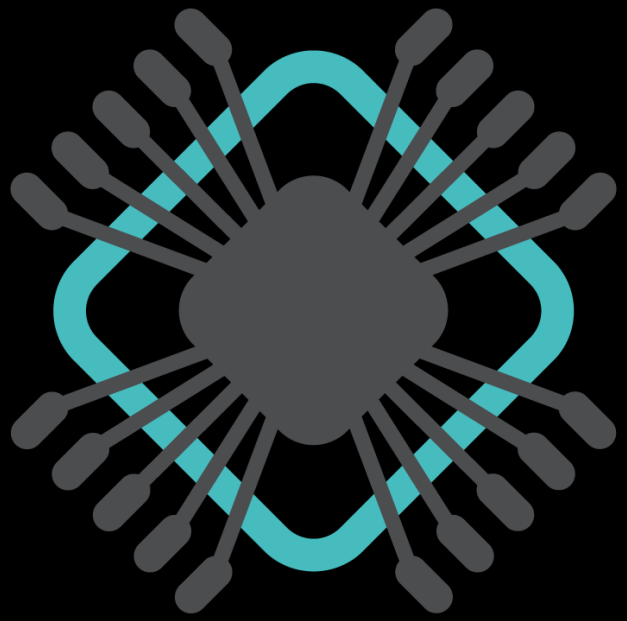


silice

Sylvain Lefebvre

https://github.com/sylefeb/Silice
@sylefeb

hardcoding algorithms

on FPGA hardware

Today

Change log

Design highlights

Change Log

Units and Algorithms in Silice

Units and Algorithms in Silice

main unit ('entry point')

algorithm inside

```
1  unit main(output uint8 leds)
2  {
3    algorithm {
4      uint24 counter = 0;
5      while (1) {
6        leds    = counter[16,8];
7        counter = counter + 1;
8      }
9    }
10 }
```

