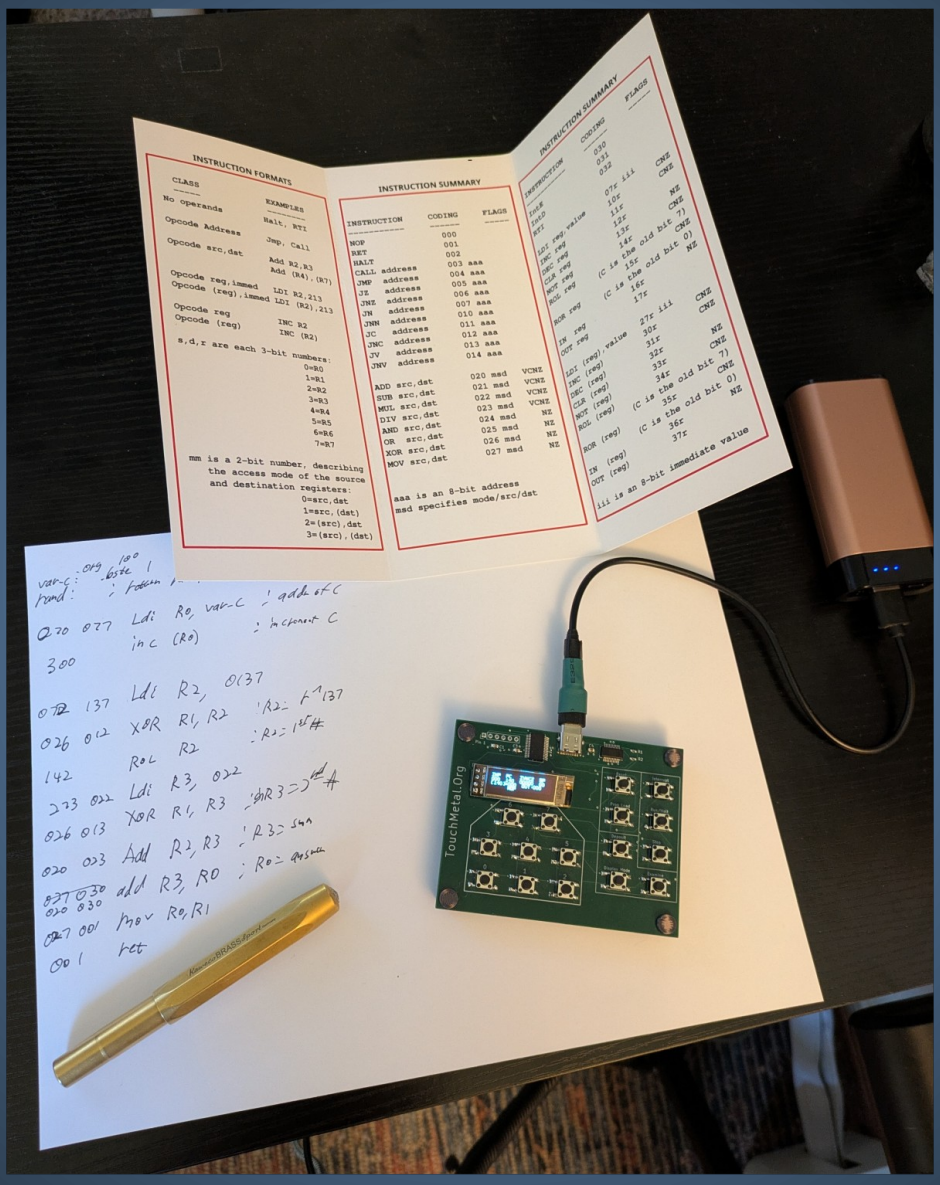
Touch Metal Practice Problems



Exercises for learning more about computer architecture, assembly language and machine code

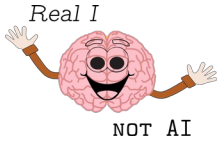
Nicholas J. Macias

Touch Metal Practice Problems

Exercises for learning more about computer architecture,
assembly language and machine code

Nicholas J. Macias

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



Foreword

This is primarily a book of exercises. There's a lot of text in here that explains different ideas and concepts, but reading that will not teach you much. Learning comes from *doing*, and that’s what this book is about. The discussions and examples are meant to prepare/help you to work on the exercises.

Ideally, this means working with a physical board (or at least the online emulator). Some of these exercises may seem tedious. It can take several button presses to, for example, examine a register or change where the program will begin executing. You may need to write down on paper various information you observe in order to work with it later. This is deliberate! It’s a way to get more familiar with various concepts and ideas, as well as with the board itself.

How to use this book

You don’t need to do every single exercise, but some of them build on prior ones, and skipping those prior ones may make the latter ones more confusing. On the other hand, there is some repetition built in, and if you feel like you’ve got the hang of something, feel free to skip ahead.

This book is intended to be a *piece* of your exploration of computer architecture and low-level programming. The biggest piece will always be your own interest, curiosity and effort.

I hope you enjoy touching metal, exploring low-level aspects of the system, and seeing what you can get a CPU to do.

Nicholas J. Macias
Vancouver, WA
June 2026

Dedication

This work is dedicated to my wonderful wife Corinna, who always encourages my projects, who believes in me more than I sometimes believe in myself, who makes it equally alright for me to go off on a crazy quest, or to come home and just drink cappuccinos and watch squirrels outside. I love you!

Table of Contents

Foreword....iii
1 Introduction....7
1.1 Structure....7
2 Board Mechanics....9
2.1 Power....9
2.2 Input Register....9
2.3 Memory....9
2.3.1 Examining Memory....10
2.3.2 Changing Memory....10
2.3.3 Clearing Memory....11
2.4 Why Only 8 Digits? Octal Numbers....11
2.4.1 Converting Between Octal and Decimal....12
2.5 Executing Code....13
2.5.1 Single Stepping....14
2.5.2 Free-Running....14
2.6 Display Mode....14
3 Basic Instructions....17
3.1 Instructions....17
3.2 General Purpose Registers....18
3.3 Examining and Changing the Value in Registers....18
3.4 The Increment Instruction....18
3.4.1 Manually Resetting the PC....19
3.5 The Decrement Instruction....20
3.5.1 Why is 377 the largest number?....21
4 Branches and Loops....23
4.1 The JMP Instruction....23
5 Assembly Language....25
5.1 Converting from Assembly to Machine Code....25
5.2 Labels....26
5.3 The Challenges of Hand-Assembling....27
5.3.1 Managing Address Changes....27
5.4 More Information....28
6 More Advanced Program Control....29
6.1 Controlling Execution: Changing the PC....29
6.2 Using the Output Register....29
6.3 The HALT Instruction....30
6.3.1 Usefulness of the HALT instruction....30
7 Conditionals....31
7.1 The Processor Status Word (PSW) – Understanding What's Happened Inside the CPU....31
7.2 The Z Bit....31
8 More Advanced Programs....33
8.1 How Do We Represent Numbers?....33
8.1.1 Binary Representation....33
8.1.2 Adding Numbers in Binary....33

8.1.3 Subtracting Numbers in Binary....34
8.1.4 Negative Numbers: Two's Compliment Notation....34
8.1.5 Some Special Values....35
8.1.6 The Sign Bit....35
8.2 Adding Numbers – the ADD Instruction....35
8.3 Immediate Addressing Mode: the LDI Instruction....36
8.4 Other Arithmetic Instructions: SUB, MUL, DIV....37
8.5 Logic Instructions: AND, OR, XOR....37
8.6 Reading the Input Register in a Program....38
8.7 The PSW Revisited....39
8.7.1 The Z (Zero) Bit Revisited....39
8.7.2 The C (Carry) Bit....39
8.7.3 The N (Negative) Bit....39
8.7.4 The V (Overflow) Bit....40
8.8 Copying One Register to Another: the MOV Instruction....40
8.9 More Single-Operand Instructions: CLR, NOT....41
8.10 Rotate Instructions: ROL, ROR....41
9 More programs....43
9.1 Display Numbers in a Loop....43
9.2 Adding Consecutive Numbers....43
9.2.1 Checking a value....44
9.2.2 Using a Counter....44
10 Subroutines, Stacks and Call/Return....45
10.1 Subroutines....46
10.2 The Stack and the Stack Pointer (SP)....47
10.3 Stack Underflow and Overflow....48
10.4 Recursion....49
11 Interrupts....51
11.1 Specific Interrupt Action....51
11.2 Returning from an Interrupt....51
12 More Programs....55
13 Indirect Addressing....57
13.1 Register Indirect Addressing....57
13.2 Code that Analyzes Itself....58
13.3 Self-Modifying Code....59
14 Final Words....61
Appendix - Opcode Table....63

1 INTRODUCTION

Hello! Welcome to Touch Metal Practice Problems – a collection of exercises for working with the Touch Metal board. This is a companion to the hardware or software Touch Metal board. If you want to purchase a physical board (currently only available in the US due to device certification requirements), you can do so through the Touch Metal Resources page (https://touchmetal.org/page-resources.html). If you prefer to use an online emulator, you can find one on the Touch Metal Emulator page (https://touchmetal.org/em/index.html).

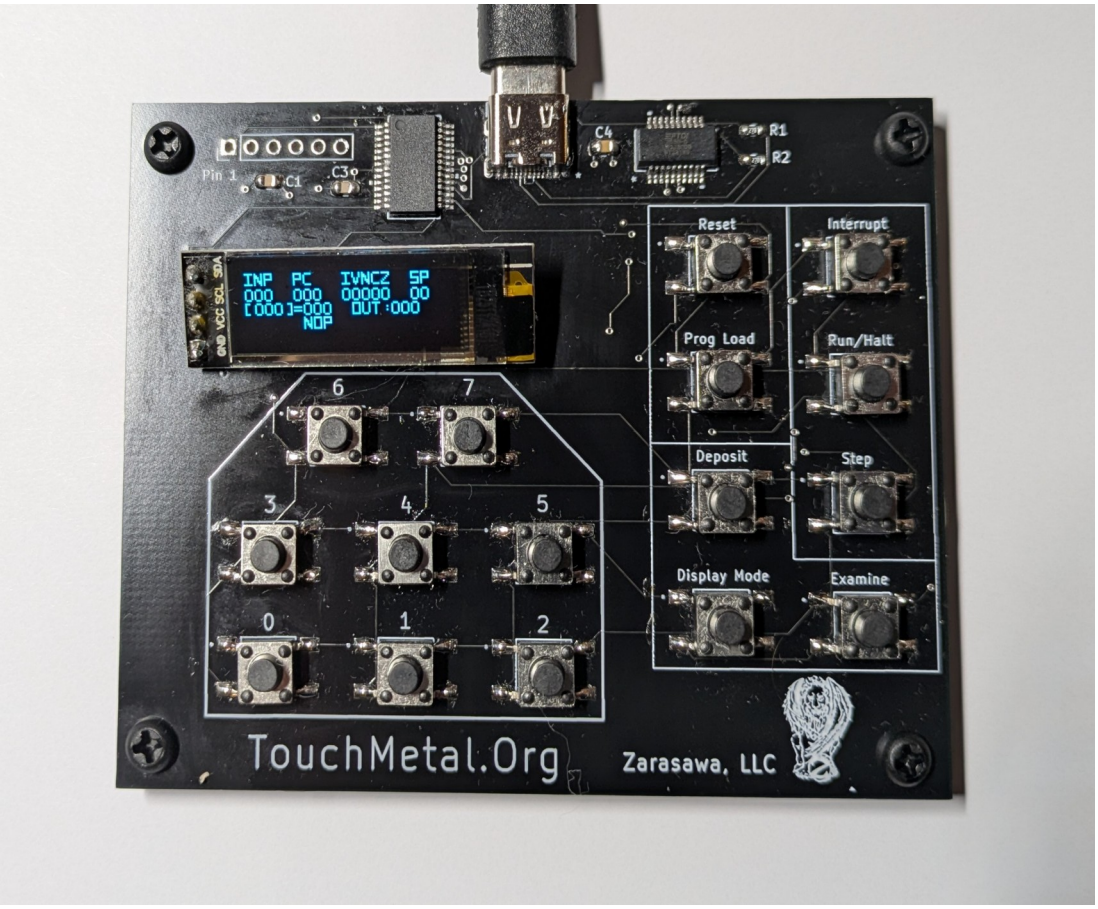


Figure 1.1 Front of the Touch Metal Board

1.1 Structure

This book covers some basic concepts in assembly language programming, machine language programming and computer architecture. Each chapter contains a mix of discussions and things to try. Things to try are marked with the symbol ✎.

While things are mostly organized linearly, working strictly from cover to cover is not necessary. You're welcome to pick and choose exercises as you like, and to hop around from chapter to chapter.

The Touch Metal Instruction Manual (https://touchmetal.org/assets/all.pdf) is really an overview of the board and the computer architecture; the book you're reading now is more for working with those concepts, testing your understanding, or getting problems to work on if you’re not feeling inspired to find your own.

Mostly, this is intended to be fun! Try things, break things, make things work; explore, experiment, get your hands dirty. It's all good, as long as you Touch Metal!

2 BOARD MECHANICS

The first goal is to boot the processor. In reality, “boot” isn’t the right term, as there is no operating system involved: you simply apply power to the board, and it’s ready to use.

