FPGA programming

Kristian Harder, RAL

Science and Technology Facilities Council

About me

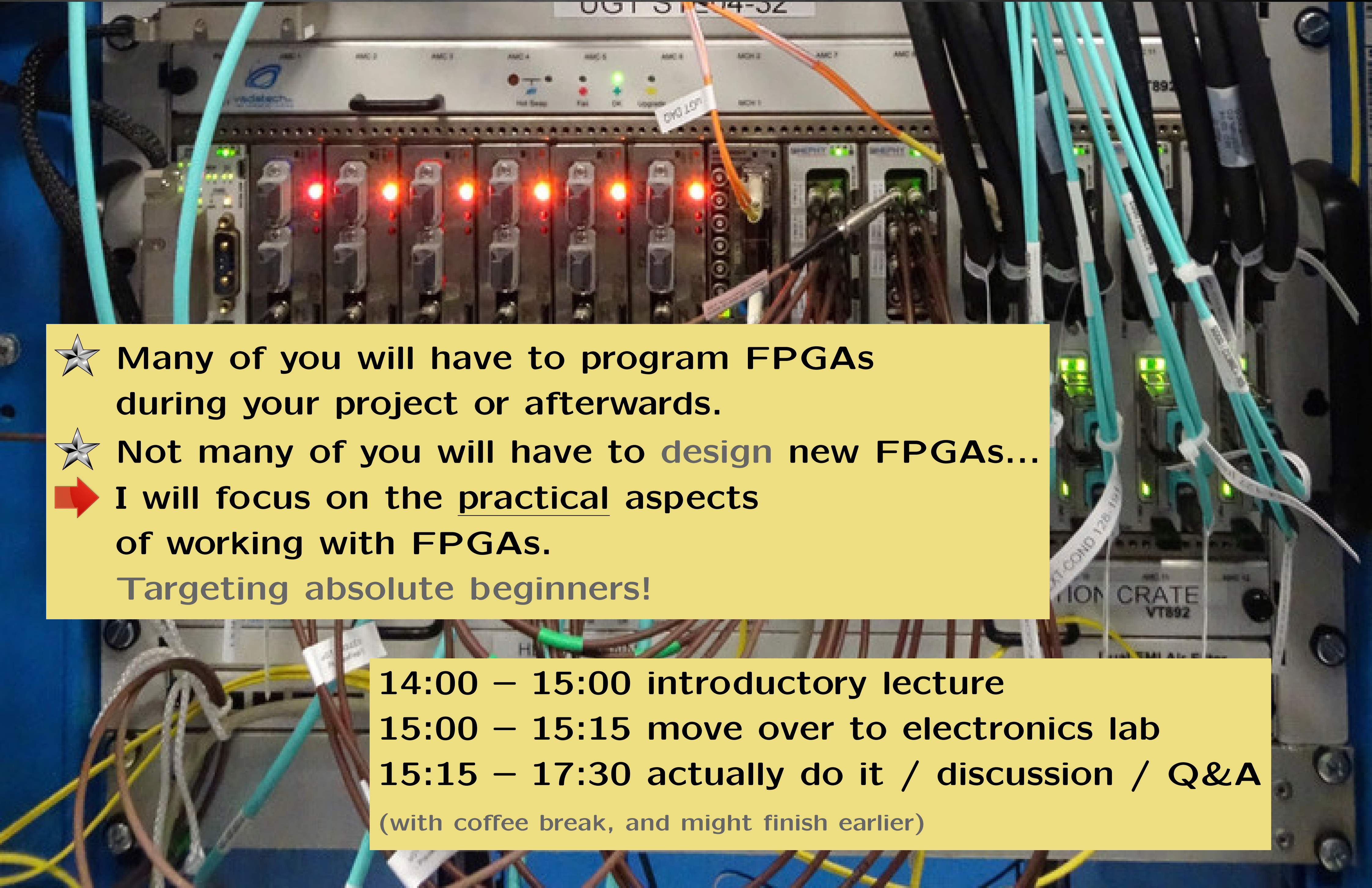
Kristian Harder

PhD Hamburg University/DESY 1998–2002:
- QCD analysis (OPAL, TESLA)
- track reconstruction software (TESLA)

Fermilab 2002–2006:
- electroweak analysis (DØ)
- silicon detector back-end electronics (DØ)

RAL 2006–:
- silicon detector simulation (ILC)
- exotica analysis (CMS)
- readout+trigger electronics (CMS, DUNE)

About this lecture



- Many of you will have to program FPGAs during your project or afterwards.
- Not many of you will have to design new FPGAs...
- I will focus on the practical aspects of working with FPGAs.

Targeting absolute beginners!

14:00 – 15:00 introductory lecture
15:00 – 15:15 move over to electronics lab
15:15 – 17:30 actually do it / discussion / Q&A

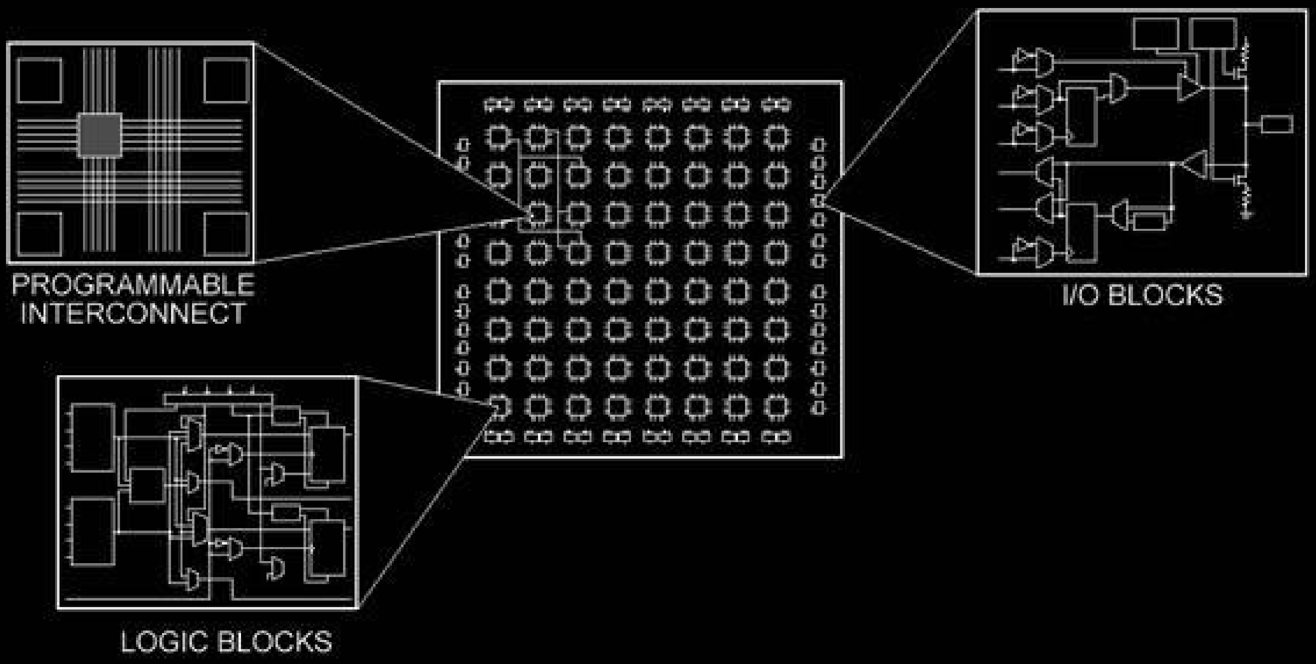
(with coffee break, and might finish earlier)

What is an FPGA?

Field Programmable Gate Array

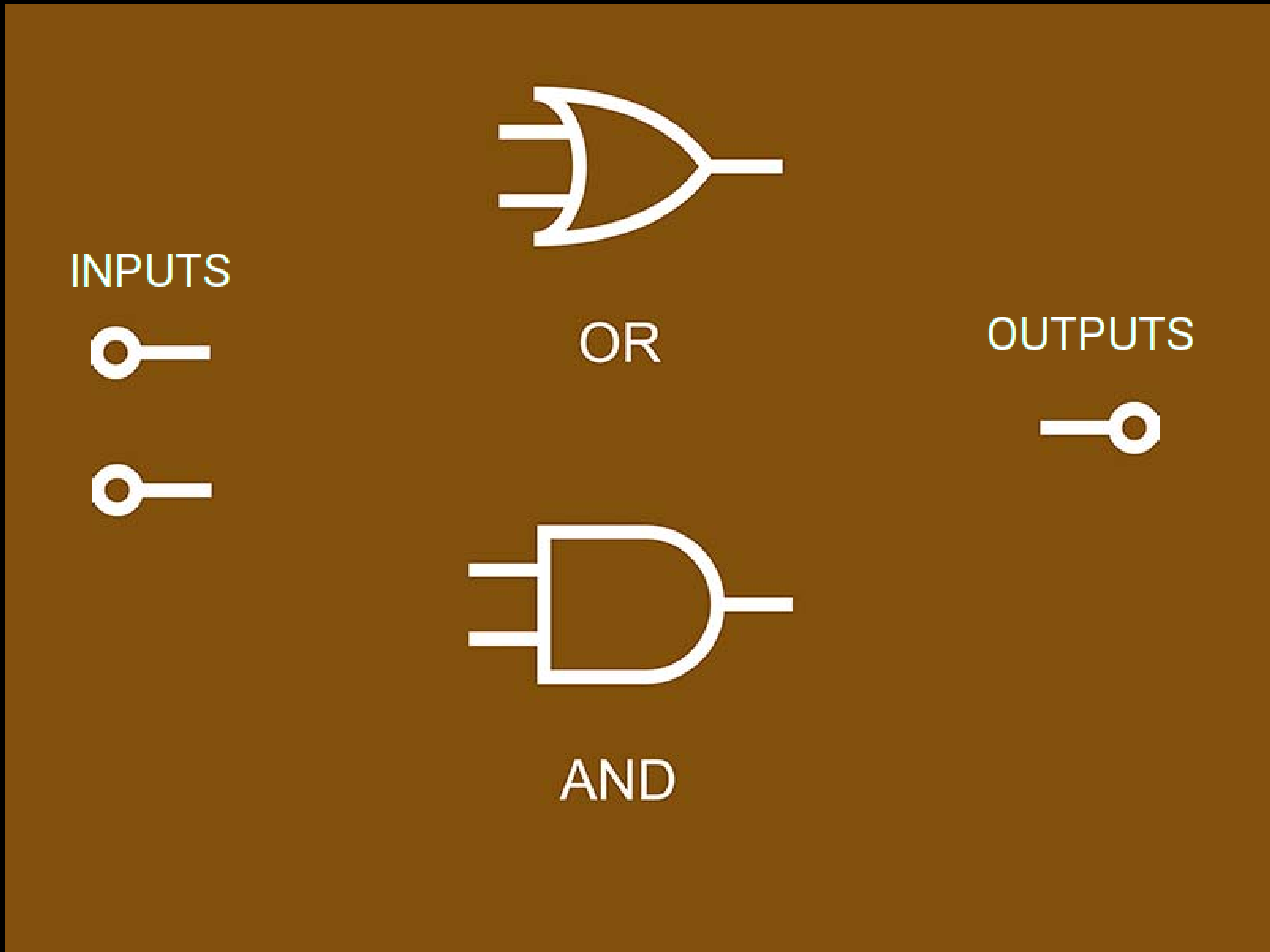
an integrated circuit consisting of

- a large number of blocks with logic gates
- connected by a programmable interconnection fabric
- accessible through I/O blocks

