CSC 350 Project Report

Intel 4004 Microprocessor Emulator

April 4th, 2019

Group Members

Trevor Harness	Akash Charitar	Alwien Dippenaar	Erik Yu
V00867541	V00875728	V00849850	V00865663
tharness@uvic.ca	akashcharitar@uvic.ca	alwiend@uvic.ca	hanxiaoyu@uvic.ca

Table of Contents

1 Introduction.........2
1.1 Purpose.........2
1.2 References.........2
1.3 Overview.........2
2 Overall Description.........2
2.1 Architecture Overview.........2
2.2 Instruction Format.........3
2.3 Emulator Features.........5
2.3.1 Supported Emulator Instructions.........6
2.3.2 Instruction Descriptions.........7
2.4 Demo Code.........7
2.4.1 Demo 1.........7
2.4.2 Demo 2.........9
2.4.3 Demo 3.........10
3 Closing Remarks.........11
3.1 Emulator Execution.........11
3.2 Emulator Limitations.........11
3.3 Unimplemented Instructions.........11

1 Introduction

1.1 Purpose

The Intel 4004 microprocessor is a 4-bit central processing unit released by Intel Corporation in 1971. The proposed project will be an emulation software for this architecture that will run any valid assembled program. The program itself will be written in C and will feature a command line interface that allows a pre-assembled program to be executed with or without a pipeline and the resulting memory to be dumped to a file for inspection and interpretation. This project is an in-depth study of the Intel 4004 microprocessor and how operations are completed with its architecture.

1.2 References

Cass, Stephen. “Chip Hall of Fame: Intel 4004 Microprocessor.” https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-intel-4004-microprocessor

“Intel 4004”. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Intel_4004

“Intel 4004 Microprocessor Architecture”. http://www.cpu-world.com/Arch/4004.html

“Mcs-40^{TM} Program Memory Organization”. http://pastraiser.com/cpu/i4040/i4040.html

Szyc, Maciej. “Intel 4004 Microprocessor”. http://e4004.szyc.org/index_en.html

1.3 Overview

This report will look at the Intel 4004 microprocessor and its architecture. It will give an overview of the emulator and its features that were implemented. A description of the Demo code will provide an oversight of the operations that is intended to showcase the emulator. Demo code will specifically showcase memory manipulation, arithmetic and logic operations.

2 Overall Description

2.1 Architecture Overview

The intel 4004 is dubbed the world’s first microprocessor, that being a complete general-purpose CPU in a single chip. The 4004 has a program memory size of 4KB, all conditional instructions work within currently selected ROM (256 bytes), and unconditional jump and jump to subroutine instructions can be used to jump to any address. The data memory is 640 bytes, RAM access done in the same way access I/O ports. Firstly, SRC instruction is used to tell the processor what memory to access, and successive WRM or RDM writes accumulator data to memory or reads data into accumulator. The data memory and the program memory are separated from each other. There is a 3-level deep stack (3 x 12-bit registers), which was also separated from program memory and data memory. The chip had 32 I/O ports, 16 4-bit input ports, and 16 4-bit output ports.

The chip has the following registers:

- Program Counter (12-bit)
- Three 12- bit stack level registers, enough to implement 3-level deep subroutine calls.
- Accumulator (4-bit), mainly used for arithmetic and logic operations.
- 16 4-bit Index registers, that can work in pairs as 8 8-bit registers.

The 4004 Instruction Set Architecture (ISA) consisted of a total of 46 instructions with a length of one or two bytes:

- Data moving instructions
- Arithmetic – add, subtract, increment and decrement.
- Logic – rotate.
- Control transfer – conditional, unconditional, call subroutine and return from subroutine.
- Input/output Instructions.
- Other – carry flag operations, decimal adjust, etc.

The chip includes 4 addressing modes: Register (4-bit), memory direct, register indirect, and immediate (4 and 8-bit data).

