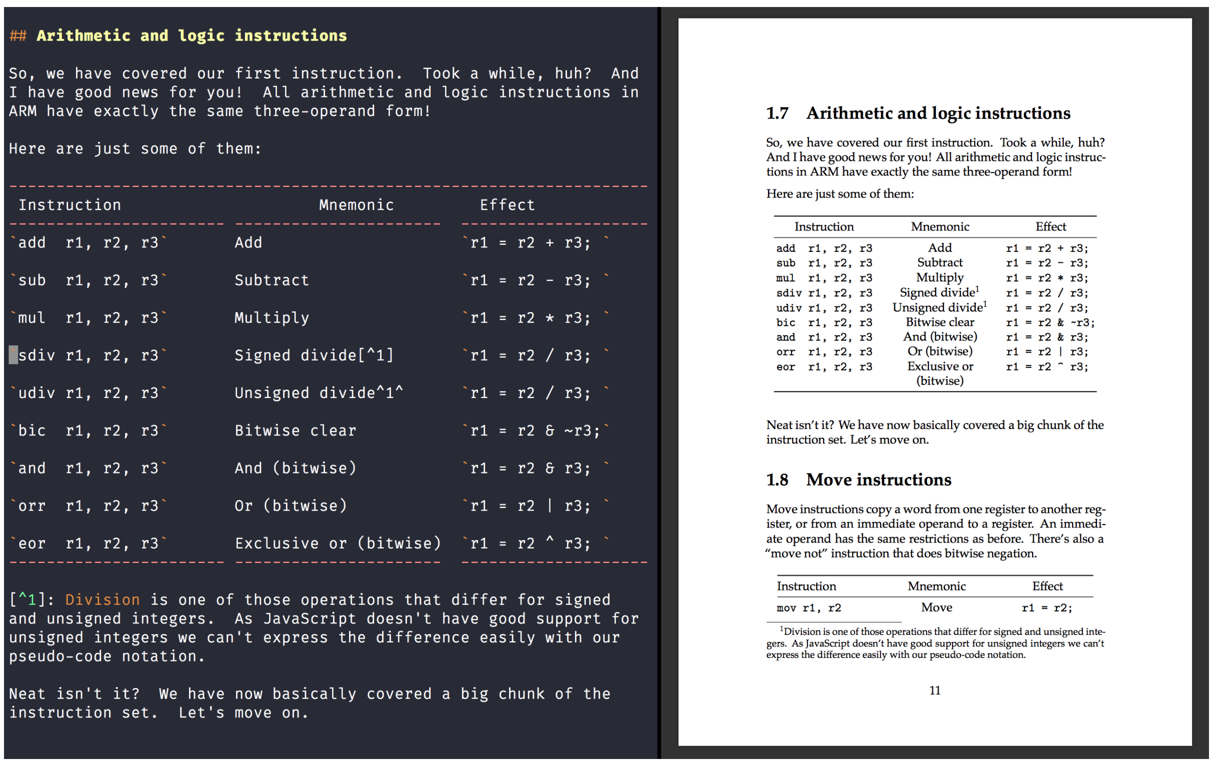
Writing a Book
with
Pandoc, Make, and Vim

2020-04-12

As you might know, I'm writing a book called Compiling to Assembly from Scratch. Recently I tweeted about my book-writing setup, and there was a lot of interest in the details of my setup. I'm only halfway through with writing the book (or, so I think), so my setup will likely to change as I go. But here it is, anyway.

