ELEC2204 – Computer Emulation

Maciej Romanski
mr12g15
Electronic Engineering with Computer Systems

Abstract

1 Introduction

The aim of this exercise was to implement the functionality of our own Von Neumann architecture in software. The architecture should feature a limited instruction set that follows a fetch/decode/execute/write-back cycle. The architecture was named "2204".

2 System Architecture

The 2204 architecture, at its core, is a 64-bit instruction set, with its memory addresses split between three "devices", the general purpose (GP) CPU registers, the RAM, and the NULL device. Each memory address stores 64 bits of data. A 64-bit block of data will be referred to as a "word". All addresses within the 64-bit space can be used, with the exception of `0x0` (the NULL device). This allows a maximum amount of addressed memory of:

$$\left(2^{64}-1\right)\frac{64}{8} \approx 147.5\,\mathrm{EB}$$

The architecture allows for variable GP register and RAM sizes, though once the system is running, the sizes are fixed. The CPU contains 4 static registers for storing intermediate data - usually instruction arguments.

Both the GP registers and the RAM are mandatory. Arithmetic and logical operations can only be carried out on data stored in the GP CPU registers. The bootloader program is loaded in to the RAM starting at the first address of the RAM.

Data in the memory can only be transferred through the GP CPU registers. Data can not be copied directly from one RAM address to another. They must be copied to GP CPU registers first, then copied in to the RAM.

Each CPU instruction can use a variable amount of memory, depending on the amount of operands it requires. Each operand takes up one word's worth of memory. Some examples are shown in table 2.

Only RAM addresses that are marked as `RAM_FREE` can be written to. After writing to a RAM address, that address will be marked as `RAM_USED`. To be able to overwrite a memory address, the `FREE` instruction should be used.

3 Implementation

The emulator for the 2204 instruction set was implemented in C. Each element of the emulator is in its own source file.

Mnemonic	Description	Operands	Size [words]
NOOP	No operation	0	1
COPY	Copy data	2	3
PRNT	Print hex	1	2
STOR	Store data	2	3
ADDA	Add	3	4
ANDA	And	3	4
JEQA	Jump if equal	3	4

Table 1: Word sizes of some of the implemented instructions. The instruction itself takes up one word, as does each argument.

3.1 Main emulator code

The code that containins the main function, initialises the emulator, loads the bootloader, and then runs the emulator, is located in `emulator.c`. This file loads the options specified in `default.cfg` using the `libconfig-1.5` library [1]. The loaded options are the path to the bootloader, the number of GP CPU registers, the first memory address of the RAM, and the number of RAM addresses. The RAM and CPU are then initialised using `ramInit()` and `cpuInit`. These functions are defined in `ram.c` and `cpu.c` respectively. Finally, the `boot()` function is called, which starts the emulator.

3.2 RAM

The code that handles the functions of the RAM is included in `ram.c`. The initialisation function, `ramInit()`, allocates the required amount of memory, and initialises it all to the `STOP` instruction. This is to prevent the CPU from running away if it jumps to an unused address. Each RAM address is also marked as `RAM_FREE`, to allow it to be written to. The `ramRead()` and `ramWrite()` functions are used to read and write data to the RAM respectively. Their `ramReadBurst()` and `ramWriteBurst()` counterparts allow a block of multiple words to be read or written, but no instructions have been written to make use of these functions. The `ramFree()` function is used to mark RAM addresses as `RAM_FREE`. This function is used by the `FREE` instruction. The `ramState` function is used to check whether a RAM address is free for writing, but no instructions that make use of this function have been implemented.

3.3 CPU

The code that handles the functions of the CPU is included in `cpu.h`. The initialisation function, `cpuInit()`, initialises the programme counter (PC) to the first RAM address, where the bootloader was stored by `emulator.c`. It also allocates the required amount of memory for the GP CPU registers. The main loop of the CPU is located in the `cpuRun()` function. The loop first loads the instruction from whatever device the address in the programme counter points to, then enters a switch/case statement which carries out the specified instruction. If any of the instructions return an error, or the CPU receives an unknown instruction, the CPU loop returns, and the emulator stops. The `checkResult()` function is mostly for debugging purposes.

It prints out any error information if it receives an unsuccessful result from an operation. The `memDirector()` function is used to determine which device a given memory address belongs to, so that checks for whether an instuction can be legally carried out can be made, as well as so that the CPU knows which method it should use to access/store data. The `debug()` function is used to print debug messages depending on the level of verbosity defined in the `CPU_DEBUG` macro in `cpu.h`. The `print_current_time_with_ms()` function citeprintCurrentTime is used for getting the current time for printing in the `debug()` function. The `getArg()` function is used for reading data from the correct device.

3.4 Errors

All the errors that the emulator is programmed to throw are defined in `errors.h` along with their more human-readable text strings, which are printed by the `printError()` function in `errors.c`. When an error is printed, the memory address at which the error occured is also given, which is useful for debugging.