24 bits counter

forever

8 MSB in leds

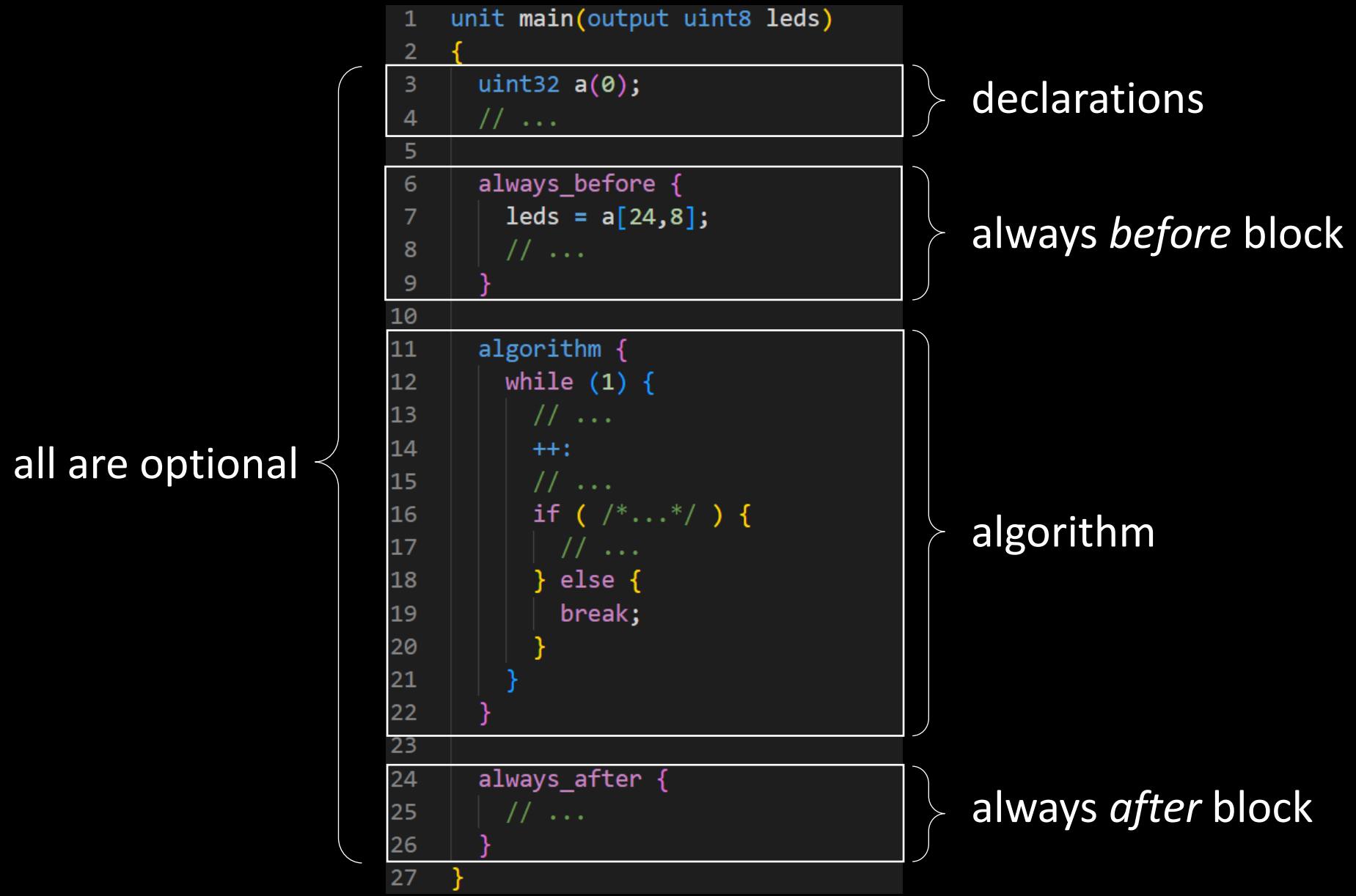
Increment counter

this becomes a circuit

Blinky!

silice

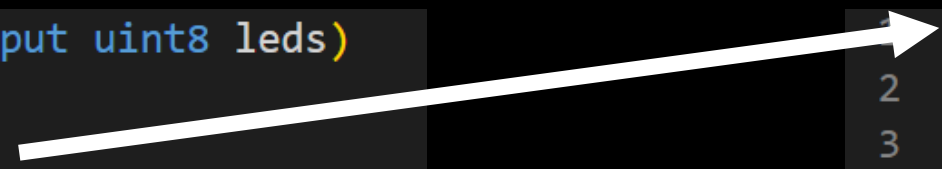
Unit structure



silice

Shortcut: Algorithm

```
1  unit main(output uint8 leds)
2  {
3    algorithm {
4      uint24 counter = 0;
5      while (1) {
6        leds    = counter[16,8];
7        counter = counter + 1;
8      }
9    }
10 }
```



```
1  algorithm main(output uint8 leds)
2  {
3    uint24 counter = 0;
4    while (1) {
5      counter = counter + 1;
6      leds    = counter[16,8];
7    }
8  }
```

A second look

```
unit main(output uint8 leds)
{
  uint32 a(0);
  // ...

  always_before {
    leds = a[24,8];
    // ...
  }

  algorithm {
    while (1) {
      // ...
      ++:
      // ...
      if ( /*...*/ ) {
        // ...
      } else {
        break;
      }
    }
  }

  always_after {
    // ...
  }
}
```

One cycle blocks

- One cycle before/after logic
  - Executes entirely within a single cycle
  - No loops, breaks, etc.

```
1  unit main(output uint8 leds)
2  {
3    uint32 a(0);
4    // ...
5
6    always_before {
7      leds = a[24,8];
8      // ...
9    }
10
11   algorithm {
12     while (1) {
13       // ...
14       ++:
15       // ...
16       if ( /*...*/ ) {
17         // ...
18       } else {
19         break;
20       }
21     }
22   }
23
24   always_after {
25     // ...
26   }
27 }
```