2.1 Power

To power-up the board, just connect the USB-C cable (supplied with the board) to the connector at the top of the board. Connect the other end to a USB source: either a computer or a USB power supply. Within 1-2 seconds, you should see the display turn on.

If the display does not turn on, check the USB source with another device. If you’re positive the USB source is working, try a different cable. If that fixes the problem, you should move the ferrite core (the small black plastic object attached to one end of the cable) to the new cable (in order to maintain FCC compliance).

If none of this works, it’s possible there is a problem with the board. The boards are tested before shipping, but it’s possible something could be damaged during shipment. In this case, please contact orders@zarasawa.org for further assistance.

2.2 Input Register

Most input takes place through the *input register*, whose contents are displayed on the main Touch Metal display beneath the “INP” heading. When you press the buttons (“0”-“7”) you are generally entering a value into the input register.

✎ 1. Go ahead and press some of the number buttons, and observe what happens to the number displayed beneath “INP” on the display. You should see the number update to reflect your key presses.

✎ 2. Try to enter the number “123” Try “321”

✎ 3. What if you want to enter the number “5”? You’ll sometimes need to enter this as “005”. Give it a try.

2.3 Memory

A computer has a few main parts. Roughly speaking, the CPU (“Central Processing Unit”) is the part that “does stuff.” Sometimes the terms “CPU” and “computer” are used interchangeably, but we’ll mostly think of the CPU as *one piece* of a computer.

A computer also has information stored inside. That information generally falls into two main categories: *data* and *code*. Data is just raw information. It could represent numbers, letters, images, sound, or almost anything else. Whatever it *represents* though, we can think of a piece of data as a number. In the Touch Metal board, a piece of data is usually a small whole number (between 0 and 255, or perhaps between -128 and +127).

Code is a different type of information. It is something which *tells the CPU to take some action*. For example, a piece of code might instruct the CPU to add two numbers. In the Touch Metal board, a piece of code is also a small number (between 0 and 255).

Both code and data need to be stored *somewhere* inside a computer. This is the function of the *memory.* Memory is basically a place where code and data are stored. For a memory to be useful, we also need some way to retrieve information from the memory. This leads to the concept of a *memory address*. Basically, each piece of information (code or data) stored in the memory is associated with a unique address. When information is stored, we specify the address at which to store it. When we (or perhaps the CPU itself) want to retrieve a piece of information, we specify the address at which it was stored.

Here's the great thing about all this: *addresses are also just numbers*.

2.3.1 Examining Memory

We can examine memory using the push buttons on the Touch Metal board. To examine the contents of memory at a particular address, simply enter that address into the input register, and then press the Examine button.

✎ 4. Enter 1 0 0 and then press Examine. The display will show "[100]=000" This is telling you that the contents of memory location 100 is the number 000.

In fact, when you first power-up the board, *all* memory locations contain 000.

✎ 5. Examine some other memory locations, and confirm that they each contain 000.

2.3.2 Changing Memory

We can also change memory contents using the push buttons on the board. Any time you examine a memory location, you can change it by entering a value into the input register and then pressing the Deposit button.

✎ 6. Press the buttons 1 0 0 and then Examine. Now press the buttons 2 3 4 and press Deposit. You'll notice the display now shows "[101]=000" Pressing the Deposit button activates an *auto-increment* feature. After depositing into a memory location, the display advances to the *next* memory address, and shows you the contents at that address. If you enter another number and press Deposit, it will be stored at that new address.

✎ 7. Enter 1 2 3 and press Deposit. You'll see the display change to show "[102]=000" Now enter 1 1 1 and press Deposit once more. The display now says "[103]=000"

You've actually stored 3 numbers into memory:

Address	Value
100	234
101	123
102	111

Let's confirm this.

✎ 8. Enter 1 0 0 and press Examine. You should see 234 display ("[100]=234")

You could next enter 1 0 1 and press Examine; but there is also an auto-increment feature for examining memory. If you simply press Examine again, it will show you the contents of the next address in memory.

✎ 9. Press Examine and you should see the contents of memory address 101. Press Examine again to look at address 102. Press it again to see address 103.

✎ 10. Try to store the numbers 11, 22, 33, 6 and 42 in memory addresses 200-204, respectively.

✎ 11. Use the auto-repeat feature of the Examine function to look at the contents of memory addresses 200-204.

2.3.3 Clearing Memory

Sometimes it's useful to reset memory to contain 000 at every address. We can do this by disconnecting power from the board, but an easier way is to use the Reset button. If you press the Reset button once, it will reset certain aspects of the computer, but it will leave memory unchanged; but if you press Reset *twice in a row*, it will give you an option of clearing memory. Press 1 to erase memory, or 0 to cancel the request.

✎ 12. Clear the memory by using the Reset button. After doing so, examine memory addresses 200-204 and confirm that those addresses now contain 0.

2.4 Why Only 8 Digits? Octal Numbers

You've probably noticed that there are number buttons labeled 0-7, but there is no "8" or "9" button. So why are there only 8 digits?

As humans, we have a particular way of writing numbers, mostly using a series of **10** digits (0-9) with a particular order depending on the number we are thinking about. The number "12" (as in "one dozen") is written as a 1 followed by a 2. If we write "21" that's a different number.

At the lowest level, computers know nothing about our way of writing numbers. Internally, numbers are effectively stored as a collection of a set of two digits (called "binary digits," or "bits"), which we can think of as "0" and "1" (in reality, these bits are (often) coded as a pair of voltage levels, such as 0V and 5V for 0 and 1, respectively). So to work with computers at the metal level, we really want to work with only 2 digits; but humans prefer 10 digits. What's a good compromise?

It turns out using 8 digits works really well. It turns out, we can represent a set of 3 bits using one of 8 *octal* digits (0-7), as follows:

Bits	Octal Digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Table 2.1 – Mapping from Binary to Octal Digits

This also means we can represent longer strings of bits by breaking the string into groups of 3 and writing the octal symbol for each group. For example, the number 10110011 could be broken into groups as 10 110 011. If we pretend there's a 0 on the left, this becomes 010 110 011, which we then write in octal (using the above table) as 263.

This is why we only have 8 buttons (0-7) for entering numbers. Most everything we do with the Touch Metal board will be done in octal.

✎ 13. Reset the board, then begin examining memory by pressing the Examine button. What happens after examining location 007?

✎ 14. Enter the number 075 into the input register. , then press Examine four times; what happens the 4th time?

The largest number we can work with is (decimal) 255, which in octal is the number 377.

✎ 15. Enter the number 375 into the input register, then press Examine four times; what happens the 4th time?

The display changes on the 4th press, because the next location (400) isn't actually part of the computer's memory. Examining location 400 is a way to look at a different part of the system: a part called the "internal registers" (R0-R7). We'll work with these more in a later lesson.

2.4.1 Converting Between Octal and Decimal

Since we tend to think in decimal, but thinking in octal works better at the metal level, it will be useful to be able to convert between octal and decimal. There's a relatively simply way to do this.

Octal To Decimal

To convert an octal number like 123 into decimal, multiply the first digit by 64; the second by 8; and add those to the third: 1*64 + 2*8 + 3 = 83 decimal.

Decimal To Octal

To convert from decimal to octal is a bit trickier. See how many times 64 goes into your number (it will be 0, 1, 2 or 3). That's your first octal digit. Multiply that by 64 and subtract from your number. Think of the result as your new number.

Now see how many times 8 goes into your new number. That's your second octal digit. Multiply that by 8 and subtract from your new number. The result is your third octal digit.

For example, to convert decimal 83 to octal:

- 64 goes into 83 once; so "**1**" is your first octal digit;
- 83-**1***64=19. This is your new number;
- 8 goes into 19 twice; so "**2**" is your second octal digit;
- 19-**2***8=**3**; so "**3**" is your third octal digit.

Your octal number is thus 123. Alternatively, you can use the following Octal/Decimal Conversion Tables :)

	0	1	2	3	4	5	6	7
00	0	1	2	3	4	5	6	7
01	8	9	10	11	12	13	14	15
02	16	17	18	19	20	21	22	23
03	24	25	26	27	28	29	30	31
04	32	33	34	35	36	37	38	39
05	40	41	42	43	44	45	46	47
06	48	49	50	51	52	53	54	55
07	56	57	58	59	60	61	62	63
10	64	65	66	67	68	69	70	71
11	72	73	74	75	76	77	78	79
12	80	81	82	83	84	85	86	87
13	88	89	90	91	92	93	94	95
14	96	97	98	99	100	101	102	103
15	104	105	106	107	108	109	110	111
16	112	113	114	115	116	117	118	119
17	120	121	122	123	124	125	126	127

	0	1	2	3	4	5	6	7
20	128	129	130	131	132	133	134	135
21	136	137	138	139	140	141	142	143
22	144	145	146	147	148	149	150	151
23	152	153	154	155	156	157	158	159
24	160	161	162	163	164	165	166	167
25	168	169	170	171	172	173	174	175
26	176	177	178	179	180	181	182	183
27	184	185	186	187	188	189	190	191
30	192	193	194	195	196	197	198	199
31	200	201	202	203	204	205	206	207
32	208	209	210	211	212	213	214	215
33	216	217	218	219	220	221	222	223
34	224	225	226	227	228	229	230	231
35	232	233	234	235	236	237	238	239
36	240	241	242	243	244	245	246	247
37	248	249	250	251	252	253	254	255

To convert from octal to decimal, look up the first 2 octal digits along the left, and the rightmost digit along to top. The decimal value is at the intersection. For example, 213 octal is 139 decimal.

To convert from decimal to octal, find the decimal value in the table; the octal value is the 2 digits on the left, followed by the one digit in the column header. For example, 252 decimal is 374 octal.

2.5 Executing Code

Examining and modifying memory is fine, but ultimately, we want to *execute* code that is stored in memory.

2.5.1 Single Stepping

The Step button will execute a single instruction from memory. But which instruction? There's a register (a tiny memory that holds a single 8-bit number) called the "Program Counter," or the PC. The PC contains the address of the next instruction to execute. You can see the value of the PC on the display (not surprisingly, under the heading "PC").

Initially, the PC has a value of 000, meaning the instruction at memory address 000 will be executed first. After an instruction is executed, the PC is changed to the address of the instruction to execute next. In most cases, *this is the instruction immediately after the one just executed*. For simple instructions, this means the PC *increments* (increases by 1) after each instruction.

✎ 16. Reset the memory by pressing Reset twice (and confirm with a 1). The PC should show 000. Now press the Step button. Two things change:

1. the PC is now 001; and
2. the display is now showing you the contents of memory location 001: "[001]=000"

If you press Step again, you'll see a similar change.

✎ 17. Step until the PC=007, then press Step once more, and observe what happens to the PC.

2.5.2 Free-Running

Sometimes we want the CPU to repeatedly execute instructions (to "free-run"), but repeatedly pressing the Step button is not very convenient. The Run/Halt button effectively acts as if we were pressing the Step button repeatedly: it causes the instruction at the current PC to be executed, and immediately after that, the instruction at the new PC will be executed, and so on, until the Run/Halt button is pressed again, or the CPU executes an instruction that causes it to halt.

✎ 18. Press the Run/Halt button and see what happens. If you let it run about 30 seconds, the PC will reach 377, after which it rolls over to 000 (remember, our largest number is 377 octal).

✎ 19. Press the Run/Halt button again to stop the free-run mode of the CPU. You can mix the Step button with the Run/Halt button as you like. For example, you might let the CPU free-run until it gets close to a certain PC value, then press Run/Halt to halt the CPU, and then press Step to slowly continue the execution one instruction at a time.

2.6 Display Mode

There are four different display modes available on the board, with different sets of information displayed in each mode. In the default mode, a lot of information is presented on a very small display: 4 lines of information in fact. In this mode, the display looks like this:

```
INP  PC   INVCZ  SP
000  000  00000  00
[000]=000  OUT:000
      NOP
```

There are seven pieces of information displayed here (we'll eventually go over what all of the following means):

1. the current contents of the input register (“INP”);
2. the current contents of the program counter (“PC”);
3. the value of all 5 status flags (“INVCZ”);
4. the value of the stack pointer (“SP”);
5. the contents of a memory location (“[000]=000”);
6. the contents of the output register (“OUT:000”); and
7. the *assembly language* instruction (a human-readable mnemonic) corresponding to the displayed memory contents (“NOP”).

Having all this information available on a single screen is useful, but that much detail on a 1" screen can be difficult to look at (especially after a long time). The Display Mode button allows you to change what is displayed.

✎ 20. Press the Display Mode button and notice the display. This is showing the disassembly of the given memory location.

✎ 21. Press Display Mode again and notice that display. This shows the numeric contents of a memory location, as well as the contents of the output register.

✎ 22. Press Display Mode again to see the next display option, which shows the status flags and the contents of the input register. Here, instead of showing 1’s and 0’s underneath the IVNCZ heading, the value of each status bit is encoded in the heading itself: an uppercase letter means that flag is SET, and a lowercase letter means that flag is CLEAR. If the display shows “ivncz” that means each flag is clear. We’ll go over status flag details in a future lesson.

✎ 23. Press Display Mode once more to return to the normal (4-line) display.

✎ 24. Try loading the value 123 into memory location 000, examining it, and then cycling through the different display modes.