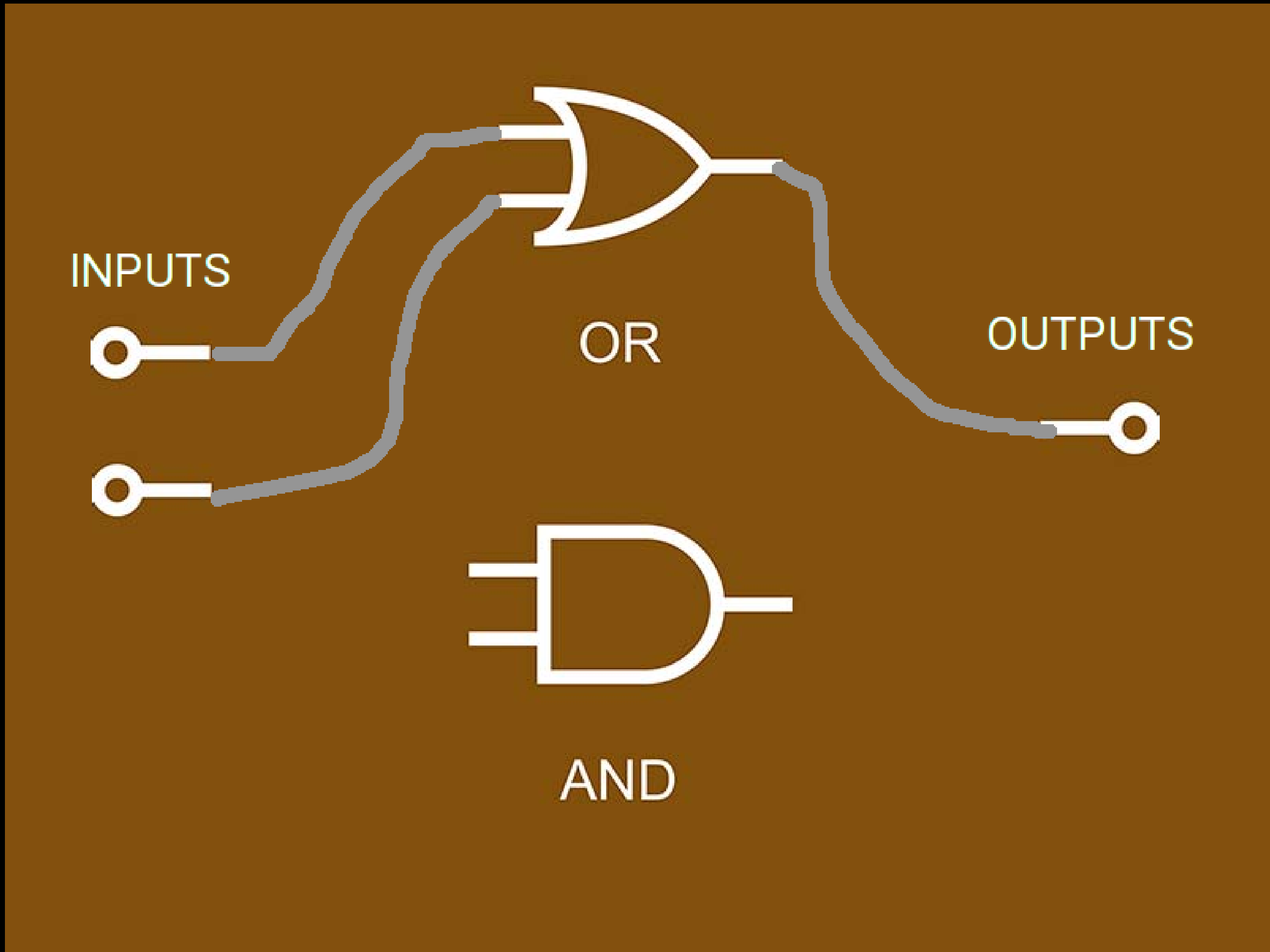


Program your own electronic circuit onto this chip, anywhere, anytime! (within available resources)

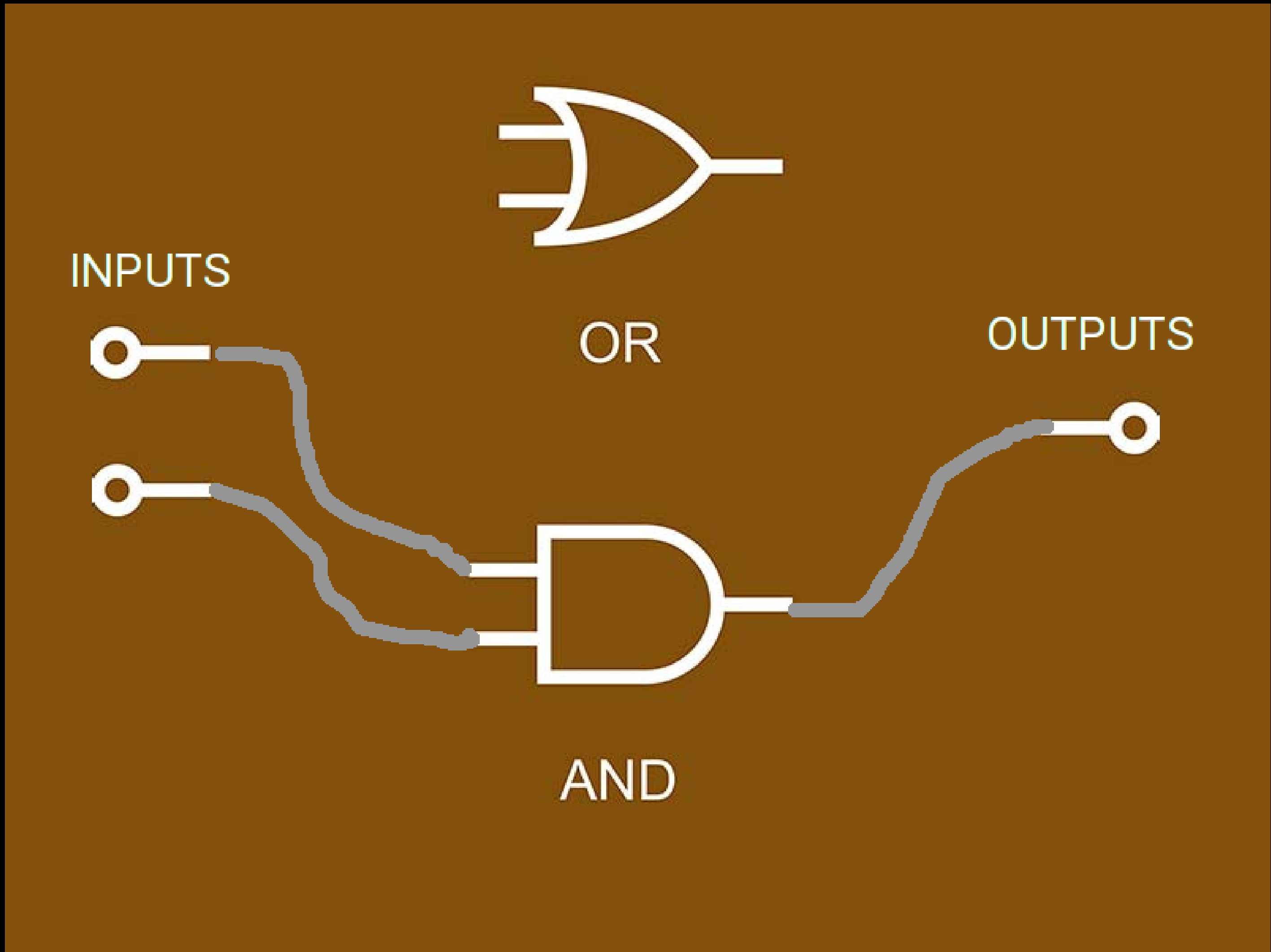
FPGA: trivial example



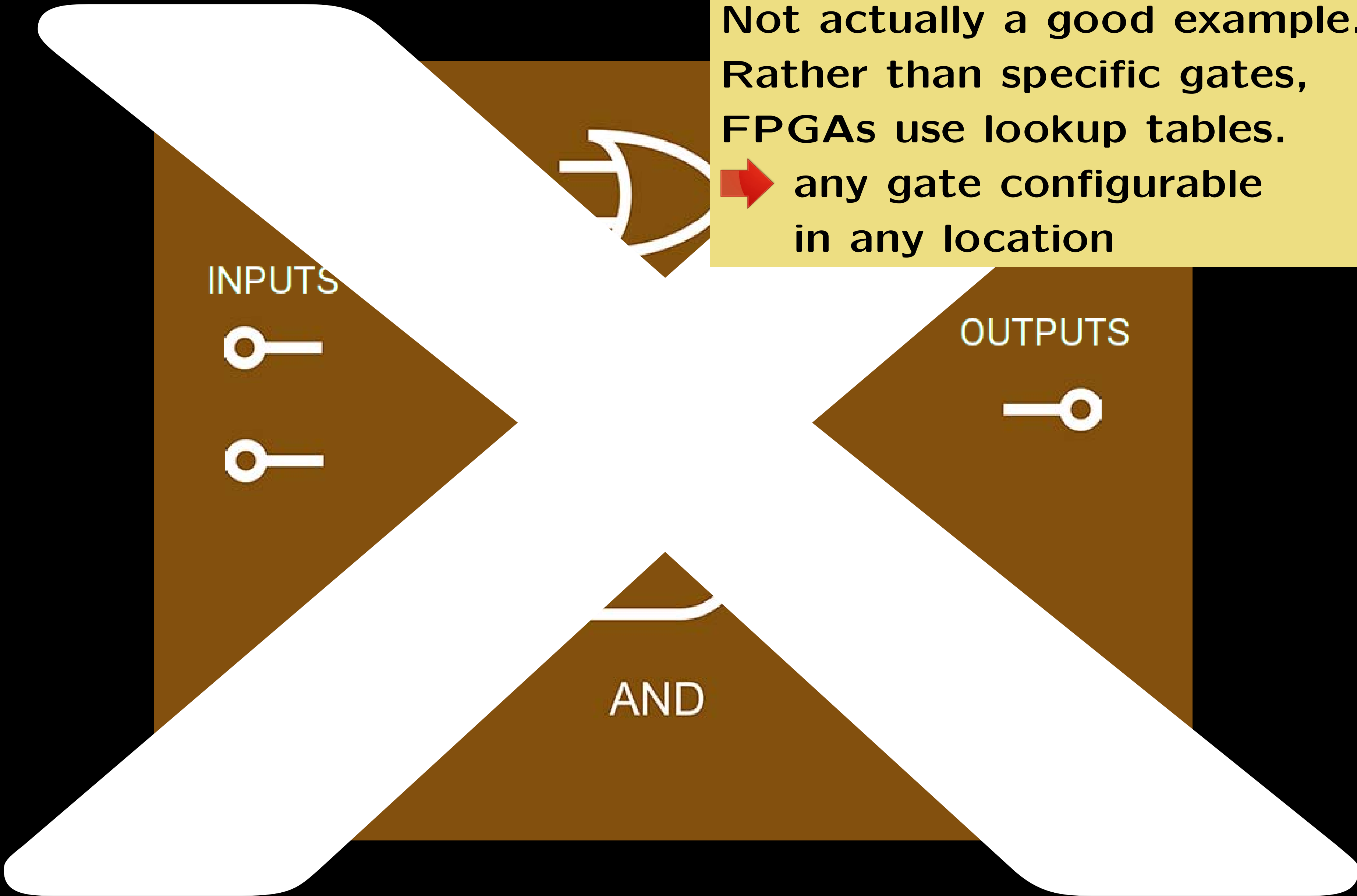
FPGA: trivial example



FPGA: trivial example



FPGA: trivial example



Not actually a good example.
Rather than specific gates,
FPGAs use lookup tables.
➡ any gate configurable
in any location