Algorithm

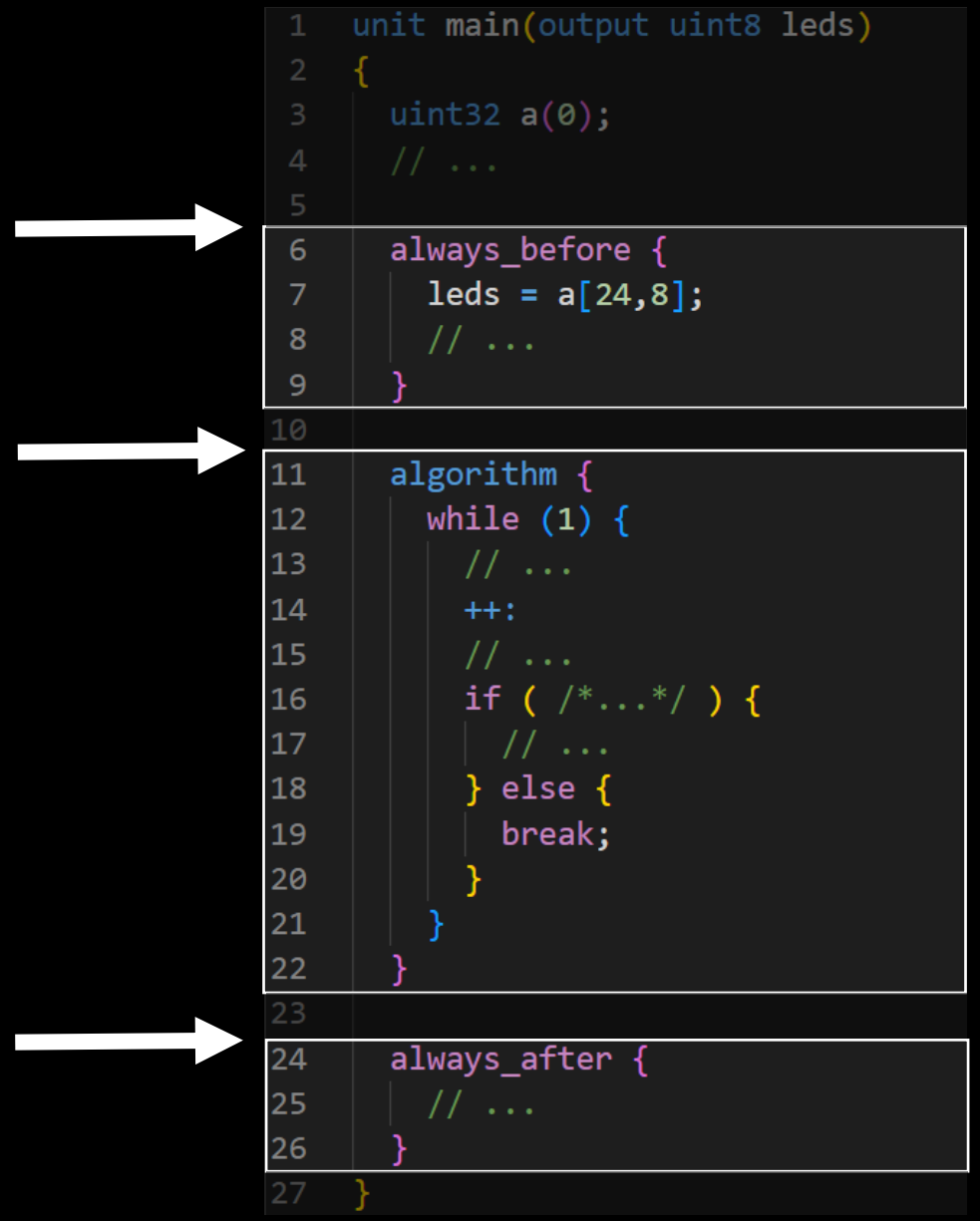
- Executes across many cycles

```
1  unit main(output uint8 leds)
2  {
3    uint32 a(0);
4    // ...
5
6    always_before {
7      leds = a[24,8];
8      // ...
9    }
10
11   algorithm {
12     while (1) {
13       // ...
14       ++:
15       // ...
16       if ( /*...*/ ) {
17         // ...
18       } else {
19         break;
20       }
21     }
22   }
23
24   always_after {
25     // ...
26   }
27 }
```

Best of both worlds

- Always logic
- Imperative algorithms

Seamlessly combined!!



```
unit main(output uint8 leds)
{
  uint32 a(0);
  // ...

  always_before {
    leds = a[24,8];
    // ...
  }

  algorithm {
    while (1) {
      // ...
      ++:
      // ...
      if ( /*...*/ ) {
        // ...
      } else {
        break;
      }
    }
  }

  always_after {
    // ...
  }
}
```

silice

Example

Pipelines

```
{       // stage 0
// ...
} -> { // stage 1
// ...
} -> { // stage 2

}
```

