ELEC2204 Computer Emulation in Software

Harry Beadle
27770834
hb11g15

February 13, 2017

Abstract

The design and implementation of a computer emulator written in C. Including a definition of a simple instruction set and assembler to allow the emulator to run generic programs. A wrapper is also written to allow the emulator's internal state to be observed though the terminal.

1 Architecture

The emulation consists of three modules:

- The Control Unit (CU)
- The Arithmetic Logic Unit (ALU)
- Memory

The CU controls the flow of data through the machine, the ALU does arithmetic and logic operations and the memory is used to store programs and data.

The processor is designed on a 16-bit architecture. Instructions contain a 4-bit opcode and one 12-bit or two 6-bit operands, which are pointers to addresses in memory. Which operands are used depend on the opcode.

`OPCODE`	`OPERAND A`	`OPERAND B`
	`OPERAND C`	

Figure 1: Graphic showing the two possible make ups of an instruction.

The instruction set used by the emulator consists of mathematical operations, logical operations, flow control and data management.

Table 1: Opcodes and their meanings.

Code	Abbreviation	Description
Mathematical Operations		
`0x0`	`ADD`	Add Operand A or B
`0x1`	`SUB`	Subtract Operand A from B
Logical Operations		
`0x2`	`AND`	Bitwise AND of A and B
`0x3`	`OR`	Bitwise OR of A and B
`0x4`	`NOT`	Bitwise NOT of A
Flow Control		
`0x5`	`JMP`	Jump to C
Data Management		
`0x6`	`STO`	Store the value in the accumulator in C

2 Memory

The memory is a generic synchronous 16-bit RAM modeled as an array of integers. It is connected directly to the Control Unit via an address bus (`addr`) and to the rest of the processor by the data bus (`data`).

The memory has a single control input, `memory_control`, which can take four different signals:

Table 2: Control signal names and descriptions for the ALU.

Signal	Description
`MEM_HIZ`	Don't drive the `data` bus.
`MEM_SET`	Set the memory at the `address` to the value of the `data` bus
`MEM_ENB`	Drive the `data` bus with the data stored at the `address`.

The memory is implemented in the `memory.c` and `memory.h` files. The function `updateMemory()` is called on each clock tick.

The size of allocated memory is determined by a `#define` called `MEMORY_SIZE`.

2.1 Testing

The memory is tested by setting each possible value to each cell in memory and then reading it back. This test can be found in `tests/memory_test.c`.

If the operation fails then the following data is output to `stdout` for debugging.

- The address being written to or read from
- The data being written or read
- The state of `memory_control`.

3 Arithmetic and Logic Unit

The ALU module takes two inputs, A and B, and produces an output based on a control signal `alu_control`.

In an effort to later simplify the control unit, the ALU's control input is just the opcode of the instruction, avoiding the need for a look-up-table. An additional signal `ALU_HIZ` is required to tell the ALU not to drive the data bus when it is not being used.

The ALU is implemented in the `alu.c` and `alu.h` files. The function `updateALU()` is called on each clock tick. It consists of a single switch case that switches on the `alu_control` input. The data bus is then driven with the output based on the inputs. Input buffering is handled by the control unit.

3.1 Testing

ALU testing is handled by `test/alu_test.c`. The test cycles through each of the operations with all possible inputs, then asserts that the output on the data bus is correct. If an output is not correct then the following debugging data is printed to `stdout`:

- The type of operation
- The two inputs
- The output

4 Control Unit

The control unit (CU) consists of registers and a state machine to control the flow of data through the CPU.

There are five registers in total. The `program_counter` and `instruction register` are used to control the CU itself, while the `accumulator` and `alu_buffer`s are used to store information for use in the ALU.

Table 3: A description of the purpose of each register in the control unit.

Register	Description
`program_counter`	Stores the address of the current instruction.
`instruction_register`	Stores the current instruction being executed.
`accumulator`	Used to store the output of the ALU.
`alu_buffer_a`	Buffer for input A of the ALU.
`alu_buffer_b`	Buffer for input B of the ALU.