2.2 Instruction Format

There are two types of instructions that the intel 4004 microprocessor has a 1-word instruction with an 8-bit code and execution time of 10.8 usec, and 2-word instruction with a 16-bit code and execution time of 21.6 usec. For 1-word instructions, the high 4-bits is the operation code (OPR) which contains the operation that is to be performed, and the lower 4-bits is called the OPA which contains the modifier. The modifier itself contains one of four things:

1. A register address
2. A register pair address
3. 4-bits of data
4. An instruction modifier

For a 2-word instruction, the first word is like that of a 1-word instruction except for the modifier containing the following:

1. A register address
2. A register pair address
3. The upper portion of another Rom address
4. A condition for jumping

The second word contains either the middle portion (in OPR) and lower portion (in OPA) of another ROM address or 8-bits of data (upper 4-bits in OPR and lower 4-bits in OPA). The upper 4-bits of an instruction is always fetched before the lower 4-bits.

The 8-bit instruction is fetched 4-bits at a time on two successive clock periods. Illustration of the formats:

ONE WORD INSTRUCTION

D3	D2	D1	D0	D3	D2	D1	D0
X	X	X	X	X	X	X	X
OPR				OPA			

OPCODE	MODIFIER

X	X	X	X	INDEX REGISTER ADDRESS			
				R	R	R	R

OR

X	X	X	X	INDEX REGISTER PAIR ADDRESS			
				R	R	R	X

OR

X	X	X	X	DATA			
				D	D	D	D

TWO WORD INSTRUCTIONS

1ST INSTRUCTION CYCLE 2ND INSTRUCTION CYCLE

D3	D2	D1	D0	D3	D2	D1	D0	D3	D2	D1	D0	D3	D2	D1	D0
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OPR				OPA				OPR				OPA			

OP CODE	MODIFIER	OP CODE	MODIFIER

X	X	X	X	UPPER ADDRESS				MIDDLE ADDRESS				LOWER ADDRESS			
				A3	A3	A3	A3	A2	A2	A2	A2	A1	A1	A1	A1

OR

X	X	X	X	CONDITION				MIDDLE ADDRESS				LOWER ADDRESS			
				C1	C2	C3	C4	A2	A2	A2	A2	A1	A1	A1	A1

OR

X	X	X	X	INDEX REGISTER ADDRESS				MIDDLE ADDRESS				LOWER ADDRESS			
				R	R	R	X	A2	A2	A2	A2	A1	A1	A1	A1

OR

X	X	X	X	INDEX REGISTER PAIR ADDRESS				UPPER DATA				LOWER DATA			
				R	R	R	X	D2	D2	D2	D2	D1	D1	D1	D1

2.3 Emulator Features

The emulator is written in C and is executed with a command line interface allowing a pre-assembled program to be executed. The emulator will match all hardware components, except for the ROM and RAM. This is because our ROM is conditional to the size of the program and historically was limited to the current computer hardware as well. The RAM will be a single bank containing a single RAM chip, this is done for simplicity sake. With the single RAM bank, the emulator will have access to 1 bank * 1 chip * 4 registers * (16 nybbles * 4 status nybbles) = 80 nybbles = 320 bits of RAM. The emulator is designed to have the option to run with or without a pipeline, using the command "-p" in the command line interface.

The idea behind the pipeline was to allow fetch and decode for instructions concurrently while simultaneously executing and writing back to memory. As for coding the emulator this is done with the help of multi-threading, specifically using pthreads. When the command recognizes the pipeline command it creates a fetch and decode threads for the first instruction. Buffers are used to share data that has been processed by the fetch and decode stage to the execute and write back stage. Two threads are created for every instruction, one thread to fetch and decode and another to execute and write back, this ensures that the two threads can work in parallel and concurrently. Ultimately the fetch and decode work on the next instruction while the execute and write back thread work on the previous one. This configuration of threads caused control hazards where the branch instruction won't set fetch/decode to the appropriate instruction. This was solved by using a branch hazard flag which has a truth assignment, it is set to true whenever there is a branch instruction that has been decoded. The program counter of the decoded instruction is saved in a buffer. After every cycle, check to see if the branch hazard flag is set to true, if so then the program counter is copied into a buffer and another fetch/decode is ran. At the end of each fetch/decode and execute/write function, we exit its respective thread.

