silice

hardcoding algorithms

on FPGA hardware

Who am I?

@sylefeb sylvain.lefebvre@inria.fr

- Researcher in Computer Graphics and Additive Manufacturing
- Image and geometry processing
- GPU expertise (20+ years)
- <u>Disclaimer:</u> Quite new to FPGAs! (2+ years)

silice

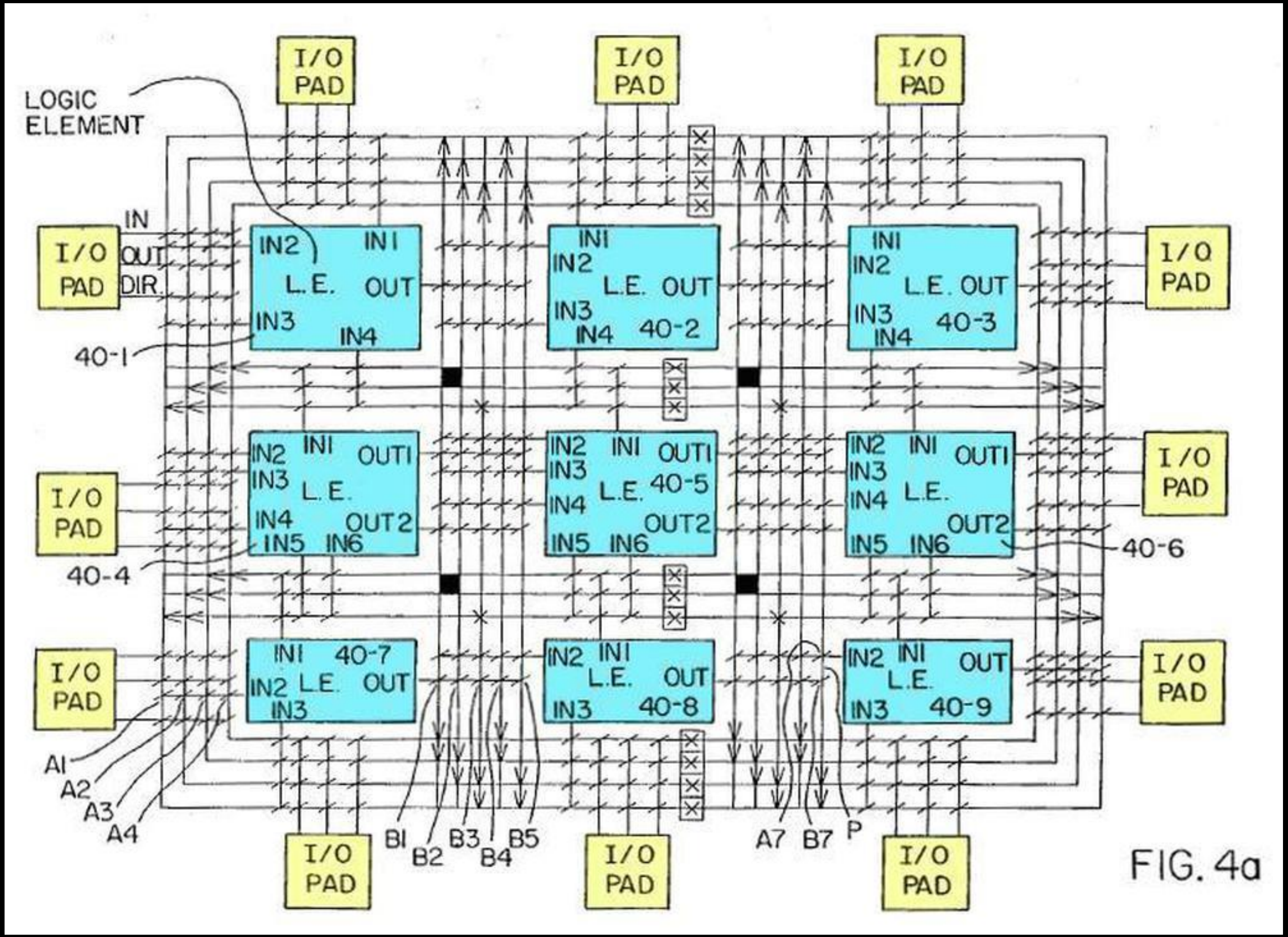
Today

Silice goals

A ‘flavor’ of Silice

One project walkthrough

FPGA 101

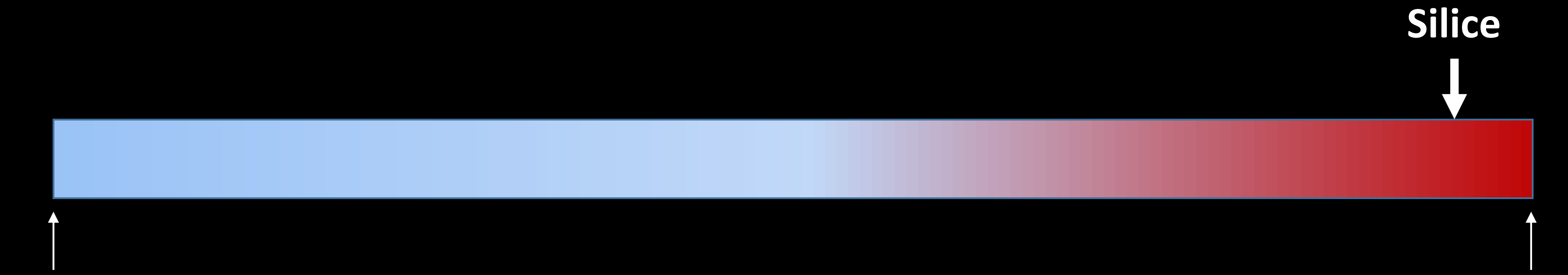


https://patents.google.com/patent/US4870302A
http://www.righto.com/2020/09/reverse-engineering-first-fpga-chip.html

silice

How to design for FPGAs?

- Using a Hardware Description Language (HDL)
  - Verilog / VHDL
- Using a High Level Synthesis language (HLS)



Verilog
Full control, direct mapping to hardware
Development + testing time is significant

Promises ‘C to hardware’
Not directly manipulating the hardware (abstractions)
Shorter development cycles

Silice goals

- “Quality-of-life” improvements
  - ➔ Groups, generic interfaces
  - ➔ Auto flip-flops
  - ➔ Powerful pre-processor (Lua)

- Make prototyping algorithms easier
  - ➔ Quickly get to POC, refine into efficient design
  - ➔ Use typical control flow (while/break/calls/subroutines)

- Without giving up direct hardware mapping
  - ➔ Cycles and clocks are exposed
  - ➔ Control over how inputs and outputs are registered
  - ➔ 1 variable = 0 or 1 flip-flop (auto-culling)
  - ➔ Typical hardware constructs (always, bindings, …)

Silice environment

- Comes with a full, extensible environment
  - IceStick, IceBreaker, ULX3S, ECPIX-5, De10-nano, MojoV3, …
  - Simulation: Icarus, Verilator (with SDRAM and VGA)
    (Powered by Edalize!)

- Comes with ready to use components:
  - VGA / HDMI, SDRAM controllers, arbiters, UART, OLED

- Lets’ get started!

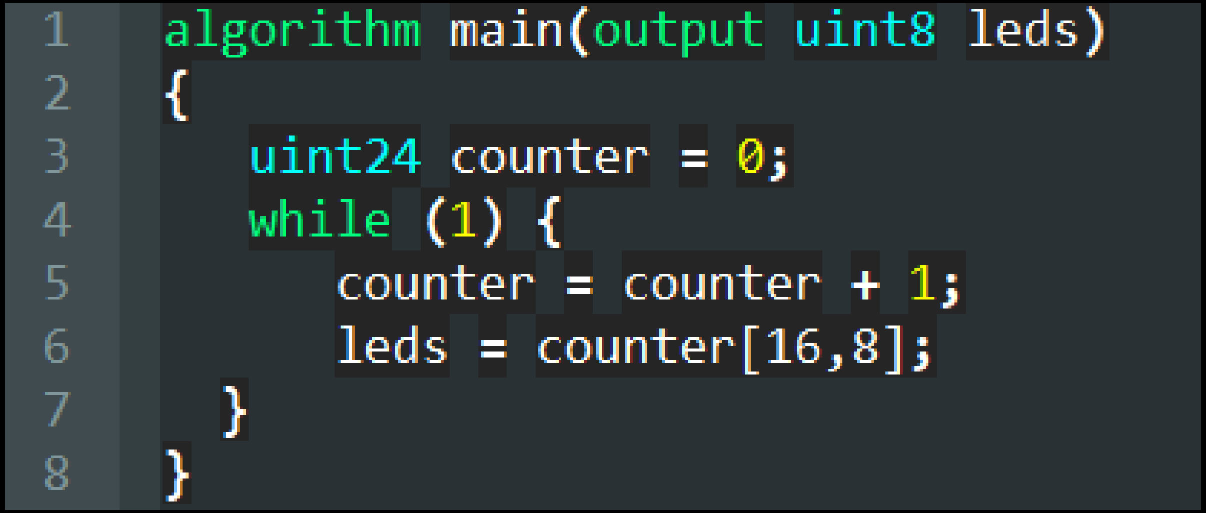
silice

Algorithms in Silice

Algorithms in Silice

main is your design 'entry point'

main inputs/outputs are typically IO pins



```
algorithm main(output uint8 leds)
{
   uint24 counter = 0;
   while (1) {
      counter = counter + 1;
      leds = counter[16,8];
   }
}
```

this becomes a circuit

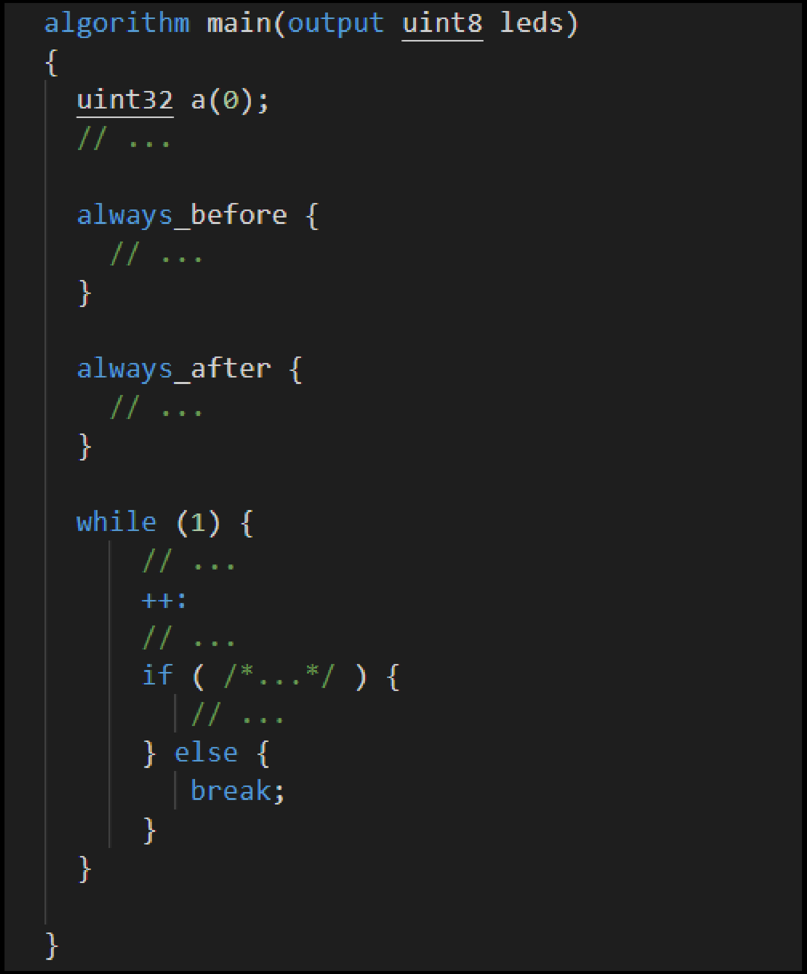
silice

Software stack

make icestick

$$\begin{tikzcd}
\text{Silice} \arrow[d, "\textit{Verilog}"] \\
\text{Yosys} \arrow[d] \\
\text{NextPNR} \arrow[d] \\
\text{FPGA}
\end{tikzcd}$$