Every instruction follows a basic flow through the control unit, however the specifics vary between opcodes. The basic flow follows these five steps:

1. The CU gets the next instruction from memory, based on the value of the program counter, and puts that into the instruction register.
2. The instruction register's value is bit shifted and pruned in order to provide the CU with values for the Opcode and Operands A, B and C.
3. The Opcode is then used to determine the sequence though which the CU must progress.
4. Any execution is performed.
5. The program counter is incremented and the process begins again.

More specifically, the CU follows the following state transition table. The system only depends on the Opcode input.

Table 4: States and descriptions for the Control Unit State Machine.

State[:Opcode]	Description	Next State
`DECODE`	Decode the opcode and decide which state transition to go though.	
`DECODE:JMP`	Set the program counter to operand C.	`GET_INSTRUCTION`
`DECODE:STO`	Store the value of the accumulator at address of operand C.	`INC_COUNTER`
`DECODE:ADD` `DECODE:SUB` `DECODE:AND` `DECODE:OR` `DECODE:NOT`	Get the value of operand A from memory.	`ALU_SETAGETB`
`ALU_SETAGETB`	Set the value of operand A from the data bus and get operand B from memory.	`ALU_SETBEXEC`
`ALU_SETBEXEC`	Set the value of operand B and start the exitution of the calculation in the ALU.	`INC_COUNTER`
`INC_COUNTER`	Set the accumulator and incriment the program counter.	`GET_INSTRUCTION`
`GET_INSTRUCTION`	Get the next instruction from memory.	`SET_INSTRUCTION`
`SET_INSTRUCTION`	Set the instruction register.	`DECODE`

The CU must be initialized before any computation can take place. The initial state is described in the following table, note that any registers or states not mentioned do not require initialization.

`next_state`	`GET_INSTRUCTION`
`program_counter`	`0x0000`

The CU is implemented in the `control_unit.c` and `control_unit.h` files.

With the control unit finished, the whole of the emulation code is ready. In order to properly test the control unit, a wrapper is required to load programs into memory and show outputs.

5 Wrapper

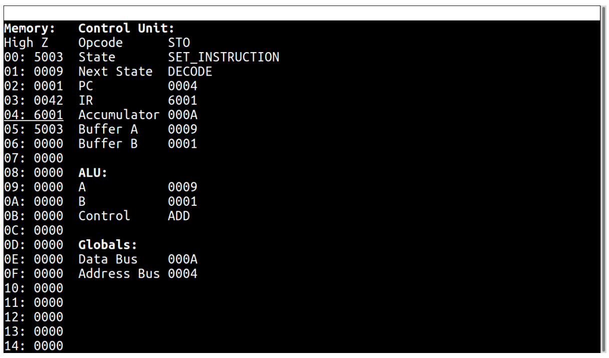
The purpose of the wrapper is to allow the user to view the internals of the emulated proccessor as it is in operation. The wrapper was designed with the following principles in mind.

- To allow you to step though a program clock-edge by clock-edge.
- To show you all of the important data in a human-readble format at a glance.

This was achived using a `ncurses` interface. Once the program is launched you are shown the system's current memory and each module's statistics.

The user can then force a clock edge by following the prompt and pressing enter. If the user wants to run an actual program rather then just what is in memory then pressing `l` loads a program.

This program can either be a `.hex` file that is specificed as an argument to the program or, by default, a simple program that adds 1 to a location in memory repeatadly. This program is defined in `emulator.h`.



```
Memory:   Control Unit:
High Z    Opcode      STO
00: 5003  State       SET_INSTRUCTION
01: 0009  Next State  DECODE
02: 0001  PC          0004
03: 0042  IR          6001
04: 6001  Accumulator 000A
05: 5003  Buffer A    0009
06: 0000  Buffer B    0001
07: 0000
08: 0000  ALU:
09: 0000  A           0009
0A: 0000  B           0001
0B: 0000  Control     ADD
0C: 0000
0D: 0000  Globals:
0E: 0000  Data Bus    000A
0F: 0000  Address Bus 0004
10: 0000
11: 0000
12: 0000
13: 0000
14: 0000
```

Figure 2: Screenshot of the wrapper program mid-way though a proccess.

The TUI underlines the currently selected address on the `addr` bus.

Please note that in order to compile the wrapper you will need the `ncurses` library and to compile with the command-line option `-lncurses`.

6 Assembler