lookup tables

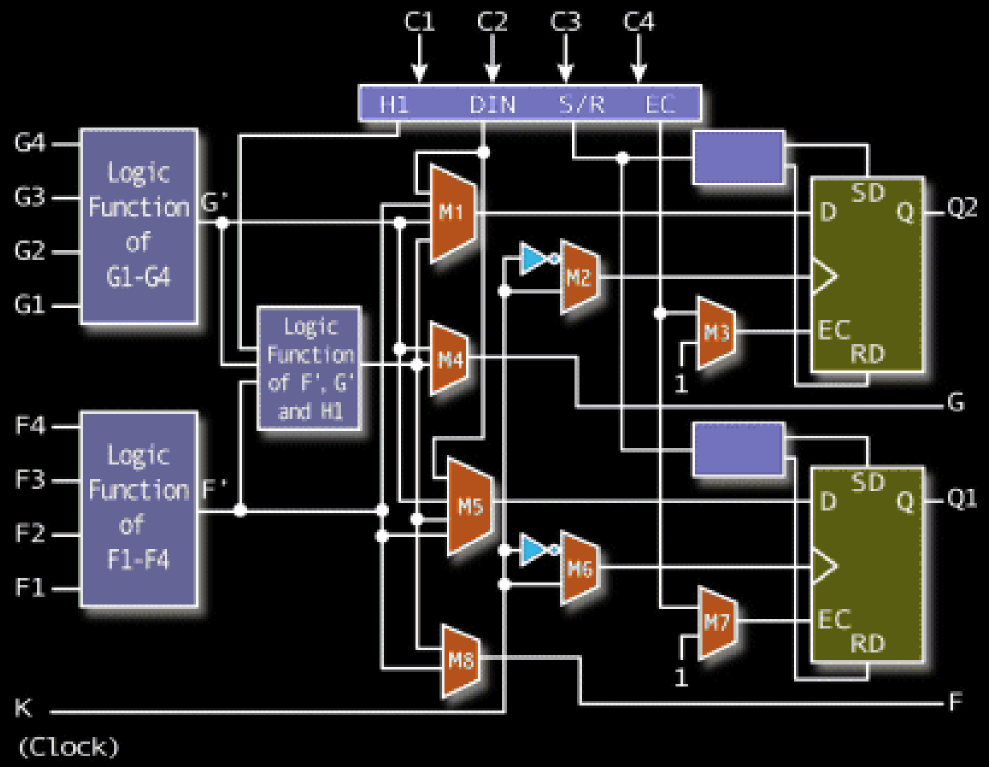
A look-up table is a small memory bank that encodes a general logic function:

input A	input B	input C	output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

LUTs can be programmed to act as basic logic gates (AND, OR, etc),
but also as complex combinations.

modern logic blocks

Configurable logic block by Xilinx (probably outdated):

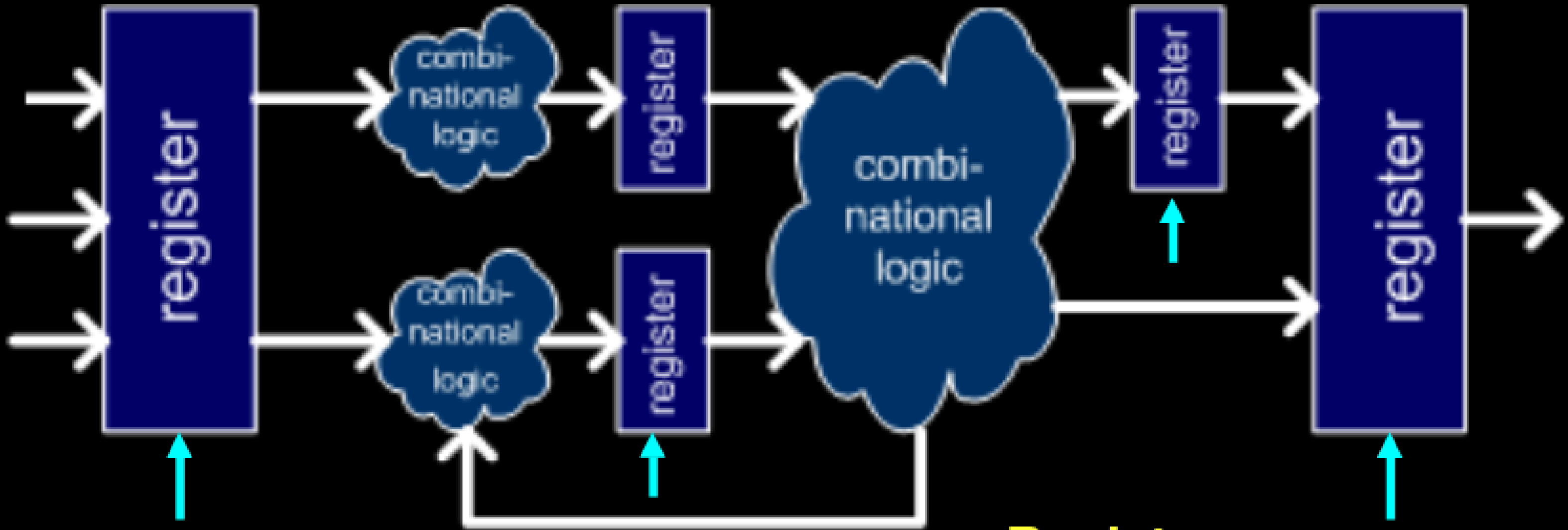


- look-up tables to manipulate inputs
- multiplexers to route the signals
- flip-flops (clocked storage devices) to hold the outputs
- multiple blocks running on same clock for synchronous operation

synchronous sequential logic

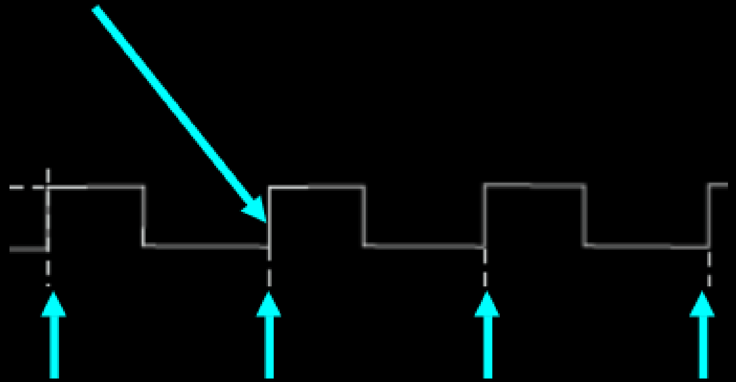
intermediate logic

inputs



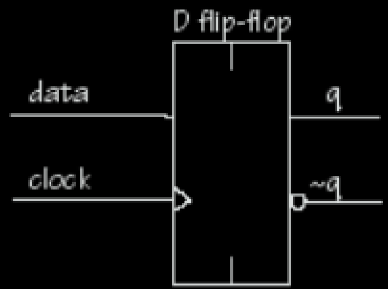
new output every clock edge

CLOCK



EDGES

Register

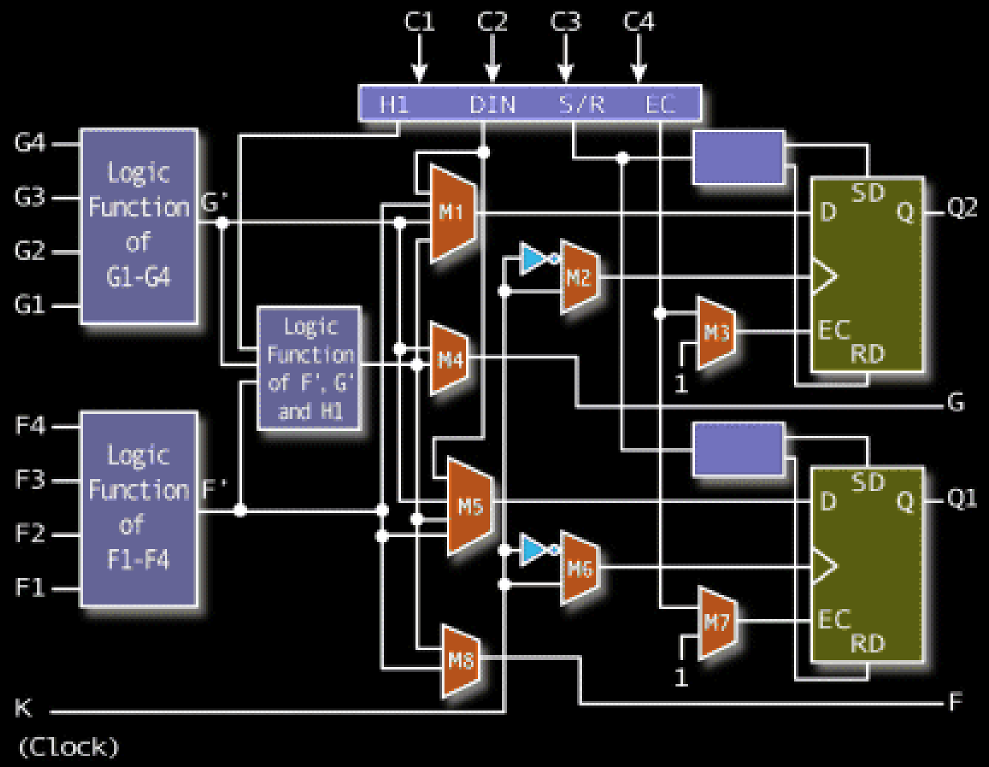


- clock rate → speed
- combinational logic must meet timing for predictable behaviour

graph by Edward Freeman (STFC)

modern logic blocks

Configurable logic block by Xilinx (probably outdated):



other blocks on modern FPGAs:

high speed transceivers, PCIe interfaces, ethernet interfaces, memory banks, clock generators, DSPs, interfaces for external RAM, even entire CPUs... (but typically digital electronics only)

modern FPGAs

example of resources available on current generation FPGAs: Xilinx Virtex Ultrascale+ devices

Device Name	VU3P	VU5P	VU7P	VU9P	VU11P	VU13P	VU19P
System Logic Cells (K)	862	1,314	1,724	2,586	2,835	3,780	8,938
CLB Flip-Flops (K)	788	1,201	1,576	2,364	2,592	3,456	8,172
CLB LUTs (K)	394	601	788	1,182	1,296	1,728	4,086
Max. Dist. RAM (Mb)	12.0	18.3	24.1	36.1	36.2	48.3	58.4
Total Block RAM (Mb)	25.3	36.0	50.6	75.9	70.9	94.5	75.9
UltraRAM (Mb)	90.0	132.2	180.0	270.0	270.0	360.0	90.0
HBM DRAM (GB)	–	–	–	–	–	–	–
HBM AXI Interfaces	–	–	–	–	–	–	–
Clock Mgmt Tiles (CMTs)	10	20	20	30	12	16	40
DSP Slices	2,280	3,474	4,560	6,840	9,216	12,288	3,840
Peak INT8 DSP (TOP/s)	7.1	10.8	14.2	21.3	28.7	38.3	10.4
PCIe® Gen3 x16	2	4	4	6	3	4	0
PCIe Gen3 x16/Gen4 x8 / CCIX(1)	–	–	–	–	–	–	8
150G Interlaken	3	4	6	9	6	8	0
100G Ethernet w/ KR4 RS-FEC	3	4	6	9	9	12	0
Max. Single-Ended HP I/Os	520	832	832	832	624	832	1,976
Max. Single-Ended HD I/Os							96
GTY 32.75Gb/s Transceivers	40	80	80	120	96	128	80

