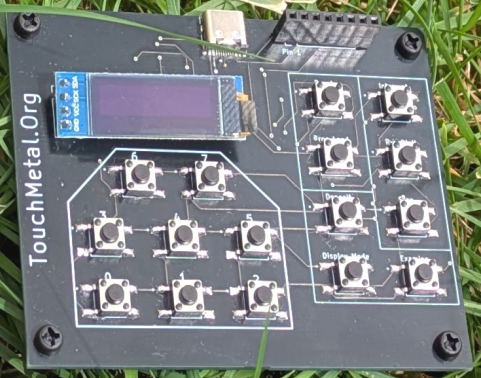
Touch Metal

Programming at the machine-language level



An introduction to assembly language and computer architecture

Nicholas J. Macias

Contents

Foreword vii

1 Introduction 1

1.1 What's this all about? 2

1.2 The Metal 3

2 Getting Started 5

2.1 Introducing The Board 5

2.2 Turning On The Board 7

2.3 Display Details 8
2.4 The Buttons 10
2.4.1 Display Modes 13

3 Intro to Computer Architecture 17
3.1 Where Do We Store Information? 18
3.2 Accessing Memory 19
3.2.1 Octal numbers 20
3.2.2 Accessing Specific Memory Locations 21
3.2.3 Changing Memory Contents 22
3.3 Why Store Numbers In Memory? 23
3.3.1 What operations can be performed? 24
3.3.2 How are these operations specified? 24
3.3.3 Data or instruction? 25
3.4 The Computer's Instruction Set 27
3.5 Some First Programs 29
3.5.1 A program that loops 31
3.5.2 Another looping program 34
3.5.3 A more interesting loop 35

4 Writing Programs - Part 1 39
4.1 Types of Instructions 40
4.2 Coding Instructions In Numbers 41
4.2.1 Bytes and bits 41
4.2.2 Number of bytes per instruction 43
4.2.3 Types of operands 45
4.3 Processor Status Word (PSW) 46
4.4 Instruction Summary 49
4.4.1 Arith/logic insts, 1 operads 50
4.4.2 Arith/logic insts, 2 operands 52
4.4.3 I/O Instructions 54
4.4.4 Move instruction 55
4.4.5 Load instruction 56
4.4.6 Jump instructions 57
4.4.7 Call and return instructions 59
4.4.8 Halt instruction 61
4.4.9 No-operation instruction 62
4.4.10 Interrupt-related instructions 62

5 Interrupts 65

5.1 Motivation 65

5.2 Complexities 67

5.3 Our CPU's Interrupt System 68

5.3.1 Interrupt Behavior 69

6 Writing Programs - Part 2 71

6.1 Assembly Language Overview 73

6.2 Language Elements 74

6.2.1 Numbers 74

6.2.2 Instructions 75

6.2.3 Comments 76

6.2.4 Directives 77

6.2.5 Labels 78

6.3 Sample Assembly Language Program 79

6.4 Analysis of Sample Program 80

6.5 Conversion Into Machine Code 80

6.6 Is This Really How It's Done? 85

7 Other Features 87

7.1 Power-Up Initialization 87

7.2 Saving Programs for Future Use 88

7.3 Special Locations 90

7.4 Program Load 91

8 The Assembler 95

8.1 Using the Assembler 95

9 Next Steps 101

A Instruction Summary 105

B Index 109

C Notes 111

Foreword

This book, the corresponding hardware board and the website (`https://touchmetal.org`) came out of a pet project that began in February 2026. I was asked about doing a high-school outreach event at the college where I work, and was thinking about what to do that might be new, original and interesting to students. I realized that this is a time when most high school students routinely engage with AI chat bots; sometimes use AI for their homework; have perhaps done vibe-coding. Rather than compete with all that, I thought about offering them something very different: a chance to explore programming from the other end of the spectrum: the "bare-metal level."

This document, the design of the board, and all code were produced without the use of AI, generative technology, grammar-checking/suggesting software, chatbots, or other assistive tools except a simple spell checker. For more information, see `https://real-i.org`

I considered coding an emulator for a simple 8-bit processor, perhaps something web-based or as a phone app. As I thought about this though, it felt like "more of the same": more screen time, more pointing and clicking, ho hum what's new. I wanted to offer a physical experience, a chance to push actual mechanical buttons; hear and feel them click; see the immediate response. Thus was born the idea of a physical board.

The first prototype was done on an Arduino, which worked well and was relatively easy to make, but felt a little clunkier than I hoped. So I switched to a custom PCB with a PIC18F27K42, a USB interface, an LCD display and 16 buttons. This is the present state of the board.

In developing the architecture of the system, I tried to focus on education rather than general-purpose usefulness. The memory is 256 bytes, which makes addressing very simple. Most data is stored in a set of 8 general purpose registers. While it's possible to treat memory addresses as operands for instructions, I've found this is some-

times a bit more confusing to students, so memory is only usable as an operand via indirect-register access. Probably not what you would choose for a general-purpose architecture, but it puts the second type of operand in the background, and allows a focus on register-based data.

There is only a single pair of input/output registers. Enough for some simple experiments, while avoiding the notion of indexing I/O registers or doing memory-mapped I/O. Similarly, there's only a single interrupt available (via the Interrupt button).

The system is very octal-oriented, which is a little easier step from decimal than, say, hexadecimal. Accordingly, the coding of instructions was chosen so that conversion from assembly code to machine code can be done easily: nothing crosses the octal-digit boundaries, and there is a lot of symmetry among the different addressing modes.

My hope is that this project will intrigue and inspire, not everyone, but some of those who are drawn to asking how and why things are as they

are, and how they might be changed to something better or at least different. Feel free to get in touch via the `touchmetal.org` website.

Acknowledgement

None of this work would have happened without the continual love, support and encouragement of my wonderful, amazing wife Corinna. She has been and continues to be the constant source of my creativity, enthusiasm and positive outlook in all things. I love you forever!

Vancouver, WA, April 2026

1
Introduction

Welcome to *Touch Metal*! If you're curious about computers, about what makes them work, about what happens inside their outer shells, then this book is for you. If you're a programmer who has wondered how your programs translate into something that can be used by a bunch of logic gates, then this book is for you. If you want to learn more about assembly language, machine code and computer architecture, then this book is for you too!

I hope you'll stay a while and see what's here, maybe download some of the resources from touchmetal.org, and maybe even get a board so you can

touch metal yourself!

1.1 What's this all about?

Sometimes a certain level of detachment can be useful. If you're hungry, you can go to a fast food restaurant, get something to eat, and satisfy your hunger, without ever thinking about cooking, cutting vegetables, and so on. It's quick, it's efficient, and it solves the immediate issue of being hungry.

Still, there are people who find great delight in cooking food themselves, and if you understand the nuances of food preparation, you can have a very different experience satisfying your hunger: deciding what to eat, how to prepare it, adjusting the ingredients and spices and other details precisely to your liking. It is the *process* that is satisfying, along with the act of eating. The fact of no longer being hungry is almost secondary.

So it goes with coding. Today, it is possible to create a fully functioning program without understanding **anything** about the underlying com-

puter. That may be great for a company looking to make a quick profit; but for someone who is curious, who wants to understand **how** things work, the experience will likely be uninspiring.

In the spirit of "touch grass," I offer here a similar suggestion: touch metal.

1.2 The Metal

"The metal" is the part of a computer that is generally hidden from direct view and access. It is the low-level CPU (central processing unit) and its internal structure: the program counter, processor status word and other internal registers, the general registers, the stack, the memory. The term "bare-metal programming" refers to writing, loading, testing and debugging code without most of the tools commonly used today. In its purest form, there is no editor, no compiler, no assembler, no loader, no debugger. You write in *machine code*, the low-level language of the CPU. You may have to do this manually, on paper. At the metal level, you may have only a few switches for

entering that code into the machine's memory and controlling its execution. There may be some lights or a simple display to see what is happening.

While this may seem like a tedious level at which to work, it is in fact sometimes the easiest level for low-level control of a system, as well as for debugging a system. When you touch metal, you're in contact with what is actually happening inside the CPU. Whatever it does, there is a very simple reason for it, and that reason is precisely because you told it to do so. When things go wrong, then tend to go horribly wrong: a single 1 or 0 in the wrong place can easily wipe out critical sections of code/memory and crash the entire machine. When things go right though, the feeling is amazing!

If any of this is appealing, or if you're simply curious, then let's get started exploring what a computer *really* is.

2
Getting Started

2.1 Introducing The Board

Figure 2.1 shows the layout of the __ board. Note the following main pieces:

- The USB port is used to supply power via a USB-C connector.
- A small display, which shows you information about the system. The display is also sometimes used to communicate with you about actions to be taken by the system.
- A numeric keypad (in the lower left of the board) that is used to enter numbers. Note that there are only

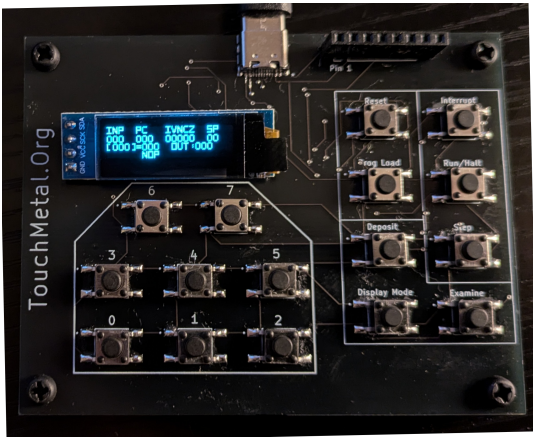


Figure 2.1: Board Overview

8 keys (0-7). At this level of programming, it's common to work in a different number system. In our case, we're using *octal*, which means all out numbers use only the digits 0, 1, 2, 3, 4, 5, 6 and 7.