3 BASIC INSTRUCTIONS

In this chapter, we'll go over two simple but important instructions: INC and DEC. We'll practice stepping the CPU, and also working with the general purpose registers (R0-R7).

3.1 Instructions

Numbers stored in the memory can be data, or they can be instructions (code) telling the CPU to take some action. The back of the Touch Metal board includes a summary of the instruction set for the CPU.

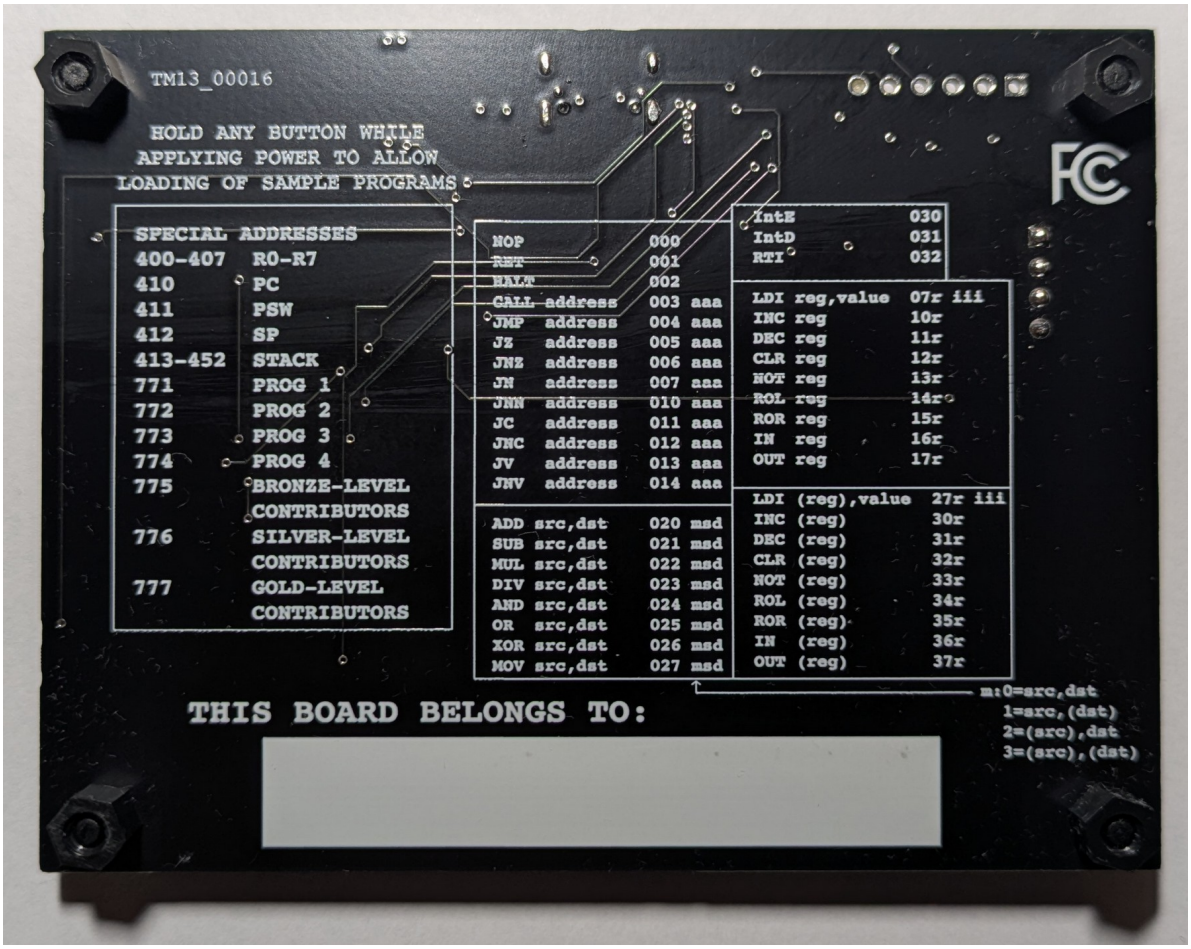


Figure 3.1 Back of the Touch Metal Board

In the middle column, you can find the instruction "NOP" (which is short for "No Operation") and see that it has an encoding as 000. The instruction 000 thus means "NOP" (though one thing does happen when a NOP is executed: after it's executed (doing nothing in itself), the PC is incremented to the next location in memory).

3.2 General Purpose Registers

Before looking at more instructions, this is a good time to introduce the General Purpose Registers. While the Touch Metal board's main memory includes 256 memory locations, there is (in a sense) a smaller memory available, containing only 8 locations. This memory is comprised of a set of 8 general purpose registers. These registers are referred to by the codes R0, R1, R2, R3, R4, R5, R6 and R7. Unlike the main memory which can hold both data and code, registers really only hold data.

Most of the instructions that manipulate data work with these registers (rather than with the main memory). In fact, we can get pretty far by using the main memory only for code, and the registers for all our data.

3.3 Examining and Changing the Value in Registers

Even though the registers aren't part of the main memory, we can examine and deposit values in the same way we do with main memory, by using the Examine and Deposit buttons. The trick is, we *pretend* that the registers are part of memory, and we access them by using (bogus) addresses 400-407:

Address	Register
400	R0
401	R1
402	R2
403	R3
404	R4
405	R5
406	R6
407	R7

✎ 25. Examine R0 by entering 400 into the input register and pressing the Examine button.

✎ 26. Press Examine a few more times to examine other registers. If you examine beyond R7, you'll begin looking at other registers, including the Program Counter (PC), the Processor Status Word (PSW) and the Stack Pointer (SP). More about those later!

✎ 27. Examine R0, and then store the number 17 into it by pressing 0 1 7 Deposit

✎ 28. Store some values into other registers, and then examine them to confirm that the values were stored.

3.4 The Increment Instruction

With our knowledge of registers, we're ready to start exploring some more interesting instructions. Look at the instruction summary in Figure 3.1. In the rightmost column, you'll see a line (near the top of the column) that says

INC reg 10r

There are some new things here:

- INC is short for "increment," which means "add one." This instruction is used to add one to something.

- "INC" is followed by the word "reg" This is an *operand*. An operand is something that an instruction will read, modify and/or write. It's the piece of data the instruction is working with.
- Instead of just showing a number (like 000) for this instruction, it shows "10r" The "r" is a placeholder for a single digit (0-7) which, in this case, indicates which register is to be incremented.

Thus this line actually describes 8 different instructions. For example,

INC R0 is coded as 100
INC R3 is coded as 103
INC R6 is coded as 106

and so on.

✎ 29. Clear the board's memory (Reset twice) and then enter the number 100 into memory locations 000, 001, 002 and 003. Now examine R0 by examining location 400. You should see R0=000. Now press Step and then Examine. What do you notice? Do this again: press Step and then Examine. You should see that the value of R0 is changing (incrementing).

You've created your first program! It's a 4-line program which, in assembly language, might look like this:

```
INC R0
INC R0
INC R0
INC R0
```

After that, the memory contains nothing but NOP instructions.

✎ 30. You can press Step/Examine 4 times, but after that, R0 stops changing. Why? (Hint: after you press Step, look at the PC, and see what instruction is stored at that location.)

3.4.1 Manually Resetting the PC

We can reset the PC (program counter) so that our program begins again from the start (the Reset button does this too, but also clears the registers).

✎ 31. Examine location 410 (this is the PC), then deposit 000 into the PC. Now re-examine location 400, then press Step and Examine. What happens? (Look at the PC after pressing Step. Can you see what's happening?

So we can execute these 4 INC instructions, and see that it repeatedly changes the value of R0. That's pretty useful!

✎ 32. Try some of the above again, but instead of using "INC R0" try something like "INC R3" (which is coded as the number 103). Notice/confirm that R3 now changes, while R0 remains unchanged.

✎ 33. Press the Reset button once, and then examine the contents of your registers. Notice that even just a single Reset clears the registers.

3.5 The Decrement Instruction

Look in Figure 3.1 for the “DEC reg” instruction. This shows a coding of “11r” Everything is the same as the INC instruction, except for that “1” in the middle, and the fact that the DEC instruction decrements (subtracts 1 from) its operand.

✎ 34. Reset the board, enter the DEC R0 instruction (110) in memory address 000, press Step, and then examine R0 (press 4 0 0 Examine). What is the value of R0?

Since “decrement” means subtract 1, and R0 is initially 0, you might have expected R0 to have the value -1. Instead, it says R0=377. What’s up with that?

Computers use numbers to store information, but how that information is *interpreted* is up to us. For various reasons (related to something called “two’s compliment notation”) we use the (octal) number 377 to represent -1. Why?

Imagine you are counting up from 000: 000, 001, 002, …, 007, 010, 011, …, 076, 077, 100, 101, … Remember, our largest digit is 7, so after 7, instead of 8, we go to 0 and bump up the number on the left (this is exactly how we’re used to counting, except we do this after 9 instead of 7).

If we keep counting, we get to 300, 301, …, 375, 376, 377, and then when we add one more, we would get to 400, but here’s the trick: our largest allowed number is 377 (see below for why this is). So we roll over, all the way back to 000. Think of the odometer on an old car (see Figure 3.2): it only counts so high, after which it rolls over to all 0’s.

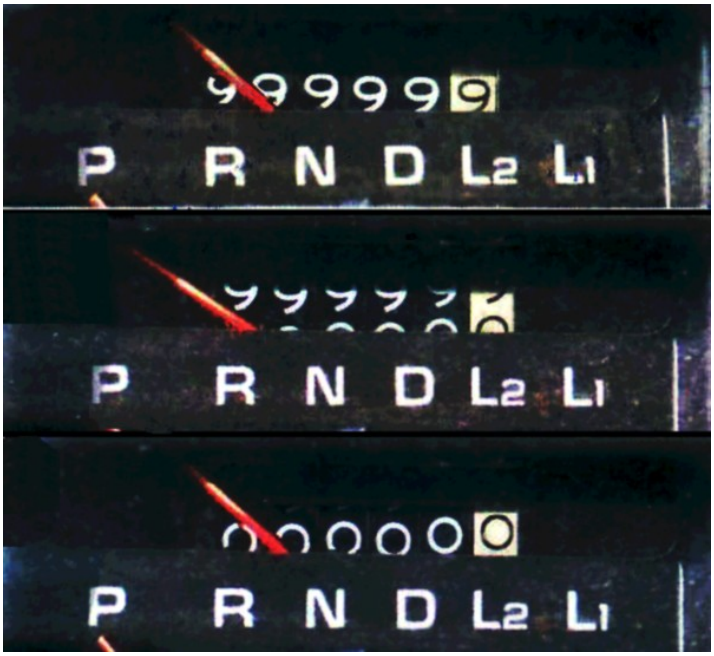


Figure 3.2 Odometer Rollover
(Source: https://commons.wikimedia.org/wiki/File:Odometer_rollover.jpg May 2026 Thank’s Hellbus!)

Okay, so if *adding* one to 377 gets you to 000, then *subtracting* one from 000 should get you 377. That's why 377 represents -1

✎ 35. Try 🙂repeating some of the experiments you did with INC but using DEC. Observe that decrementing 377 brings you to 376, then 375, and so on. If you get to 370, what does decrementing once more do?

✎ 36. Try using DEC with other registers; for example, DEC R7 (which is the instruction 117).

✎ 37. Also, after you decrement a register to a value like 377, notice the numbers beneath IVNCZ on the display. So far we've ignored these, as they've mostly been 00000; but now you may see a number like 00100 (or possible 00110). That middle "1" (beneath the "N") is the *negative flag*. The 1 tells you that you are dealing with a negative number: a further indication that decrementing from 000 has, in fact, brought us to -1 (even though it looks like 377).

✎ 38. If you really want to dig into this, set R0 to 201, then decrement it twice, looking at the IVNCZ part of the display. When R0 decrements to 177, the "N" value goes back to 0. This is because 200 is a negative number, but 177 is positive. (200 is -128, and 177 is +127). This is also related to something called *two's complement notation* (8.1.4, Negative Numbers: Two's Compliment Notation).

3.5.1 Why is 377 the largest number?

So what happened to 400? Why is 377 the largest number we can store in memory? Remember the section on Octal Numbers (**2.4 Why Only 8 Digits? Octal Numbers**). The Touch Metal board is emulating an **8-bit microprocessor**, meaning numbers are stored in 8 bits. The octal number 400 would be written in binary like this (refer to Table 2.1):

4	0	0
100	000	000

But the number 100000000 is a **9-bit** number. It is too large to write in 8 bits. Whereas the number 377 is:

3	7	7
011	111	111

which is 011111111 (and we can ignore the leftmost 0, so this is just 11111111, an 8-bit number).

So 377 is the largest number possible; if we add 1, we get 100000000, but since we only have 8 bits, that "1" on the left disappears, and we get the number 00000000; we roll over to 000.

4 BRANCHES AND LOOPS

Executing sequential instructions is fine for getting our feet wet, but to do more useful things, we will sometimes want to re-execute a set of instructions repeatedly. Later, we may want to check some condition (such as the value of the input register) and choose between two sets of instructions to execute. In other words, our code may not simply execute a linear set of instructions from start to finish, but may instead jump around from one set of instructions to another.

Doing these things involve what is called a “branch” or a “jump” (JMP). Remember, the CPU has a program counter (PC) that indicates which instruction to execute next. Normally, after an instruction finishes, the PC is incremented, so that the CPU keeps executing instructions from successive memory locations.