NB: very difficult to use anywhere near 100% of those resources due to limitations of the interconnection fabric

CPU vs FPGA

CPUs and FPGAs are capable of performing arbitrary tasks depending on programming.
But the approach is fundamentally different:

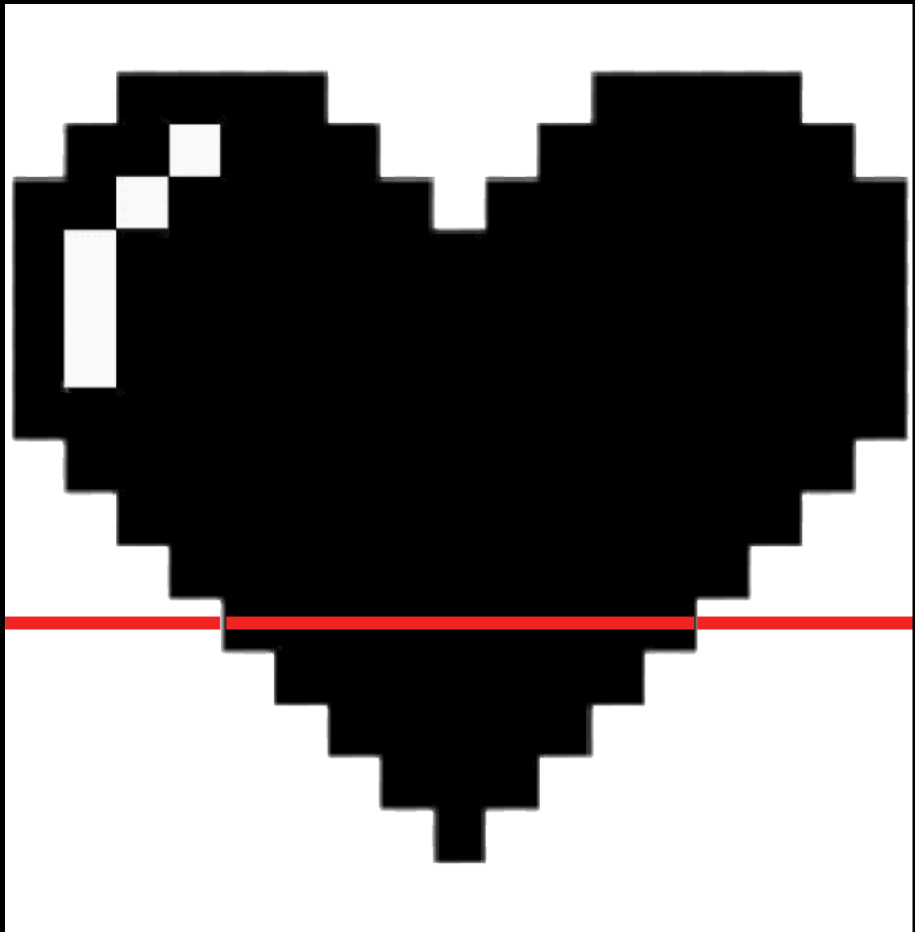
CPU
rigid silicon — the processing units remain fixed (except basics like enabling/disabling cores in multicore processors) Flexibility from stepping through instructions provided in memory

FPGA
flexibility arises from reconfiguring the fabric itself, producing a highly specialised processing unit

- very different type of device
- major differences in how they are being programmed
- suitable for different types of application

CPU vs FPGA

example: edge detection in histogram (e.g. line of video pixels)



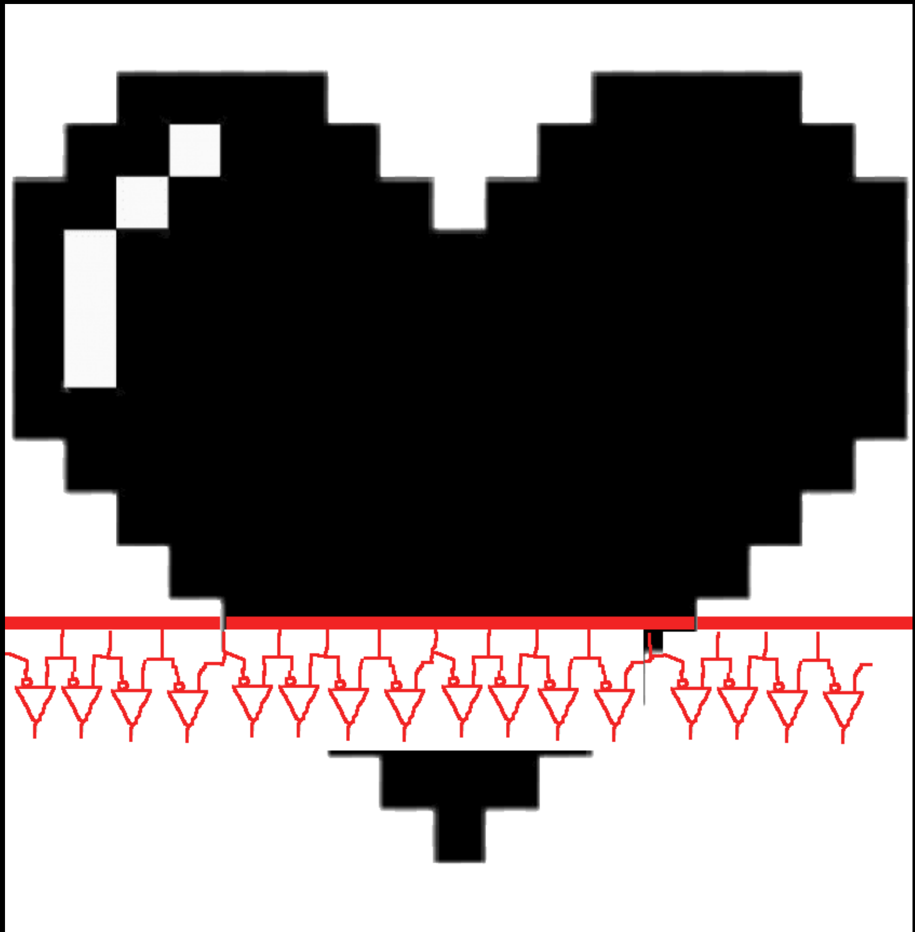
CPU

scan a line like

```
for i in range(1,17):
   edge[i] = (abs(hist[i]-hist[i-1])>thres)
```

CPU vs FPGA

example: edge detection in histogram (e.g. line of video pixels)



CPU

scan a line like

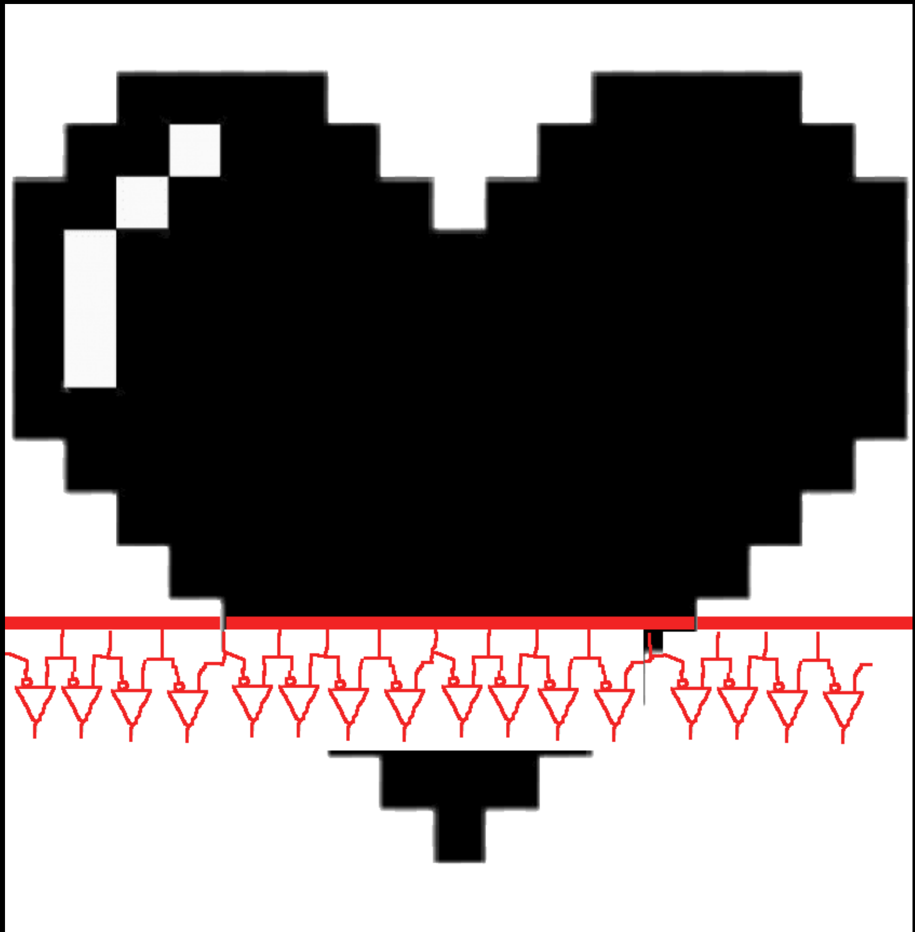
```
for i in range(1,17):
    edge[i] = (abs(hist[i]-hist[i-1])>thres)
```

FPGA

instantiate a bunch of comparators, get result in O(1) clock cycle

CPU vs FPGA

example: edge detection in histogram (e.g. line of video pixels)



CPU

scan a line like

```
for i in range(1,17):
   edge[i] = (abs(hist[i]-hist[i-1])>thres)
```

FPGA

instantiate a bunch of comparators, get result in O(1) clock cycle

➡ FPGA benefits from parallel instantiation of a large number of specialised logic circuits

NB: GPUs are somewhat in the middle — massive parallelisation with simplified CPUs

FPGA applications

FPGAs offer advantages over CPUs for specific applications:

- high degree of parallelisation, pipelining
- many high speed data links
- precise control over data path → fixed latency
- low latency (if done right)

FPGAs have weaknesses too:

- complex arithmetic (floating point numbers etc)
- cost (depending on parameters)

FPGA applications

Use of FPGAs in particle physics

- L1 trigger
- DAQ
- clock distribution (incl fast commands, triggers)

FPGA use elsewhere

- high performance computing: FPGAs supporting CPUs
- aerospace, automotive, telecommunications, BitCoin mining
- prototyping of new ASICs

FPGA applications

Use of FPGAs in particle physics

- L1 trigger
- DAQ
- clock distribution (incl fast commands, triggers)

FPGA use elsewhere

- high performance computing: FPGAs supporting CPUs
- aerospace, automotive, telecommunications, BitCoin mining
- prototyping of new ASICs

Note on ASICs (Application Specific Integrated Circuits):
actual custom chips can have higher speed, higher density, lower power, more resources, can be cheaper in large quantities, require no in situ programming
BUT: much longer time to availability, mistakes are expensive
➡ we typically use them only in detector front-end

FPGA vendors

AMD | XILINX: about 50% market share, full range incl high end

intel FPGA: formerly Altera, about 35% market share

LATTICE SEMICONDUCTOR: low power, low cost devices

QuickLogic: low power FPGA/ASIC hybrids

Microsemi: low power, radiation hard, non-volatile (flash based)

...and probably others!