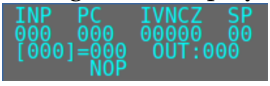
- A second set of keys (on the right) for interacting with the memory of the system, changing the state of the system, and controlling the execution of the system's program.

These pieces will be explained in more detail below.

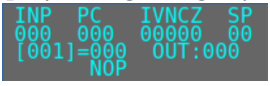
2.2 Turning On The Board

Power is supplied through the USB-C connector at the top of the board. Thee is no power switch: once power is applied, the board turns on. It takes about 1 second for the board to fully turn on. This is the nature of The Metal: there is no operating system to boot, no logging in, no checking for updates. You apply power and it's ready to go – faster than a PC, phone, TVs or music player!

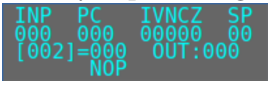
When the board starts, you'll see the following on the display:



If you press the EXAMINE button the display changes slightly:



and if you press it again:



The only change is the number on the 3rd line, which changes from `000` to `001` to `002`.

2.3 *Display Details*

Figure 2.2 shows the different pieces of the display.

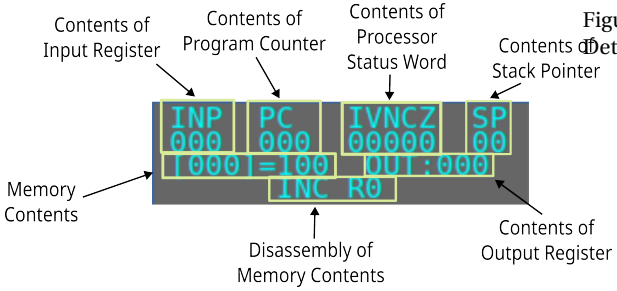


Figure 2.2: Display Details

There are seven pieces of information displayed here (all numbers are displayed in *octal*; see §3.2.1 (“Octal numbers”) for a discussion of what this means):

- The current contents on the Input Register. This is based on the last 3 numeric buttons pressed;
- The contents of the Program Counter (the address of the next instruction to execute);
- The contents of the Processor Status Word, which records certain aspects

of prior instructions. For example, if the CPU executes an ADD instruction, the digit labeled Z will be a 1 or 0 depending on whether the result of the ADD instruction was exactly 0 or not.

- The contents of the Stack Pointer, which is related to remembering from where the CPU was executing instructions;
- The contents of the Output Register, which is a special register that can be written by the CPU;
- Memory Contents, which displays an address inside brackets, followed by the number stored at that address. In Figure 2.2, the display is showing that memory location 000 contains the number 100. The address is normally taken from the Input Register, but is sometimes incremented by pressing Examine or Deposit, and shows the Program Counter when the CPU executes an instruction.
- Disassembly of Memory Contents. This corresponds to the number

shown in the Memory Contents part of the display, and indicates what instruction the displayed number corresponds to. For example, in Figure 2.2, the number 100 corresponds to the instruction “INC R0” See §3.4 (“The Computer’s Instruction Set”) for a discussion of the different instructions that the CPU can execute.

2.4 The Buttons

Some of this will make less sense now than later, but here’s a synopsis of what the different buttons do:

Button	Purpose
0-7	Used to specify the value of the *input register* This value is always shown on the display beneath `INP`

Examine	Displays the contents of the address currently in the Input Register, and make this address the target for the Deposit button. If Examine is pressed a second time, the system displays the next location in memory. See §7.3 for information on examining locations other than memory addresses.
Deposit	Deposit the contents of the Input Register into the memory address previously examined. This also displays the next location in memory. See §7.3 for information on depositing to locations other than memory addresses.
Display Mode	Switch between four different display modes. See §2.4.1(“Display Modes”) for more details.
Step	Ask the CPU to perform a single instruction (at the address specified by the Program Counter).

Run/Halt	Toggle between the Run state (where the CPU repeatedly executes instructions and increments the PC) and the Halt state (where instructions are only executed by the pressing the Step button).
Interrupt	Pressing this button requests a CPU *interrupt*, which causes the usual processing of instructions to be suspended. See Chapter 5 (“Interrupts”) for more details.
Prog Load	When you need a break from touching metal, you can use software tools to convert *assembly language* programs into binary files, and load those files into memory with this button. See Chapter 8 (“The Assembler”) for details.

Reset	Clears the Program Counter, Processor Status Work, Stack Pointer, Output Register, and all general purpose registers (R0-R7). Pressing Reset twice in a row gives the option of also clearing all memory.

2.4.1 Display Modes

The normal display (shown in Figure Display Details) has a lot of information packed into a very small display, which can sometimes be difficult to read. The "Display Mode" button can be used to switch to a larger font (which also displays less information). Pressing the button cycles between the following four modes:

1. the full display;
2. a display showing only the current memory address (the number normally shown in brackets) and the disassembly of the memory contents at that address;

3. a display showing memory contents and the contents of the output register; and

4. a display showing the processor status word and the contents of the input register.

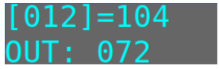
Figure 2.3 shows examples of these four display modes.

The above introduction to the board itself allows us to use the board as we begin to explore the next topic: namely the notion of what exactly a computer is, how it is organized, what it does, and how to control it. The term for these notions is “Computer Architecture.”

Figure 2.3: Initial Display

Display Mode 1: Full Display

Display Mode 2: Instruction Dissasembly
The instruction at memory address 012
is "INC R4"



Display Mode 3: Memory and Output Register
Memory location 012 contains the number 104
The output register contains the number 072

Display Mode 4: Processor Status Word (PSW) and Input Register
The uppercase N and Z mean those pieces of the PSW are 1.
The input register contains the number 012

3
Computer Architecture: A Brief Introduction

What is a computer? In general terms, it is something that examines and performs operations on certain objects. These operations are predictably controlled by something called a *program*. The objects being operated on may be simple numbers, but they could be interpreted as letters, words, formulas, pictures, music, movies, or almost anything else. They could also be physical objects, such as motors, switches and lights. For us, we will focus on manipulation of numbers. Further, we'll stick with *small* numbers: numbers between

0 and 255, or between -128 and +127.

3.1 Where Do We Store Information?

If a computer is going to perform operations on numbers, those numbers need to be stored somewhere. The place they're stored is called a *memory*. In our computer, the memory can store 256 numbers, which we can retrieve, modify and re-store. Since there are so many numbers in the memory, we need to way to indicate which number we are storing or retrieving. Each number is thus given an *address*, which identifies it among the set of all stored numbers.

Here's the slightly weird thing: the address is itself just another number! Since we have 256 numbers, we give them addresses of 0, 1, 2 and so on. Since we call the first address "0," the last address is 255 (instead of 256)[1].

While most of the numbers we want to store and retrieve are saved in memory, there is a second set of locations in which we can store a few numbers: these are called **registers**. There are 8 general purpose registers (called "R0"

[1] This business of starting from 0 and ending one less than you might expect is very common in computing. Though it's generally confusing to everyone at first, eventually it will make more sense than starting from 1.

through “R7”), as well as the *Program Counter* (PC), *Processor Status Word* (PSW) and the *Stack Pointer* (SP). There is also a multi-purpose *Input Register* (INP) and an *Output Register* (OUT). These will be explained in more detail later.

3.2 Accessing Memory

Plug in the board so that the display lights up, as shown in Figure 3.1

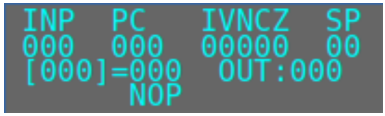


Figure 3.1: Initial Display

There are a number of pieces of information in this display. On the 3rd line, `[000]` indicates that the display is showing you the contents of memory address 000. “`=000`” means the number stored at that address is 000.

If you press the Examine button, the display now looks like Figure 3.2.

The `[001]=000` on the 3rd line means the display is showing you the contents of memory address 001, and the num-

Figure 3.2: Display After Pressing Examine

ber stored there is (again) 000. If you continue pressing the Examine button, you'll see the number inside the [brackets] increase, up to `[007]`. If you press it again, it jumps to `[010]`. What happened to 008 and 009?

3.2.1 Octal numbers

Computers don't really understand numbers the way we do. We write numbers using 10 digits (0-9). There is no particular reason why using 10 digits makes sense; it's probably mainly just because most people have 10 fingers. Computers actually work in a much simpler systems, where numbers are represented using only **two** digits (1 and 0). This system (called "binary") works great for computers, but pretty horribly for humans.

As a compromise, when working at a low level with computers, we some-

times work in **octal**, where we write numbers using 8 digits (0-7). While not as natural to us as using 10 digits, octal has the advantage of being easy to convert to binary, while still having enough digits so we aren't dealing with numbers like 10101001 and 10110101 (in octal, these numbers would be 251 and 265, respectively - a bit easier to deal with). See §4.2.1 for more details on binary numbers and their octal representation.

