-a 0

*Flags for runtime command:

-f <input file path>
-s <stack Address> -- Starting stack address(Default = 65535)
-a <starting address> -- Word address(Default = 0)
-ft <input file type> -- 0 = sequential index, -ft 1 = word address index(Default = 0)
-sp <input file spaces> -- The number of spaces following the colon(Default = 2, assembler does 7)