A screenshot:



```
## Arithmetic and logic instructions

So, we have covered our first instruction.  Took a while, huh?  And
I have good news for you!  All arithmetic and logic instructions in
ARM have exactly the same three-operand form!

Here are just some of them:

-------------------------------------------------------------------
 Instruction                     Mnemonic          Effect
----------------------  ----------------------  ------------------
`add  r1, r2, r3`        Add                     `r1 = r2 + r3; `

`sub  r1, r2, r3`        Subtract                `r1 = r2 - r3; `

`mul  r1, r2, r3`        Multiply                `r1 = r2 * r3; `

`sdiv r1, r2, r3`        Signed divide[^1]       `r1 = r2 / r3; `

`udiv r1, r2, r3`        Unsigned divide^1^      `r1 = r2 / r3; `

`bic  r1, r2, r3`        Bitwise clear           `r1 = r2 & ~r3;`

`and  r1, r2, r3`        And (bitwise)           `r1 = r2 & r3; `

`orr  r1, r2, r3`        Or (bitwise)            `r1 = r2 | r3; `

`eor  r1, r2, r3`        Exclusive or (bitwise)  `r1 = r2 ^ r3; `
----------------------  ----------------------  ------------------

[^1]: Division is one of those operations that differ for signed
and unsigned integers.  As JavaScript doesn't have good support for
unsigned integers we can't express the difference easily with our
pseudo-code notation.

Neat isn't it?  We have now basically covered a big chunk of the
instruction set.  Let's move on.
```

1.7 Arithmetic and logic instructions

So, we have covered our first instruction. Took a while, huh? And I have good news for you! All arithmetic and logic instructions in ARM have exactly the same three-operand form!

Here are just some of them:

Instruction	Mnemonic	Effect
`add  r1, r2, r3`	Add	`r1 = r2 + r3;`
`sub  r1, r2, r3`	Subtract	`r1 = r2 - r3;`
`mul  r1, r2, r3`	Multiply	`r1 = r2 * r3;`
`sdiv r1, r2, r3`	Signed divide[1]	`r1 = r2 / r3;`
`udiv r1, r2, r3`	Unsigned divide[1]	`r1 = r2 / r3;`
`bic  r1, r2, r3`	Bitwise clear	`r1 = r2 & ~r3;`
`and  r1, r2, r3`	And (bitwise)	`r1 = r2 & r3;`
`orr  r1, r2, r3`	Or (bitwise)	`r1 = r2 | r3;`
`eor  r1, r2, r3`	Exclusive or (bitwise)	`r1 = r2 ^ r3;`

Neat isn't it? We have now basically covered a big chunk of the instruction set. Let's move on.

1.8 Move instructions

Move instructions copy a word from one register to another register, or from an immediate operand to a register. An immediate operand has the same restrictions as before. There's also a "move not" instruction that does bitwise negation.

Instruction	Mnemonic	Effect
`mov r1, r2`	Move	`r1 = r2;`

[1]Division is one of those operations that differ for signed and unsigned integers. As JavaScript doesn't have good support for unsigned integers we can't express the difference easily with our pseudo-code notation.

11

I'm on macOS, so I'm using the native full-screen split functionality. On the left, I have Terminal.app running Vim, and on the right, I have Preview.app showing—ahem—a preview. Both apps ship with macOS.

I'm writing the book in Markdown, then use Pandoc to convert it to PDF and EPUB.

Pandoc

I use Pandoc as my Markdown processor. It is a great tool, written in Haskell, and available from most package managers. It converts from and to many different document formats. Still, I am mainly interested in PDF and EPUB for my book.

While Pandoc supports CommonMark and GitHub-flavored Markdown, I am using the Pandoc dialect of Markdown. It has support for many extensions: footnotes, different styles of tables, math, etc.

For example, the table style that I'm using in the screenshot allows changing column width and proportions by changing the Markdown column widths.

Although I'm using Make to run Pandoc, here's an equivalent shell command:

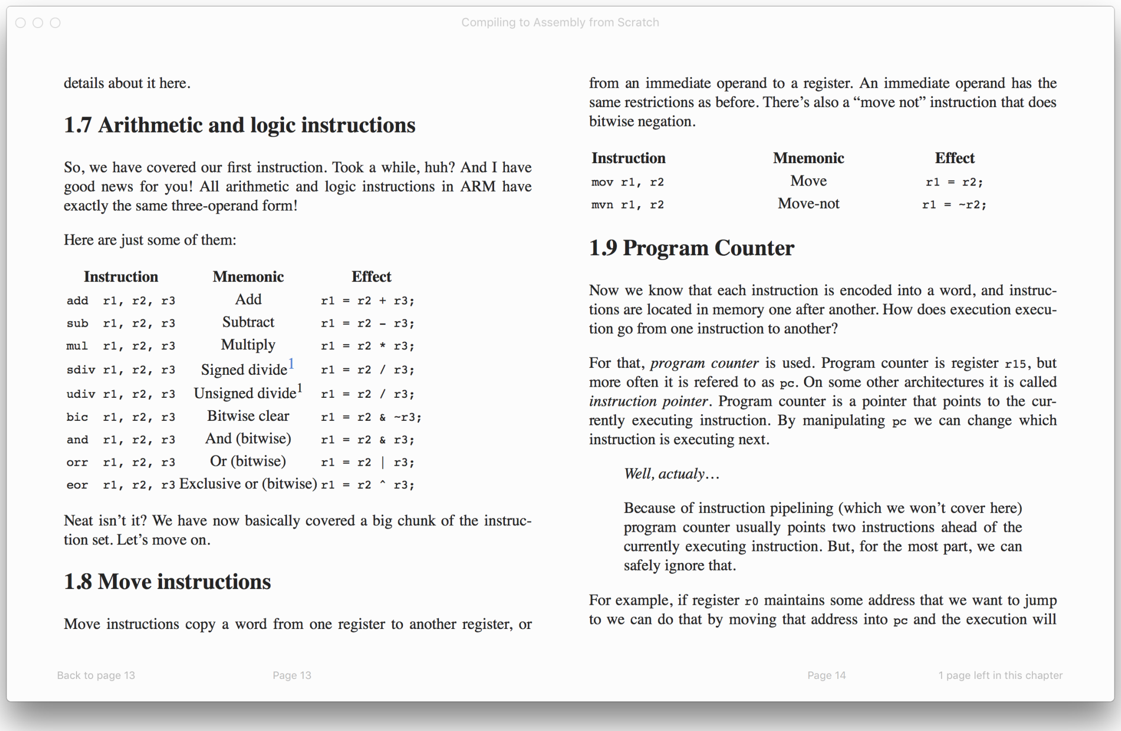
```
$ pandoc book.md -o book.pdf             \
    --table-of-contents                  \
    --number-sections                    \
    --pdf-engine=xelatex                 \
    --indented-code-classes=javascript \
    --highlight-style=monochrome         \
    -V mainfont="Palatino"               \
    -V documentclass=report              \
    -V papersize=A5                      \
    -V geometry:margin=1in
```

By default, to produce PDF, Pandoc converts the document to LaTeX first, then calls `pdflatex` to produce the PDF. Howerver, with `--pdf-engine=xelatex` I opted in to use XeTeX together with `xelatex` command to produce PDF.

First, XeTeX allows to use Unicode characters in the source of the document. I'm quite accustomed to entering en-dash and em-dash and other special characters using option-key shortcuts on macOS. Second, XeTeX allows to use arbitrary system's fonts, and that's what I did with *Palatino*.

I've set up some preliminary paper size and margin and LaTeX `documentclass`, but I will definitely be tweaking them before releasing the book.

Producing EPUB with Pandoc also works fine, but I'm sure I will do more tweaking there as well:



details about it here.

1.7 Arithmetic and logic instructions

So, we have covered our first instruction. Took a while, huh? And I have good news for you! All arithmetic and logic instructions in ARM have exactly the same three-operand form!

Here are just some of them:

Instruction	Mnemonic	Effect
`add  r1, r2, r3`	Add	`r1 = r2 + r3;`
`sub  r1, r2, r3`	Subtract	`r1 = r2 - r3;`
`mul  r1, r2, r3`	Multiply	`r1 = r2 * r3;`
`sdiv r1, r2, r3`	Signed divide[1]	`r1 = r2 / r3;`
`udiv r1, r2, r3`	Unsigned divide[1]	`r1 = r2 / r3;`
`bic  r1, r2, r3`	Bitwise clear	`r1 = r2 & ~r3;`
`and  r1, r2, r3`	And (bitwise)	`r1 = r2 & r3;`
`orr  r1, r2, r3`	Or (bitwise)	`r1 = r2 | r3;`
`eor  r1, r2, r3`	Exclusive or (bitwise)	`r1 = r2 ^ r3;`

Neat isn't it? We have now basically covered a big chunk of the instruction set. Let's move on.