2.3.1 Supported Emulator Instructions

INSTRUCTION	MNEMONIC	BINARY EQUIVALENT		MODIFIERS
		1st byte	2nd byte	
No Operation	NOP	00000000	-	None
Fetch Immediate	FIM	0010RRR0	DDDDDDDD	Register pair, Data
Send Register Control	SRC	0010RRR1	-	Register pair
Fetch Indirect	FIN	0011RRR0	-	Register pair
Jump Unconditional	JUN	0100AAAA	AAAAAAAA	Address
Increment	INC	0110RRRR	-	Register
Increment and Skip	ISZ	0111RRRR	AAAAAAAA	Register, Address
Add	ADD	1000RRRR	-	Register
Sub	SUB	1001RRRR	-	Register
Load	LD	1010RRRR	-	Register
Exchange	XCH	1011RRRR	-	Register
Load Immediate	LDM	1101DDDD	-	Data
Write RAM Memory	WRM	11100000	-	None
Write Status Char 0	WR0	11100100	-	None
Write Status Char 1	WR1	11100101	-	None
Write Status Char 2	WR2	11100110	-	None
Write Status Char 3	WR3	11100111	-	None
Clear Both	CLB	11110000	-	None
Increment Accumulator	IAC	11110010	-	None
TERMINATE	KBP	11111100	-	None

2.3.2 Instruction Descriptions

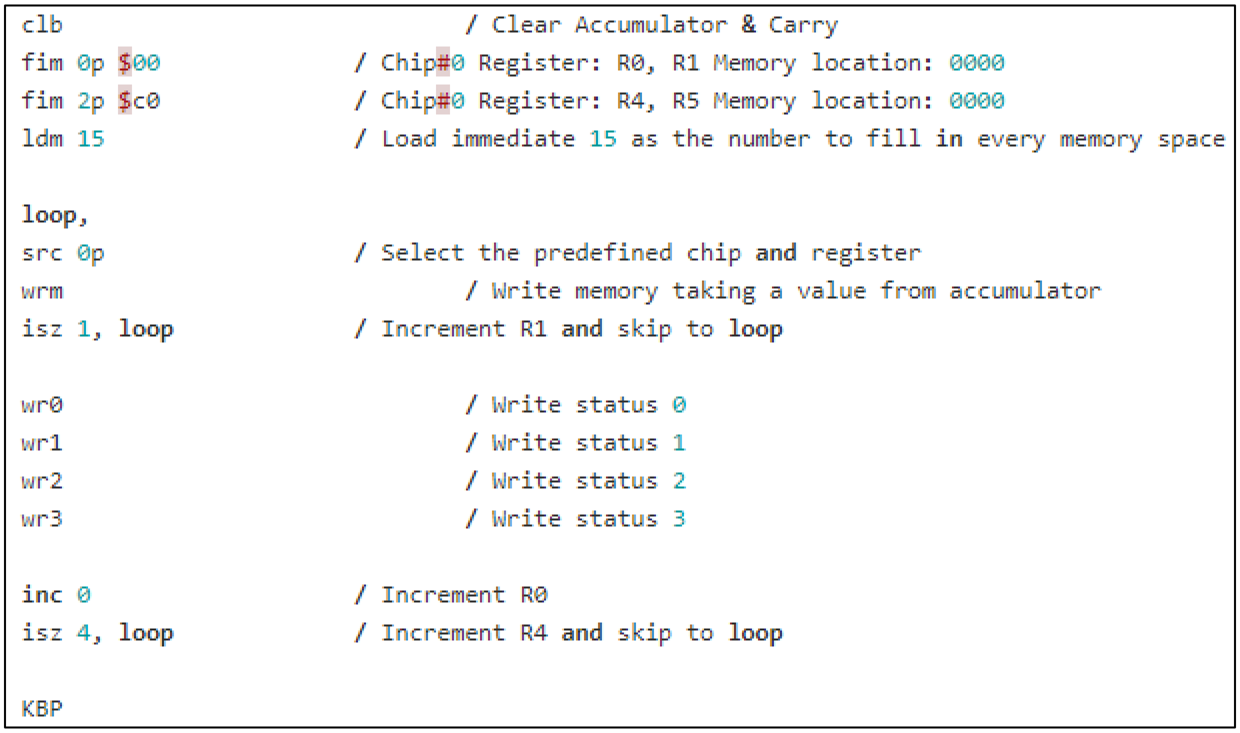
INSTRUCTION	DESCRIPTION
No Operation	No operation performed.
Fetch Immediate	Load the register pair RRR with D2, D1.
Send Register Control	Send the contents of register pair RRR to the ROMs and RAMs at cycle X_2 and X_3.
Fetch Indirect	Load register RRR with the contents of ROM in the current page at address pointed to by scratch-pad register pair 0.
Jump Unconditional	The 12-bit ROM address $A_3A_2A_1$ is loaded into the program counter.
Increment	Increment register R and set carry if overflow.
Increment and Skip	Increment register R. If the result is not zero jump to the address $A_{2,1}A_1$. Otherwise, execute next instruction.
Add	Add accumulator plus register R plus carry flag. If overflow, set carry flag.
Sub	Subtract from accumulator, register R and carry. If borrow, set carry.
Load	Load accumulator with contents of register R.
Exchange	Exchange the contents of accumulator with register R.
Load Immediate	Load accumulator with immediate data D.
Write RAM Memory	Write accumulator to previously selected RAM character.
Write Status Char S	Write accumulator to status character S of previously selected RAM chip. S = {0, 1, 2, 3}.
Clear Both	Clear accumulator and carry.
Increment Accumulator	Increment accumulator. If overflow, set carry.
TERMINATE	Signals the emulator that it has reached the end of a program and will terminate.

