ECE 586 Project Report
RISC-V ISA Simulator
Group 10

Bhoomika Mohan(bhoomika@pdx.edu)
Julia Filipchuk(bfilipc2@pdx.edu)
Mansi Dhanrajani(mansi2@pdx.edu)
Sanchita Leekha(sleekha@pdx.edu)

Design

The RISC V ISA simulator simulates the RV32I Base Integer Instruction Set. The project consists of a C program ‘rvemu’ containing the emulator and supporting Makefiles and test cases used to verify functionality. We utilize command-line flags to ‘rvemu’ to enable different levels of logging information as well as toggling features of execution. Below is a summary of the command-line interface to the emulator.

```
Usage: rvemu [options]
   or: rvemu            # Load program.mem and run in silent mode.
   or: rvemu -v         # Load program.mem and run in verbose mode.
   or: rvemu -d         # Load program.mem and run in debug mode.
   or: rvemu -i <file>  # Load <file> and run in silent mode.

Options:
  -v --verbose       Enable verbose mode. Display hex of each instruction,
final PC.
  -d --debug         Enable additional debug information.
  -r --echo-mem      Display all lines read from memory file.
  -i <file>, --input <file>
                     Read mem file instead of program.mem.
  -s <addr>, --start_addr <addr>
                     Set starting PC address for program counter.
  -S <addr>, --stack_addr <addr>
                     Set starting SP address stack pointer.
  --step             Step through the code instruction by instruction.
  --show_details     Show additional instruction details.
```

Logging of result deliverable is done to standard output with additional debugging information always logged to standard error. Additional details provided for some instructions displays extra information showing the values the instructions are working with and the results of operations.

We utilize the ‘stdint.h’ include to maintain consistently sized integer data types as well as some convenience typedef values ‘u32, s32, u16, u8’ to clearly express the underlying data we are working with.

Memory Model

We represent our address space as a union of arrays. This gives us convenient access to the underlying memory as words, half-words, or individual bytes. We provide functions to access the memory as specific values as well.

```
union memory_model {
    u08 bytes[1 << 16]; // View as bytes.
    u32 words[1 << 14]; // View as words.
    u16 hwords[1 << 15]; // View as half words.
};

u08  get_memb(u16 addr);
u32 get_mem(u16 addr);
u16 get_memh(u16 addr);
void save_mword(u16 addr, u32 reg_val);
void save_mhw(u16 addr, u16 reg_hval);
void save_mbyte(u16 addr,u08 reg_bval);
```

Register Model

We represent our register by default as an array of 32 bit signed values. We provide an enumeration of convenient register names to allow direct addressing by conventions such as *ra* and *sp* conventions. Functions are provided to write and read values and maintain x0 as an always zero register.

```
enum regs_indexs {
    ZERO, RA, SP,  GP,  TP, T0, T1, T2,
    S0,   S1, A0,  A1,  A2, A3, A4, A5,
    A6,   A7, S2,  S3,  S4, S5, S6, S7,
    S8,   S9, S10, S11, T3, T4, T5, T6,
};

extern s32 regs[32];           // 32 registers

u32 get_reg(u32 addr);
s32 get_regsign(u32 addr);
void save_reg(u32 addr,u32 value);
```

Instruction Views

Instructions are checked based on length specification. Only 32 bit length instructions are supported at this time. We utilize a union with alternate views of each instruction encoding type. Structs with bit fields that are explicitly ‘packed’ to prevent unintended optimizations provide access to all parts of encoded instructions.

```
union riscv_inst32 {
    u32 inst;

    struct __attribute__((packed)) {
       u32 opcode : 7;
    } any;

    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 rd     : 5;
       u32 funct3 : 3;
       u32 rs1    : 5;
       u32 rs2    : 5;
       u32 funct7 : 7;
    } Rtype;

    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 rd     : 5;
       u32 funct3 : 3;
       u32 rs1    : 5;
       u32 imm    :12;
    } Itype;

    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 imm_l  : 5;
       u32 funct3 : 3;
       u32 rs1    : 5;
       u32 rs2    : 5;
       u32 imm    : 7;
    } Stype;
```

…***contintued***

```
    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 rd     : 5;
       u32 imm    :20;
    } Utype;

    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 rd     : 5;
       u32 imm12  : 8;
       u32 imm11  : 1;
       u32 imm01  :10;
       u32 imm20  : 1;
    } Jtype;

    struct __attribute__((packed)) {
       u32 opcode : 7;
       u32 imm11  : 1;
       u32 imm01  : 4;
       u32 funct3 : 3;
       u32 rs1    : 5;
       u32 rs2    : 5;
       u32 imm05  : 6;
       u32 imm12  : 1;
    } Btype;
};
```

Program Counter & Instruction Decode

We keep a separate value for the program counter and the target of the next program instruction. Jump and branch instructions are free to update *pc_next* when branch will be taken. The *pc* value is then set to the *pc_next* value when the instruction is done with execution.

```
u32 pc;                 // Program Counter.
u32 pc_next;            // Program Counter next value.
```

To decode instructions we utilize masks for the relevant bits and test vectors to match an instructions unique setup. In hindsight this was too complicated and did not provide clear ways to add new instructions to the code. An alternative method using more straightforward switches on opcodes, func3, and func7 opcodes is present but not complete.