3.2.2 Accessing Specific Memory Locations

While we can use the Examine button repeatedly to look at successive memory locations, an easier approach is to specify the particular location we wish to examine. This is done using the numeric keys on the lower-left side of the board. Go ahead an press the button labeled "1" and you'll notice the display changes: the number beneath "`INP`" should now display "`001`." Now press the number "2" and the display will now show "`012`." This is the "Input Register." You can keep pressing numbers

and watch the input register change.

At any point, you can press the Examine button, and the display will show you the contents of the memory location whose address is in the Input Register. For example, press 1 2 3 Examine, and the display will look like Figure3.3.

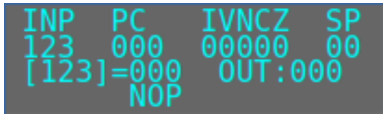


Figure 3.3: Display After Pressing 1 2 3 Examine

This is showing you the contents of memory location 123 (which is 0). If you press Examine again, the display will show you the contents of memory location 124.

3.2.3 Changing Memory Contents

As you may have surmised, all memory locations have a content of 000. That's not very useful! Let's change the contents of memory. Press 1 0 0 Examine. Now press 0 4 2 Deposit. Voila! You've just stored the number 042 at address 100. Don't believe it? Press 1 0 0 Exam-

ine and see what the display shows. You should see `[100]=042`, showing you the new contents you just stored into address 100 (see Figure 3.4).

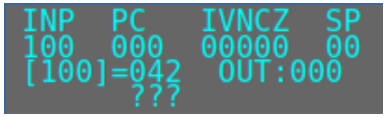


Figure 3.4: Display After Storing 042 in Location 100

You may have noticed that after pressing Deposit, the number inside brackets increased automatically. This is a handy feature. Just as pressing Examine automatically increases the address being examined, so does pressing Deposit.

3.3 Why Store Numbers In Memory?

Okay great, we can store a bunch of numbers in memory and then go back and look at them. So what? Why is this useful? Remember, our definition of a computer was "something that examines and performs operations on certain objects." We know what the object are (small numbers). But what

about the operations? What sorts of operations can be performed? And, how are those operations specified?

3.3.1 What operations can be performed on numbers?

Some of the operations performed on numbers are what you might expect: numbers can be added, subtracted, multiplied and divided. They can be incremented (increased by 1), decremented (decreased by 1) or cleared (set to 0). They can be *rotated*, which means, for example[2], taking a number like 12345 and changing it to 23451 ("rotate left") or 51234 ("rotate right"). There are also *logic operations* that can be performed: and, or, xor. We'll discuss these later.

[2] This is only an example. Remember, we don't even have numbers as large as 12345; and if we did, this is not actually what rotation would look like; but it gives you a rough idea.

3.3.2 How are these operations specified?

This is the coolest part; the thing that makes computers so useful: operations are specified by more numbers, **and those numbers are stored in the**

memory, right alongside the numbers being operated on. Contemplate that for a moment. Inside the memory, there are numbers that represent things to be manipulated, and there are also numbers *that indicate how to manipulate them.* This characteristic creates a self-contained system, and has been a central feature of digital computers since 1936.

In order to use simple numbers to specify whether to, say, add or subtract (and to specify *what* should be added or subtracted), there needs to be some sort of mapping between instructions and numbers. This mapping is part of what's called the *instruction set* of the computer (§3.4). Before going into details of the instruction set, there's one more important question to address.

3.3.3 Data or instruction?

Since the memory contains both numbers to be operated on ("data") and numbers that specify what operations to perform ("instructions"), there needs to be some way to differentiate between

these. This is done using one of the special registers mentioned above: the Program Counter (PC). The PC is another number, which stored the address of an instruction to be executed. In other words, if the PC has a value of 0, this means the contents of memory address 0 should be treated as an instruction and *executed* (meaning the CPU should do whatever that instruction says to do).

Normally, after an instruction is executed, the PC is incremented (increased by 1). So after executing the instruction at memory address 0, the instruction at memory address 1 would be executed, followed by the instruction at memory address 2, and so on.

This is somewhat useful, but always performing the same instructions is a bit limiting; so there are some instructions which change this behavior by loading a new value into the PC. Moreover, this change to the PC can be made conditional on the result of previous operations. This means the CPU can effectively make decisions.

3.4 The Computer's Instruction Set

So we know that the computer has numbers stored in its memory, and that sometimes those numbers tell the computer to take some action. The set of actions ("instructions") available on a computer, and the way those actions are represented, are part of what's called the "Computer's Architecture." Some computers have only a few instructions; others have thousands of different instructions. Our system has a relatively small number of instructions, broken into just a few different types. Chapter 4 will cover this instruction set in detail. For now, let's just play with a few simple examples.

Some instructions can be specified by a single number: "NOP" is an instruction that does nothing; "HALT" is an instruction that stops the CPU from executing further instructions; "INC R0" is an instruction that adds 1 to the general purpose register named "R0." Try the following:

- Press 0 0 0 Examine

The display should look like Figure 3.1. The 3rd line shows `[000]=000` which means the contents of memory address 000 is 000. The 4th line displays the word "`NOP`" This is telling you what instruction corresponds to the number 000. Now press 0 0 2 Deposit 0 0 0 Examine. The display now looks like Figure 3.5.

Figure 3.5: Display After Storing 002 in Address 000

`[000]=002` shows you the contents of address 000 is the number 002 (because you just deposited that); the bottom line shows "`HALT`" which means 002 is the code for the HALT instruction.

Now press 1 0 0 Deposit 0 0 0 Examine. The display should look like Figure 3.6

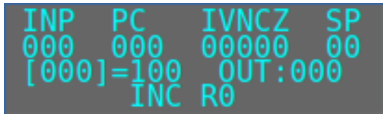


Figure 3.6: Display After Storing 100 in Address 000

Here, you've stored the number 100

in address 000, and when you look at address 000, you see it contains 100, but you also see that 100 corresponds to the instruction “INC R0” which adds 1 to the contents of register R0.

3.5 Some First Programs

Let’s start writing programs! To begin, let’s clear all memory first. Rather the unplugging the power, you can press the Reset button twice in a row, which brings up the following on the display:

```
PRESS:
   1: ERASE MEMORY
   0: CANCEL
```

Pressing 1 will erase all memory; anything else will leave memory unchanged. So do the following:

Press RESET RESET 1 1

This clears the memory, filling each memory location with 000. 000 is the code for the “NOP” instruction, which does nothing. Now

press STEP and notice how the display changes: specifically, the PC changes to 001, and the display shows

`[001]=000`. If you press STEP again, you'll see the PC change to 002. Remember, the PC is the *program counter*, which determines the memory address of the next instruction to execute. Each time you step, the NOP instruction is executed, and the PC is incremented.

If you press STEP enough times, the PC will increment from 007 to 010. That's' the octal numbering (§3.2.1) discussed previously.

Now press the RUN/HALT button and watch what happens. You'll see the PC continually increment on the display, as does the line showing memory contents. Since every memory address contains 000 – which is the code for the NOP instruction – the CPU just keeps fetching the next instruction from memory, doing nothing, fetching the next instruction, and so on.

Once the PC reaches 377, it will increment, not to 378 (that's not a legal number), and not to 400 (the next octal number after 377), but to 000. The PC has *rolled over*, meaning it tried to go beyond the largest possible value, and thus reset to 000 (the smallest or "first"

number).

If you press RUN/HALT again, the CPU will halt, and the display should remain stable. Press it again the it will continue running from where it left off. When it's halted, you can also press STEP and advance the CPU a single instruction.

Technically, this is a program, but it's not a very interesting one; it's just a bunch of 0's stored in every memory address. Let's make it a bit more interesting.

3.5.1 A program that loops

Let's try adding a *loop* to our program. This will cause the CPU to execute a small set of instructions repeatedly. First, press RUN/HALT until the CPU is halted (pressing EXAMINE will also halt the CPU). Now press the following:

0 0 3 EXAMINE 0 0 4 DEPOSIT RESET

You've just written a small program! But what does it do? Press

RESET

and then start pressing STEP. You should see the display change as in

Figure 3.7

You are executing a 4-instuction program loop. If you press RUN/HALT you can watch it run at a higher speed.

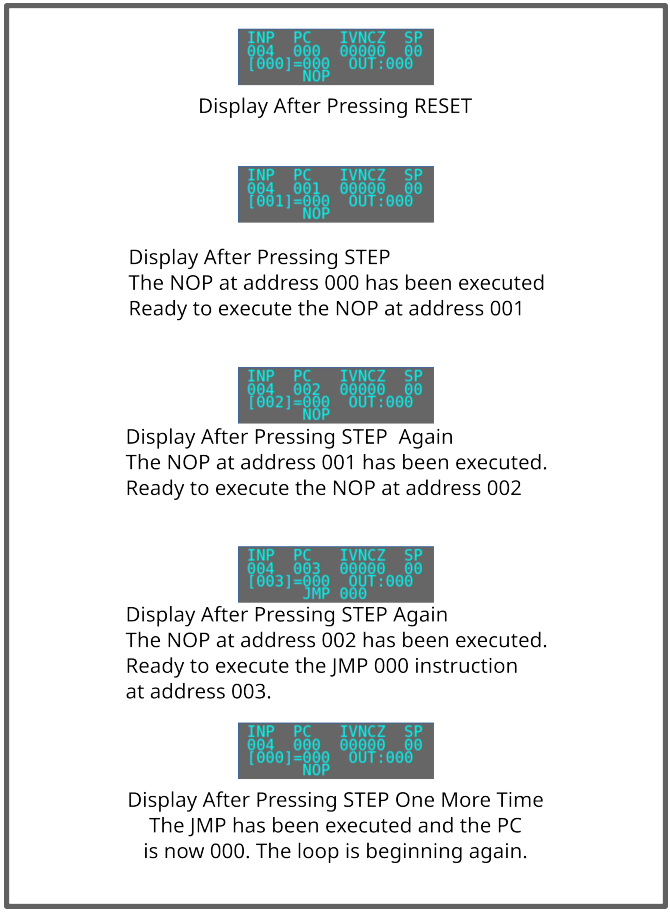
If we were to write out the details of this program, it would look like this:

```
ADDRESS     NUMBER       INSTRUCTION       COMMENTS
 000         000            NOP            Do nothing
 001         000            NOP            Do nothing
 002         000            NOP            Do nothing
 003         004            JMP 000        Set PC to 000
 004         000          (this 000 is part of the
                           "JMP 000" instruction)
```

There are a number of things going on here. The first three memory addresses (000-002) all contain 000, which is the NOP instruction. NOP does nothing, but after it is executed the PC is increased by 1, which causes the next instruction to be executed. But memory location 003 contains the number 004. 004 is the coded way of saying "Change the PC to something other than the next memory address." This is called a "jump" instruction.

