Hardware design basics

as viewed by a software person

(part 2)

This presentation (along with the sample code) lives on github: https://github.com/quarck/vhdl_intro

VHDL – a bit more formal introduction

- Strictly typed language
- Types of VHDL objects:
  - constant
  - variable
  - signal
  - file (we would ignore this one)
- Object definition:

```
<Object type> <name> : <data type> [:= Initial value];
```

VHDL: example object definitions

```
constant MATH_PI : real := 3.1415927;
variable ctr : integer :=0;
signal Q : bit := '0';
```

Note: signal assignment is different between variables and signals (on purpose, we would cover it later):

```
Q <= not Q;
ctr := ctr + 1;
```

VHDL – a bit more formal: basic types

- `type **bit** is ('0', '1');`
- `type **boolean** is (false, true);`
- `type **character** is (NUL, SOH, … ' ', '!', … 'A', 'B', … 'a', 'b', …);`
- `type **integer** is range -2.147.483.64**7** to 2.147.483.647;` -- particular implementations may extend this range
- `type **real** is range -1.0e38 to 1.0e38;` -- particular implementations may extend this range

VHDL – a bit more formal: basic types

- `subtype natural is integer range 0 to integer'high;`
- `subtype positive is integer range 1 to integer'high;`
- `type string is array (positive range <>) of character;`
- `type bit_vector is array (natural range <>) of bit;`

VHDL – a bit more formal: types

std_logic is a library defined type:

```
type std_logic is (
      'U',        -- Uninitialized
      'X',        -- Forcing unknown
      '0',        -- Forcing 0
      '1',        -- Forcing 1
      'Z',        -- High impedance
      'W',        -- Weak unknown
      'L',        -- Weak 0
      'H',        -- Weak 1
      '-'         -- Don't care
);
```

VHDL – a bit more formal: physical types

By example, system **time** type:

```
type time is range 0 to 1e20
   units
     fs;
     ps = 1000 fs;
     ns = 1000 ps;
     us = 1000 ns;
     ms = 1000 us;
     sec = 1000 ms;
     min = 60 sec;
     hr = 60 min;
   end units time;
```

VHDL: arrays, example

```
type ram_t is array (0 to 255)
    of std_logic_vector(7 downto 0);

signal ram : ram_t :=
    (others => (others => '0'));

type bit_vector is array (natural range <>)
    of bit;
```

VHDL: arrays, example

```
type ram_t is array (0 to 255) of std_logic_vector(7
downto 0);

signal ram : ram_t := (others => (others => '0'));

-- Xilinx-specific: can be also peppered with:
attribute ram_style: string;
attribute ram_style of ram : signal is "distributed";
-- or
attribute ram_style of ram : signal is “block";
-- or
attribute ram_style of ram : signal is “auto";
```

VHDL: records (structs), example

```
type t_TO_FIFO is record
    wr_en     : std_logic;
    wr_data   : std_logic_vector(7 downto 0);
    rd_en     : std_logic;
end record t_TO_FIFO;
```

VHDL: operations: relational

Operation	Comment
=	
/=	Not equal
<	
<=	
>	
>=	

VHDL: operations: logic

Operation	Comment
and	
or	
nand	a nand b = not (a and b)
nor	a nor b = not (a or b)
xor	exclusive or
not	

VHDL: operations: Arithmetic

Operation	Comment
+	
-	
*	
/	
**	Power
mod	
rem	
abs	

VHDL: operations: Concatenation

Operation	Comment
&	

```
signal a : std_logic_vector (7 downto 0) := "00000000";
signal b : std_logic_vector (8 downto 0);
…
b <= '0' & a;          -- makes 8-bit signal into 9-bit
```

VHDL: design units

- Entity
- Architecture
- Package
- Configuration (we would skip this one for now)

VHDL: entity, formal syntax

```
entity <id> is
     [<generic>];
     [<ports>];
     [<declarations>];
   [begin <sentences>];
end [entity] [<id>];
```

VHDL: entity - example

```
entity MUX21 is
    port(
        A       : in bit;
        B       : in bit;
        Ctrl    : in bit;
        Z       : out bit;
end MUX21;
```

VHDL: entity, example with generics

```
entity MUX21n is
     generic( n : integer := 2);
     port( A      : in  bit_vector(n-1 downto 0);
           B      : in  bit_vector(n-1 downto 0);
           Ctrl   : in  bit;
           Z      : out bit_vector(n-1 downto 0));
end MUX21;
```

VHDL: architecture, formal syntax

```
architecture <id> of <id_entity> is
    [<declarations>];
begin
    <concurrent sentences>;
end [architecture] [<id>];
```

VHDL: architecture, code example:

```
architecture Behaviour of MUX21n is
begin
    process(A, B, Ctrl)
    begin
        if Ctrl = '0' then
            Z <= A;
        else
            Z <= B;
        end if;
    end process;
end Behaviour;
```

VHDL: package

- Declaration:

```
package <identifier>
     [<declarations>];
end [package] [<identifier>]
```

- Body:

```
package body <identifier>
     [<Assignments and Detailed definitions>];
end [package body] [<identifier>]
```

VHDL: package usage

- Usage:

```
use <library>.<package name>.[<identifier> | all];
```

- Example standard packages:

```
std_logic_1164
std_logic_arith
```

- Used as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

VHDL: concurrent & sequential statements

- Two types of code statemtns:
  - Concurrent
  - Sequential
- All the VHDL code is concurrent by default
- Unless it resides inside the “process” statement (also applies to “procedure” and “function” declarations)
- Anything inside “process” statement is sequential

VHDL: concurrent statements

- All the lines of code are *"executed"* simultaneously
- Example:

```
signal a : std_logic := '1';
signal b : std_logic := '0';
begin
     -- code below swaps bit values between a and b
     a <= b when rising_edge(clk);
     b <= a when rising_edge(clk);
end …;
```

VHDL: sequential statements

- Code inside the “process” statement behaves as if it was just a regular software thread, *“executing”* line by line
- Each “process” is executing in parallel with other processes
- Synthesis tool would analyse the behaviour and produce the netlist that implements the required behaviour
- A lot of sequential code statements cannot be synthesised, but can be used in testbenches, for example:

```
wait for 20ns;
```

VHDL: process (sequential statements):

- Syntax:

```
[label:]
process [(<sensitivity list>)]
begin
    <sequential sentences>
    [<wait statements>;]
end process;
```

Either sensitivity list or at least one wait statement must be present.

VHDL: process, sensitivity list:

- Sequential statements inside “process” block are re-run each time any signal in the sensitivity list changes its value
- (Process statement is executed once unconditionally at “start”)
- Example:

```
process (clk, x)
begin
    if rising_edge(clk) then
        z <= x;
    end if;
end process;
```

VHDL: process, wait statement:

- “Wait” statement suspends execution of the process until the wait condition is met
- Can have simply a list of signals, in such case it is an equivalent to a sensitivity list Example:

```
process
begin
     if rising_edge(clk) then
          z <= x;
     end if;
     wait on clk, x;
end process;
```

VHDL: process, wait statement:

- Formal syntax:

```
[label:] wait [on <signal> {, ...}]
[until <boolean_expresion>]
[for <time_expresion>];
```

- Examples:

```
wait on clk, x;
wait for 10 ns;
wait until clk = '1';
wait;    -- "terminates" the process
```

VHDL: process, signals vs variables

- Both “signals” and “variables” can be used inside the process statement
- Variable assignment has an immediate effect
- Signal assignment only “applies” when process is suspended

VHDL: process, signals vs variables, example:

```
process
      variable a : std_logic = '0';
      signal b : std_logic = '1';
begin
      if rising_edge(clk) then
            a := y;     -- value of 'a' is changed immediately
            b <= y;     -- value of 'b' is 'cached'
            if x = '1' then
                  a := z;     -- value of 'a' is changed again
                  b <= z;     -- 'b' is still cached
            end if;
      end if;
      wait on (clk, x, y, z); -- here 'b' is finally applied
end process;
```

VHDL: process, signals vs variables

- “signal” translates directly into a wire of flip-flop
- “variable” gives some flexibility for the tool to analyse the behaviour and produce netlist that implements it, might as well get translated into the wire of flip-flop
- “signal” can be viewed in the simulation waveform, while “variable” is usually not

VHDL: Variable assignment (process only)

```
[label:] <variable name> := <expression>;
```

VHDL: signal assignment

```
[label:] <signal_name> <= [delay_type] <expression>
{after <delay>};
```

Can take quite complicated form, but these can be only used in testbenches:

```
A <= '0', '1' after 10 ns, '0' after 15 ns, '1'
after 20 ns, '0' after 30 ns, '1' after 50 ns, '0'
after 70 ns;
B1 <= transport A after 10 ns;
B2 <= A after 10 ns;
B3 <= reject 5 ns A after 10 ns;
```

We would skip these ones

VHDL: conditional signal assignment

```
[<label>:] <signal> <= [delay_type]
{<expression|waveform> when <boolean expression> else}
< expression|waveform> [when <boolean expression>];
```

Example:

```
z <= a when s = "00" else
     b when s = "11" else
   unnafected when others;
```

VHDL: selective signal assignment

```
[<label>:]
with <expression> select
     <signal> <= [delay_type]
          {<expression|waveform> when <value>,}
          <expression|waveform> when <value>;
```

Example:

```
with opcode select
     res <= A + B when “00”,
            A - B when “01”,
            A and B when “10”,
            A or B when “11”;
```

WOW! We’ve just implemented a very basic ALU (arithmetic logic unit**) by accident!**

VHDL: sequential stm, if-statements, formal syntax

```
[label:] if <condicion> then
             <sentencias secuenciales>
         {elsif <condicion> then
             <sentencias secuenciales}
         [else
             <sentencias secuenciales>]
         end if [label];
```

VHDL: sequential stm, case-statements, formal syntax

```
[label:] case <expresion> is
              {when <value1> =>
                   <sequential sentences>;}
              {when <value2> =>
                   <sequential sentences>;}
              [when others =>
                   <sequential sentences>;]
         end case [label];
```

VHDL: sequential stm, while loops

```
[label:] while <boolean_condition>
         loop
              <sequential sentences>
         end loop [label];
```

VHDL: sequential stm, for loops

```
[label:] for <repetition_control>
         loop
              <sequential sentences>
         end loop [label];

example:
     for i in 1 to 10 loop
         <statements>;
     end loop example;
```

Loop variable should not be declared in advance, its lifetime is limited to the loop body

'to' can be changed to 'downto' for going down instead of going up.

VHDL: generate statements, ‘if’

Conditionally generate or skip a piece of definition, syntax:

```
<label>: if <condition>
    generate
        {<concurrent sentences>}
    end generate;
```

VHDL: generate statements, ‘if’

Example:

```
if_32_bit: if n = 32
      generate
            a: ALU_32_bit port map (…);
      end generate;
if_lt32_bit: if n < 32
      generate
            a: ALU_simple port map (…);
      end generate;
-- to ensure we've covered all cases, let's assert a bit:
assert n <= 32 report "Only 1-32 bits are supported"
      severity failure;
```

VHDL: generate statements, “for”

Generates repeating blocks. Syntax:

```
<label>: for <range specification>
    generate
        {<concurrent sentences>}
    end generate;
```

VHDL: generate statements, ‘for’

Example:

```
gen_adder: for i in 0 to 31
    generate
        adder: full_adder
            port map(
                a => a(i), b => b(i),
                c_in => carry(i),
                S => res(i),
                c_out => carry(i + 1));
    end generate;
```

VHDL: function, just a syntax

Declaration:

```
function <name> [(<parameters list>)] return
<data_type>;
```

Definition:

```
function <name> [(<parameters list>)] return
<data_type> is
{<declarative part>}
begin
{<sequential sentences>}
end [function] [<name>];
```

One or more “return [expression];” in here

VHDL: procedure, just a syntax

Declaration:

```
procedure <name> [(<parameters list>)];
```

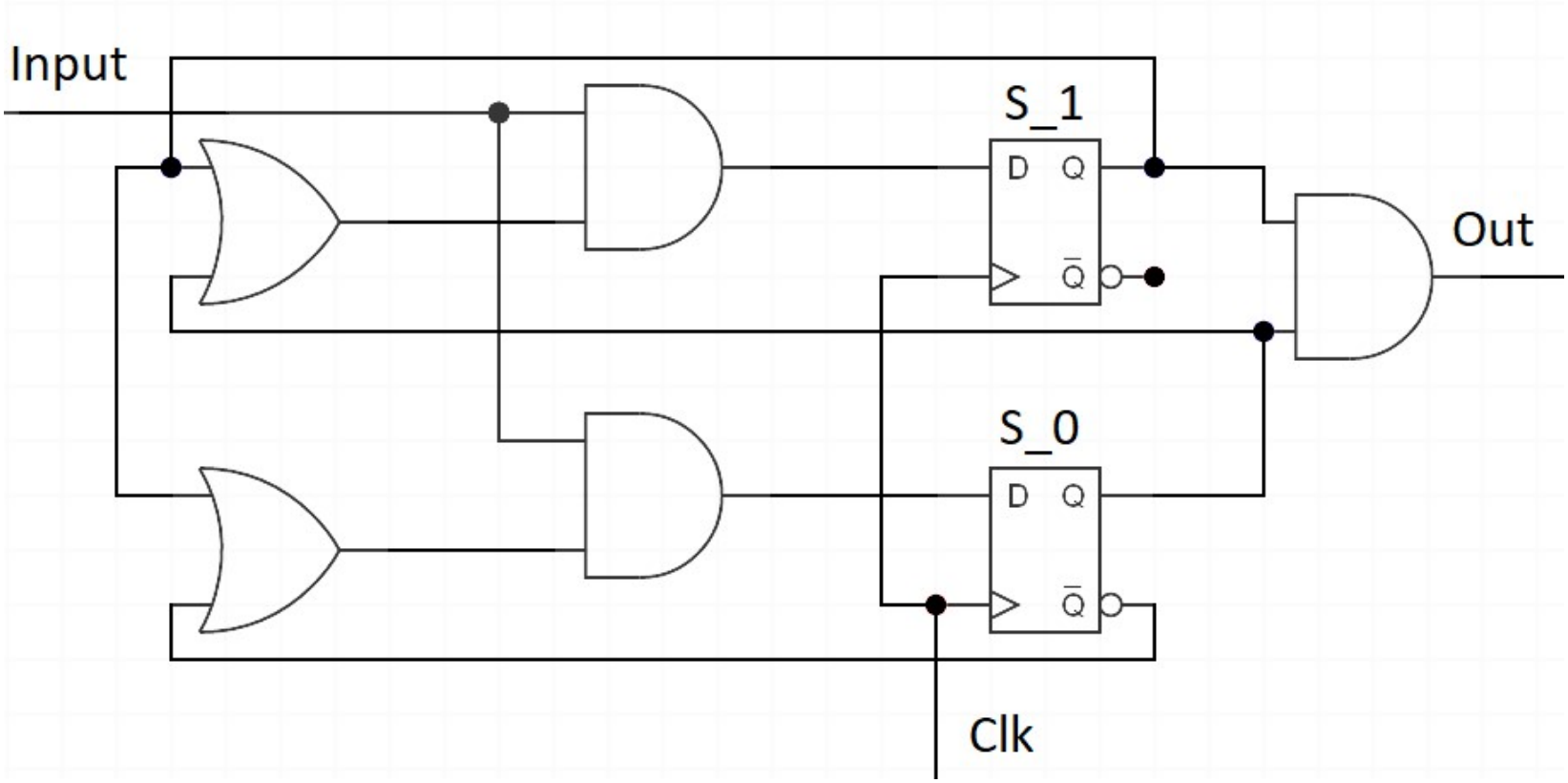
Definition:

```
procedure <name> [(<parameters list>)] is
{<declarative part>}
begin
{<sequential sentences>}
end [procedure] [<name>];
```

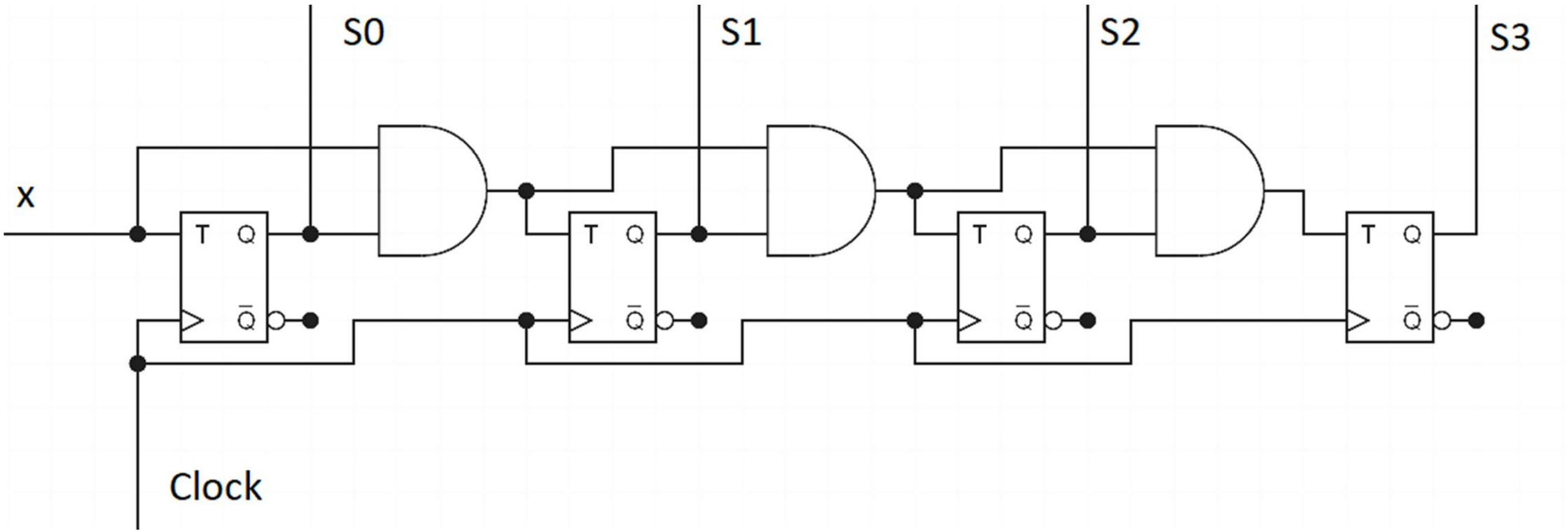
FPGAs

- First, let’s recall our state machines that we did already in the part 1

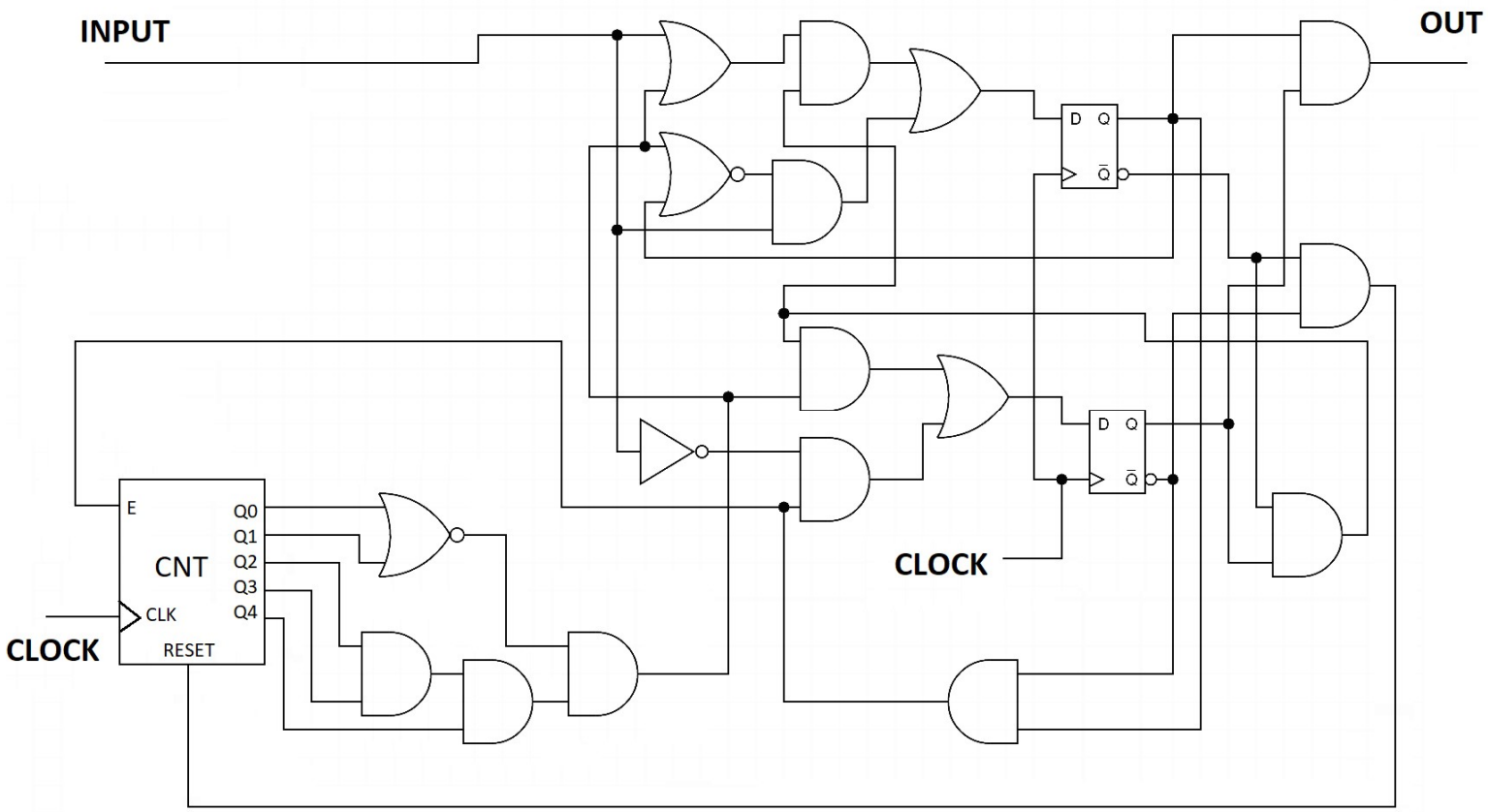
Sequence 111 detection FSM



Sync counter FSM



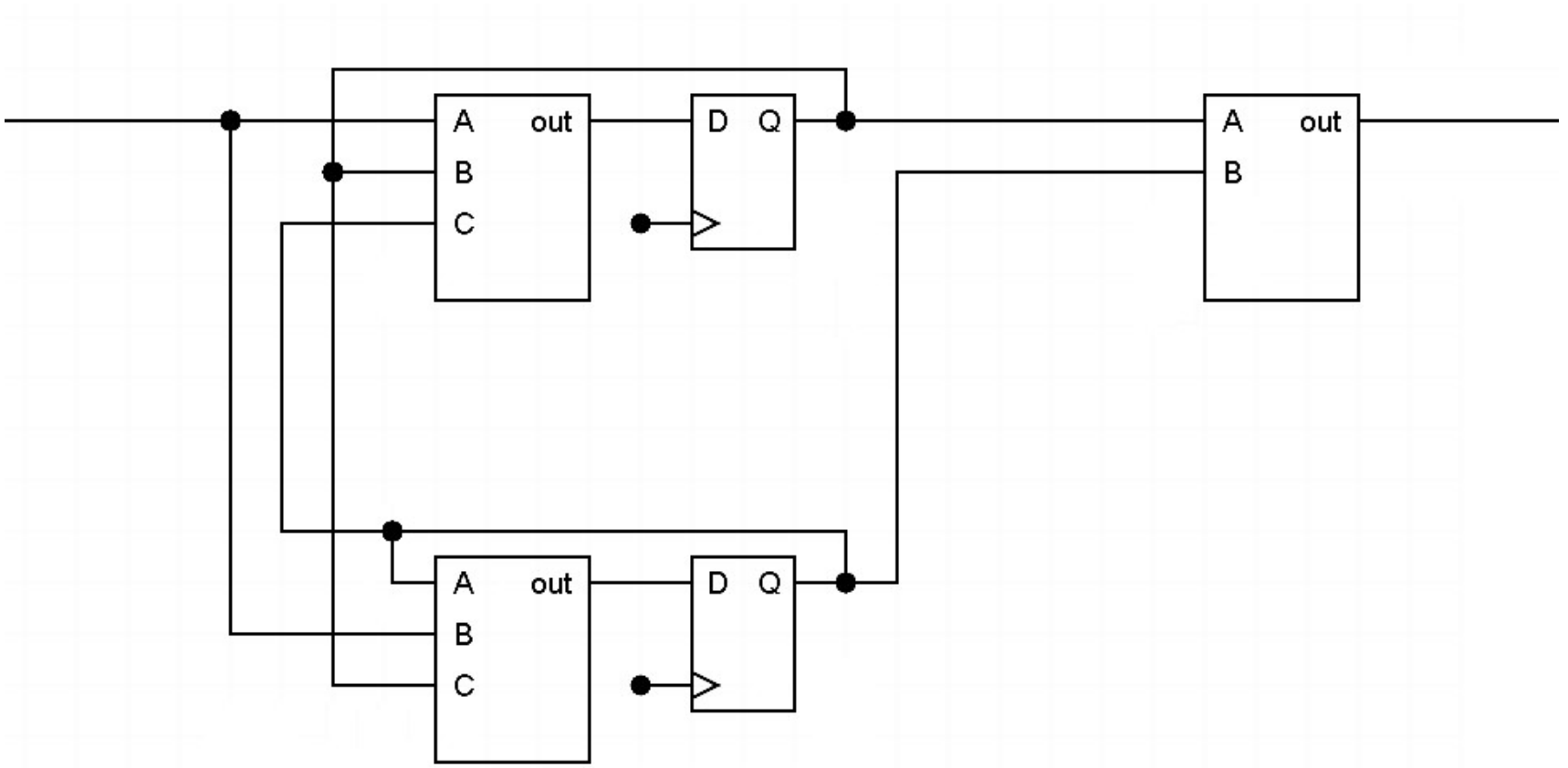
Ethernet preamble detection FSM



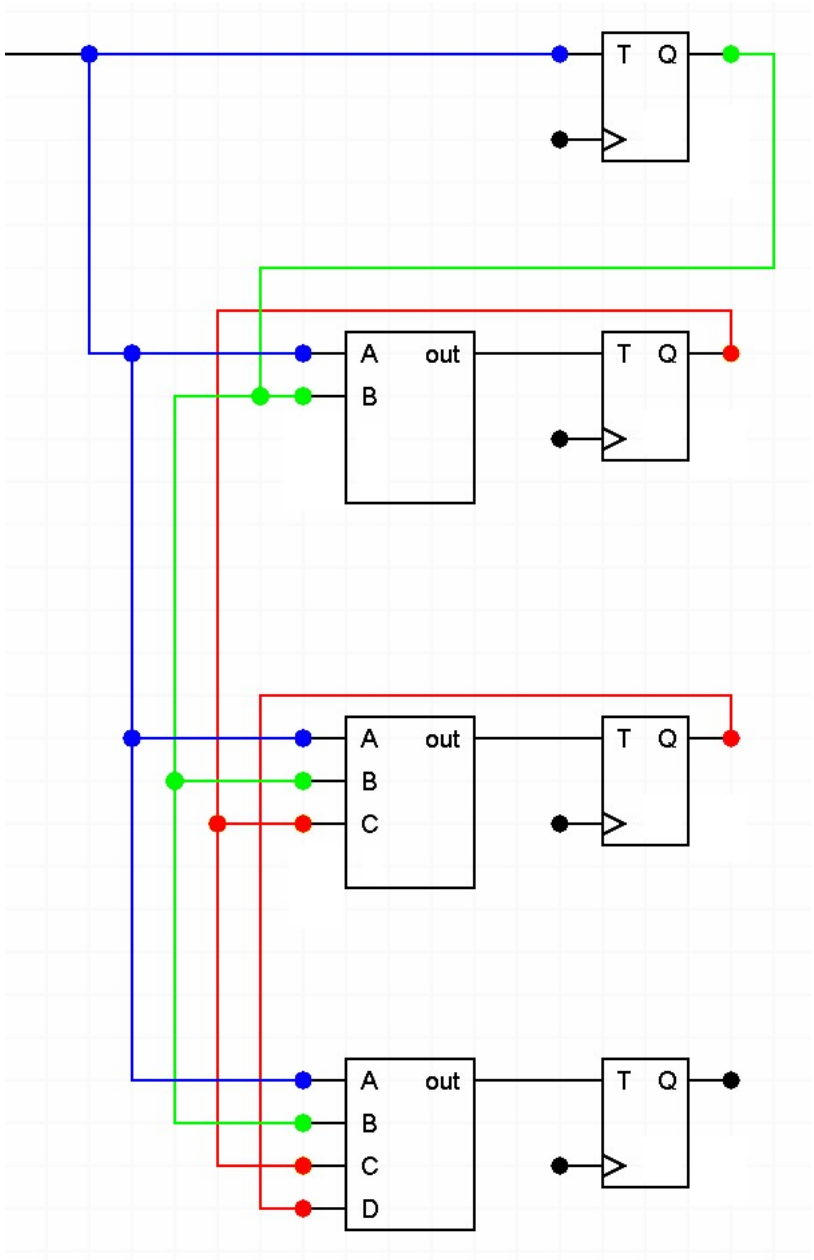
FPGAs..