1.8 Move instructions

Move instructions copy a word from one register to another register, or from an immediate operand to a register. An immediate operand has the same restrictions as before. There's also a "move not" instruction that does bitwise negation.

Instruction	Mnemonic	Effect
`mov r1, r2`	Move	`r1 = r2;`
`mvn r1, r2`	Move-not	`r1 = ~r2;`

1.9 Program Counter

Now we know that each instruction is encoded into a word, and instructions are located in memory one after another. How does execution execution go from one instruction to another?

For that, *program counter* is used. Program counter is register `r15`, but more often it is refered to as `pc`. On some other architectures it is called *instruction pointer*. Program counter is a pointer that points to the currently executing instruction. By manipulating `pc` we can change which instruction is executing next.

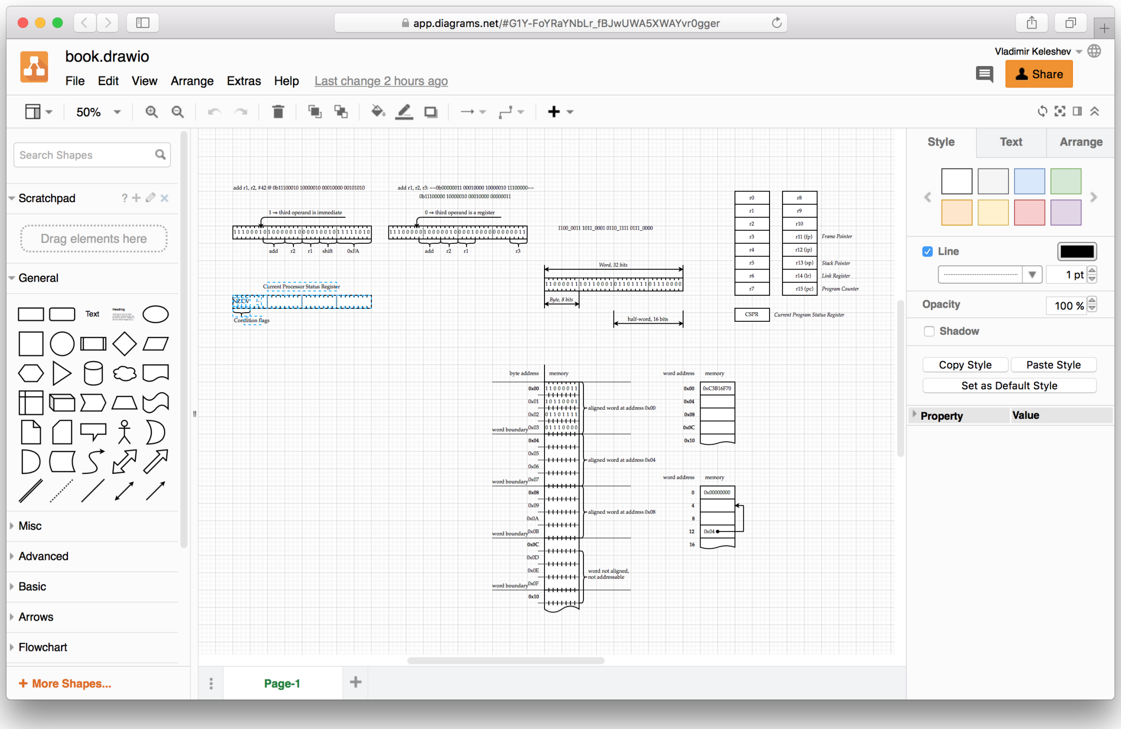
> *Well, actualy…*
>
> Because of instruction pipelining (which we won't cover here) program counter usually points two instructions ahead of the currently executing instruction. But, for the most part, we can safely ignore that.

