2020

Max Rottenkolber <max@mr.gy>

Monday, 19 April 2021

What a year! It's been pretty quiet on this blog. However, I've been busy and it can't hurt to recap some of the things I did last year. Last year actually started out with FOSDEM. Big crowd. Huge event. I gave two talks, in person. Can you imagine?

RaptorJIT VM

At FOSDEM 2020 I gave a talk (https://archive.fosdem.org/2020/schedule/event/raptorjit_lua/)
on the work done on RaptorJIT, and the goals and motivations of the project. I also gave *a more general talk at ELS 2020* (https://mr.gy/screen/Later%20Binding%20ELS2020%20talk%20video%202.mp4) on the workings, benefits, and caveats of just-in-time compilation, using LuaJIT as an example. That talk, *Later Binding* (LaterBinding.pdf), is also available in a more detailed paper format.

A milestone was reached on the work of porting LuaJIT's interpreter to C. After fixing a *handful more bugs* (https://github.com/eugeneia/

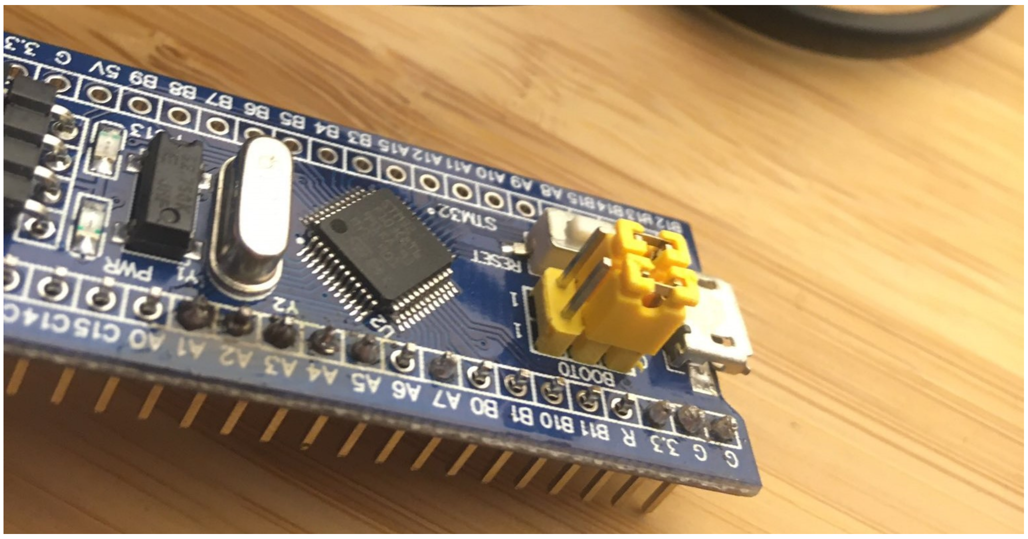
raptorjit/compare/f56b4db...9d61343) I managed to run the Snabb *basic1* benchmark successfully using the new VM, and currently our new interpreter passes almost two thirds of the Snabb test suite. Famous last mile, huh?

Rush

Mostly to learn Rust, I ported *Snabb* (https://github.com/snabbco/snabb) to Rust, and you can relive my experience doing so in a *screencast series I recorded while hacking on Rush* (https://mr.gy/screen/rush). That work turned out commercially viable, and I'm currently working with a startup on network shaping tools based on Rush.

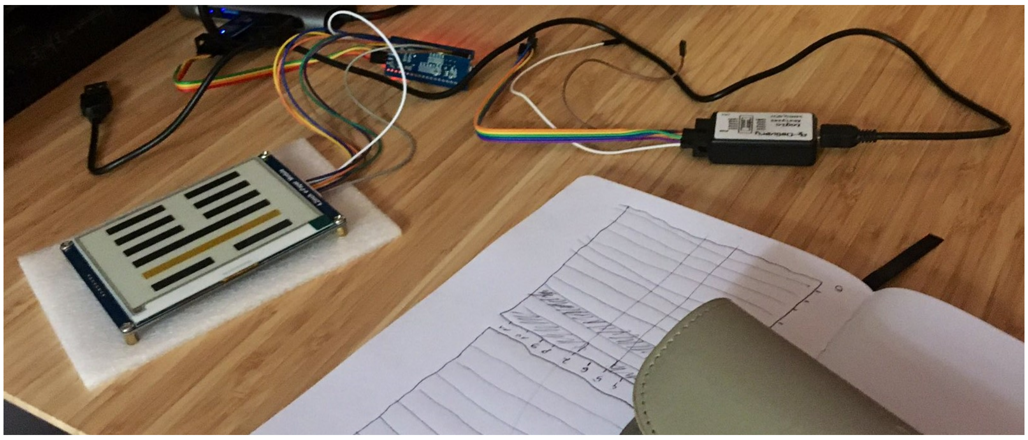
Embedded hacking

I also wanted to give Rust a spin for embedded hacking, and ended up doing a couple projects around the STM32 bluepill and Waveshare e-ink screens. I had to learn some very basic soldering—behold my first ever solder, which I thought I botched but turned out to work fine:



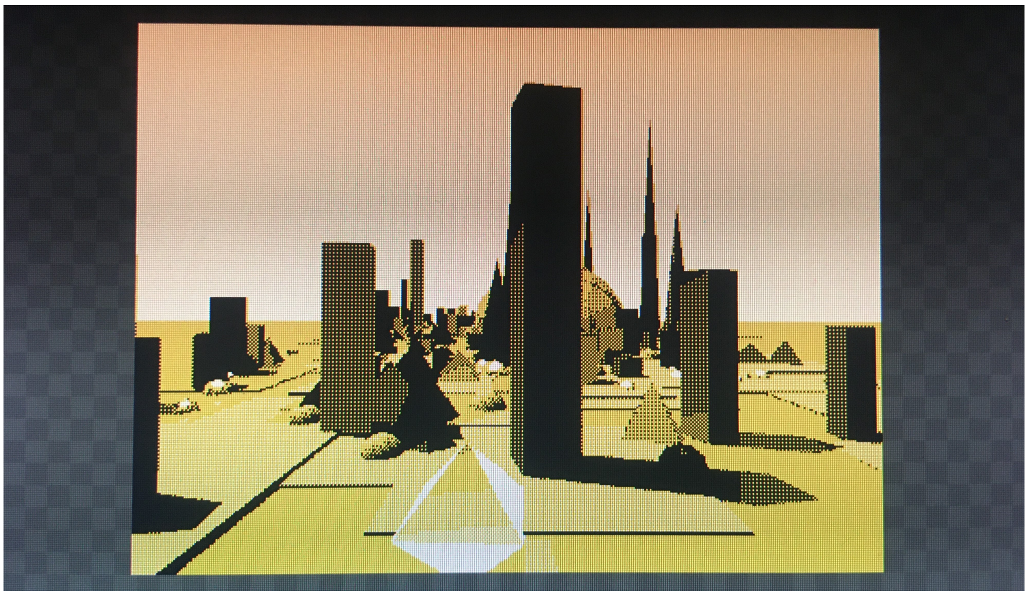
My first solder

. . . and I also used a logic analyzer for the first time. A new form of debugging for me which I enjoyed thoroughly! I ended up doing an *I Ching* (https://en.wikipedia.org/wiki/I_Ching) divination appliance, that collects entropy by tuning two clock crystals on the STM32F103C8 against each other, and renders the resulting hexagram to a e-ink screen.