An important thing to note about the jump instruction is that *it actually*

Figure 3.7: Running a Simple Loop



requires two *numbers instead of just one*. The full instruction

JMP 000

is stored as a pair of numbers:

- 004 is the code for "jump"; and
- 000 says where to jump to, i.e., what the new value of the PC should be.

Since the second number is 000, the jump instruction is saying "Change the PC to 000." This is why, after executing the instruction **JMP 000**, the program again begins executing the instruction at address 000. §4.2.2 discusses in more detail instructions that require two numbers.

3.5.2 Another looping program

Let's change this behavior. Press the following:

0 0 2 EXAMINE 0 0 4 DEPOSIT 0 0 1 DEPOSIT RESET

Now press the STEP button and observe what happens. The PC will change from 000 to 001, then 001 to 002, then back to 001, 002, 001, 002,

and so on. We've made a smaller loop. Here's what the program looks like:

```
ADDRESS     NUMBER       INSTRUCTION       COMMENTS
 000         000            NOP            Do nothing
 001         000            NOP            Do nothing
 002         004            JMP 001        Set PC to 001
 003         001         (this 001 is part of the
                          "JMP 001" instruction)
 004         000            NOP            This is never
                                           executed
```

As you can see, the instruction at address 002 is now a JMP, and it says JMP 001, which means "set the PC to 001." So after you press RESET, the PC is set to 000, and that address contains a NOP. The next instruction (address 001) is another NOP; but at 002, the instruction says "Set the PC to 001." So the program executes the instruction at address 001, then 002, then back to 001, then 002, and so on. This is a very small loop.

3.5.3 A more interesting loop

Let's do something more than just loop; let's change the contents of a register,

and display it in the output register.
Here's the program we're going to enter:

```
ADDRESS  NUMBER  INSTRUCTION    COMMENTS
 000      100      INC R0       Add 1 to the value of
                                register R0
 001      170      OUT R0       Copy contents of R0
                                to the output register
 002      004      JMP 000      Set PC to 000
                               (so repeat the above
                                forever)
 003      000                   (this 000 is part of the
                                "JMP 000" instruction)
```

This will repeatedly increment the value of register R0, and copy R0 to the output register (which is visible on the display).

To enter this program, type the following (comments are written in italics):

RESET RESET 1 EXAMINE ; this clears the memory of the computer

0 0 0 EXAMINE ; this shows the contents of address 000

1 0 0 DEPOSIT ; this deposits the number 100 into the current address (000) and changes the current address to 001

1 7 0 DEPOSIT ; this deposits the

number 170 into address 001

0 0 4 DEPOSIT ; this deposits the number 004 into address 002

If you now press RUN/HALT you'll see the program run, and the value of the output register (the number displayed following "`OUT:`") will increase each time the program loops. You can press STEP repeatedly to run the program one instruction at a time. You can also press DISPLAY MODE while the program is running to observe different aspects of the system: either the PC + current instruction; the memory contents + output register; the just the flags (which will change very slowly) and input register; or back to the full display.

You can also press 0 0 0 EXAMINE and then press EXAMINE multiple times to see your program in memory.

Okay, we've written a few sample programs and watched them execute. Time for the next step, which is **writing your own programs!**

4
Writing Your Own Programs - Part 1

This chapter will explain the connection between numbers in memory and the instructions they represent. With that understanding, it's then relatively easy to write your own programs to do what you want. Keep in mind though, the available instructions tend to be very simple: add two numbers, subtract one from a number, and so on. Most instructions do a single simple operation, which means more complex behavior is made up of multiple instructions.

This board is primarily a learning system, so the memory is quite lim-

ited: only 256 numbers can be stored in the main memory, which means at most 256 instructions can be stored. Remember though that some instructions (like "JMP") require two numbers for their coding, so typically your program will need to have fewer than 256 instructions.

4.1 Types of Instructions

Before looking at the detailed mappings between instructions and their coding as numbers (§4.4), it will be useful to look at the entire instruction set, in terms of the types of instructions that are available. In what follows, the term "operand" refers to a piece of data that is acted upon by the instruction.

The instructions break out into roughly the following categories:

- arithmetic, rotate and logic instructions for one operand;
- arithmetic and logic instructions for combining two operands;
- an input ("IN") and an output ("OUT") instruction;

- a move instruction for copying one operand to another;
- a load instruction (“LDI”) for copying a given number into an operand;
- 9 instructions for unconditional and conditional jumps (which change what instruction is executed next);
- call (“CALL”) and return (“RET”) instructions for temporarily executing a different set of code and then returning to where you were;
- a halt instruction (“HALT”) for stopping the CPU from running;
- a no-operation instruction (“NOP”) which does nothing; and
- three interrupt (Chapter 5) instructions (“INTE” “INTD” and “RTI”).

4.2 Coding Instructions In Numbers

4.2.1 Bytes and bits

We’ve been using the term “number” for our small numbers, but the tech-

nical term is "byte." A byte is simply a small number, typically from 0 to 255, or from -128 to +127. More precisely, it is a number that can be stored in 8 bits. A bit is a single piece of information, often thought of as simple a 1 or a 0. If we have 8 such bits, we have 8 1's and 0's. Depending on how we interpret these 8 bits, they can mean different things. For us, we'll continue to think in terms of small numbers. Suppose we have the following 8 bits:

0 0 1 1 0 1 0 1

We can think of this as a number by doing what we do with regular numbers: thinking of each digit as a multiplier. When you see a number "1273" you know this means $1 \times 1000 + 2 \times 100 + 7 \times 10 + 3$ (which is, of course, 1273). With "binary numbers" we do exactly the same thing, but instead of multiplying by 1000, 100, 10 and so on, we'll multiply each digit by 128, 64, 32, 16, 8, 4, 2 and 1. So a binary number such as **00110101** represents $\mathbf{0} \times 128 + \mathbf{0} \times 64 + \mathbf{1} \times 32 + \mathbf{1} \times 16 + \mathbf{0} \times 8 + \mathbf{1} \times 4 + \mathbf{0} \times 2 + \mathbf{1} \times 1$ (which adds up to 53).

As mentioned previously, writing numbers in binary can be error prone; humans aren't very good at talking about bunches of 1's and 0's without getting confused. Instead, we write our numbers in octal. In octal, we only need 3 digits to represent an 8-bit number. We do this by breaking the 8 bits into three groups. Continuing with our example number **00110101** this looks like:

00 | 110 | 101

Then we write a single digit for each of those groups, using table 4.1 (for the first group – 00 in this example – we just append a 0 in the front). So for our example, the octal number would be 065:

```
00 110 101
0   6   5
```

4.2.2 Number of bytes per instruction

There are some instructions such as NOP which are represented by a single number (000); but there are other instructions such as JMP which require

bits	digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Table 4.1: Digits Corresponding to Octal Bit Patterns

two numbers: 004 followed by the address of the instruction to which the CPU should jump. Thus, when looking at the numbers corresponding to a program, one needs to start from the beginning in order to interpret those numbers. For example, if you see the number 000, it could be the "NOP" instruction, or it could be part of a "JMP 000" instruction.

There are three types of instructions that require two bytes:

1. calls and jumps;

2. arithmetic, logic and move instructions that require a pair of operands; and

3. the load instruction that requires a value to be loaded into an operand.

All other instructions are specified with a single byte.

4.2.3 Types of operands

A few instructions (HALT, for example) don't take any operands. Most instructions though require one or two operands. There are four different types of operands.

1. An address is a location in memory where the CPU should next start executing instructions. For example, `JMP 137` says to execute the instruction at memory address 137.

2. A register direct operand means the operand is one of the eight general purpose registers R0-R7. For example, `INC R2` says to increment the contents of register R2. If R2 contains the number 013, then after executing the instruction `INC R2` the register will contain the number 014.

3. A register indirect operand (indicated by a register name in paren-

theses, i.e. "(R3)") specifies a general purpose register, but instead of that register being the actual operand, **the register contains the memory address of the operand**. A register indirect operand is specified by placing parenthesis around a register name, for example, `INC (R5)`. In this case, if R5 contains the value 057, `INC (R5)` does not increment R5; **it increments memory location 057.**

4. An immediate operand means the 2nd number of the instruction is the actual value being used. For example, `LDI R2,023` means "set R2 equal to the value 023."