Testing

Arithmetic Operators: The arithmetic operators are ADD, SUB, and ADDI operation which considers the sign of the input for its functionality others and does the operation depending on the sign
For example Subtraction of a positive and a negative number should yield an addition of two inputs.
The addition of a positive and a negative number should yield the subtraction of those two inputs, thus considering the sign of these functions becomes very important.
The addition of two numbers can also yield a negative number

1) If both the inputs are negative
2) If one of the inputs in negative value is greater than the other inputs positive value

Thus these operations consider the sign very carefully and do the operation.

Testing strategy:

- To check the functionality of this function, different combinations of positive and negative values are loaded, all 4 combinations are checked and combinations of greater and smaller numbers are also checked, thus covering all the cases.
- Tried to write to register R0

Logical Operators:
The other operations are XOR, OR, AND, XORI, ORI, ANDI in case of positive inputs it just does the operation normally but when given a negative input, it will take the two's complement of it and do the operation normally.

Testing strategy:

- To check the functionality of this function, different combinations of positive and negative values are loaded, all 4 combinations are checked.
- In the case of logical operators even though a negative value is given it takes its 2's complement and does an unsigned operation.
- Tried to write to register R0

Shift Operators:
SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. SLLI and SRLI perform shifts by using shift amounts from the immediate field. The simulator displays an error statement if the shift amount is negative.

Testing Strategy:

Set Less than instructions:
SLT and SLTU perform signed and unsigned comparisons respectively. SLTI performs a set less than operation and treats both the operands as signed. SLTIU is similar but performs the operation by treating the value of both inputs assigned.

Testing Strategy:
Setup an initial set of values to test against covering negatives, zero, maximum and minimum. Test cases where the LT results in 0 in t6 watching for 0 in that register. Test cases where the LT results in 1 in t6 watching for 1 in that register.

Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

Loads:
The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are used for loading a 8 bit signed and unsigned value from memory into a register.

Stores: needs to read two registers, rs1 for a base memory address, and rs2 for data to be stored, as well as need immediate offset value.
The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

Testing Strategy:

The test cases generated covered the following cases:
- Testing for zeroth register
- Invalid Instructions
- Signed and unsigned values for some of the R and I type instructions
- Testing for misaligned instructions

Arithmetic and logic instructions:
For R and I type instructions, the test cases cover two positive numbers, one positive number, and the other negative, both the numbers as negative. Writing the value to the zeroth register pops an error.

Branches:
BLT and BLTU: (i) compare two registers a0 and a1 having values 1 and -1, in this case, it shouldn't branch on BLTU (ii) BLT a0, a1 when a0 is having value 1 and a1 5, It should branch on BLT
BGEU: BGEU a0, a1 for a0 = -1 and a1 = 1 should be taken.
BGEU: BGEU a0 , a1 for a0 =
BGE: BGE a0, a1 for values of a0 = 5 and a1 = 4 should be taken.
BGE: BGE a0, a1 for values of a0 = -1 and a1 = 1 should not be taken
BGE: BGE a0, a1 for values of a0 = 4 and a1 = 10 should not be taken
BEQ: BEQ a0, a1 for values of a0=1 and a1 =1 should be taken.
BEQ: BEQ a0, a1 for values of a0 = -1 and a1 = 1 should not be taken

Jumps:
JALR x5, x4, 16 should store PC + 4 at x 5, and PC should point to x4 + 16
JALR will terminate when it gets an instruction with all zeroes
JAL x5, 16 should store PC + 4 to x5 and PC should be updated to PC + 16
JAL will just stall when it gets an instruction with all zeroes and it continues with the next instruction.

Testing Strategy:
JALR and JAL were checked with a negative offset and positive offset
A branch to where the instruction is all zeroes

Results

We produced the working emulator. Implementing all the required instructions. We implemented a step execution mode (enabled with '--step-' command line arg) to allow stepping through code. We were not able to implement system calls or a more complete debugger.

Our source and implementation is available on:
https://github.com/jbfil/ece586w22t10-riscv-emulator

Testing was very slow and tedious and involved a large amount of manual register analysis. We would benefit greatly from implementing an 'assert' system call to check conditions on a value. Structuring test cases as a progressing series of executions and error causing operations (such as instruction 0x00000000) would be advised for future test construction.