A JMP is a request to execute an instruction from a different location in memory. This is achieved by simply specifying new contents for the PC.

4.1 The JMP Instruction

JMP is represented by the number 004. In assembly language, we might say something like

```
JMP 123
```

Here again, the “JMP” is followed by an operand “123” This operand indicates the address of the instruction that should be executed next. In this case, no matter where the “JMP 123” instruction is stored in memory, the next instruction executed will be the one at memory address 123.

Unlike the INC instruction (coded as 10r), which stores the operand (r) in the same 8 bit number as the opcode (10), the jump instruction requires *two* numbers: one to say JMP (004) and another to indicate *to where the program should jump* (123 in this case).

✎ 39. Enter the following numbers into memory:

ADDRESS	DATA
000	000
001	000
002	000
003	004
004	000

This is a 4-instruction program. The first 3 instructions are NOP, and the 4th is “JMP 000”

✎ 40. Reset the CPU (press Reset once) and then start single-stepping this. After executing the instruction at PC=003, you should notice the PC going back to 000. You’ve made a loop!

✎ 41. Press the Run/Halt button, and you’ll see your program run on its own (remember, this is sometimes called “free-running”).

✎ 42. Change the jump instruction (stored at locations 003 and 004) to say “JMP 001” (you only need to change memory location 004 to do this). You can then examine memory address 003 to confirm what your new jump instruction is. Now step/run the program. This is a tighter (smaller) loop.

✎ 43. What happens if you change the jump instruction to say “JMP 003”? It may look like nothing is happening, but the Step button will definitely cause an instruction to be executed. This is the tightest loop possible.

✎ 44. Now change the jump to, say, “JMP 100” and step the CPU. The CPU is now executing instructions from memory location 100, 101, and so on.

5 ASSEMBLY LANGUAGE

At this point, it's useful to start looking at things in terms of *assembly language*. This is a language that is very close to the metal, but still fairly easy for humans to understand (once you're familiar with it!).

In assembly language, we write our instructions using *mnemonics*: short abbreviations (like "NOP" or "INC") in place of the numbers corresponding to them. This is how we develop our code, instead of thinking too much about octal numbers. But after we're done, *then* we add the octal numbers corresponding to each instruction. We also add in the memory address where each of these numbers (instructions) will be stored.

Our programs may include other things:

- *Directives*, which aren't instructions for the CPU, but which help us understand the assembly language. For example, we may add a directive saying "all numbers I write are in *decimal* instead of octal";
- *Labels*, which are ways to refer to a particular memory address without knowing its actual value; and, most importantly,
- *Comments*, which are human-readable remarks describing what we are doing in the code *and why we are doing it*.

Here's an example on an assembly language program that increments R0 4 times:

```
.org 0      ; This is a directive which says
            ; the next instruction should be
            ; stored at memory address 0
            ; These things after the
            ; semicolon (;) are comments
            ; and have no impact on the
            ; actual program
Inc  R0     ; Let's increment R0
Inc  R0     ; and again
Inc  R0
Inc  R0     ; and two more times
.End        ; This indicates that we haven't
            ; written anything beyond
            ; this point. It's almost like
            ; another type of comment.
```

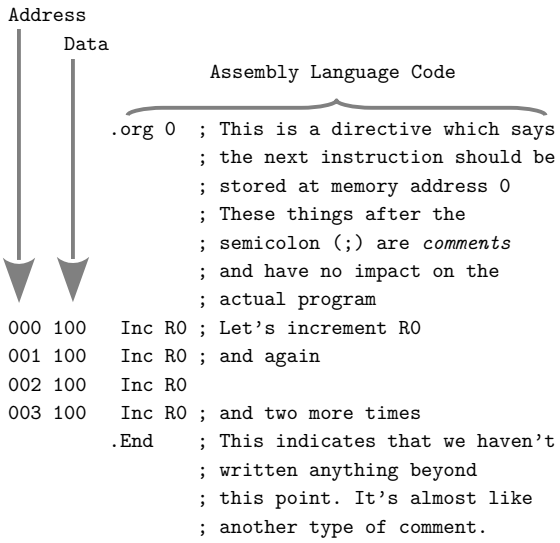
While this isn't a very complex program, it has many of the elements (instructions, directives and comments) that we'll use repeatedly in writing assembly language code.

5.1 Converting from Assembly to Machine Code

After writing the above, we would then "assemble" it, meaning we would convert it into machine code (the 1's and 0's that the CPU understands). We do this mostly by looking up the coding for each mnemonic. The Appendix (page 63) lists the coding for all possible instructions. If you have a physical Touch Metal board,

you can also just look on the back. There is also a Programmer's Card that you can download from https://touchmetal.org/assets/ProgCard.pdf (and is included as a physical card with each Touch Metal board) which summarizes the architecture (a fancy word for "details") of the CPU in more detail. Finally, you can refer to the Touch Metal Instruction Manual for the most complete description of these details.

Here's what the above program might look like once we've assembled it:




```
                .org 0  ; This is a directive which says
                        ; the next instruction should be
                        ; stored at memory address 0
                        ; These things after the
                        ; semicolon (;) are comments
                        ; and have no impact on the
                        ; actual program
000 100          Inc R0 ; Let's increment R0
001 100          Inc R0 ; and again
002 100          Inc R0
003 100          Inc R0 ; and two more times
                .End    ; This indicates that we haven't
                        ; written anything beyond
                        ; this point. It's almost like
                        ; another type of comment.
```

As you can see, we've added two columns on the left. The first column shows a memory address; the second column shows the number stored in that address.

5.2 Labels

Let's add a label and make a loop. I'll show the assembled program below, but in general you will want to start by writing the assembly language, and then assembling it into the machine code.

```
          .org    0
000 100   Inc R0
001 100   Inc R0
002 100   Inc R0     ; this should set R0 to 003
         Loop:      ; this is a label: it's equal to 003
003 110    Dec R0   ; subtract 1 from R0
004 004    Jmp Loop ; Go back and keep decrementing
005 003             ; 003 is the operand for the JMP
         .end
```

Since the instruction following the "Loop:" line is stored at location 003 in memory, the label "Loop" now means "address 003."

✎ 45. Reset the board, then enter the above machine code at the given addresses. Examine R0, then press Step and Examine to see R0's new value. You can repeatedly press Step and Examine to execute the next instruction and then re-examine R0. What does the program do? Does the code make sense?

5.3 The Challenges of Hand-Assembling

Hand assembling takes a bit of effort. For each instruction, you need to look up its opcode; if it's something like "10r" then replace the "r" with the register number; if it involves a label then figure out the value of the label; and so on. The guide on the back of the board is a quick handy reference for this. The Appendix and the Programmer's Card (see 5.1, Converting from Assembly to Machine Code) are also useful while doing these manual conversions. Unless your program is very short, you will probably want to write out your assembled program on paper, so you can refer to it and modify it as needed.

Beyond this initial level of effort, more things can happen when something changes. Suppose for example that you add another "Inc R0" in the beginning of the program on page 26. What happens to the jump statement? Since there's now *4* increment statements before the label ("Loop"), the value of Loop will now be 004. This means the "Jmp Loop" statement will be assembled as "004 004" instead of "004 003." This is one challenge of assembly language: one small change may introduce other changes upon re-assembling.

5.3.1 Managing Address Changes

One way to mitigate this is by putting space around sections of code. The "Org" directive can help with this. Remember, "Org" indicates where the next instruction should be stored in memory. So suppose we write our program like this:

```
        .org    0
000 100   Inc R0
001 100   Inc R0
002 100   Inc R0   ; this should set R0 to 003
003 004   Jmp Loop ; go to our main loop
004 100            ; 100 is the address to which we jump

        .org 100   ; our next instruction
                   ; will be stored at 100

         Loop:     ;  this is a label: it's equal to 100
100 110   Dec R0   ; subtract 1 from R0
101 004   Jmp Loop ; Go back and keep decrementing
102 100            ; 100 is the operand for the JMP
         .end
```

If you decide to add another instruction (or several!) in the beginning of this program, there's no need to change the code in our loop. Rather than just having it placed right after the "Jmp Loop" statement, we've placed it at a *fixed location* in memory.

✎ 46. If you want more practice with the front panel (the buttons and the display), enter the above machine code into memory and run the program. *What's the fewest number of key-presses you need?* Remember, after you press Deposit, the internal address advances to the next location, so you only need to enter the address in the beginning and at Loop. *Also note that pressing Deposit doesn't change the input register,* so if you enter 1 0 0 for the first instruction, you can just press Deposit twice more to store the same instruction at addresses 001 and 002.

✎ 47. Practice finding shortcuts for entering values into the input register. For example, after pressing 1 0 0, if you want to change the input register to 004, you don't really need to press 0 0 4; just pressing 4 will do the same thing! (Remember, the input register records the last 3 key presses, so if you enter 1 0 0 4, the input register is now 004).

5.4 More Information

The Touch Metal Instruction Manual has more information on assembly language, including information on software that will assemble code for you (but where’s the fun in that?).

6 MORE ADVANCED PROGRAM CONTROL

We've used a few operations (Reset, Step, Examine, etc.) to run small programs and see what the CPU is doing. There are other ways we can interact with the system though.

6.1 Controlling Execution: Changing the PC

The Step and Run/Halt buttons are fine for basic execution of a program, but sometimes we don't want to run a program beginning at location 000. For example, we may want to skip part of our program and begin at a later address. We can do this by manually manipulating the program counter.

Resetting the CPU (by pressing the Reset button) and then pressing Step will execute the instruction at address 000; pressing Step again will execute the next instruction; and so on. Let's modify that.

✎ 48. Examine address 410 (this address lets us look at and change the PC). Now deposit a number into it (such as 123) and press Step. What does the PC show?

If you entered 123, and memory location 123 contained a NOP (the default instruction), then pressing Step executed that instruction, and incremented the PC to 124.

✎ 49. Set the PC to 376 and press Step twice. What happens?

6.2 Using the Output Register

The output register offers a useful way to see what your program is doing while it runs, without having to manually examine registers and so on. To send a value to the output register, we use the instruction

OUT reg

which is coded as 17r (where, as before, r is a single octal digit (0 through 7) to represent R0 through R7). This instruction copies the contents of a register to the output register, which is visible on the display.

✎ 50. Write a loop that increments R0, then says OUT R0 (and then jumps to the start). Step through this and observe the displayed output value.

Did you get this to work? A sample program could be:

```
000 100  INC R0   ; add 1 to R0
001 170  OUT R0   ; and display the new value
002 004  JMP 000  ; Do this forever
003 000
```

✎ 51. Use the Run/Halt button to run this program

✎ 52. Add several NOP statements before the JMP and run the program. Do you notice it slowing down?

✎ 53. Based on this program, make an estimate of how many instructions execute per second. This gives you an idea of the system's *clock speed*. This is quite far from the 3 GHz (3 *billion* instruction per second) that are typical on modern processors!

6.3 The HALT Instruction

This is a good place to mention a special instruction: HALT (opcode=002). If a program is running and it executes the HALT instruction, the program stops running. Pressing Run/Halt at that point will cause the program to continue from where it stopped.

HALT will not appear to have any special effect if you are single-stepping your program with the Step button, since single-stepping effectively halts the CPU after each instruction.

✎ 54. Clear all memory, then write a program with a few NOPs followed by HALT. Reset the CPU and run the program with the Run/Halt button. What happens?

✎ 55. Press Run/Halt and see what happens. The program should continue until the PC rolls over, after which it will execute the Halt again.

✎ 56. Reset the CPU and use the Step button to execute the NOP and HALT instructions. Does the Halt do anything special?

6.3.1 Usefulness of the HALT instruction

Why do we want a HALT instruction? Suppose you write a program that's only 2 instructions long:

```
INC R0
INC R0
```

and you put that into memory and execute it. What happens after the 2nd instruction has executed? Even though your program looks like it's only 2 instructions long, the CPU just sees instructions in memory, and executes them, one after the other, forever...unless it executes a HALT instruction. So HALT can be used to say "Stop executing instructions after this point."

An alternative to halting is to execute an *infinite loop*; something like

```
Done:   JMP Done
```

While the CPU will continue executing instructions, those instructions don't do much: they won't change the values of the general purpose registers, for example.

Without a HALT or a loop at the end of your code, the CPU will simply execute whatever it finds in memory. While this may be just a series of NOP instructions, remember that after executing a NOP at address 377, the PC will reset to 000, and the CPU will effectively be re-executing your program (which may or may not be what you want).

So in general, if the last instruction of our program is not already looping, we'll want to add a HALT instruction or an infinite loop.

7 CONDITIONALS

Programming languages need three things to be general purpose:

1. a way to execute consecutive statements;
2. a way to loop; and
3. a way to make decisions.

We've already seen the first two; in this chapter, we'll explore the third.

7.1 The Processor Status Word (PSW) – Understanding What's Happened Inside the CPU

At different points in a program, the next actions the CPU takes may be dependent on the results of prior actions. For example, the CPU may increment a register, and then take different actions depending on whether or not the resulting value is 0. *This effectively allows programs to make decisions.*

Rather than explicitly ask if a certain register is 0, the CPU maintains a set of *flags* (small pieces of information) that records certain results from the execution of prior instructions. For example, there is a "Zero flag" ("Z") which records whether or not an instruction results in a 0. After an INC instruction is executed, this flag is "set" (recorded as 1 or "true") if the result of incrementing made the register's value 0; otherwise the flag is cleared (recorded as 0 or "false"). Each flag is stored as a single bit (binary digit), which is a 1 (true) or 0 (false).