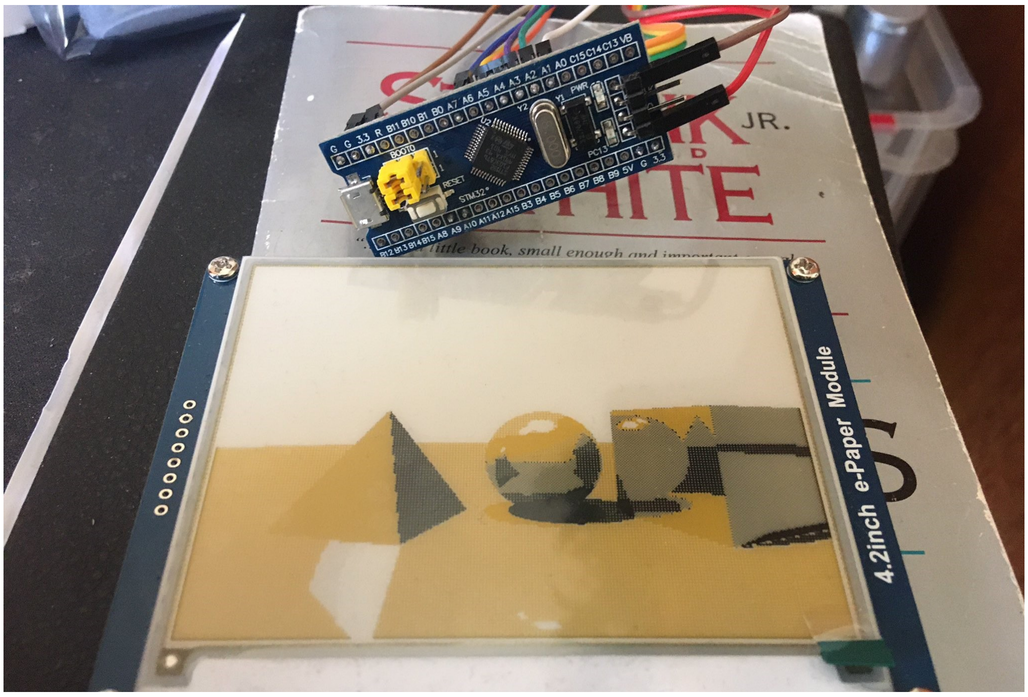
I Ching on E-Paper

Following Dmitry V. Sokolov's *Understandable RayTracing in 256 lines of bare C++* (https://github.com/ssloy/tinyraytracer/wiki/Part-1:-understandable-raytracing) I hacked up a scruffy ray tracer. Mind you the e-ink displays I was working with support just two colors, so I came up with a dithering based approach.



Attempts at ray tracing

I then ported that ray tracer to the STM32F103C8 board, this time using Mecrisp-Stellaris Forth, because I always wanted to do some Forth hacking. That turned out to be quite the fun exercise given the constraints of this particular micro-controller: 72 MHz core clock frequency, 20 Kbytes of RAM, 128 Kbytes of ROM, and no *FPU* (https://en.wikipedia.org/wiki/Floating-point_unit).



STM32 ray tracing

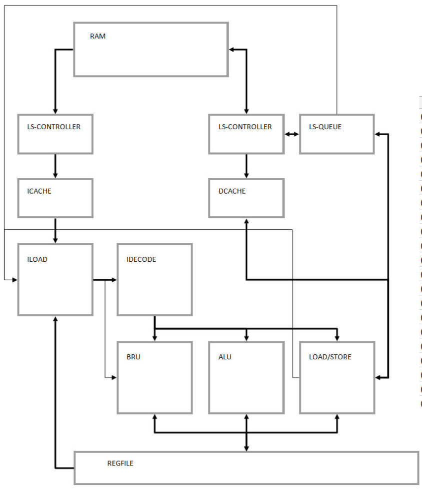
Forth was lots of fun. In some ways it is truly elegant, and fully interactive given the compiler and interpreter run on the freakin' micro-controller, snappy, even at 8 MHz. Below you can marvel at my breakpoint-to-REPL debugger in six lines of Forth. Nevertheless, I feel I saturated my hunger for Forth hacking for now.

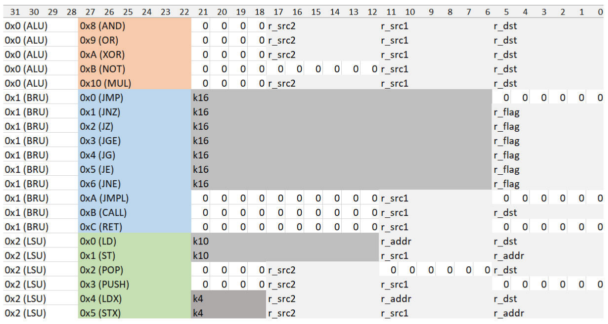
```
false variable break-continue
: continue true break-continue ! ;
: break
  cr ." Breakpoint hit"
  cr begin cr query interpret ." ok." break-continue @ until
  cr false break-continue ! ;
```

Forth breakpoint-to-REPL debugger

Designing a CPU, and an ISA, and an assembler, and a compiler!

Continuing the hardware theme, I decided I wanted to learn Verilog and what better way to learn than to design and implement your own tiny CPU core and ISA?