- Notice a pattern in all our state machines:
- In consists of flip-flops paired with some combinatory circuit that defines the next state of the flip flop
- Inputs into the combinatory circuit could be either current circuit's input or the current value of the circuit flipflop(s) or both.
- If we denote our combinatory circuits as black boxes, we can draw these state machines as (next slides):

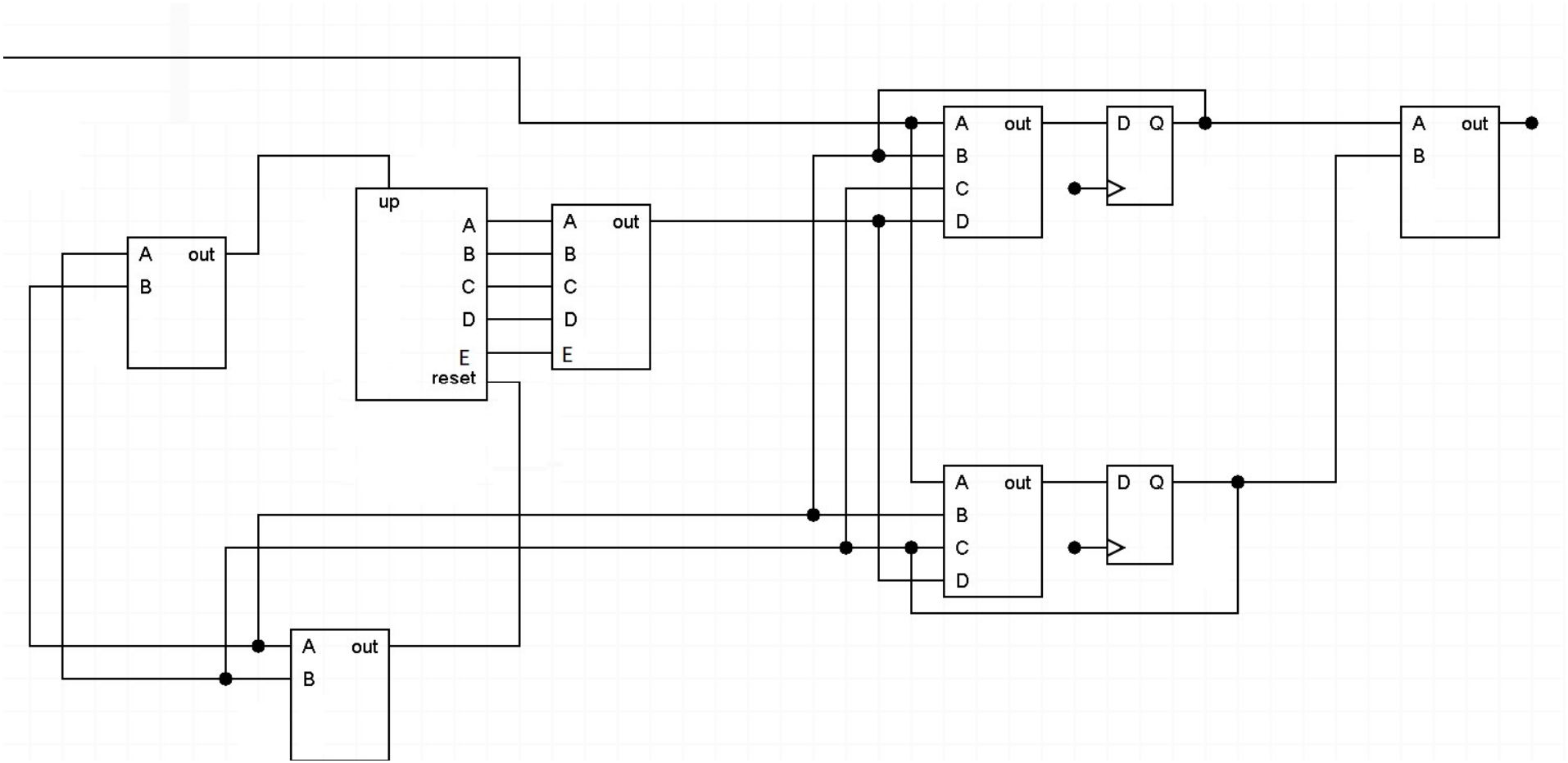
Sequence 111 detection, generalized



Sync counter:



Ethernet preamble detection



FPGA

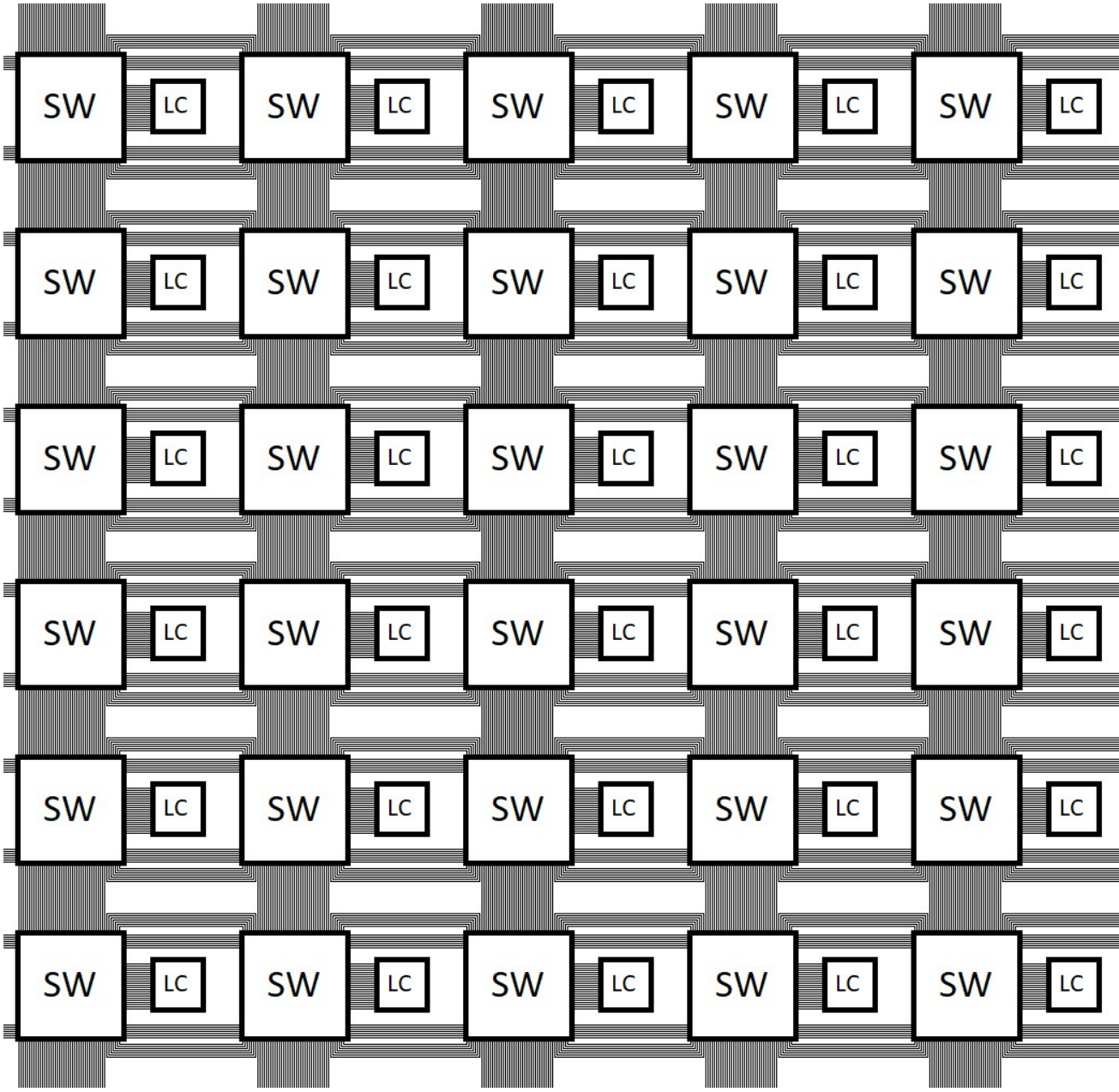
- The pair of configurable logical function coupled with the flip-flop is basically an elementary construction of any FPGA, called **“Logic Cell”**
- Configurable logical function is called LUT, implemented as a simple lookup table, can have between 3 and 6 inputs and 1 or several bits of output.
- Logic Cell can be configured to bypass either LUT or flip-flop, acting as just a LUT on its own or as a D flip-flop
- These CLBs are arranged into a huge arrays with configurable interconnects…

NOTE: FPGAs are not the only programmable logic devices that do exist, but the most capable ones

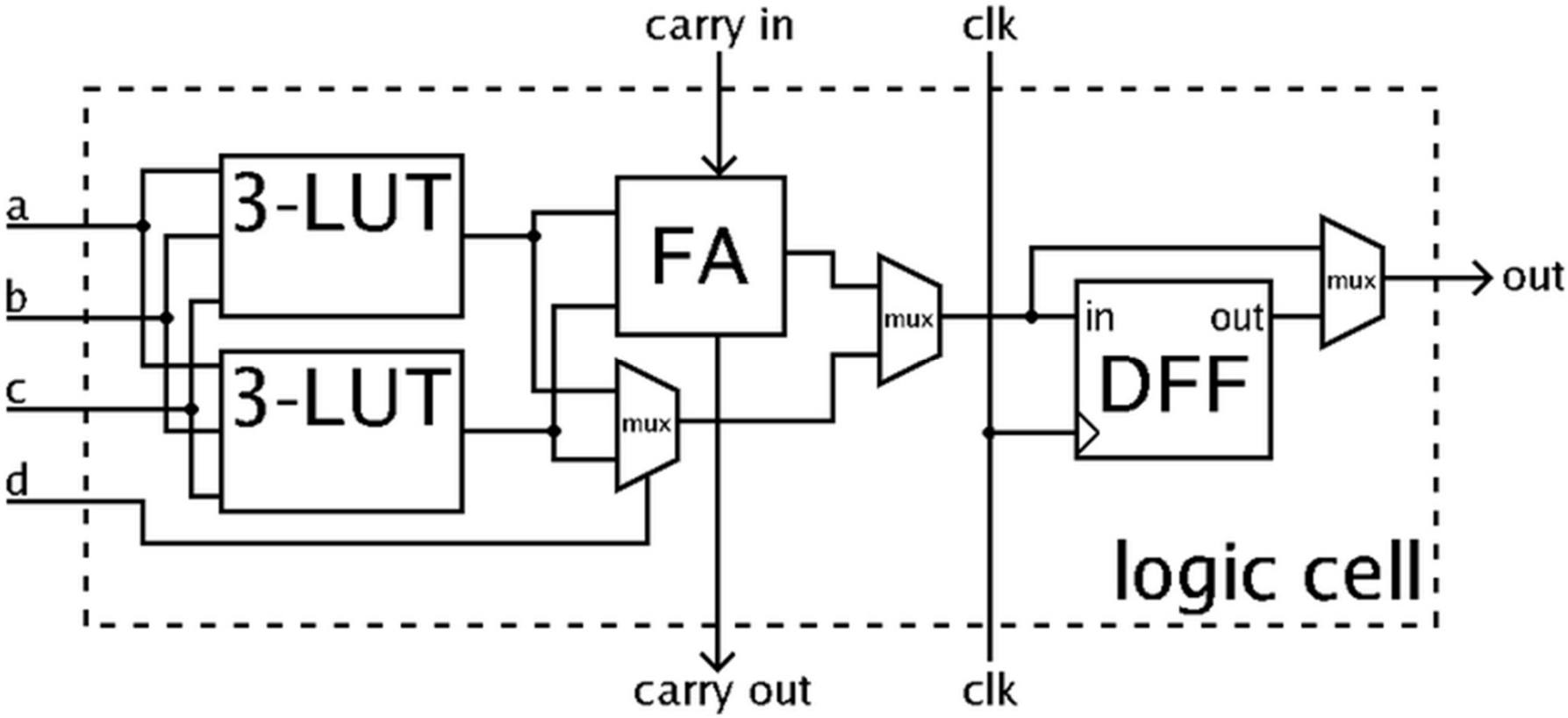
FPGA

A very generic FPGA architecture:

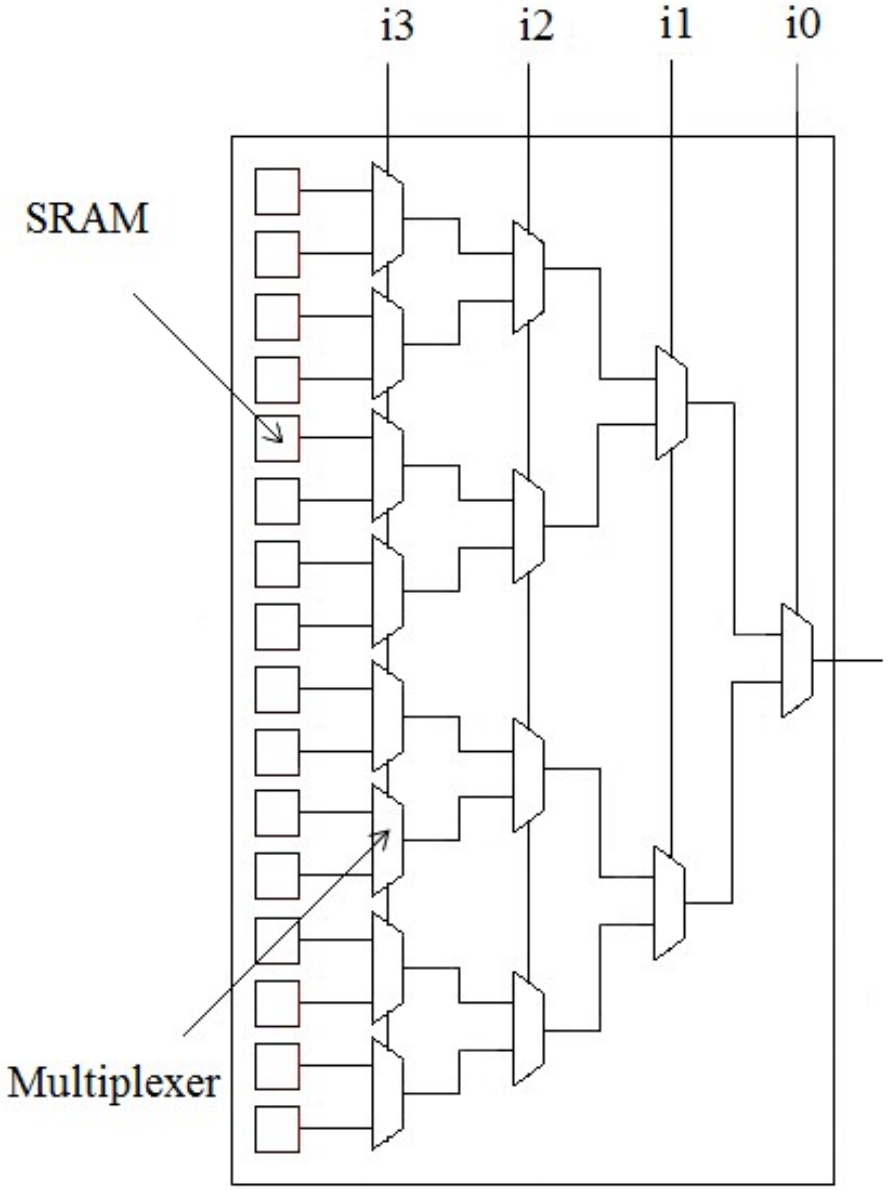
- An ocean (sea, lake, puddle) logic cells (“fabric”) - **LC**
- Connected by via switch matrixes – **SW** - to array of interconnects
- Often has various “silicon” components as well – hardware modules designed for particular function (e.g.: memory controllers, multipliers, CPU cores, etc)



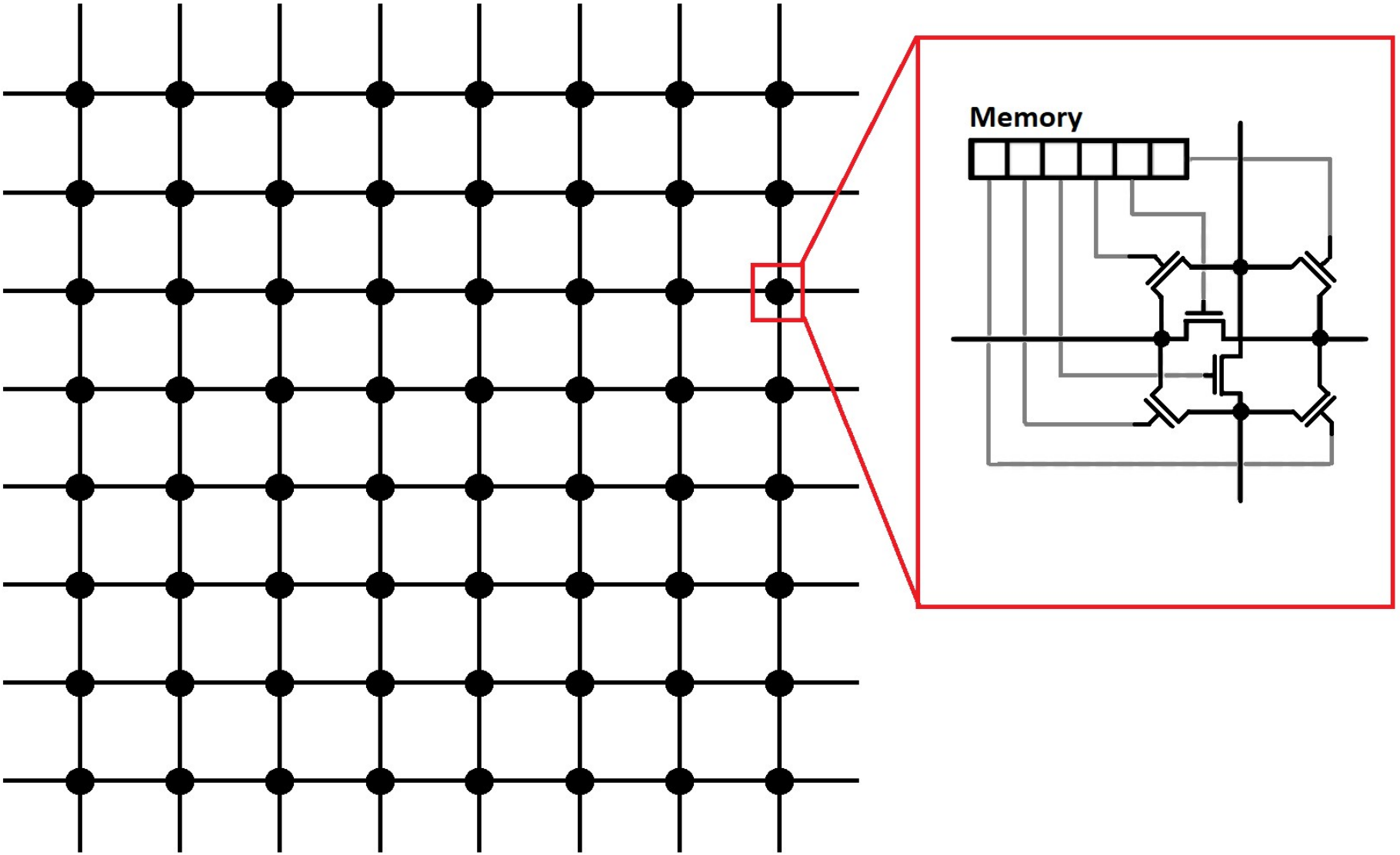
FPGA, example (simplified) logic cell



LUT structure

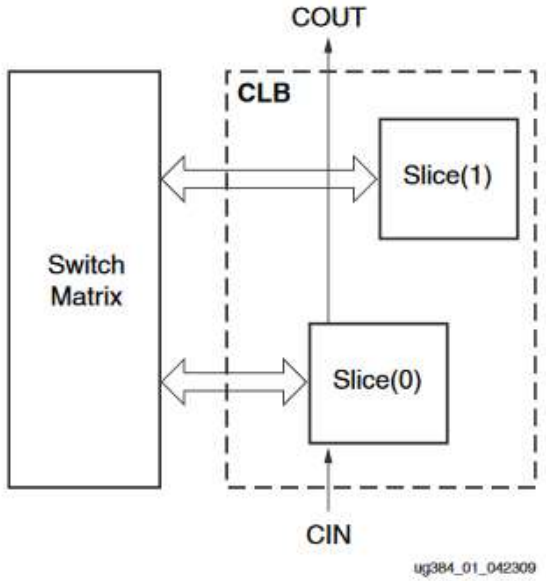


Switch matrix



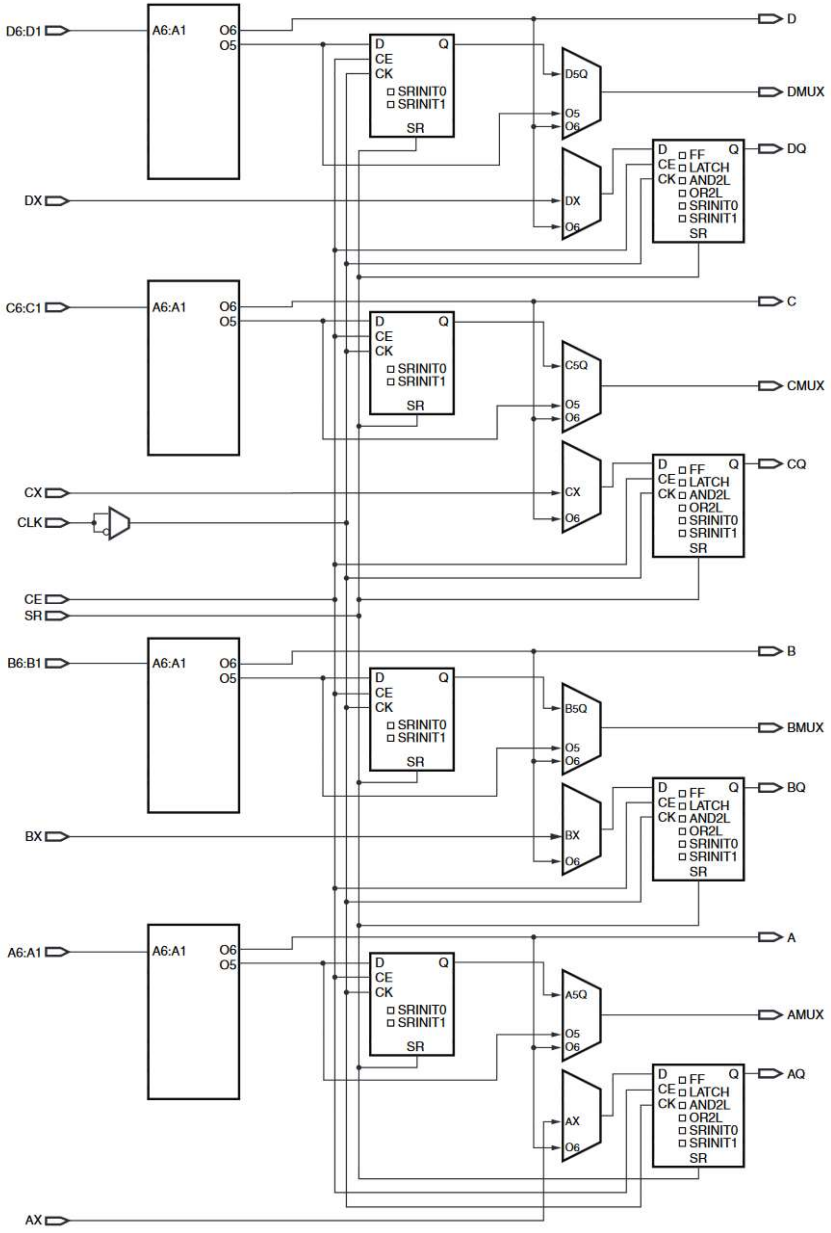
FPGA: LC in a real world, Spartan 6 example

- Xilinx Spartan 6, logic cells are grouped into “Configurable Logic Blocks” – CLB
- Each CLB contains two “slices”
- Each slice is approximately identical and includes actual programmable logic – LUTs and FFs
- There are three slice types on Spartan 6: SLICEM, SLICEL, SLICEX
- Each type of slice contains a set of 6-input LUTs and 8 Flip-Flops
- SLICEL and SLICEM also contain carry logic and wide multiplexers
- SLCIEM also has Distributed RAM and Shift Registers on board.



