HANDv8 and the Sqrtr 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 8, 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.

Chapter 1: The Sqrtr

Let's get this out of the way so we can get to work:

1.1 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 it uses is called HANDv8 because it's based on ARMv8, just as your hand is based on your arm. It's not quite ARMv8 because I made some visual changes to the syntax to make it easier to interpret. The changes will become apparent in the next chapter.

1.2 Meet the Sqrtr

The Sqrtr is a simulation of a sequential CPU made for Logisim-evolution.

On a computer, all of your important data is in the data memory. However, doing any meaningful amount of operations directly on data in the RAM is unacceptably slow. We need to load data into the Sqrtr's registers, do all necessary operations there, and them move it back. The Sqrtr has 4 general purpose registers and each has 8 bits of storage (which can hold one addressed portion of RAM data or even an address). We call them `R0`, `R1`, `R2`, and `R3`. They don't go by `W` or `X` because the Sqrtr's registers store bytes, not words or double words. Using `R` avoids the confusion of whether you should be calling registers `W` or `X`, since neither applies.

You can write instructions in HANDv8 to load data into registers from the memory, do operations on them, and send them back to the data memory. You can also use it to start your program with some data already stored in the data memory. The Sqrtr treats all numbers as positive because we don't have the budget for signed arithmetic (remember, it's an 8-bit processor).

The Sqrtr is not pipelined or forwarded, and as such it is purely sequential. It takes one clock cycle to execute one instruction and it only executes one instruction at a time.

Chapter 2: Programming in HANDv8

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, but you cannot have one inside the other. The entire language is case-insensitive but I usually stick to making everything capital except for comments for reasons I don't remember. I will otherwise not prescribe good coding practices because I don't care. To see how the language is best used, check the included demo files.

2.1 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.

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`.

The syntaxes of `LDR` and `STR` with no third argument are shorthand for `LDR Rw Ra 0` and `STR Rt Ra 0`. The first argument of `STR` is called `Rt` rather than `Rw` to signal that you are not writing to the first register, whereas you are doing so in all other instructions.

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.

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 also cannot do math with two immediate operands. If you've made it this far, you can add or floor divide two numbers on your own before you write the program.

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.

You can emulate `MOV Rw Ra` by using `ADD Rw Ra 0`. You can also emulate `MOV Rw imm8` in two instructions with `DIV Rw Rw 0` then `ADD Rw Rw imm8`. The interpreter won't translate these for you because it threatened to resign if I added more instructions without increasing its salary.

2.2 Data Memory

You can include a `.data` header in your program if you want to have some data for your program to load in. The Sqrtr's data memory only holds 256 bytes because the register can only store 256 different addresses anyway.

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.

You may be wondering, "why have two syntaxes if you can do with the colon syntax the same thing you can do with the equals syntax?" My answer is that programmers love needlessly adding specifications to their languages. The equals syntax is exclusively to set *one* memory address. If you add more the interpreter will complain and refuse. The equals syntax keeps you accountable.

2.3 Comments

The interpreter will ignore anything between the characters `//` and the end of a line, no matter what is on the other side. 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 more expensive CPU.

2.4 FAQ

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:

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

Since it's quick enough to brute force, you can check that the final $root$ will always be exactly $\lfloor\sqrt{n}\rfloor$ for all $n \in [0, 255]$.

As a mathematician, your `DIV` vexes me.

Not a question. 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 easy enough 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 3: Assembling

You've got your instructions and your data in a `.s` file. To get it into the Sqrtr, you'll want to use the interpreter. It will will output Logisim-evolution-ready memory images.

First, get `interpreter.py` into the same directory as your file. Run it in Powershell, Batch, or Shell with `python interpreter.py <yourfile>.s`. You can exclude the `.s` if you like, but it's good luck to include it. It will output `<yourfile>-instructions.txt` and `<yourfile>-data.txt`. If these files already exist, the interpreter will overwrite them.

The interpreter will throw a tantrum and quit if...

