HANDv8 and The Sqrter for Dummies

1st Edition

Understanding My Revolutionary Contribution
to the Field of Computer Science

Daniel Detore, ddetore@stevens.edu
CS 382-D
Professor Shudong Hao
I pledge my honor that I have abided by the Stevens Honor System.

December 2, 2024

There is no reason for any individual to have a computer in his home.

- Ken Olsen

Contents

Author's Note **iii**

I What You Need To Know 1

1 Programming in HANDv8 **2**
1.1 Instructions 2
1.2 Data Memory 3

2 Assembling **5**

3 Running **6**

II What You (Don't) Want To Know 7

4 What's with the name? **8**

5 How does the machine code work? **9**

Author's Note

I am a computer scientist. I am neither an electrical engineer nor a mathematician. Anyone with meaningful knowledge of those fields (including computer science) will, in all likelihood, be vexed and enraged by my decisions herein. Please direct any complaints to your local recycling center. Questions or wellness checks can be sent to my email on the cover page.

This manual is dedicated to my parents, who are alive but would be instantly sent into cardiac arrest if I tried explaining any of this technology to them.

Part I

What You Need To Know

Section 1: Programming in HANDv8

You get 4 registers and each has 8 bits of storage. We call them `R0`, `R1`, `R2`, and `R3`. Their values are treated as unsigned because we don't have the budget for signed arithmetic. Put your instructions in a `.s` file under the `.text` header if you plan on also having a `.data` segment. Otherwise, the interpreter will assume your file only contains instructions.

These instructions are loosely based on ARMv8 names and syntaxes, but I made them look more uniform.

1.1 Instructions

When instructions take multiple registers as parameters, they can be different or the same. All immediates are unsigned and 8 bits long, where 0 is always valid.

Miscellaneous

`END` Ends the program. The interpreter will ignore any instructions after this instruction. This is not necessary as the program terminates at the end of the `.text` section anyway, but it makes you look more professional or something.

`MOV Rw Ra` Overwrites the contents of `Rw` with that of `Ra`. This does not effect `Ra`.

`MOV Rw imm8` Overwrites the contents of `Rw` with `imm8`.

Arithmetic

`ADD Rw Ra Rb` Writes into `Rw` the sum of the contents of `Ra` and `Rb`.

`ADD Rw Ra imm8` Writes into `Rw` the sum of the content of `Ra` and `imm8`.

`MUL Rw Ra imm8` Writes into `Rw` the product of the contents of `Ra` $\times$ `imm8`.

`DIV Rw Ra Rb` Writes into `Rw` the integer quotient $\lfloor \frac{\texttt{Ra}}{\texttt{Rb}} \rfloor$. If `Rb` $= 0$, then `Rw` $= 0$.

`DIV Rw Ra imm8` Writes into `Rw` the integer quotient $\lfloor \frac{\texttt{Ra}}{\texttt{imm8}} \rfloor$. If `imm8` $= 0$, then `Rw` $= 0$.

Memory Access

`LDR Rw Ra` Writes into `Rw` a byte from data memory which is addressed by the content of `Ra`.

`LDR Rw Ra Rb` Writes into `Rw` a byte from data memory which is addressed by the sum of the contents of `Ra` + `Rb`.

`LDR Rw Ra imm8` Writes into `Rw` a byte from data memory which is addressed by the sum of the content of `Ra` + `imm8`.

`STR Rw Ra` Stores the content of `Rw` into data memory addressed by the contents of `Ra`.

`STR Rw Ra Rb` Stores the content of `Rw` into data memory addressed by the contents of `Ra` + `Rb`.

`STR Rw Ra imm8` Stores the content of `Rw` into data memory addressed by the content of `Ra` + `imm8`.

1.2 Data Memory

The interpreter will let you declare a `.data` header as well, if you want to have some data already stored in memory. If you don't, it won't generate a data memory image file.

You can set one byte of memory as such:

```
.data
AA = BB
a6 = ff
```

In this case, the value `0xBB` gets written to memory at address `0xAA`. **TAKE NOTE** that both values are written in **hexidecimal**, whereas immediates in `.text` use **decimal**.

You can also use this syntax to set a range of values starting at one address:

```
.data
0xAA: 0xNN, 0xNN, ...
08:  ff, ab, 00, c1
```

In this case, each subsequent byte of data, starting with `0xAA` will contain the following arguments. This means address `08` contains `ff`, address `09` contains `ab`, and so on.

If you like, you can mix and match these syntaxes within `.data`.

```
.data
00:  a0, a1, a2, a3
8 = 1
```

The above code snippet will have your memory looking as such:

address	00	01	02	03	04	05	06	07	08	...
contents	a0	a1	a2	a3	00	00	00	00	01	00

Just be aware that if you assign multiple values to the same address, the interpreter will only use whichever one you assign last chronologically.

Comments

The interpreter will ignore anything between the characters // and the end of a line. This works in all sections anywhere in the file.

Section 2: Assembling

You've got your instructions and your data. To get it into the Sqrtr, you'll want to use the interpreter. First, get `interpreter.py` into the same directory as your file. Run it in Powershell or Batch with `py interpreter.py <yourfile>.s`. You can leave off the `.s` if you like, but it's good luck to include it.

The interpreter will quit and spit out a broken file if...

- you use a bad (misspelled or nonexistent) instruction. No matter how bad you want it, the Sqrtr does not have `SUB`.
- you use an invalid syntax. You cannot multiply by a register. I am deeply sorry.
- you use too many instructions. The instruction memory can only fit 255 instructions. (`MUL` uses `imm8` instructions, so be careful with it.) You can mod it to hold more, but it will void your warrantee.
- you forget to separate `.data` and `.text` sections with their respective headers.

The interpreter will complain but still work if...

- there are instructions after `END`.
- you `ADD` an immediate 0 to a register. The interpreter will optimize its output by excluding this line.
- you `DIV` or `MUL` a register by 1. Same deal as adding 0.

If you only have a `.text` section, you will only get an instruction memory file. Once you have the Sqrtr open in Logisim-evolution, you can right-click on the `instructionMemory`, click "Load Image..." and open the instruction file. If you also have a `.data` section, follow the same steps with the `dataMemory`.

Section 3: Running

Once you've loaded everything into Logisim-evolution, you can run it however you please. I personally reccomend choosing the poke tool and ticking clock twice for each instruction. You may also use Auto-tick, or you can mash `Ctrl+T` or `Ctrl+F9` until the program has completed. Be aware, however, that when the PC reaches the end of the memory, it will overflow and restart the program from the top.

There's really nothing to do with the data once the program has completed other than look at it with a sense of fatherly/motherly pride, so feel free to do that for as long as you need.

Part II

What You (Don't) Want To Know

Section 4: What's with the name?

My mother gave it to me.

The CPU is called the Sqrtr (i.e. SQuare RooTeR) because it has the capability to approximate the integer square root of a number. I needed some sort of driver program to work toward, and I chose that one. It's stupid because it's only the *integer* square root of a number bounded by $[0, 255]$, but who cares? It's funny.

The language is called HANDv8 because it's based on ARMv8, just as your hand is based on your arm.

Section 5: How does the machine code work?

It depends heavily on the command and its specific syntax. Most generally, every instruction gets translated into a 2-byte binary number of this form:

$$abcdeeff gghhhhhh \tag{5.1}$$