§4.4 gives more specific information about instructions using these different types of operands.

4.3 Processor Status Word (PSW)

A useful computer is generally able to look at itself and make decisions based on what has recently happened. For example, after adding two numbers, the program may wish to do something

special if the sum was exactly 0. Such decisions are handled through the use of conditional jumps. For example the instruction `JZ 100` immediately following an ADD instruction means "if the result of the ADD was exactly 0, then jump to address 100 (otherwise, just continue to the next instruction, as usual)."

In order for conditional jumps to work, it's necessary for the CPU to remember the results of previous calculations. Rather than remember, say, the number that resulted from a previous ADD instruction, the CPU uses a series of flags to record information. A "flag" is just a single bit (a 1 or a 0). The CPU has five flags, which are stored in a special purpose register called the Processor Status Word, or "PSW." The PSW is organized as follows:

-	-	-	I	V	N	C	Z

with the following bit meanings:

- Z is the "zero" bit, and is set to 1 if the prior instruction resulted in 0;
- C is the "carry" bit, and is set to 1 if the prior instruction generated a

"carry";

- N is the "negative" bit, and is set to 1 if the prior instruction resulted in a negative number;
- V is the "overflow" bit, and is set to 1 if the prior instruction resulted in a overflow condition (i.e. the result was too large or too small to fit in a single number); and
- I is the "Interrupt Enable" bit (see Chapter 5).

Remember, our numbers can be thought of as being in the range -128 to +127. So if we add +1 and -1. the result will be 0, and the Z bit will be set. The N bit will be clear, since 0 is not negative. Also, the result (0) is within the range of our numbers, so the overflow bit (V) is also clear. The overflow bit is set when the result is too large or too small to fit in a single number. If you add 120 + 10, the result (130) is too large, so V would be set. Similarly if you add -120 and -10.

The carry bit (C) is a bit more complex to understand. We need to know

that the number -1 is written in binary as 11111111 (this uses something called “two’s complement”), while +1 is written as 00000001. If we add these in binary, we would get:

```
  11111111
+ 00000001
 100000000
```

The extra 1 at the left end of the sum is the carry bit. It comes from adding each column, and carrying an extra bit from each column to the one on the left. It’s exactly the same as adding 99 and 1; if you’re only allowed 2 digits, your answer would be 00; but there is an extra “1” the appears where a 3rd digit would be. This is what the C bit records.

4.4 Instruction Summary

These instructions are arranged in the same categories as above in §4.1.

4.4.1 Arithmetic logic and rotate instruction, one operand

In the coding for these instructions, "r" is 0, 1, 2, 3, 4, 5, 6 or 7 and specifies one of R0, R1, R2, R3, R4, R5, R6 or R7, respectively.

Remember, "reg" means the contents of a register, whereas "(reg)" (see "Types of operands" on page 45) means the contents of the memory address specified by the register.

Instruction	Coding	Flags Affected	Description
INC reg	10r	CNZ	$reg + 1 \rightarrow reg$
INC (reg)	30r	CNZ	The memory address to which reg points is incremented: $(reg) + 1 \rightarrow (reg)$
DEC reg	11r	CNZ	$reg - 1 \rightarrow reg$
DEC (reg)	31r	CNZ	$(reg) - 1 \rightarrow (reg)$
CLR reg	12r	(none)	$0 \rightarrow reg$
CLR (reg)	32r	(none)	$0 \rightarrow (reg)$

NOT reg	13r	NZ	The operand is logically negated, meaning each bit is changed from 1 to 0 or 0 to 1. $\sim reg \rightarrow reg$
NOT (reg)	33r	NZ	$\sim (reg) \rightarrow (reg)$
ROL reg	14r	CNZ	Each bit of the operand is moved one position to the left. The leftmost bit it moved to the rightmost spot, and also copied to the C flag. $reg << 1 \rightarrow reg$
ROL (reg)	34r	CNZ	$(reg) << 1 \rightarrow (reg)$
ROR reg	15r	CNZ	Each bit of the operand is moved one position to the right. The rightmost bit it moved to the leftmost spot, and also copied to the C flag. $reg >> 1 \rightarrow reg$
ROR (reg)	35r	CNZ	$(reg) >> 1 \rightarrow (reg)$

4.4.2 Arithmetic and logic instructions, two operands

These instructions require two bytes: the first indicates the type of operation to perform, and the second specifies the operands. In the following instructions, “msd” is a 3-digit number whose digits are:

s (0-7) is the src register number

d (0-7) is the dst register number

m (0-3) is the access mode of the src and dst registers:

0=src,dst

1=src,(dst)

2=(src),dst

3=(src),(dst)

Instruction	Coding	Flags Affected	Description
ADD src,dst	020 msd	VCNZ	Add $src + dst \rightarrow dst$
SUB src,dst	021 msd	VCNZ	Subtract $src - dst \rightarrow dst$
MUL src,dst	022 msd	VCNZ	Multiply $src \times dst \rightarrow dst$ The result must be between -128 and +127; otherwise the V bit is set.
DIV src,dst	023 msd	VCNZ	Divide $src \div dst \rightarrow dst$ Only the whole number part of the quotient is used. If dst is 0, V will be set.
AND src,dst	024 msd	NZ	Bitwise AND $src \wedge dst \rightarrow dst$
OR src,dst	025 msd	NZ	Bitwise OR $src \vee dst \rightarrow dst$
XOR src,dst	026 msd	NZ	Bitwise XOR $src \oplus dst \rightarrow dst$
MOV src,dst	027 msd	NZ	Copy $src \rightarrow dst$

The bitwise operations act on each

pair of bits from the two operands, according to the following logic table:

operand 1's bit	operand 2's bit	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

So for example, if we write two numbers in binary (their octal representation is also shown), the AND would be computed as follows:

10101100=254
00101110=056
00101100=054

the OR would be:

10101100=254
00101110=056
10101110=256

and the XOR would be:

10101100=254
00101110=056
10000010=202

4.4.3 Input ("IN") and output ("OUT") instructions

These instructions move data from the input register (which is changed by

pressing numeric keys on the board) to an operand, or from an operand to the output register (which is displayed on the board's display).

Instruction	Coding	Flags Affected	Description
IN reg	16r	(none)	Copy input register to reg $input\ register \rightarrow reg$
IN (reg)	36r	(none)	$input\ register \rightarrow (reg)$
OUT reg	17r	(none)	Copy reg to output register $reg \rightarrow output\ register$
OUT (reg)	37r	(none)	$(reg) \rightarrow output\ register$

4.4.4 Move instruction

There are four forms of the move instruction, which moves a number from a source operand to a destination operand. In each, "rs" is a register specifying the source operand, and "rd" is a register specifying the destination operand. Even though this instruction is called "move," it is in fact a copy operation: the source operand is copied to the destination operand; the source operand is not changed by this opera-

tion.

Instruction	Coding	Flags Affected	Description
MOV R**s**,R**d**	027 0**sd**	NZ	Move register to register $Rs \rightarrow Rd$
MOV R**s**,(R**d**)	027 1**sd**	NZ	Move register to register-indirect $Rs \rightarrow (Rd)$
MOV (R**s**),R**d**	027 2**sd**	NZ	Move register-indirect to register $(Rs) \rightarrow Rd$
MOV (R**s**),(R**d**)	027 3**sd**	NZ	Move register-indirect to register-indirect $(Rs) \rightarrow (Rd)$

4.4.5 Load instruction

The load instruction ("LDI") is used to copy any given number into a register. This instruction thus required two bytes: the first indicates this is a LDI instruction and which register is being loaded; and the second is the number

("nnn") to be loaded into that register.

Instruction	Coding	Flags Affected	Description
LDI R**d**,nnn	07**d** nnn	(none)	$nnn \rightarrow Rd$

4.4.6 Jump instructions

In these instructions, "aaa" is a number indicating the address in memory of the next instruction to execute.

Instruction	Coding	Flags Affected	Description
JMP address	004 aaa	(none)	Jump to location aaa The next instruction to be executed is the one at memory address aaa $aaa \rightarrow PC$
JZ address	005 aaa	(none)	Jump if Z (zero) bit is SET
JNZ address	006 aaa	(none)	Jump if Z bit is CLEAR
JN address	007 aaa	(none)	Jump if N (negative) bit is SET
JNN address	010 aaa	(none)	Jump if N bit is CLEAR
JC address	011 aaa	(none)	Jump if C (carry) bit is SET
JNC address	012 aaa	(none)	Jump if C bit is CLEAR
JV address	013 aaa	(none)	Jump if V (overflow) bit is SET
JNV address	014 aaa	(none)	Jump if V bit is CLEAR

4.4.7 Call and return instructions

Normally, the CPU executes consecutive instructions in memory, unless a jump instruction directs it to begin executing instructions from a different location.

A CALL instruction is similar to a jump, except that before executing the instruction (at the new address), it remembers the address of the instruction that it would normally execute next (this is called the "return address").

The return instruction (RET) is like a jump to that return address.

Thus, a CALL can be used to temporarily begin executing a different part of the program, at the end of which a RET instruction will return to what would normally have been executed after the call.

This call/return mechanism is used for executing what are commonly called "subroutines" or "functions": pieces of code that are generally useful and may want to be run from different places in your program. For example, maybe your program wants to periodically

check the input register for a certain value. Rather than writing the instructions for that check again and again, they would be written once, and that set of instructions could be called using the CALL instruction.

