Team Stonylake: Simulating the Intel 8008 8-bit microprocessor

Keanu Enns - keanelekenns@uvic.ca
Neal Manning - nmanning@uvic.ca
Nat Comeau - ncomeau@uvic.ca
Louis Kedziora - kedziora@uvic.ca

Introduction

Our team set out to simulate a substantial subset of Intel's first 8-bit microprocessor chip: the Intel 8008. We decided to do this project in order to get a better understanding of the structure and control paths of basic microprocessors; we believed the best way to do this was through simulating one ourselves. Of course, we could not simulate every aspect of the microprocessor with the time allotted, so we had to be selective as to what we would implement.

After reading over the 8008's user manual, we decided to divide the software into major component files that would interact with each other in a similar yet reasonably accomplishable fashion to what they do in hardware. Our goal was to implement a large portion of the microprocessor's instruction set and to deliver software that could execute programs using the set of developed instructions while imitating the actions taken by the hardware.

Additionally, we were asked to implement a two-stage pipeline, a feature that is not part of the original microprocessor. This obviously introduced extra logic that is not specified in the microprocessor's manual and required us to develop structures and mechanisms that would translate to additional hardware in the original microprocessor.

All code for the software was programmed in C as it is a familiar language to the members and provides lower-level bitwise operations for manipulating data in a fine-tuned manner.

Intel 8008 Overview

The responsibilities of the Intel 8008 are divided among five major components: Instruction Decoding and Control, Arithmetic Logical Unit (ALU), I/O, Data Registers and Program Counters (Internal Memory), and Timing (these can be seen in figure 1). Additionally, the processor communicates over an 8-bit data and address bus. With 14-bit addresses, the

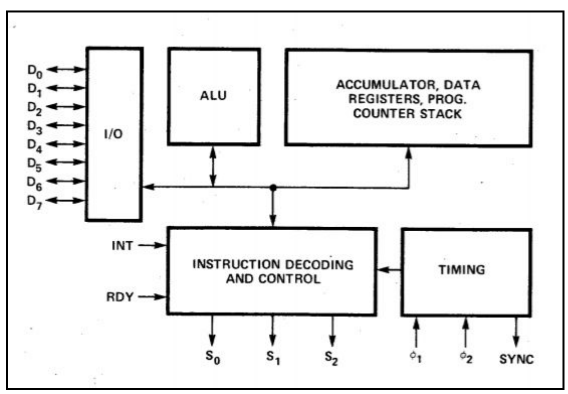

Figure 1. A block diagram of the major components of the Intel 8008 CPU.

processor can directly address 16KiB of external memory. Note that the 8008 has a von Neumann architecture.

The full instruction set includes 48 instructions that can be divided into 3 categories: data manipulation (loads and stores), arithmetic and logical operations, and conditional jump instructions. Instructions are one, two, or three bytes in size. The first byte for every instruction is the opcode. Instructions that perform operations on registers and previously calculated memory locations only require the one-byte opcode. Operations involving immediate values require 2 bytes, the second byte being the immediate value. Jump instructions require a memory location address and therefore have a size of 3 bytes, the second and third bytes compose the 14-bit address.

Memory or machine cycles are divided into five main states (T1, T2, T3, T4, T5). The first two send low and high bytes of an address to memory (this is due to the fact that the data bus is 8-bits wide). T3 is responsible for fetching and decoding instructions or data. The last two states are used for executing the instruction. However, instructions vary in the number of memory cycles they use. Therefore there must be control flow management to ensure the proper execution of instructions on different cycles (in the 8008, this control information is stored in the two most significant bits of the high byte of the address when it is sent to memory in T2). Consider the LrM (load register with contents of memory addressed by H and L data registers) instruction, the instruction is fetched during T3 in memory cycle 1, but in memory cycle 2, T3 is responsible for fetching the contents of the memory location specified by the two designated data registers.

Software Design Overview

When comparing our software to the physical microprocessor, differences often arise as a result of the fact that we do not have control over the physical hardware. In order to work around this, we have introduced some structures to help with the control of the system. One major structure we have created is the DecodeControl struct. This struct acts as a master control reference for the current condition of the system. It holds values for the number of memory cycles required for the current instruction, as well as the current cycle, the source and destination registers of an operation, and identifiers for what type of ALU operation needs to occur and what jump conditions need to be checked. The struct also holds control signals to manage execution during each of the T-states. It does this using a 3 byte (one for each potential memory cycle of an instruction) array for each T-state. These control signals are passed into execution functions (named T1_execute, etc.) as arguments during the main loop. For a more detailed description of the signals, refer to Decode.md in the StonyLake project repository.

We have two versions of our simulation: the non-pipelined version and the 2-stage pipelined version. Here we discuss the sequential version's main file. This file loads a program into memory (program memory is stored in addresses 0x0000 to 0x00FF of memory), initializes a DecodeControl struct, and enters into an infinite loop. It then calls each of the execution functions for the T-states and passes in the control values of the DecodeControl. Once the program reads in a HLT instruction (0xFF, 0x01, or 0x00), it stops execution. Because we do not have a device to display I/O, we created a function to print out the contents of the different elements of memory. After the execution of each instruction, as is determined by the control values and the current cycle value, we print out the contents of memory and prompt the user to continue before moving on to the next instruction. A crucial stage in this loop is T3 on the first cycle of an instruction, this is where the instruction is retrieved and decoded. After the decoding of the instruction, the control signals are placed in the DecodeControl struct to determine the execution path taken for the rest of the instruction's execution.