For example, if register `r0` maintains some address that we want to jump to we can do that by moving that address into `pc` and the execution will

Figures

I've been using draw.io↗ for various diagrams at work and for this blog. But for the book, I wanted to use something open-source, something reliable, something that would not *incredible journey* me over halfway through writing the book, leaving me without a critical tool.

So I started searching, and imagine my surprise when I learn that draw.io is, in fact, open-source↗. That was a relief!



I've been using one big draw.io document so far and I exported individual figures to SVG by using *export selection* feature.

SVG is well supported with EPUB, since EPUB is just glorified HTML and CSS under-the-hood. For PDF output, Pandoc requires to install `librsvg` (written in Rust) and handles the conversion transparently.

Makefile

Even though my book right now is written in a single Markdown file, I started using Make anyway from the get-go. I'm using my system's build-in GNU Make. Here's my Makefile:

```
.PHONY: phony

FIGURES = $(shell find . -name '*.svg')

PANDOCFLAGS =                         \
  --table-of-contents                 \
  --pdf-engine=xelatex                \
  --from=markdown                     \
  --number-sections                   \
  --indented-code-classes=javascript \
  --highlight-style=monochrome        \
  -V mainfont="Palatino"              \
  -V documentclass=report             \
  -V papersize=A5                     \
  -V geometry:margin=1in

all: phony output/book.pdf

output/%.pdf: %.md $(FIGURES) Makefile | output
        pandoc $< -o $@ $(PANDOCFLAGS)

output/%.epub: %.md $(FIGURES) Makefile | output
```

```
        pandoc $< -o $@ $(PANDOCFLAGS)

output:
        mkdir ./output

clean: phony
        rm -rf ./output

open: phony output/book.pdf
        open output/book.pdf
```

A few things that are worth mentioning:

`Makefile` itself is a dependency of some rules. This is done so that if I change a font (or some other Pandoc flag), Make would pick it up and rebuild the target.

The list of all figures (`*.svg` glob) is a dependency as well, to make sure that the book is rebuilt when a figure is updated.

The build artifacts are stored in the `output` folder, which is an order-only prerequisite↗ (specified with "`| output`" syntax). This is useful because we don't care about the timestamp of this directory, only that it exists.

Running "`make open`" uses macOS `open` command (similar to `xdg-open` on Linux) to open the PDF in Preview.app. If it is already open, the Preview.app will refresh the existing window.

I have a special "`.PHONY: phony`" rule, that allows me to write:

```
clean: phony
        rm -rf ./output
```

Instead of the usual:

```
.PHONY: clean
clean:
        rm -rf ./output
```

Note that this trick can slow down huge Makefiles.

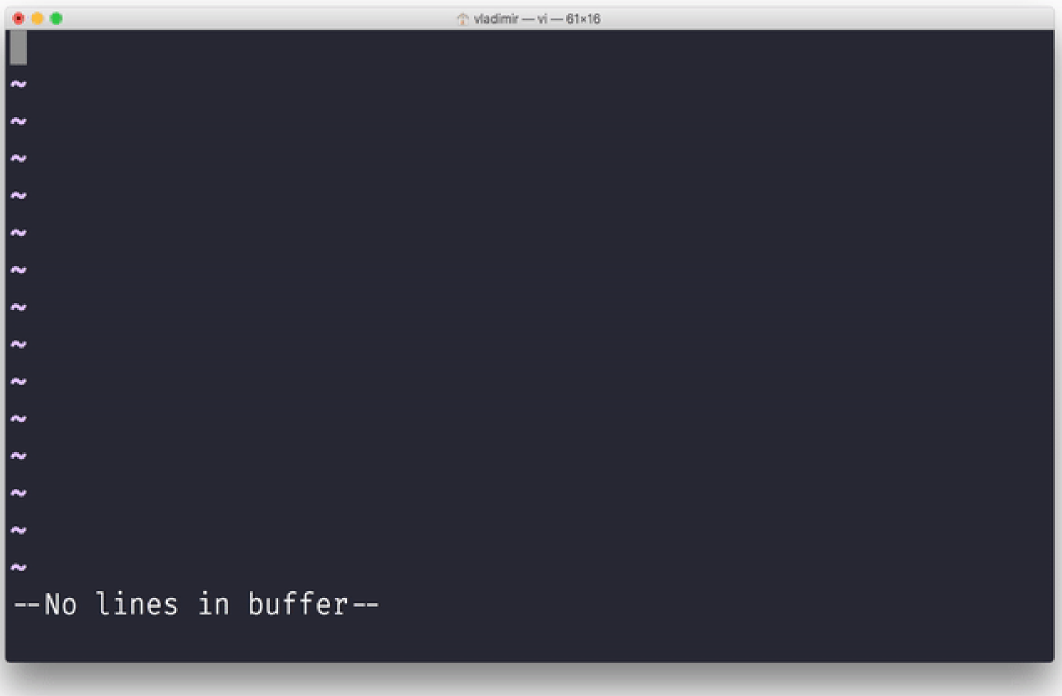
To be honest, this is mostly just me flexing my knowledge of Make, rather than anything useful. A small build script could do just as well.