This lecture focusing on Xilinx — used by CMS-UK, ATLAS-UK

commercial FPGA platforms

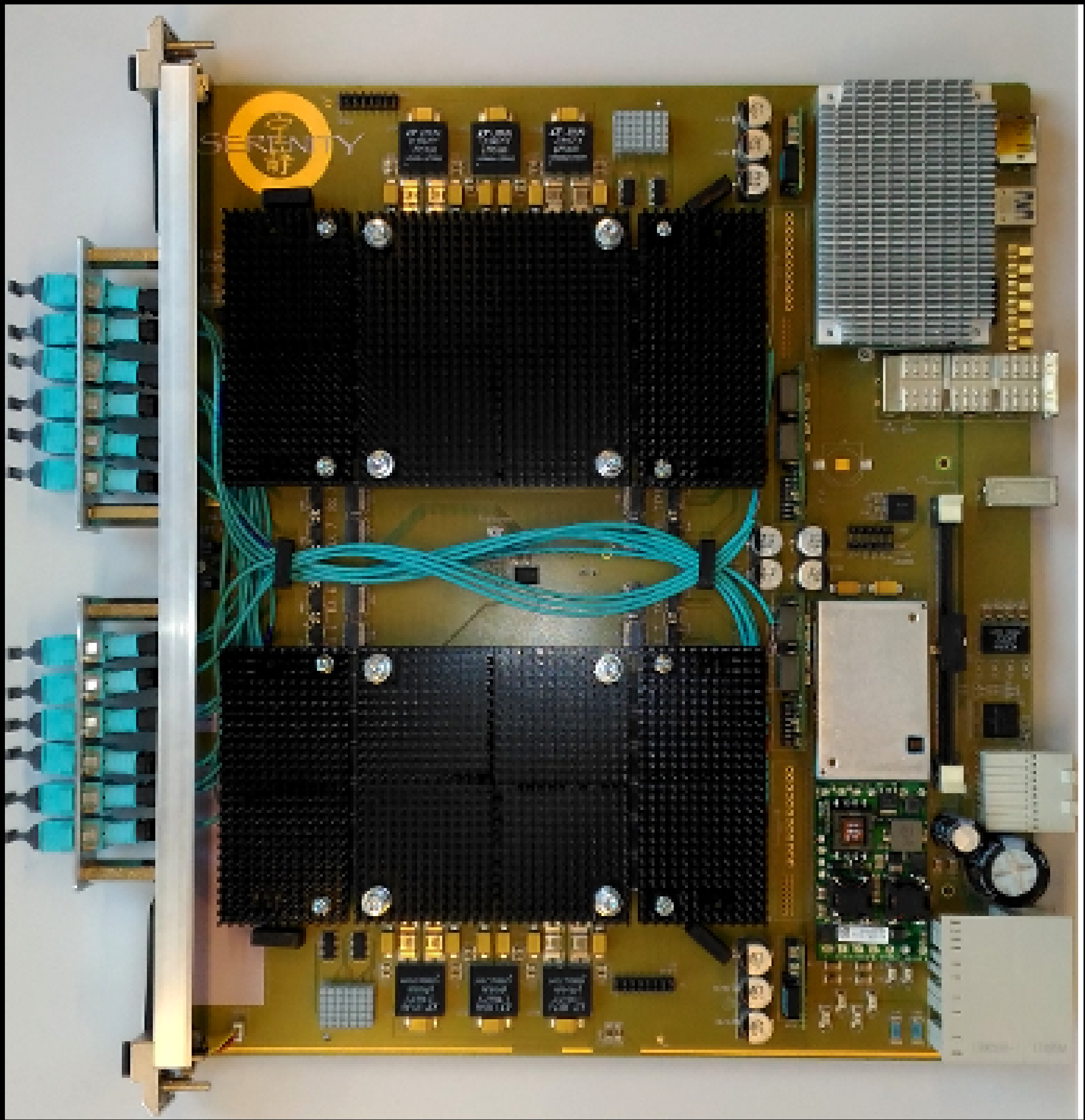
examples for commercially available FPGA boards:

FPGA accelerator card based on Altera device

FPGA RF/optical I/O card based on Xilinx device

custom hardware

Imperial College's Serenity board



- multi-purpose board
- mostly for CMS upgrade
- two FPGA sites
- FPGAs easily replaceable
- optical high speed links
- ATCA form factor
- comes with single board PC

development boards

FPGA manufacturers provide development platforms for their FPGAs:
FPGA on circuit board with peripherals and infrastructure

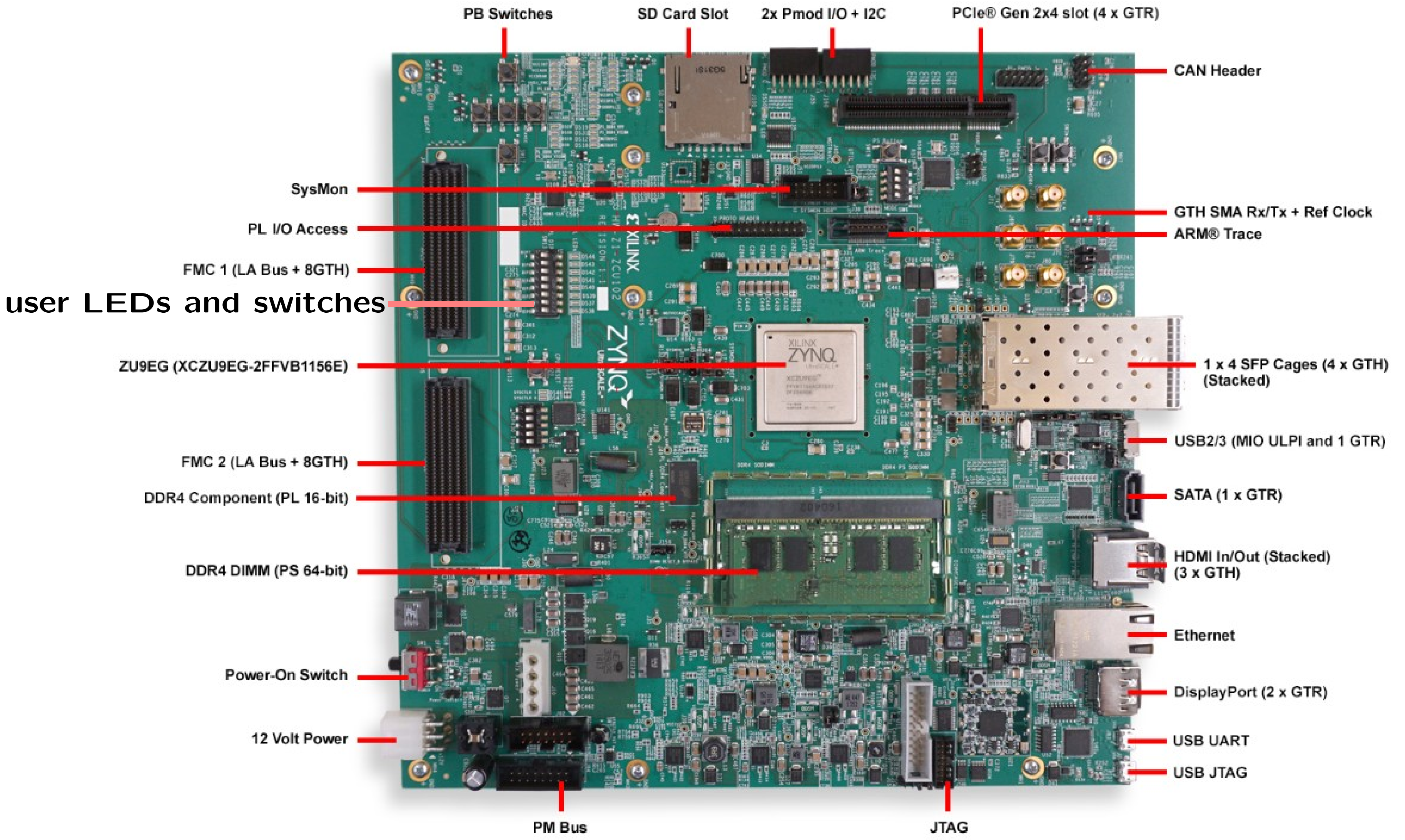
benefits:

- often available very early on after release of new devices
- often relatively low cost because subsidised

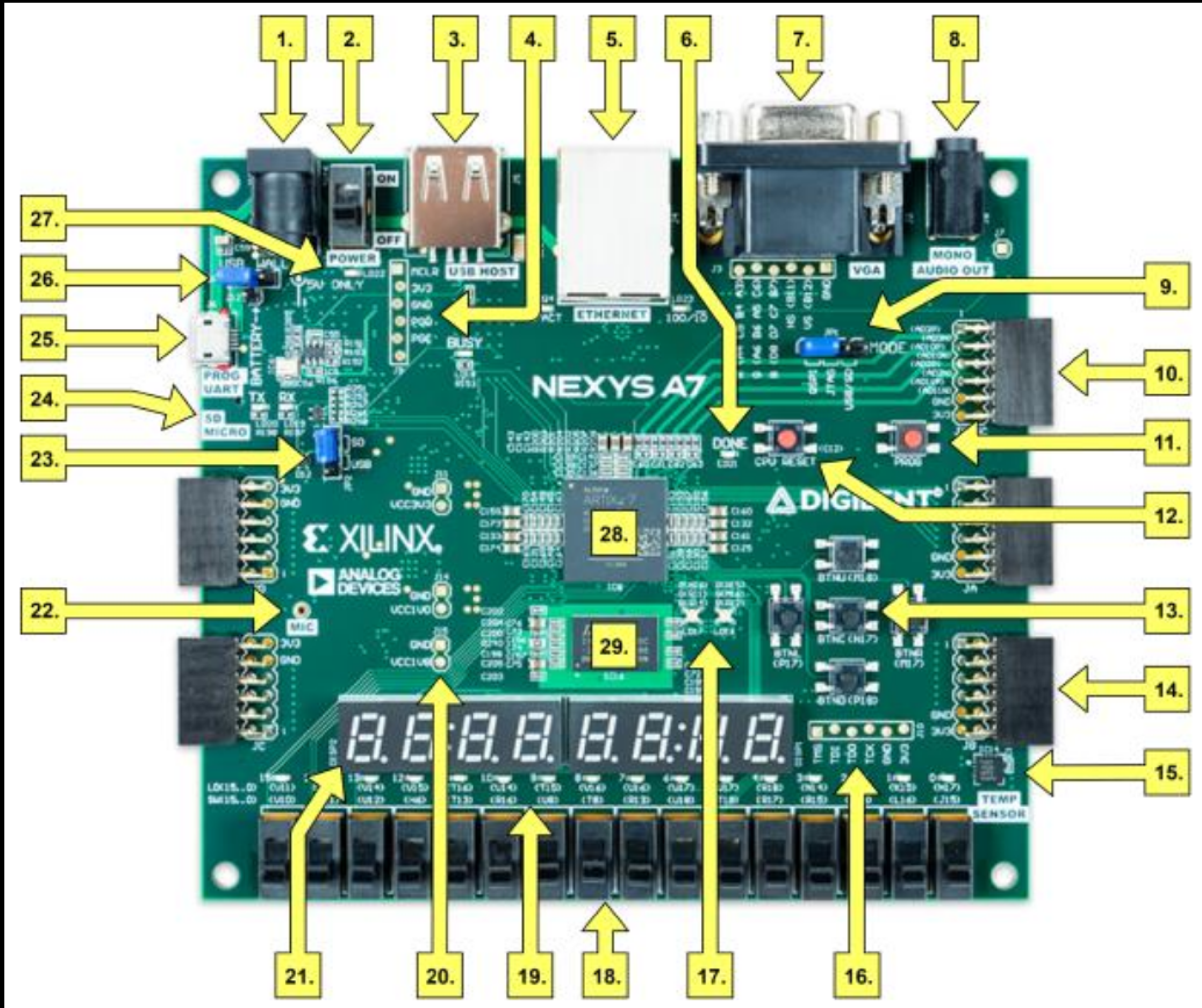
In widespread use in our labs!