Algorithm structure



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

  always_before {
    // ...
  }

  always_after {
    // ...
  }

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

}
```

declarations

always *before* block

always *after* block

state machine

all are optional

Declarations: Variables

- Reset state initialization

SintN name = C | *uninitialized* ;

Signedness
<u>int</u> or <u>uint</u>

Bitwidth

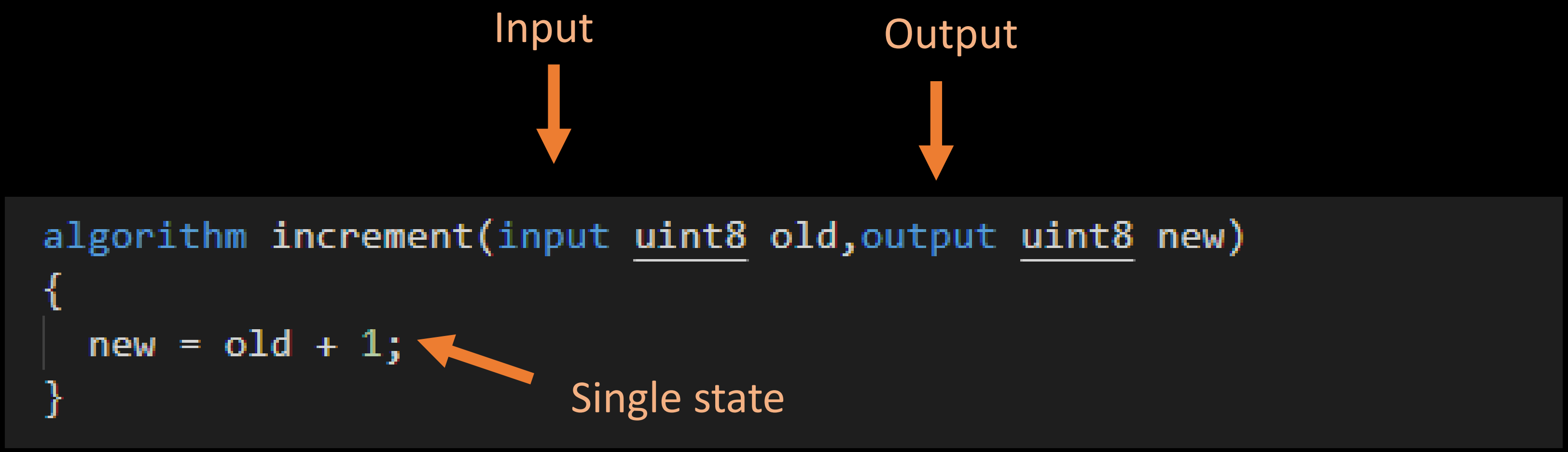
Constant or keyword *uninitialized*

Examples:

```
uint1  b          = 0;
int7    val        = - 335;
int16   xcoord = 5;
uint19 c           = uninitialized;
uint6   mo(6b001000);
```

- Power-up initialization: SintN name(C);

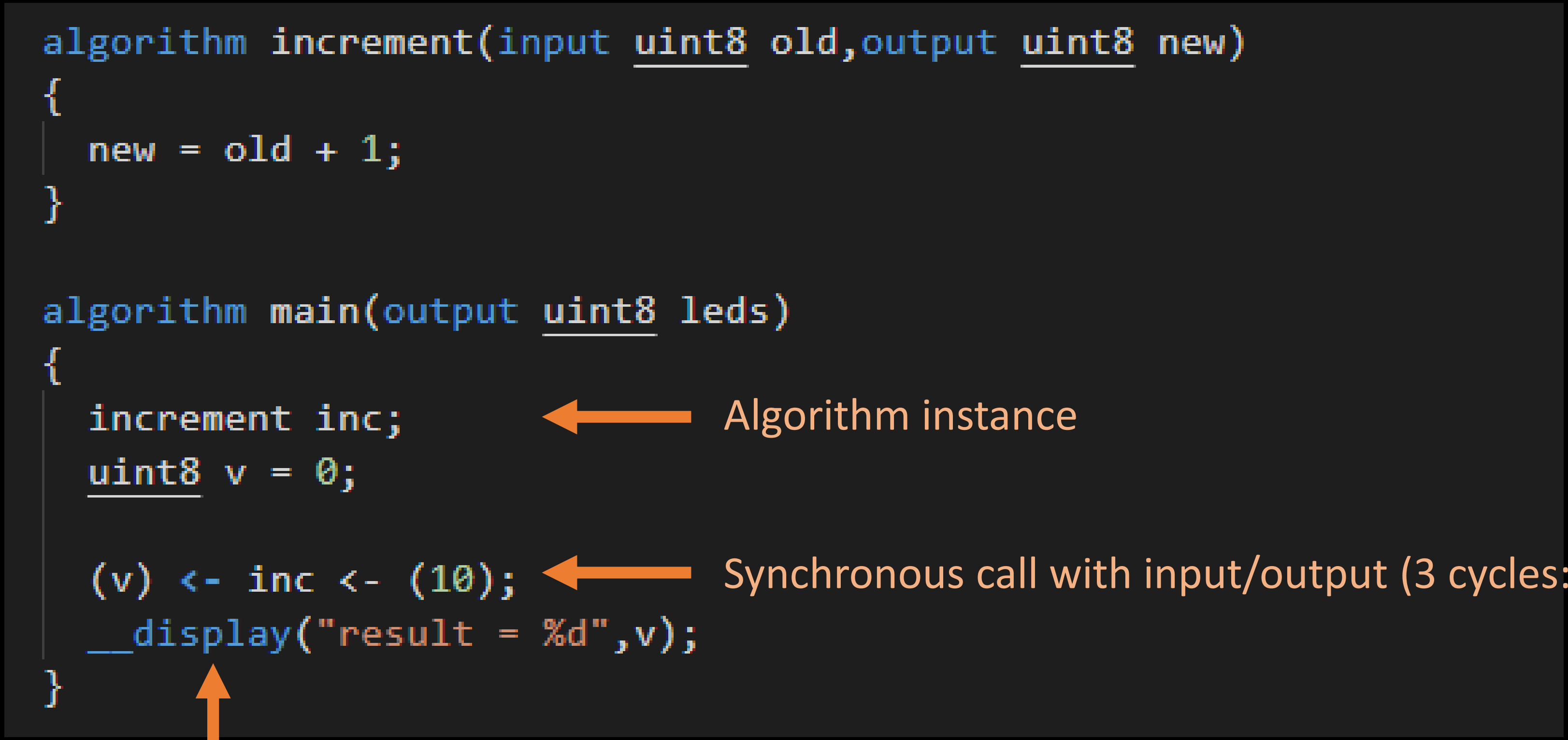
Algorithms



```
algorithm increment(input uint8 old,output uint8 new)
{
  new = old + 1;
}
```

silice

Algorithms: call



```
algorithm increment(input uint8 old,output uint8 new)
{
  new = old + 1;
}

algorithm main(output uint8 leds)
{
  increment inc;
  uint8 v = 0;