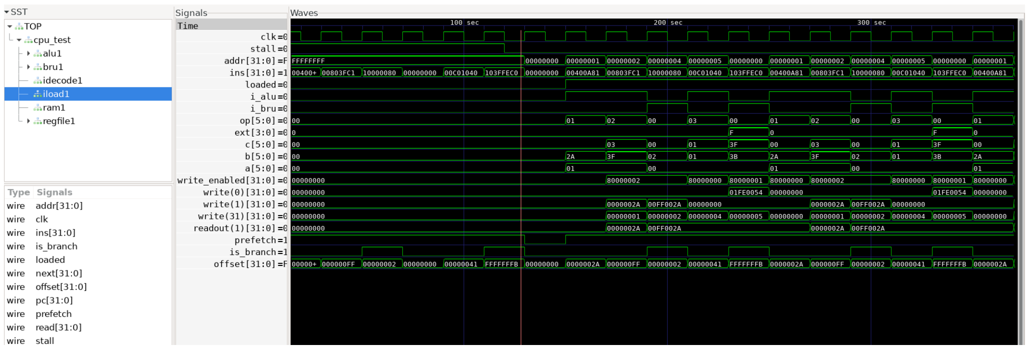


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0 (ALU)				0x8 (AND)						0	0	0	0	r_src2						r_src1						r_dst					
0x0 (ALU)				0x9 (OR)						0	0	0	0	r_src2						r_src1						r_dst					
0x0 (ALU)				0xA (XOR)						0	0	0	0	r_src2						r_src1						r_dst					
0x0 (ALU)				0xB (NOT)						0	0	0	0	0	0	0	0	0	0	r_src1						r_dst					
0x0 (ALU)				0x10 (MUL)						0	0	0	0	r_src2						r_src1						r_dst					
0x1 (BRU)				0x0 (JMP)						k16																0	0	0	0	0	0
0x1 (BRU)				0x1 (JNZ)						k16																r_flag					
0x1 (BRU)				0x2 (JZ)						k16																r_flag					
0x1 (BRU)				0x3 (JGE)						k16																r_flag					
0x1 (BRU)				0x4 (JG)						k16																r_flag					
0x1 (BRU)				0x5 (JE)						k16																r_flag					
0x1 (BRU)				0x6 (JNE)						k16																r_flag					
0x1 (BRU)				0xA (JMPL)						0	0	0	0	0	0	0	0	0	0	r_src1						0	0	0	0	0	0
0x1 (BRU)				0xB (CALL)						0	0	0	0	0	0	0	0	0	0	r_src1						r_dst					
0x1 (BRU)				0xC (RET)						0	0	0	0	0	0	0	0	0	0	r_src1						0	0	0	0	0	0
0x2 (LSU)				0x0 (LD)						k10										r_addr						r_dst					
0x2 (LSU)				0x1 (ST)						k10										r_src1						r_addr					
0x2 (LSU)				0x2 (POP)						0	0	0	0	r_src2						0	0	0	0	0	0	r_dst					
0x2 (LSU)				0x3 (PUSH)						0	0	0	0	r_src2						r_src1						0	0	0	0	0	0
0x2 (LSU)				0x4 (LDX)						k4				r_src2						r_addr						r_dst					
0x2 (LSU)				0x5 (STX)						k4				r_src2						r_src1						r_addr					

A fledgeling CPU architecture and ISA

While I think the ISA turned out OK, my first core design isn't very good as could be expected. I'll have to re-architect this one to get it to synthesize on a real FPGA. It does however simulate fine in Verilator already, and that's good enough for quite a test drive.

And its already somewhat fully featured too, at least conceptually. I made a branch predictor, instruction and data caches, an ALU, and a load/store unit.



Debugging my puny branch predictor

To play with my core, I had to write an assembler for my brand new ISA of course, but I didn't stop there. Learning about SSA based compilers was high on my bucket list, so I built my first compiler for a subset of Lua targeting my fledgeling architecture.

Compilers are quite the rabbit hole! I ended up implementing function calls, call inlining, code folding, dead code elimination, structs and arrays, and a simple compile-time type system. And while its still in a very early stage of development, it kind of works nicely? I can compile and assemble non-trivial programs, and correctly run them on my CPU core under Verilator. How cool?

Below is a test program and its compiler output, just to show off. I can't wait to get back to hacking on this next time I can free up some significant time!

```lua
function foo (a, b)
   local sum = 0
   for i = a, b do
      sum = sum + i
   end
   if sum > 100 then
      return bar(sum)
      -- return -sum
   else
      return sum
   end
end

function bar (x) return baz(x) end

function baz (x) return -x  end

function main ()
   return foo(1, 100)
end
```

Non-trivial Lua program

```
k r0 @main
jmpl r0

@main:
k r1 0
k r2 100
k r3 1
k r4 1
L1:
cmps r5 r2 r4
jge r5 >L2
jmp >L3
L2:
add r1 r1 r4
add r4 r4 r3
jmp <L1
L3:
k r4 100
cmps r4 r4 r1
jge r4 >L4
jmp >L5
L4:
mov r0 r1
ret r29
L5:
k r4 0
sub r0 r4 r1
ret r29
```

Compiler output for our non-trivial Lua program