Major Components

Here we discuss our implementations of the major components of the microprocessor:

1. ALU: Most operations carried out by the ALU involve the accumulator (i.e. data register A). The different types of operations include logical, arithmetic, rotate, and compare operations. The functions in ALU.c accept one or two arguments (that in hardware would be stored into temporary registers a and b) and return the resulting value from the operations. The ALU is also responsible for setting the values for the four flag flip flops (Carry, Zero, Parity, and Sign). All operations are carried out as specified in the 8008 user manual. The add and subtract functions set the carry bit if there is overflow or underflow; however, the

microprocessor's policy for overflow and underflow was not defined, so the policy used in slides 4 and 5 of the 03_arithmetic slides was used to determine whether overflow or underflow occurred.

2. Instruction Decode and Control: Instructions are decoded by a function located in decode.c. The function takes an 8-bit value and determines what instruction is represented by it. Once the instruction has been determined, control signals are set in the DecodeControl struct. These control signals specify what should be done during each T-state, and are a custom design based on the available documentation.

3. I/O and Devices: Discuss On call of input or output functions, based on the opcode, one of eight devices can be selected. Then based on the opcode the 8008 can either ask for input from the selected device, or give output to the selected device to memory address on the device based on the 14 bits in the high and low address registers. Since these devices are outside of the chip the only device that is interfaced is the memory which is 16KB.

4. Memory: To represent both external and internal memory, we have created a struct in memory.h named mem that contains a 16 KiB array named memory (for external memory), a 7 byte array named scratch_pad which holds the seven data registers (A,B,C,D,E,H,L), an array of 8 uint16_t values which represents the address stack of the system, a program counter byte which acts as an index for the address stack, two bytes to represent high and low memory bytes of an address, an instruction register, and two temporary registers: reg_a and reg_b. Note for the address stack, each value in the array is a potential value for the program counter, upon executing a CALL instruction, the program counter index would be increased and would be used to index the next program counter value. However, we have not implemented CALL or RET instructions in our simulation, so the program counter index remains 0 and we always use the first element of the address stack for the program counter.

Challenges

As previously mentioned, not every aspect of the microchip could be reasonably simulated with the allotted time. An example of this would be the input and output mechanism for communicating with peripheral devices. Although we developed a framework for this interfacing between devices via ports, we did not develop it enough to actually communicate with outside devices because we were not focusing on that element of computer architecture during this project.

Another component we decided not to implement accurately was the timing of the microchip. This should not be a surprise given that we are writing programs in a high level language that could not possibly match the timing of the microchip. We also do not use any libraries for physical timing in our programs; however, as previously discussed, we do keep the structure of the memory cycles by dividing them into the five main T-states. Though there are additional states within the T-states (i.e. T1I (interrupted), WAIT, and STOPPED), we decided not to simulate them as we believe they are out of the scope of this project and they deal primarily with I/O feedback.

A complex aspect of the project was the division of processor execution into two types of state changes: memory cycles and T-states. First, each instruction takes anywhere from one to three memory cycles to complete. Second, each memory cycle takes between one and five T-states to complete. Table 1 gives two examples of this execution structure. The first instruction is Lrr, which requires one memory cycle, the second instruction is JMP, which requires three memory cycles. Note that blank T-state blocks represent an IDLE, SKIP or non- existent control value.

Table 1. Examples of the division of memory cycles into T-states and their specific signals.

Instruction:	Lrr	JMP
# Memory Cycles:	1	3
# T-States:	5	11
Mem Cycle 1		
T-State 1	PCL_out	PCL_out
T-State 2	PCH_out	PCH_out
T-State 3	Fetch Instruction to IR and Reg b.	Fetch Instruction to IR and Reg b.
T-State 4	Source register to Reg b	
T-State 5	Reg b to destination register	
Mem Cycle 2		
T-State 1		PCL_out
T-State 2		PCH_out
T-State 3		Fetch lower address to Reg b.
T-State 4		
T-State 5		
Mem Cycle 3		
T-State 1		PCL_out
T-State 2		PCH_out
T-State 3		Fetch higher address to Reg a.
T-State 4		Reg a to PCH
T-State 5		Reg b to PCL

This was the main inspiration for the design of the DecodeControl struct, which allows the main loop to be less involved with the manipulation of current states and cycles.

Finally, pipelining caused numerous challenges that we discuss in more detail in the next section.

Pipelining

After completing a non-pipelined version of the 8008 we went on to simulate a version with a two-stage fetch-decode (IF/ID) / execute-writeback (EX/WB) pipeline. We did so by noticing that T-states 1 to 3 in the first memory cycle make up the fetch-decode stage and the remaining t-states and memory cycles make up the execute-writeback stages. This can be seen in figure 2, which is taken from pages 16 and 17 of the Intel 8008 User manual [1].