  (v) <- inc <- (10);
  __display("result = %d",v);
}
```

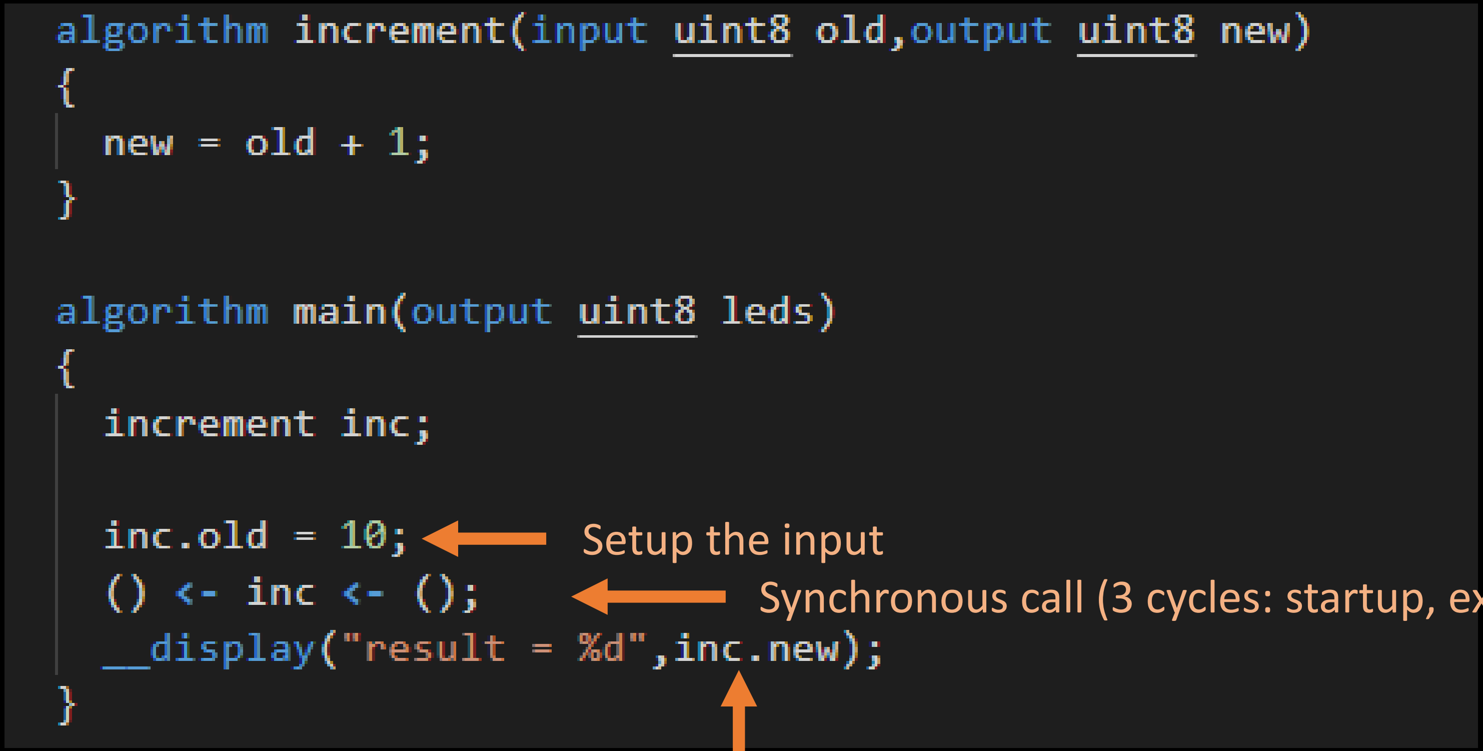
Algorithm instance

Synchronous call with input/output (3 cycles: startup, exec, wait)

Display result (simulation, e.g. *make verilator*)

silice

Algorithms: dot syntax



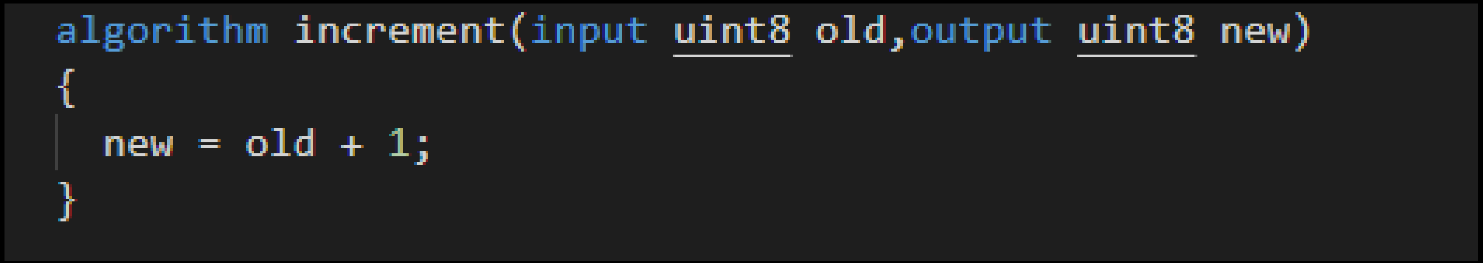
```
algorithm increment(input uint8 old,output uint8 new)
{
  new = old + 1;
}

algorithm main(output uint8 leds)
{
  increment inc;

  inc.old = 10;
  () <- inc <- ();
  __display("result = %d",inc.new);
}
```

Setup the input

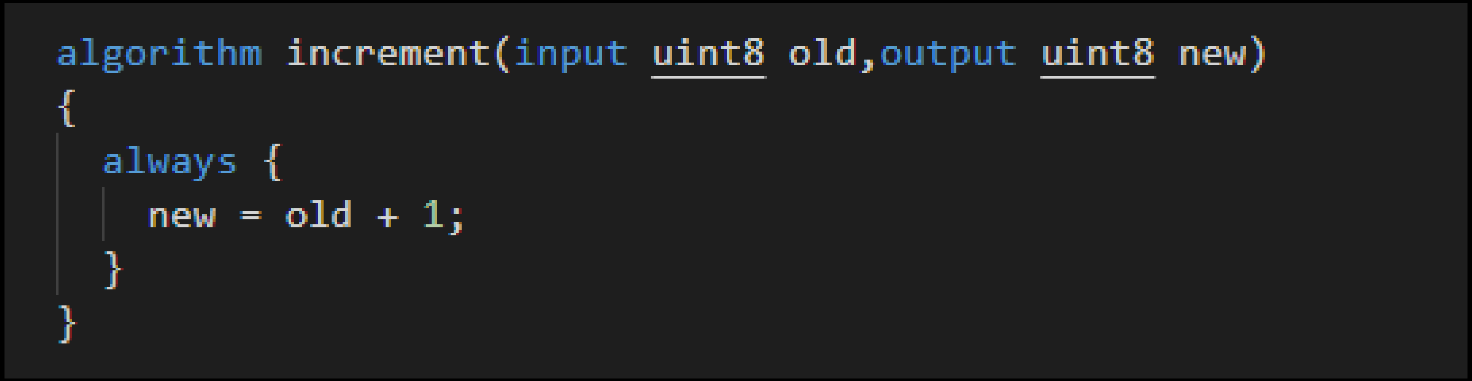
Synchronous call (3 cycles: startup, exec, wait)

Read the output

Algorithms: dot syntax

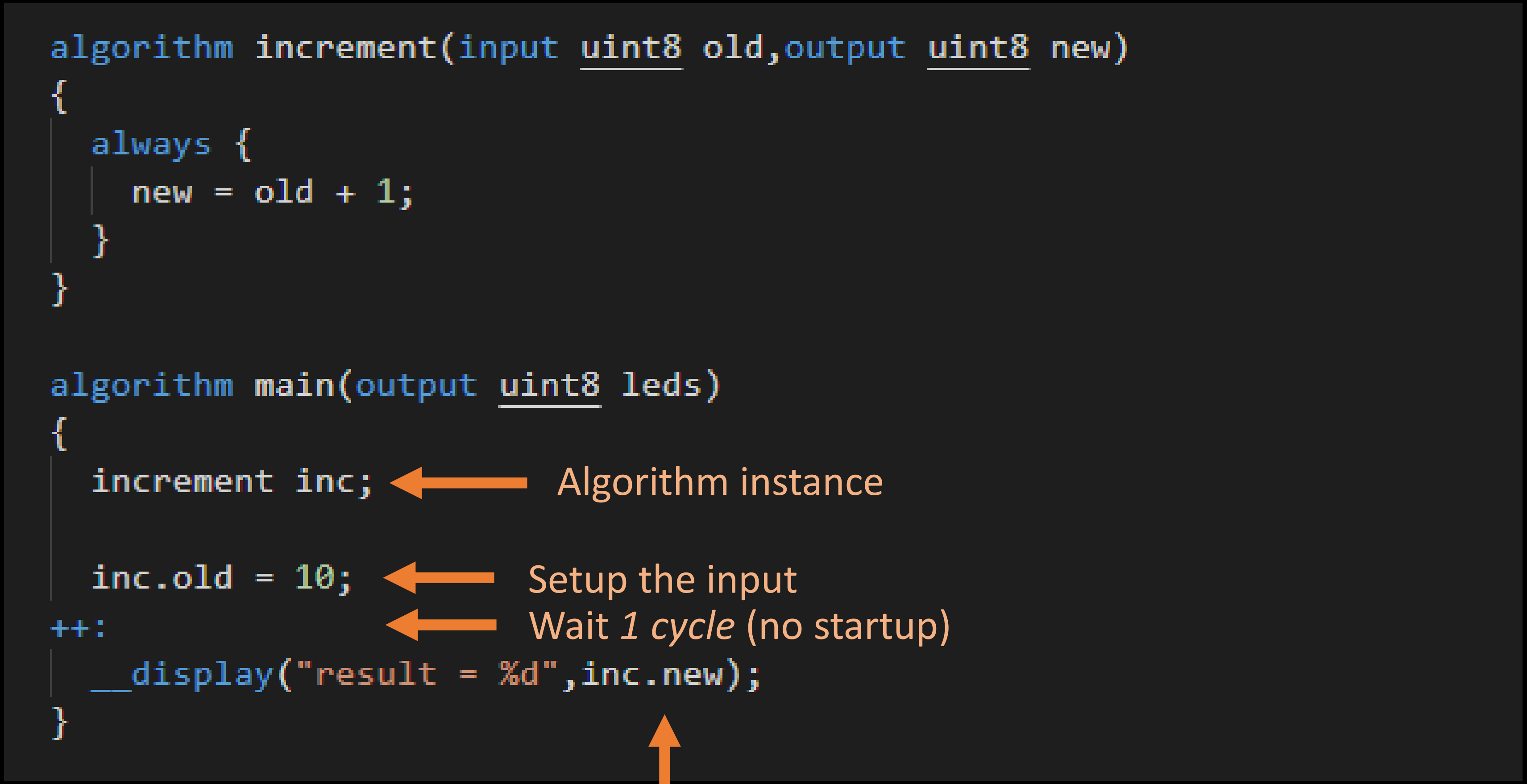


```
algorithm increment(input uint8 old,output uint8 new)
{
  new = old + 1;
}
```



```
algorithm increment(input uint8 old,output uint8 new)
{
  always {
    new = old + 1;
  }
}
```

Algorithms: always and dot syntax



```
algorithm increment(input uint8 old,output uint8 new)
{
  always {
    new = old + 1;
  }
}

algorithm main(output uint8 leds)
{
  increment inc;