2.4 Demo Code

The demo code will include three programs featuring arithmetic and logic operations, including addition, subtraction, incrementing and decrementing. All programs will be terminated with the instruction “KBP” acting as an emulation termination signal, this instruction was chosen as the instruction is rarely used and so that the emulator won’t be stuck in an infinite loop.

2.4.1 Demo 1

Code demo one is a program that is related to memory manipulation, it uses nested loops to fill in every memory location with the value 0x0F into every cell of RAM. This demonstrates that the emulator can perform register loads, writing to memory, and conditional jumps.



```
clb                                  / Clear Accumulator & Carry
fim 0p $00                   / Chip#0 Register: R0, R1 Memory location: 0000
fim 2p $c0                   / Chip#0 Register: R4, R5 Memory location: 0000
ldm 15                       / Load immediate 15 as the number to fill in every memory space

loop,
src 0p                       / Select the predefined chip and register
wrm                                  / Write memory taking a value from accumulator
isz 1, loop                  / Increment R1 and skip to loop

wr0                                  / Write status 0
wr1                                  / Write status 1
wr2                                  / Write status 2
wr3                                  / Write status 3

inc 0                        / Increment R0
isz 4, loop                  / Increment R4 and skip to loop

KBP
```

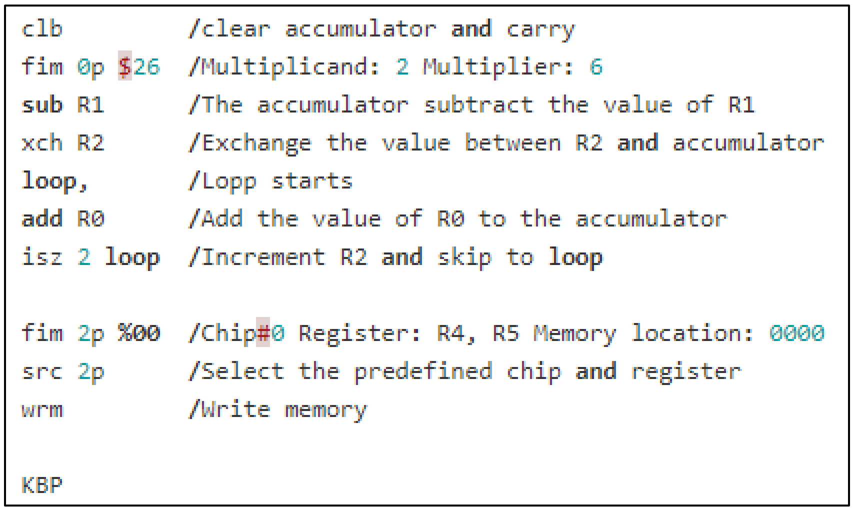
Figure 1: Demo Code 1

The expected output of this demo is as follows:

```
$ ./4004pem Demo1_hex
Number of cycles: 221
                REGISTERS
r00 to r03: 4 0 0 0
r04 to r07: 0 0 0 0
r08 to r11: 0 0 0 0
r12 to r15: 0 0 0 0
Accumulator: f
                MAIN MEMORY
Register:0  f f f f f f f f f f f f f f f f Status:  f f f f
Register:1  f f f f f f f f f f f f f f f f Status:  f f f f
Register:2  f f f f f f f f f f f f f f f f Status:  f f f f
Register:3  f f f f f f f f f f f f f f f f Status:  f f f f
```

2.4.2 Demo 2

Code demo two is a program that multiplies two numbers together that are set by the user in registers R0 and R1, both the multiplier and the multiplicand can be manipulated to produce different results. The demo code achieves the multiplying abilities by looping and adding the multiplicator to the accumulator, this will loop the amount of times determined by the multiplicand.



```
clb          /clear accumulator and carry
fim 0p $26   /Multiplicand: 2 Multiplier: 6
sub R1       /The accumulator subtract the value of R1
xch R2       /Exchange the value between R2 and accumulator
loop,        /Lopp starts
add R0       /Add the value of R0 to the accumulator
isz 2 loop   /Increment R2 and skip to loop

fim 2p %00   /Chip#0 Register: R4, R5 Memory location: 0000
src 2p       /Select the predefined chip and register
wrm          /Write memory

KBP
```