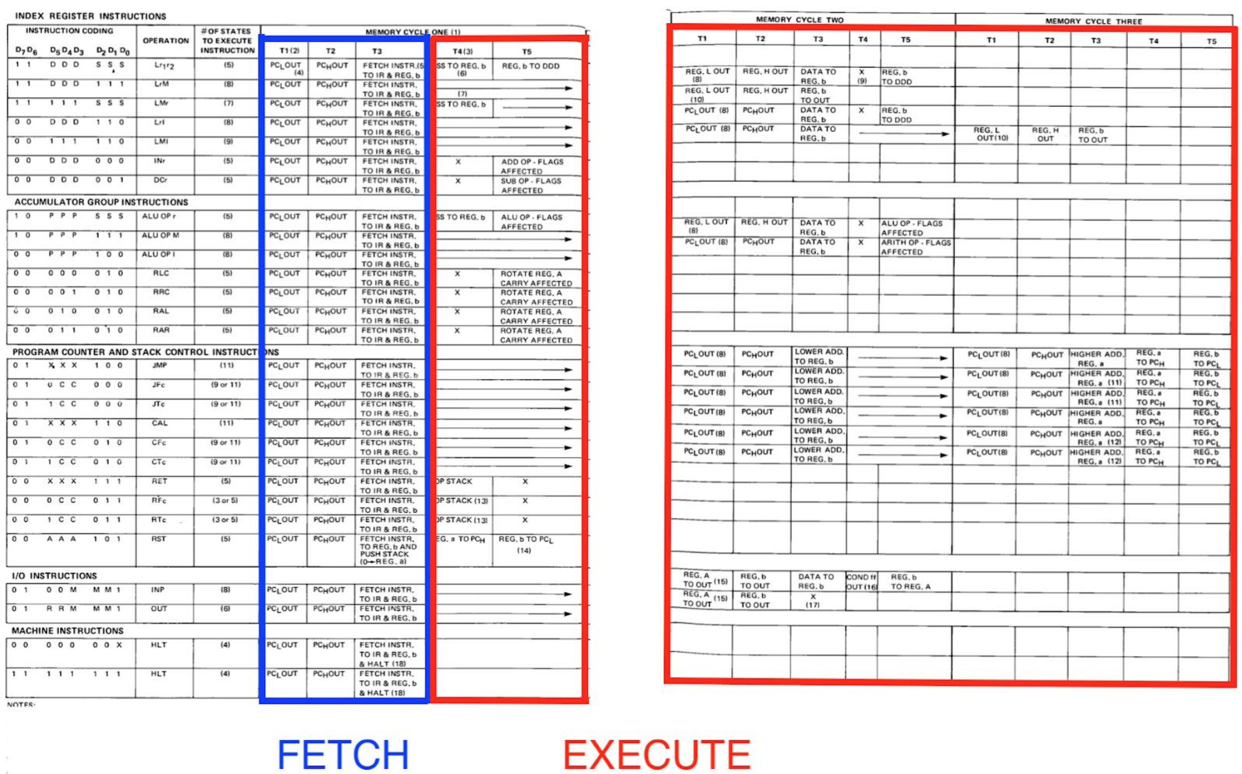

Figure 2. The division of T states from the 8008 manual for the 2-stage pipeline [1].

Note this microprocessor did not implement a pipeline in reality, so we had to make changes to the actual execution process in order to implement our two stage pipeline. Additionally, since we do not have access to hardware, and we did not create a multithreaded simulation, we cannot have a true pipeline that executes stages in parallel. Therefore, the execution is technically still sequential, but the IF/ID stages are interleaved with the EX/WB stages causing the need for detecting control hazards.

To simulate this in sequential order we:

- Fetch an instruction, decode it, and increment the program counter based on the size of the instruction in memory (1, 2, or 3 bytes).
- Fetch the next instruction, decode it, and decrement the program counter to match the state as if the first instruction had finished its IF/ID stage. This again uses the byte size of the first instruction.
- Execute the first instruction, checking for jump instructions (conditional and unconditional).
- Unless the first instruction was a jump or halt instruction, execute the second instruction fetched.
- If there is a jump instruction, we assume it is not taken. The policy for a taken branch is to “stall-on-branch” in which case we ignore the previously fetched instruction that would have executed after the jump. This resolves all control hazards in our simulation.
- If we identify a HALT instruction, then the program stops in the EX/WB stage. The HALT needs to occur in EX/WB in the event that it is the second instruction fetched, this way the previous instruction will still execute.

A visualization of this pipeline can be seen in figure 3. The pipeline stages are labeled with an ‘S’ followed by the stage number. On the left is a theoretical pipeline with Fetch and Execute stages happening concurrently in time. Because we are simulating the processor in software (and not using threads or multiprocessing) we needed to run our pipeline sequentially in software. On the right is our implementation of a 2-stage Fetch/Execute pipeline.

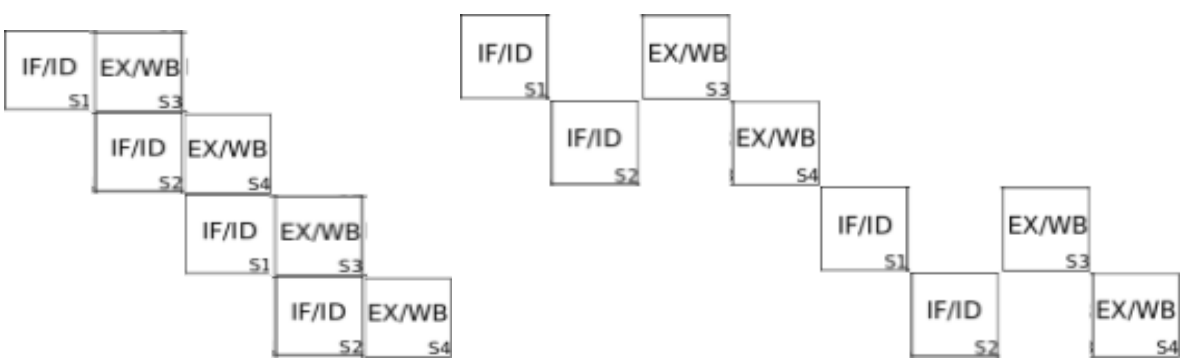

Figure 3. The theoretical pipeline vs the actual pipeline. Time increases from left to right.

Note that all data and structural hazards are resolved by the nature of the execution process (i.e. it is still sequential). Two stages never actually occur simultaneously, and therefore do not conflict. The results of any instruction are computed and written to memory before another instruction executes.

Some duplication was required to save control values for each instruction. For example, the temporary registers and the DecodeControl structs are saved between pipeline stages. In reality this would require extra hardware or some way of pushing those values to memory and retrieving them afterwards.

Conclusion

We designed, implemented and tested simulations of a non-pipelined Intel 8008. Furthermore, we applied principles of computer architecture to design and implement a two-stage Fetch/Decode Execute/Write-Back pipeline. We learned valuable lessons about simulating hardware in software with this project.