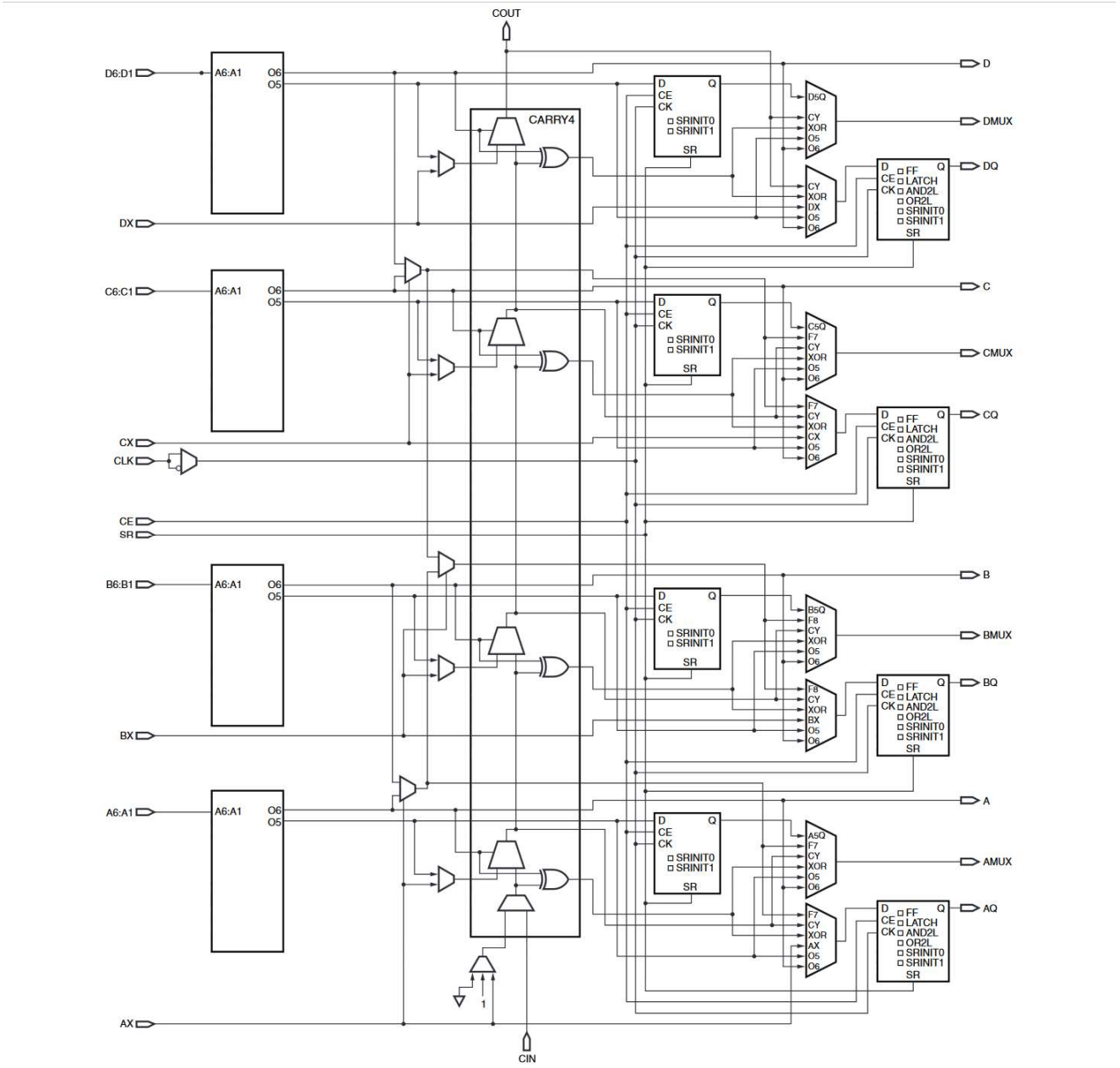
Image, Source: https://www.xilinx.com/support/documentation/user_guides/ug384.pdf

FPGA: Xilinx Spartan 6, SLICEX:

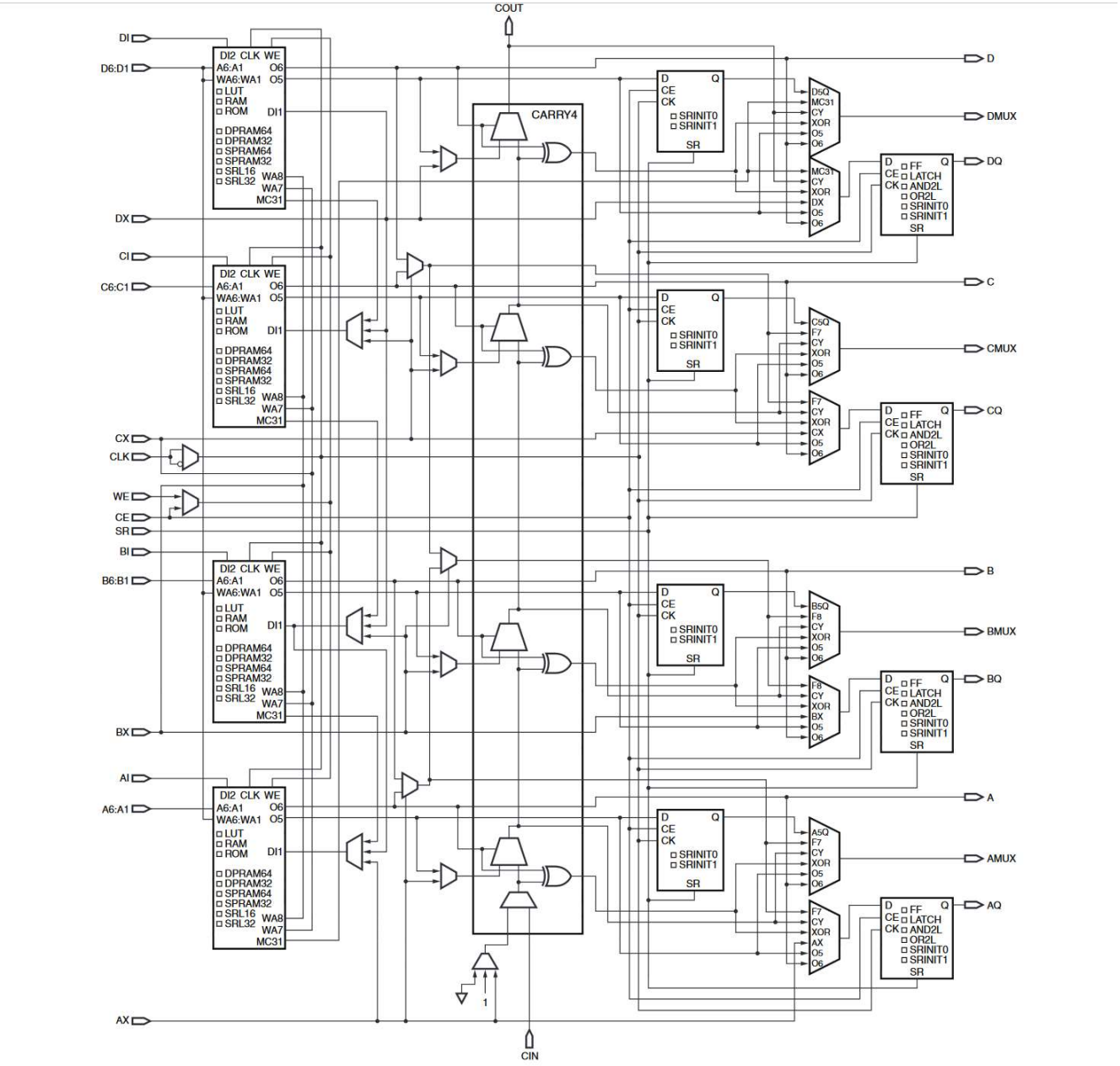


https://www.xilinx.com/support/documentation/user_guides/ug384.pdf

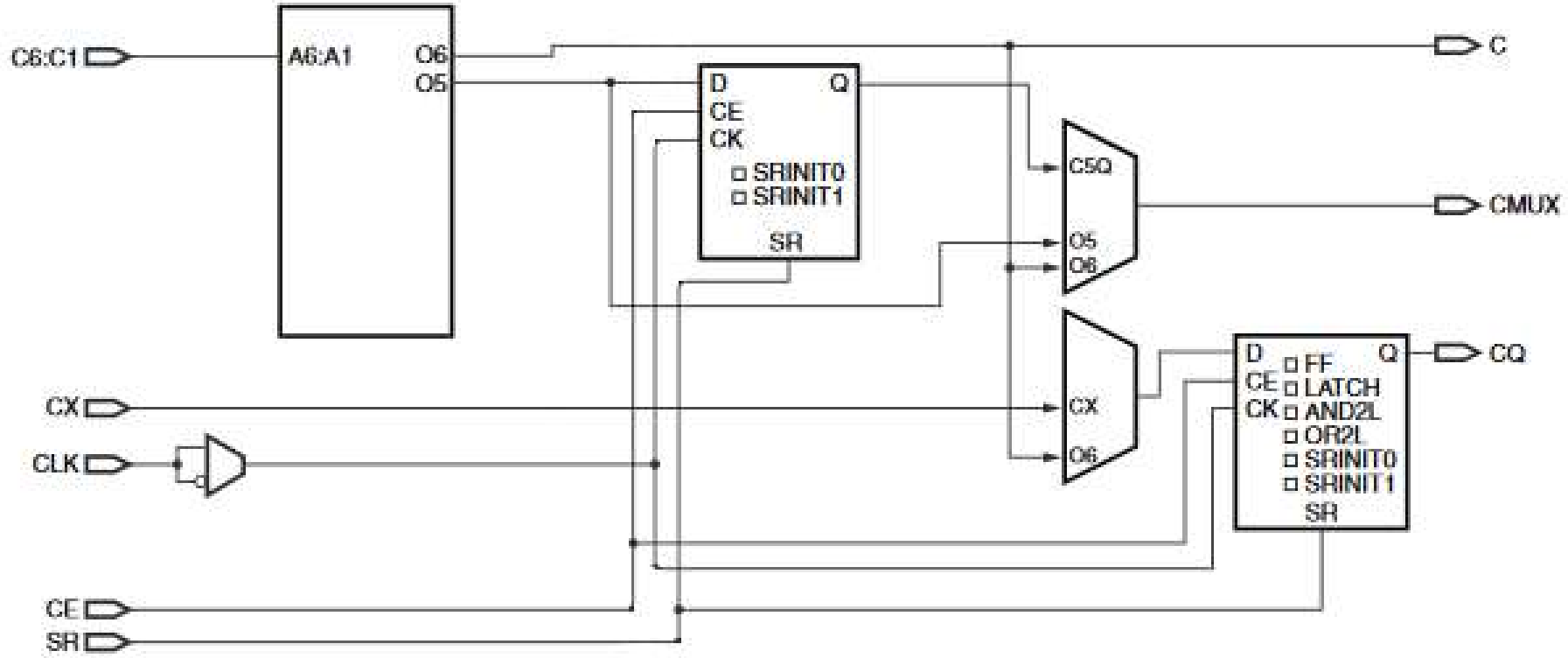
FPGA: Xilinx Spartan 6, SLICEL:



FPGA: Xilinx Spartan 6, SLICEM:

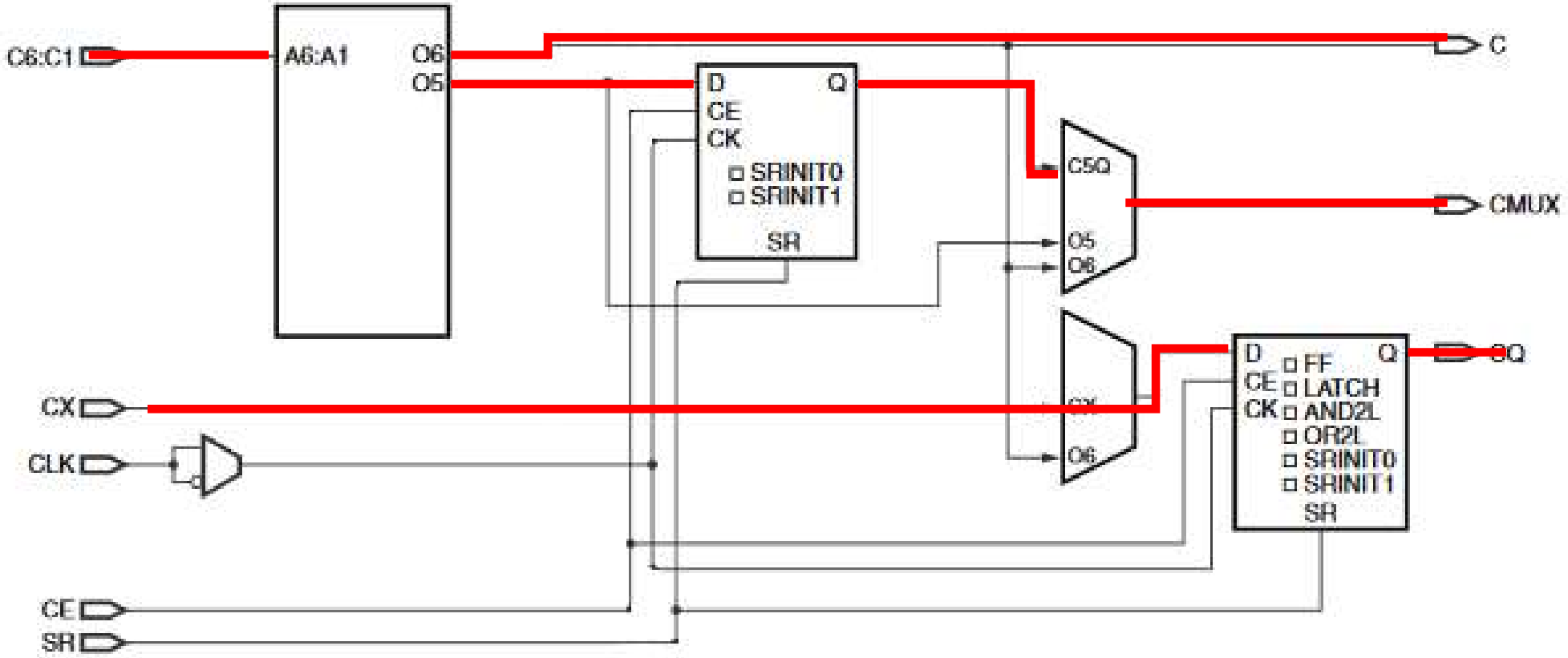


Spartan 6, ¼ of SLICEX, configuration variants



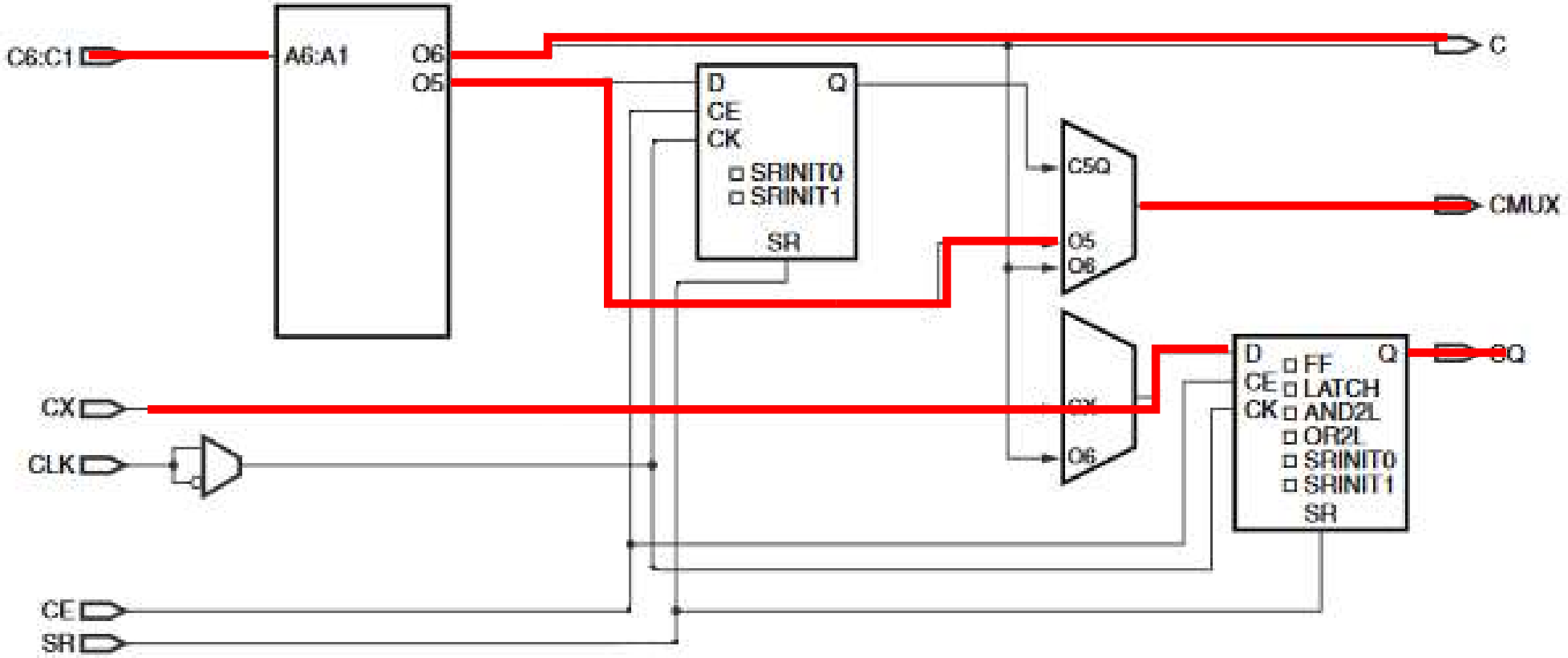
- Two pre-programmable muxes (ignore the clock one), 6 combinations all together
- Let’s review these combinations

Spartan 6, ¼ of SLICEX config., variant 1



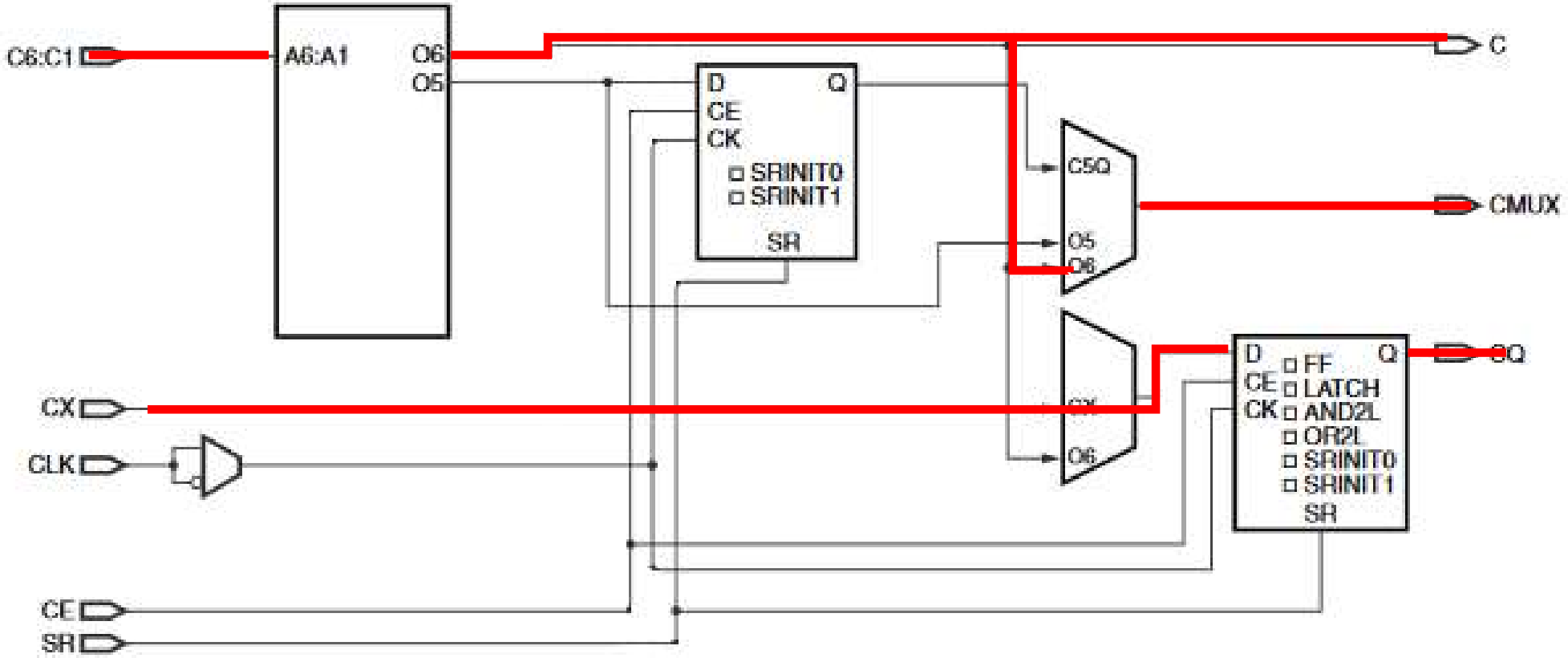
- One isolated FF with CX as input and CQ as output
- LUT output O5 paired with FF into CMUX output
- LUT output O6 connected directly to the slice output C

Spartan 6, ¼ of SLICEX config., variant 2



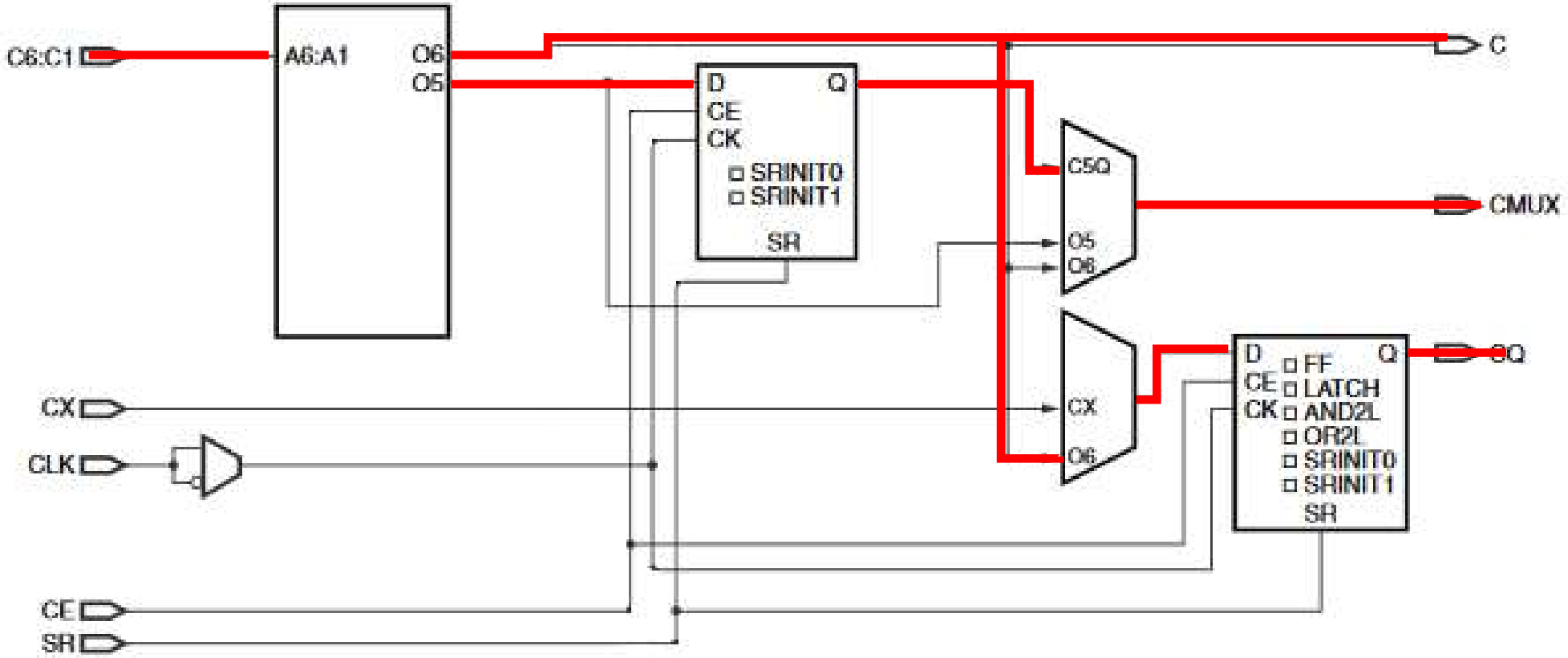
- One isolated FF with CX as input and CQ as output
- LUT output O6 connected directly to output C
- LUT output O5 connected directly to output CMUX
- Another FF is unused

Spartan 6, ¼ of SLICEX config., variant 3



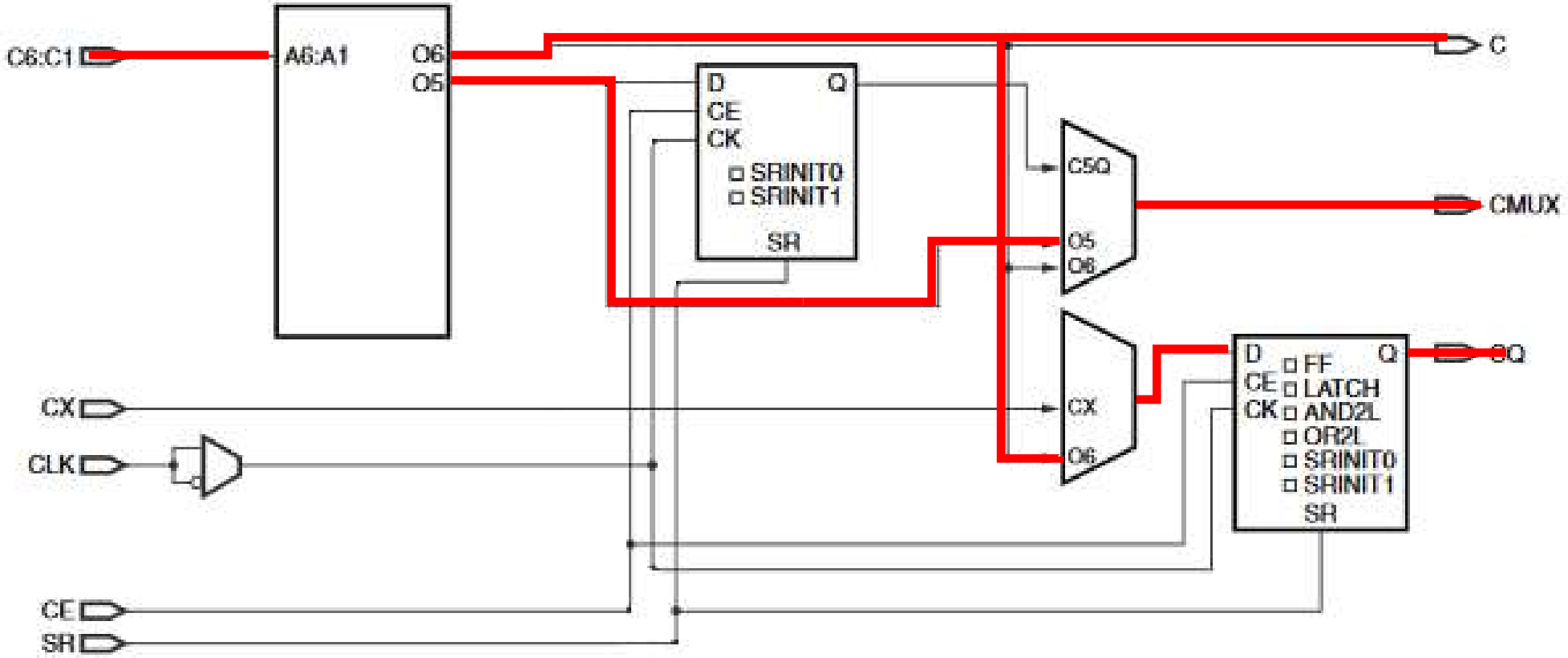
- One isolated FF with CX as input and CQ as output
- LUT output O6 connected directly to both output C and CMUX
- LUT output O5 is not used (perhaps this config. can help with routing in some cases)
- Another FF is unused

