COMP 3370 Assignment 1

Burke Libbey

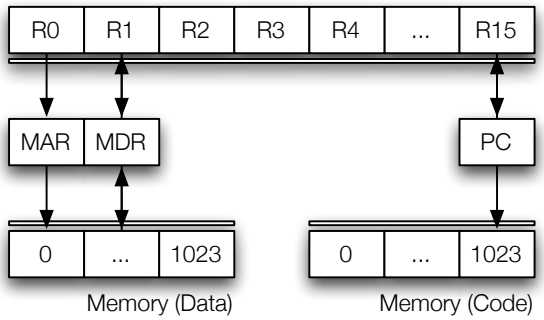
October 8, 2009

1 Design Documentation

1.1 Architechture Description

The CPU consists of several registers and two distinct memory spaces, for separate storage of program code and data. All registers and memory locations are 16 bits in size, modeled using `word_t`, which is typedef'd to `uint16_t`.

All the registers and memory spaces are contained in a single `struct cpu` or `cpu_t`. I probably should have made it a global variable, but instead, a pointer to this struct is passed around quite a bit.

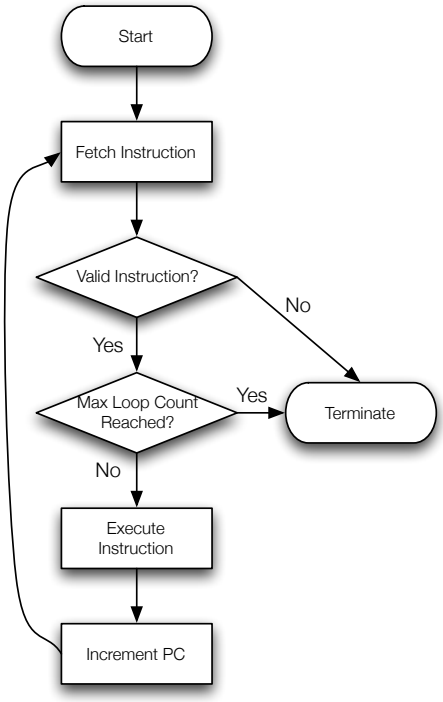


Like the spec suggests, I have `R0..R15` and `PC`. I've also added `MAR` and `MDR` to interface with memory as per a suggestion in class.

1.2 Execution

After initializing the CPU, the main loop will continue to fetch instructions, parse out the opcode and operands, execute instructions, and increment the program counter. It will terminate when an invalid instruction is reached, or an infinite loop is encountered.

Each time an instruction is fetched, both operands are extracted, whether or not they are of any use, and a function is then called to modify the CPU's state using the opcode and operands.



1.3 Program Implementation

The main program file is `src/sim.c`. This contains the main loop and various functions related to reading and parsing input to initialize the simulator.

`src/architecture.{c,h}` contains a description of the CPU architecture. The `cpu_t` struct contains all of the registers and memory locations available to a program for this CPU. The makeCpu() function initializes a `cpu_t` with sane default values (mostly zeroes, `0xFFFF` for code space).

`src/bitmacros.h` contains convenience macros for extracting data from interestingly-sized types. `OPCODE`, for example, extracts the 6 bits representing the opcode from an instruction value (16 bits).

Finally `src/instructions.{c,h}` defines operations for each instruction supported by the CPU. Instructions are stored as an array of function pointers. Each invocation of `INSTRUCTION` defines a function. These are later loaded into an array by `makeInstructionTable()` to return to the main program.

2 Test Programs

2.1 Test 1

This program calculates Fibonacci numbers. It stores 0 in `Data[0]` and 1 in `Data[1]`, then $Fib(n)$ in `Data[n]`, up to $Fib(30)$.

2.2 Test 2

This program searches a set of data for a value. It will search up to `Data[1]` words for the value in `Data[0]`. Once it is found the index at which it was found is stored in `Data[0]`. If no match is found, 0 is stored instead. It's worth noting, I guess, that if the target is in the first value searched, 0 is returned as the result, so it looks like an error.

2.3 Test 3

This program loops infinitely, since the `JR` jumps back to one instruction *after* the `BEQ` it was probably "supposed" to jump to.

2.4 Test 4

This program... doesn't even pretend to make sense? I guess it would terminate if `Data[1]` was 0... instead it just loops, adding to `R5` then comparing `R4`.

2.5 Test 5

This is a very similar search to the one in question 2, except it will never terminate if the data is not found.

2.6 Test 6

This program multiplies the numbers in `Data[0]` and `Data[1]`, storing the result to `Data[0]`. Obviously, overflow is a concern.

2.7 Test 7

This program essentially multiplies 9 and -1, storing the result in `Data[0]` and `Data[1]`. It uses a mostly-out-of-spec register-to-register move operation.