- tested algorithms for Serenity long before prototypes available
- experience can actually influence custom board design
- ideal for learning: availability, example designs

Xilinx zcu102



Digilent Nexys A7



How to program an FPGA

The set of configuration instructions for an FPGA

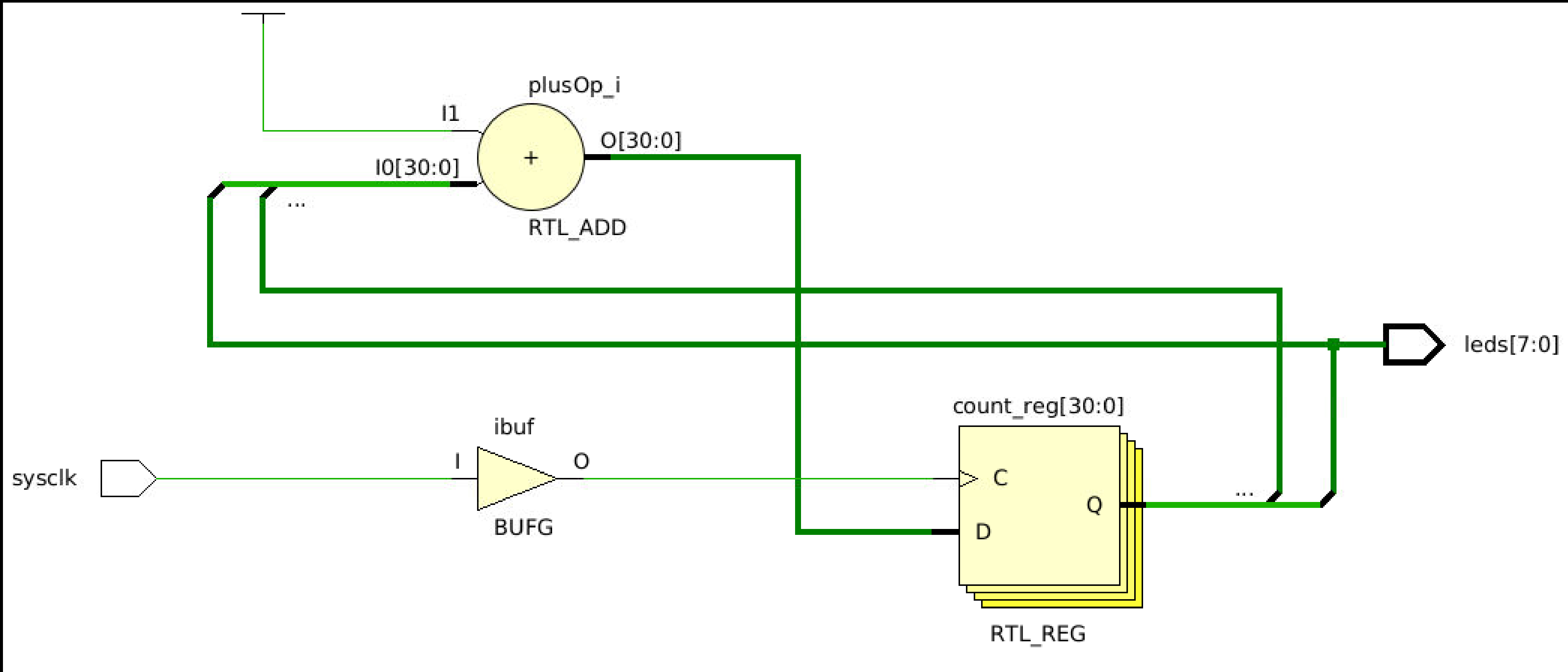
- is not actually modifying hardware,
- but it is not a software algorithm either.

It is somewhat in between, which is why it is called

Description either graphically as **schematics**,
or in a **hardware description language**.

hardware description language looks similar to software, but is very different.

schematics



We won't use schematics today, but we will implement this design.
Note inputs, outputs, blocks, signals, busses, constants, and a loop!
Relatively easy to understand, but not very practical for complex tasks.

Science and Technology Facilities Council

hardware description languages

Two main languages in use:

	VHDL	Verilog
resemblance	Pascal/Ada	C
strong types	yes	no
composite data types	yes	no
case sensitive	no	yes
library management	yes	no
who in CMS likes it	Brits	Americans

VHDL:

```
process ({S0,S1},A,B,C,D)
begin
    case {S0,S1}, is
        when "00" => Y <= A;
        when "01" => Y <= B;
        when "10" => Y <= C;
        when "11" => Y <= D;
        when others => Y <= A;
    end case;
end process;
```

Verilog:

```

always @({S0,S1}, A, B, C, D)
    case ({S0,S1})
        2'b00: Y = A;
        2'b01: Y = B;
        2'b10: Y = C;
        2'b11: Y = D;
    endcase
```

from blog.digilentinc.com

hardware description languages

Two main languages in use:

	VHDL	Verilog
resemblance	Pascal/Ada	C
strong types	yes	no
composite data types	yes	no
case sensitive	no	yes
library management	yes	no
who in CMS likes it	Brits	Americans

We use VHDL in our projects because

- complex data types make interfaces easier to read (and write)
- strong typing reduces margin for error
- it does seem to be a bit easier to read

Firmware design is intrinsically modular.
Can mix Verilog, VHDL and schematic design in one project.
(Prefer not to.)

example project

Let's make some LEDs blink on our Nexys A7 development board!
We will need:

LEDs

- Nexys A7 has a set of 16 user LEDs connected to FPGA I/O pins
- pin location and required voltage levels are documented

clock

- all development boards have oscillators
- some connected to FPGA directly
- some connected through programmable clock chips
- high speed clock signals are often differential

firmware

- VHDL design discussed on following pages

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk    : in  STD_LOGIC;
           leds      : out STD_LOGIC_VECTOR (7 downto 0)
     );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
           count <= count + 1;
        end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;
```

This is all the VHDL we need today!

- This block connects to a 100 MHz clock input,
- sends the clock signal through a buffer,
- runs a 31 bit counter on that clock (highest bit should then alternate at about 0.1 Hz),
- and connects the highest (i.e. slowest) bits to LEDs

➡ Let’s look at the code in detail

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk   : in  STD_LOGIC;
           leds     : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

⟵ This is a comment

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
            sysclk   : in  STD_LOGIC;
            leds     : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

} ⟵ Load packages from libraries

- IEEE.std_logic_1164 has types for logic signals
- IEEE.numeric_std has numeric data types
- unisim.VComponents has declarations and simulation data for device-specific primitives

top level block

```vhdl
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
            sysclk    : in  STD_LOGIC;
            leds      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
           count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

} ← **Declare a VHDL block with ports**

- We give this block a name (top)
- and define connections (ports) to the outside
- top level ports correspond to actual FPGA I/O pins

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk   : in  STD_LOGIC;
           leds     : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

← Describe the VHDL block

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
             sysclk    : in  STD_LOGIC;
             leds      : out STD_LOGIC_VECTOR (7 downto 0)
        );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

} ⟵ Declare internal signals we need

- consider signals more like wires, not as variables
- can assign an initial state, though

top level block

```vhdl
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk   : in  STD_LOGIC;
           leds     : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;
```

← **Instantiate a different block**

- this one is from a library
- it buffers an incoming clock for distribution
- we name this instance ibuf
- we connect it to our signals

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk    : in  STD_LOGIC;
           leds      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;
```

a process

- runs when specific events occur
- here: rising edge of clk
- allocate incremented value to count
- (almost like software, isn't it?)

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
            sysclk    : in  STD_LOGIC;
            leds      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

←

connecting counter bits with LED

- this is NOT a one time assignment
- it connects signals like wires
- every change in count will change the state of leds

top level block

```
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk    : in  STD_LOGIC;
           leds      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

one missing ingredient:

our design software needs to be told what pins clock and LEDs are connected to and what logic standard to use

➡ define <u>constraints</u> (separate file)

```
# Nexys A7 constraints file
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

# System clock (100MHz)
set_property IOSTANDARD LVCMOS33 [get_ports {sysclk}]
set_property PACKAGE_PIN E3 [get_ports sysclk]
create_clock -period 10 -name sysclk [get_ports sysclk]

# LEDs
set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
set_property PACKAGE_PIN H17 [get_ports {leds[0]}]
set_property PACKAGE_PIN K15 [get_ports {leds[1]}]
set_property PACKAGE_PIN J13 [get_ports {leds[2]}]
set_property PACKAGE_PIN N14 [get_ports {leds[3]}]
set_property PACKAGE_PIN R18 [get_ports {leds[4]}]
set_property PACKAGE_PIN V17 [get_ports {leds[5]}]
set_property PACKAGE_PIN U17 [get_ports {leds[6]}]
set_property PACKAGE_PIN U16 [get_ports {leds[7]}]
```

(this information from Nexys A7 documentation)

firmware workflow

There is a number of steps between VHDL and a blinking LED:

synthesis
translate VHDL into netlist (optimised components with connections)

implementation
map design onto actual FPGA resources, assign place to entities and route signals along the fabric

bitfile generation
create actual bitstream that can be uploaded to device

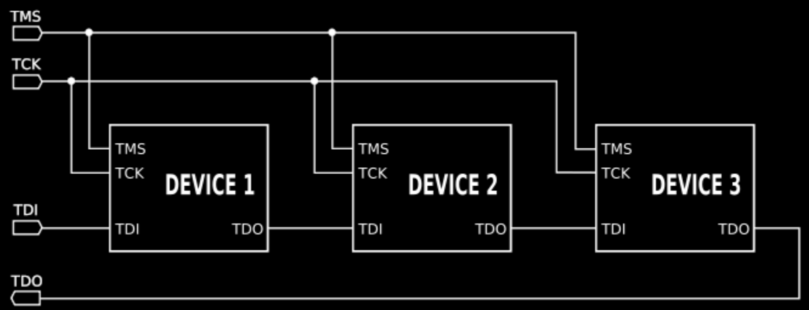
JTAG configuration
connect to device via serial JTAG interface and configure it! most high-end FPGAs use volatile RAM to store configuration $\rightarrow$ need to reconfigure with JTAG after each power-up or store firmware in external flash ROM

JTAG

JTAG interface can have several devices in series

- e.g. multiple FPGAs
- or a FPGA and a flash ROM for non-volatile firmware storage