Spartan 6, ¼ of SLICEX config., variant 4



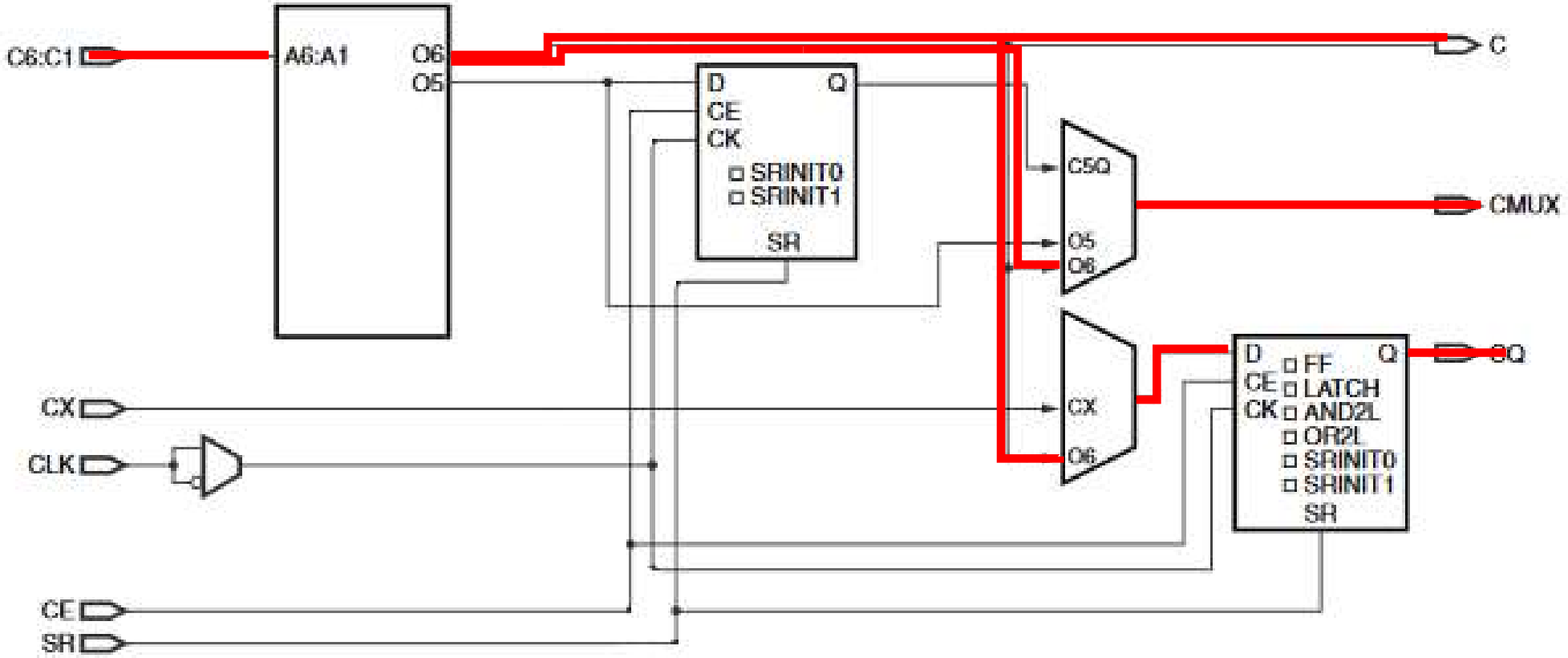
- Both LUT outputs are fed through FF I
- LUT output O5 paired with FF into CMUX output
- LUT output O6 paired with FF into CQ output
- LUT output O6 also connected directly into C (may be unused)

Spartan 6, ¼ of SLICEX config., variant 5



- LUT output O5 is now goes directly into CMUX
- LUT output O6 paired with FF into CQ output
- LUT output O6 also connected directly into C (may be unused)
- One FF is unused

Spartan 6, ¼ of SLICEX config., variant 6



- LUT output O6 paired with FF into CQ output
- LUT output O6 also connected directly into C, and into CMUX
- One FF is unused, as well as LUT O5
- Not sure if this combination is used and if so - how

FPGA interconnects In the real world

This

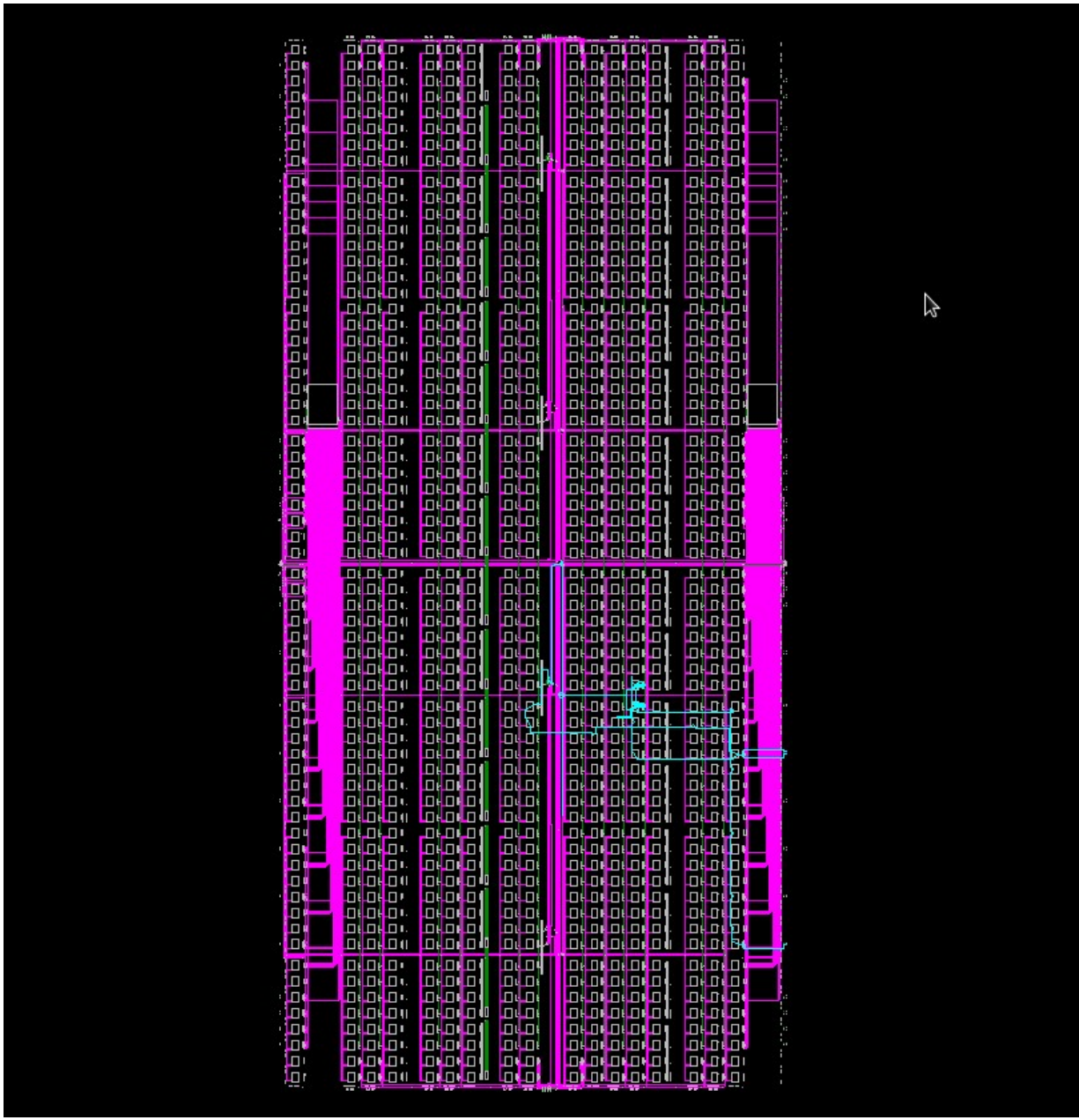
Is

Spartan!

(Spartan 6, XC6SLX9,
1430 slices, 5720 luts, 11440 FFs)

Displaying:

- Long lines (magenta)
- Pin lines (green)
- Switch matrixes (‘big’ squares)
- Slices (tiny dots)
- **Local slice-level lines are not displayed**



FPGA interconnects In the real world

This

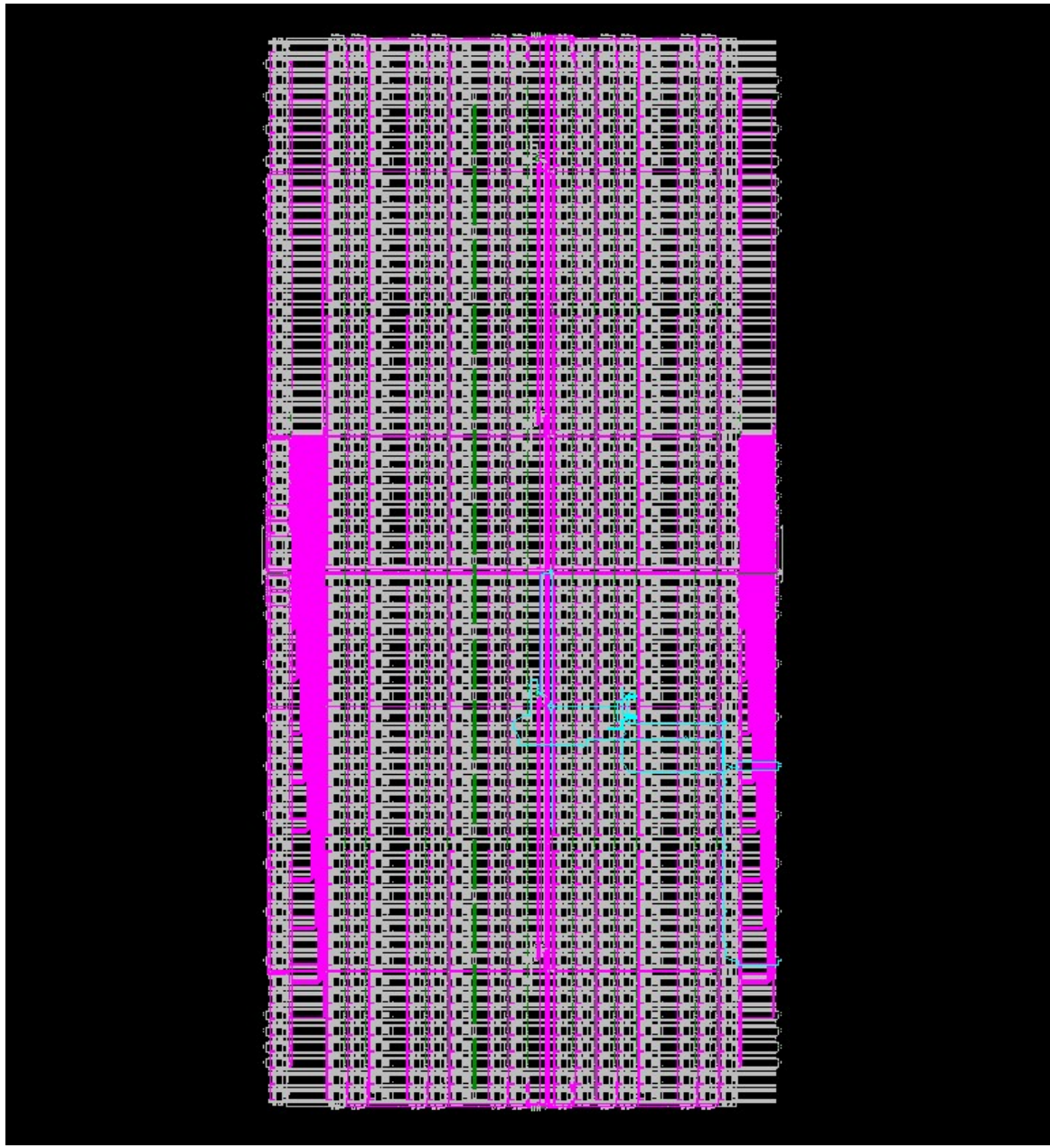
Is

Spartan!

(Spartan 6, XC6SLX9,
1430 slices, 5720 luts, 11440 FFs)

Displaying:

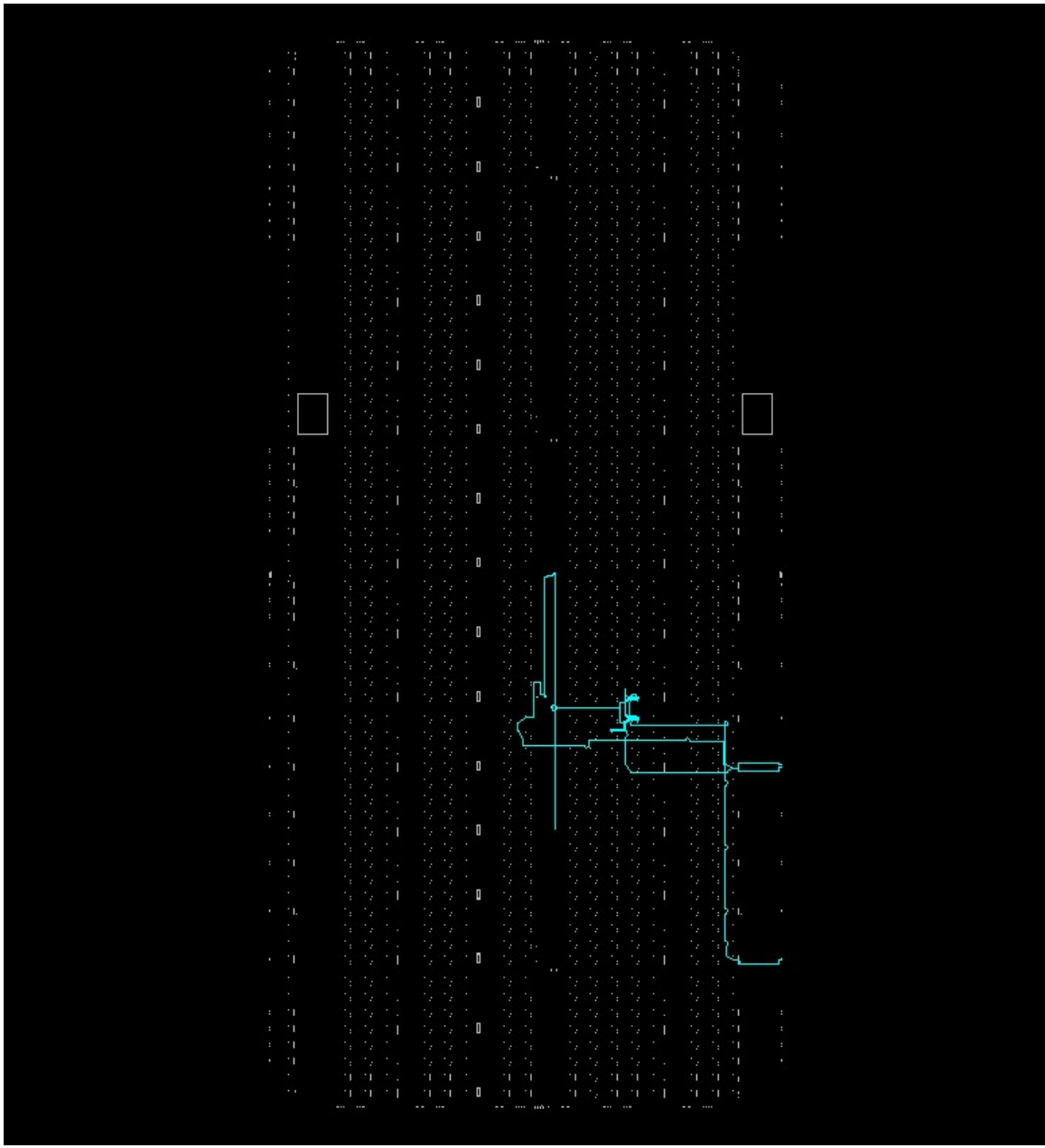
- Long lines (magenta)
- Pin lines (green)
- Switch matrixes (‘big’ squares)
- Slices (tiny dots)
- Local slice-level lines (white mess)



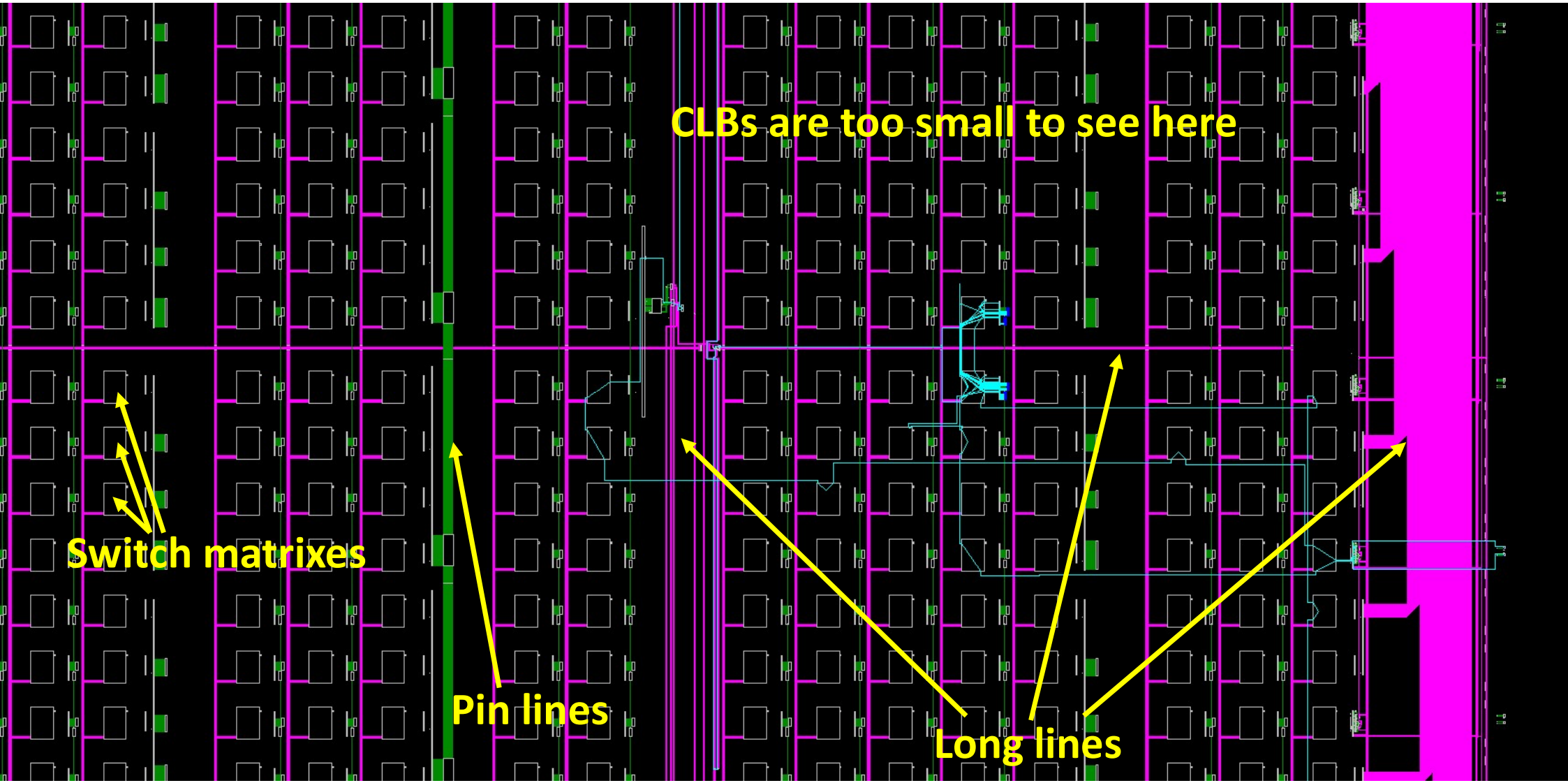
Spartan 6 XC6SLX9 structure, only slices

Only CLBS and the “eth preamble” bitstream routes are displayed

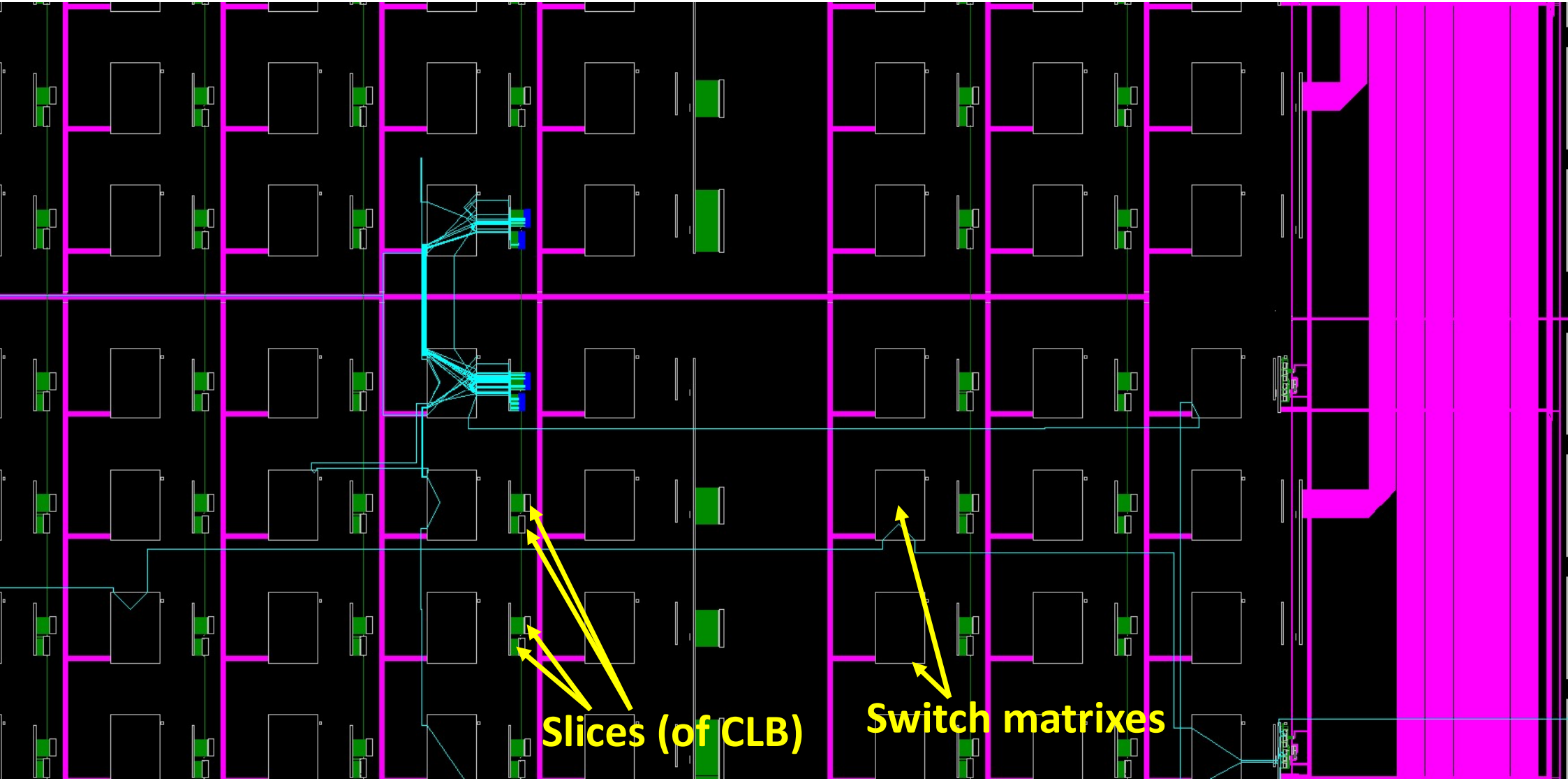
Key point: in the FPGA, most of the space is taken by the configurable routing / interconnects, and not by the logic cells



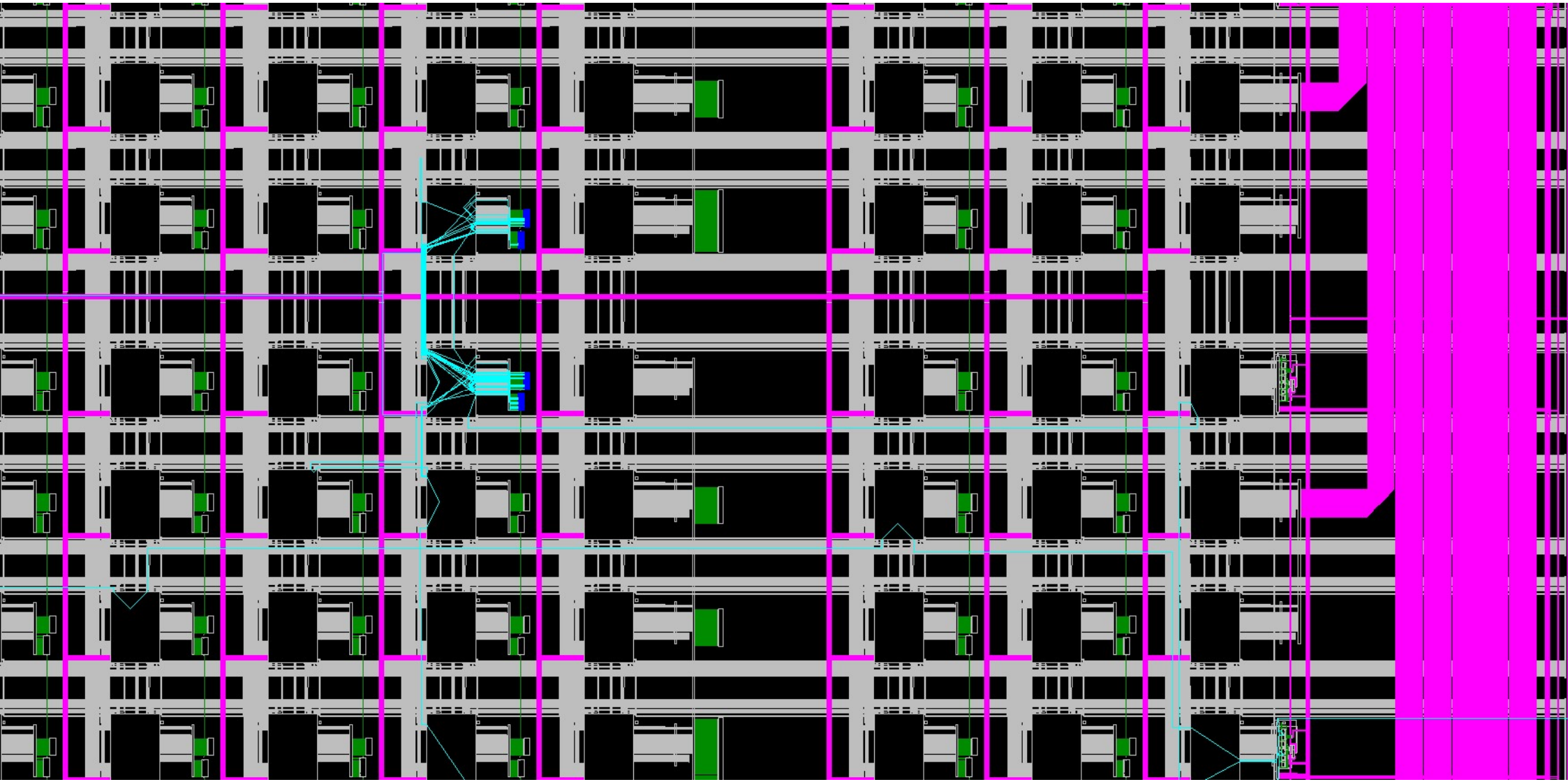
Spartan 6 XC6SLX9, zoom in, local lines not displayed