As a case study we ran the "increment_mem.asm" on the non-pipelined simulation and obtain an execution time of 7 microseconds by timing user and system time. This simulation contains 353 T-states. Multiplying T-states by 2 and dividing by execution time we obtain 100857 machine cycles per second. With the original machine executing at a frequency of 500KHz to 800KHz our simulated processor is within an order of magnitude of the performance of the original.

User Guide

Downloading and Building the Project

To download the software from the git repository:

1. run "git clone https://gitlab.csc.uvic.ca/courses/201901/csc350/stonylake.git".
2. Change to the project directory: "cd stonylake"
3. Select the Pipelined or Non-Pipelined version:
    a. To use the pipelined version, checkout the "2-stage" branch.

   b. To use the non-pipelined version, stay on the master branch.
4. Build the project by running “make”.

You should be ready to start running programs.

Running a Program

The simulator runs executes one instruction at a time. To run:

- Use the command “./main <input_file_name>”
- Two arguments enables debug printing.
- Three arguments runs the program without printing.

Writing Programs for the 8008

Input File Format:

The format of the input file is a file containing 8-bit machine code instructions (1s and 0s) and comments. ***Comments cannot contain “0” or “1” characters.*** See examples for a detailed file format.

Programmers Instruction Reference:

To make writing the machine code easier in appendix A we include subsections of the 8008 User Manual as a programmers instruction reference. We have implemented all instructions in the manual with the exception of all variants of the CALL and RETURN instructions and the INP and OUT instructions.

Testing the System

Sample machine code test files and expected results are included in the “test_programs” directory.

Error Handling

When an instruction is invalid, the error message “Opcode = 0x____ Is not recognized as an instruction” is printed.

Examples

Incrementing a value in data memory

We demonstrate a program to initialize memory address 256 (mem[256]) to 0 and increment it 5 times. The program should terminate with mem[256] containing 5.

Instructions used:

Index reg instructions:

- Lrr
- LrI
- LrM
- LMr
- DCr

Accumulator group instructions:

- ADI
- CPI

Program Counter and Stack Control Instructions:

- JFc

Machine Instructions:

- HLT

Pseudocode:

- Store 0 in mem[256].
- Store 5 in register C; decrement this in each loop and when it reaches zero we're done.
- Loop:
    - Load mem[256] into accumulator.
    - Increment accumulator
    - Store accumulator into mem[256].
    - Decrement register C.
    - Load accumulator with register C.
    - Compare accumulator with zero.
    - Branch if not zero to Loop.
    - Halt

```
Address        Instruction          BINARY INSTR   Comment

0                      LLI 0         00110110
1                                    00000000
2                      LHI 256 00101110
3                                    00000001
4                      ADI 0         00000100            From accumulator
5                                    00000000            store zero in mem[256]
6                      LMA           11111000
7                      LCI 5         00010110
8                                    00000101
9              Loop:   LAM           11000111            Load mem[256] into accumulator
10                     ADI 1         00000100            Increment accumulator
11                                   00000001
12                     LMA           11111000            Store accumulator to mem[256]
13                     DCC           00010001            Decrement register C
14                     LAC           11000010            Copy register C to accumulator
15                     CPI 0         00111100            Accumulator = 0?
16                                   00000000
17                     JFZ loop      01001000            If not, jump to Loop.
18                                   00001001
19                                   00000000
20                     HLT           11111111            Else halt.
```

Storing a String in Data Memory

We demonstrate a program to load the string “Stoneylake.” into data memory addresses 200 - 210.

```
Memory address,
Ascii char,
Binary representation of Ascii:

  256      257      258      259      260

  'S'   |  't'   |  'o'   |  'n'   |  'y'   |

  261      262      263      264      265     266

01010011 01110100 01101111 01101110 01111001

  'l'   |  'a'   |  'k'   |  'e'   |  '.'   |  0

01101100 01100001 01101011 01100101 00101110 00000000
```

Instructions used:

Index reg instructions:

- LrI

- LMr
- INr

Pseudocode:

- Set L and H to point to mem[256]
- Store 'S' into mem[256]
- Set L and H to point to mem[257]
- Store 't' into mem[257]

...

- Set L and H to point to mem[266]
- Store 0 into mem[266]
- Halt

```
Address     Instruction       BINARY INSTR

0                 LLI 0       00110110
1                             00000000
2                 LHI 256     00101110
3                             00000001
4                 LMI   'S'   00111110
5                             01010011
6                 INL         00110000
5                 LMI   't'   00111110
6                             01110100
7                 INL         00110000
8                 LMI   'o'   00111110
9                             01101111
10                INL         00110000
11                LMI   'n'   00111110
12                            01101110
13                INL         00110000
14                LMI   'y'   00111110
15                            01111001
16                INL         00110000
17                LMI   'l'   00111110
18                            01101100
19                INL         00110000
20                LMI   'a'   00111110
21                            01100001
22                INL         00110000
23                LMI   'k'   00111110
24                            01101011
25                INL         00110000
26                LMI   'e'   00111110
```

```
27                          01100101
28              INL         00110000
29              LMI   '.'   00111110
30                          00101110
31              INL         00110000
32              LMI   0     00111110
33                          00000000
34              INL         00110000
35              HLT         11111111
```

References

[1] Intel 8008 User Manual. http://bitsavers.informatik.uni-stuttgart.de/components/intel/MCS8/Intel_8008_8-Bit_Parallel_Central_Processing_Unit_Rev4_Nov73.pdf. Retrieved March 7th 2019.