Vim

I am using the version of Vim that ships with macOS, with minimal tuning. One `~/.vimrc` option worth mentioning here is `virtualedit`:

```
set virtualedit=all
```

It allows to move the cursor past the last character. If you insert a new character there, it is automatically padded with spaces. It is easier to see it than to explain it:



```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
--No lines in buffer--
```

This is very useful for dealing with tables and much more! In fact, I use this option for all my editing in Vim for about a decade now. My first programming environment was Turbo Pascal, and this is exactly how the cursor works there, which I grew accustomed to.

When I open the editor I write `:!make open` to open the preview, and then use `:!!` to repeat that command. In fact, I have a shortcut that maps space bar to save current file and run previous command:

```
map <Space> :w<Enter>:!!<Enter>
```

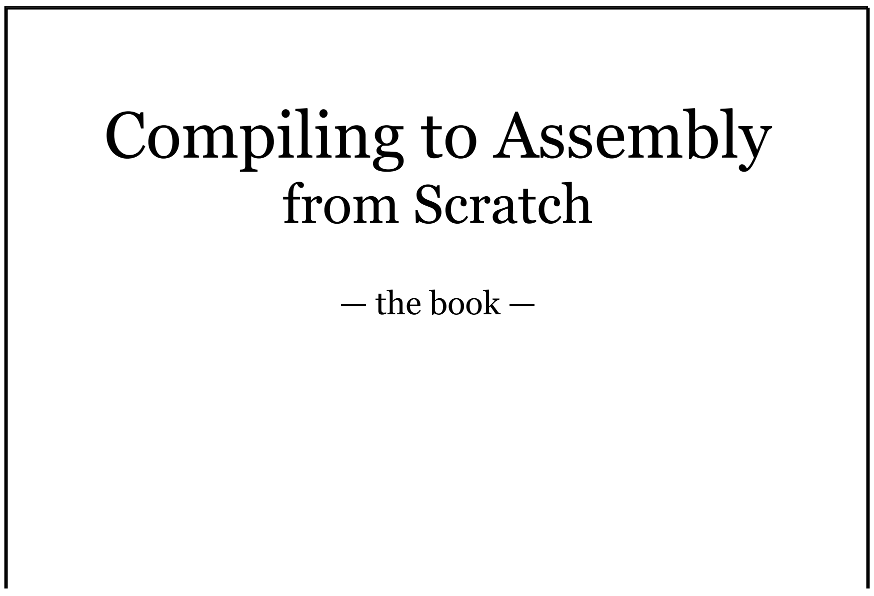
This *save-file-and-repeat-last-command* command is is useful for other things, for example, running test suites.

⁂

And that's about it. When I finish my book, I will write a new blog post about any significant changes to this process. ■

⁂

By now, you have probably heard that I'm writing a book about compilers!



Compiling to Assembly
from Scratch

— the book —

Vladimir Keleshev

TypeScript — ARM — Summer 2020