	i	i+1	i+2	i+3	i+4
Stage 0	A	B	C		
Stage 1		A	B	C	
Stage 2			A	B	C

Pipelines

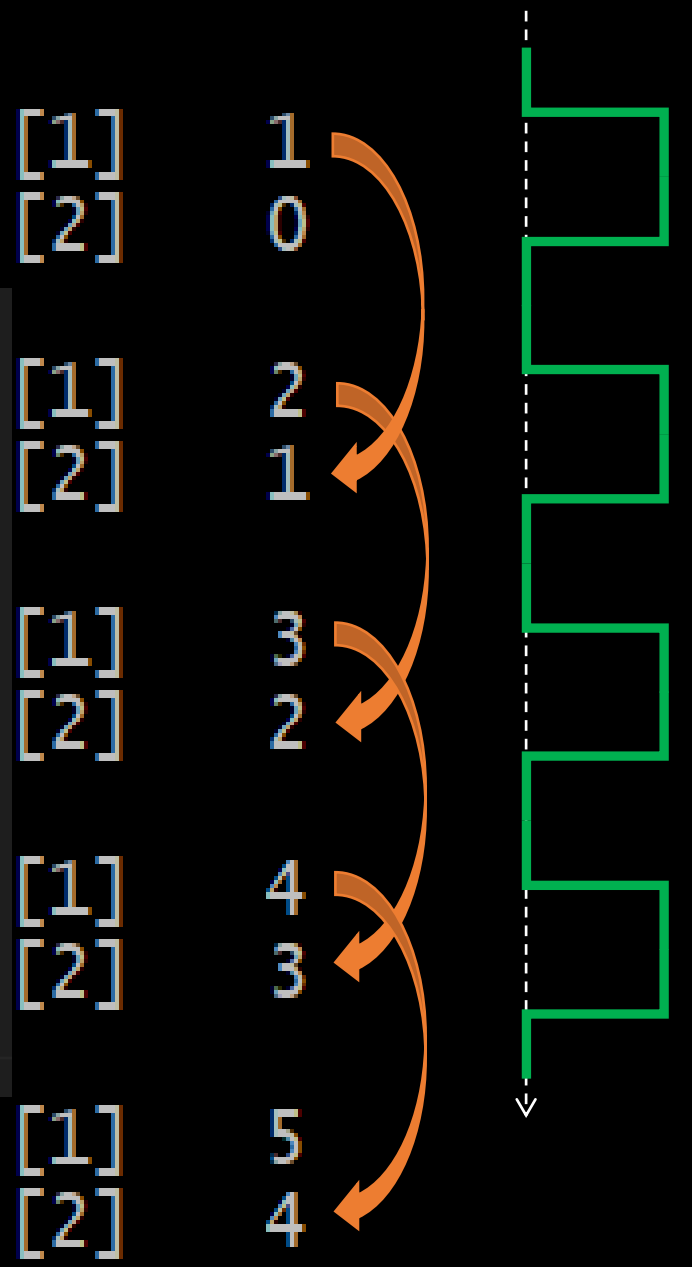
```
uint8 a(0);
{       // stage 0
    a = a + 1;
} -> { // stage 1
    __display("[1] %d",a);
} -> { // stage 2
    __display("[2] %d",a);
}
```

when assigned, a is captured in pipeline

it now *trickles* along the pipeline

Pipelines

```
uint8 a(0);
{       // stage 0
    a = a + 1;
} -> { // stage 1
    __display("[1] %d",a);
} -> { // stage 2
    __display("[2] %d",a);
}
```



silice

Pipelines

a no longer trickles

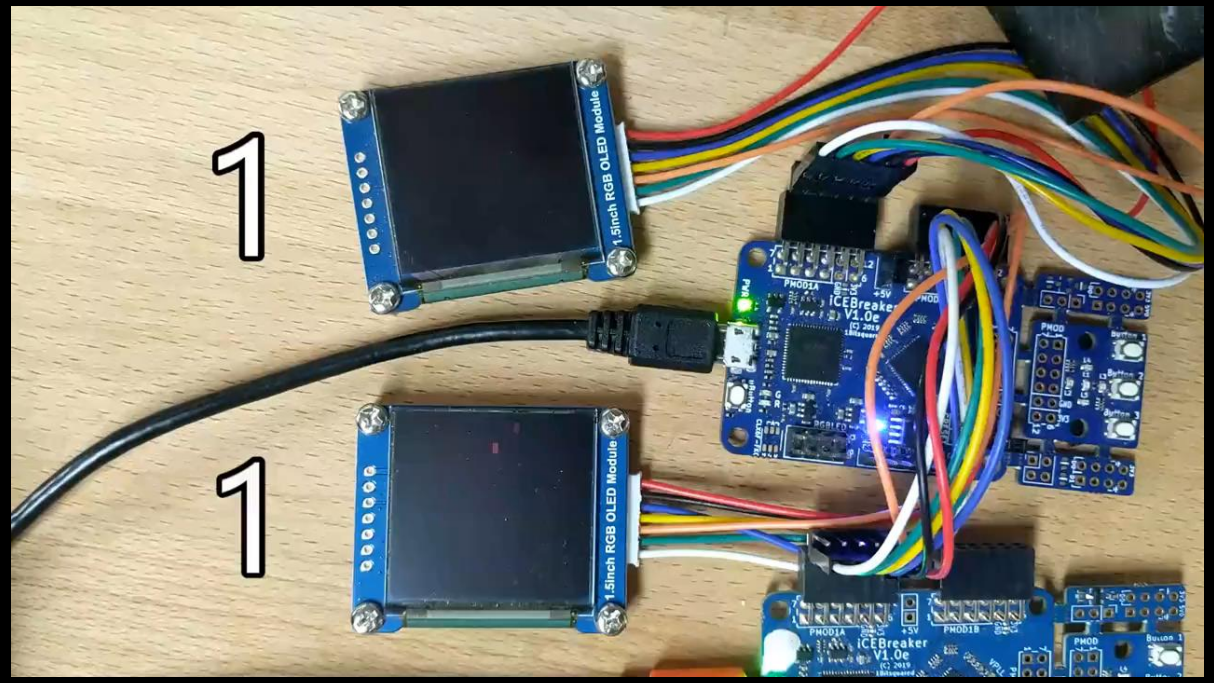
Prevent capture,
as if assigned after

```
uint8 a(0);
{       // stage 0
    a v= a + 1;
} -> { // stage 1
    __display("[1] %d",a);
} -> { // stage 2
    __display("[2] %d",a);
}
```

```
[1]    0
[2]    0

[1]    1
[2]    1

[1]    2
[2]    2

[1]    3
[2]    3

[1]    4
[2]    4
```