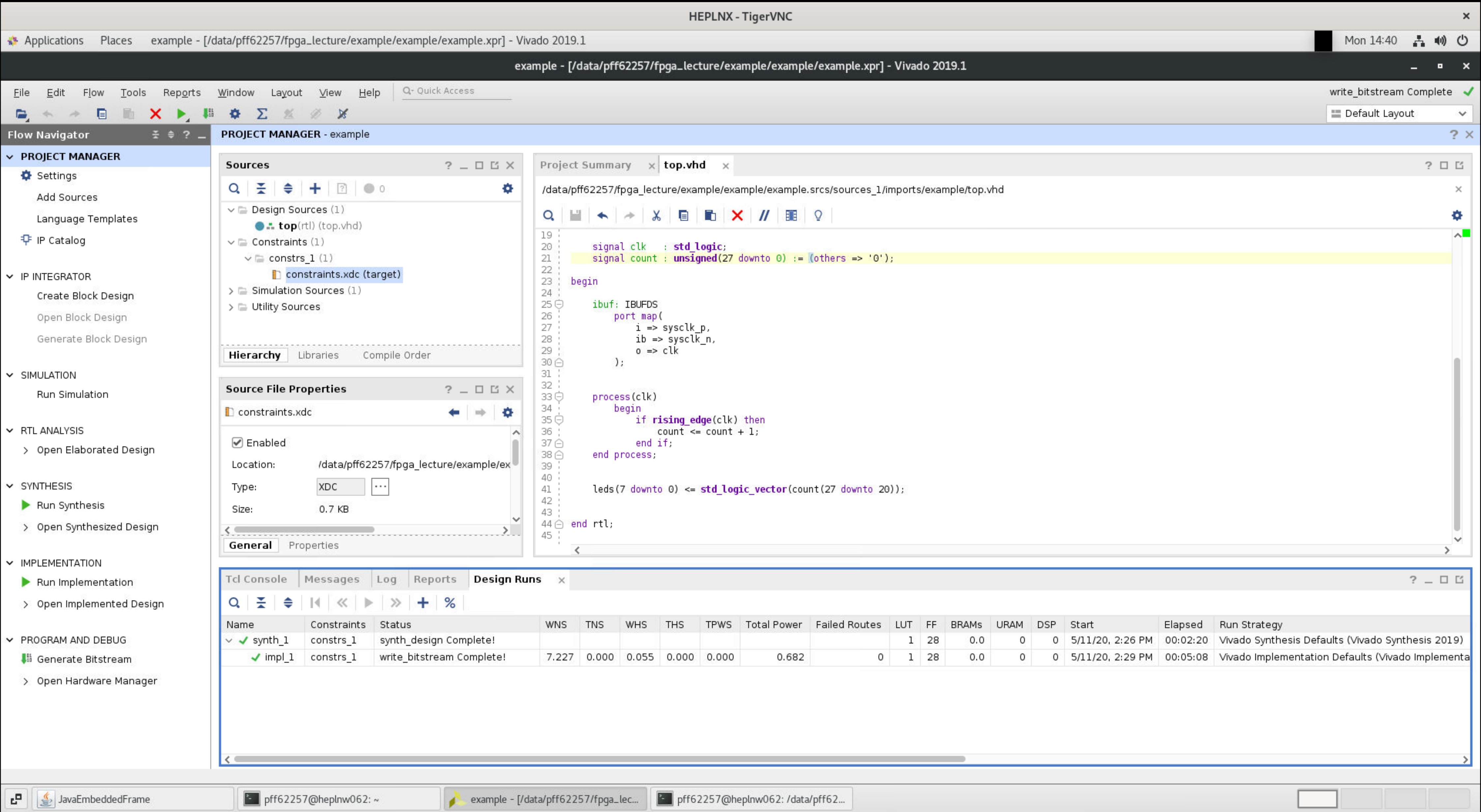
All devices identify themselves, so software can verify expected type



Some boards have built-in JTAG controllers, just need USB cable.
Others need external USB programmers:

JTAG connection can be used for debugging! Logic analyser cores

Xilinx Vivado



Vivado report

Report after building our example firmware for the Nexys A7:

Settings Edit

Project name:	nexys_a7
Project location:	/net/home/ppd/pff62257/fpga_tutorial/nexys_a7
Product family:	Artix-7
Project part:	xc7a100tcsg324-1
Top module name:	top
Target language:	VHDL
Simulator language:	Mixed

Synthesis

Status:	✔ Complete
Messages:	1 warning
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	Automatically selected checkpoint

Implementation **Summary** | Route Status

Status:	✔ Complete
Messages:	2 warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

No DRC violations were found.

Implemented DRC Report

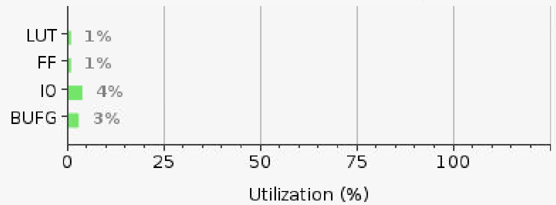
Timing **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS):	7.1 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	31

Implemented Timing Report

Utilization Post-Synthesis | **Post-Implementation**

Graph | Table



Power **Summary** | On-Chip

Total On-Chip Power:	**0.104 W**
Junction Temperature:	**25.5 °C**
Thermal Margin:	59.5 °C (12.9 W)
Effective θJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

simulation

More complex firmware is best tested in simulation first

A very powerful tool for verification and debugging!

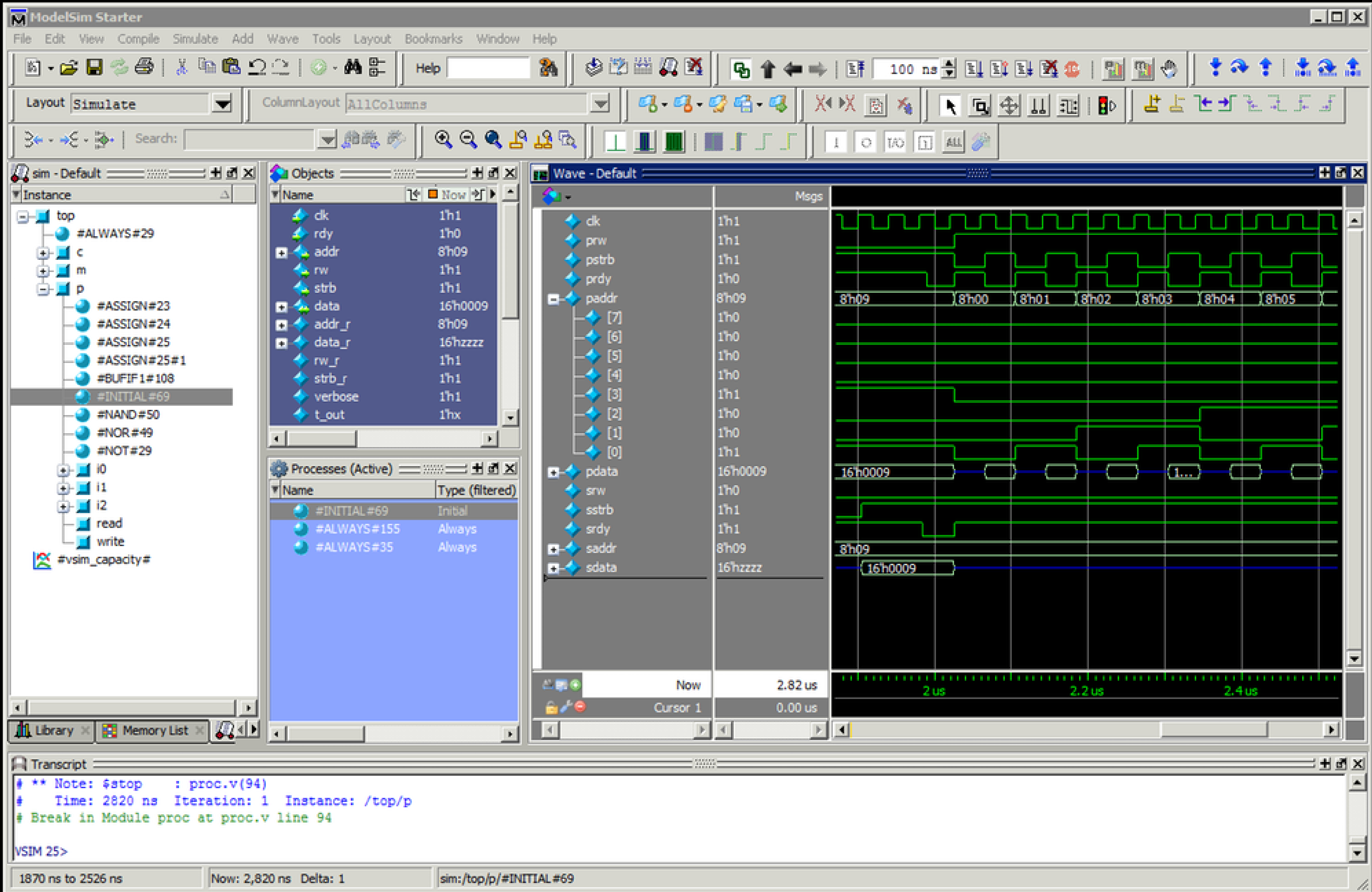
- exactly reproducible inputs
- much faster turnaround time than tests on hardware
- but not always a fully accurate reflection of timing, especially when timing is marginal or outside specifications

➡ comparison of firmware output on simulation and on actual hardware is often part of verification procedure

Vivado has an integrated simulator

Third party software exists (e.g. Siemens/Mentor Graphics QuestaSim)

