HANDv8 and the Sqrter for Dummies

1st Edition

A Once-In-A-Lifetime Guide to
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 5, 2024

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

- Ken Olsen

Contents

Author's Note

I am a computer scientist. I am neither an electrical engineer nor a mathematician. Anyone with meaningful knowledge of these 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

Chapter 1: Programming in HANDv8

The Sqrtr 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. The sections can be in any order. The entire language is case-insensitive but I usually stick to making everything capital except for comments for reasons I don't remember.

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. Register numbers and immediates must be written in decimal.

`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 lets the interpreter finish slightly faster.

Arithmetic

`ADD Rw Ra Rb`	Writes into `Rw` the sum of the contents of `Ra` + `Rb`.
`ADD Rw Ra imm8`	Writes into `Rw` the sum of the content of `Ra` + `imm8`.
`DIV Rw Ra Rb`	Writes into `Rw` the integer quotient $\lfloor \frac{Ra}{Rb} \rfloor$. If `Rb` $= 0$, then `Rw` $= 0$.
`DIV Rw Ra imm8`	Writes into `Rw` the integer quotient $\lfloor \frac{Ra}{imm8} \rfloor$. If `imm8` $= 0$, then `Rw` $= 0$.

Note that, just like the regular mathematical operations, `ADD` is commutative while `DIV` is not. That is, swapping `Ra` and `Rb` in an `ADD` instruction will produce the same result (though for the interpreter's sake, `imm8` must always come last). Doing the same for `DIV` does not necessarily give equal results.

You can emulate `MOV Rw Ra` by using `ADD Rw Ra 0`. You can also do `MOV Rw imm8` in two instructions with `DIV Rw Rw 0` then `ADD Rw Ra imm8`.

The Sqrtr does not have `SUB` or `MUL` or a built-in emulation of either. The implementation of either is left as a challenge to the reader.

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

1.2 Data Memory

You can include a `.data` header in your program if you want to have some data already stored in memory. The Sqrtr's data memory only holds 256 bytes.

You can set one byte of memory as such:

```
.data
0xAA = 0xBB
a6 = ff
```

In this case, the value `0xBB` gets written to memory at address `0xAA`. **TAKE NOTE** that both values are written in **hexadecimal**, whereas immediates in `.text` use **decimal**. It works this way to make the data memory easier to debug.

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. Be careful not to comment out anything you actually need. There are no multiline comments. If you are doing something complicated enough to require multiline comments you should be using a better language on a mor expensive CPU.

I will not prescribe good coding practices because I don't care. To see how the language is best used, check the included demo files.

Chapter 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. Note that it will overwrite any existing files with the names it's trying to output to. It will output `<yourfile>-instructions.txt` and `<yourfile>-data.txt`.

The interpreter will rage quit and possibly 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 load from an immediate address. I am deeply sorry.
- you use too many instructions. The instruction memory can only fit 256 instructions. You can mod the CPU to hold more, but it will void your warrantee.
- you forget to separate `.data` and `.text` sections with their respective headers.

This is by no means an exhaustive list, so double-check your code and heed the interpreter's warnings before you email me.

Once you have the Sqrtr open in Logisim-evolution, you can right-click on the `instructionMemory`, click "Load Image..." and open `<yourfile>-instructions.txt`. If you also have a `.data` section, follow the same steps to load `<yourfile>-data.txt` into the `dataMemory`.

Chapter 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 manually ticking the 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, or otherwise parental pride, so feel free to do that for as long as you need. If you think of something else, let me know.

Part II

What You (Don't) Want To Know

Chapter 4: On the construction

4.1 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. If you're wondering, I took the brackets out of the syntaxes for `LDR` and `STR` so that they look just as confusing as all the other instructions. I also lost the commas because they were harder to parse through with Python.

4.2 Why does HANDv8 have DIV and not SUB?

I could only fit one ALUop bit into the machine code. (It'll make more sense in a few pages.) Plus, I'd need to add two more operations if I did add another, which is too much freedom for this project. Aside from the required `ADD`, I chose division because it is a compulsory operation to do what the Sqrtr set out to do, which is calculate the integer square root of an 8-bit number. You can find my implementation of the algorithm in `sqrt.s`.