There are 5 flags for this CPU:

- Z is the "zero flag," and is set to 1 if the prior[1] instruction resulted in 0;
- C is the "carry flag," and is set to 1 if the prior instruction generated a "carry" (see section 8.1.2 Adding Numbers in Binary for details on the carry and overflow flags);
- N is the "negative flag," and is set to 1 if the prior instruction resulted in a negative number;
- V is the "overflow flag," 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 8-bit number); and
- I is the "Interrupt Enable flag" (see Chapter 11).

The PSW is a special register containing these 5 flags. You can examine and modify it by using address 411 on the board (but you can't examine or manipulate it directly from a program).

7.2 The Z Bit

Let's dig deeper into the Z bit. In addition to the JMP instruction, there are a series of *conditional jump* instructions, which will only jump in certain cases.

1 Note that not every instruction affects every flag. For example, the JMP instruction does not affect *any* flags; the Z flag would be the same immediately after a JMP as it was immediately before. So here, "prior" actually means "the most recent instruction that affects that flag." See the opcode table in the Appendix (page 63) for a list of which flags are affected by which instructions.

JZ is the “Jump if Zero” instruction, and before jumping, it checks the Z bit. If the bit is set, then the jump occurs; but if the bit is clear, then the jump does **not** occur; the instruction basically acts like a NOP.

JNZ is the “Jump if Not Zero” instruction, and does the opposite of JZ: if the Z bit is set, the jump does not occur; if the Z bit is clear, the jump **does** occur.

✎ 57. Try the following program (you’ll need to assembly this into machine code first):

```
.org  0
INC   R0
OUT   R0
JNZ   000
HALT
.end
```

Before running this, load a value like 333 into R0. Now run the program. What happens?

✎ 58. Write your own program to decrement a register and stop when it reaches 0.

✎ 59. You can also jump when a result is negative (“JN”) or when it is not negative (“JNN”). Use these to decrement a number until its value is -1 (377).

8 MORE ADVANCED PROGRAMS

Now that we have conditional statements, we can begin to make more interesting programs. There are a few more instructions that will be helpful, as well as a new addressing mode: *immediate mode*. But first, it will be helpful to look at how numbers are represented in a computer.

8.1 How Do We Represent Numbers?

This section is somewhat optional. We can work with numbers – adding and subtracting them, comparing them, and so on – without necessarily needing to know how those numbers are represented in the CPU. But a basic understanding of binary representation, and of two's complement notation, will serve us well further down the road.

8.1.1 Binary Representation

A binary number is a string of 1's and 0's. On the Touch Metal board, binary numbers are *8 bits*, meaning they are comprised of exactly 8 1's and 0's, like **11001001**. We can convert between binary and decimal in the same way we converted between octal and decimal, except that each binary digit ("bit") represents a power of 2: 1, 2, 4, 8, 16, 32, 64 and 128.
So as an example, the number 11001001 represents the following number:

```
128  64  32  16  8  4  2  1 ← powers of 2
  1   1   0   0  1  0  0  1 ← bits
```

To get the decimal value of 11001001, add the powers of 2 corresponding to where the binary digit is 1:

```
128  64  32  16  8  4  2  1 ← powers of 2
  1   1   0   0  1  0  0  1 ← bits
128 +64  +0  +0 +8 +0 +0 +1 = 201
```

So 11001001 is the decimal number 201.

Incidentally, if we convert the number to octal:

11001001 = 11 001 001 = 311 octal

we can look in our octal conversion table (pg 12) and see that 311 octal=201 decimal.

8.1.2 Adding Numbers in Binary

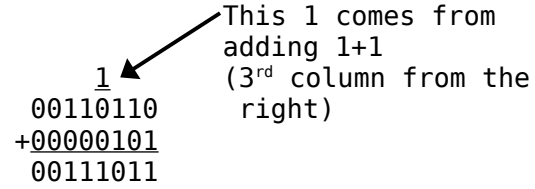
We can add binary numbers the same way we add decimal numbers. Start at the right and add, column by column:

```
 00110100
+00001001
 00111101
```

That was fairly easy: just remember 0+0=0, 1+0=1 and 0+1=1. What about 1+1? This is like adding 5+5 in decimal: we get a *carry* that needs to go into the next column. Just like 5+5=10, 1+1=10 (10 in binary is the number 2 in decimal). So consider this:

```
 00110110
+00000101
 ????????
```

Let's do what we do in decimal: we'll write any carries above the column with a line below them.



Not too bad. Sometimes a carry can have a ripple effect:

```
    111
 00000111
+00000001
 00001000
```

So addition by hand can get a bit tricky, but it's not too bad.

8.1.3 Subtracting Numbers in Binary

Subtraction in binary – as in decimal – is a bit harder than addition for most people. We deal with *borrows* instead of carries, but the idea is similar to addition.

```
      1
 00001010
-00000001
 00001001
```

Things get interesting though if we do something like 0-1:

```
11111111
 00000000
-00000001
 11111111
```

So there you have it: 0-1=11111111; a strange result indeed! But this actually makes sense, and is a key to how we can think about *negative numbers*.

8.1.4 Negative Numbers: Two's Compliment Notation

There are different conventions for representing negative numbers, but the most common is called *two's complement*. Basically, for positive numbers (and for 0), a binary number is converted to decimal as described above (8.1.1 Binary Representation). To represent a negative number, we do the following:

- convert its *positive* value to binary;
- *complement it* (change all 1's to 0's and 0's to 1's); and
- add 1.

So for example, to write the number -7:

- convert +7 to binary: 00000111
- complement it: 11111000
- add 1: 11111001

So -7 is written as 11111001

As a check on what we said above, let's convert -1 to binary using this procedure:

- convert +1 to binary: 00000001
- complement it: 11111110
- add 1: 11111111

and there we are again: -1 = 11111111

8.1.5 Some Special Values

It's instructive to look at some special values in 2's complement notation. First consider -0:

- convert +0 to binary: 00000000
- complement it: 11111111
- add 1: 00000000

So +0 and -0 look the same (as one would hope!)

How about -127:

- convert +127 to binary: 01111111
- complement it: 10000000
- add 1: 10000001

That's fine. What about -128?

- convert +128 to binary: 10000000
- complement it: 01111111
- add 1: 10000000

Uh oh! It appears that +128 and -128 look the same! But this is not a problem, because the largest number you can write in 8-bit 2's complement notation is +127. **There is no +128**.

To summarize: if we're using 8-bit 2's complement notation, the range of numbers is from -128 to +127 (and if we don't worry about negatives, the range is from 0 to +255).

8.1.6 The Sign Bit

After writing different negative numbers in 2's compliment notation, you may notice something: for a negative number, **the leftmost bit is always 1**. That leftmost bit (also called the "most significant bit" or "msb") is considered a *sign bit*, because that one bit tells you whether the number you're looking at is negative (sign bit=1) or not negative (sign bit=0). Note that 0 qualifies as "not negative."

So we can quickly tell whether a number is negative by looking at the leftmost bit. This is another reason why the largest number is 127: it's value in binary is 01111111. If we add one more, that leftmost bit would become a 1.

8.2 Adding Numbers – the ADD Instruction

The ADD instruction is one of several instructions that take *two* operands. For example, the instruction

```
ADD src,dst
```

will add the contents of the src register to the contents of the dst register, and leave the sum in the dst register. So for example:

ADD R2,R0

will add the contents of R2 and R0 and save the sum in R0.

The ADD instruction is coded as:

020 0sd

where s is the source register number and d is the destination register number. So for example,

ADD R4,R5

is coded as

020 045

✎ 60. Enter the instruction 020 045 at locations 000 and 001, then examine address 000 and confirm that the instruction shown on the display is
"ADD R4,R5"

✎ 61. Reset the CPU, then Set R4=3 and R5=4, and then execute the above instruction (just press Step). Confirm that R4 is unchanged and R5 is now 7 (3+4)

✎ 62. Try this with your own values for R4 and R5. Keep in mind that these are *octal* numbers, so, for example, if you do 3+5 you won't get 8, but rather 10 (which is how we write the decimal number 8 in octal).

✎ 63. Set R4=377 and R5=3, and execute the
ADD R4,R5 instruction. What is the result? This is another way to see that 377 is the same as -1.

8.3 Immediate Addressing Mode: the LDI Instruction

It's pretty common in programming that we'll want to load a specific value into a register. We can load a 0 with the CLR (clear) instruction, but what if we want to load a value like, say, 7?

The Load Immediate (LDI) instruction lets us do this. This is another 2-byte instruction. The first byte (a byte is just a term for an 8-bit number) is the opcode (07r, where r indicates into which register we are loading); and the 2nd byte is the value we wish to load into the register. So for example, LDI R0,137 would be coded as 070 137 and would load the *value* 137 into R0.

✎ 64. Use the LDI instruction to set R0 to 7. Do this by pressing reset, then depositing 070 into memory location 000 and 007 into memory location 001:

Reset 0 0 0 Examine 0 7 0 Deposit 0 0 7 Deposit

Check the initial value of R0 by examining location 400:

4 0 0 Examine

Now execute that LDI instruction by pressing Step, and then re-examine R0 by pressing Examine.

Did R0 get set to 7?

✎ 65. Add an OUT instruction after LDI to make it easier to see the results. Try something like

```
LDI R0,007      (070 007)
OUT R0          (170)
```
Make sure you press Reset and then press Step twice.

✎ 66. Try loading other values into other registers.

8.4 Other Arithmetic Instructions: SUB, MUL, DIV

There are other instructions that work similarly to ADD: SUB, MUL and DIV are used for subtraction, multiplication and division. Each takes two operands, and is coded similarly to ADD (see the Appendix, page 63, for the coding of each instruction).

✎ 67. Code the SUB instruction and use it to subtract one register from another. If you like, use the OUT instruction to look at the result.

✎ 68. Confirm which way SUB works: if you say

```
SUB R0,R1
```

- Does this calculate R0-R1 or R1-R0?
- Where is the result stored (R0 or R1)?

✎ 69. Code the MUL instruction and try to multiply two numbers (but keep the numbers small, say 4*5; remember, the largest number we can store is 255 decimal (or 127 is we are also allowing negative numbers).

✎ 70. Try the DIV instruction; for example, divide 18 by 6 (remember, 18 is 022 in octal!)

8.5 Logic Instructions: AND, OR, XOR

CPUs may spend a fair amount of time performing *logic operations*. These operations have names like “and,” “or” and “xor.” Their functions are based on Boolean logic (on which most CPUs operate). This means these operators only accept input values of 0 or 1, and produce an output of 0 or 1. We can describe the behavior of these functions using a “truth table”:

A	B	A and B	A or B	A xor B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

This table tells us that, for example the value of “1 and 1” is 1, while the value of “1 xor 1” is 0.

In an 8-bit CPU, we can apply these logic operators to 8-bit numbers, by applying them to each pair of bits. For example, we can AND the following two numbers:

```
11001001 ←A
01101111 ←B
01001001 ←A and B
```

Each bit in the output is just the “and” of the corresponding bit from A and from B. “Or” and “xor” work the same way.

✎ 71. Convert the values from this example (11001001 and 01101111) into octal.

✎ 72. Code an AND instruction to execute

```
AND R0,R1
```

Modify R0 and R1 (by depositing at addresses 400 and 401) so that R0 and R1 have the above values. Execute your instruction. Now look at R1; is its value 01001001 (in octal)?

✎ 73. Try this with OR and with XOR

8.6 Reading the Input Register in a Program

We can load values into registers manually by depositing into addresses 400-407, and we can load values into registers inside a program by using the LDI instruction. What if we want to load a value *while the program is running*?

This is what the IN instruction does. The format is:

```
IN reg
```

where reg is R0 or R1, etc. IN is coded as

```
16r
```

where r is the register number (0-7). When this instruction executes, the current value of the input register is loaded into the given register.

✎ 74. Use the instruction “IN R0” to read a value from the input register into R0:

Reset 0 0 0 Examine 160 Deposit 1 2 3 Step

This stores the “IN R0” instruction at address 000; puts 123 into the input register; and then executes that IN instruction.

Now examine R0 (4 0 0 Examine) and confirm that 123 was loaded into R0

✎ 75. You can repeat this by pressing Reset followed by a new input register value and then Step. Remember, Reset only clears registers (unless you press it twice in a row).

✎ 76. Follow the IN instruction with an OUT instruction to display an input value in the output register.

✎ 77. Recall in section 4.1 (The JMP Instruction) that we can execute instructions repeatedly by using a JMP to specify which instruction to execute next. Write a loop that reads input into R0, then sends R0 to the output register, and repeats this forever. Run it (by pressing the Run/Halt button) and observe its behavior. Try entering different values into the input register and observe the output register.

✎ 78. While this program is running, try entering 777 into the input register. What does the output register show? Can you imagine a reason for this?