  inc.old = 10;
++:
  __display("result = %d",inc.new);
}
```

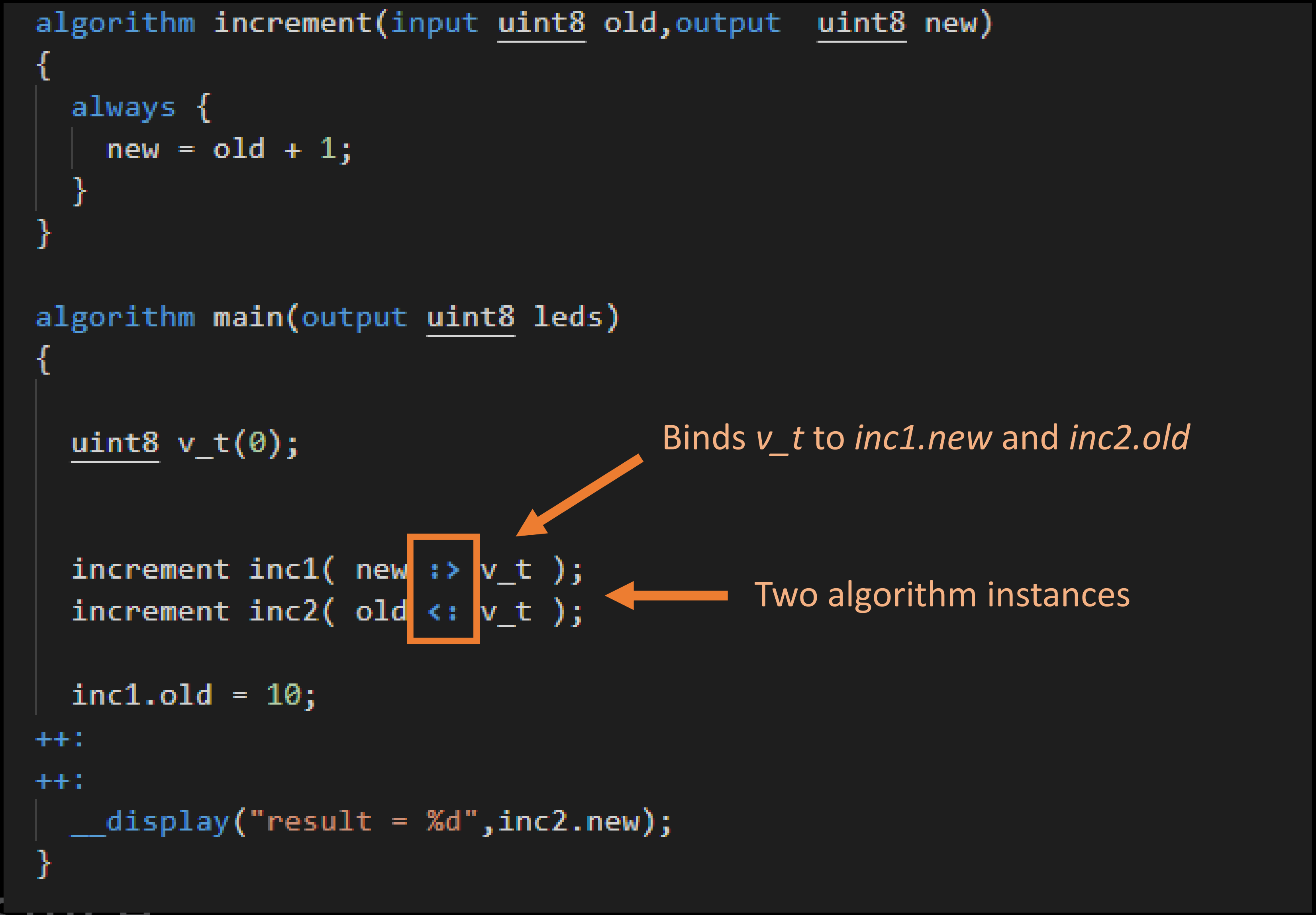
Algorithm instance

Setup the input

Wait *1 cycle* (no startup)

Result is available

Algorithms: bindings



```
algorithm increment(input uint8 old,output  uint8 new)
{
  always {
    new = old + 1;
  }
}

algorithm main(output uint8 leds)
{

  uint8 v_t(0);


  increment inc1( new :> v_t );
  increment inc2( old <: v_t );

  inc1.old = 10;
++:
++:
  __display("result = %d",inc2.new);
}
```

Binds *v_t* to *inc1.new* and *inc2.old*

Two algorithm instances

Groups and interfaces

silice

Groups and interfaces

- Groups: groups of variables

Definition

```
group sdram_r16w16_io
{
  uint26  addr      = 0,
  uint1   rw        = 0,
  uint16  data_in   = 0,
  uint8   wmask     = 0,
  uint1   in_valid  = 0,
  uint16  data_out  = uninitialized,
  uint1   done      = 0
}
```

Usage

```
sdram_r16w16_io sio;

sdram_controller_autoprecharge_r16_w16 sdram(
  sd        <:> sio,
  // ...
);

sio.rw       = 1;
sio.addr     = iter;
sio.data_in  = 64h8877665544332211;
sio.wmask    = 8b10101010;
sio.in_valid = 1;
while ( ! sio.done ) { }
```

Groups and interfaces

```
group sdram_r16w16_io
{
  uint26  addr       = 0,
  uint1   rw         = 0,
  uint16  data_in    = 0,
  uint8   wmask      = 0,
  uint1   in_valid   = 0,
  uint16  data_out   = uninitialized,
  uint1   done       = 0
}
```

- Interfaces: groups of inputs / outputs

```
// interface for user
interface sdram_user {
  output  addr,
  output  rw,
  output  data_in,
  output  in_valid,
  output  wmask,
  input   data_out,
  input   done,
}
```

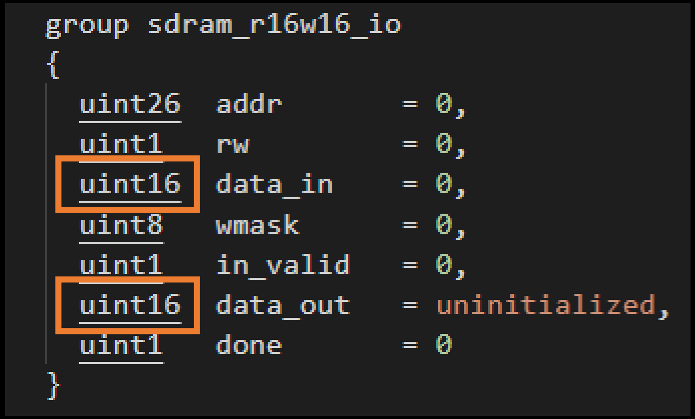
```
// interface for provider
interface sdram_provider {
  input   addr,
  input   rw,
  input   data_in,
  input   in_valid,
  input   wmask,
  output  data_out,
  output  done
}
```

```
algorithm sdram_controller_autoprecharge_r16_w16(
        // interface
        sdram_provider sd,
        // ...
)
```

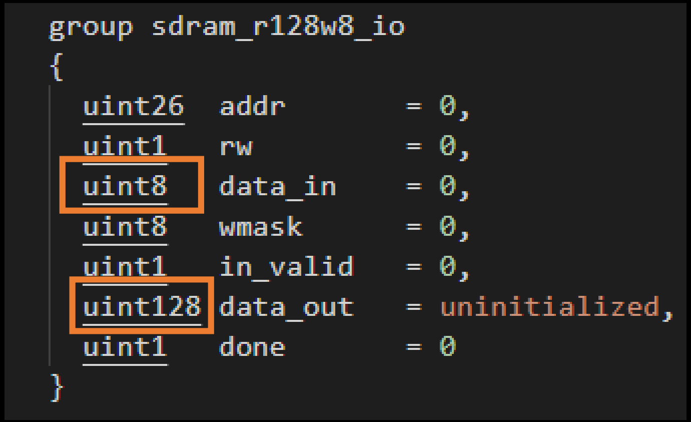
```
sdram_r16w16_io sio;

sdram_controller_autoprecharge_r16_w16 sdram(
  sd         <:> sio,
  // ...
);
```

silice

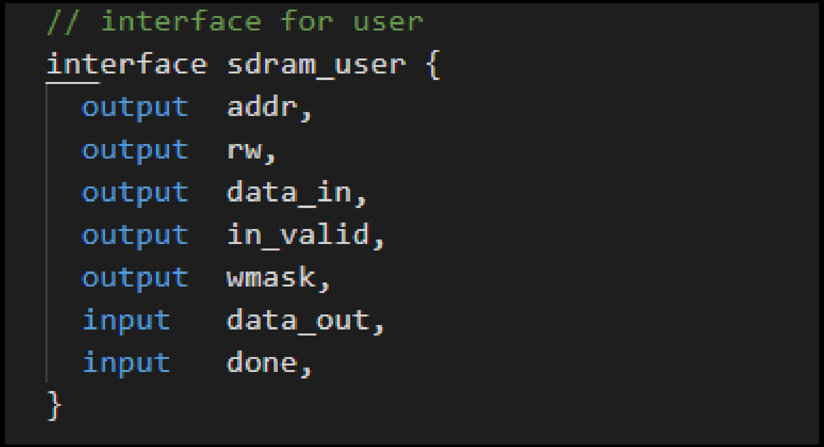
Groups and interfaces: genericity



```
group sdram_r16w16_io
{
  uint26  addr      = 0,
  uint1   rw        = 0,
  uint16  data_in   = 0,
  uint8   wmask     = 0,
  uint1   in_valid  = 0,
  uint16  data_out  = uninitialized,
  uint1   done      = 0
}
```



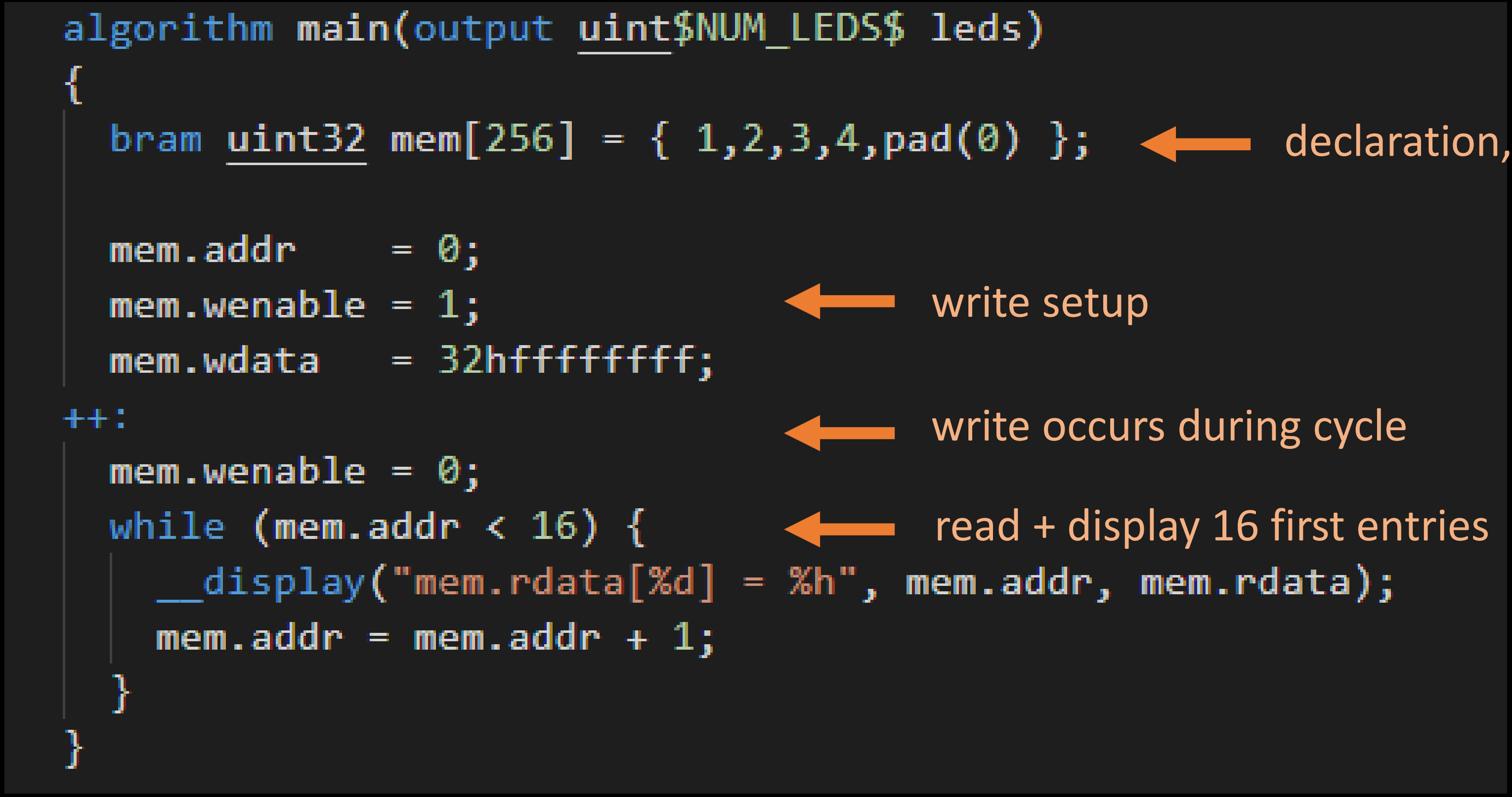
```
group sdram_r128w8_io
{
  uint26  addr      = 0,
  uint1   rw        = 0,
  uint8   data_in   = 0,
  uint8   wmask     = 0,
  uint1   in_valid  = 0,
  uint128 data_out  = uninitialized,
  uint1   done      = 0
}
```