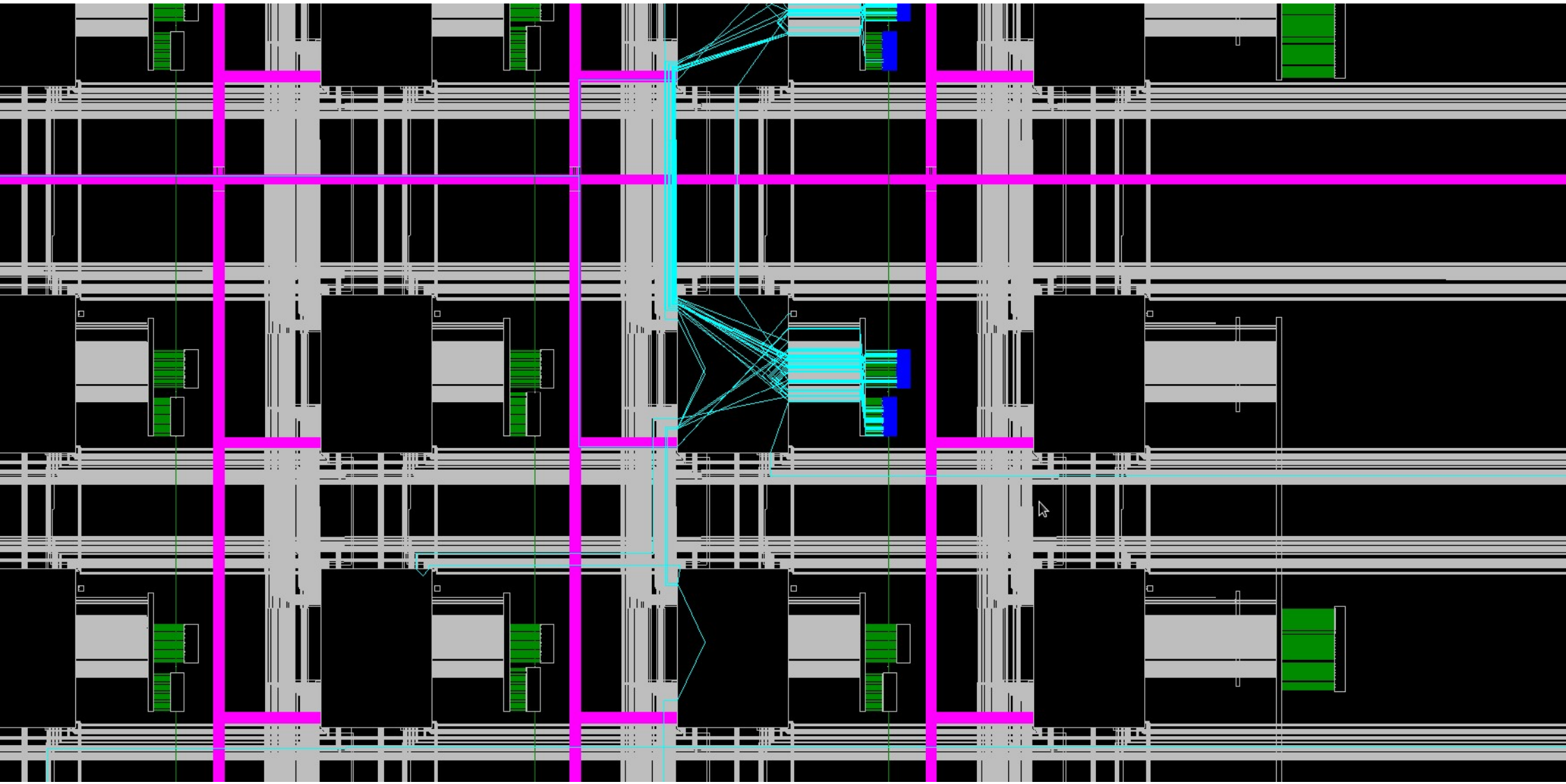
Spartan 6 XC6SLX9, further zoom in, no local lines



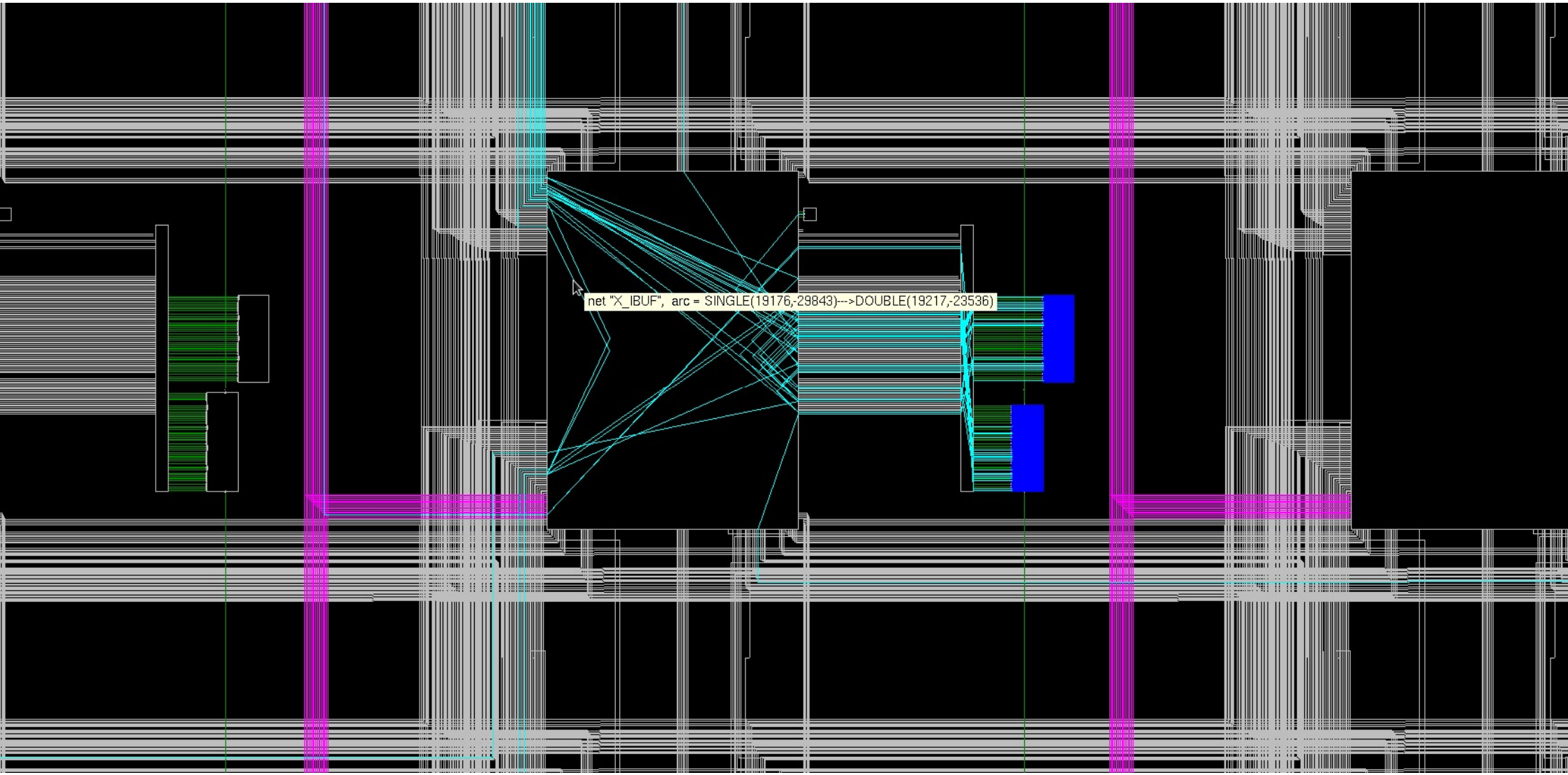
Spartan 6 XC6SLX9, local lines displayed



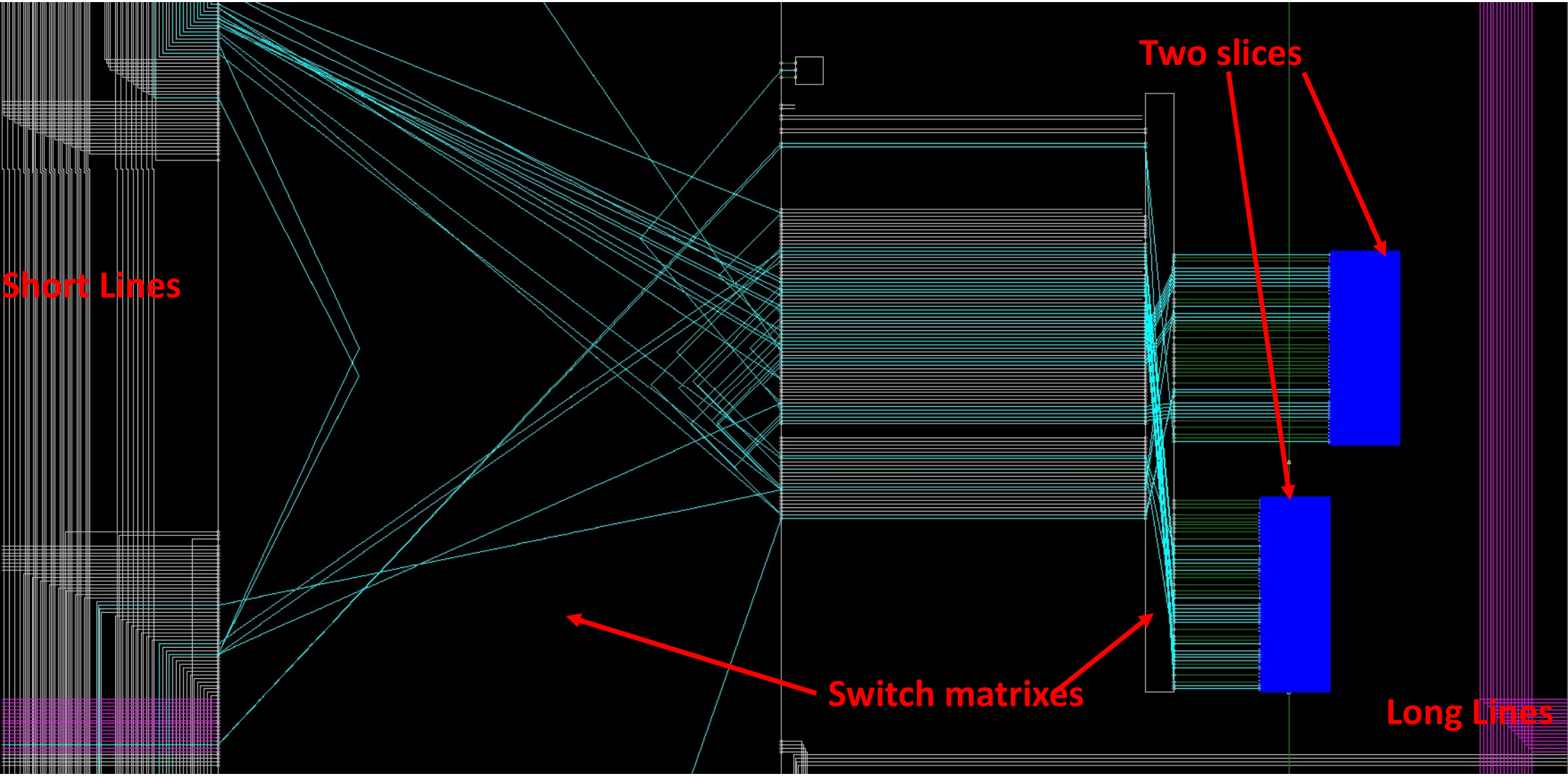
Spartan 6 XC6SLX9, further zoom in

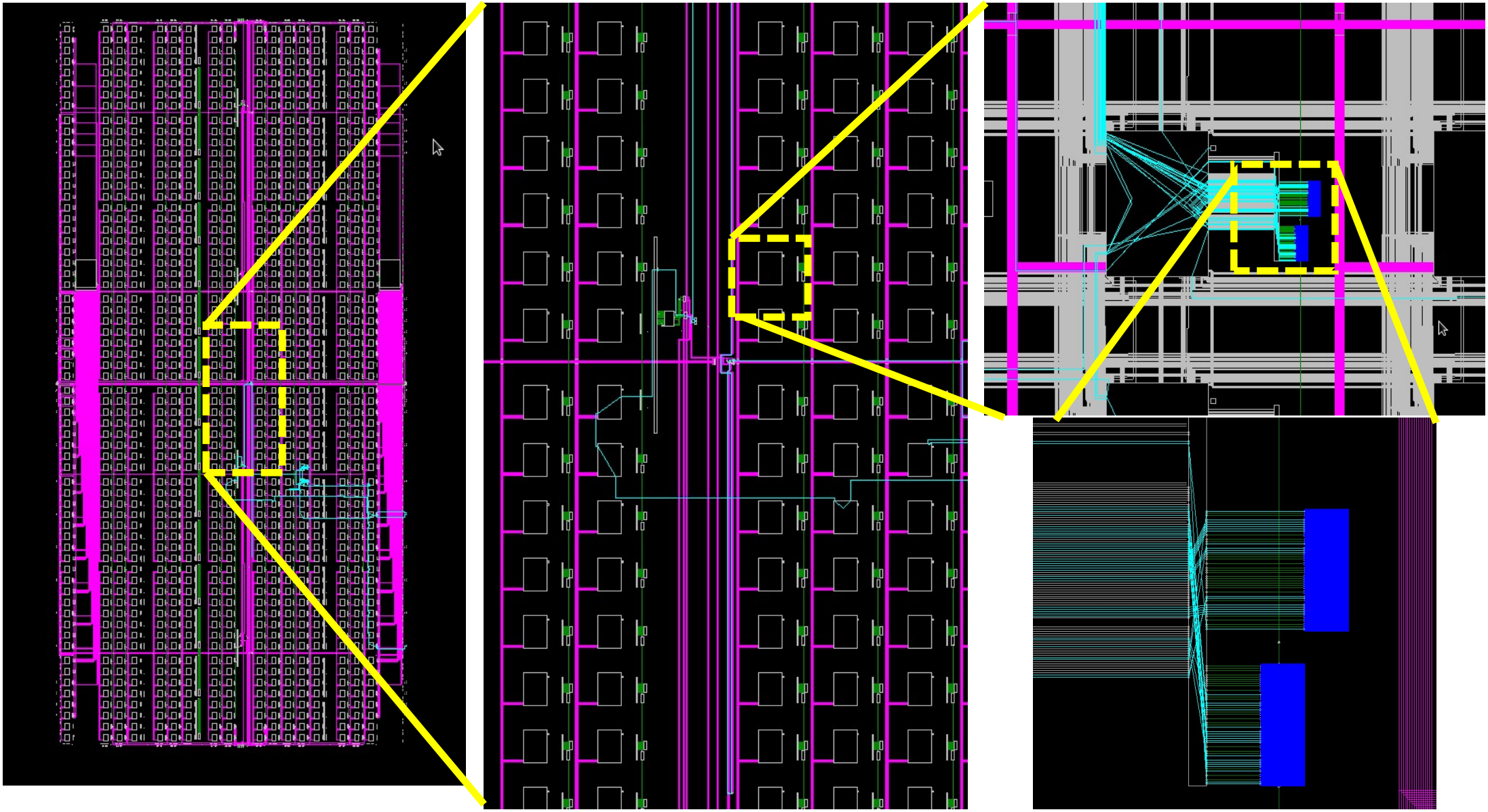


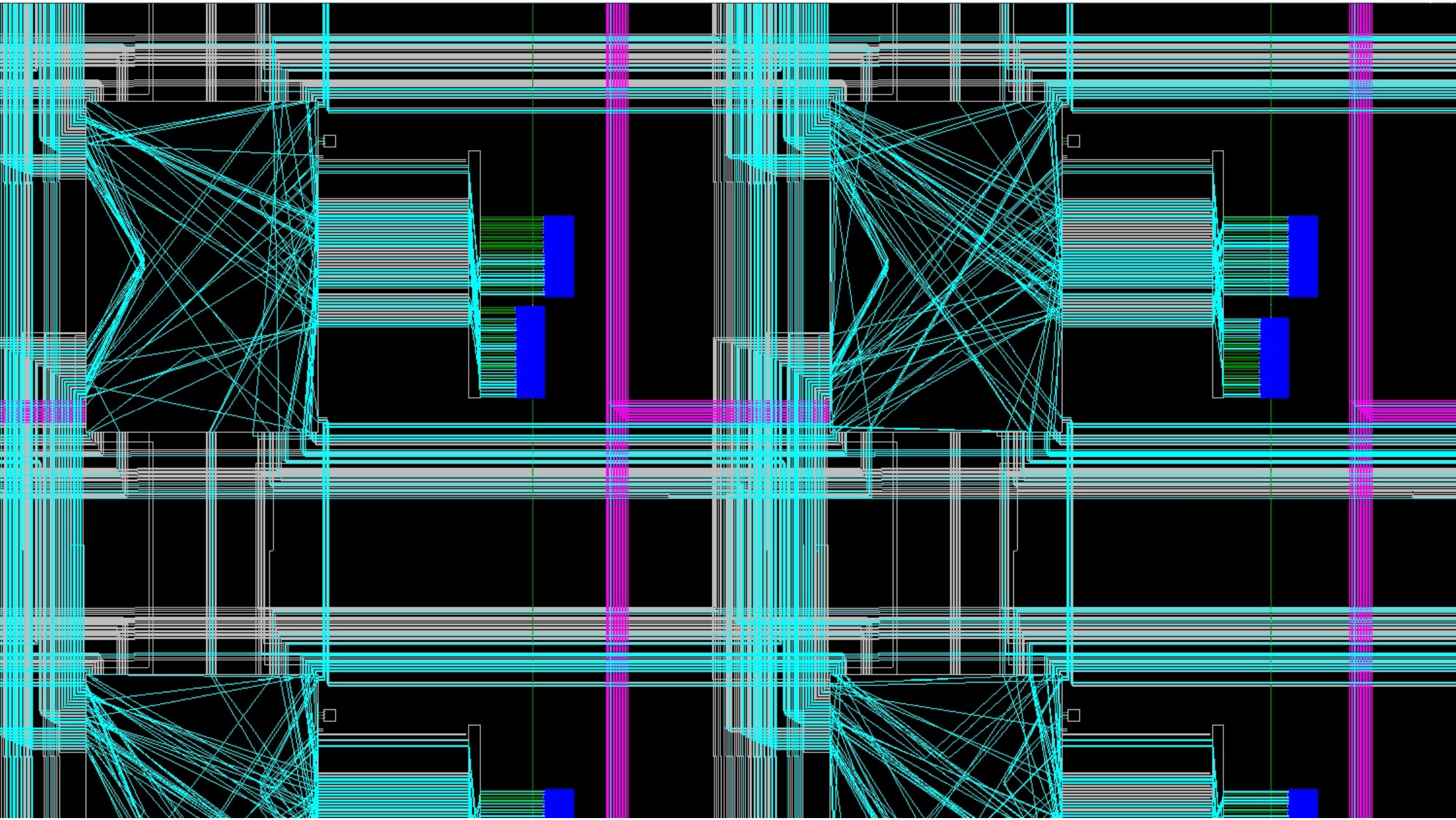
Spartan 6 XC6SLX9, further zoom in



Spartan 6 XC6SLX9, at a max zoom







Time to reveal some magic?

“Final year project”

Simple 8-bit CPU

- Let’s now try to design a very simple 8-bit central processing unit
- Once we understand the basics would carry on with more advanced designs
- Design is based on “Ahmes” 8-bit CPU core (link in the appendix C), but the actual design went far from the original design.

Our simple 8-bit CPU: architecture overview

- **von Neumann memory** model – same memory for instructions and data (as opposite to Harvard commonly used in microcontrollers)
- **accumulator machine**: single general purpose register called "accumulator"
  - Any math/logical/etc instruction implicitly refers to the accumulator as the first argument
  - Many "old days" computers were accumulator machines
    - ENIAC (20 accumulators operating in parallel)
    - PDP-8
    - MOS 6502
    - Intel 4004 and 8008 (modern day x86 has echoes of those days, for example "mul ecx" instruction would multiply registers "ecx" and "eax" and put result into pair of registers "eax" and "edx")
    - Intel 8051 microcontroller (this family is still in production and new derived chips are being produced these days)
  - Thus, "accumulator machine" is not a sort of academic simplification of reality (aka "assume there no other bodies in the whole universe but two mass-points A and B"), but a production-used type of computers

Our simple 8-bit CPU: architecture overview, cont.

- 8-bit address and data buses – can address only 256 bytes (hey, we are only playing here)
- Simple 8-bit ALU with no multiply / divide operations
- One register: 8-bit accumulator
- Flags register: 4 flags
  - **n:** negative – last ALU result was negative
  - **z:** zero – last ALU result was zero
  - **c:** carry – carry/borrow bit from the last ADD / SUB
  - **v:** overflow – last ALU resulted in a signed overflow
- ~50 instructions all together (15 instructions in the “first cut”)
- All instructions are encoded as 2-bytes for simplicity, first byte identifies operation, second byte – an argument

Our simple 8-bit CPU: instruction set

Instruction	Action (C++-like syntax)	Comment
STA addr	mem[addr] = A	Store content of the accumulator into address ‘addr’
LDA addr	A = mem[addr]	Load accumulator with the content of memory address ‘addr’
LDC val	A = val	Load accumulator with direct value from the instruction

Our simple 8-bit CPU: instruction set

Instruction	Action (C++-like syntax)	Comment
ADD addr	A += mem[addr]	
ADDC addr	A += mem[addr] + carry	Add with the carry that was set by the previous operation
SUB addr	A -= mem[addr]	
SUBC addr	A -= mem[addr] + carry	
SUBR addr	A = mem[addr] - A	
SUBRC addr	A = mem[addr] - A - carry	
NEG	A = - A	Treats A as signed, inverts sign (Not a distinct instruction, just another name for "SUBR_V 0")

Our simple 8-bit CPU: instruction set

Instruction	Action (C++-like syntax)	Comment
ADD_V val	A += val	
ADDC_V val	A += val + carry	Add with the carry that was set by the previous operation
SUB_V val	A -= val	
SUBC_V val	A -= val + carry	
SUBR_V val	A = val - A	
SUBRC_V val	A = val - A - carry	

Our simple 8-bit CPU: instruction set, cont.

Instruction	Action (C++-like syntax)	Comment
OR addr	A |= mem[addr]	
AND addr	A &= mem[addr]	
XOR addr	A ^= mem[addr]	
NOT	A = !A	(Not a distinct instruction, just another name for "XOR_V 255")
SHR addr	A = ((unsigned)A) >> mem[addr]	
SHL addr	A = A << mem[addr]	
SHAR addr	A = ((signed)A) >> mem[addr]	
ROL addr	TL;DR;	Rotate left by mem[addr]
ROR addr	TL;DR;	Rotate right mem[addr]
RCL addr	TL;DR;	Rotate through carry left mem[addr]
RCR addr	TL;DR;	Rotate through carry right mem[addr]

Our simple 8-bit CPU: instruction set, cont.