Use case: Pipelined CPU

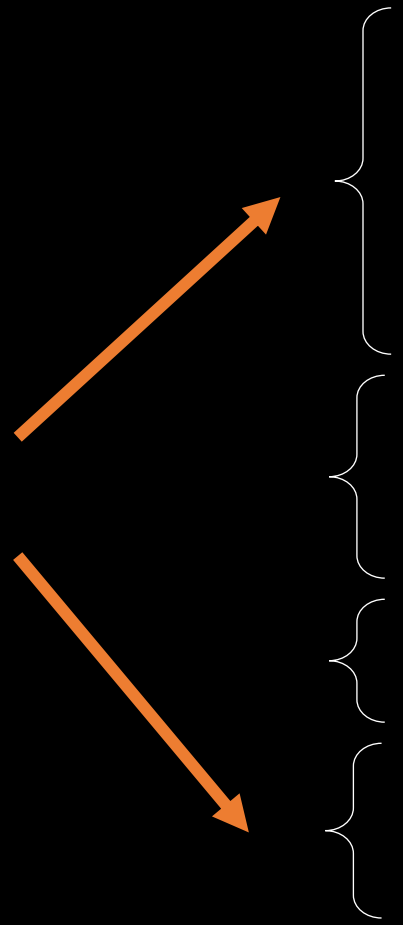
IceV-swirl




```
unit rv32i_cpu_swirl(bram_port mem,bram_port rom)
{
  // register file, uses two BRAMs to fetch two registers at once
  simple_dualport_bram int32 xregsA[32] = {pad(0)};
  simple_dualport_bram int32 xregsB[32] = {pad(0)};
  // decoder + ALU, executes the instruction and tells processor what to do
  decode_and_ALU_swirl exec<reginputs>;
  // ..
  always {
    { // ==== stage 1
      // capture pc, instr in pipeline
      pc       = (exec.working | hold) ? pc    : rom.addr;
      instr    = (exec.working | hold) ? instr : rom.rdata;
      //         ^^^^^^^^^^^^^^^^^^^^^ hold if ALU is busy or hazard (c)
      // insert a bubble on a refetch or ALU busy or reset
      bubble   = refetch | exec.working | reset;
      // setup register read
      xregsA.addr0    = Rtype(instr).rs1;  xregsB.addr0    = Rtype(instr).rs2;
      // fetch next instruction
      rom.addr        = // ...
    } -> { // ==== stage 2
      // give instruction, pc and registers to decoder+ALU
      exec.instr   = instr;
      exec.pc      = pc;
      // data hazards detection
      // ..
    } -> { // ==== stage 3
      // memory address from which to load/store
      mem.addr  = (exec.n >> 2);
      // ...
    } -> { // ==== stage 4
      // register write back
      xregsA.wenable1  = ~no_rd & ~bubble & ~refetch;
      xregsA.addr1     = rd;
      xregsA.wdata1    = // ...
    }
    // set decoder+ALU inputs, depending on data hazards
    exec.xa    = // ..
    exec.xb    = // ..
  }
```

Use case: Pipelined CPU

Four stages



```
unit rv32i_cpu_swirl(bram_port mem,bram_port rom)
{
  // register file, uses two BRAMs to fetch two registers at once
  simple_dualport_bram int32 xregsA[32] = {pad(0)};
  simple_dualport_bram int32 xregsB[32] = {pad(0)};
  // decoder + ALU, executes the instruction and tells processor what to do
  decode_and_ALU_swirl exec<reginputs>;
  // ..
  always {
    { // ==== stage 1
      // capture pc, instr in pipeline
      pc       = (exec.working | hold) ? pc    : rom.addr;
      instr    = (exec.working | hold) ? instr : rom.rdata;
      //         ^^^^^^^^^^^^^^^^^^^^^ hold if ALU is busy or hazard (c)
      // insert a bubble on a refetch or ALU busy or reset
      bubble   = refetch | exec.working | reset;
      // setup register read
      xregsA.addr0    = Rtype(instr).rs1;  xregsB.addr0    = Rtype(instr).rs2;
      // fetch next instruction
      rom.addr        = // ...
    } -> { // ==== stage 2
      // give instruction, pc and registers to decoder+ALU
      exec.instr   = instr;
      exec.pc      = pc;
      // data hazards detection
      // ..
    } -> { // ==== stage 3
      // memory address from which to load/store
      mem.addr  = (exec.n >> 2);
      // ...
    } -> { // ==== stage 4
      // register write back
      xregsA.wenable1  = ~no_rd & ~bubble & ~refetch;
      xregsA.addr1     = rd;
      xregsA.wdata1    = // ...
    }
    // set decoder+ALU inputs, depending on data hazards
    exec.xa    = // ..
    exec.xb    = // ..
  }
```