```
// interface for user
interface sdram_user {
  output  addr,
  output  rw,
  output  data_in,
  output  in_valid,
  output  wmask,
  input   data_out,
  input   done,
}
```

Partial matches are also allowed (fewer entries in interface)

BRAMs and co.



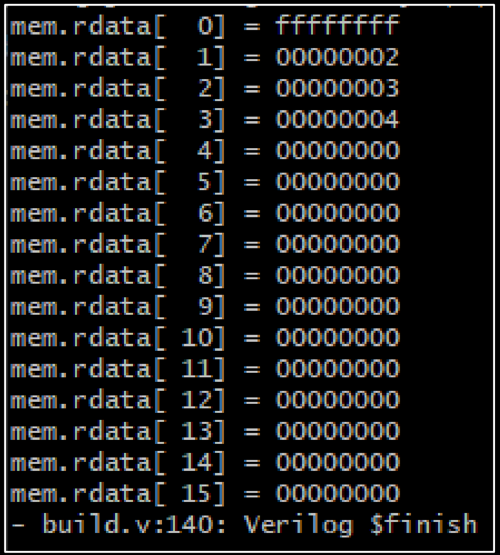
```
algorithm main(output uint$NUM_LEDS$ leds)
{
  bram uint32 mem[256] = { 1,2,3,4,pad(0) };

  mem.addr    = 0;
  mem.wenable = 1;
  mem.wdata   = 32hffffffff;
++:
  mem.wenable = 0;
  while (mem.addr < 16) {
    __display("mem.rdata[%d] = %h", mem.addr, mem.rdata);
    mem.addr = mem.addr + 1;
  }
}
```

declaration, *pad(c)* fills the remainder with *c*

write setup

write occurs during cycle

read + display 16 first entries



```
mem.rdata[  0] = ffffffff
mem.rdata[  1] = 00000002
mem.rdata[  2] = 00000003
mem.rdata[  3] = 00000004
mem.rdata[  4] = 00000000
mem.rdata[  5] = 00000000
mem.rdata[  6] = 00000000
mem.rdata[  7] = 00000000
mem.rdata[  8] = 00000000
mem.rdata[  9] = 00000000
mem.rdata[ 10] = 00000000
mem.rdata[ 11] = 00000000
mem.rdata[ 12] = 00000000
mem.rdata[ 13] = 00000000
mem.rdata[ 14] = 00000000
mem.rdata[ 15] = 00000000
- build.v:140: Verilog $finish
```

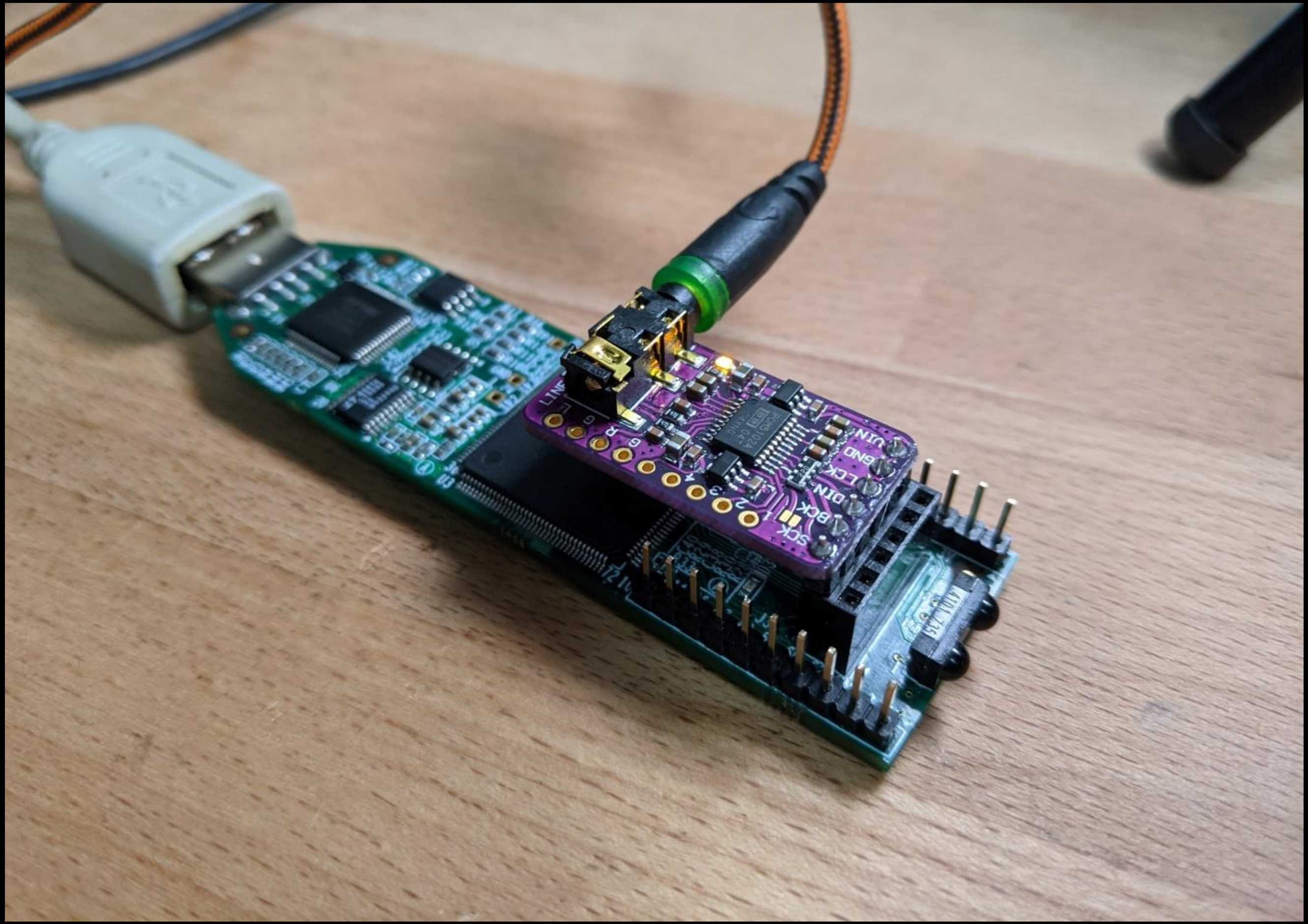
silice

There’s more

But we have discussed enough to read the projects!

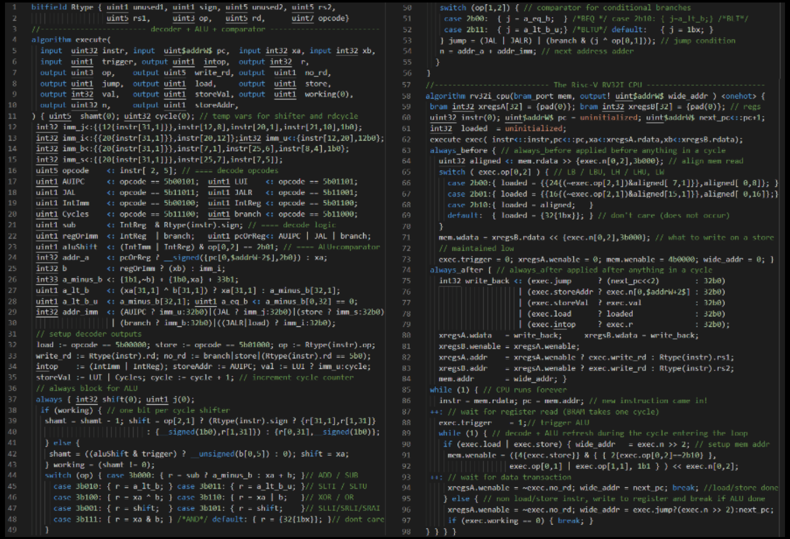
https://github.com/sylefeb/Silice/tree/master/learn-silice

PCM5102 DAC



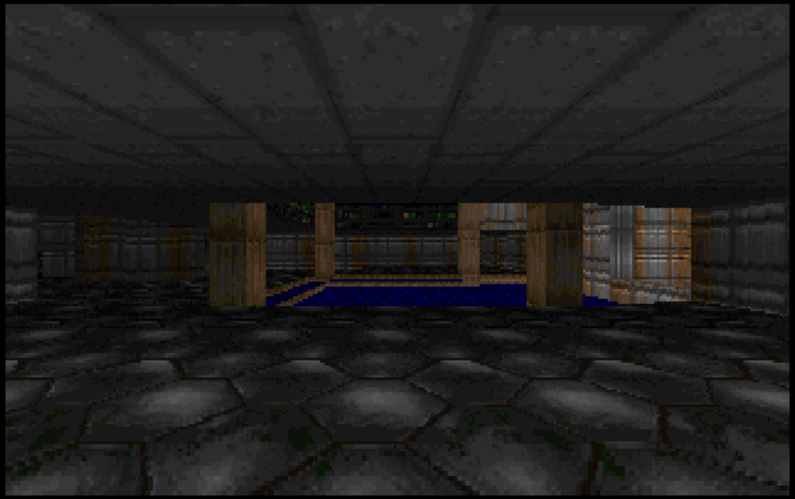
Many other projects

- audio_sdcard_streamer
- blinky
- bram_interface
- bram_wmask
- buttons_and_leds
- divint_bare
- doomchip
- fire-v
- hdmi_test
- i2s_audio
- ice-v
- ice40-dynboot
- ice40-warmboot
- inout
- lcd_test
- oled_sdcard_test
- oled_test
- oled_text
- pipeline_sort
- sdram_memtest
- sdram_test
- terrain
- uart_echo
- vga_demo
- vga_test
- vga_text_buffer
- vga_wfc
- video_sdram_test
- wolfpga