Instruction	Action (C++-like syntax)	Comment
OR_V val	A |= val	
AND_V val	A &= val	
XOR_V val	A ^= val	
SHR_V val	A = ((unsigned)A) >> val	
SHL_V val	A = A << val	
SHAR_V val	A = ((signed)A) >> val	
ROL_V val	A = (A << val) | (A >> (8-val))	Rotate left by val
ROR_V val	A = (A >> val) | (A << (8-val))	Rotate right by val
RCL_V val	TL;DR;	Rotate through carry left by val
RCR_V val	TL;DR;	Rotate through carry right by val

Our simple 8-bit CPU: instruction set

Instruction	Action (C++-like syntax)	Comment
JMP addr	pc = addr	Jump to address addr
JMP_A arg	pc = A + addr	Jump to address A + addr
JN addr	if (n==1) { pc = addr }	Jump to address if negative result (n=1)
JP addr	if (n==0) { pc = addr }	Jump to address if positive result (n=0)
JNZ addr	if (z == 0) { pc = addr }	Jump to address if non zero result (n=0)
JZ addr	if (z == 1) { pc = addr }	Jump to address if zero result (n=1)
JNC addr	if (c == 0) { pc = addr }	Jump to address if carry is not set (c=0)
JC addr	if (c == 1) { pc = addr }	Jump to address if carry is set (c=1)
JNV addr	if (v == 0) { pc = addr }	Jump to address if no overflow (v=0)
JV addr	if (v == 1) { pc = addr }	Jump to address if overflow (v=1)

Our simple 8-bit CPU: instruction set

Instruction	Action (C++-like syntax)	Comment
IN port	A = io_port[port]	Read byte from I/O port
OUT port	io_port[port] = A	Write byte to I/O port
NOP	do {} while(false);	No operation, argument byte is unused
HLT	exit(0);	Full CPU stop until reset, argument byte is unused
SEVENSEGTRANSLATE		Performs table lookup to encode 4-bit number into 7-segment LCD display led configuration displaying given number

Our simple 8-bit CPU: instruction set

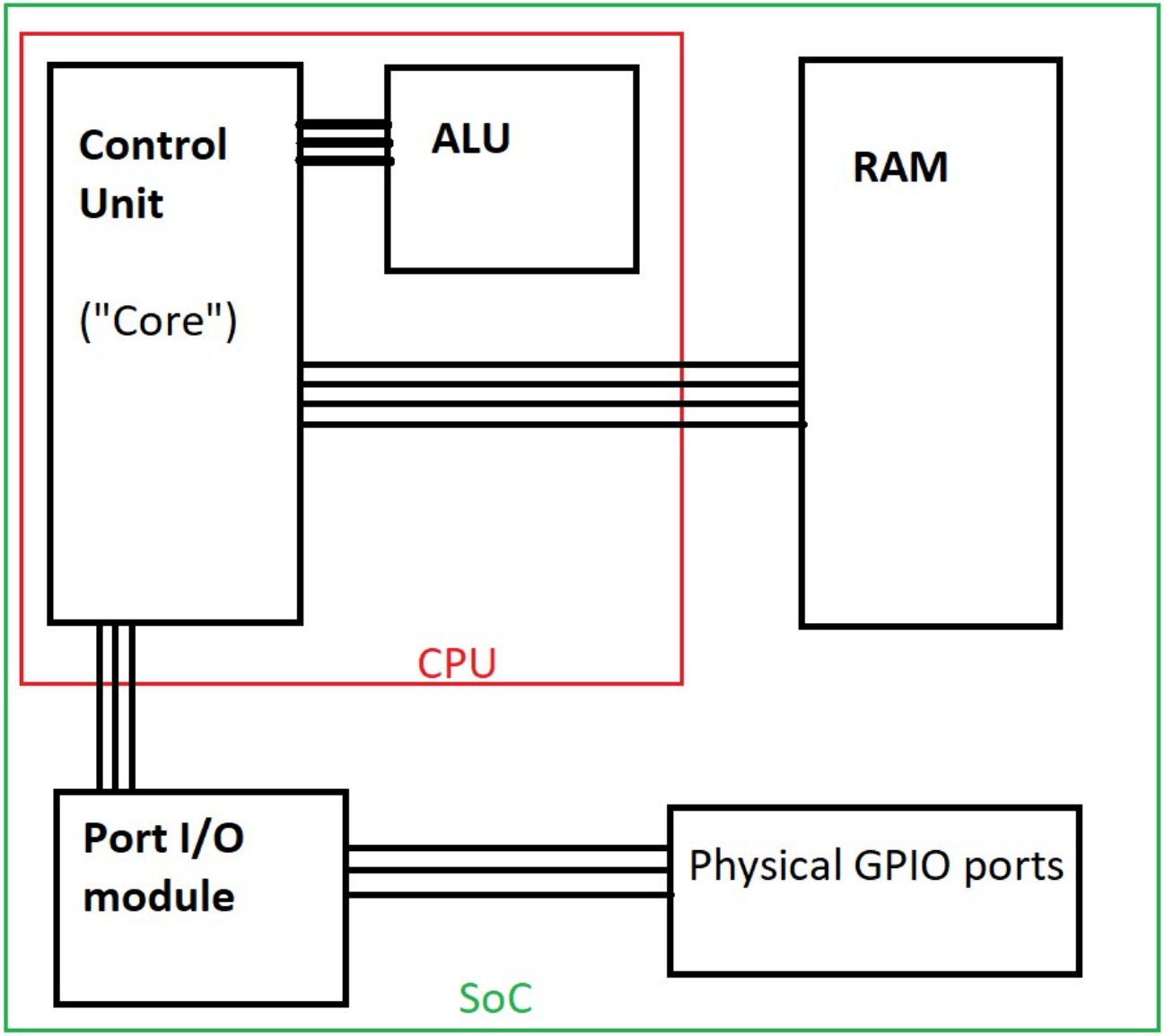
On a first iteration, we would implement even simpler CPU that can handle only the following sub-set of instructions:

- STA / LDA / LDC
- ADD / ADDC / SUB / SUBC
- OR / AND / NOT
- JMP / JNZ / JZ
- HLT / NOP

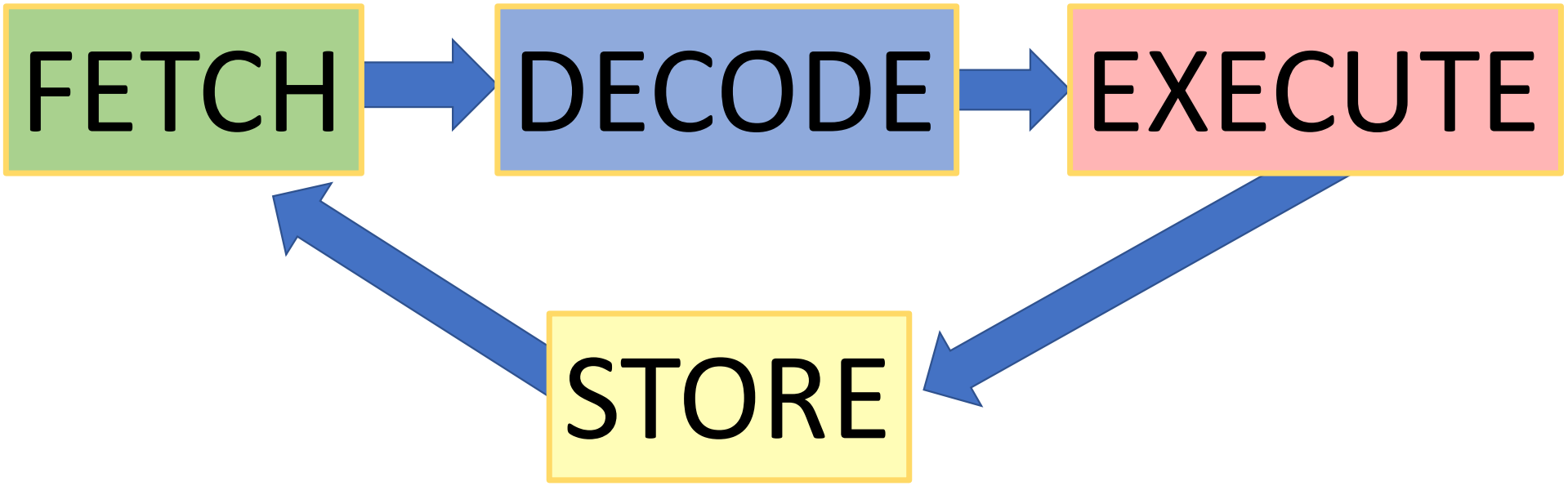
Such minimalistic CPU isn't very practical, but works well for generating Fibonacci Sequence, thus let's call it "Fibonacci Processing Unit"- FibPU

Our VHDL code would also include memory module, so in fact we are building a SoC – System on a Chip

High-Level-Design



Instruction execution cycle



types.vhd

```
package types is
    type ALU_flags is record
        zero            : std_logic;
        carry_out       : std_logic; -- means "borrow out" for sub
    end record ALU_flags;

    type cpu_state_type is (
        FETCH_0,
        FETCH_1,
        DECODE,
        EXECUTE_STA_1,
        EXECUTE_STA_2,
        EXECUTE_LDA_1,
        EXECUTE_LDA_2,
        EXECUTE_LDA_3,
        EXECUTE_LDC_1,
        EXECUTE_ALU_REGMEM_1,
        EXECUTE_ALU_REGMEM_2,
        EXECUTE_ALU_REGMEM_3,
        EXECUTE_JMP,
        STORE,
        STOP
        );

end package types;
```

opcodes.vhd

```
package opcodes is
    type alu_opcode_type is (
            ALU_ADD,        -- add
            ALU_SUB,        -- sub Left-Right
            ALU_OR,         -- bitwise OR
            ALU_AND,        -- .. AND
            ALU_NOT,        -- .. NOT
            ALU_NOP         -- no operation
        );

    -- load and store instructions, prefix 011
    constant OP_STA         : std_logic_vector(7 downto 0):="01100000";
    constant OP_LDA         : std_logic_vector(7 downto 0):="01101000";
    constant OP_LDC         : std_logic_vector(7 downto 0):="01110000";
    -- math instructions, prefix 010
    constant OP_ADD         : std_logic_vector(7 downto 0):="01000000";
    constant OP_ADDC        : std_logic_vector(7 downto 0):="01000100";
    constant OP_SUB         : std_logic_vector(7 downto 0):="01001000";
    constant OP_SUBC        : std_logic_vector(7 downto 0):="01001100";
    -- logical and bit instructions, prefix 11
    constant OP_OR          : std_logic_vector(7 downto 0):="11000000";
    constant OP_AND         : std_logic_vector(7 downto 0):="11001000";
    constant OP_NOT         : std_logic_vector(7 downto 0):="11010000";
    -- branching instructions, prefix 1010
    constant OP_JMP         : std_logic_vector(7 downto 0):="10100000";
    constant OP_JNZ         : std_logic_vector(7 downto 0):="10100100";
    constant OP_JZ          : std_logic_vector(7 downto 0):="10100101";
    -- special instructions, prefix 10111
    constant OP_HLT         : std_logic_vector(7 downto 0):="10111000";
    constant OP_NOP         : std_logic_vector(7 downto 0):="10111001";
end package opcodes;
```

RAM – memory.vhd

```
entity memory is
    port
    (
        clk          : in std_logic;
        address_bus  : in std_logic_vector(7 downto 0);
        data_write   : in std_logic_vector(7 downto 0);
        data_read    : out std_logic_vector(7 downto 0);
        mem_write    : in std_logic;
        rst          : in std_logic
    );
end memory;
```

RAM – memory.vhd

```
architecture rtl of memory is
type mem_type is array (0 to 255) of std_logic_vector(7 downto 0);
signal mem: mem_type := (
    -- some pre-loaded code to be added later
);
begin
    process (clk, rst, mem_write, address_bus, data_write)
    begin
        if rising_edge(clk) then
            if mem_write = '1' then
                mem(to_integer(unsigned(address_bus))) <= data_write;
                data_read <= data_write;
            else
                data_read <= mem(to_integer(unsigned(address_bus)));
            end if;
        end if;
    end process;
end rtl;
```

pio.vhd – does nothing in the first cut (just a stub)

```
entity pio is
    port (
        clk             : in std_logic;
        address         : in std_logic_vector(7 downto 0);
        data_w          : in std_logic_vector(7 downto 0); -- data entering IO port
        data_r          : out std_logic_vector(7 downto 0);
        write_enable    : in std_logic;
        read_enable     : in std_logic;
        io_ready        : out std_logic;

        in_port_0       : in std_logic_vector (7 downto 0); -- dp switches
        in_port_1       : in std_logic_vector (7 downto 0); -- push btns
        in_port_2       : in std_logic_vector (7 downto 0); -- pin header 6
        in_port_3       : in std_logic_vector (7 downto 0); -- pin header 7

        out_port_4      : out std_logic_vector (7 downto 0); -- individual leds
        out_port_5      : out std_logic_vector (7 downto 0); -- 7-segment digits
        out_port_6      : out std_logic_vector (7 downto 0); -- 7-segment enable signals
        out_port_7      : out std_logic_vector (7 downto 0); -- pin header 8
        out_port_8      : out std_logic_vector (7 downto 0) -- pin header 9
    );
end pio;
```

alu.vhd – getting closer…

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_unsigned.all ;

use work.opcodes.all;
use work.types.all;

entity ALU is
    port
    (
        operation           : in alu_opcode_type;
        left_arg            : in std_logic_vector(7 downto 0);
        right_arg           : in std_logic_vector(7 downto 0);
        carry_in            : in std_logic;
        result              : out std_logic_vector(7 downto 0);
        flags               : out ALU_flags
    );
end ALU;
```

alu.vhd – architecture

```
architecture behaviour of ALU is
begin
    process (left_arg, right_arg, operation, carry_in)
        variable temp : std_logic_vector(8 downto 0);
    begin
        case operation is
            when ALU_ADD =>
                temp := ('0' & left_arg) + ('0' & right_arg) + ("00000000" & carry_in);
            when ALU_SUB =>
                temp := ('0'&left_arg) - ('0'&right_arg) - ("00000000" & carry_in);
            when ALU_OR =>
                temp := ('0' & left_arg) or ('0' & right_arg);
            when ALU_AND =>
                temp := ('0' & left_arg) and ('0' & right_arg);
            when ALU_NOT =>
                temp := not ('0' & left_arg);
            when others =>
                temp := "000000000";
        end case;

        if temp(7 downto 0) = "00000000" then  flags.zero <= '1';
        else  flags.zero <= '0';
        end if;

        flags.carry_out <= temp(8);
        result <= temp(7 downto 0);
    end process;
end behaviour;
```

core.vhd – the control unit with registers

```
entity core is
    port
    (
        clk                 : in std_logic;
        reset               : in std_logic;
        error               : out std_logic;

        address_bus         : out std_logic_vector(7 downto 0);
        data_in             : in std_logic_vector(7 downto 0);
        data_out            : out std_logic_vector(7 downto 0);
        mem_write           : out std_logic;

        alu_opcode          : out alu_opcode_type;
        alu_carry_in        : out std_logic;
        alu_left            : out std_logic_vector(7 downto 0);
        alu_right           : out std_logic_vector(7 downto 0);
        alu_result          : in std_logic_vector(7 downto 0);
        alu_flags_in        : in ALU_flags;

        debug_program_counter       : out std_logic_vector(7 downto 0);
        debug_accumulator           : out std_logic_vector(7 downto 0);
        debug_instruction_code      : out std_logic_vector(7 downto 0);
        debug_cpu_state             : out cpu_state_type
    );
end core;
```

core.vhd

```
architecture ahmes of core is

    signal cpu_state                : cpu_state_type;
    signal program_counter          : std_logic_vector(7 downto 0);
    signal accumulator              : std_logic_vector(7 downto 0);
    signal flags                    : ALU_flags := (others => '0');

    signal instruction_code         : std_logic_vector(7 downto 0);

begin

    debug_program_counter <= program_counter;
    debug_accumulator     <= accumulator;
    debug_instruction_code <=   instruction_code;
    debug_cpu_state <= cpu_state;
```

core.vhd

```
process (clk, reset, program_counter, accumulator)
begin
    if reset = '1'
    then
        cpu_state <= FETCH_0;
        program_counter <= "00000000";
        mem_write <= '0';
        address_bus <= "00000000";
        data_out <= "00000000";
        alu_opcode <= ALU_NOP;

        flags <= (others => '0');
        error <= '0';

    elsif rising_edge(clk)
    then
        case cpu_state is
```

core.vhd

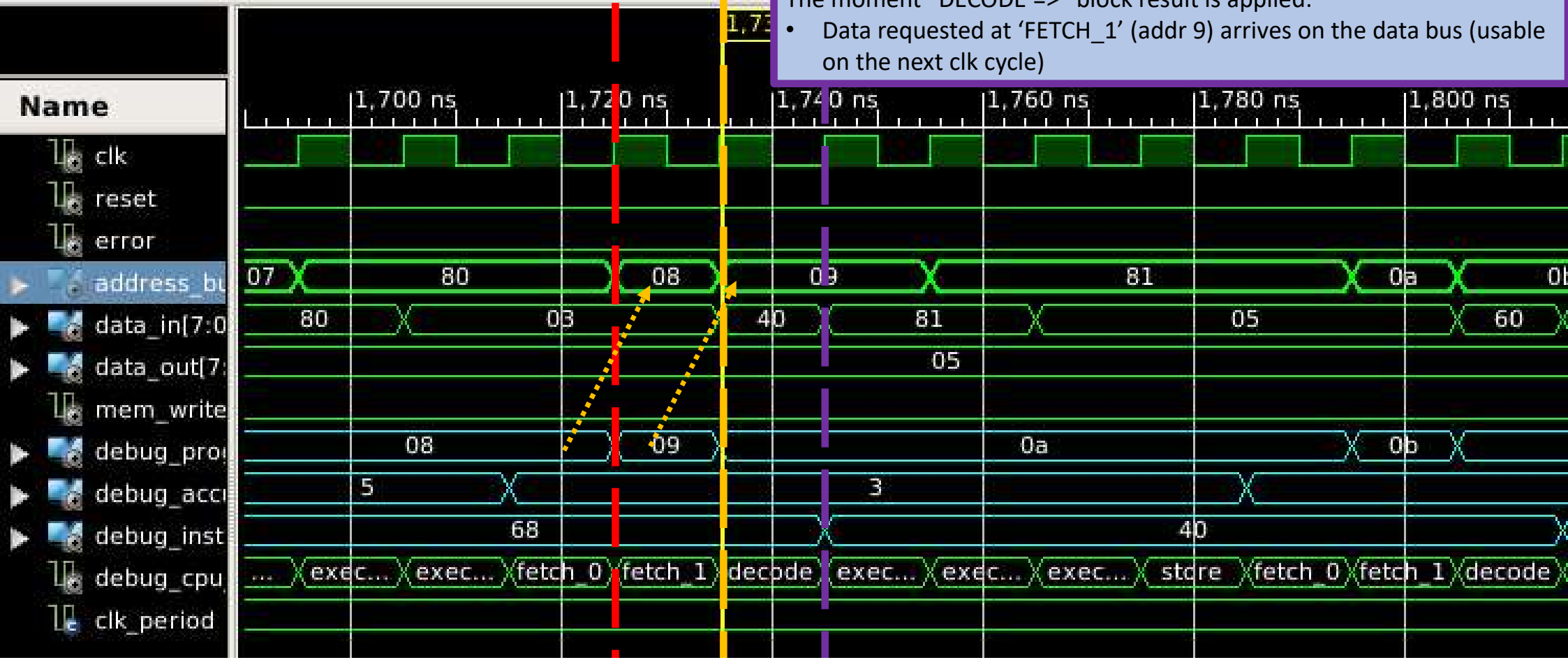
```
elsif rising_edge(clk)
then
    case cpu_state is
        when STOP =>
            cpu_state <= STOP;

        when FETCH_0 =>
            -- set instruction address on the memory bus
            address_bus <= program_counter;
            program_counter <= program_counter + 1;
            cpu_state <= FETCH_1;

        when FETCH_1 =>
            -- set instruction address on the memory bus,
            -- data from the FETCH_0 is still travelling through FF-s
            address_bus <= program_counter;
            program_counter <= program_counter + 1;
            cpu_state <= DECODE;
```

RAM access sequence

The moment “FETCH_0=>” block is result is applied:
- Address bus set to ‘8’ (PC)
- PC incremented by 1, now 9

The moment “FETCH_1 =>” block result is applied:
- Address bus set to ‘9’
- PC incremented by 1, now 9 (0xa)
- Data requested at ‘FETCH_0’ (addr 8) arrives on the data bus (only usable on the next clk cycle)

The moment “DECODE =>” block result is applied:
- Data requested at ‘FETCH_1’ (addr 9) arrives on the data bus (usable on the next clk cycle)



core.vhd

```
when DECODE =>
    instruction_code <= data_in;

    case data_in is
        when OP_NOP =>
            cpu_state <= FETCH_0;

        when OP_STA =>
            cpu_state <= EXECUTE_STA_1;

        when OP_LDA =>
            cpu_state <= EXECUTE_LDA_1;

        when OP_LDC =>
            cpu_state <= EXECUTE_LDC_1;

        when OP_ADD =>
            alu_carry_in <= '0';
            alu_opcode <= ALU_ADD;
            cpu_state <= EXECUTE_ALU_REGMEM_1;

        when OP_ADDC =>
            alu_carry_in <= flags.carry_out;
            alu_opcode <= ALU_ADD;
            cpu_state <= EXECUTE_ALU_REGMEM_1;
```

NOTE:
This code would generate Mealy FSM, since outputs - control selectors for various signals depend on both current state (DECODE) and current FSM input (data_in).

This should be OK though, as all inputs are internal for our circuit and data_in is synchronized to the same clock.

core.vhd

```
when EXECUTE_STA_1 =>
    address_bus <= data_in;
    data_out <= accumulator;
    mem_write <= '1';
    cpu_state <= EXECUTE_STA_2;
when EXECUTE_STA_2 =>
    mem_write <= '0';
    cpu_state <= FETCH_0;

when EXECUTE_LDA_1 =>
    address_bus <= data_in;
    cpu_state <= EXECUTE_LDA_2;
when EXECUTE_LDA_2 =>
    cpu_state <= EXECUTE_LDA_3;
when EXECUTE_LDA_3 =>
    accumulator <= data_in;
    cpu_state <= FETCH_0;

when EXECUTE_LDC_1 =>
    accumulator <= data_in;
    cpu_state <= FETCH_0;
```

core.vhd

```
when EXECUTE_ALU_REGMEM_1 =>
    address_bus <= data_in;
    cpu_state <= EXECUTE_ALU_REGMEM_2;

when EXECUTE_ALU_REGMEM_2 =>
    cpu_state <= EXECUTE_ALU_REGMEM_3;

when EXECUTE_ALU_REGMEM_3 =>
    alu_left <= accumulator;
    alu_right <= data_in;
    cpu_state <= STORE;
```

core.vhd

```
when STORE =>
    accumulator <= alu_result;
    flags <= alu_flags_in;
    cpu_state <= FETCH_0;
```

core.vhd

Back to ‘DECODE’:

```
when OP_SUB =>
    alu_carry_in <= '0';
    alu_opcode <= ALU_SUB;
    cpu_state <= EXECUTE_ALU_REGMEM_1;

when OP_SUBC =>
    alu_carry_in <= flags.carry_out;
    alu_opcode <= ALU_SUB;
    cpu_state <= EXECUTE_ALU_REGMEM_1;

when OP_OR =>
    alu_opcode <= ALU_OR;
    cpu_state <= EXECUTE_ALU_REGMEM_1;

when OP_AND =>
    alu_opcode <= ALU_AND;
    cpu_state <= EXECUTE_ALU_REGMEM_1;
```

core.vhd

```
when OP_NOT =>
    alu_left <= accumulator;
    alu_opcode <= ALU_NOT;
    cpu_state <= STORE;

when OP_JMP =>
    cpu_state <= EXECUTE_JMP;

when OP_JZ | OP_JNZ =>
    if flags.zero = data_in(0) then
        cpu_state <= EXECUTE_JMP;
    else
        cpu_state <= FETCH_0;
    end if;

when OP_HLT =>
    cpu_state <= STOP;

when others =>
    error <= '1';
    cpu_state <= STOP;
```

Finally, the last bit

core.vhd

```
when EXECUTE_JMP =>
    program_counter <= data_in;
    cpu_state <= FETCH_0;
```

TB_core.vhd – let's test it

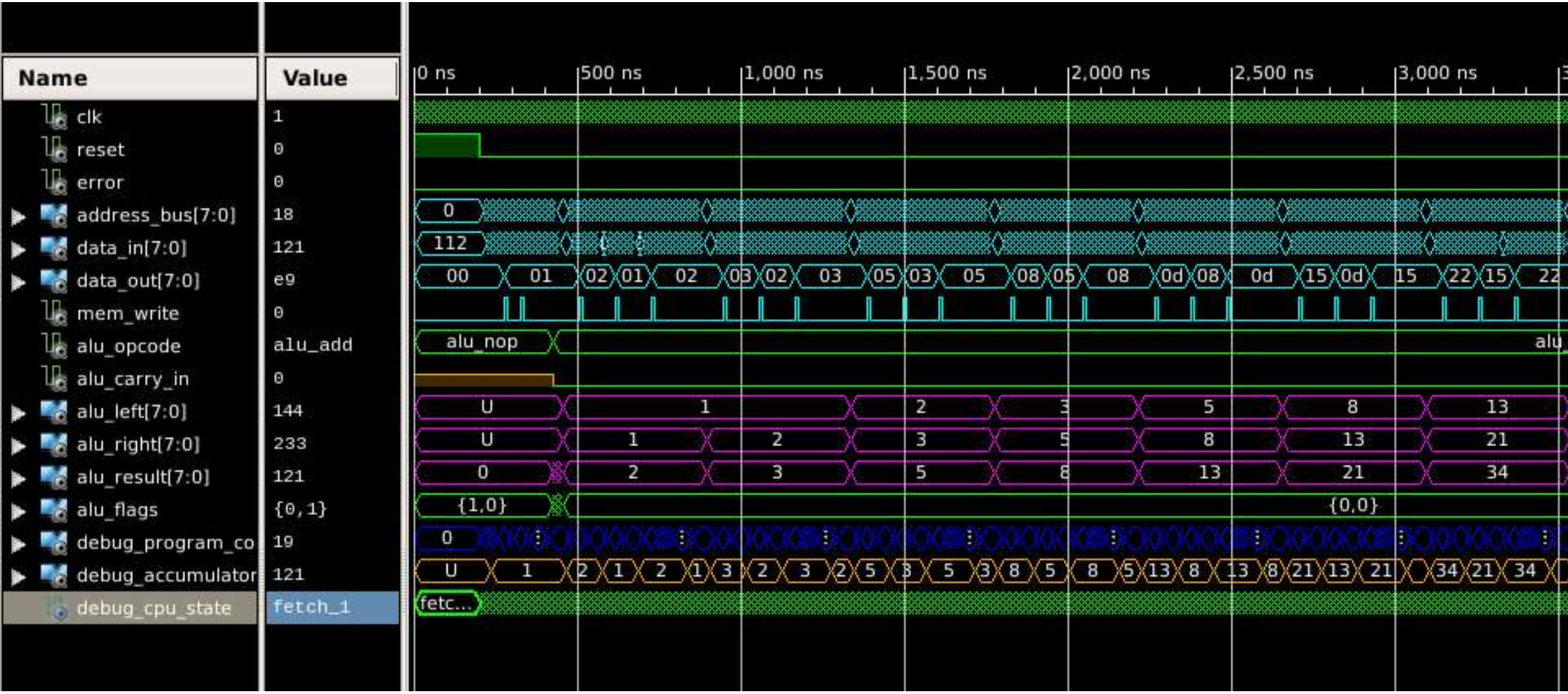
Nothing really there, we just create and link components, drive the clock signal and let the CPU run the code from the memory block