simulation



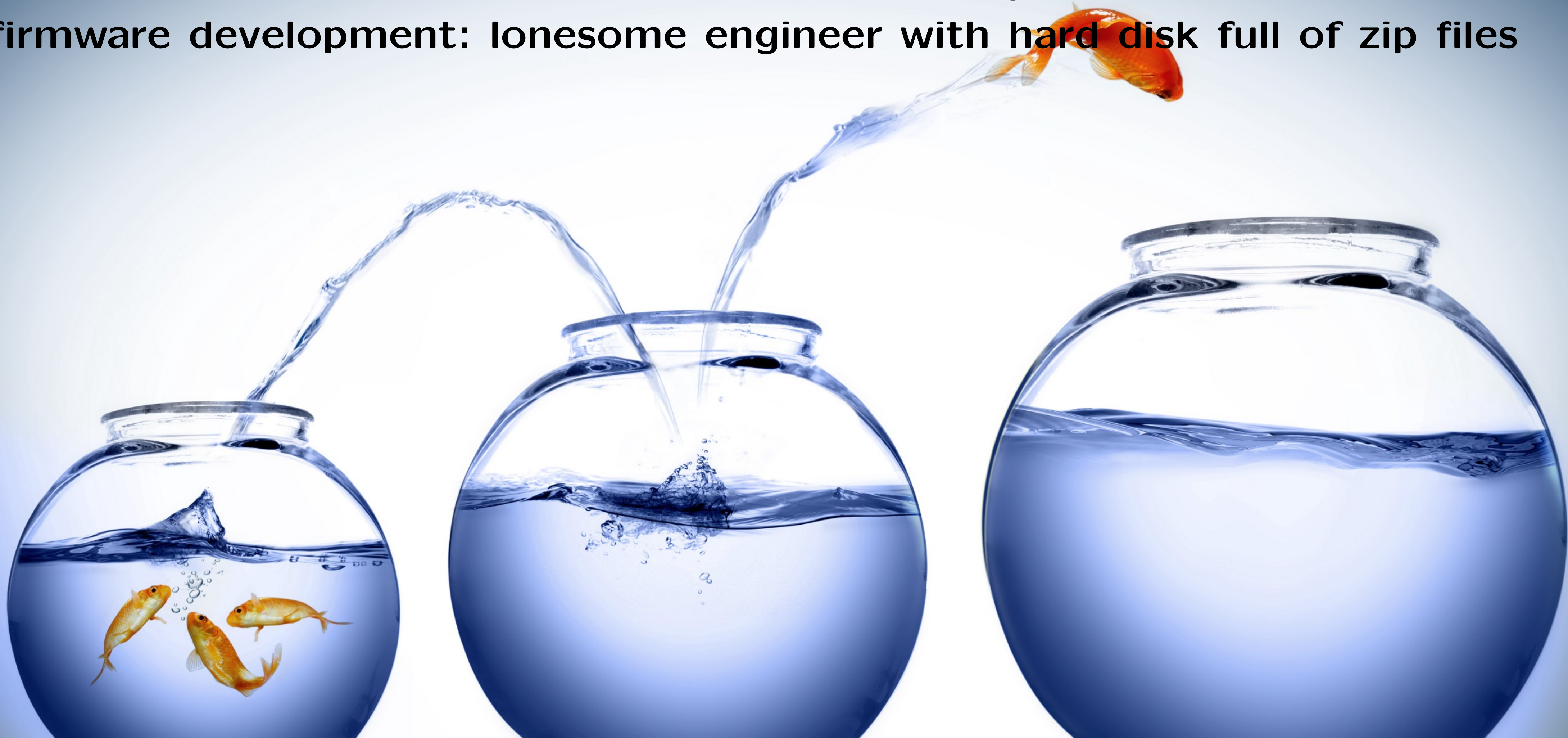
ModelSim Starter
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Layout Simulate
ColumnLayout AllColumns
sim - Default
Instance
top
#ALWAYS#29
c
m
p
#ASSIGN#23
#ASSIGN#24
#ASSIGN#25
#ASSIGN#25#1
#BUFIF1#108
#INITIAL#69
#NAND#50
#NOR#49
#NOT#29
i0
i1
i2
read
write
#vsim_capacity#
Library
Memory List
Objects
Name
clk 1'h1
rdy 1'h0
addr 8'h09
rw 1'h1
strb 1'h1
data 16'h0009
addr_r 8'h09
data_r 16'hzzzz
rw_r 1'h1
strb_r 1'h1
verbose 1'h1
t_out 1'hx
Processes (Active)
Name Type (filtered)
#INITIAL#69 Initial
#ALWAYS#155 Always
#ALWAYS#35 Always
Wave - Default
Msgs
clk 1'h1
prw 1'h1
pstrb 1'h1
prdy 1'h0
paddr 8'h09
[7] 1'h0
[6] 1'h0
[5] 1'h0
[4] 1'h0
[3] 1'h1
[2] 1'h0
[1] 1'h0
[0] 1'h1
pdata 16'h0009
srw 1'h0
sstrb 1'h1
srdy 1'h1
saddr 8'h09
sdata 16'hzzzz
8'h09 8'h00 8'h01 8'h02 8'h03 8'h04 8'h05
16'h0009
16'h0009
Now 2.82 us
Cursor 1 0.00 us
2 us 2.2 us 2.4 us
Transcript
** Note: $stop : proc.v(94)
Time: 2820 ns Iteration: 1 Instance: /top/p
Break in Module proc at proc.v line 94
VSIM 25>
1870 ns to 2526 ns
Now: 2,820 ns Delta: 1
sim:/top/p/#INITIAL#69

scaling up

last topic: how to approach a BIG firmware project

software development: distributed, collaborative, version controlled,
unit tests, release management

firmware development: lonesome engineer with hard disk full of zip files

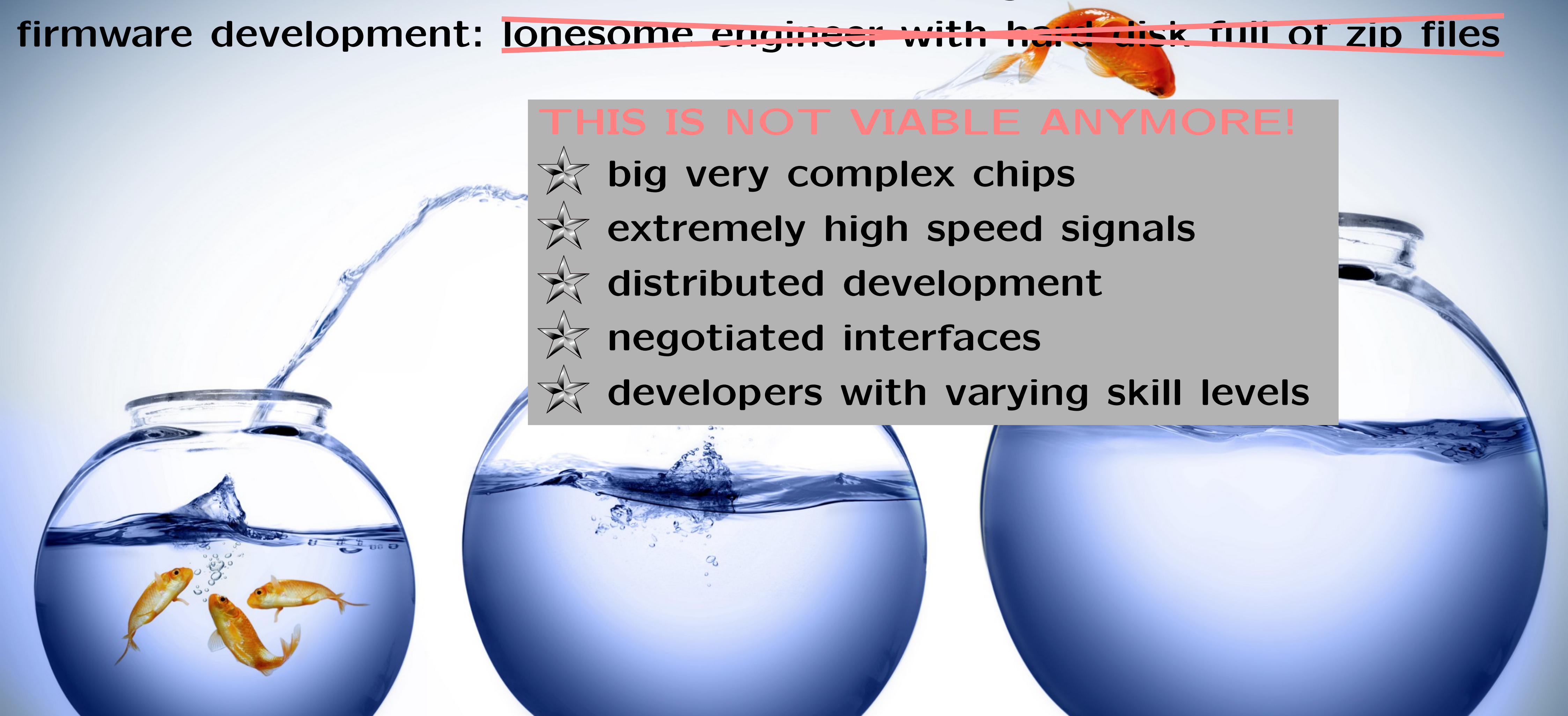


scaling up

last topic: how to approach a BIG firmware project

software development: distributed, collaborative, version controlled, unit tests, release management

firmware development: ~~lonesome engineer with hard disk full of zip files~~



THIS IS NOT VIABLE ANYMORE!

- big very complex chips
- extremely high speed signals
- distributed development
- negotiated interfaces
- developers with varying skill levels

large firmware projects

Firmware projects like CMS L1 trigger are very demanding:

- very complex
- need to be very reliable
- subject to international collaboration and peer review

Need to work much more like with large software projects:

- modularity (leave the hardcore stuff to top experts)
- version control and release management
- rigorous testing, project supervision

CMS L1 trigger firmware project:

- separate framework and algorithm firmware
- script-based firmware build system (also enforces module structure)
- git repository (with automatic nightly builds)
- formal developer and user support (ticket system)

→ very successful model, proven in LHC run 2 already

conclusion

Key points:

- ☆ FPGAs are a very powerful tool for low latency high throughput applications
- ☆ FPGA programming by firmware has many similarities with software development, but important differences

We are moving more and more functionality into FPGAs, e.g. in L1 trigger.
We have a lot more people who know how to write software than how to write firmware. This has to change.

I hope I demonstrated today that writing firmware is no voodoo.

next: try it all yourself!

We will have a coffee break now,
then move to R1 PPD lab 6.
We have five PCs with an FPGA board attached
➡ work in groups if needed

We will:

- go through the design+programming step by step
- test the example on an actual FPGA
- maybe try a few modifications
- discuss any questions you might have

Nexys A7 example overview

```vhdl
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
            sysclk   : in  STD_LOGIC;
            leds     : out STD_LOGIC_VECTOR (7 downto 0)
    );
end top;


architecture rtl of top is

  signal clk     : std_logic;
  signal count   : unsigned(30 downto 0) := (others => '0');

begin

   ibuf: BUFG
      port map(
         i => sysclk,
         o => clk
      );



   process(clk)
      begin
        if rising_edge(clk) then
            count <= count + 1;
         end if;
      end process;


   leds <= std_logic_vector(count(30 downto 23));


end rtl;

```

```tcl
# Nexys A7 constraints file
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

# System clock (100MHz)
set_property IOSTANDARD LVCMOS33 [get_ports {sysclk}]
set_property PACKAGE_PIN E3 [get_ports sysclk]
create_clock -period 10 -name sysclk [get_ports sysclk]

# LEDs
set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
set_property PACKAGE_PIN H17 [get_ports {leds[0]}]
set_property PACKAGE_PIN K15 [get_ports {leds[1]}]
set_property PACKAGE_PIN J13 [get_ports {leds[2]}]
set_property PACKAGE_PIN N14 [get_ports {leds[3]}]
set_property PACKAGE_PIN R18 [get_ports {leds[4]}]
set_property PACKAGE_PIN V17 [get_ports {leds[5]}]
set_property PACKAGE_PIN U17 [get_ports {leds[6]}]
set_property PACKAGE_PIN U16 [get_ports {leds[7]}]
```

Nexys A7 example with button

```vhdl
--- simple "hello world" example

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library UNISIM;
use UNISIM.vcomponents.all;


entity top is port(
           sysclk   : in  STD_LOGIC;
           leds     : out STD_LOGIC_VECTOR (7 downto 0);
           button   : in  STD_LOGIC);
end top;


architecture rtl of top is

  signal clk      : std_logic;
  signal count    : unsigned(30 downto 0) := (others => '0');
  signal reset    : std_logic;

begin

   ibuf: BUFG
      port map(
        i => sysclk,
        o => clk
      );


   process(clk)
          begin
            if rising_edge(clk) then
                   if reset = '1' then
                           count <= (others => '0');
                        else
                     count <= count + 1;
                        end if;
                 end if;
          end process;


   leds <= std_logic_vector(count(30 downto 23));

        process(clk)
          begin
            if rising_edge(clk) then
                     reset <= button;
                 end if;
        end process;

end rtl;
```

```tcl
# Nexys A7 constraints file
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

# System clock (100MHz)
set_property IOSTANDARD LVCMOS33 [get_ports {sysclk}]
set_property PACKAGE_PIN E3 [get_ports sysclk]
create_clock -period 10 -name sysclk [get_ports sysclk]

# LEDs
set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
set_property PACKAGE_PIN H17 [get_ports {leds[0]}]
set_property PACKAGE_PIN K15 [get_ports {leds[1]}]
set_property PACKAGE_PIN J13 [get_ports {leds[2]}]
set_property PACKAGE_PIN N14 [get_ports {leds[3]}]
set_property PACKAGE_PIN R18 [get_ports {leds[4]}]
set_property PACKAGE_PIN V17 [get_ports {leds[5]}]
set_property PACKAGE_PIN U17 [get_ports {leds[6]}]
set_property PACKAGE_PIN U16 [get_ports {leds[7]}]

# button
set_property IOSTANDARD LVCMOS33 [get_ports {button}]
set_property PACKAGE_PIN N17 [get_ports {button}]
```