8.7 The PSW Revisited

We looked a bit at the Processor Status Word (PSW) in Section 7.1 (The Processor Status Word (PSW) – Understanding What's Happened Inside the CPU), but mainly at the Z bit. With our understanding of binary representations, binary addition, carries, negative numbers and the sign bit; and with our LDI IN and OUT instructions; we can explore the PSW more deeply now.

8.7.1 The Z (Zero) Bit Revisited

✎ 79. Write a loop that just inputs into R0 and loops forever. Run this and look at the status flags as you enter different input values.

✎ 80. Use the Display Mode button to switch to the display that shows the PSW flags ("invcz" for example) and the value of the input register ("IN: 000" for example).

✎ 81. Set the input to 000 and observe the Z bit.

✎ 82. Set the input to 200 and observe the N bit.

When Z is displayed in uppercase, it means the Z bit is "set" (equal to 1), meaning the last instruction that updated Z resulted in a value of 0.

Similarly for N, which means a prior instruction resulted in a negative value (i.e. the sign bit was 1).

✎ 83. Look at theAppendix, or at the Touch Metal Programmer's Card (if you don't have one, you can download it from https://touchmetal.org/assets/ProgCard.pdf). In the Instruction Summary, there is a column labeled FLAGS which shows which PSW flags are affected by each instruction. Notice that for IN, the N and Z flags are updated (but C and V are not).

8.7.2 The C (Carry) Bit

The carry bit tells you when an arithmetic operation results in a "1" being generated out of the leftmost bit position. For example, if R0=377 octal and you say INC R0, what happens is this:

```
C bit→11111111  ←These are the carries
       11111111 ←Initial value of R0
       00000001 ←INC means add 1
       00000000 ←Result is 0
```

Even though the result is 0, there's a carry out of the leftmost position, which makes the C bit a 1.

✎ 84. Make a loop that inputs into R0, increments R0 and repeats forever. Run this and enter different values into the input register. You may see the Z and N bits change. Enter 377 and observe the C bit.

8.7.3 The N (Negative) Bit

The N bit is easy to describe: it tells you whether the leftmost bit is 1 (negative) or 0 (not negative).

✎ 85. Repeat the previous exercise. Enter 200 and observe the N bit. Try other negative numbers (numbers between 200 and 377).

✎ 86. Enter the number 377 and observe what happens. Why is it doing this?

✎ 87. Now enter 177 and observe the N bit. Note that both the IN and the INC instructions update the N bit.

✎ 88. Halt the program, and use the Step button to see more clearly what is happening. After executing the IN instruction, N should be 0; after executing INC, N should be 1. Try 377 again and see if it makes sense.

8.7.4 The V (Overflow) Bit

Overflow is a concept that applies when we think about numbers that can be either positive or negative, i.e. two's compliment (Section 8.1.4, Negative Numbers: Two's Compliment Notation). An overflow basically means the result of an ADD, SUB, MUL or DIV instruction was too large (or too negative) to fit in 8 bits. The rules for overflow are as follows:

ADD:
- ⚠ Adding two positives gives a negative; or
- ⚠ Adding two negatives gives a positive

SUB:
- ⚠ Positive minus Negative gives Negative; or
- ⚠ Negative minus Positive gives Positive

MUL:
- ⚠ Product is too large to fit in 8 bits

DIV:
- ⚠ Division by 0

✎ 89. Write a loop that reads the input register into R0, loads the number 1 into R1 and then adds R0 to R1. Put a HALT after the addition, and then a JMP 0 to make it loop. Reset the CPU, enter a number and press Run/Halt. The CPU will halt after the ADD, allowing you to study all the flags.

✎ 90. Run this loop with an input of 377. You should see the C and Z bits set, but not V. Why not? Remember, 377 is actually -1 (a negative number) so 377+1 is doing a negative plus a positive: no overflow is possible.

✎ 91. Try 176. No flags should be set (176+1=177). Now try 177. What happens? What is 177+1? The N flag is a clue :)

8.8 Copying One Register to Another: the MOV Instruction

A very common operation is copying one register to another. The MOV instruction does this. Note that even though it's called "move," it doesn't actually remove the value from the old register; it *copies* it.[2]

✎ 92. Load a value into R0 (use LDI or IN), then MOV it to R7. Execute this and then examine the values of R0 and R7.

2 What would it even mean to "remove" something from a register? A register always has *something* in it.

Notice that while LDI does not affect any PSW flags, the MOV instruction updates N and Z. So whatever value you load with LDI can be *tested* with the MOV instruction. Note that it is perfectly fine to copy a register to itself, which is a useful way to see if a register is zero or negative.

✎ 93. Make a loop that uses LDI to load a value into R0, and then copies R0 to itself with the instruction "MOV R0,R0" Try this with

```
LDI R0,200  and then with

LDI R0,000
```

Single-step your code to observe that the N and Z bits are unchanged by the LDI but are updated after the MOV.

8.9 More Single-Operand Instructions: CLR, NOT

The Clear (CLR) instruction simply loads a 0 into a register. For example:

```
CLR R3
```

would set R3 to 0. Since we know the result will be 0, the Z bit is **not** changed by the CLR instruction.

There's also the "not" instruction (NOT), which just swaps 1's and 0's. This is also called "complementing." For example: if R5=11000001, then executing NOT R5 would make R5=00111110

✎ 94. Make a loop that inputs into a register, complements it, and sends it to the output register. Run this and observe the results for different input values. Converting to/from binary (Section 2.4.1, Page 12) may help you make more sense out of what you see.

8.10 Rotate Instructions: ROL, ROR

There are a pair of instructions for shifting the bits of a number to the left or the right. These instructions are called "Rotate Left" (ROL) and "Rotate Right" (ROR). They are called "rotate" rather than "shift" because when you shift (say to the left), the leftmost bit would normally fall off the end; but with a rotate, instead of being lost, that bit is copied back to the rightmost position. So for example, if R0 contains the value

```
abcdefgh
```

(where a-h are 1's and 0's), ROL would change this to

```
bcdefgha
```

Additionally, the bit which is rotated (bit a in this case) is also copied to the PSW's carry (C) bit. So if you ROL a number like 01000110, the result would be 10001100 and C would be 0. If you ROL that number again, the result would be 00011001 and C would be 1. C basically shows you the bit that would have been lost if this was a simple shift instead of a rotate.

✎ 95. Make a loop that repeatedly executes

```
ROL R0
```

followed by

```
OUT R0
```

Load an initial value into R0 by depositing into location 400 and then step through the loop and observe the output values.

✎ 96. Try this with an initial value of R0=1. What do you notice about the values of R0 as it rotates left?

The “Rotate Right” (ROR) instruction is the same basic function as ROL, except it rotates to the right:

abcdefgh would become
habcdefg

In this case, the carry bit is set from the value rotating off the right side: bit h in this example.

✎ 97. Write a loop that rotates a register repeatedly to the right, and observe the behavior.

✎ 98. Try this with an initial value of 0200. What do you notice about the values after each rotate to the right?

9 MORE PROGRAMS

We have enough in our coding bag of tricks now to write some full programs. These may not be the most exciting things in themselves, but they will show how our instructions, flags, conditionals and such can work together to perform higher-level functions.

9.1 Display Numbers in a Loop

Let's make a loop that repeatedly displays consecutive numbers. Here's a bit of assembly language to do this:

```
      Loop:
            Inc   R0    ; next number
            Out   R0    ; display it
            Jmp   Loop  ; and repeat forever
```

✎ 99. Convert this to machine code, enter it into the board and execute it. You can step it to watch how it works, then free-run it (press the Run/Halt button) to watch it run at full speed.

✎ 100. Change this to repeatedly *decrement* R0 by 1

We can also count by something other than 1, but we'll need to do some initialization:

```
      Ldi   R1,3  ; Initialize our
                  ; increment value
Loop:
      Add   R1,R0 ; add 3 to R0
      Out   R0    ; display it
      Jmp   Loop  ; and repeat
```

✎ 101. Run this and see what happens.

✎ 102. Change this to count up by 4.

✎ 103. Now change it to count *down* by 3

9.2 Adding Consecutive Numbers

Let's write a program that adds consecutive numbers:

```
      Clr   R0    ; R0 is our running sum
      Ldi   R1,12 ; 12 is the number 10.
                  ; this is how many
                  ; numbers we want to add
Loop:
      Add   R1,R0 ; Add to our running sum
      Dec   R1    ; Count down
      Jne   Loop  ; If R1 is not 0, repeat

; Here, we are done
      Out   R0    ; Display the answer
      Halt
```

✎ 104. Enter this program and run it

✎ 105. Change this to display the running sum as it is calculated, instead of just at the end.

Our final answer is correct, but we're really adding 10+9+8+… instead of 1+2+3+… Let's change the order of the addition. Here are two ways to do this.

9.2.1 Checking a value

We can keep adding R1 to R0 and increasing R1, until R1 is 10.

```
      Clr   R0    ; Running Sum
      Ldi   R1,1  ; Number to add to sum
Loop:
      Add   R1,R0 ; Do the addition
      Ldi   R4,12 ; This is the last number
                  ; we want to add (10.)
      Sub   R1,R4 ; Calculate R1-12
      Jz    Done  ; If R1-12=0 then
                  ; we're done
      Inc   R1    ; else go to next number
      Jmp   Loop  ; and repeat

Done: Halt        ; All finished
```

✎ 106. Add some OUT instructions to this code and try it.

9.2.2 Using a Counter

Another approach is to use a separate register to count how many times we loop. Here's some code using that idea:

```
      Clr   R0    ; Running Sum
      Ldi   R1,1  ; Number to add to sum
      Ldi   R4,12 ; This is our counter
Loop:
      Add   R1,R0 ; Do the addition
      Inc   R1    ; Move R1 to next number
; Now let's see if we're done
      Dec   R4    ; Count down from 10.
      Jnz   Loop  ; If not at 0, do more

      Halt        ; else we're done
```

✎ 107. Try this code, adding some OUT statements to see what is happening

10 SUBROUTINES, STACKS AND CALL/RETURN

We already know the JMP instruction can be used to change what code the CPU is executing. This is very useful for making loops (to repeat the same actions over and over again), and for making decisions (such as "if a register is 0, do one thing, otherwise do something else").

Another reason for switching to a different set of code is for *modularization* of our code: breaking it up into smaller pieces, which are then used together to perform some task.

For example, here's a simple program that displays consecutive numbers in the output register:

```
Loop:
      Inc   R0
      Out   R0
      Jmp   Loop
```

You've already looked at this program, and if you run it, the numbers count up, not very quickly, but a few per second. Suppose we wanted to slow that down.

Since everything takes time in the CPU, we can create a *delay* routine by just doing stuff for a while. Here's a simple set of code that makes about a 1 second delay:

```
Delay:
      Nop
      Nop
      Nop
      Nop
      Nop
      Nop
```

Exciting, huh? Now suppose we wanted to add a 1 second delay after we display our incremented number. We could add that code after the OUT instruction, but that is a bit cumbersome:

```
Loop:
      Inc   R0
      Out   R0
      Nop
      Nop
      Nop
      Nop
      Nop
      Nop
      Jmp   Loop
```

Having all those NOPs in the middle of our loop is potentially confusing (especially if our program was more complex to begin with).

Instead, we could put that Delay code somewhere else and jump to it:

```
Loop:
      Inc   R0
      Out   R0
      Jmp   Delay
      Jmp   Loop
```

Okay, this is better; it's clean, we only added one statement, and it's pretty easy to guess that "Jmp Delay" is there to introduce a delay. But there is a problem: after jumping to Delay, how do we get back to the top of the loop?

We could add a "Jmp Loop" at the end of the Delay code, and that would fix this problem. But what if we wanted to use the Delay code from somewhere else?

10.1 Subroutines

This is the idea of a "subroutine" (sometimes also called a "function"). A subroutine is a set of code that can be run from different places. When we ask this code to run, we say we are "calling the subroutine." The CALL instruction is used to make such a call. The format for a call is

CALL *address*

where *address* is the memory location of the code we wish to start executing.

What makes a subroutine so useful is that after it runs, *it can return to the place from where it was called*."

So, for example, we might have code like this:

```
Loop:
      Inc   R0
      Add   R0,R1 ; just random calculations
      Out   R1    ; display R1
      Call  Delay ; wait 1 second
      Out   R0    ; display R0
      Call  Delay ; wait another second
      Sub   R0,R1
      Out   R1    ; Display a new value
      Call  Delay ; and wait another second
      Jmp   Loop
```

Each time we say "Call Delay," the code at the label "Delay" is executed.

But wait: we added a "Jmp Loop" at the end of Delay. That's not going to work! What we want is for the code at Delay to eventually return to the place from which it was called. The "Return" (RET) instruction does exactly this. RET says "*go back to the instruction* after *the Call instruction that brought us here.*"

✎ 108. Code the above Delay function; remember to put a RET at the end. Write it starting somewhere far away in memory, say somewhere like address 100

✎ 109. Make a loop that increments R0, copies R0 to the output register, calls your Delay subroutine, and loops back to the start. Can you observe the delay between output changes?

✎ 110. Try increasing the delay. You can probably get the delay up to 2 seconds. Can you imagine what you would do for a 60 second delay?

A 60 second delay would involve a lot of NOPs! Let's make a more-useful subroutine that does the following:

- call Delay to cause a 1-second delay;
- decrement a fixed register like R7;

- if R7 does not reach 0, go back and repeat
- otherwise (if R7 is 0) return (use the RET instruction).