```
begin
    -- Instantiate the Unit(s) Under Test (UUT)
    c: core port map(

    m: memory port map(

    a: ALU port map(

clock_process:
    process -- clock generator process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

stim_proc:
   process   -- Stimulus process - main process that drives things
   begin
        reset <= '1';
        wait for 200 ns;
        reset <= '0';

        wait for clk_period*1000;

        wait;
   end process;

end behavior;
```

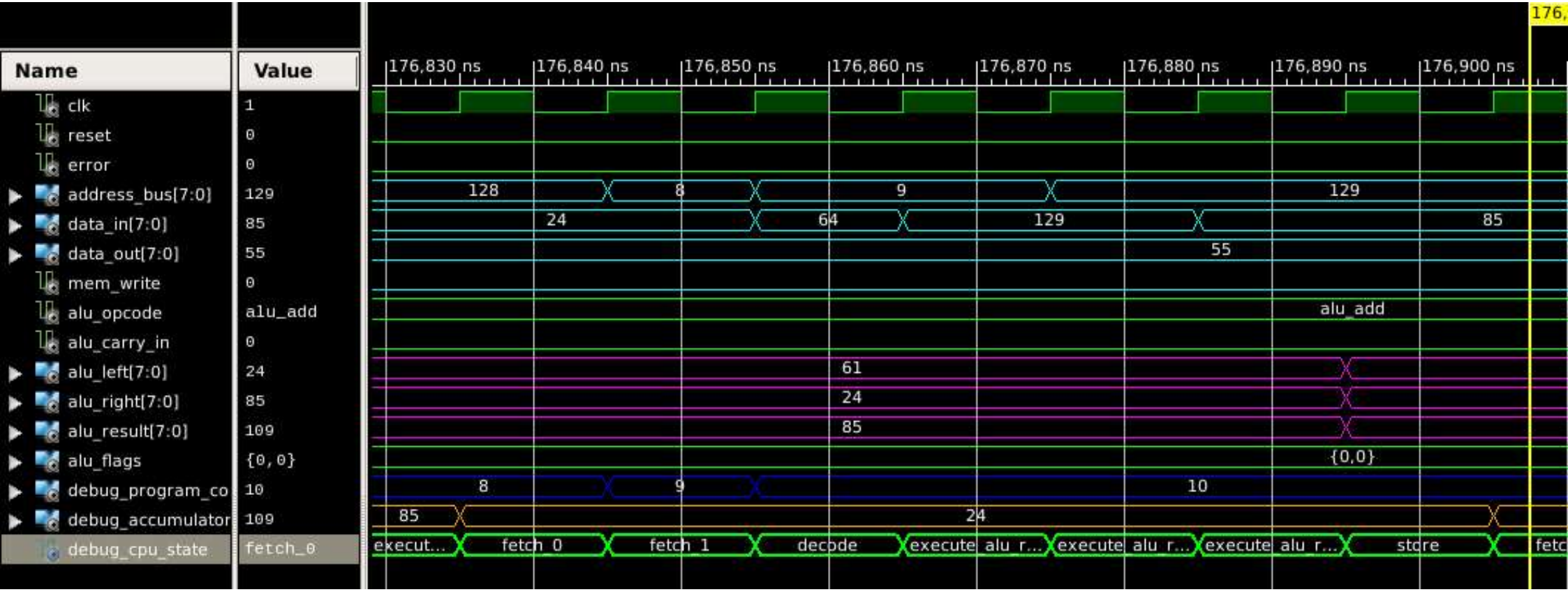
RAM – memory.vhd – preloaded code

```
signal mem: mem_type := (
    OP_LDC, b"00000001", -- A=1
    OP_STA, b"10000000", -- mem[128]=A
    OP_STA, b"10000001", -- mem[129]=A

-- addr 6:
    OP_LDA, b"10000000", -- mem[128]=A
    OP_ADD, b"10000001", -- A+=mem[129]
    OP_STA, b"10000010", -- mem[130]=A

    OP_LDA, b"10000001", -- A=mem[129]
    OP_STA, b"10000000", -- mem[128]=A

    OP_LDA, b"10000010", -- A=mem[130]
    OP_STA, b"10000001", -- mem[129]=A

    OP_JMP, b"00000110", -- jmp 6

    others => b"00000000"
);
```

TB_core.vhd – simulation run



TB_core.vhd – simulation run – zoom in



Can we do it without *HDL?

Se vi ege volas, vi povas. (Sed vi ne devus).

State	Next State																Outputs									
	OP: NOP	OP: STA	OP: LDA	OP: LDC	OP: ADD	OP: ADDC	OP: SUB	OP: SUBC	OP: OR	OP: AND	OP: NOT	OP: JMP	OP: JZ & Z=1	OP: JZ & Z=0	OP: JNZ & Z=1	OP: JNZ & Z=0	OP: HLT	ALU operation	Carry out	ALU left select	ALU right select	Addr select	Data out select	Mem write select	PC select	Accum select	Flags select
FETCH_0	FETCH_1																									
FETCH_1	DECODE																									
DECODE																										
E_STA_1																										
E_STA_2																										
E_LDA_1																										
E_LDA_2																										
E_LDA_3																										
E_LDC_1																										
E_ALU_RM_1																										
E_ALU_RM_2																										
E_ALU_RM_3																										
E_JMP																										
STORE																										

GOOD LUCK! ☺

Let’s look at the rest of the code

types.vhd

```
package types is

    type ALU_flags is record
        negative        : std_logic;
        zero            : std_logic;
        carry_out       : std_logic; -- means "borrow out" for sub
        overflow        : std_logic;
    end record ALU_flags;

    type cpu_state_type is (
```

types.vhd

```
type cpu_state_type is (
    FETCH_0,
    FETCH_1,
    DECODE,
    EXECUTE_STA_1,
    EXECUTE_STA_2,
    EXECUTE_LDA_MEM_1,
    EXECUTE_LDA_MEM_2,
    EXECUTE_LDA_MEM_3,
    EXECUTE_LDA_VAL_1,
    EXECUTE_ALU_REGMEM_1,
    EXECUTE_ALU_REGMEM_2,
    EXECUTE_ALU_REGMEM_3,
    EXECUTE_ALU_REGMEM_INV_1,
    EXECUTE_ALU_REGMEM_INV_2,
    EXECUTE_ALU_REGMEM_INV_3,
    EXECUTE_ALU_REGVAL_1,
    EXECUTE_ALU_REGVAL_INV_1,
    EXECUTE_SEVENSEGMENTTRANSLATE_1,
    EXECUTE_SEVENSEGMENTTRANSLATE_2,
    EXECUTE_JMP,
    EXECUTE_JMP_A,
    EXECUTE_PORT_IN_1,
    EXECUTE_PORT_IN_2,
    EXECUTE_PORT_OUT_1,
    EXECUTE_PORT_OUT_2,
    STORE,
    STOP
);
```

opcodes.vhd

```
type alu_opcode_type is (
        ALU_ADD,          -- add
        ALU_SUB,          -- sub Left-Right
        -- ALU_NEG,       -- negative

        ALU_OR,           -- bitwise OR
        ALU_AND,          -- .. AND
        ALU_XOR,          -- .. XOR
        -- ALU_NOT,           -- .. NOT

        ALU_SHL,          -- shift left
        ALU_SHR,          -- shift right
        ALU_SHAR,         -- shift-arithmetic left
        ALU_ROL,          -- rotate left
        ALU_ROR,          -- rotate right
        ALU_RCL,          -- rotate-carry-left
        ALU_RCR,          -- rotate-carry-right

        ALU_NOP           -- no operation
    );
```

opcodes.vhd

```
-- constant definition for various CPU instructions

-- load and store instructions
-- prefix 001
constant OP_STA          : std_logic_vector(7 downto 0):="00100000";   -- mem[arg] = A
constant OP_LDA          : std_logic_vector(7 downto 0):="00100001";   -- A = mem[arg]
constant OP_LDC          : std_logic_vector(7 downto 0):="00100010";   -- A = arg

-- math instructions
-- prefix 010
constant OP_ADD          : std_logic_vector(7 downto 0):="01000000";   -- A = A + mem[arg]
constant OP_ADDC         : std_logic_vector(7 downto 0):="01000010";   -- A = A + mem[arg] + carry
constant OP_SUB          : std_logic_vector(7 downto 0):="01000011";   -- A = A - mem[arg]
constant OP_SUBC         : std_logic_vector(7 downto 0):="01000100";   -- A = A - mem[arg] - carry
constant OP_SUBR         : std_logic_vector(7 downto 0):="01000101";   -- A = mem[arg] - A
constant OP_SUBRC        : std_logic_vector(7 downto 0):="01000110";   -- A = mem[arg] - A - carry

constant OP_ADD_V        : std_logic_vector(7 downto 0):="01010000";   -- A = A + arg
constant OP_ADDC_V       : std_logic_vector(7 downto 0):="01010010";   -- A = A + arg + carry
constant OP_SUB_V        : std_logic_vector(7 downto 0):="01010011";   -- A = A - arg
constant OP_SUBC_V       : std_logic_vector(7 downto 0):="01010100";   -- A = A - arg - carry
constant OP_SUBR_V       : std_logic_vector(7 downto 0):="01010101";   -- A = arg - A
constant OP_SUBRC_V      : std_logic_vector(7 downto 0):="01010110";   -- A = arg - A - carry

-- no need for a separate opcode for OP_NEG: equals to OP_SUBR with arg = 0
-- constant OP_NEG       : std_logic_vector(7 downto 0):="01011000";
```

opcodes.vhd

```
-- logical and bit instructions
-- prefix 011
constant OP_OR          : std_logic_vector(7 downto 0):="01100000";  -- A = A or mem[arg]
constant OP_AND         : std_logic_vector(7 downto 0):="01100001";  -- A = A and mem[arg]
constant OP_XOR         : std_logic_vector(7 downto 0):="01100010";  -- A = A xor mem[arg]
constant OP_SHR         : std_logic_vector(7 downto 0):="01100011";  -- shift A right by mem[arg]
constant OP_SHL         : std_logic_vector(7 downto 0):="01100100";  -- shift A left by mem[arg]
constant OP_SHAR        : std_logic_vector(7 downto 0):="01100101";  -- shift A right by mem[arg]
constant OP_ROL         : std_logic_vector(7 downto 0):="01100110";  -- rotate right by mem[arg]
constant OP_ROR         : std_logic_vector(7 downto 0):="01100111";  -- rotate left by mem[arg]
constant OP_RCL         : std_logic_vector(7 downto 0):="01101000";  -- rotate through carry right by mem[a
constant OP_RCR         : std_logic_vector(7 downto 0):="01101001";  -- rotate through carry left by mem[ar

constant OP_OR_V        : std_logic_vector(7 downto 0):="01110000";  -- A = A or arg
constant OP_AND_V       : std_logic_vector(7 downto 0):="01110001";  -- A = A and arg
constant OP_XOR_V       : std_logic_vector(7 downto 0):="01110010";  -- A = A xor arg
constant OP_SHR_V       : std_logic_vector(7 downto 0):="01110011";  -- shift A right by arg
constant OP_SHL_V       : std_logic_vector(7 downto 0):="01110100";  -- shift A left by arg
constant OP_SHAR_V      : std_logic_vector(7 downto 0):="01110101";  -- shift A right by arg
constant OP_ROL_V       : std_logic_vector(7 downto 0):="01110110";  -- rotate right by arg
constant OP_ROR_V       : std_logic_vector(7 downto 0):="01110111";  -- rotate left by arg
constant OP_RCL_V       : std_logic_vector(7 downto 0):="01111000";  -- rotate through carry right by arg
constant OP_RCR_V       : std_logic_vector(7 downto 0):="01111001";  -- rotate through carry left by arg

-- constant OP_NOT_V       : std_logic_vector(7 downto 0):="11000010"; -- same as OP_XOR_V with arg = 255
```

opcodes.vhd

```
-- constant OP_NOT_V        : std_logic_vector(7 downto 0):="11000010"; -- same as OP_XOR_V with arg

-- branching instructions
-- prefix 100
constant OP_JMP          : std_logic_vector(7 downto 0):="10000001"; -- jump to arg
constant OP_JMP_A        : std_logic_vector(7 downto 0):="10000000"; -- jump to arg + A

constant OP_JN           : std_logic_vector(7 downto 0):="10000010";  -- jump to arg if negative
constant OP_JP           : std_logic_vector(7 downto 0):="10000011";  -- jump to arg if positive
constant OP_JV           : std_logic_vector(7 downto 0):="10000101";  -- jump to arg if overflow
constant OP_JNV          : std_logic_vector(7 downto 0):="10000100";  -- jump to arg if no overflow
constant OP_JZ           : std_logic_vector(7 downto 0):="10000111";  -- jump to arg if zero
constant OP_JNZ          : std_logic_vector(7 downto 0):="10000110";  -- jump to arg if non zero
constant OP_JC           : std_logic_vector(7 downto 0):="10001001";  -- jump to arg if carry
constant OP_JNC          : std_logic_vector(7 downto 0):="10001000";  -- jump to arg if no carry


-- port i/o instructions
-- prefix 101
constant OP_IN           : std_logic_vector(7 downto 0):="10100000";
constant OP_OUT          : std_logic_vector(7 downto 0):="10100001";

-- special instructions
-- prefix 000
constant OP_HLT                 : std_logic_vector(7 downto 0):="00000000";
constant OP_NOP                 : std_logic_vector(7 downto 0):="00000001";
constant OP_SEVENSEGTRANSLATE   : std_logic_vector(7 downto 0):="00000010";
```

core.vhd – remaining bits

```
when OP_SUBC =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGMEM_1;
    alu_opcode <= ALU_SUB;

when OP_SUBR =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGMEM_INV_1;
    alu_opcode <= ALU_SUB;

when OP_SUBRC =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGMEM_INV_1;
    alu_opcode <= ALU_SUB;

when OP_OR =>
    cpu_state <= EXECUTE_ALU_REGMEM_1;
    alu_opcode <= ALU_OR;

when OP_AND =>
```

```
when OP_ADD_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ADD;

when OP_ADDC_V =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ADD;

when OP_SUB_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SUB;

when OP_SUBC_V =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SUB;

when OP_SUBR_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_INV_1;
```

core.vhd – remaining bits

```
when OP_ADD_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ADD;

when OP_ADDC_V =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ADD;

when OP_SUB_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SUB;

when OP_SUBC_V =>
    alu_carry_in <= flags.carry_out;
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SUB;

when OP_SUBR_V =>
    alu_carry_in <= '0';
    cpu_state <= EXECUTE_ALU_REGVAL_INV_1;
```

```
when OP_SHL_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SHL;

when OP_SHAR_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SHAR;

when OP_SHR_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_SHR;

when OP_ROL_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ROL;

when OP_ROR_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_opcode <= ALU_ROR;

when OP_RCL_V =>
    cpu_state <= EXECUTE_ALU_REGVAL_1;
    alu_carry_in <= flags.carry_out;
```

core.vhd – remaining bits

```
-- Port-I/O instructions
--
--
when OP_IN =>
    cpu_state <= EXECUTE_PORT_IN_1;

when OP_OUT =>
    cpu_state <= EXECUTE_PORT_OUT_1;

--
--
-- Branching instructions
--
--

when OP_JMP =>
    cpu_state <= EXECUTE_JMP;

when OP_JMP_A =>
    cpu_state <= EXECUTE_JMP_A;

when OP_JN | OP_JP =>
    if flags.negative = data_in(0) then
        cpu_state <= EXECUTE_JMP;
    else
        cpu_state <= FETCH_0;
    end if;

when OP_JV | OP_JNV =>
    if flags.overflow = data_in(0) then
        cpu_state <= EXECUTE_JMP;
    else
        cpu_state <= FETCH_0;
    end if;

when OP_JZ | OP_JNZ =>
    if flags.zero = data_in(0) then
        cpu_state <= EXECUTE_JMP;
    else
        cpu_state <= FETCH_0;
    end if;

when OP_JC | OP_JNC =>
    if flags.carry_out = data_in(0) then
        cpu_state <= EXECUTE_JMP;
    else
        cpu_state <= FETCH_0;
    end if;

--
--
-- Other/Special instructions
--
--

when OP_HLT =>
    cpu_state <= STOP;
```

core.vhd – remaining bits

```
when EXECUTE_ALU_REGMEM_1 =>
    address_bus <= data_in;
    mem_read <= '1';
    cpu_state <= EXECUTE_ALU_REGMEM_2;

when EXECUTE_ALU_REGMEM_2 =>
    cpu_state <= EXECUTE_ALU_REGMEM_3;

when EXECUTE_ALU_REGMEM_3 =>
    alu_left <= accumulator;
    alu_right <= data_in;
    cpu_state <= STORE;
```

```
when EXECUTE_ALU_REGMEM_INV_1 =>
    address_bus <= data_in;
    mem_read <= '1';
    cpu_state <= EXECUTE_ALU_REGMEM_INV_2;

when EXECUTE_ALU_REGMEM_INV_2 =>
    cpu_state <= EXECUTE_ALU_REGMEM_INV_3;

when EXECUTE_ALU_REGMEM_INV_3 =>
    alu_left <= data_in;
    alu_right <= accumulator;
    cpu_state <= STORE;
```

```
when EXECUTE_ALU_REGVAL_1 =>
    alu_left <= accumulator;
    alu_right <= data_in;
    cpu_state <= STORE;

when EXECUTE_ALU_REGVAL_INV_1 =>
    alu_left <= data_in;
    alu_right <= accumulator;
    cpu_state <= STORE;
```

core.vhd – remaining bits

```
when EXECUTE_JMP =>
    program_counter <= data_in;
    cpu_state <= FETCH_0;

when EXECUTE_JMP_A =>
    program_counter <= accumulator + data_in;
    cpu_state <= FETCH_0;
```

```
when EXECUTE_PORT_IN_1 =>
    pio_address <= data_in;
    pio_read_enable <= '1';
    cpu_state <= EXECUTE_PORT_IN_2;

when EXECUTE_PORT_IN_2 =>
    if pio_io_ready = '1' then
        cpu_state <= FETCH_0;
        pio_read_enable <= '0';
        accumulator <= pio_data_r;
    end if;

when EXECUTE_PORT_OUT_1 =>
    pio_address <= data_in;
    pio_write_enable <= '1';
    pio_data_w <= accumulator;
    cpu_state <= EXECUTE_PORT_OUT_2;

when EXECUTE_PORT_OUT_2 =>
    if pio_io_ready = '1' then
        cpu_state <= FETCH_0;
        pio_write_enable <= '0';
    end if;
```

core.vhd – remaining bits

```
when EXECUTE_SEVENSEGMENTTRANSLATE_1 =>
    alu_opcode <= ALU_SHR;
    alu_left <= accumulator;
    alu_right <= data_in;
    cpu_state <= EXECUTE_SEVENSEGMENTTRANSLATE_2;

when EXECUTE_SEVENSEGMENTTRANSLATE_2 =>
    case alu_result(3 downto 0) is
        when "0000" => accumulator <= "11111100";
        when "0001" => accumulator <= "01100000";
        when "0010" => accumulator <= "11011010";
        when "0011" => accumulator <= "11110010";
        when "0100" => accumulator <= "01100110";
        when "0101" => accumulator <= "10110110";
        when "0110" => accumulator <= "10111110";
        when "0111" => accumulator <= "11100000";
        when "1000" => accumulator <= "11111110";
        when "1001" => accumulator <= "11110110";
        when "1010" => accumulator <= "11101110";
        when "1011" => accumulator <= "00111110";
        when "1100" => accumulator <= "10011100";
        when "1101" => accumulator <= "01111010";
        when "1110" => accumulator <= "10011110";
        when "1111" => accumulator <= "10001110";
        when others => accumulator <= "01010101";
    end case;
    cpu_state <= FETCH_0;
```

alu.vhd – remaining bits

```
when ALU_SHR =>
    -- a bit wordy...
    case right_arg is
        when "00000000" => temp := carry_in & left_arg(7 downto 0);
        when "00000001" => temp := carry_in & "0" & left_arg(7 downto 1);
        when "00000010" => temp := carry_in & "00" & left_arg(7 downto 2);
        when "00000011" => temp := carry_in & "000" & left_arg(7 downto 3);
        when "00000100" => temp := carry_in & "0000" & left_arg(7 downto 4);
        when "00000101" => temp := carry_in & "00000" & left_arg(7 downto 5);
        when "00000110" => temp := carry_in & "000000" & left_arg(7 downto 6);
        when "00000111" => temp := carry_in & "0000000" & left_arg(7);
        when others     => temp := carry_in & "00000000";
    end case;
```

```
when ALU_SHL =>
    -- a bit wordy... also...
    case right_arg is
        when "00000000" => temp := carry_in & left_arg(7 downto 0);
        when "00000001" => temp := carry_in & left_arg(6 downto 0) & '0';
        when "00000010" => temp := carry_in & left_arg(5 downto 0) & "00";
        when "00000011" => temp := carry_in & left_arg(4 downto 0) & "000";
        when "00000100" => temp := carry_in & left_arg(3 downto 0) & "0000";
        when "00000101" => temp := carry_in & left_arg(2 downto 0) & "00000";
        when "00000110" => temp := carry_in & left_arg(1 downto 0) & "000000";
        when "00000111" => temp := carry_in & left_arg(0)          & "0000000";
        when others     => temp := carry_in & "00000000";
    end case;
```

alu.vhd – remaining bits

```
when ALU_ROR =>
    -- a bit wordy...
    case right_arg(2 downto 0) is
        when "000"  => temp := carry_in & left_arg(7 downto 0);
        when "001"  => temp := carry_in & left_arg(0)           & left_arg(7 downto 1);
        when "010"  => temp := carry_in & left_arg(1 downto 0 ) & left_arg(7 downto 2);
        when "011"  => temp := carry_in & left_arg(2 downto 0 ) & left_arg(7 downto 3);
        when "100"  => temp := carry_in & left_arg(3 downto 0 ) & left_arg(7 downto 4);
        when "101"  => temp := carry_in & left_arg(4 downto 0 ) & left_arg(7 downto 5);
        when "110"  => temp := carry_in & left_arg(5 downto 0 ) & left_arg(7 downto 6);
        when "111"  => temp := carry_in & left_arg(6 downto 0 ) & left_arg(7);
        when others => temp := carry_in & "00000000";
    end case;

when ALU_ROL =>
    -- a bit wordy... also...
    case right_arg(2 downto 0) is
        when "000"  => temp := carry_in & left_arg(7 downto 0);
        when "001"  => temp := carry_in & left_arg(7 downto 1) & left_arg(0);
        when "010"  => temp := carry_in & left_arg(7 downto 2) & left_arg(1 downto 0);
        when "011"  => temp := carry_in & left_arg(7 downto 3) & left_arg(2 downto 0);
        when "100"  => temp := carry_in & left_arg(7 downto 4) & left_arg(3 downto 0);
        when "101"  => temp := carry_in & left_arg(7 downto 5) & left_arg(4 downto 0);
        when "110"  => temp := carry_in & left_arg(7 downto 6) & left_arg(5 downto 0);
        when "111"  => temp := carry_in & left_arg(7)          & left_arg(6 downto 0);
        when others => temp := carry_in & "00000000";
    end case;
```

memory.vhd – now with read_enable

```
entity memory is
    port
    (
        clk          : in std_logic;
        address_bus  : in std_logic_vector(7 downto 0);
        data_write   : in std_logic_vector(7 downto 0);
        data_read    : out std_logic_vector(7 downto 0);
        mem_read     : in std_logic;
        mem_write    : in std_logic;
        rst          : in std_logic
    );
end memory;
```

pio.vhd

```
entity pio is
    port (
        clk             : in std_logic;
        clk_unscaled    : in std_logic;
        rst             : in std_logic;
        address         : in std_logic_vector(7 downto 0);
        data_w          : in std_logic_vector(7 downto 0); -- data entering IO port
        data_r          : out std_logic_vector(7 downto 0);
        write_enable    : in std_logic;
        read_enable     : in std_logic;
        io_ready        : out std_logic;

        in_port_0       : in std_logic_vector (7 downto 0); -- dp switches
        in_port_1       : in std_logic_vector (7 downto 0); -- push btns
        in_port_2       : in std_logic_vector (7 downto 0); -- pin header 6
        in_port_3       : in std_logic_vector (7 downto 0); -- pin header 7

        out_port_4      : out std_logic_vector (7 downto 0); -- individual leds
        out_port_5      : out std_logic_vector (7 downto 0); -- 7-segment digits
        out_port_6      : out std_logic_vector (7 downto 0); -- 7-segment enable signals
        out_port_7      : out std_logic_vector (7 downto 0); -- pin header 8
        out_port_8      : out std_logic_vector (7 downto 0) -- pin header 9
    );
end pio;
```

pio.vhd

```
architecture beh of pio is

    component sevenseg is
        generic (
            num_segments: integer := 3  -- up to 8
        );
        port
        (
            clk                     : in std_logic;
            rst                     : in std_logic;
            segment_select          : in std_logic_vector(7 downto 0); -- binary encoded
            segment_led_mask        : in std_logic_vector(7 downto 0);
            sel_port_out            : out std_logic_vector(7 downto 0);
            data_port_out           : out std_logic_vector(7 downto 0)
        );
    end component;


    type io_state_type is (IO_IDLE, IO_BUSY);
    signal state : io_state_type := IO_IDLE;

    signal segment_select   : std_logic_vector(7 downto 0); -- binary encoded
    signal segment_led_mask : std_logic_vector(7 downto 0);

begin
    ss: sevenseg port map
```

pio.vhd

```
ss: sevenseg port map
    (
        clk                     => clk_unscaled,
        rst                     => rst,
        segment_select          => segment_select,
        segment_led_mask        => segment_led_mask,
        sel_port_out            => out_port_6,
        data_port_out           => out_port_5
    );

process (clk, data_w, write_enable, read_enable)
begin
    if rising_edge(clk) then

        case state is
            when IO_IDLE =>

                if write_enable = '1' then
                    case address is
                        when "00000100" => out_port_4 <= data_w;
                        when "00000101" => segment_led_mask <= data_w;
                        when "00000110" => segment_select <= data_w;
                        when "00000111" => out_port_7 <= data_w;
                        when "00001000" => out_port_8 <= data_w;
                        when others     =>
                    end case;

                    state <= IO_BUSY;

                elsif read_enable = '1' then
                    case address is
                        when "00000000" => data_r <= in_port_0;
                        when "00000001" => data_r <= in_port_1;
                        when "00000010" => data_r <= in_port_2;
                        when "00000011" => data_r <= in_port_3;
                        when others     => data_r <= "00000000";
                    end case;

                    state <= IO_BUSY;
                end if;

            when others =>
                state <= IO_IDLE;
        end case;
    end if;
end process;

io_ready <= '1' when state = IO_IDLE else '0';
```

sevenseg.vhd

```
entity sevenseg is
    generic (
        num_segments: integer := 3  -- up to 8
    );
    port
    (
        clk                 : in std_logic;
        rst                 : in std_logic;
        segment_select      : in std_logic_vector(7 downto 0); -- binary enc
        segment_led_mask    : in std_logic_vector(7 downto 0);
        sel_port_out        : out std_logic_vector(7 downto 0);
        data_port_out       : out std_logic_vector(7 downto 0)
    );
end sevenseg;
```

sevenseg.vhd

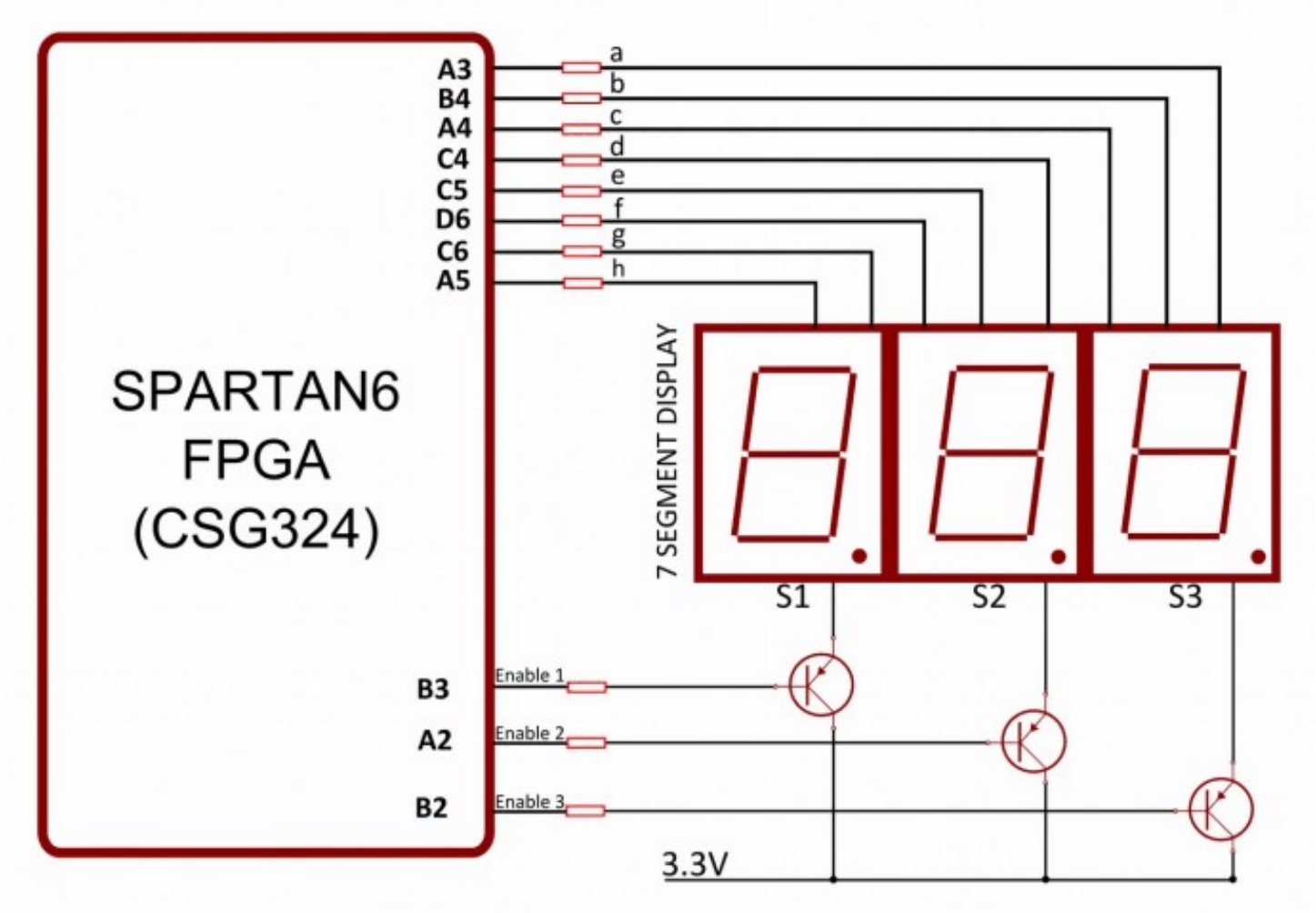


Image © 2018 Numato Systems Pvt. Ltd.,
source: https://numato.com/docs/mimas-v2-spartan-6-fpga-development-board-with-ddr-sdram/

sevenseg.vhd