While the return address could simply be saved in a special purpose register, it's possible (in fact, it's likely) that subroutines will sometimes call other subroutines, in which case more than one return address may need to be remembered. There is a part of the CPU named the "stack" which is used for remembering return addresses. It can remember up to 32 return addresses. There is another special purpose register called the "stack pointer" ("SP") which indicates how many return addresses are currently saved on the stack. This is necessary to know, so that the correct return address can be used when a return is executed.

As with the jump instructions, "aaa" is the address in memory of the next instruction to execute.

Instruction	Coding	Flags Affected	Description
CALL address	003 aaa	(none)	Save the address of the following instruction on the stack, then execute the instruction at memory address aaa. $next\ PC \rightarrow stack$ $aaa \rightarrow PC$
RET	001	(none)	Return from a CALL $value\ on\ stack \rightarrow PC$

4.4.8 Halt instruction

This instruction stops the CPU from running; once executed, the CPU no longer executes instructions until it is re-started (by pressing the RUN/HALT button).

Instruction	Coding	Flags Affected	Description
HALT	002	(none)	Halt the CPU. No further instructions are executed until the RUN/HALT button is pressed.

4.4.9 No-operation instruction

The No-Operation (“NOP”) instruction is really just a space filler. Memory always contains something, and by default it contains all 0's; so the number 0 corresponds to the instruction NOP, which changes nothing. Once executed, the CPU simply reads the next instruction in memory and executes it.

Instruction	Coding	Flags Affected	Description
NOP	000	(none)	No operation

4.4.10 Interrupt-related instructions

Interrupts are described in more detail in Chapter 5. Basically, they are a way to trigger a CALL (to a pre-determined address) in response to an external event (a button press in our case). For this to happen, the interrupt system needs to be enabled (otherwise the button press is ignored). Once enabled, interrupts can later be disabled.

After an interrupt has triggered a CALL, a return needs to happen in order to go back to where the system was before the interrupt. For returning from

an interrupt, there is a special return instruction: “Return From Interrupt” (“RTI”).

Instruction	Coding	Flags Affected	Description
INTE	030	(none)	Enable Interrupts
INTD	031	(none)	Disable interrupts
RTI	032	(none)	Return from the interrupt code and re-enable interrupts after the return.

5
Interrupts

5.1 Motivation

Suppose our CPU were more complex; say it was controlling a number of systems in a factory. It might be regulating the flow of water through a cooling system, the pressure of steam driving an engine, the movement of conveyor belts carrying materials from one part of the factory to another, and perhaps hundreds of other systems. Also, suppose the system needs to have a STOP button; a single button that, when pressed, shuts down all these systems in an orderly way. So the CPU could periodically check the state of the STOP

"Knock knock."
"Who's there?"
"The interrupting cow."
"The interru"
"MOOOOO!"

button, and if it finds it pressed, it could go ahead and perform the shutdown procedures.

But how often should it check that button? Every second? Every 10 seconds? And if the program is very complex, this check would need to appear in multiple places in the code - wherever the code may be executing when the button happens to be pressed. This is impractical.

A solution is to use an interrupt to respond to the button press. Basically, the system is set up so that when the STOP button is pressed, it causes the CPU to pause what it is doing and immediately jump to a specified set of code (called an "interrupt handler"). That interrupt handler would then double-check that the button was pressed, and once confirmed, it would begin the shutdown procedures. The advantage of this approach is that, if desired, the interrupt can occur at any point in the program, regardless of what else the program is doing.

Of course, there may be certain parts of the control program that should

not be interrupted (for example, doing some short but critical movement of a piece of equipment). So there should be a way to enable and disable interrupts at will. Note also that typically, one might not want an interrupt handler to be interrupted. Thus, when an interrupt handler is called, interrupts may be automatically disabled.

5.2 Complexities

On a more complex system, there might be numerous types of interrupts, each with a different priority level. For example, a mouse click might generate an interrupt, but at a lower priority than, say, the arrival of a network message. If the system were processing a mouse click when a network message arrived, the mouse handler might be interrupted so the network handler could run. Systems can easily have hundreds of possible sources of interrupts, with dozens of different priority levels. There may be multiple steps involved in enabling or disabling particular interrupts; and

some interrupts can be tailored even further (for example, an interrupt might be generated when a button is pressed, or when it is released, or perhaps both). Some systems may also contain non-maskable interrupts, meaning they cannot be disabled.

Fortunately, the interrupt system of our CPU is extremely simple.

5.3 Our CPU's Interrupt System

Our CPU has a single interrupt, which is triggered by pressing the "Interrupt" button. By default, interrupts are disabled, so pressing this button has no effect.

Interrupts can be enabled by executing the INTE instruction. They can be disabled by executing the INTD instruction.

The current state of the interrupt system is reflected by the processor status word (PSW)'s I bit (see §4.3). When this bit is 1, interrupts are enabled; otherwise interrupts are disabled.

5.3.1 Interrupt Behavior

When an interrupt occurs, the following take place:

1. the currently-executing instruction finishes.

2. After that, the current PSW is saved on the stack (§4.4.7). This is important because we may be getting ready to do a conditional jump (which depends on the PSW flags), but the interrupt handler may change the PSW flags. Saving the PSW thus ensures that after the interrupt handler is finished, the PSW will be in the same state it was immediately prior to the interrupt.

3. The PC of the next instruction is also saved on the stack. This will be used to return to the code that was running prior to the interrupt.

4. The I flag of the PSW is cleared, thus disabling further interrupts.

5. Finally, the PC is set to 0376 (the 2nd to the last available memory address). This address will usually

contain a JMP instruction, which jumps to the start of the interrupt handler's code.

At the end of the interrupt handler, a Return From Interrupt (RTI) instruction is executed. This does the following:

1. Restore the PC to the value that was saved in the stack.
2. Restore the PSW to the value that was saved in the stack. This also re-enables interrupts (since the I bit of the PSW is restored).

These steps cause the interrupt handler to effectively exit, and the system resumes processing where it was before the interrupt occurred.

6
Writing Your Own Programs - Part 2

In Chapter 4 we looked at the instruction set of our CPU, what these instructions do and how they are coded in numbers. That's important information, but by itself does not really explain how to write a complete program yourself.

If you use something like Cursor for vibe-coding, you may simply describe the program you want to create, and tools may create the program for you. If you write programs in C, you use a compiler to convert your program into machine code - the actual numbers that

the CPU understands. Other languages (Python and Java, for example) include tools that work with a higher-level machine code.

In all these cases, your description of a desired program may be fairly high-level. Remember, we're trying to touch metal: to get back to basics. So we want to work closer to the actual level of the CPU. Now, we could write programs directly in machine code, but that's fairly challenging. Instead, we will work in what's called "assembly language." Assembly language is similar to machine code (the actual numbers that are stored in memory and make up the program that the CPU executes), but assembly language is meant to be human-readable. It uses mnemonics – short abbreviations – in place of the numeric codes that the CPU understands. You've seen these mnemonics, in the sample programs of §3.5 and the tables of, for example, §4.4.1. They are symbols like "NOP" or "JMP" or "HALT."

6.1 Assembly Language Overview

Assembly language programs have some specific aspects to them. For one thing, there are no variables per se; you specify operands by memory address or by a register number. The instructions you specify are low-level: they are specifically the instructions understood by the CPU. You also (directly or indirectly) need to decide where things should be placed in memory.

Assembly language varies from one CPU to another (and even a single CPU may have different assembly languages, depending on what software you are using). The assembly language for our system is very simple, with relatively few elements.

Your assembly language program is converted into machine code ("assembled") by a tool called an "assembler." For now, **you** will be the assembler, converting your assembly language code into machine code by hand.

Each line of your program will be either an instruction, a comment (a description of something, written in

human-readable form, and ignored by the system), or a directive which instructs the assembler to do something. Lines may also have a label, which is a way to remember a particular location in memory.

6.2 Language Elements

Note that the assembler is case-insensitive; unlike most programming languages, writing something in uppercase, lowercase, or mixed-case makes no difference. For example, HALT, halt, Halt and HaLt all mean the same thing.

Spacing is very flexible. A space is required, for example, between and instruction and its operand, but the spacing could be one space, or two, or ten, or a tab, etc.

Empty lines are also allowed in your program.

6.2.1 Numbers

Numbers can be written in a few different ways:

23 this will be interpreted as an octal

number (§3.2.1). 23 would be the number 2*8 + 3*1 = 19 in decimal

023 is the same number as 23, but traditionally, putting a "0" in front of a number specifies it is octal. Since octal is the default, 023 and 23 mean the same thing.

23. is the decimal number 23. The decimal point at the end (".") indicates this is a standard (base-10) number, as opposed to octal.

0x23 will be interpreted in hexadecimal, which is based 16. The 2 is in the 16's place, and the 3 in the 1's place. Thus, 0x23 is the number 2*16 + 3*1 = 35 in decimal.

6.2.2 Instructions

Instructions are simply mnemonics, followed by (depending on the instruction) 0, 1 or 2 operands. For example:

```
INC R0
ADD (R2),(R5)
LDI R6,123
JMP 52
HALT
```

and so on. An instruction will code into one or two numbers (bytes). As instructions appear in your program, their corresponding numbers will be stored in consecutive memory addresses. For example, if INC R0 is stored in address 0, then ADD (R2),(R5) will be stored in addresses 1 and 2; LDI R6,123 will be stored in addresses 3 and 4; JMP 52 in addresses 5 and 6; and HALT in address 7.