The algorithm I settled on is an adaptation of Heron's method. Instead of using ϵ and checking for accuracy, I just ran the algorithm a bunch and found that for all 8-bit numbers, running the algorithm 4 times computes the correct integer square root. For the nerds: to find $\lfloor \sqrt{n} \rfloor$, I use the seed guess of $root = \lfloor \frac{n}{2} \rfloor$ and repeat this approximation 4 times:

$$root = \lfloor \frac{\lfloor \frac{n}{root} \rfloor + root}{2} \rfloor.$$

4.3 As a mathematician, this vexes me.

I guessed. Because we are in a world of my creation, I decided to define the famously undefined operation of dividing by 0. I needed a way to zero a register with one operation, so I defined zero-division to do that. It was very easy to implement the simple conditional of `if (denominator == 0) then return 0` so I did. I couldn't find any reputable division circuit that did this for me, so I heavily adapted the work of one SDSpivey [1] and took out all that nasty electrical engineering stuff.

[1] You can find his work at https://github.com/logisim-evolution/logisim-evolution/discussions/1660.

Chapter 5: On the interpreter

5.1 How does the interpreter work?

The interpreter will output Logisim-evolution-ready memory images. I'll be general here because the interpreter is written in Python. Had I written it in a worse language, I'd consider going really in-depth. If you really need to know exactly how it works, you can go and look at it.

In case you forgot, its useage is as such: `py interpreter.py yourfile.s`

It goes line by line and identifies what you're doing with each instruction. It takes the instruction, its syntax, register numbers, and immediates (if applicable) and translates it into machine code. It outputs at most two files. Check Section 2: Assembling for more information on how to use it.

5.2 Machine Code

The interpreter translate every instruction into a 2-byte binary number of this form:

$$a, b, c, d, ee, ff, gg, hhhhhh.$$

1. The first 4 bits are the instruction's opcode.
    a) a represents `ReadToReg`. It is a switch to tell the register file whether it's writing to the register from data memory (1) or from the ALU (0).
    b) b represents `MemWrite`. It is a switch to tell the CPU whether this instruction involves writing to data memory (1) or a register (0).
    c) c represents `ALUop`. This switch is 0 to do addition, which is required by all instructions except `DIV`, and 1 to do division.
    d) d represents `UseImm` which is quite self-explanatory. If 1, the ALU receives the immediate which is stored in bits $gghhhhhh$. If 0, the ALU will receive the contents of the register number stored in bits gg. While the contents of $hhhhhh$ are irrelevant if `UseImm` $= 0$, the interpreter will set them all to 0.
2. ee is `WriteReg`. For all syntaxes of all instructions except `STR`, a register is written to. This is the number of that register. For `STR`, these bits are irrelevant so the interpreter will set them to 0.
3. ff is `ReadReg1`. For all syntaxes of all instructions, at least one register must be read from. I usually call this `Ra`. This is the number of the mandatory one.

4. *gg*, if and only if `UseImm` = 0, is `ReadReg2`. For some instructions, two registers must be read from. This is the number of the second one. I usually call this `Rb`. For `STR Rt Ra Rb`, this is `Rt`.

5. *gghhhhhh*, if and only if `UseImm` = 1, is the immediate to be used as `imm8`.

For reference, here are some example instructions and their machine code equivalents:

```
ADD R1 R0 R3   =  0000010011000000
ADD R2 R1 15   =  0001100100001111
DIV R3 R2 R1   =  0010111001000000
DIV R0 R3 5    =  0011001100000101
LDR R0 R1      =  1001000100000000
LDR R1 R2 4    =  1001011000000100
LDR R2 R3 R0   =  1000101100000000
STR R0 R1             =
STR R3 R2 8           =
STR R2 R0 R1          =
```

Chapter 6: Conclusions

> No one ever said on their deathbed, "Gee, I wish I had spent more time alone with my computer."
>
> Dani Berry

I feel feverish and unwell.