```
bitfield Rtype { uint1 unused1, uint1 sign, uint5 unused2, uint5 rs2,
                 uint5 rs1,     uint3 op,   uint5 rd,      uint7 opcode}
//------------------------ decoder + ALU + comparator ------------------------
algorithm execute(
  input  uint32 instr, input  uint$addrW$ pc,  input int32 xa, input int32 xb,
  input  uint1  trigger, output uint1  intop,  output int32  r,
  output uint3  op,    output uint5  write_rd, output  uint1  no_rd,
  output uint1  jump,  output uint1  load,     output  uint1  store,
  output int32  val,   output uint1  storeVal, output  uint1  working(0),
  output uint32 n,     output uint1  storeAddr,
) { uint5  shamt(0); uint32 cycle(0); // temp vars for shifter and rdcycle
 int32 imm_j<:{{12{instr[31,1]}},instr[12,8],instr[20,1],instr[21,10],1b0};
 int32 imm_i<:{{20{instr[31,1]}},instr[20,12]};int32 imm_u<:{instr[12,20],12b0};
 int32 imm_b<:{{20{instr[31,1]}},instr[7,1],instr[25,6],instr[8,4],1b0};
 int32 imm_s<:{{20{instr[31,1]}},instr[25,7],instr[7,5]};
 uint5 opcode    <: instr[ 2, 5]; // ==== decode opcodes
 uint1 AUIPC     <: opcode == 5b00101;  uint1 LUI    <: opcode == 5b01101;
 uint1 JAL       <: opcode == 5b11011;  uint1 JALR   <: opcode == 5b11001;
 uint1 IntImm    <: opcode == 5b00100;  uint1 IntReg <: opcode == 5b01100;
 uint1 Cycles    <: opcode == 5b11100;  uint1 branch <: opcode == 5b11000;
 uint1 sub       <: IntReg  & Rtype(instr).sign; // ==== decode logic
 uint1 regOrImm  <: IntReg  | branch;   uint1 pcOrReg<: AUIPC | JAL | branch;
 uint1 aluShift  <: (IntImm | IntReg) & op[0,2] == 2b01; // ==== ALU+comparator
 int32 addr_a    <: pcOrReg ? __signed({pc[0,$addrW-2$],2b0}) : xa;
 int32 b         <: regOrImm ? (xb) : imm_i;
 int33 a_minus_b <: {1b1,~b} + {1b0,xa} + 33b1;
 uint1 a_lt_b    <: (xa[31,1] ^ b[31,1]) ? xa[31,1] : a_minus_b[32,1];
 uint1 a_lt_b_u  <: a_minus_b[32,1]; uint1 a_eq_b <: a_minus_b[0,32] == 0;
 int32 addr_imm  <: (AUIPC ? imm_u:32b0)|(JAL ? imm_j:32b0)|(store ? imm_s:32b0)
                  | (branch ? imm_b:32b0)|((JALR|load) ? imm_i:32b0);
 // setup decoder outputs
 load := opcode == 5b00000; store := opcode == 5b01000; op := Rtype(instr).op;
 write_rd := Rtype(instr).rd; no_rd := branch|store|(Rtype(instr).rd == 5b0);
 intop    := (IntImm | IntReg); storeAddr := AUIPC; val := LUI ? imm_u:cycle;
 storeVal := LUI | Cycles; cycle := cycle + 1; // increment cycle counter
 // always block for ALU
 always { int32 shift(0); uint1 j(0);
  if (working) { // one bit per cycle shifter
   shamt = shamt - 1; shift = op[2,1] ? (Rtype(instr).sign ? {r[31,1],r[1,31]}
                      : {__signed(1b0),r[1,31]}) : {r[0,31],__signed(1b0)};
   } else {
    shamt = ((aluShift & trigger) ? __unsigned(b[0,5]) : 0); shift = xa;
   } working = (shamt != 0);
   switch (op) { case 3b000: { r = sub ? a_minus_b : xa + b; }// ADD / SUB
     case 3b010: { r = a_lt_b; } case 3b011: { r = a_lt_b_u; }// SLTI / SLTU
     case 3b100: { r = xa ^ b; } case 3b110: { r = xa | b;   }// XOR / OR
     case 3b001: { r = shift;  } case 3b101: { r = shift;    }// SLLI/SRLI/SRAI
     case 3b111: { r = xa & b; } /*AND*/ default: { r = {32{1bx}}; }// dont care
   }
   switch (op[1,2]) { // comparator for conditional branches
    case 2b00:  { j = a_eq_b;  } /*BEQ */ case 2b10: { j=a_lt_b;} /*BLT*/
    case 2b11:  { j = a_lt_b_u;} /*BLTU*/ default:   { j = 1bx; }
   } jump = (JAL | JALR) | (branch & (j ^ op[0,1])); // jump condition
   n = addr_a + addr_imm; // next address adder
  }
}
//------------------------ The Risc-V RV32I CPU ------------------------
algorithm rv32i_cpu(bram_port mem, output! uint$addrW$ wide_addr ) <onehot> {
 bram int32 xregsA[32] = {pad(0)}; bram int32 xregsB[32] = {pad(0)}; // regs
 uint32 instr(0); uint$addrW$ pc = uninitialized; uint$addrW$ next_pc<::pc+1;
 int32  loaded  = uninitialized;
 execute exec( instr<::instr,pc<::pc,xa<:xregsA.rdata,xb<:xregsB.rdata);
 always_before { // always_before applied before anything in a cycle
   uint32 aligned <: mem.rdata >> {exec.n[0,2],3b000}; // align mem read
   switch ( exec.op[0,2] ) { // LB / LBU, LH / LHU, LW
    case 2b00:{ loaded = {{24{(~exec.op[2,1])&aligned[ 7,1]}},aligned[ 0,8]}; }
    case 2b01:{ loaded = {{16{(~exec.op[2,1])&aligned[15,1]}},aligned[ 0,16]};}
    case 2b10:{ loaded = aligned;   }
    default:  { loaded = {32{1bx}}; } // don't care (does not occur)
   }
   mem.wdata = xregsB.rdata << {exec.n[0,2],3b000}; // what to write on a store
   // maintained low
   exec.trigger = 0; xregsA.wenable = 0; mem.wenable = 4b0000; wide_addr = 0; }
 always_after { // always_after applied after anything in a cycle
   int32 write_back <: (exec.jump      ? (next_pc<<2)        : 32b0)
                     | (exec.storeAddr ? exec.n[0,$addrW+2$] : 32b0)
                     | (exec.storeVal  ? exec.val            : 32b0)
                     | (exec.load      ? loaded              : 32b0)
                     | (exec.intop     ? exec.r              : 32b0);
   xregsA.wdata   = write_back;     xregsB.wdata = write_back;
   xregsB.wenable = xregsA.wenable;
   xregsA.addr    = xregsA.wenable ? exec.write_rd : Rtype(instr).rs1;
   xregsB.addr    = xregsA.wenable ? exec.write_rd : Rtype(instr).rs2;
   mem.addr       = wide_addr; }
 while (1) { // CPU runs forever
   instr = mem.rdata; pc = mem.addr; // new instruction came in!
 ++: // wait for register read (BRAM takes one cycle)
   exec.trigger    = 1;// trigger ALU
   while (1) { // decode + ALU refresh during the cycle entering the loop
    if (exec.load | exec.store) { wide_addr   = exec.n >> 2; // setup mem addr
     mem.wenable = ({4{exec.store}} & { { 2{exec.op[0,2]==2b10} },
                   exec.op[0,1] | exec.op[1,1], 1b1 } ) << exec.n[0,2];
 ++: // wait for data transaction
     xregsA.wenable = ~exec.no_rd; wide_addr = next_pc; break; //load/store done
    } else { // non load/store instr, write to register and break if ALU done
     xregsA.wenable = ~exec.no_rd; wide_addr = exec.jump?(exec.n >> 2):next_pc;
     if (exec.working == 0) { break; }
} } } }
```

ice-v, a tiny Risc-V processor in Silice



The DooM-chip

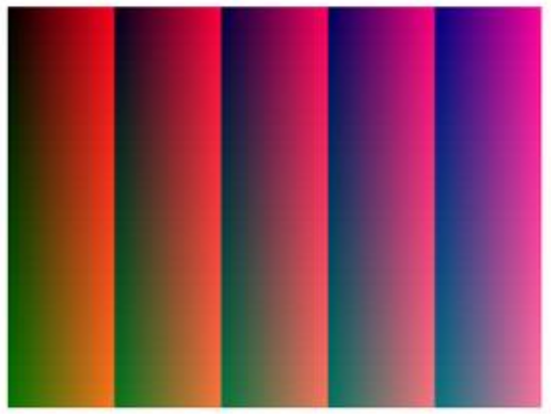
Thank you!

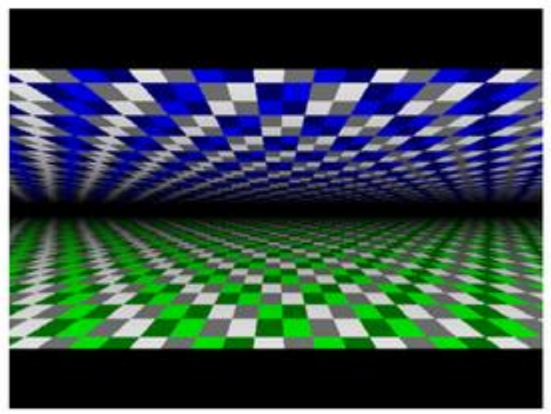
I hope you'll find Silice useful

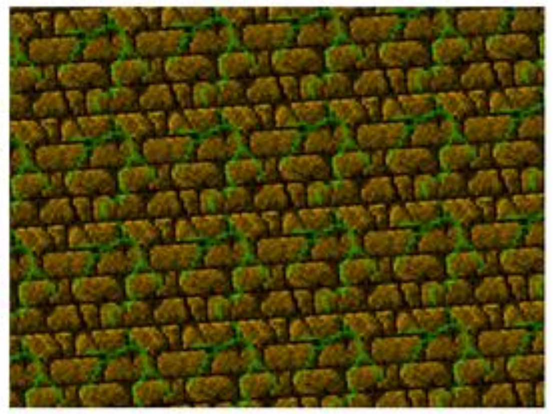
Getting started: https://github.com/sylefeb/Silice/

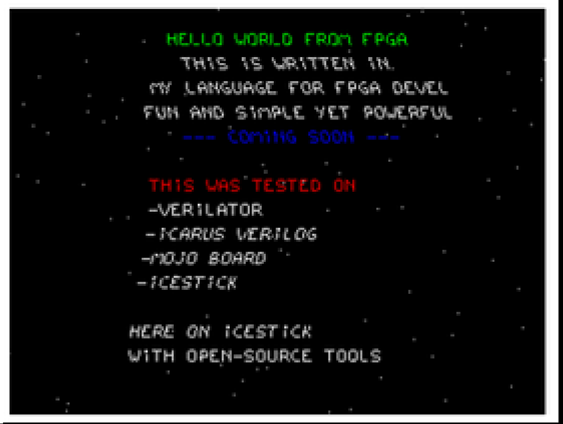
Thanks to Silice contributors!

robng15, mesabloo, trabucayre, osnr, diadatp, ttricard, umarcor, juanmard









