CSC3050 Project 1 MIPS Assembler & Simulator

Yongjin, Huang 119010115

Introduction

This project implements a MIPS assembler and simulator in a single `simulator.cpp` file. The assembler and the simulator are designed separately so that each of them is supposed to work independently.

Features & Tricks

Nested class

Wrap `Scanner` and `Parser` into `Assembler` using nested class, which enhances code reusability.

```
class Assembler
{
  public:
    inline static vector<string> data_seg;
    inline static vector<string> text_seg;
    inline static vector<string> output;
    class Scanner
    {
      Assembler &assembler;
    };
    class Parser
    {
      Assembler &assembler;
    };
}
```

Process data segment in `Assembler`

Data segment is interpreted to machine code in `Assembler` since `Simulator` should only perform simulation but not assembling.

Hash table mapping machine code to function pointer

Hash table `unordered_map<string, function<void(const string &)>>` can replace numerous conditional statements (`if` `else if`).
In its generating function `gen_opcode_to_func` , `std::bind` should be used to locate the function pointer of the current `Simulator` instance. e.g.

```
// in void Simulator::gen_opcode_to_func(unordered_map<string,
function<void(const string &)>> &m)
m.emplace("000100", bind(&Simulator::instr_beq, this, placeholders::_1));
```

Thus, the code is much more neater when calling the corresponding function.

```
// in void Simulator::exec_instr(const string &mc)
if (opcode == R_opcode[0] || opcode == R_opcode[1])
{
    // R instructions
    string funct = mc.substr(mc.size() - 6, 6);
    auto it = opcode_funct_to_func.find(opcode + funct);
    if (it == opcode_funct_to_func.end())
        signal_exception("function not found!");
    (it->second)(mc);
}
```

Test multiple cases with makefile

With this `makefile`, I can compile and test multiple cases by a single command `make`.
Take `sim_test` for example, which tests both assembler and simulator, it will echo fail message and exit when the current case fails.

```
sim_test: $(PROM)
    for t in $(SIM_TESTS); do \
        ./$(PROM) $(TEST_DIR)/$$t.asm $(TEST_DIR)/$$t.in $(TEST_DIR)/$$t.out
2>&1; \
        diff -q $(TEST_DIR)/$$t.out $(TEST_DIR)/$$t.simout > /dev/null || \
            echo "Test $$t failed" && exit 1; \
    done
    echo -e "All simulator tests passed!\n"
```

Tricks

- Use fixed width integer types such as `int32_t` instead of `int`.
- Get the maximum possible number of a data type, e.g. the maximum number for `int16_t` is `numeric_limits<int16_t>::max()`
- Conversion between binary string `std::string` and integer `int32_t`
  - int to binary string: `bitset<str_length>(int32_t_number).to_string()`
  - binary string to int: `stoi(str,nullptr,base=2)`
- Detect overflow by `__builtin_add_overflow`, `__builtin_sub_overflow`, `__builtin_mul_overflow`.
- `inline static` Declare and initialize static member variables together in class.
- `static const` Declare and initialize constant member variables in class.
- `#define`, `#ifdef`, `#endif` Use C preprocessor to help debug.

Implementation

There are two classes in the program: `Assembler` and `Simulator`.
`Assembler` takes MIPS assembly code as input and output machine code in the format of `vector<string>`.
`Simulator` takes the output machine code from `Assembler` and executes the simulation.
Example usage of two classes:

```
// input stream and output stream should be passed.
Assembler assembler;
Simulator simulator(assembler.output, cin, cout);
assembler.scanner.scan(cin);
assembler.parser.parse();
simulator.simulate();
```

Assembler `Assembler`

Scanner `Assembler::Scanner`

- `void remove_comments();` Remove comments, empty lines and tabs `\t`
- `void split_data_and_text();` Split data segment `.data` and text segment `.text` and it can handle multiple occurrences of `.text` and `.data`
- `void preprocess_text();`: preprocess data segment
  - Put label and its corresponding code together in the same line
  - Replace `,` with space ` `
  - Throw tabs `\t`
- `void scan(istream &in);` Main process of `Scanner`