6.2.3 Comments

A comment has no effect on the actual program; it is generally a human-readable message used to explain the purpose of some section of code. Comments can contain pretty much anything: sentences, long explanations, equations, artwork, poetry, song lyrics. Whatever the programmer decides to include.

The only rules are that comments begin with a semicolon and end at the end of the line. Some comments are included to help while the program is being written; others are included

to help explain the code in the future. Others are put in just because the programmer wanted to.

A comment could start in the beginning of a line, or after one or more spaces, or at the end of a line. The assembler basically ignores the semicolon and everything after it.

6.2.4 Directives

Directives are different from instructions: they direct how the assembler does its work. The assembler recognizes three directives:

```
.org nnn
```

This specifies that the next number stored in memory should be stored at address nnn. Since by default the CPU begins execution of a program at address 0, your program will normally begin with the statement .org 0

```
.byte nnn
```

This says to store the number nnn in the next memory location. This is useful for storing data in the memory.

```
.end
```

This indicates the end of your program, and should be the last thing in your program.

6.2.5 Labels

Labels can appear throughout your program, and are a convenient way to later refer to a memory location. A label can be any word, followed by a colon (":"). Let's add a label to the sample code from §6.2.2:

```
          INC R0
          ADD (R2),(R5)
MyLabel:
          LDI R6,123
          JMP 52
          HALT
```

It was noted that the LDI instruction was stored at addresses 3 and 4; so the label "MyLabel" refers to address 3 (the start of the LDI instruction). If later there was an instruction that said "JMP MyLabel" that would be equivalent to "Jmp 3"

6.3 *Sample Assembly Language Program*

Here's an assembly language program that adds the numbers 1, 2, ..., 10 and displays the result in the output register.

```
;
; Sample program for adding the numbers 1-10
; and displaying the sum in the output register
;
  .org   0       ; Begin at memory location 0
  clr    R0      ; R0 will hold the sum of the numbers
  ldi    R1,10.  ; Load the number 10. into R1

Loop:            ; come here to add each number
  add    R1,R0   ; R0 is the running sum
  dec    R1      ; count down to next number to add
  jnz    Loop    ; if the result is not zero,
                 ; go back and add more
; Here, we've added all 10 numbers
; display the result
  out    R0      ; copy sum to the output register

Done:
  jmp    Done    ; just loop here forever
```

```
.end
```

Once you're familiar with assembly language, a program like this is fairly easy to understand (especially with all the comments included). Let's break it down into pieces and see what's going on. Then we'll look at how to convert it intio machine code.

6.4 Analysis of Sample Program

Figure 6.1 shows an annotated version of the above program.

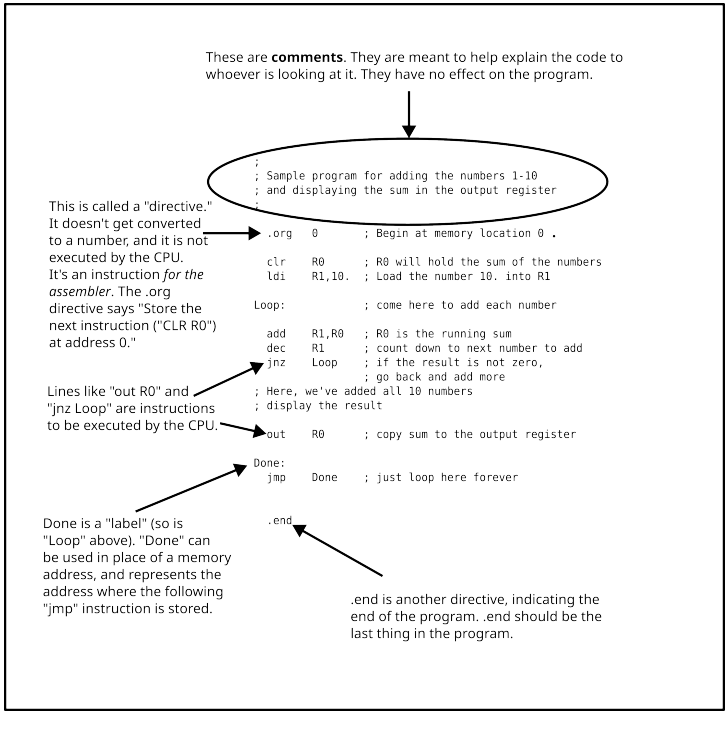
6.5 Conversion Into Machine Code

Let's now convert the sample program into machine code. This is a process of:

1. converting each instruction into its correspionding numeric encoding;
2. keeping track of where each instruction will be stored in memory; and
3. resolving labels.

Let's start with our sample program again:

Figure 6.1: Annotated Assembly Language Program



```
;
; Sample program for adding the numbers 1-10
; and displaying the sum in the output register
;
  .org   0       ; Begin at memory location 0
  clr    R0      ; R0 will hold the sum of the numbers
  ldi    R1,10.  ; Load the number 10. into R1

Loop:            ; come here to add each number
  add    R1,R0   ; R0 is the running sum
  dec    R1      ; count down to next number to add
  jnz    Loop    ; if the result is not zero,
                 ; go back and add more
; Here, we've added all 10 numbers
; display the result
  out    R0      ; copy sum to the output register

Done:
  jmp    Done    ; just loop here forever

  .end
```

Here's how each line would be assembled:

- .org 0 means the next instruction will be stored at location 0.
- "clr R0" is coded as 120, and stored at address 000.

- “ldi R1,10.” is coded as 071 012 (because 10. is 10 decimal, which is 012 octal) and stored at addresses 001 and 002.
- “Loop:” is a label, whose value is the address of the next instruction, which is 003.
- “add R1,R0” is coded as 020 010 and stored at addresses 003 and 004.
- “dec R1” is 111 and is stored at address 005.
- “jnz Loop” will be coded as 006 followed by the number corresponding to the label “Loop,” which is 3. So this is coded as 006 003, and is stored at addresses 006 and 007.
- “out R0” is coded as 170, stored at address **010** (which is the address after 007, since this is octal).
- “Done:” is another label, whose value is the address of the next instruction, which is 011.
- “jmp Done” is coded as 004 011 and stored at addresses 011 and 012.

- ".end" marks the end of the program.

If we put this all together, here's what memory will look like:

Memory Address	Contents
000	120
001	071
002	012
003	020
004	010
005	111
006	006
007	003
010	170
011	004
012	011

If you clear the memory of the board, examine address 000, and then begin to enter 1 2 0 DEPOSIT 0 7 1 DEPOSIT 0 1 2 DEPOSIT and so on, then press RESET and RUN/HALT, you should see the program run and eventually display the number 067 on the output register (067 octal is 55 in decimal, which is 1+2+3+...+9+10).

6.6 Is This Really How It's Done?

This hand-assembly is an interesting and useful exercise, but it can be a slow and tedious process. In addition to the work involved in converting instructions into numbers, if you accidentally forget an instruction and later need to go back and add it, that will potentially change the value of subsequent labels. A large, clean sheet of paper, a sharp #2 pencil and a good eraser are thus useful tools in assembling programs by hand.

Another useful tool is an assembler: a program that does this conversion for you. See Chapter 8 for information on an available assembler for this board.

7
Other Features

There are a few additional features of this board worth mentioning. These aren't meant to be secret, but it may not be obvious what they do (or even that they exist) from just looking at the board.

7.1 Power-Up Initialization

Normally, as soon as you apply power to the board, it starts up and is ready to go. However, if you hold down any button while connecting power, the system will display a message:

```
PRESS A NUMBER
```

```
  TO LOAD A
SAMPLE PROGRAM:
```

Release the button you were holding. At this point, if you press a number key (1, 2, 3 or 4), a preset program will be loaded into memory. These programs are hardwired when the board is made. It's just an easy way to start the board with something other than all 0's in memory.

7.2 Saving Programs for Future Use

A (probably more useful) feature is the ability to copy any program you've stored in memory into a non-volatile memory (meaning the program will be saved even when power is removed from the board). Of course, you can also restore such a saved program.

This is accomplished by using the EXAMINE and DEPOSIT buttons with a special memory location (771, 772, 773 or 774). For example, if you press

7 7 1 EXAMINE

the system will display:

```
HIT EXAMINE TO LOAD
```

```
PROG 1: DEPOSIT TO
SAVE/OVERWRITE PROG 1
ANY OTHER KEY TO QUIT
```

At this point you have three options:

1. If you press DEPOSIT, the current contents of memory (all 256 addresses) will be saved in non-volatile storage location 1. After the display says “SUCCESS” press any key to continue.

2. If instead you press EXAMINE, the current contents of non-volatile storage location 1 will be copied into the system’s memory. After the display says “SUCCESS” press any key to continue.

3. If you press any other key, the operation will be canceled.

You can experiment with this easily. For example, enter a program of your choice (just deposit a few values into memory to test this). The press

7 7 1 EXAMINE DEPOSIT 0 (or any other key). This will save your program in location 1. Now disconnect and reconnect power to the board, then press

7 7 1 EXAMINE EXAMINE 0 and you should find your saved program is back in memory.

Note the following:

- There are 4 available storage location: 1, 2, 3, 4 access by examining/depositing memory address 771, 772, 773, 774 respectively.
- All 256 memory locations are saved/restored, regardless of how large your actual program is.
- Each storage location should be reusable at least 100,000 times (there's a limit because, like most flash memory, writing to it uses it up a little).
- The non-volatile memory should remember stored programs for around 40 years.