```
HELLO WORLD FROM FPGA
THIS IS WRITTEN IN
MY LANGUAGE FOR FPGA DEVEL
FUN AND SIMPLE YET POWERFUL
--- COMING SOON ---

THIS WAS TESTED ON
-VERILATOR
-ICARUS VERILOG
-MOJO BOARD
-ICESTICK

HERE ON ICESTICK
WITH OPEN-SOURCE TOOLS
```

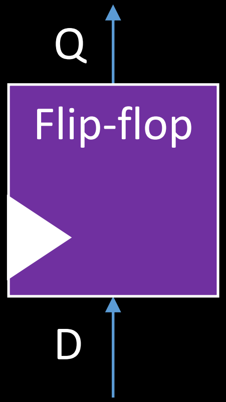
silice

Wiring operators

<:   binds the variable as it is changed in the cycle (D)

<:: binds the variable as it was at the cycle start (Q)

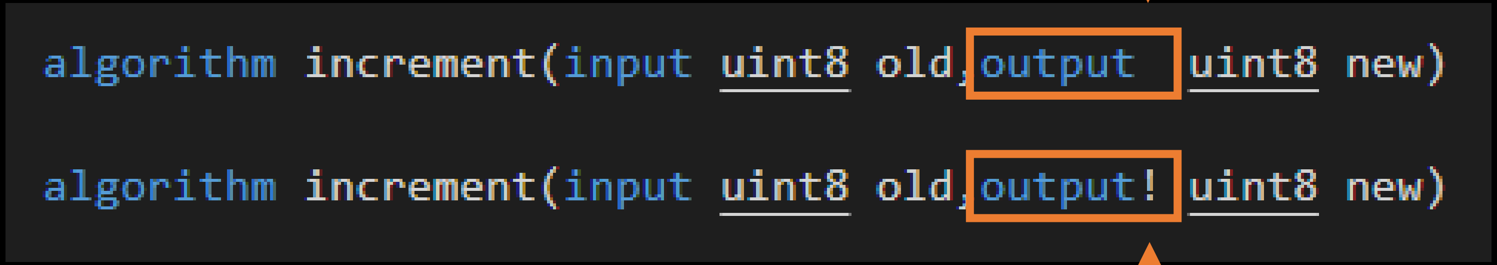
Pay 1 in latency, gain in F_max



silice

Algorithm outputs

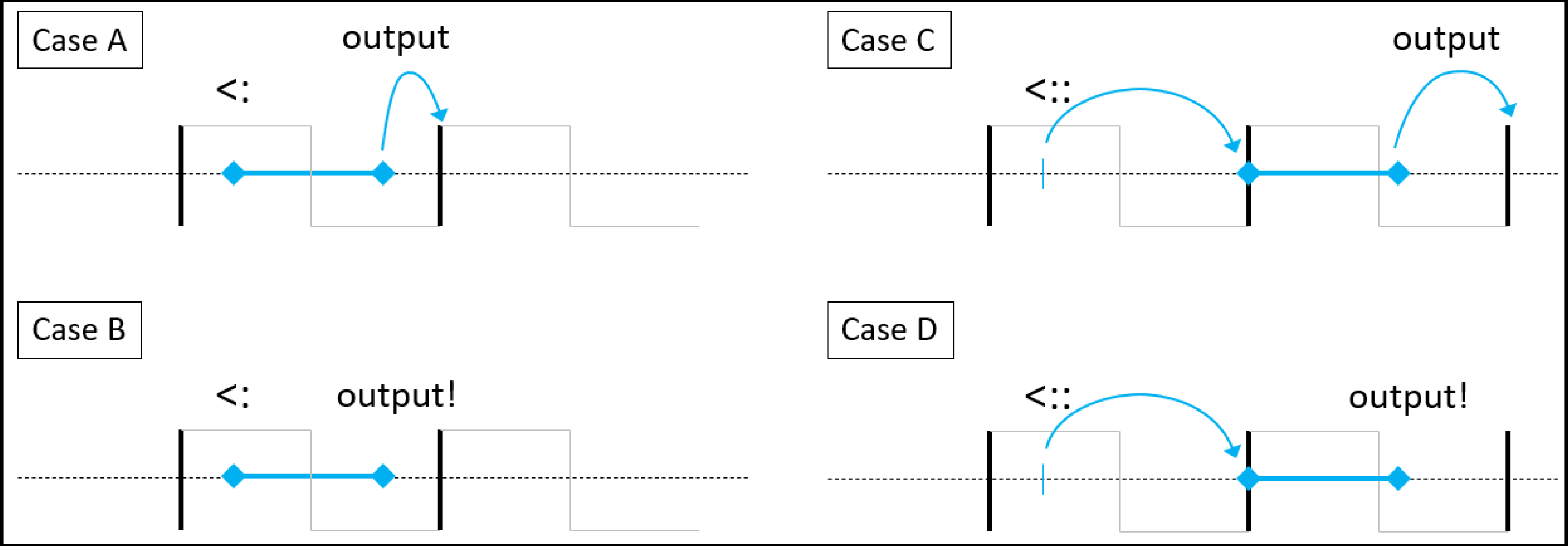
Typical, registered output (1 cycle latency)



```
algorithm increment(input uint8 old, output  uint8 new)

algorithm increment(input uint8 old, output! uint8 new)
```

Immediate output (no latency)

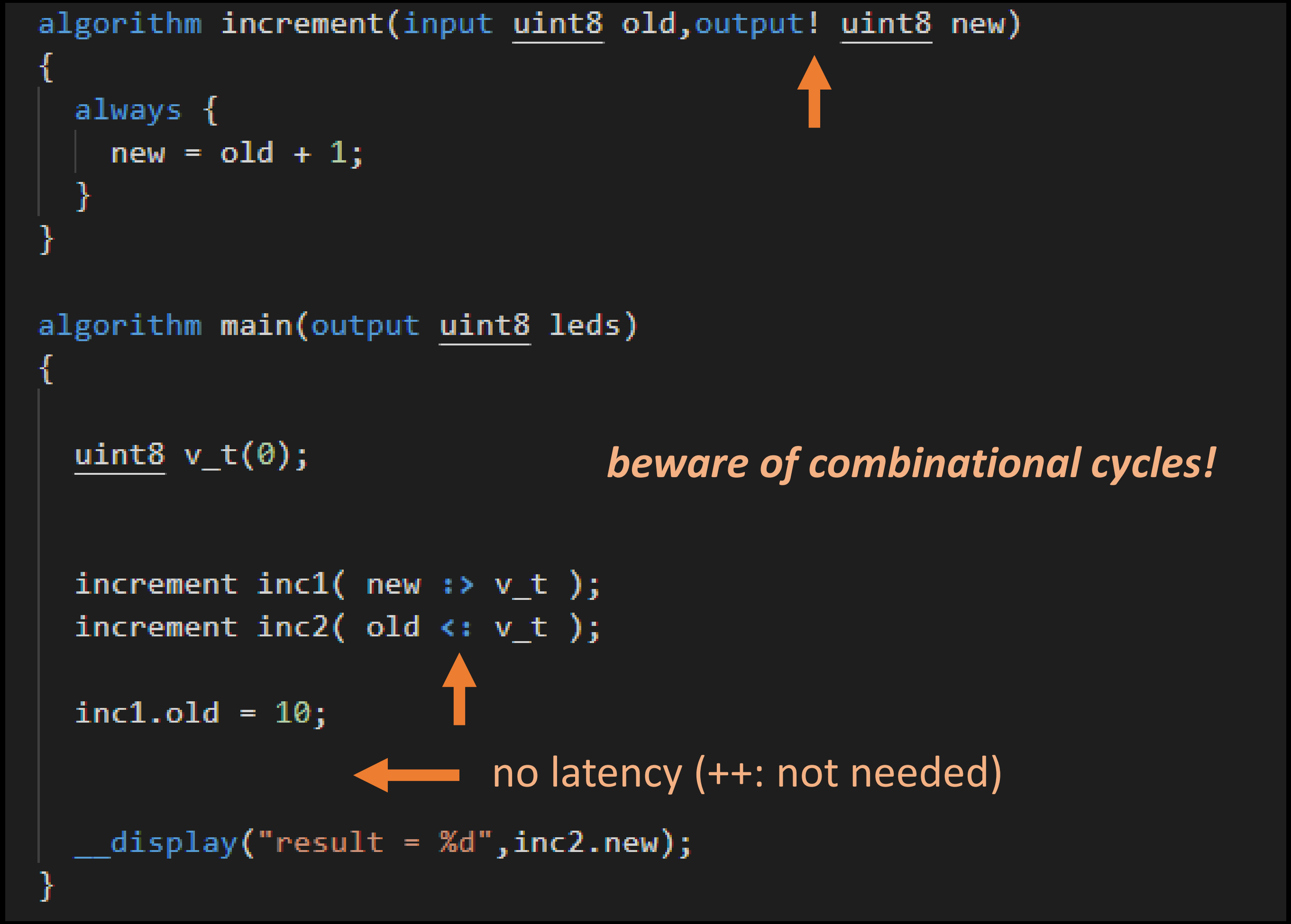
Fine grain control over latencies



https://github.com/sylefeb/Silice/blob/master/learn-silice/AlgoInOuts.md

silice

Case B: example



```
algorithm increment(input uint8 old,output! uint8 new)
{
  always {
    new = old + 1;
  }
}

algorithm main(output uint8 leds)
{

  uint8 v_t(0);


  increment inc1( new :> v_t );
  increment inc2( old <: v_t );

  inc1.old = 10;


  __display("result = %d",inc2.new);
}
```

beware of combinational cycles!

no latency (++: not needed)

silice

Control flow rules

```
a = ... ;
if (...) {
   b = ... ;
} else {
  if (...) {
    alg <- ();
  }
}
c = ... ;
```

one-cycle block

always blocks are required to be one-cycle blocks

```
...  cycle i
while ( ... ) {
    cycle i+1
    ...
    cycle n
}
while ( ... ) {
    cycle n+1
    ...