Parser `Assembler::Parser`

- `void process_dataseg();` Interpret data segment to machine code:
  - Based on different data types perform different operations:
    - `asciiz` `string get_ascii_data(const string &data);` Handle special characters like `\n` , `\t` , `\'` , `\"` , `\\` , `\r` . Append `\0` terminator to its end.
    - `ascii` The same as `asciiz` except for `\0` terminator.
    - `word`
    - `half`
    - `byte`
  - Append zero or truncate to generate fixed-length (32bits) machine code
- `void find_label();` Locate all labels in the text segment and store them using hash table `unordered_map<string, uint32_t> label_to_addr` which maps label to the address of the code.
- `void parse();` Main function of `Assembler::Parser`:
  - Interpret each line of code in the data segment into machine code
  - Based on different types of instruction, certain interpretation methods are performed:
    - R instructions
    - I instructions
    - J instructions
    - O instructions (`syscall` only)
  - And in each type of instructions, instructions of the same format are grouped together
    - e.g. `sll` , `srl` , `sra` are in `op rd rt shamt` so they are grouped together
  - Concat string to generate machine code

Simulator `Simulator`

- Member:

    - Memory structure:
        - ```
          stack_st_idx = 6 * 1024 * 1024 = 6MB
          | <- stack data
          stack_end_idx
          |
          dynamic_end_idx
          | <- dynamic data
          dynamic_st_idx = static_end_idx
          | <- static data
          static_st_idx = 1 * 1024 * 1024 = 1MB
          |
          text_end_idx
          | <- text data
          text_st_idx = 0
          ```
    - Basic data types:
        - `typedef array<char, 32> word_t;`
        - `typedef array<char, 16> half_t;`
        - `typedef array<char, 8> byte_t;`
    - 2D character array to simulate memory `array<byte_t, memory_size> memory;`
    - Integer array to simulate register value `int32_t reg[reg_size]`
    - Hash table `unordered_map`:
        - Map register code to its index in simulated array `unordered_map<string, size_t> regcode_to_idx;`
        - Map machine code to function pointer
            - `unordered_map<string, function<void(const string &)>> opcode_to_func;`
            - `unordered_map<string, function<void(const string &)>> opcode_funct_to_func;`
            - `unordered_map<string, function<void(const string &)>> rt_to_func;`
- Methods:
    - Retrieve or store data in memory:
        - `word_t get_word_from_memory(uint32_t addr);`
        - `half_t get_half_from_memory(uint32_t addr);`
        - `byte_t get_byte_from_memory(uint32_t addr);`
        - `int32_t get_wordval_from_memory(uint32_t addr);`
        - `int16_t get_halfval_from_memory(uint32_t addr);`
        - `int8_t get_byteval_from_memory(uint32_t addr);`
    - Map between memory array index and address:
        - `size_t addr2idx(uint32_t vm);`
        - `size_t idx2addr(size_t idx);`
    - Generate hash table `unordered_map`:
        - `void gen_regcode_to_idx();`
        - `void gen_opcode_to_func(unordered_map<string, function<void(const string &)>> &m);`
        - `void gen_opcode_funct_to_func(unordered_map<string, function<void(const string &)>> &m);`
          ```

        - `void gen_rt_to_func(unordered_map<string, function<void(const string &)>> &m)`
    - Store data segment to memory `void store_static_data();`
    - Store text segment to memory `void store_text();`
    - Perform MIPS operations:
        - `void instr_add(const string &mc);`
        - ……
    - `void simulate();` Main process of `Simulator`.
        1. Get current operation at `pc`.
        2. `pc+=4` increase program counter value.
        3. `exec_instr(mc);` Execute current operation.

How to run the program?

Just type `make` and it will automatically compile and test all cases under `./test` in `makefile`.

If you want to add test cases, you can either test them using CLI or add your files under `./test`:

1. CLI
    - `make` will run `g++ simulator.cpp -o simulator -std=c++17`
    - `./simulator <test>.asm <test>.in <test>.out` will run both assembler and simulator
    - `./simulator <test>.asm <test>.tasmout` will only run assembler
2. makefile
    - Required files to test Assembler:
        - `<test>.asm` MIPS assembly code
        - `<test>.asmout` sample output of Assembler, including both `.data` and `.text`, please refer to `.asmout`.
    - Required files to test both Assembler and Simulator:
        - `<test>.asm` MIPS assembly code
        - `<test>.in` input file
        - `<test>.simout` sample output file
    - Then add the test `<test>` to `ASM_TESTS` and `SIM_TESTS`:
        - `ASM_TESTS = 1 2 3 4 5 6 7 8 9 10 11 12 a-plus-b fib memcpy-hello-world <test>`
        - `SIM_TESTS = a-plus-b fib memcpy-hello-world <test>`

Conclusion

Through the developing process, I have learned lots of MIPS knowledge as well as C++ grammar.