7.3 Examining and Depositing: Special Locations

There are a number of other locations you can ask to examine or deposit that

are actually not memory locations. Table 7.1 shows these locations and what they represent. So for example, if you examine location 400, you'll see the contents of R0. If you then enter a number and press Deposit, that number will be saved in R0. Similarly, if you examine location 410, you'll be looking at the PC. If you deposit a number in there, that will be the address of the next instruction executed. To try this, do the following:

RESET RESET 1 0 (this clears the memory)

4 1 0 EXAMINE (this shows you the PC, which is 000)

1 2 3 DEPOSIT (this sets the PC to 123)

STEP STEP STEP (you'll see the PC incrementing to 124, 125, 126, etc. as the program executes from instruction 123).

7.4 Program Load

The "Prog Load" button brings up a display:

Location	Meaning
400	R0
401	R1
402	R2
403	R3
404	R4
405	R5
406	R6
407	R7
410	PC
411	PSW
412	SP
413-452	Stack Entries (most recent at 413, then 414, 415, etc.)
771	Saved Prog 1
772	Saved Prog 2
773	Saved Prog 3
774	Saved Prog 4

Table 7.1: Special Memory Locations

```
LOADING MEMORY
(HIT ANY KEY TO EXIT)
```

This is a feature that allows you to load memory from an external computer. If you have the upload program (available from `https://touchmetal.org`) you can take a file of memory locations and values and load it into the board through the USB port. Such a file can be made using the assembler (Chapter 8).

8
The Assembler

The way assembly language programs are described above (Chapter 6) was not merely a suggestion for how to write such programs. If you write your programs like that in a file, you can use a software tool called an "assembler" to convert the program into machine code (which can then be entered into the board manually, or automatically using the Program Load button).

8.1 Using the Assembler

You can download the source code for the assembler from `https://touchmetal.org`, or you can use the

online assembler (paste your program into the given area and press the "Assemble" button).

Figure 8.1 shows an example assembly language file. If you have the assembler ("asm" in this example) on your own computer, you can put this code into a file (say it's called "main") and then say

```
asm main
```

to assemble it. The result will be two new files: main.lst (the listing file) and main.bin (the binary output file). If you run this from the website, you can paste this code into the editor window, press the Assemble button, and the listing file ahd binary files will appear on the screen.

Figure 8.2 shows the corresponding lsiting file. Each line contains a 5-digit line number, potentially followed by a memory address, a colon, and 1-2 bytes of machine code, and then followed by the original line of source code.

8.3 shows the binary output file corresponding to this program. The format of this file is as follows:

Figure 8.1: Sample Input File For Assembler

```
;
; Sample code here; enter your own :)
;
    .org    0         ; start of memory
    ldi     R0,10.    ; This is our counter
Loop:
    out     R0        ; send to output register
    dec     R0        ; count down
    jnz     Loop      ; and continue till 0

; all done here
Spin:
    jmp     Spin      ; sit here forever

 .end
```

Figure 8.2: Sample Assembled Listing File

```
00001                   ;
00002                   ; Sample code here; enter your own :)
00003                   ;
00004                           .org    0         ; start of memory
00005 000:070 012               ldi     R0,10.    ; This is our counter
00006                   Loop:
00007 002:170                   out     R0        ; send to output register
00008 003:110                   dec     R0        ; count down
00009 004:006 002               jnz     Loop      ; and continue till 0
00010
00011                   ; all done here
00012                   Spin:
00013 006:004 006               jmp     Spin      ; sit here forever
00014
00015                           .end
```

- all numbers are octal;
- spaces are ignored;
- a colon (:) is followed by a 3 digit address, followed by 3 digit numbers to be stored in memory beginning at that address;
- consecutive numbers are stored in consecutive memory locations;
- a new address may appear at any point (as a colon followed by a 3 digit number); and
- the letter “Z” marks the end of the file.

Such a file can be loaded into the board with the “Prog Load” button (§7.4).

- ```
  :00007001217011000600200400
  6
  Z
  ```

Figure 8.3: Corresponding Binary File
  ```

9
Next Steps

This is the end of this part of the journey, but this is only the beginning. Learning assembly language and machine language helps one understand computer architecture; the design and organization of computers. Once you understand the basics of one assembly language, you can more easily learn others.

There are many other architectures to learn and study. Some systems to consider:

- PIC processors, such as a the PIC18F1220 (a relatively simple yet powerful device) and the PIC18F27K42 (you can

find one on the back of the board!). These are 8-bit processors, but with a lot of capability, including multiple interrupt levels, numerous internal devices (timers, communication systems, analog/digital converters, etc.), and lots of memory.

- Older 8-bit processors such as the Z80 (which contained several hundred instructions, and powered numerous devices such as the Tandy TRS-80); the MOS 6502 (popular in early gaming consoles and home computers); and of course the Intel 8080 (which is the predecessor of the processors that still power many PCs).
- Even older systems such as the PDP-11/70, for which you can find emulators and copies of old software from the 70s.
- More modern systems such as those based on ARM.

All these systems have similarities and differences that make it fun to explore and work with them. For learning, I

tend the underlying system, I tend to recommend older systems because they are generally simpler and more accessible. You can do assembly language programming on, say, a modern Intel X86 system, but you will be dealing with thousands of instructions, virtual memory, and very likely the constraints of working within an operating system. For advanced programming practice this may be ideal, but it tends to be far removed from the metal level. There is a time for that too.

I hope you've enjoyed learning about this processor, and will continue to explore how to program and debug it. Please check out `https://touchmetal.org` for more resources, including a downloadable version of this book.

Happy exploring!

A
Instruction Summary

Classes of Instructions

Class	Examples
No Operands	HALT
Opcode Address	JMP 000
Opcode Src, Dst	ADD R2,R3
Opcode Reg, Immediate	LDI R2,23
Opcode (Reg), Immediate	LDI (R2),23
Opcode Reg	INC R2
Opcode (Reg)	INC (R2)

Conventions for Opcode Table

s,d,r are each 3-bit numbers:

```
0=R0
1=R1
2=R2
3=R3
4=R4
5=R5
6=R6
7=R7
```

m is a 2-bit number, describing the access mode of the source and destination registers:

```
0=src,dst
1=src,(dst)
2=(src),dst
3=(src),(dst)
```

iii is an 8-bit immediate value

aaa is an 8-bit address

Opcode Table

Instruction	Coding	Flags Affected
NOP	000	-
RET	001	-
HALT	002	-
CALL address	003 aaa	-
JMP address	004 aaa	-
JZ address	005 aaa	-
JNZ address	006 aaa	-
JN address	007 aaa	-
JNN address	010 aaa	-
JC address	011 aaa	-
JNC address	012 aaa	-
JV address	013 aaa	-
JNV address	014 aaa	-
ADD src,dst	020 msd	VCNZ
SUB src,dst	021 msd	VCNZ
MUL src,dst	022 msd	VCNZ
DIV src,dst	023 msd	VCNZ
AND src,dst	024 msd	NZ
OR src,dst	025 msd	NZ
XOR src,dst	026 msd	NZ
MOV src,dst	027 msd	NZ

INTE	030	-
INTD	031	-
RTI	032	-
LDI reg,value	07r iii	-
INC reg	10r	CNZ
DEC reg	11r	CNZ
CLR reg	12r	-
NOT reg	13r	NZ
ROL reg	14r	CNZ
ROR reg	15r	CNZ
IN reg	16r	NZ
OUT reg	17r	-
LDI (reg),value	27r iii	-
INC (reg)	30r	CNZ
DEC (reg)	31r	CNZ
CLR (reg)	32r	-
NOT (reg)	33r	NZ
ROL (reg)	34r	CNZ
ROR (reg)	35r	CNZ
IN (reg)	36r	NZ
OUT (reg)	37r	-

B
Index

A
address, 18
assembler, 73
- online, 96
assembly language, 72

B
bare-metal programming, 3
binary, 20
binary numbers, 42
binary output file, 96
bit, 42
bitwise, 53
byte, 42

C
CPU, 3
call, 59
carry, 47
central processing unit, 3
comment, 76
computer, 17
computer architecture, 27
conditional, 41

D
directives, 77
display mode, 13

F
flags, 47
functions, 59

G
general purpose registers, 18

I
I bit, 68
increment, 50
initialization, 87
input register, 8, 21, 55
Instruction Set, 27
instruction set, 40
instructions, 27
interrupt, 41
Interrupt, 62
interrupt, 66

- disable, 62
interrupt enable, 48
interrupt
- enable, 62
interrupt handler, 66
interrupt
- non-maskable, 68
- priority level, 67

J
jump, 32
jumps
- conditional, 47

L
labels, 78
- resolving, 80
load, 41
loop, 31

M
machine code, 3, 72
memory, 18
- non-volatile, 88
metal, 3
mnemonics, 72
move, 41

N
NOP, 27
negative, 48
numbers, 74

O
octal, 21, 43
operand, 40
- address, 45
- immediate, 46
- register direct, 45
- register indirect, 45
output register, 9, 55
overflow, 48

P
PC, 26, 30
PSW, 47
Processor Status Word, 8, 47
program, 17
program counter, 8, 26, 30

R
registers, 18
return, 59
return address, 60
roll over, 30
rotate, 24

S
SP, 60
special locations, 90
stack, 60
stack pointer, 9, 60
subroutines, 59

U
USB-C connector, 7
unconditional, 41
upload, 93

Z
zero, 47

C
Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes