



# FPGA programming

Kristian Harder, RAL

## 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)

- ★ 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 – 15:20 coffee break

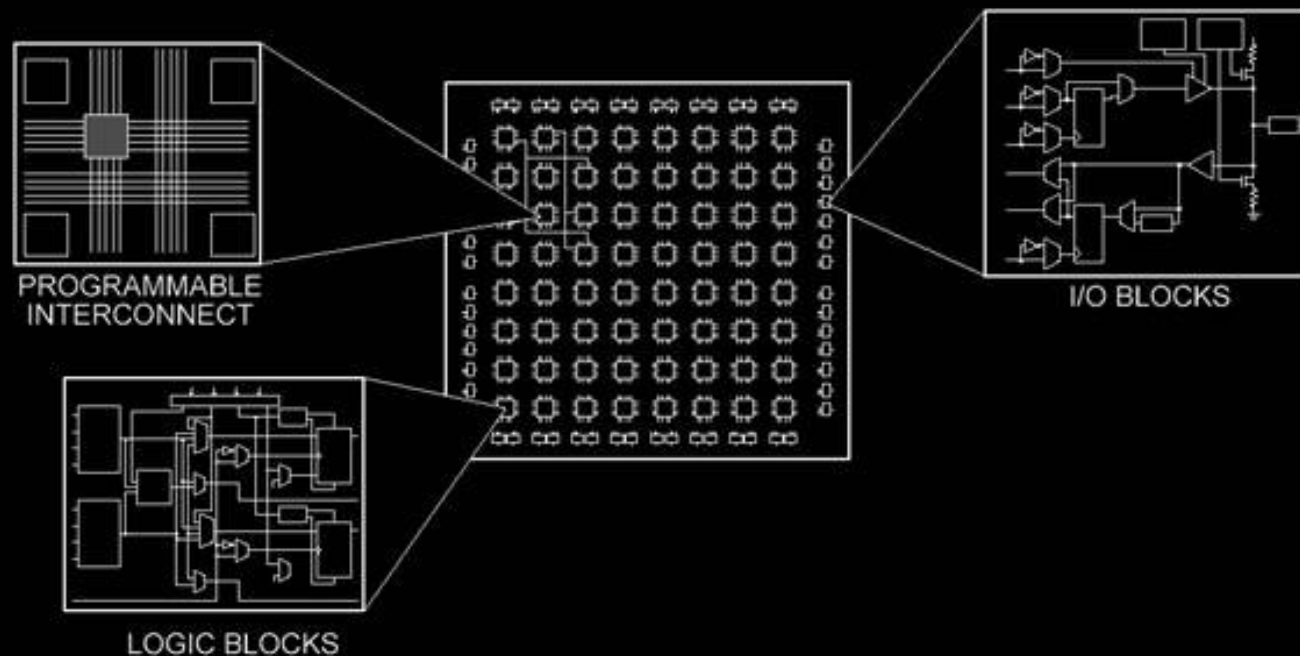
15:15 – 17:30 work in lab / discussion / Q&A

(might finish earlier)

## Field Programmable Gate Array

an integrated circuit consisting of

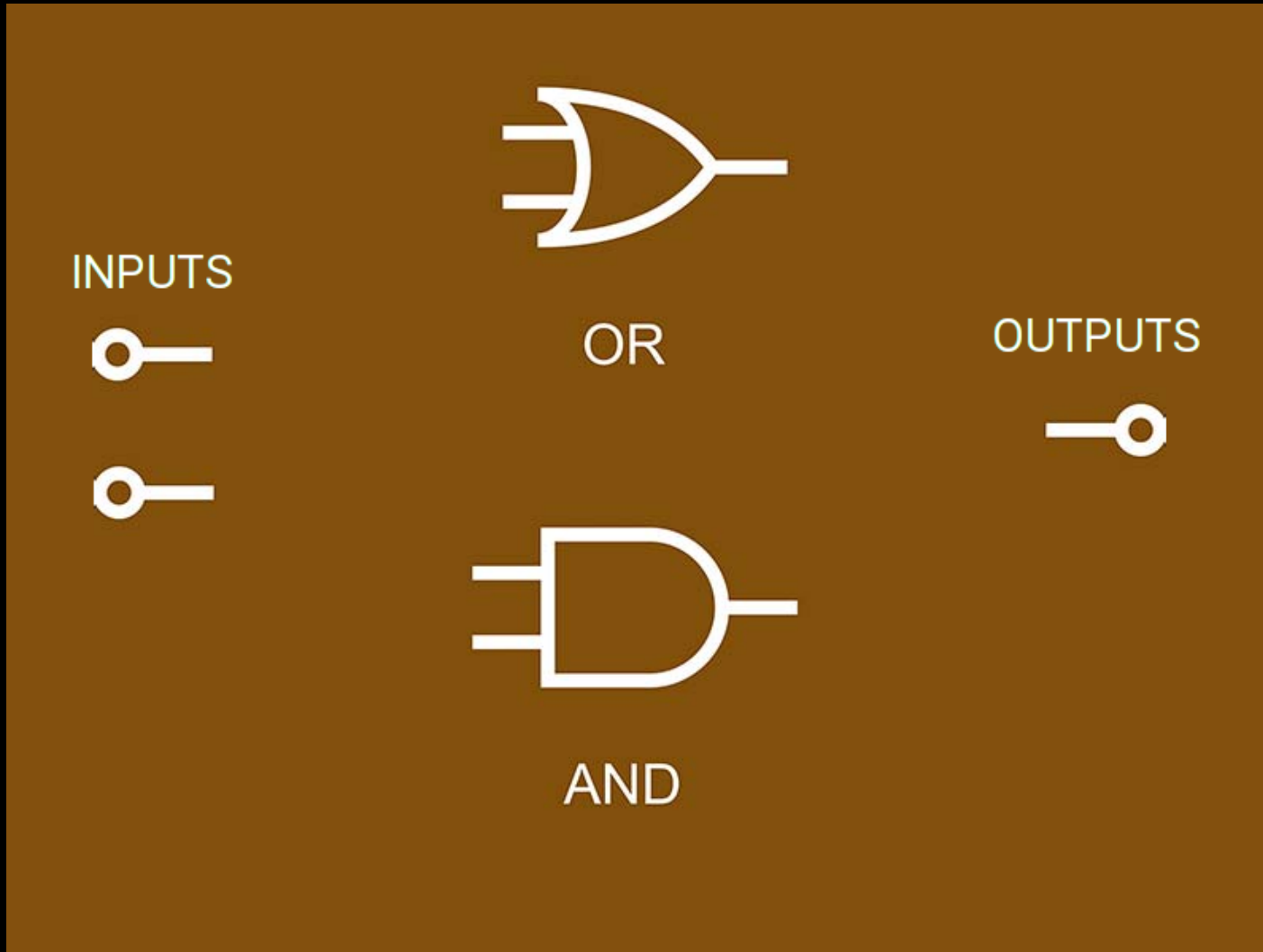
- ★ a large number of blocks with logic gates
- ★ connected by a programmable interconnection fabric
- ★ accessible through I/O blocks



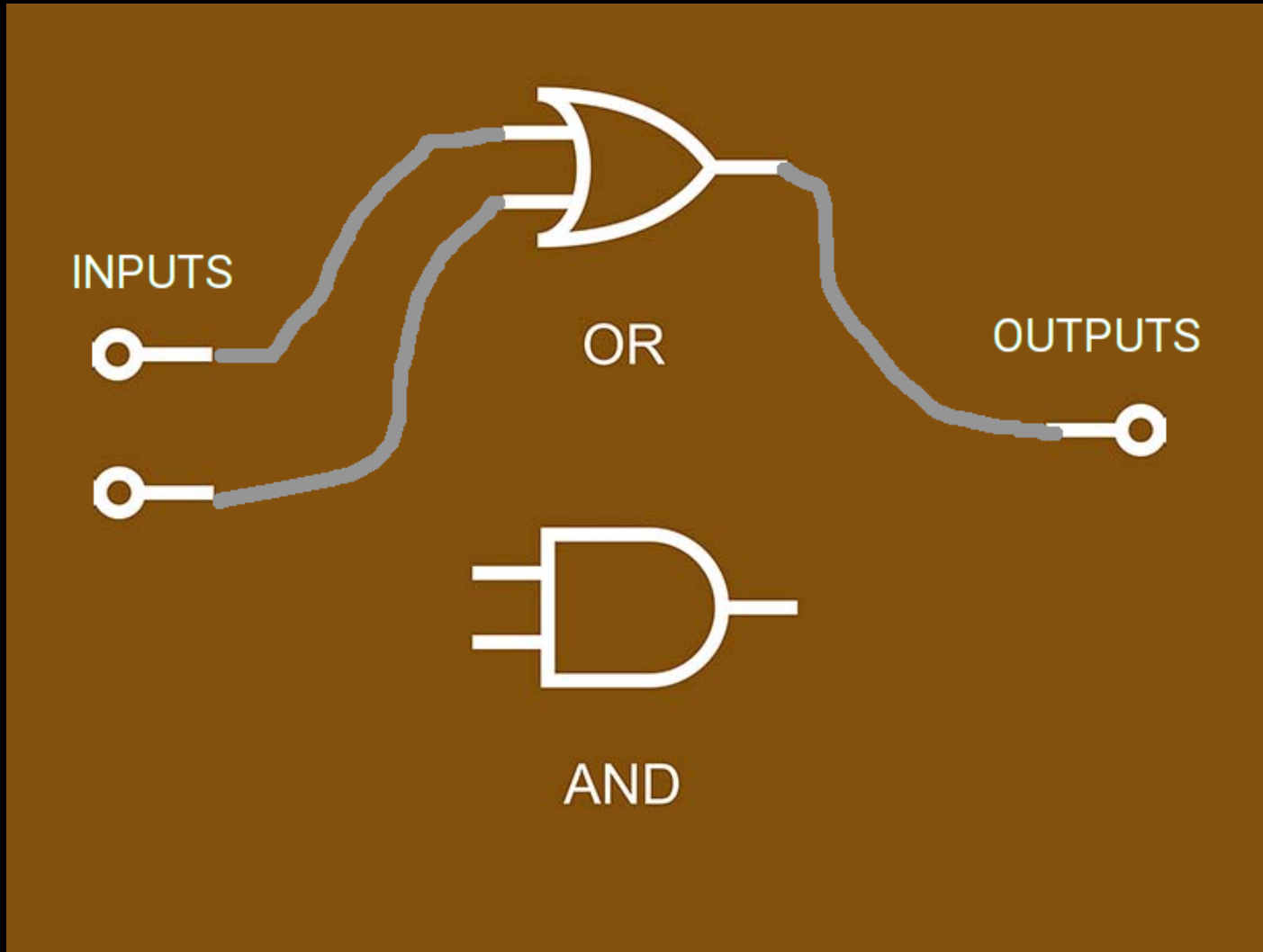
**Program your own electronic circuit onto this chip, anywhere, anytime!  
(within available resources)**



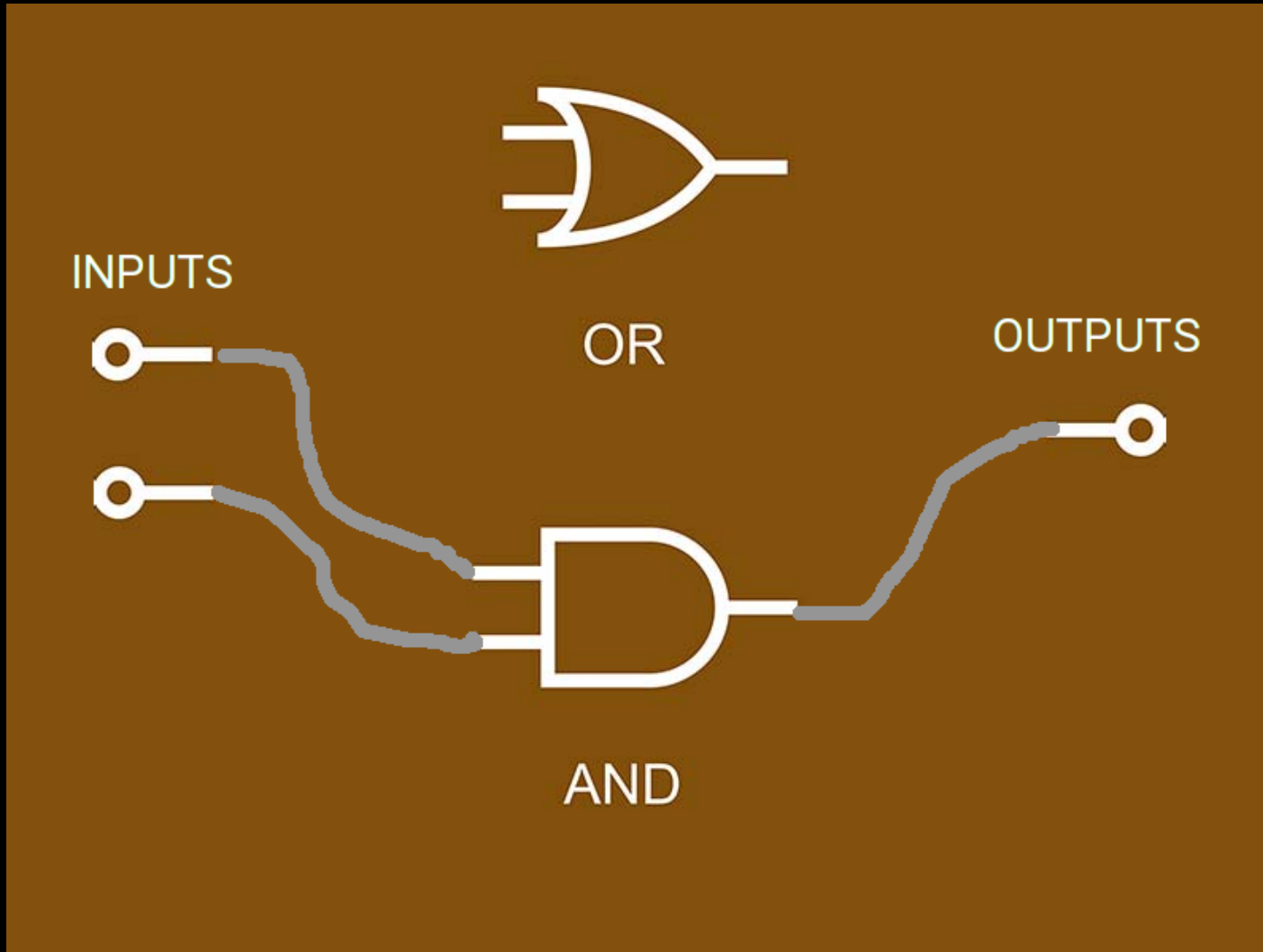
# FPGA: trivial example



# FPGA: trivial example

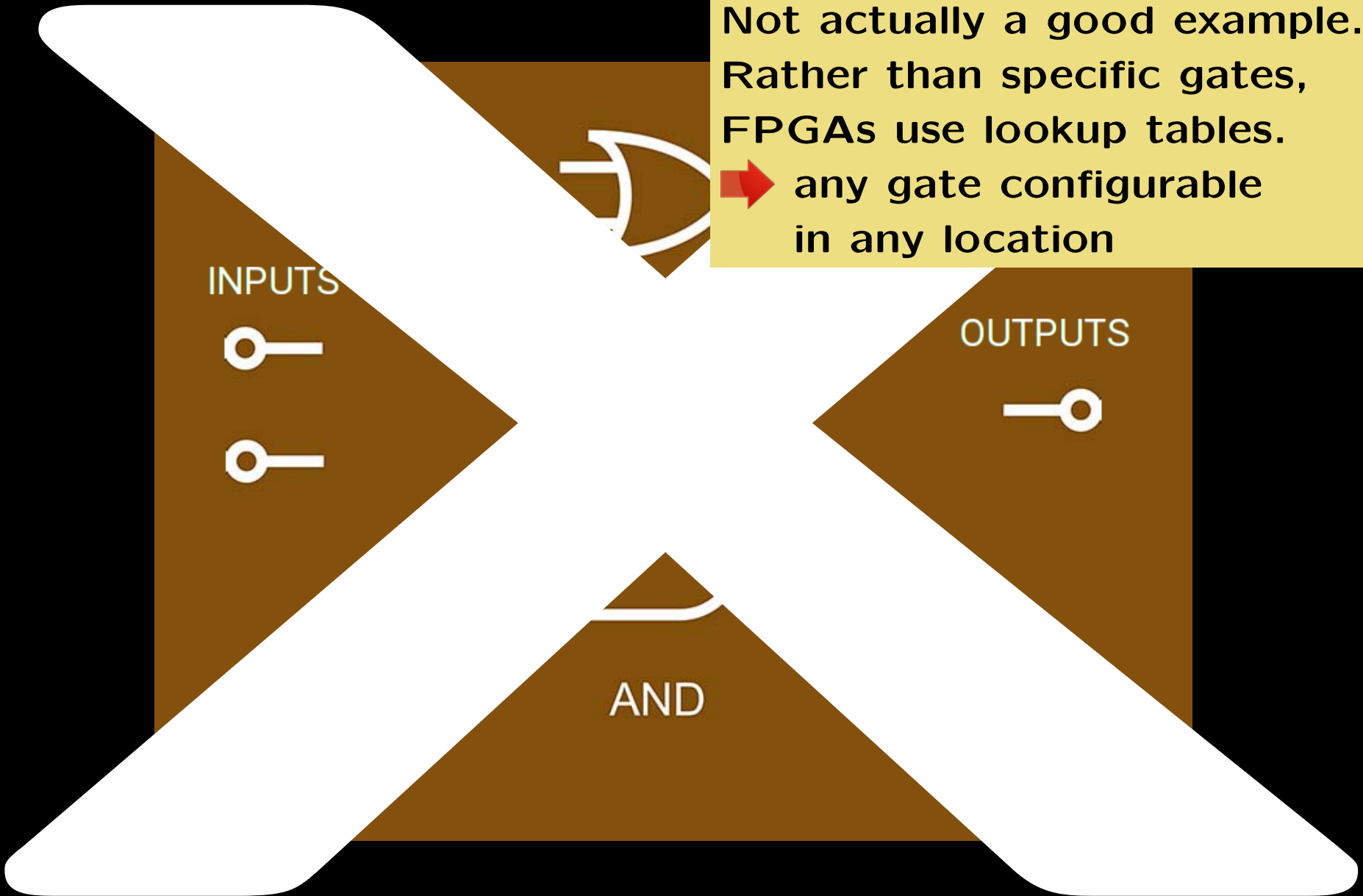


# FPGA: trivial example



# FPGA: trivial example

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



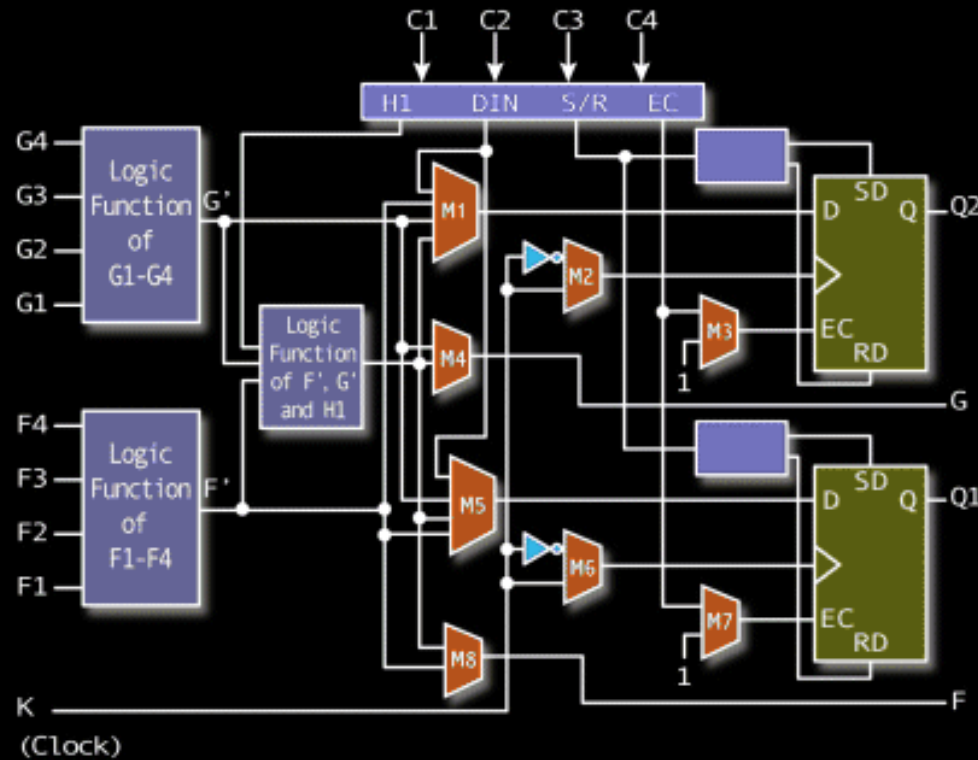


**A look-up table is a small memory bank that encodes a general logic function:**

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.**

## 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

## Multiple logic blocks working together

What happens if block A output changes just as block B is reading it?

➡ need synchronous operation – everything running on same clock

## Any individual task needs to

- ★ either finish well within one clock cycle
- ★ or be broken down into subtasks with registers in between

## How much can be done within one clock cycle

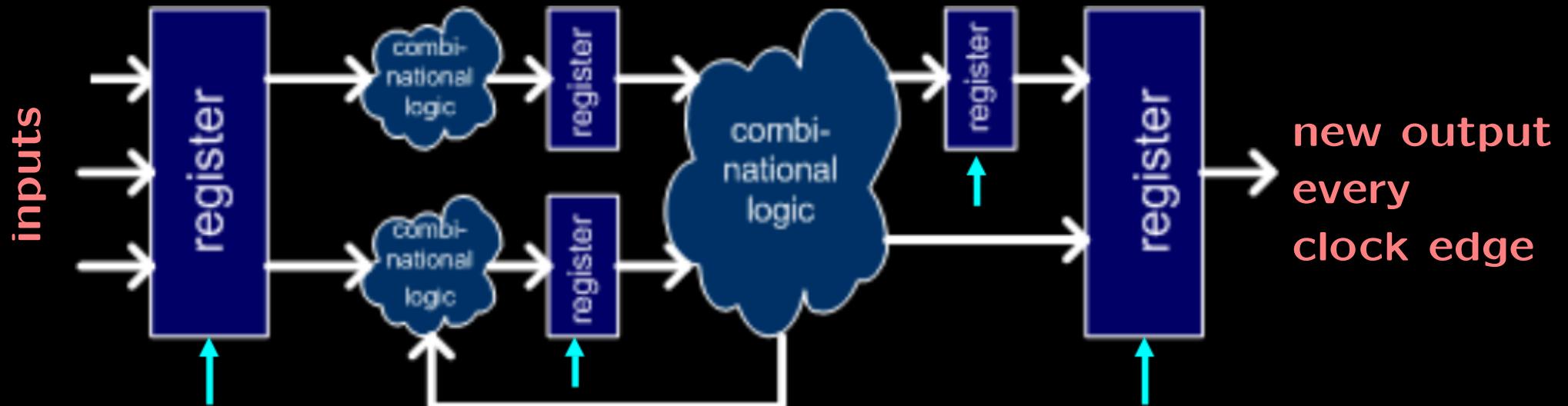
depends on the clock frequency

Faster clocks means things get done more quickly,  
but you might have to break them down into smaller bits.

➡ faster clock is not always better

# synchronous sequential logic

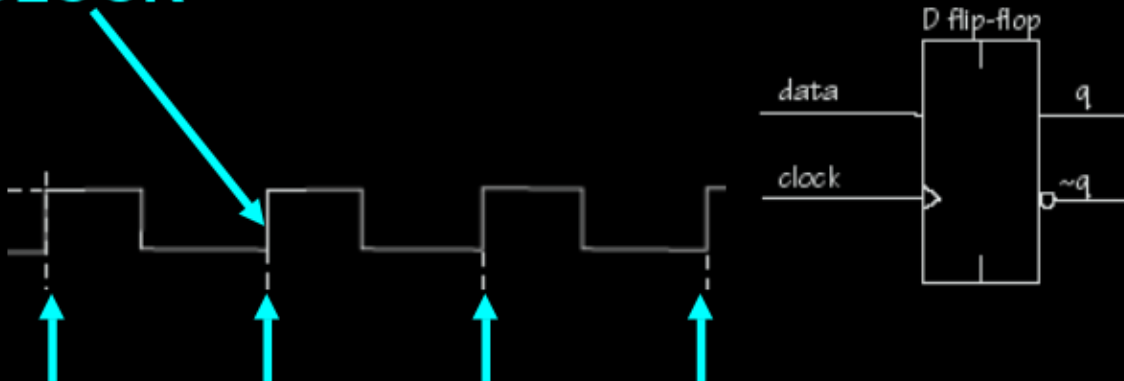
## intermediate logic



## Register

## CLOCK

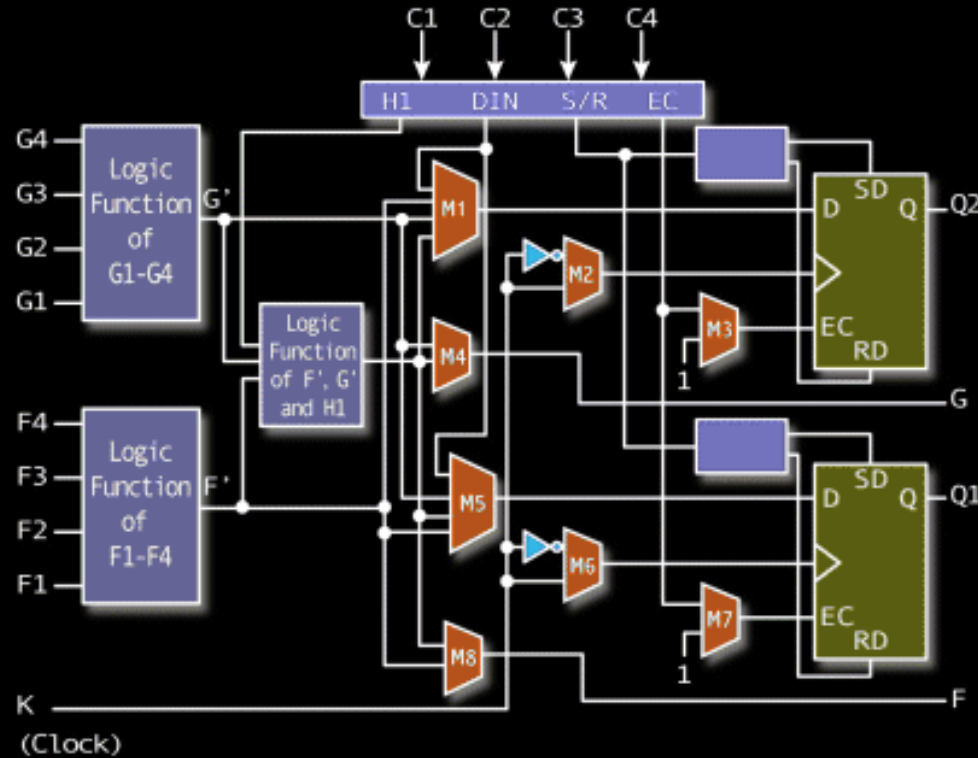
## EDGES



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

graph by Edward Freeman (STFC)

## 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)

## 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 <sup>(1)</sup>	–	–	–	–	–	–	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**



**CPUs and FPGAs are capable of performing arbitrary tasks depending on programming.**

**But the approach is fundamentally different:**

## **CPU**

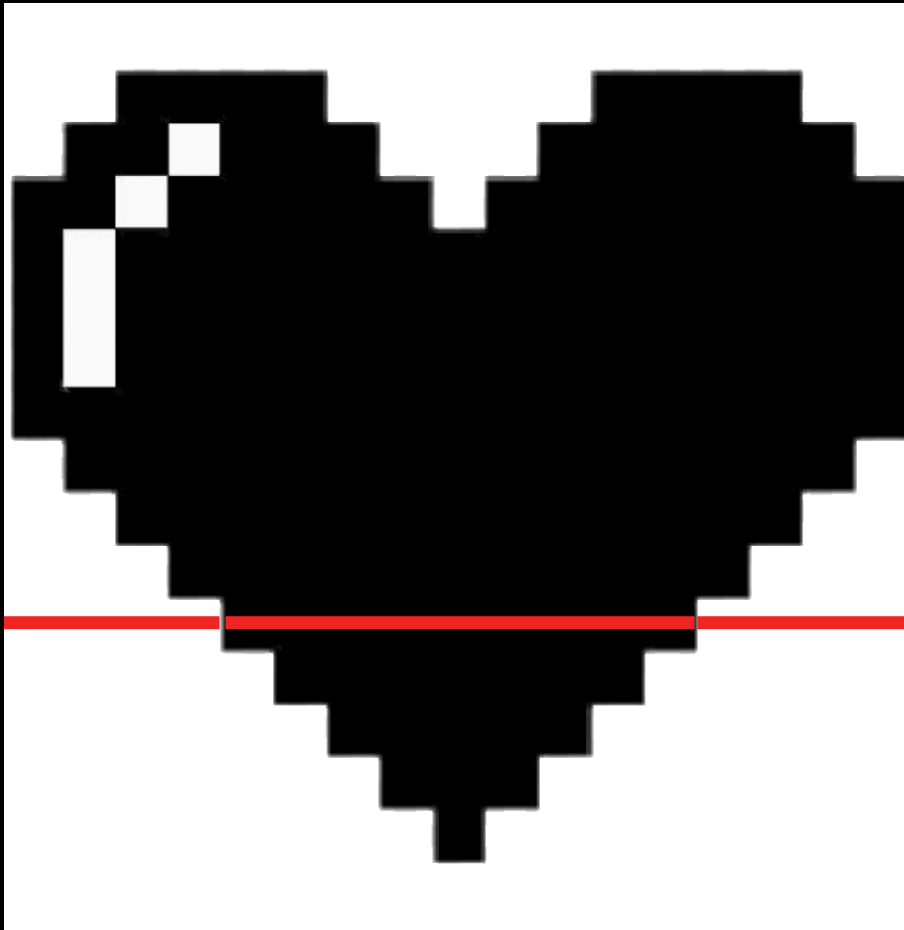
**rigid silicon — the processing units remain fixed  
(except basics like enabling/disabling cores in multicore processors)  
Flexibility from stepping through instructions provided in memory**

## **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**

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

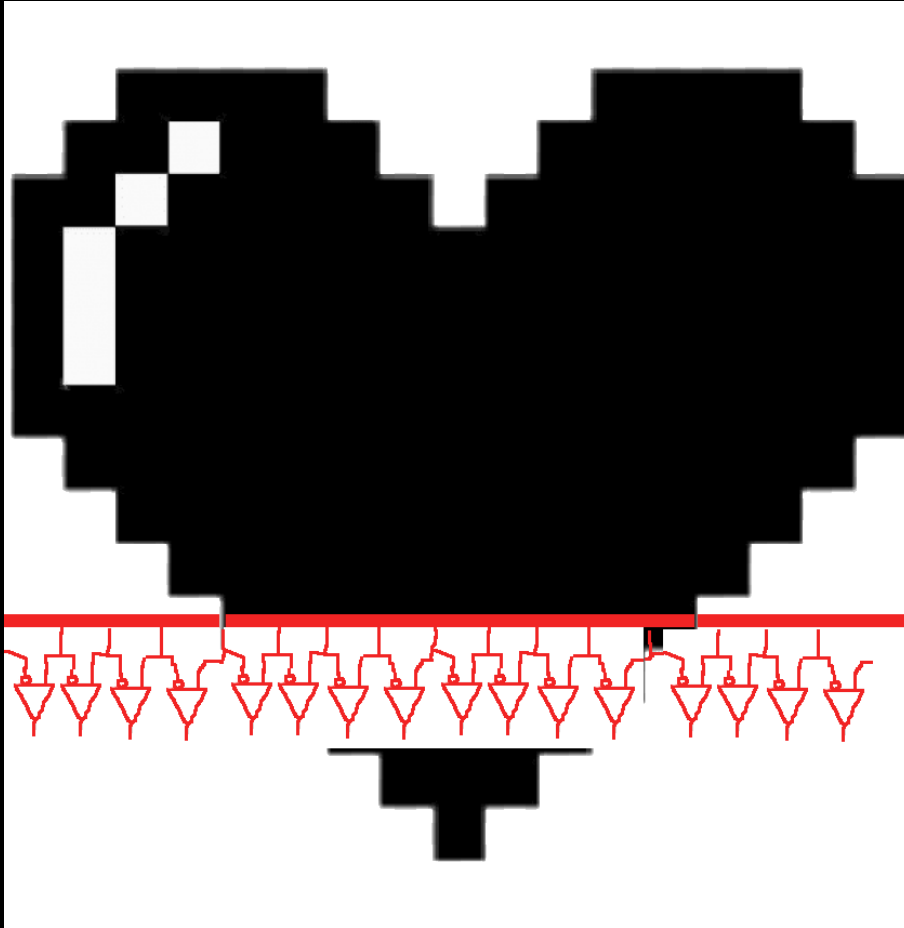


## CPU

scan a line like

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

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



## CPU

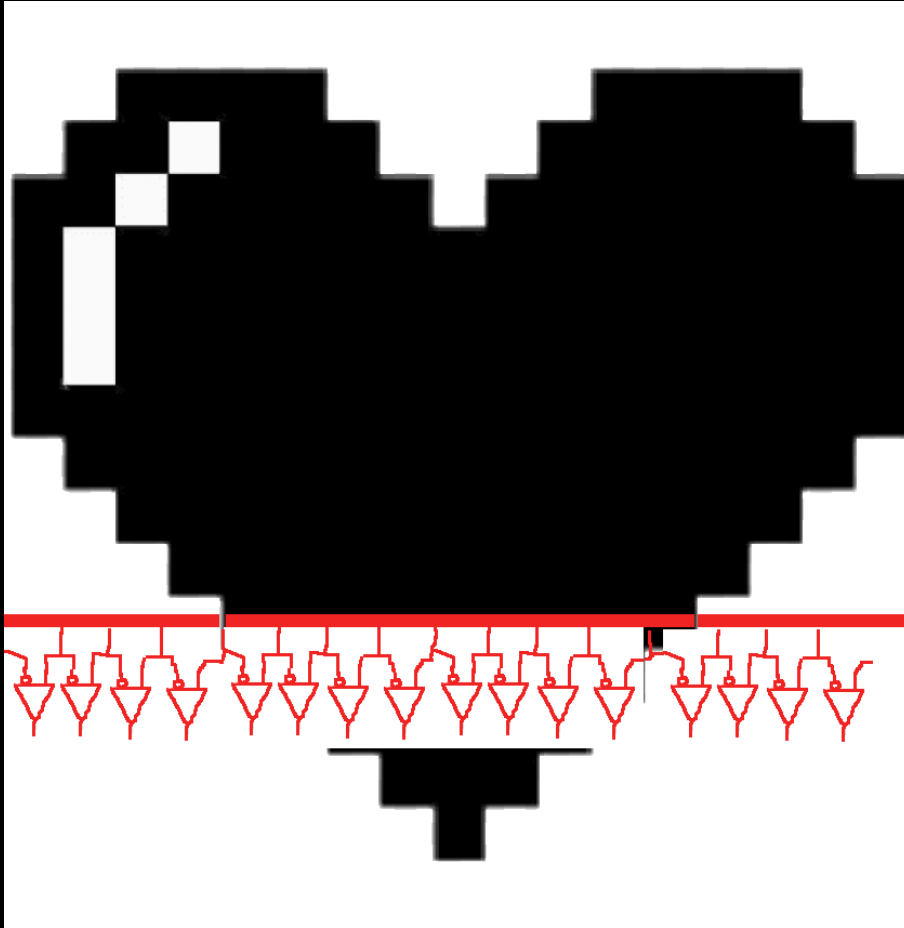
scan a line like

```
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

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

**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)

## 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



## 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

 |  **about 50% market share, full range incl high end**

 **formerly Altera, about 35% market share**

 **low power, low cost devices**

 **low power FPGA/ASIC hybrids**

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

...and probably others!

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

examples for commercially available FPGA boards:

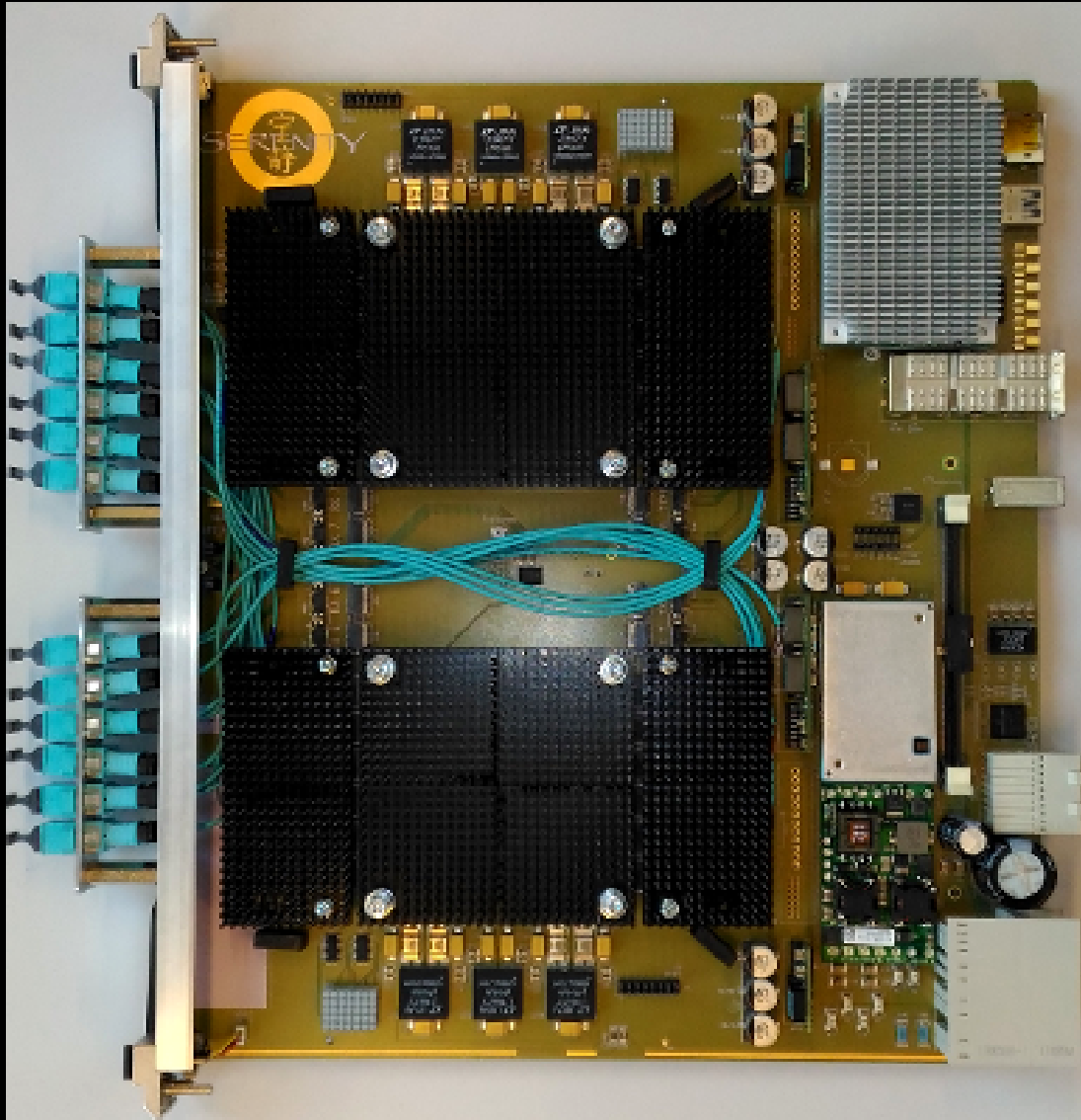


FPGA accelerator card based on Altera device



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

## 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

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

benefits:

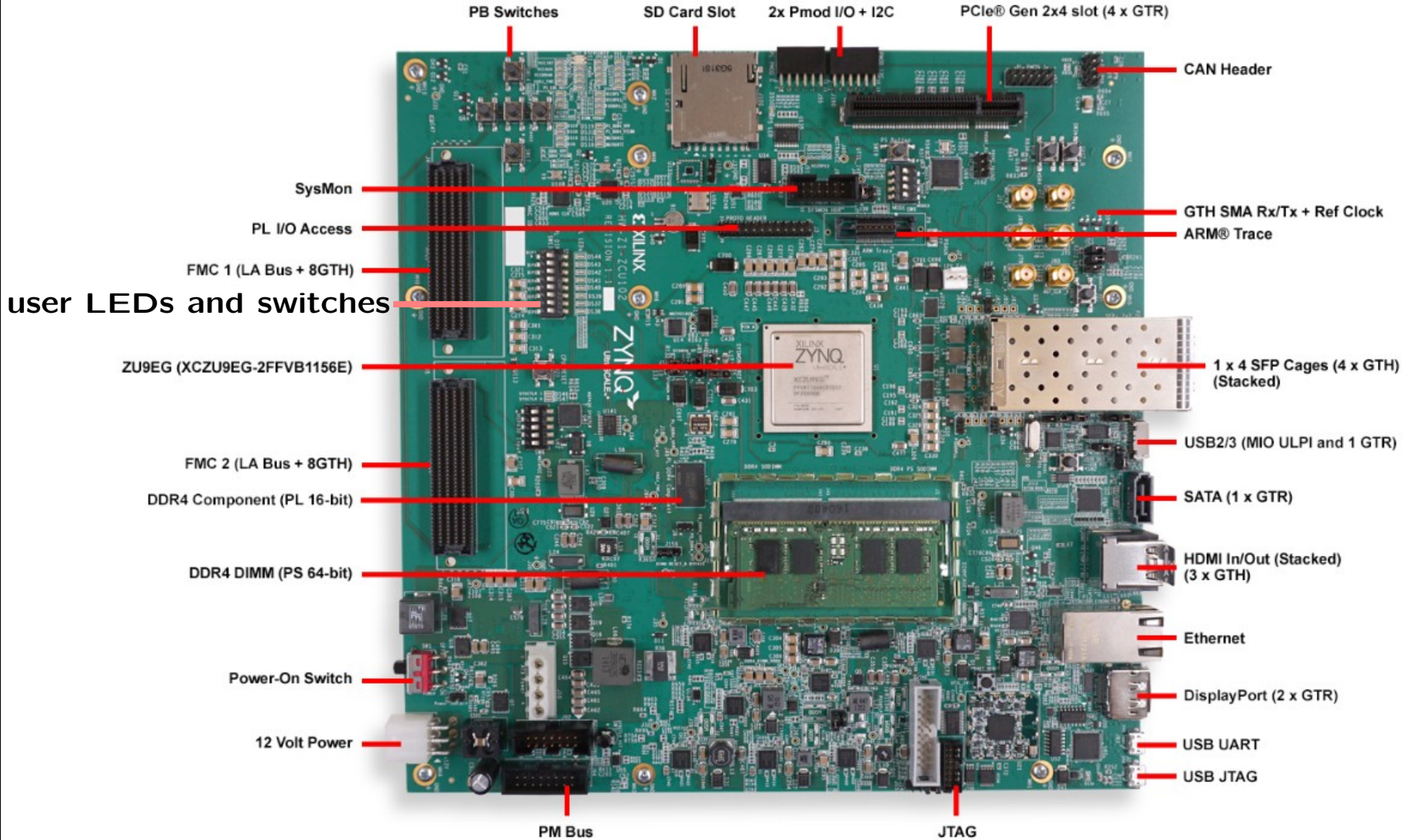
- ★ **often available very early on after release of new devices**
- ★ **often relatively low cost because subsidised**

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**

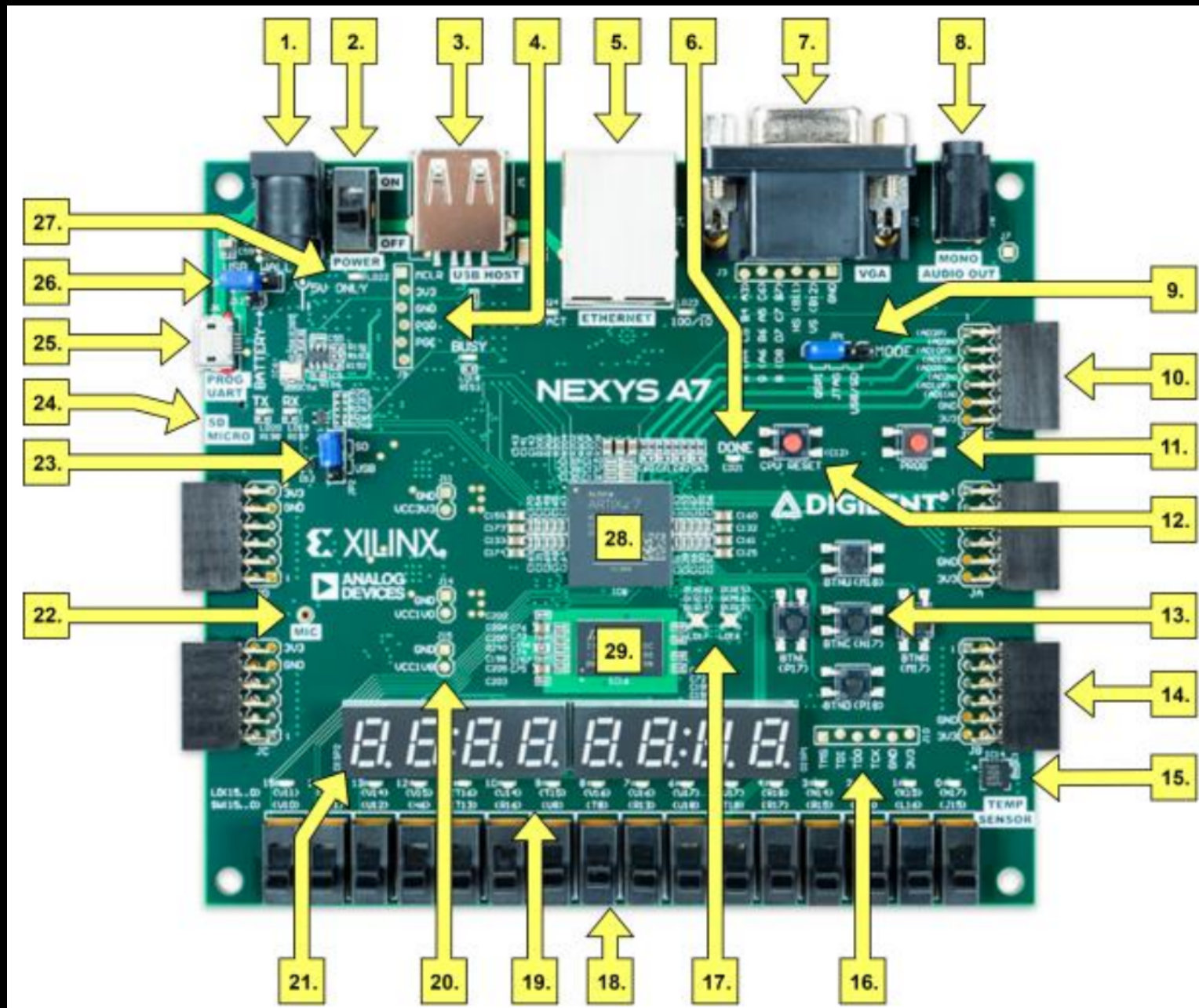


# Xilinx zcu102





# Digilent Nexys A7

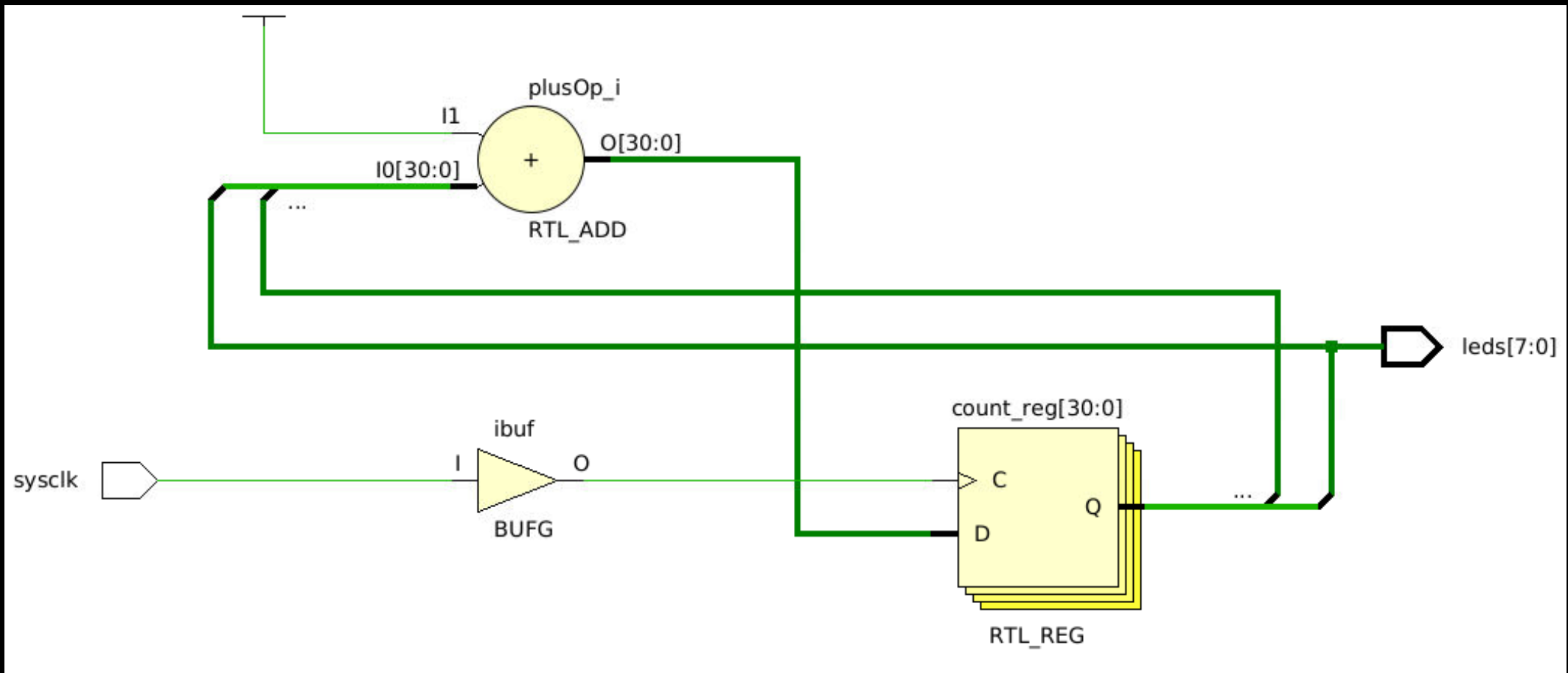


The set of configuration instructions for an FPGA  
★ is not actually modifying hardware,  
★ but it is not a software algorithm either.  
It is somewhat in between, which is why it is called



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

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



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.

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:

```

2 process ((S0,S1),A,B,C,D)
3 begin
4     case {S0,S1}, is
5         when "00" => Y <= A;
6         when "01" => Y <= B;
7         when "10" => Y <= C;
8         when "11" => Y <= D;
9         when others => Y <= A;
10    end case;
11 end process;

```

Verilog:

```

1
2
3 always @((S0,S1), A, B, C, D)
4     case ((S0,S1))
5         2'b00: Y = A;
6         2'b01: Y = B;
7         2'b10: Y = C;
8         2'b11: Y = D;
9     endcase
10

```

from [blog.digilentinc.com](http://blog.digilentinc.com)



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.)

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



```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk   : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk   : std_logic;
21     signal count : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25         ibuf: BUFG
26             port map(
27                 i => sysclk,
28                 o => clk
29             );
30
31
32
33         process(clk)
34             begin
35                 if rising_edge(clk) then
36                     count <= count + 1;
37                 end if;
38             end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

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

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds      : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

← This is a comment

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk   : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk   : std_logic;
21     signal count : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

← 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

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk   : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk   : std_logic;
21     signal count : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

← 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

```
1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45
```

← Describe the VHDL block

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds      : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

} ← Declare internal signals we need



consider signals more like wires,  
not as variables



can assign an initial state, though



```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds      : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

← Instantiate a different block

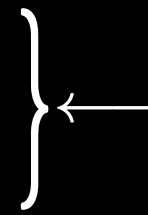
- ★ this one is from a library
- ★ it buffers an incoming clock for distribution
- ★ we name this instance ibuf
- ★ we connect it to our signals

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

## a process



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

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk   : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk      : std_logic;
21     signal count    : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34         begin
35             if rising_edge(clk) then
36                 count <= count + 1;
37             end if;
38         end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

## 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



```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk   : in  STD_LOGIC;
13     leds     : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk   : std_logic;
21     signal count : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34     begin
35         if rising_edge(clk) then
36             count <= count + 1;
37         end if;
38     end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

**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 constraints** (separate file)

```

1  # Nexys A7 constraints file
2  set_property CFGBVS VCC0 [current_design]
3  set_property CONFIG_VOLTAGE 3.3 [current_design]
4
5  # System clock (100MHz)
6  set_property IOSTANDARD LVCMOS33 [get_ports {sysclk}]
7  set_property PACKAGE_PIN E3 [get_ports sysclk]
8  create_clock -period 10 -name sysclk [get_ports sysclk]
9
10 # LEDs
11 set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
12 set_property PACKAGE_PIN H17 [get_ports {leds[0]}]
13 set_property PACKAGE_PIN K15 [get_ports {leds[1]}]
14 set_property PACKAGE_PIN J13 [get_ports {leds[2]}]
15 set_property PACKAGE_PIN N14 [get_ports {leds[3]}]
16 set_property PACKAGE_PIN R18 [get_ports {leds[4]}]
17 set_property PACKAGE_PIN V17 [get_ports {leds[5]}]
18 set_property PACKAGE_PIN U17 [get_ports {leds[6]}]
19 set_property PACKAGE_PIN U16 [get_ports {leds[7]}]

```

(this information from Nexys A7 documentation)

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

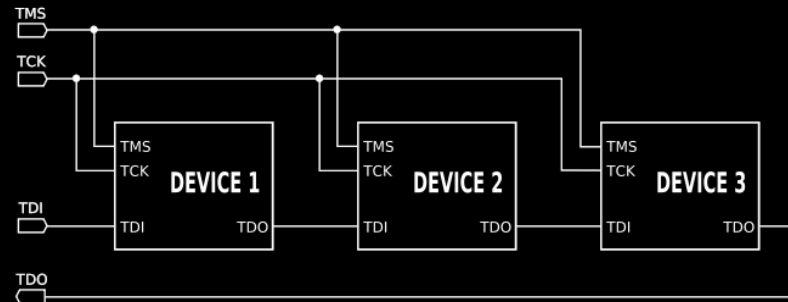
connect to device via serial JTAG interface and configure it!  
most high-end FPGAs use volatile RAM to store configuration  
→ need to reconfigure with JTAG after each power-up  
or store firmware in external flash ROM

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



HEPLNX - TigerVNC

example - [/data/pff62257/fpga\_lecture/example/example/example.xpr] - Vivado 2019.1

example - [/data/pff62257/fpga\_lecture/example/example/example.xpr] - Vivado 2019.1

File Edit Flow Tools Reports Window Layout View Help

write\_bitstream Complete

Flow Navigator PROJECT MANAGER - example

PROJECT MANAGER

- Settings
- Add Sources
- Language Templates
- IP Catalog
- IP INTEGRATOR
  - Create Block Design
  - Open Block Design
  - Generate Block Design
- SIMULATION
  - Run Simulation
- RTL ANALYSIS
  - Open Elaborated Design
- SYNTHESIS
  - Run Synthesis
  - Open Synthesized Design
- IMPLEMENTATION
  - Run Implementation
  - Open Implemented Design
- PROGRAM AND DEBUG
  - Generate Bitstream
  - Open Hardware Manager

Sources

- Design Sources (1)
  - top (rtl) (top.vhd)
- Constraints (1)
  - constrs\_1 (1)
    - constraints.xdc (target)
- Simulation Sources (1)
- Utility Sources

Hierarchy Libraries Compile Order

Source File Properties

constraints.xdc

- Enabled
- Location: /data/pff62257/fpga\_lecture/example/ex
- Type: XDC
- Size: 0.7 KB

General Properties

Project Summary x top.vhd x

```

/data/pff62257/fpga_lecture/example/example/srcs/sources_1/imports/example/top.vhd
19 :
20 :     signal clk      : std_logic;
21 :     signal count : unsigned(27 downto 0) := (others => '0');
22 :
23 : begin
24 :
25 :     ibuf: IBUFDS
26 :     port map(
27 :         i => sysclk_p,
28 :         ib => sysclk_n,
29 :         o => clk
30 :     );
31 :
32 :
33 :     process(clk)
34 :     begin
35 :         if rising_edge(clk) then
36 :             count <= count + 1;
37 :         end if;
38 :     end process;
39 :
40 :
41 :     leds(7 downto 0) <= std_logic_vector(count(27 downto 20));
42 :
43 :
44 : end rtl;
45 :

```

Tcl Console Messages Log Reports Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	synth_design Complete!								1	28	0.0	0	0	5/11/20, 2:26 PM	00:02:20	Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	write_bitstream Complete!	7.227	0.000	0.055	0.000	0.000	0.682	0	1	28	0.0	0	0	5/11/20, 2:29 PM	00:05:08	Vivado Implementation Defaults (Vivado Implementa

JavaEmbeddedFrame pff62257@heplnw062: ~ example - [/data/pff62257/fpga\_lec... pff62257@heplnw062: /data/pff62...

## 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	Implementation
Status: <span style="color: green;">✔</span> Complete Messages: <span style="color: orange;">⚠</span> 1 warning Part: xc7a100tcsg324-1 Strategy: <a href="#">Vivado Synthesis Defaults</a> Report Strategy: <a href="#">Vivado Synthesis Default Reports</a> Incremental synthesis: <a href="#">Automatically selected checkpoint</a>	Status: <span style="color: green;">✔</span> Complete Messages: <span style="color: orange;">⚠</span> 2 warnings Part: xc7a100tcsg324-1 Strategy: <a href="#">Vivado Implementation Defaults</a> Report Strategy: <a href="#">Vivado Implementation Default Reports</a> Incremental implementation: <a href="#">None</a>

DRC Violations	Timing															
No DRC violations were found. <a href="#">Implemented DRC Report</a>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #f0f0f0;"> <th>Setup</th> <th>Hold</th> <th>Pulse Width</th> </tr> </thead> <tbody> <tr> <td>Worst Negative Slack (WNS):</td> <td>7.1 ns</td> <td></td> </tr> <tr> <td>Total Negative Slack (TNS):</td> <td>0 ns</td> <td></td> </tr> <tr> <td>Number of Failing Endpoints:</td> <td>0</td> <td></td> </tr> <tr> <td>Total Number of Endpoints:</td> <td>31</td> <td></td> </tr> </tbody> </table> <a href="#">Implemented Timing Report</a>	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	
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															

Utilization	Power																				
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #f0f0f0;"> <th>Post-Synthesis</th> <th>Post-Implementation</th> </tr> </thead> <tbody> <tr> <td colspan="2" style="text-align: center;"><b>Graph</b>   <a href="#">Table</a></td> </tr> <tr> <td colspan="2"> <p style="text-align: center;">Utilization (%)</p> </td> </tr> </tbody> </table>	Post-Synthesis	Post-Implementation	<b>Graph</b>   <a href="#">Table</a>		<p style="text-align: center;">Utilization (%)</p>		<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #f0f0f0;"> <th>Summary</th> <th>On-Chip</th> </tr> </thead> <tbody> <tr> <td><b>Total On-Chip Power:</b></td> <td><b>0.104 W</b></td> </tr> <tr> <td><b>Junction Temperature:</b></td> <td><b>25.5 °C</b></td> </tr> <tr> <td>Thermal Margin:</td> <td>59.5 °C (12.9 W)</td> </tr> <tr> <td>Effective <math>\theta_{JA}</math>:</td> <td>4.6 °C/W</td> </tr> <tr> <td>Power supplied to off-chip devices:</td> <td>0 W</td> </tr> <tr> <td>Confidence level:</td> <td><a href="#">Medium</a></td> </tr> </tbody> </table> <a href="#">Implemented Power Report</a>	Summary	On-Chip	<b>Total On-Chip Power:</b>	<b>0.104 W</b>	<b>Junction Temperature:</b>	<b>25.5 °C</b>	Thermal Margin:	59.5 °C (12.9 W)	Effective $\theta_{JA}$ :	4.6 °C/W	Power supplied to off-chip devices:	0 W	Confidence level:	<a href="#">Medium</a>
Post-Synthesis	Post-Implementation																				
<b>Graph</b>   <a href="#">Table</a>																					
<p style="text-align: center;">Utilization (%)</p>																					
Summary	On-Chip																				
<b>Total On-Chip Power:</b>	<b>0.104 W</b>																				
<b>Junction Temperature:</b>	<b>25.5 °C</b>																				
Thermal Margin:	59.5 °C (12.9 W)																				
Effective $\theta_{JA}$ :	4.6 °C/W																				
Power supplied to off-chip devices:	0 W																				
Confidence level:	<a href="#">Medium</a>																				

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)

The screenshot displays the ModelSim Starter interface during a simulation. The main window is titled "sim - Default" and shows a hierarchical tree of objects on the left, including a process named "#INITIAL #69". The central pane shows a list of objects with their names and values, such as "ck" (1h1) and "rdy" (1h0). The right pane shows a waveform viewer with a table of signals and their values over time. The signals include "ck", "prw", "pstrb", "prdy", "paddr", "pdata", "srw", "srdy", "saddr", and "sdata". The waveform shows a sequence of events, with a cursor positioned at 2.82 us. The transcript at the bottom shows a break in the module "proc" at line 94, with the following text:

```

** Note: $stop : proc.v(94)
Time: 2820 ns Iteration: 1 Instance: /top/p
Break in Module proc at proc.v line 94
VSIM 25>

```

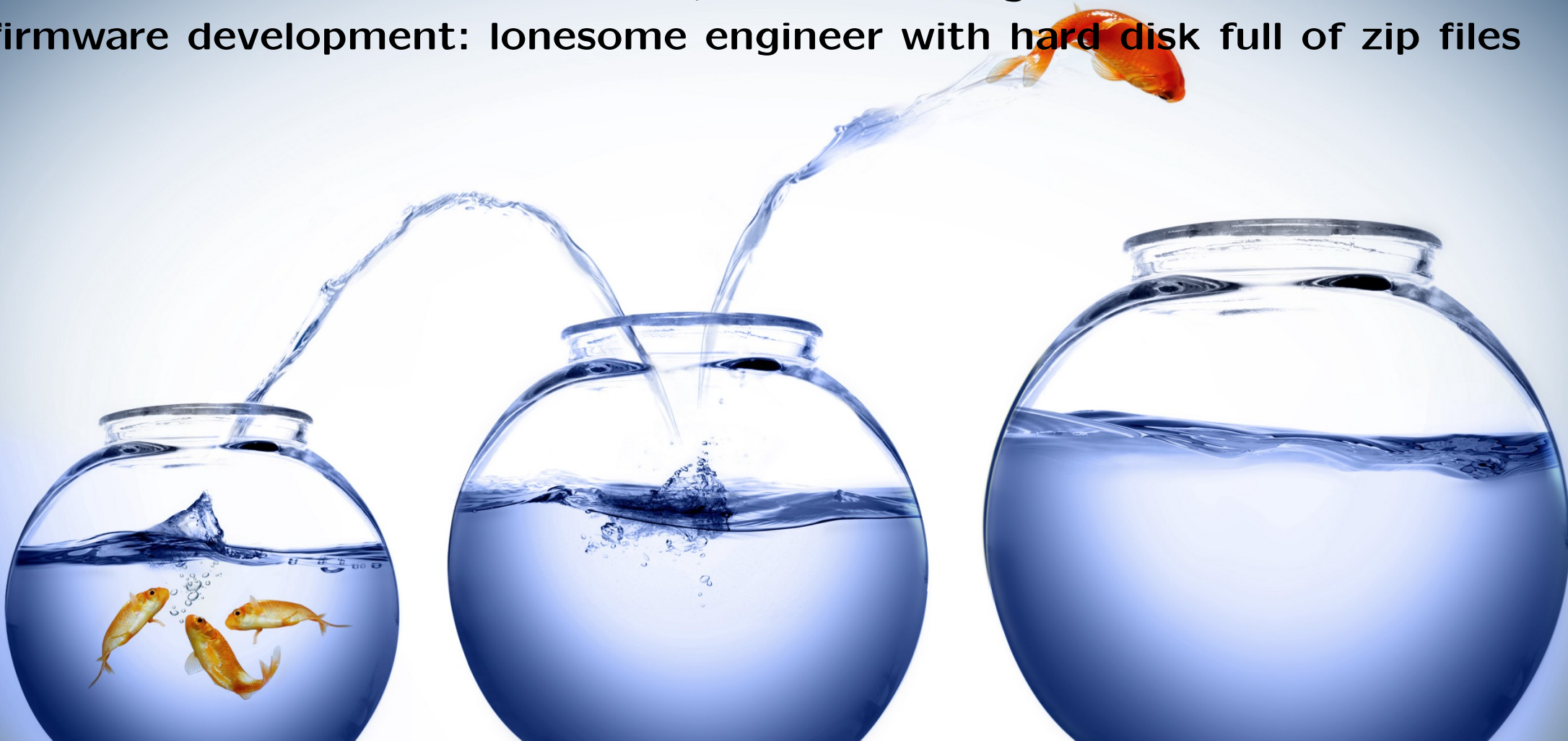
The status bar at the bottom indicates the current time is 1870 ns to 2526 ns, with a cursor at 2,820 ns and a delta of 1 ns. The current simulation point is "sim:/top/p/#INITIAL #69".



# 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



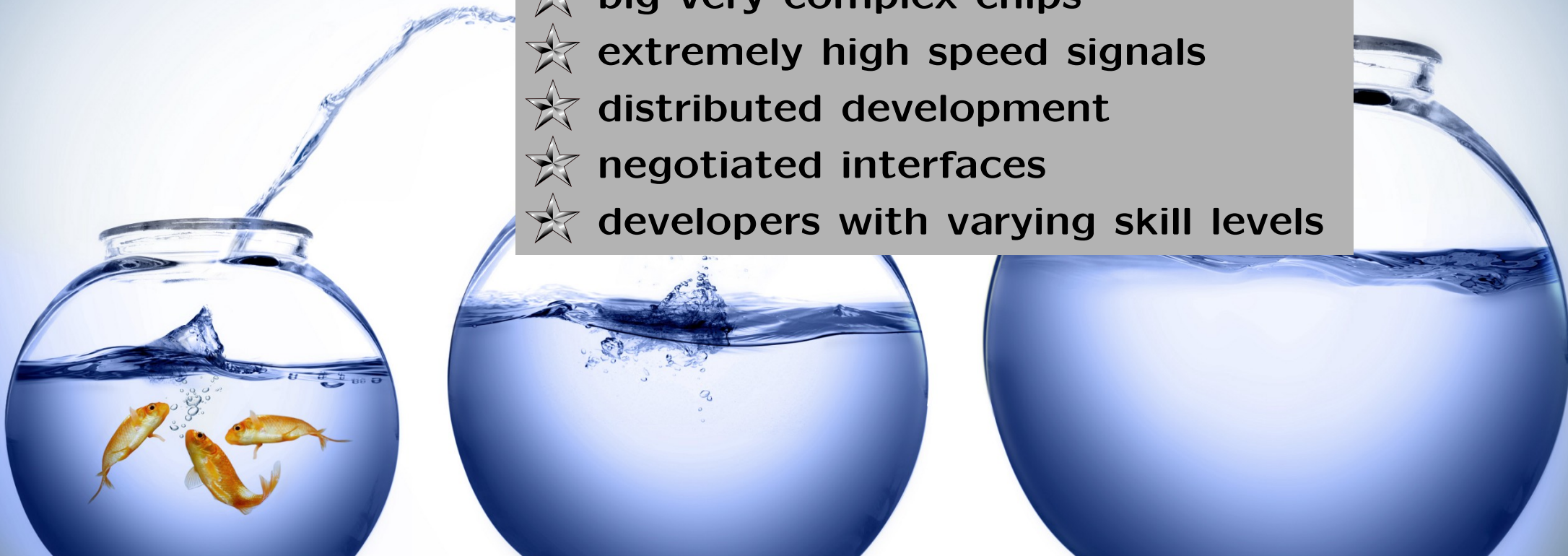
## last topic: how to approach a BIG firmware project

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





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**

**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.**

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 of 2–3 people

**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

```

1  --- simple "hello world" example
2
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6
7  Library UNISIM;
8  use UNISIM.vcomponents.all;
9
10
11 entity top is port(
12     sysclk    : in  STD_LOGIC;
13     leds      : out STD_LOGIC_VECTOR (7 downto 0)
14 );
15 end top;
16
17
18 architecture rtl of top is
19
20     signal clk    : std_logic;
21     signal count  : unsigned(30 downto 0) := (others => '0');
22
23     begin
24
25     ibuf: BUFG
26         port map(
27             i => sysclk,
28             o => clk
29         );
30
31
32
33     process(clk)
34     begin
35         if rising_edge(clk) then
36             count <= count + 1;
37         end if;
38     end process;
39
40
41     leds <= std_logic_vector(count(30 downto 23));
42
43
44 end rtl;
45

```

```

1  # Nexys A7 constraints file
2  set_property CFGBVS VCC0 [current_design]
3  set_property CONFIG_VOLTAGE 3.3 [current_design]
4
5  # System clock (100MHz)
6  set_property IOSTANDARD LVCMOS33 [get_ports {sysclk}]
7  set_property PACKAGE_PIN E3 [get_ports sysclk]
8  create_clock -period 10 -name sysclk [get_ports sysclk]
9
10 # LEDs
11 set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
12 set_property PACKAGE_PIN H17 [get_ports {leds[0]}]
13 set_property PACKAGE_PIN K15 [get_ports {leds[1]}]
14 set_property PACKAGE_PIN J13 [get_ports {leds[2]}]
15 set_property PACKAGE_PIN N14 [get_ports {leds[3]}]
16 set_property PACKAGE_PIN R18 [get_ports {leds[4]}]
17 set_property PACKAGE_PIN V17 [get_ports {leds[5]}]
18 set_property PACKAGE_PIN U17 [get_ports {leds[6]}]
19 set_property PACKAGE_PIN U16 [get_ports {leds[7]}]
--

```

# Nexys A7 example with button

```
--- 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;
```

```
# Nexys A7 constraints file
```

```
set_property CFGBVS VCC0 [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}]
```