3.5 Instructions

All the implemented instructions are defined in `./2204/2204.h`. This header file also contains human readable comments on the structure of each instruction, and how they should be implemented in the assembly language. The operation of every instrution, with the exception of `NOOP` and `STOP`, is defined in its own source file in the `./2204` directory. Some of the names of the source files differ from their mnemonics. In these cases, `./2204/2204.h` can be referred to again for the appropriate function name. For example, the function that carries out the `JGTP` (Jump Greater Than Positively), is called `jumpgtp2204()`. In this case, the appropriate source file will be `jumpgtp.c`. Every instruction source file is included in `./2204/2204.c`. This is so that the compilation command line does not have to contain the path to every single instruction's source file, just the path to `./2204/2204.c`.

3.5.1 COPY

This instruction was implemented before the `getArg()` function, so it does not make use of it for reading the source or destination addresses. Its operation is defined in `./2204/copy.c`. The PC is incremented to point to the first argument, which is the source address. The address is read, and then the operation is repeated to read the destination address. The addresses are then checked to make sure that the data are being copied to or from the the GP CPU registers. If so, the data from the source address is copied to the destination address according to their respective devices.

3.5.2 STOR

This instruction takes the literal value of the second argument, and stores it in the address specified in the first address. This instruction's operation is defined in `./2204/stor.c`. Firstly, the destination address is read, and the respective device is determined. Then the data is read, and is then written to the specified memory address on the appropriate device.

3.5.3 PRNT, PRND

This instructions get the data at the specified memory address, and then print the data to the terminal in hexadecimal or unsigned decimal form respectively. Both source files `./2204/prnt.c` and `./2204/prnd.c` are identical, apart from the differing `sprintf()` format specifiers - `PRIx64` for hexadecimal, `PRIu64` for unsigned decimal. The hexadecimal version also puts `"0x"` before the value.

3.5.4 FREE

This instruction takes a RAM address and a literal length. Starting at the given address, it marks a number equal to the specified length of consecutive RAM addresses starting at the specified address, as `RAM_FREE`, marking it as available for writing.

3.5.5 ADDA

This instruction adds the values at the first and second specified addresses, and stores the result in the third specified address. Both of the first two addresses must reside in the GP CPU registers. The function gets the first address, checks to make sure it's in the GP CPU registers, and gets the first operand and stores it in the CPU's first static register. It then does the same for the second address, gets the address to write back to, then writes the result to the address.

3.5.6 MULA

This instruction multiplies the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.7 ORAD

This instruction carries out a bitwise OR on the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.8 ANDA

This instruction carries out a bitwise AND on the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.9 NNDA

This instruction carries out a bitwise NAND on the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.10 NORA

This instruction carries out a bitwise NOR on the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.11 XORA

This instruction carries out a bitwise XOR on the values at the first and second specified addresses, and stores the result in the third specified address. The process is identical to that of ADDA's.

3.5.12 SUBA

This instruction subtracts the value of the second address from the value of the first address, and stores the result in the third address. The process is identical to that of ADDA's.

3.5.13 DIVA

This instruction divides the value of the first address by the value of the second address, and stores the result in the third address. The process is identical to that of ADDA's.

3.5.14 MODA

This instruction divides the value of the first address by the value of the second address, and stores the remainder of the result in the third address. The process is identical to that of ADDA's.

3.5.15 NOTA

This instruction carries out a bitwise NOT operation on the value of the first address, and stores the remainder of the result in the second address. The function gets the value of the first address, checks to make sure it's in the GP CPU registers, then writes the result of the NOT operation to the second address.

3.5.16 JUML

This instruction sets the PC to the specified address.

3.5.17 JEQA

This instruction checks if the values of the first and second addresses are equal. If so, the PC is set to the third address. Both operands must be located in the GP CPU registers.

3.5.18 JEQP

This instruction checks if the values of the first and second addresses are equal. If so, the PC is incremented by the amount specified in the third argument. Both operands must be located in the GP CPU registers.

3.5.19 JGTP

This instruction checks if the value of the first address is greater than that of then second address. If so, the PC is incremented by the amount specified in the third argument. Both operands must be located in the GP CPU registers.

3.5.20 JLGP

This instruction checks if the value of the first address is less than that of then second address. If so, the PC is incremented by the amount specified in the third argument. Both operands must be located in the GP CPU registers.

3.5.21 JEQN, JGTN, JLTN

These instructions are identical to their `JxxP` counterparts, but they decrement the PC by the value of the third argument, as opposed to increment.

4 Testing

5 Conclusion

6 Evaluation

Appendix A Baseline data

something

References

[1] hyperrealm. libconfig C/C++ Configuration File Library. Accessed: 2017-04-16. [Online]. Available: http://www.hyperrealm.com/libconfig/