Appendix A: Programmers Instruction Reference

MNEMONIC	MINIMUM STATES REQUIRED	INSTRUCTION CODE $D_7 D_6$	$D_5 D_4 D_3$	$D_2 D_1 D_0$	DESCRIPTION OF OPERATION
NDr	(5)	1 0	1 0 0	S S S	Compute the logical AND of the content of index register r,
NDM	(8)	1 0	1 0 0	1 1 1	memory register M, or data B . . . B with the accumulator.
NDI	(8)	0 0 B B	1 0 0 B B B	1 0 0 B B B	
XRr	(5)	1 0	1 0 1	S S S	Compute the EXCLUSIVE OR of the content of index register
XRM	(8)	1 0	1 0 1	1 1 1	r, memory register M, or data B . . . B with the accumulator.
XRI	(8)	0 0 B B	1 0 1 B B B	1 0 0 B B B	
ORr	(5)	1 0	1 1 0	S S S	Compute the INCLUSIVE OR of the content of index register
ORM	(8)	1 0	1 1 0	1 1 1	r, memory register m, or data B . . . B with the accumulator.
ORI	(8)	0 0 B B	1 1 0 B B B	1 0 0 B B B	
CPr	(5)	1 0	1 1 1	S S S	Compare the content of index register r, memory register M,
CPM	(8)	1 0	1 1 1	1 1 1	or data B . . . B with the accumulator. The content of the
CPI	(8)	0 0 B B	1 1 1 B B B	1 0 0 B B B	accumulator is unchanged.
RLC	(5)	0 0	0 0 0	0 1 0	Rotate the content of the accumulator left.
RRC	(5)	0 0	0 0 1	0 1 0	Rotate the content of the accumulator right.
RAL	(5)	0 0	0 1 0	0 1 0	Rotate the content of the accumulator left through the carry.
RAR	(5)	0 0	0 1 1	0 1 0	Rotate the content of the accumulator right through the carry.

Program Counter and Stack Control Instructions

(4) JMP	(11)	0 1 B_2 B_2 X X	X X X B_2 B_2 B_2 B_3 B_3 B_3	1 0 0 B_2 B_2 B_2 B_3 B_3 B_3	Unconditionally jump to memory address $B_3 \ldots B_3 B_2 \ldots B_2$.
(5) JFc	(9 or 11)	0 1 B_2 B_2 X X	0 C_4 C_3 B_2 B_2 B_2 B_3 B_3 B_3	0 0 0 B_2 B_2 B_2 B_3 B_3 B_3	Jump to memory address $B_3 \ldots B_3 B_2 \ldots B_2$ if the condition flip-flop c is false. Otherwise, execute the next instruction in sequence.
JTc	(9 or 11)	0 1 B_2 B_2 X X	1 C_4 C_3 B_2 B_2 B_2 B_3 B_3 B_3	0 0 0 B_2 B_2 B_2 B_3 B_3 B_3	Jump to memory address $B_3 \ldots B_3 B_2 \ldots B_2$ if the condition flip-flop c is true. Otherwise, execute the next instruction in sequence.
CAL	(11)	0 1 B_2 B_2 X X	X X X B_2 B_2 B_2 B_3 B_3 B_3	1 1 0 B_2 B_2 B_2 B_3 B_3 B_3	Unconditionally call the subroutine at memory address $B_3 \ldots B_3 B_2 \ldots B_2$. Save the current address (up one level in the stack).
CFc	(9 or 11)	0 1 B_2 B_2 X X	0 C_4 C_3 B_2 B_2 B_2 B_3 B_3 B_3	0 1 0 B_2 B_2 B_2 B_3 B_3 B_3	Call the subroutine at memory address $B_3 \ldots B_3 B_2 \ldots B_2$ if the condition flip-flop c is false, and save the current address (up one level in the stack.) Otherwise, execute the next instruction in sequence.
CTc	(9 or 11)	0 1 B_2 B_2 X X	1 C_4 C_3 B_2 B_2 B_2 B_3 B_3 B_3	0 1 0 B_2 B_2 B_2 B_3 B_3 B_3	Call the subroutine at memory address $B_3 \ldots B_3 B_2 \ldots B_2$ if the condition flip-flop c is true, and save the current address (up one level in the stack). Otherwise, execute the next instruction in sequence.
RET	(5)	0 0	X X X	1 1 1	Unconditionally return (down one level in the stack).
RFc	(3 or 5)	0 0	0 C_4 C_3	0 1 1	Return (down one level in the stack) if the condition flip-flop c is false. Otherwise, execute the next instruction in sequence.
RTc	(3 or 5)	0 0	1 C_4 C_3	0 1 1	Return (down one level in the stack) if the condition flip-flop c is true. Otherwise, execute the next instruction in sequence.
RST	(5)	0 0	A A A	1 0 1	Call the subroutine at memory address AAA000 (up one level in the stack).

Input/Output Instructions

INP	(8)	0 1	0 0 M	M M 1	Read the content of the selected input port (MMM) into the accumulator.
OUT	(6)	0 1	R R M	M M 1	Write the content of the accumulator into the selected output port (RRMMM, RR ≠ 00).

Machine Instruction

HLT	(4)	0 0	0 0 0	0 0 X	Enter the STOPPED state and remain there until interrupted.
HLT	(4)	1 1	1 1 1	1 1 1	Enter the STOPPED state and remain there until interrupted.

NOTES:

(1) SSS = Source Index Register, DDD = Destination Index Register — These registers, r_i, are designated A(accumulator—000), B(001), C(010), D(011), E(100), H(101), L(110).

(2) Memory registers are addressed by the contents of registers H & L.

(3) Additional bytes of instruction are designated by BBBBBBBB.

(4) X = "Don't Care".

(5) Flag flip-flops are defined by C_4C_3: carry (00-overflow or underflow), zero (01-result is zero), sign (10-MSB of result is "1"), parity (11-parity is even).