This function would let us put a value into R7, call it, and it should return "R7" seconds later (in other words, if we put a 60. into R7, it should return after 60 seconds).

✎ 111. Try coding this function. Call it with test values of 5, 10 and 15 (remember, 10 is 012 in octal and 15 is 017 in octal).

✎ 112. The timing is probably not very precise. Can you change the number of NOPs in your Delay code to make it more precise?

10.2 The Stack and the Stack Pointer (SP)

How does the call/return mechanism work? Calling is pretty simple: it's basically a jump. The trick is knowing where to go back to when the RET instruction is executed. This is accomplished by saving the *return address* (the address to which the RET instruction should jump). But where to save this address?

We could simply save it in a special register. That would work fine, except for one thing: **what if the subroutine wants to call another subroutine?** We may need to remember more than one return address.

So instead of a single register for saving the return address, we have a small memory, called the "stack." The stack can store up to 32 return addresses. This means, for example, that a function A could call function B which might call function C and so on, 32 times (this is unlikely to happen with the relatively small memory we have for writing programs, but such levels of nested calls are easy to find on larger systems).

Well, if we have a memory, we need to have addresses, and a way to know at which address we should store the next return address. This is the function of another special register called the "stack pointer" (SP). The stack pointer is initialized to 0, and indicates at what address in the stack the next return address will be stored. When a CALL instruction is executed, the address of the next (sequential) instruction is stored in the stack, at the address given by SP, and then SP is incremented.

Consider the following situation:

```
.org  0
call  100
halt
.org  100
nop
ret
```

If we were to assemble this into machine code, it would look like this:

```
Address     Contents
000         003
001         100   ; this is the CALL 100
002         002   ; HALT instructions
100         000   ; NOP
101         001   ; RET
```

So when the call instruction at address 000 is executed, here's the sequence of events that occurs:

1. the address of the next instruction (002) is saved at address 0 in the stack memory;
2. the stack pointer is incremented to 1; and

3. the value 100 is loaded into the program counter (PC).

That's it! Now the CPU is executing the instruction at address 100, while the return address (002) is safely saved in the stack.

The NOP is executed, the PC is incremented to 101, and now the RET is executed. Here's what happens now:

1. the stack pointer is *decremented* to 0; and
2. the contents of the stack memory at address 0 (which contains 002) is loaded into the PC.

And there you have it; the return loads 002 into the PC and the HALT instruction is executed.

✎ 113. Try coding this example, and single-stepping it. Observe the PC and the SP (the SP is displayed on the display).

✎ 114. Try calling a set of code that calls a second set that calls a third set. Step through this, and observe the stack pointer as it increases through the calls, and then decreases through the returns.

✎ 115. We can examine the stack by using the Examine button. If you examine location 413, you'll see its contents displayed as "S[00]=" followed by its contents. After the 3rd call, examine the stack's contents. Note that addresses 413-452 correspond to stack addresses 000-037 (0-31 decimal).

You can also examine (and change) the SP by accessing address 412.

✎ 116. Examine the stack's contents *after the returns are executed*. What do you notice?

Even though we've already used the return addresses, they are still stored in the stack's memory. This is actually a real-world issue. On some systems, previous contents of the stack remain intact until something else uses those stack addresses. Often the stack is used for saving not just return addresses, but also *local variables* (variables used in a subroutine or function, but whose values don't need to be kept indefinitely). It's sometimes possible to exploit this to read the values of local variables in other programs, which can be a security issue.

10.3 Stack Underflow and Overflow

✎ 117. What happens if you execute a RET without first executing a CALL? Reset the CPU and try it.

Normally this would generate an error called a "stack underflow." This is considered an unexpected/abnormal condition, and might cause a program to terminate. On some systems, it might cause the CPU to reset itself. In some contexts, it might cause an entire system to crash and reboot.

On the Touch Metal board, an underflow is quietly ignored: a RET without a matching CALL is treated like a NOP.

✎ 118. Write an instruction that repeatedly calls itself:

```
Here:  Call Here
```

Step through this and observe what happens to the PC and the SP.

✎ 119. What happens if you do this until the SP reaches 40?

This is called a “stack overflow,” and is also considered an error condition. On our system, the only effect is that the return address is not saved in the stack memory (and the SP is not changed). In this case, the CALL basically acts like a JMP.

10.4 Recursion

It’s sometimes actually very useful for a set of code to *call itself*. This is something called “recursion,” and it’s central to a lot of more advanced programming, including working with structures called “binary search trees” (a topic for a different book!).

Here’s a sample program that doesn’t do anything terribly useful, but shows a program calling itself a few times and then returning (instead of calling itself endlessly until the stack overflows).

```
        .org    0
Top:
        ldi     R0,10   ; count # of calls
        ldi     R1,10   ; and # of returns

; call ourself 10 times
Loop1:
        dec     R0
        jz      doneCalling
        Call    Loop1

; stack should have 10 things on it
; start returning
doneCalling:
        dec     R1
        jz      Top     ; we did 10 returns!
        ret             ; else return
                        ; (to doneCalling)

        .end
```

✎ 120. Try this program Congratulations! You’ve written a recursive program :)

11 INTERRUPTS

Other than the input register, we don't seem to have a lot of control over what a program does. We can use the Run/Halt button to start and stop it, but how do we more gracefully affect what the program is doing?

One way is with an "interrupt." An interrupt is, in effect, like a call that is initiated by something other than executing a Call instruction. On the Touch Metal board, there is a button labeled "Interrupt." Normally, pressing this seems to have no effect: by default, interrupts are not allowed (it's like a default "Do Not Disturb" setting).

We can enable interrupts by using the IntE instruction. This has the effect of setting the "I" (interrupt) bit in the PSW. Whenever that bit is set, interrupts are enabled. We can later *disable* interrupts with the IntD instruction (which clears the I bit in the PSW).

Other systems may have several (or hundreds!) of possible interrupts, which can be enabled or disabled selectively. Some systems have *priority levels* associated with interrupts, and might allow, for example, only interrupts above a certain priority level. Some systems allow *software interrupts* (sometimes called "exceptions").

On the Touch Metal board, we have only a single interrupt, which is requested by pressing the Interrupt button.

11.1 Specific Interrupt Action

When an interrupt is requested (and if the I bit is *set* in the PSW), the CPU takes the following actions:

1. save the current PSW on the stack;
2. clear the I bit in the PSW; and
3. call address 0376

Note that step 2 means that interrupts are disabled, and step 3 means the CPU begins executing instructions at address 0376. The address of the instruction *that was about to be executed* is also saved on the stack (because of the call). Typically, you'd want address 0376 to contain a JMP instruction to somewhere in memory where the *interrupt handler* (the code that will respond to the interrupt request) is stored.

11.2 Returning from an Interrupt

At the end of the interrupt handler, the system should return things to how they were before the interrupt occurred. This is done with the RTI (return from interrupt) instruction, which does the following:

1. read the return address from the stack (and decrement SP);
2. read the saved PSW from the stack (and decrement the SP again); and
3. set the PC to the return address that was found in step 1.

So the RTI basically undoes the effects of the interrupt (except for whatever changes the interrupt handler has deliberately made). Note that the PSW is restored to the exact value it had right before the interrupt. Thus, for example, if we are executing the following code:

```
DEC R0          <-- interrupt occurs
JNZ Somewhere       between these instructions
```

and an interrupt occurs between these instructions, the Z bit from the DEC instruction will be restored when the RTI returns to the JNZ instruction (so our conditional branch will work as expected). This also means that interrupts are re-enabled by the RTI instruction (since the I flag is part of the PSW).

✎ 121. Write some code that enables interrupts and then loops forever. Put a HALT instruction at address 0376. Run this program; it should look like nothing much is happening.

✎ 122. While it is running, press the Interrupt button. What happens? Observe the PC and the SP

✎ 123. Change the HALT instruction to an RTI instruction. Run this and press Interrupt. What happens? You may need to press it a few times to catch everything that happens.

We can use interrupts to alert our code to take some action. For example, consider this code:

```
      .org  0
      IntE        ; enable interrupts
Loop: Jmp   Loop  ; spin here forever

IntHandler:       ; our interrupt handler
      Inc   R0
      Out   R0    ; display next number
      Rti         ; go back where we were

      .org  376
      Jmp   IntHandler

      .end
```

✎ 124. Assemble this code (convert it to machine code), load it and execute it. It will appear to do nothing, but press the Interrupt button (while it is running) and see what happens.

✎ 125. Halt the program, step it a few times, and then press the Interrupt button. What happens?

✎ 126. Step the program some more and observe the behavior.

We can use interrupts however we like. For example, we could use an interrupt to tell our program that we have entered a number in the input register and it should do something with that number.

✎ 127. Write a program that sets a running sum to 0. Each time the Interrupt button is pressed, read the input register, add it to the sum, and display the result in the output register.

✎ 128. Write a program that displays the number 001 in the output register, and then loops forever. When the Interrupt button is pressed, display the number 377 instead. When it is pressed again, go back to displaying 001, and so on.

✎ 129. Write a program that continually increments a register and displays it in the output register. When the Interrupt button is pressed, it should switch to decrementing (counting down instead of up). When it is pressed again, it should switch back to counting up.

12 MORE PROGRAMS

Can you think of other programs to write? Here are some more ideas.

✎ 130. Write a program that reads the input register when the Interrupt button is pressed, and saves the value in R0. The next time the Interrupt button is pressed, read the input register and save it in R1. Now add R0 and R1 and display the sum in the output register. Go back to the top and repeat this forever (now you've got the world's most-difficult-to-use adding machine!).

✎ 131. Write a program that counts the number of times a number button (0-7) is pressed. You can do this (in most cases) by reading the initial value of the input register, and then repeatedly reading it and comparing to the initial value. If it changes, bump up the count and display it (and remember the new input register value, so you compare against that next time). Now start pressing buttons and see if the count increases.

✎ 132. There are some cases where this will not count a new button press. What are those cases?

✎ 133. Read the input register, and break its value into two numbers. For example, if the octal value of the input register is 023, break this into the numbers 2 and 3. Add these and display the sum in the output register. This is a simply one-digit calculator! For example, if you input 011 the output should be 002. This will require using the rotate instructions, as well as using logic operations like AND to pay attention to some bits while ignoring others.

13 INDIRECT ADDRESSING

In most of what we've discussed so far, data is stored in the general purpose registers R0-R7. While this is very convenient, it's also somewhat limiting: 8 registers is not a lot!

Most CPUs let you store data in the system's memory. We can do this on the Touch Metal system, but it takes two steps to do so.

13.1 Register Indirect Addressing

Consider the increment instruction

    INC R0

The behavior is straightforward: if R0 initially contains 020, this instruction changes R0 to 021. This type of instruction is called a "register direct" instruction.

There is a second type of instruction, called a "register indirect" instruction. This would be written as

    INC (R0)

The parenthesis mean "don't increment R0; increment **what is at the memory address** specified by R0." In the above example, if R0 contains 020, then

    INC (R0)

does not change R0; **it increments memory location 020**.

This is quite different! For the first time, *our program can change values that are stored in memory.* This is a powerful capability. Moreover, while we might imagine an instruction (this doesn't exist in our CPU) like "INC 020" to increment the contents of memory location 020, "INC (R0)" lets the program change *which location* is incremented, simply by changing the value of R0.

✎ 134. Write a program that sets R0 to 20 and does INC (R0). Observe the value of R0 after this; then observe the value of memory location 20.

Most instructions work with indirect addressing: ADD, SUB, MUL, DIV, AND, OR, XOR and MOV (which take two operands) can use register direct or register indirect addressing on either, both or neither operand. For example:

    ADD (R0),R2

would look at R0; treat its contents as a memory address; read from that address in memory; add the contents of R2; and save the result back in R2.

LDI, INC, DEC, CLR, NOT, ROL, ROR, IN and OUT also work with register indirect addressing.

Here's a program that reads two numbers from memory, adds them and displays their sum. The two numbers are stored in memory addresses 100 and 101.

```
.org  0
Ldi   R4,100    ; points to first #
Mov  (R4),R0    ; R0 is the first #
```

```
Inc  R4         ; R4 is 101, which is
                ; the address of the
                ; second number
Add  (R4),R0    ; Add number from memory
                ; address 101 to R0
Out  R0         ; display result
Halt            ; All done
```

✎ 135. Assemble this code (convert it to machine code) and enter it into the memory. Now store your two favorite numbers in memory at addresses 100 and 101. Run the code and observe the output register. Is it the sum of your numbers? As always, remember these are *octal* numbers.

✎ 136. Write a program that reads numbers from memory, starting at address 100, until it reads the number 0. It should add all the numbers and display the result in the output register (hint: add each number to a running sum *as you read it*).

13.2 Code that Analyzes Itself

Register indirect addressing lets us read/write data from/to different locations in memory. But it also lets us read *code* from memory.

✎ 137. Write a program that reads the value stored in each memory address from 0 to 20; adds these values; displays the result and then halts. What does it show at the end? Why? What numbers did it add?

Yes, this code added its own opcodes and operands! This is a quirky thing to consider, but it's actually very useful.

✎ 138. **Reset all memory** (press Reset twice). Change the code you just wrote so the program sits in memory starting at address 200 (this means your jump instructions will need to be changed). Now set the PC to 200 (examine location 410 and deposit 200 into it) and run the code. What value does it display?