```
architecture behaviour of sevenseg is
    subtype counter_type is integer range 0 to 65535;
    subtype digit_idx_type is integer range 0 to num_segments-1;
    type digits_array is array (num_segments-1 downto 0) of std_logic_vector(7 downto 0);

    signal counter: counter_type := 0;
    signal current_digit: digit_idx_type := 0;
    signal digits : digits_array := (others => (others => '0'));
begin
    -- first process - state machine handling output refreshes
```

sevenseg.vhd

```
-- first process - state machine handling output refreshes
process (clk, rst)
    constant lsb_one : unsigned(7 downto 0) := "00000001";
begin
    if rst = '1' then
        counter <= 0;
        current_digit <= 0;
    elsif rising_edge(clk) then
        -- signals are active low on the wire
        data_port_out <= not digits(current_digit);
        sel_port_out <= not std_logic_vector(shift_left(lsb_one, current_digit));

        if counter = counter_type'high then
            counter <= 0;
            if current_digit = digit_idx_type'high
            then
                current_digit <= 0;
            else
                current_digit <= current_digit + 1;
            end if;
        else
            counter <= counter + 1;
        end if;

    end if;
end process;
```

sevenseg.vhd

```
-- second process - handling internal register updates
process (clk, rst, segment_select, segment_led_mask)
    variable idx : integer := 0;
begin
    if rst = '1' then
        digits <= (others => (others => '0'));
    elsif rising_edge(clk) then

        idx := to_integer(unsigned(segment_select));
        if idx >= 0 and idx <= num_segments-1 then
            digits(idx) <= segment_led_mask;
        end if;

    end if;
end process;
```

soc.vhd – simply linking everything together

It is boring…

```
library ieee ;
use ieee.std_logic_1164.all;
use work.opcodes.all;
use work.types.all;

entity soc is
    port (
        CLK_100MHz : in std_logic;

        DPSwitch_0 : in std_logic; -- pull up by default
        DPSwitch_1 : in std_logic; -- pull up by default
        DPSwitch_2 : in std_logic; -- pull up by default
        DPSwitch_3 : in std_logic; -- pull up by default
        DPSwitch_4 : in std_logic; -- pull up by default
        DPSwitch_5 : in std_logic; -- pull up by default
        DPSwitch_6 : in std_logic; -- pull up by default
        DPSwitch_7 : in std_logic; -- pull up by default
        Switch_5 : in std_logic; -- pull up by default
        Switch_4 : in std_logic; -- pull up by default
        Switch_3 : in std_logic; -- pull up by default
        Switch_2 : in std_logic; -- pull up by default
        Switch_1 : in std_logic; -- pull up by default
        Switch_0 : in std_logic; -- pull up by default

        LED_7 : out std_logic;
        LED_6 : out std_logic;
        LED_5 : out std_logic;
        LED_4 : out std_logic;
        LED_3 : out std_logic;
        LED_2 : out std_logic;
        LED_1 : out std_logic;
        LED_0 : out std_logic;

        SevenSegment_7  : out std_logic;    -- a
        SevenSegment_6  : out std_logic;    -- b
        SevenSegment_5  : out std_logic;    -- c
        SevenSegment_4  : out std_logic;    -- d
        SevenSegment_3  : out std_logic;    -- e
        SevenSegment_2  : out std_logic;    -- f
        SevenSegment_1  : out std_logic;    -- g
        SevenSegment_0  : out std_logic;    -- dot
        SevenSegmentEnable_2 : out std_logic;
        SevenSegmentEnable_1 : out std_logic;
        SevenSegmentEnable_0 : out std_logic;

        IO_P6_7 : in std_logic;  --  #Pin 1
        IO_P6_6 : in std_logic;  --  #Pin 2
        IO_P6_5 : in std_logic;  --  #Pin 3
        IO_P6_4 : in std_logic;  --  #Pin 4
        IO_P6_3 : in std_logic;  --  #Pin 5
        IO_P6_2 : in std_logic;  --  #Pin 6
        IO_P6_1 : in std_logic;  --  #Pin 7
        IO_P6_0 : in std_logic;  --  #Pin 8
        IO_P7_7 : in std_logic;  --  #Pin 1
        IO_P7_6 : in std_logic;  --  #Pin 2
        IO_P7_5 : in std_logic;  --  #Pin 3
        IO_P7_4 : in std_logic;  --  #Pin 4
        IO_P7_3 : in std_logic;  --  #Pin 5
        IO_P7_2 : in std_logic;  --  #Pin 6
        IO_P7_1 : in std_logic;  --  #Pin 7
        IO_P7_0 : in std_logic;  --  #Pin 8

        IO_P8_7 : out std_logic;  --  #Pin 1
        IO_P8_6 : out std_logic;  --  #Pin 2
        IO_P8_5 : out std_logic;  --  #Pin 3
        IO_P8_4 : out std_logic;  --  #Pin 4
        IO_P8_3 : out std_logic;  --  #Pin 5
        IO_P8_2 : out std_logic;  --  #Pin 6
        IO_P8_1 : out std_logic;  --  #Pin 7
        IO_P8_0 : out std_logic;  --  #Pin 8
        IO_P9_7 : out std_logic;  --  #Pin 1
        IO_P9_6 : out std_logic;  --  #Pin 2
        IO_P9_5 : out std_logic;  --  #Pin 3
        IO_P9_4 : out std_logic;  --  #Pin 4
        IO_P9_3 : out std_logic;  --  #Pin 5
        IO_P9_2 : out std_logic;  --  #Pin 6
        IO_P9_1 : out std_logic;  --  #Pin 7
        IO_P9_0 : out std_logic  --  #Pin 8

        );
end soc;

architecture structural of soc is

    component cpu is
        port
        (
            clk                 : in std_logic;
            reset               : in std_logic;
            error               : out std_logic;

            mem_address         : out std_logic_vector(7 downto 0);
            mem_data_r          : in std_logic_vector(7 downto 0);
            mem_data_w          : out std_logic_vector(7 downto 0);
            mem_write           : out std_logic;

            pio_address         : out std_logic_vector(7 downto 0);
            pio_data_w          : out std_logic_vector(7 downto 0); -- data entering IO port
            pio_data_r          : in std_logic_vector(7 downto 0);
            pio_write_enable    : out std_logic;
            pio_read_enable     : out std_logic;
            pio_io_ready        : in std_logic
        );
    end component;

    component memory is
        port
        (
            clk         : in std_logic;
            address_bus : in std_logic_vector(7 downto 0);
            data_write  : in std_logic_vector(7 downto 0);
            data_read   : out std_logic_vector(7 downto 0);
            mem_write   : in std_logic;
            rst         : in std_logic
        );
    end component;

    signal reset            : std_logic;
    signal error            : std_logic;

    signal mem_address      : std_logic_vector(7 downto 0);
    signal data_from_mem_to_cpu : std_logic_vector(7 downto 0);
    signal data_from_cpu_to_mem : std_logic_vector(7 downto 0);
    signal mem_write        : std_logic;

begin
    c : cpu port map (
            clk             => CLK_100MHz,
            reset           => reset,
            error           => error,

            mem_address     => mem_address,
            mem_data_r      => data_from_mem_to_cpu,
            mem_data_w      => data_from_cpu_to_mem,
            mem_write       => mem_write,

            pio_address     => pio_address,
            pio_data_w      => data_from_cpu_to_pio,
            pio_data_r      => data_from_pio_to_cpu,
            pio_write_enable    => pio_write_enable,
            pio_read_enable     => pio_read_enable,
            pio_io_ready        => pio_io_ready
    );

    m: memory port map (
        clk             => CLK_100MHz,
        address_bus     => mem_address,
        data_write      => data_from_cpu_to_mem,
        data_read       => data_from_mem_to_cpu,
        mem_write       => mem_write,
        rst             => reset
    );

    -- Finally - manual signal wirings
    reset <= not Switch_5; -- it is pull up

    in_port_1(7 downto 5) <= "000"; -- NC really
    in_port_1(4) <= not Switch_4;
    in_port_1(3) <= not Switch_3;
    in_port_1(2) <= not Switch_2;
    in_port_1(1) <= not Switch_1;
    in_port_1(0) <= not Switch_0;

    LED_7 <= error;
    LED_6 <= out_port_4(6);
    LED_5 <= out_port_4(5);
    LED_4 <= out_port_4(4);
    LED_3 <= out_port_4(3);
    LED_2 <= out_port_4(2);
    LED_1 <= out_port_4(1);
    LED_0 <= out_port_4(0);

    in_port_0(0) <= DPSwitch_0;
    in_port_0(1) <= DPSwitch_1;
    in_port_0(2) <= DPSwitch_2;
    in_port_0(3) <= DPSwitch_3;
    in_port_0(4) <= DPSwitch_4;
    in_port_0(5) <= DPSwitch_5;
    in_port_0(6) <= DPSwitch_6;
    in_port_0(7) <= DPSwitch_7;

    in_port_2(0) <= IO_P6_7;
    in_port_2(1) <= IO_P6_6;
    in_port_2(2) <= IO_P6_5;
    in_port_2(3) <= IO_P6_4;
    in_port_2(4) <= IO_P6_3;
    in_port_2(5) <= IO_P6_2;
    in_port_2(6) <= IO_P6_1;
    in_port_2(7) <= IO_P6_0;

    in_port_3(0) <= IO_P7_7;
    in_port_3(1) <= IO_P7_6;
    in_port_3(2) <= IO_P7_5;
    in_port_3(3) <= IO_P7_4;
    in_port_3(4) <= IO_P7_3;
    in_port_3(5) <= IO_P7_2;
    in_port_3(6) <= IO_P7_1;
    in_port_3(7) <= IO_P7_0;

    SevenSegment_7 <= out_port_5(7);
    SevenSegment_6 <= out_port_5(6);
    SevenSegment_5 <= out_port_5(5);
    SevenSegment_4 <= out_port_5(4);
    SevenSegment_3 <= out_port_5(3);
    SevenSegment_2 <= out_port_5(2);
    SevenSegment_1 <= out_port_5(1);
    SevenSegment_0 <= out_port_5(0);

    SevenSegmentEnable_2 <= out_port_6(2);
    SevenSegmentEnable_1 <= out_port_6(1);
    SevenSegmentEnable_0 <= out_port_6(0);
```

Program…

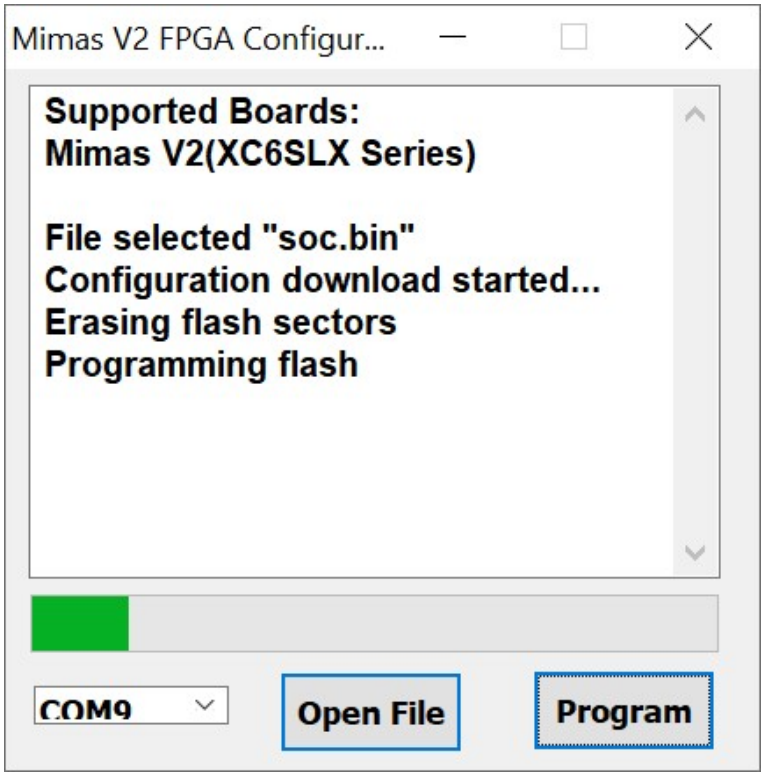
```
signal mem: mem_type:= (
--0: start:
    OP_LDC, x"01",
    OP_STA, A0,          -- A0 = 1
    OP_STA, B0,          -- B0 = 1
    OP_LDC, x"00",
    OP_STA, A1,          -- A1 = 0
    OP_STA, B1,          -- B1 = 0

--0x0c: loop:
    OP_LDA, A0,
    OP_ADD, B0,
    OP_STA, C0,          -- C0 = A0 + B0
    OP_LDA, A1,
    OP_ADDC, B1,
    OP_STA, C1,          -- C1 = A1 + B1 + carry

    OP_LDA, B0,
    OP_STA, A0,          -- A0 = B0
    OP_LDA, B1,
    OP_STA, A1,          -- A1 = B1
    OP_LDA, C0,
    OP_STA, B0,          -- B0 = C0
    OP_LDA, C1,
    OP_STA, B1,          -- B1 = C1

    -- now - display the thing
    OP_LDC, x"00",       -- select LCD display
    OP_OUT, x"06",       -- out[6] = 0
    OP_LDA, C0,          -- output the number part
    OP_SEVENSEGTRANSLATE, x"00",
    OP_OUT, x"05",       -- out[5] = acc

    OP_LDC, x"01",       -- select LCD display
    OP_OUT, x"06",       -- out[6] = 1
    OP_LDA, C0,          -- output the number part
    OP_SEVENSEGTRANSLATE, x"04",
    OP_OUT, x"05",       -- out[5] = acc

    OP_LDC, x"02",       -- select LCD display
    OP_OUT, x"06",       -- out[6] = 2
    OP_LDA, C1,          -- output the number part
    OP_SEVENSEGTRANSLATE, x"00",
    OP_OUT, x"05",       -- out[5] = acc

    OP_LDA, C0,          -- output the number part into 8-leds
    OP_OUT, x"04",       -- out[4] = acc

--0x4a:
    OP_LDC, x"ff",
    OP_STA, S0,          -- S0 = 255
    OP_STA, S1,          -- S1 = 255
    OP_LDC, x"0f",
    OP_STA, S2,          -- S2 = 15

-- 0x54: sleep_loop:
    OP_LDA, S0,
    OP_SUB_V, x"01",
    OP_STA, S0,          -- S0 = S0 - 1
    OP_LDA, S1,
    OP_SUBC_V, x"00",
    OP_STA, S1,          -- S1 = S1 - 0 - carry
    OP_LDA, S2,
    OP_SUBC_V, x"00",
    OP_STA, S2,          -- S2 = S2 - 0 - carry
    OP_JNZ, x"54",       -- goto sleep_loop if new S2 != 0

    OP_LDA, C1,
    OP_AND_V, x"f0",     -- Acc = C1 & 0xf0
    OP_JNZ, x"00",       -- go start if Acc != 0 (12-bit ovflow)
    OP_JMP, x"0c",       -- go loop in all other cases

    others => x"00"
```

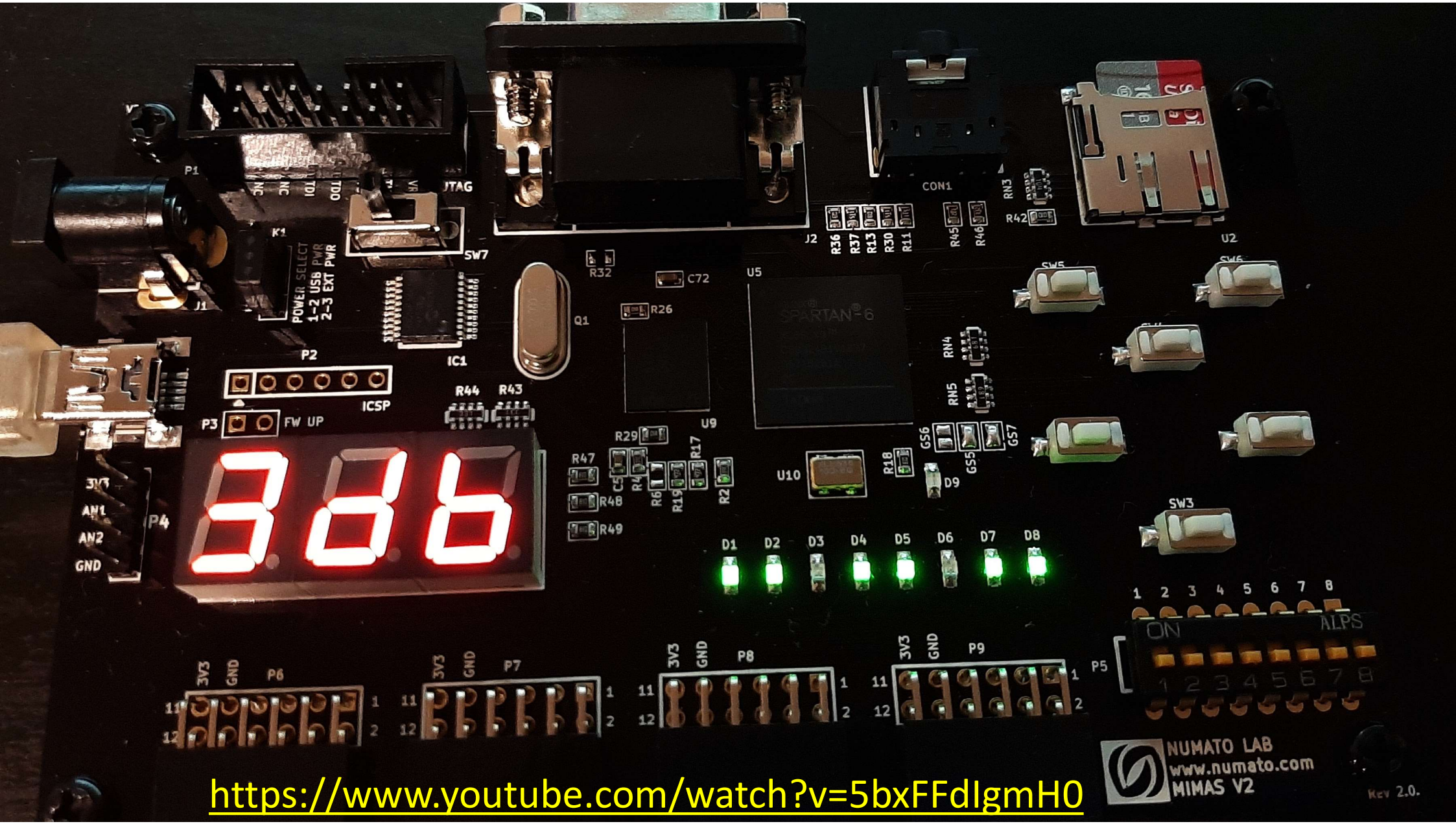
Expected result?

```
void Main()
{
    ushort A = 1, B = 1, C = 0;

    for (;;)
    {
        C = (ushort)(A + B);
        Console.WriteLine($"{C&0xfff:X3}");
        A = B;
        B = C;
        if ((C&0xf000) != 0)
            break;
    }
}
```

002
003
005
008
00D
015
022
037
059
090
0E9
179
262
3DB
63D
A18
055

Finally…


Mimas V2 FPGA Configur...
Supported Boards:
Mimas V2(XC6SLX Series)
File selected "soc.bin"
Configuration download started...
Erasing flash sectors
Programming flash
COM9
Open File
Program


https://www.youtube.com/watch?v=5bxFFdIgmH0
NUMATO LAB
www.numato.com
MIMAS V2
Rev 2.0.

What I didn’t tell you?

Pure lack of time! No conspiracy here

Items not covered

- Timing
  - Flip-Flop timings
  - Whole design timing calculation / etc
- Multiplier and Divider circuits
- Clock domain crossing
- Workflow of FPGA design (synthesis is only the first step)
- IP components / libraries
- User constraints
- Advanced topics on the CPU design – microcode, pipeline, out of order execution, etc – I'm about to cover it myself yet.
- And a lot more… can't fit everything into our format

Appendix A. Xilinx Spartan 6 family summary

Table 3: **Spartan-6 FPGA Logic Resources**

Device	Logic Cells	Total Slices	SLICEMs	SLICELs	SLICEXs	Number of 6-Input LUTs	Maximum Distributed RAM (Kb)	Shift Registers (Kb)	Number of Flip-Flops
XC6SLX4	3,840	600	300	0	300	2,400	75	38	4,800
XC6SLX9	9,152	1,430	360	355	715	5,720	90	45	11,440
XC6SLX16	14,579	2,278	544	595	1,139	9,112	136	68	18,224
XC6SLX25	24,051	3,758	916	963	1,879	15,032	229	115	30,064
XC6SLX45	43,661	6,822	1,602	1,809	3,411	27,288	401	200	54,576
XC6SLX75	74,637	11,662	2,768	3,063	5,831	46,648	692	346	93,296
XC6SLX100	101,261	15,822	3,904	4,007	7,911	63,288	976	488	126,576
XC6SLX150	147,443	23,038	5,420	6,099	11,519	92,152	1,355	678	184,304
XC6SLX25T	24,051	3,758	916	963	1,879	15,032	229	115	30,064
XC6SLX45T	43,661	6,822	1,602	1,809	3,411	27,288	401	200	54,576
XC6SLX75T	74,637	11,662	2,768	3,063	5,831	46,648	692	346	93,296
XC6SLX100T	101,261	15,822	3,904	4,007	7,911	63,288	976	488	126,576
XC6SLX150T	147,443	23,038	5,420	6,099	11,519	92,152	1,355	678	184,304

Appendix B. Xilinx Virtex Ultrascale family summary

Table 9: **Virtex UltraScale+ FPGA Feature Summary**

	VU3P	VU5P	VU7P	VU9P	VU11P	VU13P	VU19P	VU27P	VU29P
System Logic Cells	862,050	1,313,763	1,724,100	2,586,150	2,835,000	3,780,000	8,937,600	2,835,000	3,780,000
CLB Flip-Flops	788,160	1,201,154	1,576,320	2,364,480	2,592,000	3,456,000	8,171,520	2,592,000	3,456,000
CLB LUTs	394,080	600,577	788,160	1,182,240	1,296,000	1,728,000	4,085,760	1,296,000	1,728,000
Max. Distributed RAM (Mb)	12.0	18.3	24.1	36.1	36.2	48.3	58.4	36.2	48.3
Block RAM Blocks	720	1,024	1,440	2,160	2,016	2,688	2,160	2,016	2,688
Block RAM (Mb)	25.3	36.0	50.6	75.9	70.9	94.5	75.9	70.9	94.5
UltraRAM Blocks	320	470	640	960	960	1,280	320	960	1,280
UltraRAM (Mb)	90.0	132.2	180.0	270.0	270.0	360.0	90.0	270.0	360.0
HBM DRAM (GB)	–	–	–	–	–	–	–	–	–
CMTs (1 MMCM and 2 PLLs)	10	20	20	30	12	16	40	16	16
Max. HP I/O(1)	520	832	832	832	624	832	1,976	520	676
Max. HD I/O(2)	–	–	–	–	–	–	96	–	–
DSP Slices	2,280	3,474	4,560	6,840	9,216	12,288	3,840	9,216	12,288
System Monitor	1	2	2	3	3	4	4	4	4
GTY Transceivers 32.75Gb/s(3)	40	80	80	120	96	128	80	32	32
GTM Transceivers 58.0Gb/s(3)	–	–	–	–	–	–	–	48	48
100G / 50G KP4 FEC	–	–	–	–	–	–	–	24/48	24/48
Transceiver Fractional PLLs	20	40	40	60	48	64	40	40	40
PCIe Gen3 x16	2	4	4	6	3	4	0	1	1
PCIe Gen3 x16 / Gen4 x8 / CCIX(4)	–	–	–	–	–	–	8	–	–
150G Interlaken	3	4	6	9	6	8	0	8	8
100G Ethernet w/RS-FEC	3	4	6	9	9	12	0	15	15

Appendix C. Further reading

- http://www.simplecpudesign.com/simple_cpu_v1/index.html
- https://blog.classycode.com/implementing-a-cpu-in-vhdl-part-1-6afd4c1ed491
- http://embeddedsystems.io/ahmes-a-simple-8-bit-cpu-in-vhdl/

Plus all the books and materials from part 1!

Appendix D. Mimas V2 Dev Board

- https://numato.com/product/mimas-v2-spartan-6-fpga-development-board-with-ddr-sdram