IV. BASIC INSTRUCTION SET

The following section presents the basic instruction set of the 8008. For a detailed description of the execution of each instruction, refer to Appendix I.

Data and Instruction Formats

Data in the 8008 is stored in the form of 8-bit binary integers. All data transfers to the system data bus will be in the same format.

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

DATA WORD

The program instructions may be one, two, or three bytes in length. Multiple byte instructions must be stored in successive words in program memory. The instruction formats then depend on the particular operation executed.

		TYPICAL INSTRUCTIONS
One Byte Instructions		
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	OP CODE	Register to register, memory reference, I/O arithmetic or logical, rotate or return instructions
Two Byte Instructions		
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	OP CODE	
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	OPERAND	Immediate mode instructions
Three Byte Instructions		
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	OP CODE	
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	LOW ADDRESS	JUMP or CALL instructions
X X D_5 D_4 D_3 D_2 D_1 D_0	HIGH ADDRESS*	

*For the third byte of this instruction, D_6 and D_7 are "don't care" bits.

For the MCS-8 a logic "1" is defined as a high level and a logic "0" is defined as a low level.

Index Register Instructions

The load instructions do not affect the flag flip-flops. The increment and decrement instructions affect all flip-flops except the carry.

MNEMONIC	MINIMUM STATES REQUIRED	INSTRUCTION CODE D_7 D_6	D_5 D_4 D_3	D_2 D_1 D_0	DESCRIPTION OF OPERATION
(1) Lr_1r_2	(5)	1 1	D D D	S S S	Load index register r_1 with the content of index register r_2.
(2) LrM	(8)	1 1	D D D	1 1 1	Load index register r with the content of memory register M.
LMr	(7)	1 1	1 1 1	S S S	Load memory register M with the content of index register r.
(3) LrI	(8)	0 0	D D D	1 1 0	Load index register r with data B . . . B.
		B B	B B B	B B B	
LMI	(9)	0 0	1 1 1	1 1 0	Load memory register M with data B . . . B.
		B B	B B B	B B B	
INr	(5)	0 0	D D D	0 0 0	Increment the content of index register r ($r \neq A$).
DCr	(5)	0 0	D D D	0 0 1	Decrement the content of index register r ($r \neq A$).

Accumulator Group Instructions

The result of the ALU instructions affect all of the flag flip-flops. The rotate instructions affect only the carry flip-flop.

ADr	(5)	1 0	0 0 0	S S S	Add the content of index register r, memory register M, or data B . . . B to the accumulator. An overflow (carry) sets the carry flip-flop.
ADM	(8)	1 0	0 0 0	1 1 1	
ADI	(8)	0 0	0 0 0	1 0 0	
		B B	B B B	B B B	
ACr	(5)	1 0	0 0 1	S S S	Add the content of index register r, memory register M, or data B . . . B from the accumulator with carry. An overflow (carry) sets the carry flip-flop.
ACM	(8)	1 0	0 0 1	1 1 1	
ACI	(8)	0 0	0 0 1	1 0 0	
		B B	B B B	B B B	
SUr	(5)	1 0	0 1 0	S S S	Subtract the content of index register r, memory register M, or data B . . . B from the accumulator. An underflow (borrow) sets the carry flip-flop.
SUM	(8)	1 0	0 1 0	1 1 1	
SUI	(8)	0 0	0 1 0	1 0 0	
		B B	B B B	B B B	
SBr	(5)	1 0	0 1 1	S S S	Subtract the content of index register r, memory register M, or data B . . . B from the accumulator with borrow. An underflow (borrow) sets the carry flip-flop.
SBM	(8)	1 0	0 1 1	1 1 1	
SBI	(8)	0 0	0 1 1	1 0 0	
		B B	B B B	B B B	

APPENDIX I

FUNCTIONAL DEFINITION

Symbols	Meaning
<B2>	Second byte of the instruction
<B3>	Third byte of the instruction
r	One of the scratch pad register references: A, B, C, D, E, H, L
c	One of the following flag flip-flop references: C, Z, S, P
C_4C_3	Flag flip-flop codes / Condition for True 00 carry — Overflow, underflow 01 zero — Result is zero 10 sign — MSB of result is "1" 11 parity — Parity of result is even
M	Memory location indicated by the contents of registers H and L
( )	Contents of location or register
$\wedge$	Logical product
$\forall$	Exclusive "or"
V	Inclusive "or"
A_m	Bit m of the A-register
STACK	Instruction counter (P) pushdown register
P	Program Counter
$\leftarrow$	Is transferred to
XXX	A "don't care"
SSS	Source register for data
DDD	Destination register for data

Register # (SSS or DDD)	Register Name
000	A
001	B
010	C
011	D
100	E
101	H
110	L

INDEX REGISTER INSTRUCTIONS

LOAD DATA TO INDEX REGISTERS – One Byte

Data may be loaded into or moved between any of the index registers, or memory registers.

Lr_1r_2 (one cycle – PCI)	11 DDD SSS	$(r_1) \leftarrow (r_2)$ Load register r_1 with the content of r_2. The content of r_2 remains unchanged. If SSS=DDD, the instruction is a NOP (no operation).
LrM (two cycles – PCI/PCR)	11 DDD 111	$(r) \leftarrow (M)$ Load register r with the content of the memory location addressed by the contents of registers H and L. (DDD≠111 – HALT instr.)
LMr (two cycles – PCI/PCW)	11 111 SSS	$(M) \leftarrow (r)$ Load the memory location addressed by the contents of registers H and L with the content of register r. (SSS≠111 – HALT instr.)