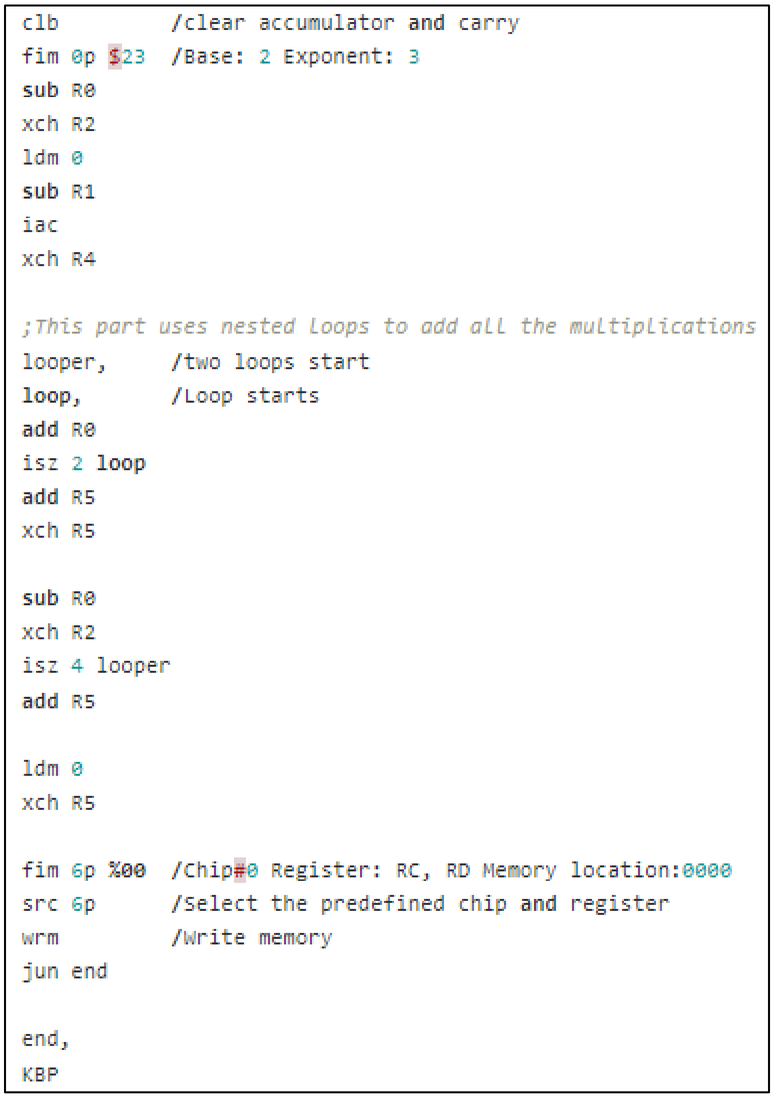
Figure 2: Demo Code 2

The expected output for demo code 2 is as follows:

```
$ ./4004pem Demo2_hex
Number of cycles: 20
               REGISTERS
r00 to r03: 2 6 0 0
r04 to r07: 0 0 0 0
r08 to r11: 0 0 0 0
r12 to r15: 0 0 0 0
Accumulator: c
               MAIN MEMORY
Register:0  c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:2  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:3  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
```

2.4.3 Demo 3

Demo code 3 is a program like demo code 2 however it calculates exponents of a number, for the demo it calculates 2^3 using nested loops with the result being stored in the accumulator.



```
clb         /clear accumulator and carry
fim 0p $23  /Base: 2 Exponent: 3
sub R0
xch R2
ldm 0
sub R1
iac
xch R4

;This part uses nested loops to add all the multiplications
looper,     /two loops start
loop,       /Loop starts
add R0
isz 2 loop
add R5
xch R5

sub R0
xch R2
isz 4 looper
add R5

ldm 0
xch R5

fim 6p %00  /Chip#0 Register: RC, RD Memory location:0000
src 6p      /Select the predefined chip and register
wrm         /Write memory
jun end

end,
KBP
```

Figure 3: Demo Code 3

The expected output for demo 3 is as follows:

```
$ ./4004pem Demo3_hex
Number of cycles: 34
               REGISTERS
r00 to r03: 2 3 2 0
r04 to r07: 0 0 0 0
r08 to r11: 0 0 0 0
r12 to r15: 0 0 0 0
Accumulator: 8
```

```
                    MAIN MEMORY
Register:0  8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:2  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
Register:3  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Status:  0 0 0 0
```

3 Closing Remarks

3.1 Emulator Execution

The emulator supports the following:

- Memory access
- Setting/reading/changing values both in memory and registers
- Conditional and unconditional branches

With these three points supported it is implied that the emulator is Turing complete and then therefore it could run any pre-assembled program.

3.2 Emulator Limitations

The most apparent limitation is the support for subroutine calls and the three-layer stack present in the 4004. This because the 3 12-bit stack registers needed for the three-layer subroutine calls as well as the instruction “jump to subroutine” were not implemented in the emulator.

3.3 Unimplemented Instructions

As for the remaining instructions that were not implemented for the emulator, they could be implemented by adding more cases in the decode and execute functions. The emulator that we implemented has a total of 19 instructions implemented and the termination instruction, leaving 26 unimplemented instructions remaining. However, the emulator that has been implemented is more than capable to showcase arithmetic and logic operations demonstrated with the demo programs.