Inouts

Change Log

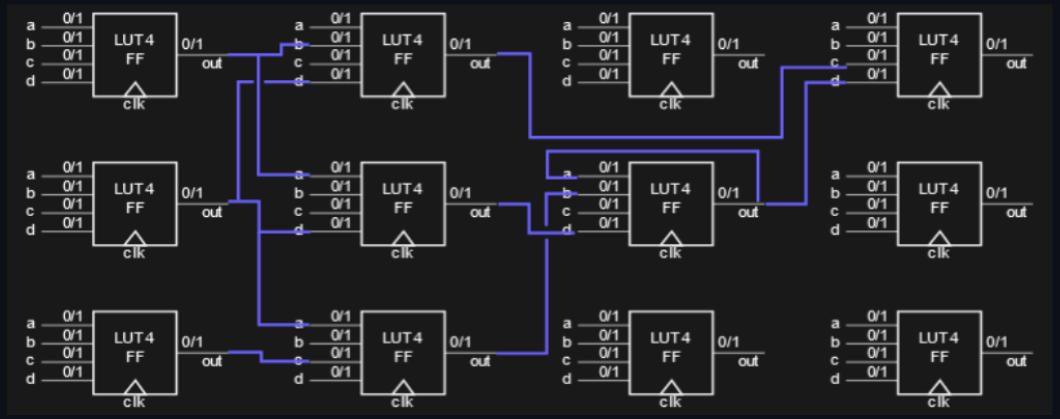
- A lot more features:
  - Genericity! (auto, sameas, widthof)
  - Instantiation time pre-processing
    - ➔ Preprocessor generates the code in context of the instance
  - Python API

Brand new tutorial

https://github.com/sylefeb/Silice/tree/master/learn-silice

FPGA hardware design 101

The most important thing to remember, as you enter these tutorials, is that the code we write describes a *hardware design*. This is not code that will be *executed*. This is code that gets turned into an actual circuitry, composed of FPGA blocks configured and connected together to produce the circuit we described. This is illustrated below, where each block is a configurable FPGA lookup table (LUT) and blue connections are configurable routes between these blocks. In this example all blocks are *synchronous*: they only update their outputs from the inputs when the clock raises. Such blocks are implementing so-called *flip-flops*.



> If you want to more explanations about this, please refer my talk about the Doomchip-onice (slide,video). Also checkout my FPGA gate simulator page.

From the illustration we can guess a few things:

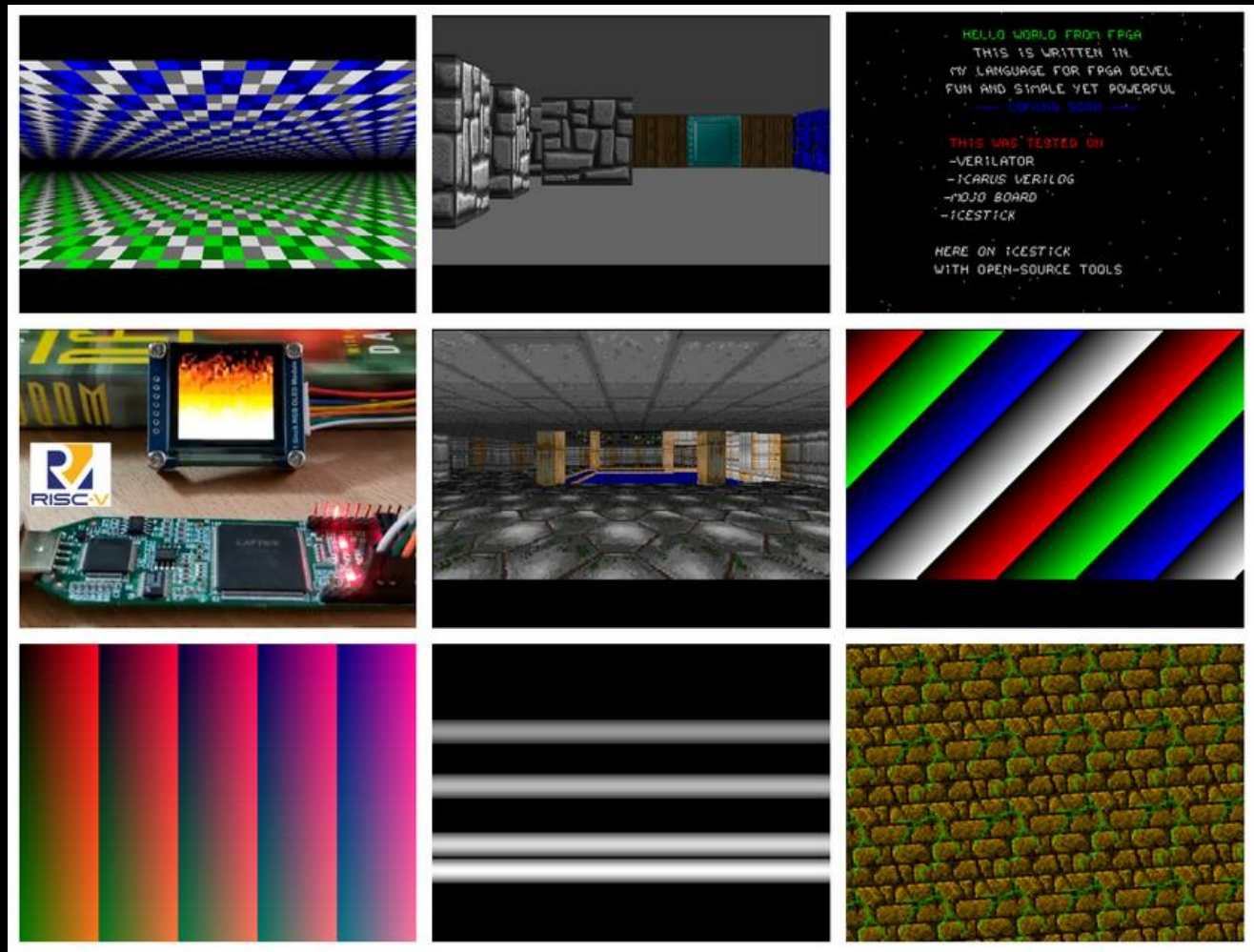
- The circuit uses FPGA blocks (or LUTs for Lookup Up Tables), so there is a notion of how big it will become. Complex designs may not fit on smaller FPGAs. This *resource usage* is fun and often important to optimize, an activity often called *LUT golfing*.