It should display 000, since the first 20 locations in memory all contain 000 (because you recently reset all memory).

✎ 139. Now enter a small program loop beginning at address 0: some code that repeatedly increments R0 and displays the result in the output register. Run your code *(at location 200)*. What does it display now?

✎ 140. Change your small program loop to repeatedly *decrement* R0 instead of increment it. Run your code at location 200. What does it display now?

This is an example of *fingerprinting* code. The number being displayed is a reflection of the code you entered in the beginning of memory. If you change the code, you (most likely) change the fingerprint. This specific type of fingerprint is called a "checksum," because we are adding ("summing") the program bytes (and we could then check them against an expected sum, to confirm that the program bytes are (most likely) correct). When you change the INC to a DEC, the fingerprint changes. This is a quick way to know that the program has changed.

This idea of fingerprinting code is very relevant today; for example, when you download code and see an *MD5 hash.* The algorithm for calculating an MD5 hash is more complex than just adding bytes, but the idea is

the same: producing a (very long!) number from the bytes of a file, in such a way that if any of the bytes change, the resulting number will also likely change. This is a way to (in most cases) confirm that a program has not been changed (either accidentally or maliciously) from its original code.

13.3 Self-Modifying Code

If we can read code using register indirect addressing, can we also *write* code? The answer is yes!

✎ 141. Write a loop which initializes R0 to 10, then starts a loop where it repeatedly increments R0, executes

    LDI (R0),17

and repeats forever. Step through the loop a few times, then examine memory starting from address 010. What do you observe?

✎ 142. You can let this free-run for a while and re-examine memory. How many address got changed?

✎ 143. If you let this run for several minutes, you should notice an eventual change in behavior what happened? Why?

Our code changed itself! This is called “self-modifying code,” and is a key feature of the “stored-program” computer. Most machines designed since the 1940s have been based on this stored-program model. The idea is actually simple: remember, we said that the memory stores both data and instructions, and there’s really no way to tell one from the other. If we read or write memory using register indirect operations, then we are working with data. If the PC is pointing at a memory address, then the contents of that address are treated like code.

Consider the following program:

```
      .org  0
Loop:
      Inc   R0
      Out   R0      ; the usual inc/display
      JZ    Flip    ; If R0=0 goto Flip code
      Ldi   R1,10
      Sub   R0,R1   ; compute R0-10
      JZ    Flip    ; if R0-10=0, goto Flip
                    ; (if R0-10=0 then R0=10)
      Jmp   Loop    ; else just continue

Flip: ; let’s have some fun...
      Ldi   R2,010  ; this is a “bitmask”
      Ldi   R3,0    ; points to our
                    ; Inc instruction
      Xor   R2,(R3) ; What does this do???
      Jmp   Loop    ; go back to the top
      .end
```

✎ 144. Try running this code. Make sure you run it for at least 15 or 20 seconds. What does it do?

✎ 145. Consider the opcode for “Inc R0”; write it in binary.

✎ 146. Write the octal number 010 in binary.

✎ 147. The Xor instruction computes the xor (exclusive or) of the octal value 010 and the contents of memory address 0 (which is the opcode for “Inc R0”). Compute this xor yourself (see section 8.5 for the XOR’s truth table). What binary value do you end up with?

✎ 148. Convert that binary number to octal, and look up what instruction it corresponds to. Note that the Xor instruction writes this new value into memory address 0. Can you see why your program does what it does?

✎ 149. Now repeat this: take the xor of this new value with the octal number 010. Convert the result to octal and look up the instruction. What do you find?

This idea of changing instructions rather than writing two sets of code is very old, and was used in some of the earliest stored-program computers from the 1950s.

✎ 150. This would be very complex to code; but think about a program that lets you enter a number, press Interrupt, and that number would be stored at (say) address 200. The next number you enter would be stored (after pressing Interrupt again) at address 201; and so on. This is an example of a *loader*; a program that is used to load a set of values into memory. Usually, after a loader runs, the data that was loaded into memory is *executed*, i.e. the system would say “JMP 200” and would treat the numbers you entered as *code*. In this case, you’d be entering the values by hand; but on other systems, they might be coming from a paper tape reader or from a disk drive (or maybe from a network connection?)

14 Final Words

This is really just the beginning. The discussion, examples and problems presented in here are intended to help you understand computer design and programming at a low level. This may give you some ideas for other programs you can write.

The Touch Metal board has a few sample programs pre-loaded into it. The descriptions below are accurate as of the time of this writing, but may change in future versions. Check the Touch Metal website (https://touchmetal.org/page-sample.html) for more details.

✎ 151. Remove power from the board, then press and hold any button while applying power. At the prompt, press the number 1. Now press the Run/Halt button and observe what happens. This is our old friend, a program for adding consecutive numbers.

✎ 152. Repeat the power-up while holding a button, but this time, when prompted, press the number 2. Press Run/Halt and observe the output. If you write the displayed numbers in decimal, you may recognize them as *prime numbers* (numbers that are only divisible by themselves and 1).

✎ 153. Programs 3 is a dice-playing game, and program 4 is a (human) memory tester. The code for these programs is fairly involved, and makes good use of interrupts, register-indirect addressing and so on. Try these programs, look at the code, make changes, experiment!

✎ 154. Copy the random number generating code from program 3 or 4, and write a program that repeatedly displays a random number in the output register.

✎ 155. Write a program that displays a new random number each time the Interrupt button is pressed.

✎ 156. Can you code a high-low game? Have the computer think of a number (i.e. generate a random number). Let the user type in a guess followed by pressing the Interrupt key. The program can display 0 if the guess is low, 2 if it is high, and 1 if it is correct. Keep going until the user guesses the number.

✎ 157. Try flipping this around: let the user think of a number, and have the computer make a guess. The user enters 0, 2 or 1 (followed by the Interrupt key) to indicate whether the guess was too low, too high, or correct.

✎ 158. Make a stick-picking game. Type in the number of sticks in a pile, then take turns removing 1, 2 or 3 sticks from the pile. Whoever removes the last stick wins. On the computer's turn, it can simply display how many sticks it wants to take. On your turn, enter 1 2 or 3 and press the Interrupt button. The computer's move can be based on a random number, but there's also a strategy to winning this game. If you know it, you can use that to determine how many sticks the computer takes each turn.

✎ 159. Can you write a program that tests your reflexes? Start by displaying 000 in the output register. Generate a random number (say between 1 and 20), count down from there, and when you reach 0, display 377 in the output register. Start incrementing a register until the user presses the Interrupt button. The number of times you incremented tells you how quick the user responded.

✎ 160. Can you code a blackjack game? You will need to deal cards one at a time, decide how to request more cards, and so on.

What else can you do? Hopefully you're getting the idea that even on a very simple architecture with limited input/output options, ones own creativity can go far! I hope you'll keep exploring, experimenting, asking questions and working towards answers. Thanks for reading and sticking with all this.

Best wishes on your continuing journey!

APPENDIX - OPCODE TABLE

Opcode	Coding	Flags Affected	Function
NOP	000	-	No Operation
RET	001	-	Load PC from stack
HALT	002	-	Stop the CPU
CALL address	003 aaa	-	Save return address to stack then jump to address
JMP address	004 aaa	-	Load address into PC
JZ address	005 aaa	-	Jump if Z (zero flag) set
JNZ address	006 aaa	-	Jump if Z clear
JN address	007 aaa	-	Jump if N (negative flag) set
JNN address	010 aaa	-	Jump if N clear
JC address	011 aaa	-	Jump if C (carry flag) set
JNC address	012 aaa	-	Jump if C clear
JV address	013 aaa	-	Jump if V (overflow flag) set
JNV address	014 aaa	-	Jump if V clear
ADD src,dst	020 msd	VCNZ	src + dst → dst
SUB src,dst	021 msd	VCNZ	src - dst → dst
MUL src,dst	022 msd	VCNZ	src × dst → dst
DIV src,dst	023 msd	VCNZ	src ÷ dst → dst
AND src,dst	024 msd	NZ	src AND dst → dst
OR src,dst	025 msd	NZ	src OR dst → dst
XOR src,dst	026 msd	NZ	src XOR dst → dst
MOV src,dst	027 msd	NZ	src → dst
INTE	030	-	Enable interrupts
INTD	031	-	Disable interrupts
RTI	032	-	Remove PSW from stack then return from interrupt
LDI reg,value	07r iii	-	value → reg
INC reg	10r	CNZ	reg + 1 → reg
DEC reg	11r	CNZ	reg - 1 → reg
CLR reg	12r	-	0 → reg
NOT reg	13r	NZ	Compliment of reg → reg
ROL reg	14r	CNZ	Rotate reg left
ROR reg	15r	CNZ	Rotate reg right
IN reg	16r	NZ	Input register → reg
OUT reg	17r	-	reg → output register
LDI (reg),value	27r iii	-	value → mem[reg]
INC (reg)	30r	CNZ	mem[reg] + 1 → mem[reg]
DEC (reg)	31r	CNZ	mem[reg] - 1 → mem[reg]
CLR (reg)	32r	-	0 → mem[reg]
NOT (reg)	33r	NZ	Compliment of mem[reg] → mem[reg]
ROL (reg)	34r	CNZ	Rotate mem[reg] left
ROR (reg)	35r	CNZ	Rotate mem[reg] right
IN (reg)	36r	NZ	Input register → mem[reg]
OUT (reg)	37r	-	mem[reg] → output register

Conventions for Opcode Table

s,d,r

are each a 3-bit number:

```
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

mem[reg]

means the contents of memory at the address given by the value of reg. For example, if reg contains the value 015, mem[reg] is the contents of memory address 015

Alphabetical Index

0376....51
410....19
413-452....48
8-bit microprocessor....21
access mode....64
ADD....35
address....10, 64
addresses 400-407....18
addressing mode....33
and....37
architecture....26
assemble....25, 28
assembly language....7, 15, 19, 23, 25, 28
binary digit....31, 33
binary number....33
binary representation....33
bit....31, 33
bits....11
blackjack....61
Boolean logic....37
borrows....34
branch....23
byte....36
C....31, 39
CALL....46
calling....46
carries....34
carry....31, 33, 39
Central Processing Unit....9
checksum....58
Clear....41
clearing memory....11
CLR....41
code....17
Comments....25
complement....34
conditional jump....31
Counter....44
CPU....9
DEC....20
decimal....33
decisions....31
decrement....20
delay....45
Deposit button....10
destination....64
dice....61
Directives....25
disable....51
display modes....14
DIV....37
division....37
enable....51
exceptions....51
execute....13
fingerprinting....58
flag....
  cleared....31
  set....31
flags....31
free-run....14
free-running....23
front panel....27
function....46
General Purpose Registers....18, 57
HALT....30
Hand assembling....27
high-low....61
I....31, 51
Immediate....36
immediate value....64
IN....38
INC....18
increment....18
increments....14
Indirect Addressing....57
infinite loop....30
input register....9, 38
instruction set....17
instructions....17
IntE....51
internal registers....12
interrupt....51
Interrupt button....51
Interrupt Enable....31
interrupt handler....51
invcz....39
IVNCZ....15
JNZ....32
jump....23
Jump if Not Zero....32
Jump if Zero....32
JZ....32
label....26

Labels....25
LDI....36
Load Immediate....36
loader....60
local variables....48
logic operations....37
loop....23, 26, 31
machine code....25, 26
MD5 hash....58
mem[reg]....64
memory....10
memory address....10
metal....25
mnemonics....25
modularization....45
most significant bit....35
MOV....40
move....40
msb....35
MUL....37
multiplication....37
N....21, 31, 39
negative....31, 39
negative flag....21
negative number....21
negative numbers....34
NOP....17
NOT....41
octal....11
Octal/Decimal Conversion Tables....12
operand....19, 23
or....37
Org....27
OUT....29
output register....29
overflow....31, 40
PC....14, 29
powers of 2....33
prime numbers....61
priority levels....51
Processor Status Word....39
program....19
Program Counter....14, 29
Programmer's Card....26
programs....43
PSW....31, 39
R0....12, 18
R0-R7....57
random number....61
recursion....49
reflexes....61
register....
destination....36
source....36
register direct....57
register indirect....57
registers....18, 38
Reset button....11
RET....46
Return....46
return address....47
return from interrupt....51
ROL....41
roll over....20
ROR....42
Rotate Left....41
Rotate Right....42
RTI....51
Run/Halt button....14
Running Sum....44
S[00]....48
sample programs....61
self-modifying code....59
shift....41
sign bit....35
single-stepping....30
software interrupts....51
source....64
SP....15, 47
stack....47
stack addresses....48
stack overflow....49
stack pointer....15, 47
stack underflow....48
status bit....15
status flags....15
Step button....14
stick-picking....61
stored-program....59
SUB....37
subroutine....46
Subtraction....34, 37
Touch Metal Instruction Manual....26, 28
Touch Metal Programmer's Card....**39**
truth table....37
two's complement....34
two's complement notation....21
two's compliment notation....20
V....31, 40
xor....37
Z....31, 39
Z bit....31
Zero....39
Zero flag....31

-1.....20
-128.....35
+0.....35
+127.....35
✎.....7

NOTES

NOTES

NOTES

NOTES

NOTES