- you run the interpreter the wrong way. It will remind you of its usage.
- you use a bad (misspelled or nonexistent) instruction. It will tell you what the bad instruction is and what line it was on.
- you use an invalid syntax. It will tell you what was wrong with it, unless it was so bad it crashes the interpreter entirely.
- you use too many instructions. The instruction memory can only fit 256 instructions. You can mod it to hold more, but it will void your warrantee. You can do as much in `.data` as you want because it is handled by the interpreter, not the RAM.
- you use an immediate anywhere that is not in $[0, 255]$. Any number outside of this range cannot be represented by 8 bits of unsigned binary or 2 digits of hexadecimal.
- you forget to separate `.data` and `.text` sections with their respective headers. This has undefined behavior but will most likely cause one of the above messages.

The interpreter will complain but continue (assuming no other rules are broken) if...

- you use `END` but have more data after it. It will let you know that you might have prematurely `END`ed the program, but it won't stop you.
- you don't include any labels. It will assume everything is `.text` and move on. If it assumes wrong, the interpreter's behavior is undefined.
- you only include `.data` and no `.text`. I don't know why you'd want to, but you can.

This is not an exhaustive list, so double-check your code and heed the interpreter's warnings before you email me.

Chapter 4: Running

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 have a nontrivial `.data` section, follow the same steps to load `<yourfile>-data.txt` into the `dataMemory`.

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.

Chapter 5: Machine Code

This is not very important to the programmer, but intimate knowledge of machine code is absolutely vital to fully understanding the Sqrtr. The interpreter translates every instruction into a 2-byte binary number of this form:

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

To understand *how* the Sqrtr reads it, we must understand what each segment is.

1. The first 4 bits are the instruction's opcode. Each bit is a switch, so each only needs to be one bit.
    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`. 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. The ALU always receives `ReadReg1` so there's no switch for it.
2. ee is `WriteReg`. For all syntaxes of all instructions except `STR`, a register is written to. This is the number of that register, encoded into binary. It needs 2 bits to hold the labels in $[0, 4]$. For `STR`, this is `Rt`, to signal that `Rt` does not get written to.
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, encoded into binary. It needs 2 bits to hold the labels in $[0, 4]$.
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, encoded into binary. It needs 2 bits to hold the labels in $[0, 4]$. I usually call this `Rb`.
5. $gghhhhhh$, if and only if `UseImm` $= 1$, is the immediate to be used as `imm8`. While the contents of $hhhhhh$ are irrelevant if `UseImm` $= 0$, the interpreter will set them all to 0. Of course, this field needs to be 8 bits as to fit a full `imm8`.

More specifically, here is what the opcode and layout looks like for each command and their syntaxes:

Instruction	ReadToReg	MemWrite	ALUop	UseImm	WriteReg	ReadReg1	ReadReg2	imm
ADD Rw Ra Rb	0	0	0	0	0bw	0ba	0bb	-
ADD Rw Ra imm8	0	0	0	1	0bw	0ba	-	0bimm8
DIV Rw Ra Rb	0	0	1	0	0bw	0ba	0bb	-
DIV Rw Ra imm8	0	0	1	1	0bw	0ba	-	0bimm8
LDR Rw Ra Rb	1	0	0	0	0bw	0ba	0bb	-
LDR Rw Ra imm8	1	0	0	1	0bw	0ba	-	0bimm8
STR Rt Ra imm8	0	1	0	1	0bt	0ba	-	0bimm8

Note: I excluded syntaxes of LDR and STR with no third argument. They are shorthand for their respective imm8 syntaxes with $\texttt{imm8} = 0$.

This construction leaves open the possibility of a 4th secret memory interaction when ReadToReg and MemWrite are both 1, but now is not the time to theorize on something like that.

The interpreter takes the numbers generated by this scheme, translates them into hexadecimal, and then puts them into a format that Logisim-evolution can read. The Sqrtr then translates it back into decimal and sends each bit (or group of bits) where it needs to go. The datapath is near enough to Figure 3.27 in the textbook (with some simplifications and elaborations) that it would patronize the reader if I explained it at length.