}
...
```

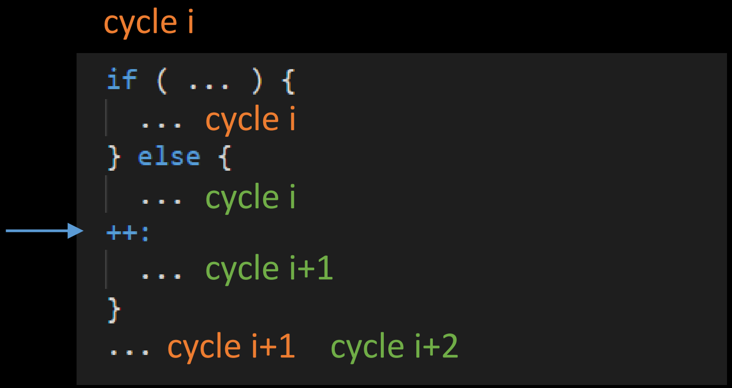
assuming one-cycle

iterates *every* cycle

```
cycle i     a = ... ;
            ++:
cycle i+1   b = ... ;
```

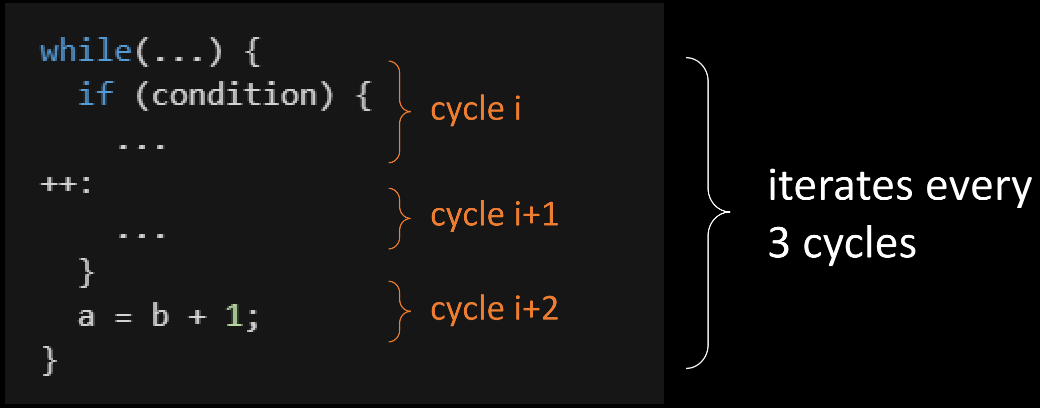
step operator
(introduces one cycle)

Control flow rules



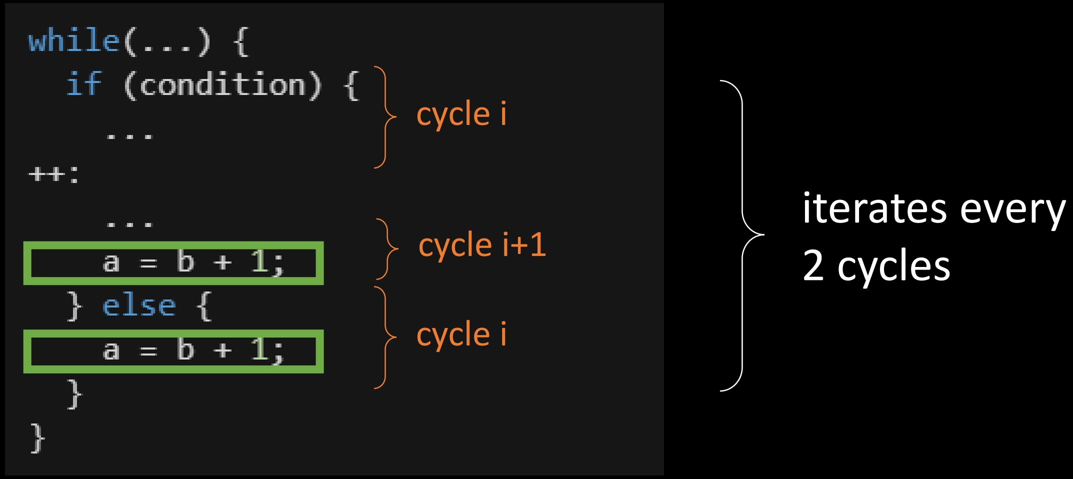
```
cycle i
if ( ... ) {
  ... cycle i
} else {
  ... cycle i
++:
  ... cycle i+1
}
... cycle i+1    cycle i+2
```

One branch *not* one-cycle ➔ one cycle after if-then-else



```
while(...) {
  if (condition) {   } cycle i
    ...
++:                  } cycle i+1
    ...
  }
  a = b + 1;         } cycle i+2
}
```

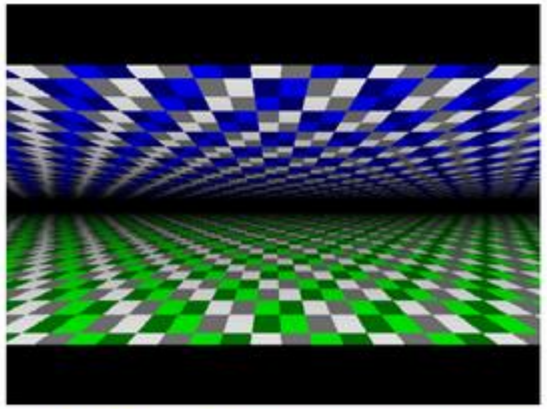
iterates every 3 cycles

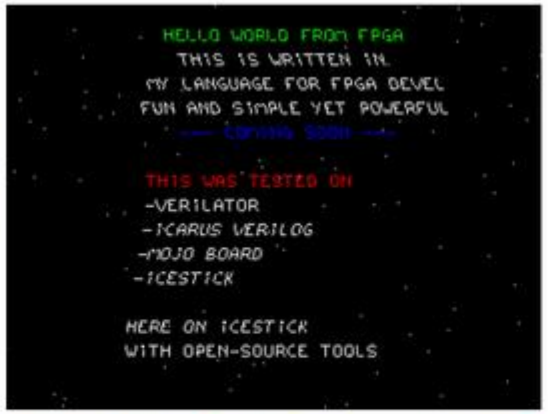


```
while(...) {
  if (condition) {   } cycle i
    ...
++:
    ...
    a = b + 1;       } cycle i+1
  } else {
    a = b + 1;       } cycle i
  }
}
```

iterates every 2 cycles

silice

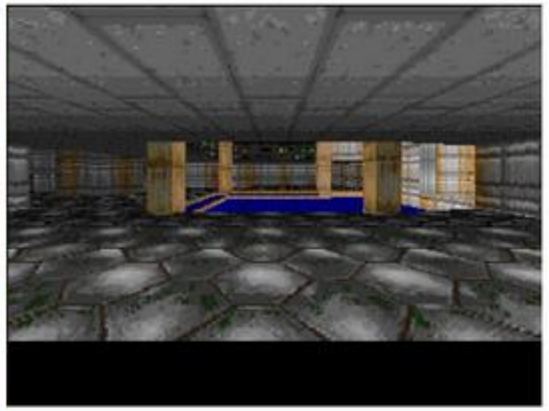


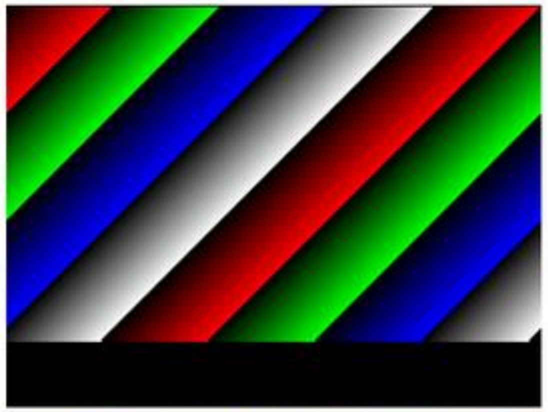


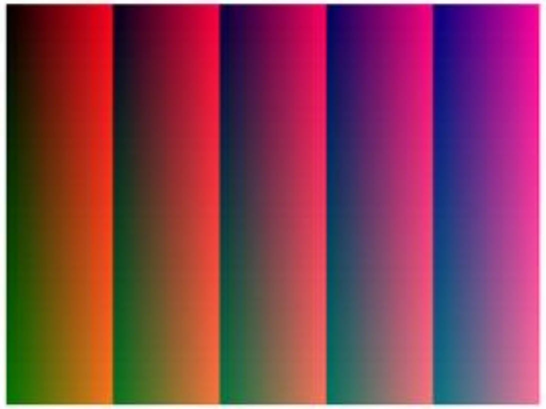
HELLO WORLD FROM FPGA
THIS IS WRITTEN IN
MY LANGUAGE FOR FPGA DEVEL
FUN AND SIMPLE YET POWERFUL
—— COMING SOON ——

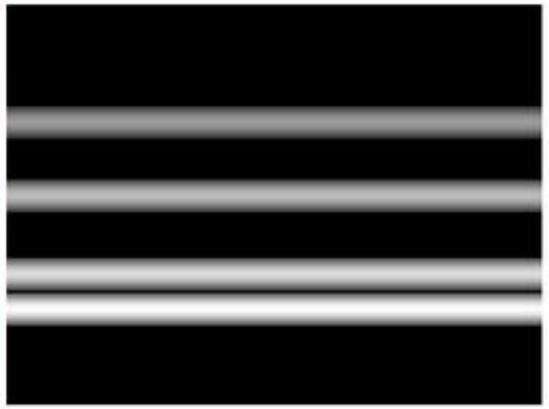
THIS WAS TESTED ON
-VERILATOR
-ICARUS VERILOG
-MOJO BOARD
-ICESTICK

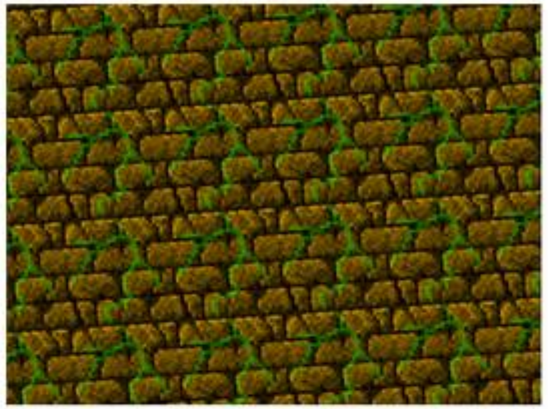
HERE ON ICESTICK
WITH OPEN-SOURCE TOOLS











silice