silice

It’s also about the designs

- 30+ example projects
  - Most have detailed walkthroughs
- A few highlights …

https://github.com/sylefeb/Silice/

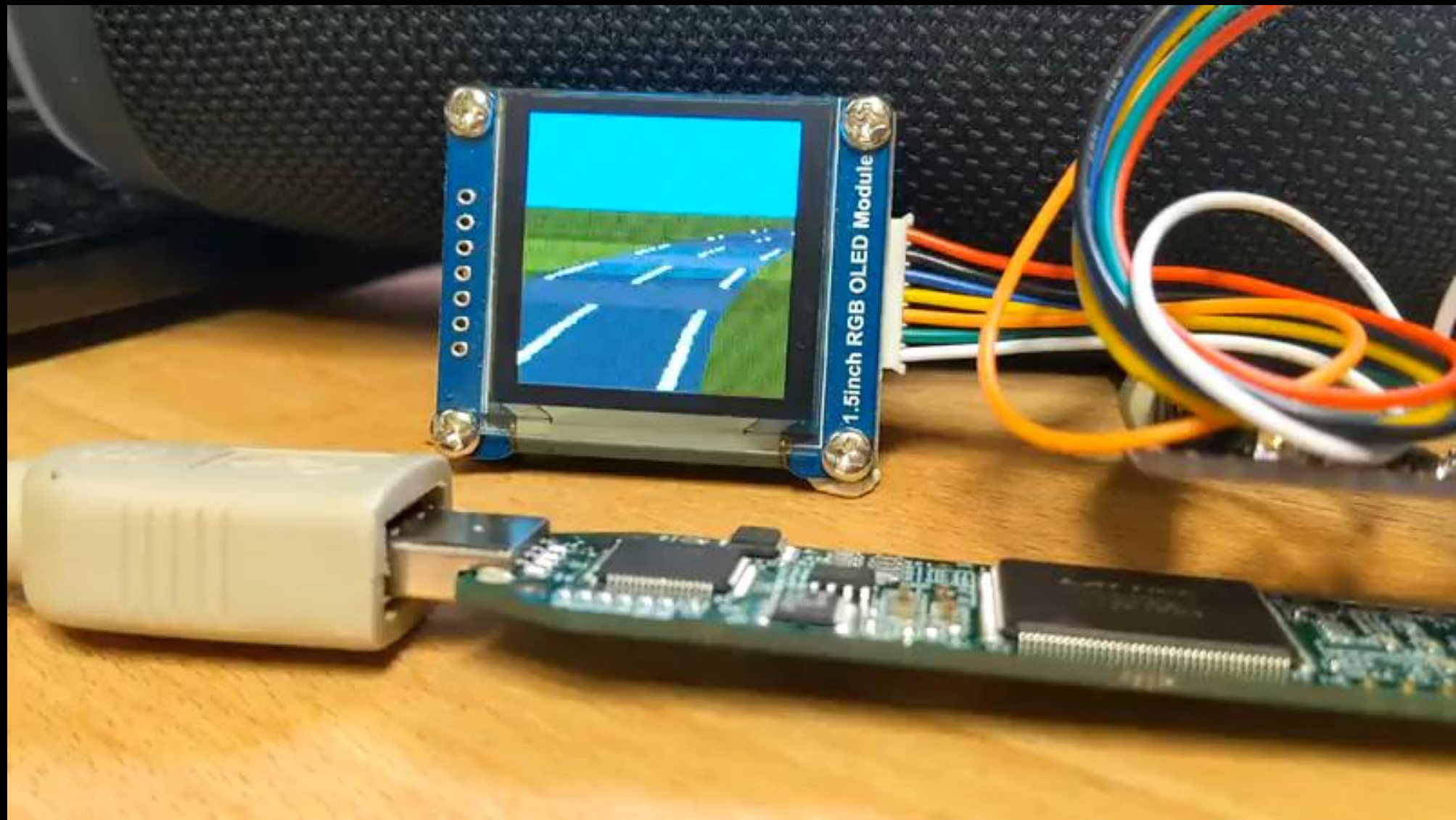


silice

Driving a Neopixel

silice

Driving a SPIscreen



silice

The ice-v family

RV32I(M) tiny processors

- dual — Tiny dual core!
- fermata — Dual memory interface
- conveyor — Pipelined, basic but small
- swirl — Pipelined, faster

120 lines, dual core

silice

PAWSv2 by @rob-ng15

https://github.com/rob-ng15/PAWSv2

- Very complete design for the ULX3S
  - RV32IMAFC dual thread RISC-V
  - FPU, GPU, audio, SDcard, UART, PS/2 keyboard
- Many games!

silice

The DMC-1 GPU

- A GPU written in Silice for 1990s games
  https://github.com/sylefeb/tinygpus (work in progress)

- See the dedicated talk:
  https://www.youtube.com/watch?v=2ZAIIDXoBis

silice

Acknowledgments

Silice is only possible thanks to the community!

- Silice contributors:

  rob-ng15, trabucayre, emard, Mesabloo, diadatp,
  tommythorn, osnr, cbalint13, ttricard, umarcor, juanmard

- Yosys / NextPNR / Edalize / Verilator / Icarus / OpenFPGALoader …
- ULX3S (Radiona) / IceBreaker (1bitsquared) / ECPIX5 (LambdaConcept) …

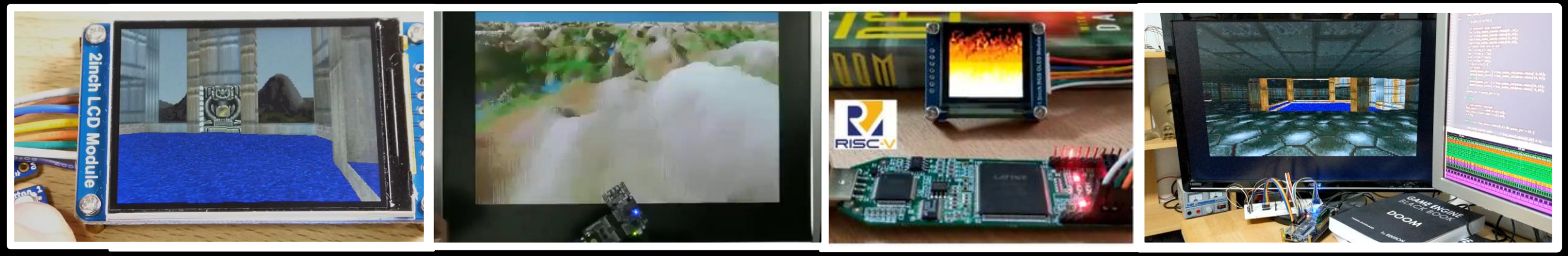
  (many, many other great projects!)

- Easy interop with Verilog, Python (Litex, Migen, Amaranth …), …

Thank you!

Getting started: https://github.com/sylefeb/Silice

@sylefeb



silice