LOAD DATA IMMEDIATE – Two Bytes

A byte of data immediately following the instruction may be loaded into the processor or into the memory

LrI (two cycles – PCI/PCR)	00 DDD 110 $<B_2>$	$(r) \leftarrow <B_2>$ Load byte two of the instruction into register r.
LMI (three cycles – PCI/PCR/PCW)	00 111 110 $<B_2>$	$(M) \leftarrow <B_2>$ Load byte two of the instruction into the memory location addressed by the contents of registers H and L.

INCREMENT INDEX REGISTER – One Byte

INr (one cycle – PCI)	00 DDD 000	$(r) \leftarrow (r)+1$. The content of register r is incremented by one. All of the condition flip-flops except carry are affected by the result. Note that DDD≠000 (HALT instr.) and DDD≠111 (content of memory may not be incremented).

DECREMENT INDEX REGISTER – One Byte

DCr (one cycle – PCI)	00 DDD 001	$(r) \leftarrow (r)-1$. The content of register r is decremented by one. All of the condition flip-flops except carry are affected by the result. Note that DDD≠000 (HALT instr.) and DDD≠111 (content of memory may not be decremented).

ACCUMULATOR GROUP INSTRUCTIONS

Operations are performed and the status flip-flops, C, Z, S, P, are set based on the result of the operation. Logical operations (NDr, XRr, ORr) set the carry flip-flop to zero. Rotate operations affect only the carry flip-flop. Two's complement subtraction is used.

ALU INDEX REGISTER INSTRUCTIONS – One Byte

(one cycle – PCI)

Index Register operations are carried out between the accumulator and the content of one of the index registers (SSS=000 thru SSS=110). The previous content of register SSS is unchanged by the operation.

ADr	10 000 SSS	$(A) \leftarrow (A)+(r)$ Add the content of register r to the content of register A and place the result into register A.
ACr	10 001 SSS	$(A) \leftarrow (A)+(r)+(carry)$ Add the content of register r and the contents of the carry flip-flop to the content of the A register and place the result into Register A.
SUr	10 010 SSS	$(A) \leftarrow (A)-(r)$ Subtract the content of register r from the content of register A and place the result into register A. Two's complement subtraction is used.

ACCUMULATOR GROUP INSTRUCTIONS - Cont'd.

SBr	10	011	SSS	(A)←(A)−(r)−(borrow) Subtract the content of register r and the content of the carry flip-flop from the content of register A and place the result into register A.
NDr	10	100	SSS	(A)←(A)∧(r) Place the logical product of the register A and register r into register A.
XRr	10	101	SSS	(A)←(A)⊻(r) Place the "exclusive - or" of the content of register A and register r into register A.
ORr	10	110	SSS	(A)←(A)V(r) Place the "inclusive - or" of the content of register A and register r into register A.
CPr	10	111	SSS	(A)−(r) Compare the content of register A with the content of register r. The content of register A remains unchanged. The flag flip-flops are set by the result of the subtraction. Equality or inequality is indicated by the zero flip-flop. Less than or greater than is indicated by the carry flip-flop.

ALU OPERATIONS WITH MEMORY — One Byte
(two cycles — PCI/PCR)
Arithmetic and logical operations are carried out between the accumulator and the byte of data addressed by the contents of registers H and L.

ADM	10	000	111	(A)←(A)+(M) ADD
ACM	10	001	111	(A)←(A)+(M)+(carry) ADD with carry
SUM	10	010	111	(A)←(A)−(M) SUBTRACT
SBM	10	011	111	(A)←(A)−(M)−(borrow) SUBTRACT with borrow
NDM	10	100	111	(A)←(A)∧(M) Logical AND
XRM	10	101	111	(A)←(A)⊻(M) Exclusive OR
ORM	10	110	111	(A)←(A)V(M) Inclusive OR
CPM	10	111	111	(A)−(M) COMPARE

ALU IMMEDIATE INSTRUCTIONS — Two Bytes
(two cycles —PCI/PCR)
Arithmetic and logical operations are carried out between the accumulator and the byte of data immediately following the instruction.

ADI	00	000 <B_2>	100	(A)←(A)+<B_2> ADD
ACI	00	001 <B_2>	100	(A)←(A)+<B_2>+(carry) ADD with carry
SUI	00	010 <B_2>	100	(A)←(A)−<B_2> SUBTRACT
SBI	00	011 <B_2>	100	(A)←(A)−<B_2> −(borrow) SUBTRACT with borrow
NDI	00	100 <B_2>	100	(A)←(A)∧<B_2> Logical AND
XRI	00	101 <B_2>	100	(A)←(A)⊻ <B_2> Exclusive OR
ORI	00	110 <B_2>	100	(A)←(A)V <B_2> Inclusive OR
CPI	00	111 <B_2>	100	(A)− <B_2> COMPARE

ROTATE INSTRUCTIONS – One Byte
(one cycle – PCI)
The accumulator content (register A) may be rotated either right or left, around the carry bit or through the carry bit. Only the carry flip-flop is affected by these instructions; the other flags are unchanged.

RLC	00	000	010	$A_{m+1} \leftarrow A_m$, $A_0 \leftarrow A_7$, $(carry) \leftarrow A_7$ Rotate the content of register A left one bit. Rotate A_7 into A_0 and into the carry flip-flop.
RRC	00	001	010	$A_m \leftarrow A_{m+1}$, $A_7 \leftarrow A_0$, $(carry) \leftarrow A_0$ Rotate the content of register A right one bit. Rotate A_0 into A_7 and into the carry flip-flop.
RAL	00	010	010	$A_{m+1} \leftarrow A_m$, $A_0 \leftarrow (carry)$, $(carry) \leftarrow A_7$ Rotate the content of Register A left one bit. Rotate the content of the carry flip-flop into A_0. Rotate A_7 into the carry flip-flop.
RAR	00	011	010	$A_m \leftarrow A_{m+1}$, $A_7 \leftarrow (carry)$, $(carry) \leftarrow A_0$ Rotate the content of register A right one bit. Rotate the content of the carry flip-flop into A_7. Rotate A_0 into the carry flip-flop.

PROGRAM COUNTER AND STACK CONTROL INSTRUCTIONS

JUMP INSTRUCTIONS – Three Bytes
(three cycles – PCI/PCR/PCR)
Normal flow of the microprogram may be altered by jumping to an address specified by bytes two and three of an instruction.

JMP (Jump Unconditionally)	01	XXX <B_2> <B_3>	100	$(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$ Jump unconditionally to the instruction located in memory location addressed by byte two and byte three.
JFc (Jump if Condition False)	01	$0C_4C_3$ <B_2> <B_3>	000	If $(c) = 0$, $(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$. Otherwise, $(P) = (P)+3$. If the content of flip-flop c is zero, then jump to the instruction located in memory location $\langle B_3 \rangle \langle B_2 \rangle$; otherwise, execute the next instruction in sequence.
JTc (Jump if Condition True)	01	$1C_4C_3$ <B_2> <B_3>	000	If $(c) = 1$, $(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$. Otherwise, $(P) = (P)+3$. If the content of flip-flop c is one, then jump to the instruction located in memory location $\langle B_3 \rangle \langle B_2 \rangle$; otherwise, execute the next instruction in sequence.

CALL INSTRUCTIONS – Three Bytes
(three cycles – PCI/PCR/PCR)
Subroutines may be called and nested up to seven levels.

CAL (Call subroutine Unconditionally)	01	XXX <B_2> <B_3>	110	$(Stack) \leftarrow (P)$, $(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$. Shift the content of P to the pushdown stack. Jump unconditionally to the instruction located in memory location addressed by byte two and byte three.
CFc (Call subroutine if Condition False)	01	$0C_4C_3$ <B_2> <B_3>	010	If $(c) = 0$, $(Stack) \leftarrow (P)$, $(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$. Otherwise, $(P) = (P)+3$. If the content of flip-flop c is zero, then shift contents of P to the pushdown stack and jump to the instruction located in memory location $\langle B_3 \rangle \langle B_2 \rangle$; otherwise, execute the next instruction in sequence.
CTc (Call subroutine if Condition True)	01	$1C_4C_3$ <B_2> <B_3>	010	If $(c) = 1$, $(Stack) \leftarrow (P)$, $(P) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$. Otherwise, $(P) = (P)+3$. If the content of flip-flop c is one, then shift contents of P to the pushdown stack and jump to the instruction located in memory location $\langle B_3 \rangle \langle B_2 \rangle$; otherwise, execute the next instruction in sequence.

In the above JUMP and CALL instructions $\langle B_2 \rangle$ contains the least significant half of the address and $\langle B_3 \rangle$ contains the most significant half of the address. Note that D_6 and D_7 of $\langle B_3 \rangle$ are "don't care" bits since the CPU uses fourteen bits of address.

RETURN INSTRUCTIONS — One Byte
(one cycle — PCI)
A return instruction may be used to exit from a subroutine; the stack is popped-up one level at a time.

RET	00 XXX 111	(P)←(Stack). Return to the instruction in the memory location addressed by the last value shifted into the pushdown stack. The stack pops up one level.
RFc (Return Condition False)	00 $0C_4C_3$ 011	If (c) = 0, (P)←(Stack); otherwise, (P) = (P)+1. If the content of flip-flop c is zero, then return to the instruction in the memory location addressed by the last value inserted in the pushdown stack. The stack pops up one level. Otherwise, execute the next instruction in sequence.
RTc (Return Condition True)	00 $1C_4C_3$ 011	If (c) = 1, (P)←(Stack); otherwise, (P) = (P)+1. If the content of flip-flop c is one, then return to the instruction in the memory location addressed by the last value inserted in the pushdown stack. The stack pops up one level. Otherwise, execute the next instruction in sequence.

RESTART INSTRUCTION — One Byte
(one cycle — PCI)
The restart instruction acts as a one byte call on eight specified locations of page 0, the first 256 instruction words.

RST	00 AAA 101	(Stack)←(P),(P)←(000000 00AAA000) Shift the contents of P to the pushdown stack. The content, AAA, of the instruction register is shifted into bits 3 through 5 of the P-counter. All other bits of the P-counter are set to zero. As a one-word "call", eight eight-byte subroutines may be accessed in the lower 64 words of memory.

INPUT/OUTPUT INSTRUCTIONS

One Byte
(two cycles — PCI/PCC)
Eight input devices may be referenced by the input instruction

INP	01 00M MM1	(A)←(input data lines). The content of register A is made available to external equipment at state T1 of the PCC cycle. The content of the instruction register is made available to external equipment at state T2 of the PCC cycle. New data for the accumulator is loaded at T3 of the PCC cycle. MMM denotes input device number. The content of the condition flip-flops, S,Z,P,C, is output on D_0, D_1, D_2, D_3 respectively at T4 on the PCC cycle.

Twenty-four output devices may be referenced by the output instruction.

OUT	01 RRM MM1	(Output data lines)←(A). The content of register A is made available to external equipment at state T1 and the content of the instruction register is made available to external equipment at state T2 of the PCC cycle. RRMMM denotes output device number ($RR \neq 00$).

MACHINE INSTRUCTION

HALT INSTRUCTION — One Byte
(one cycle — PCI)

HLT	00 000 00X or 11 111 111	On receipt of the Halt Instruction, the activity of the processor is immediately suspended in the STOPPED state. The content of all registers and memory is unchanged. The P-counter has been updated and the internal dynamic memories continue to be refreshed.