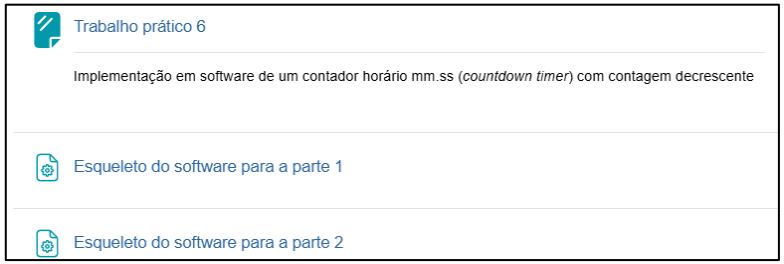
Designing a Custom AXI-lite Slave Peripheral

LECTURE 9

PEDRO SANTOS (SLIDES BY IOULIIA SKLIAROVA)

Re-cap.

- Lecture 7 and 8 (and Lab.6)
  - Hardware project: update MicroBlaze platform with:
    - Timers (Fixed Interval and AXI4)
    - Interrupts
  - Software project:
    - Polling: Dedicated file under Aula 7
    - Interrupts: Lab.6 files

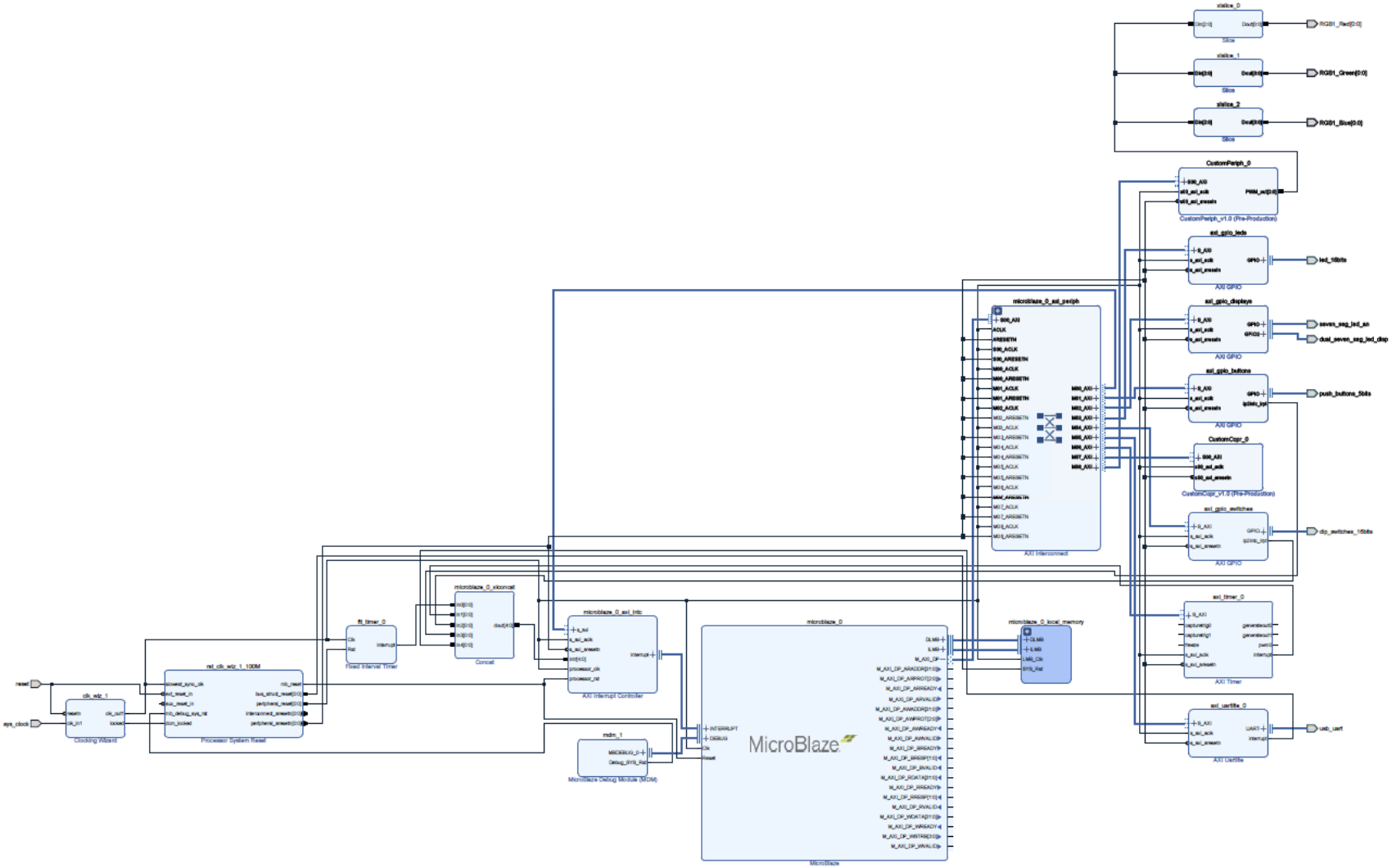


- Lab.6
  - Implementar um contador horário mm.ss (countdown timer) (…) semelhante ao do trabalho prático 4, mas em que todo o controlo e atualização são realizadas integralmente por software a executar sobre um SoC baseado no processador MicroBlaze.
  - Parte 1: utilizando Polling
  - Parte 2: utilizando Interrupções
  - **Lab.6 – Part 1 or 2 are expected for submission**. Submission form is **open** in E-learning

Custom Hardware

- Custom hardware blocks are used to:
  - Delegate to hardware time-critical functions
  - Save MicroBlaze resources
- One example will be considered, which includes:
  - A custom coprocessor with no output ports
  - A custom peripheral with output ports
- Next lab (lab. 7) will be dedicated to the design and use of a custom peripheral IP – Display Driver.

Block Design (BD)



AXI - Recap

- **Advanced eXtensible Interface** is a point to point interconnect that is designed for high performance, high speed microcontroller systems.
- The **AXI** protocol is based on a point-to-point interconnect to avoid bus sharing and therefore allow higher bandwidth and lower latency.
- There are three types of AXI4 interfaces:
  - **AXI4-Lite**—for simple, low-throughput memory-mapped communication, providing a register-like structure with reduced features and complexity (1 transfer per transaction)
  - **AXI4** —for high-performance memory-mapped requirements (up to 256 data transfers)
  - **AXI4-Stream**—for high-speed streaming data (unlimited amount of data)
- The AXI specifications describe an interface between a single AXI master and a single AXI slave.

Consult the **AXI Reference Guide** and the **AXI4 Protocol Specification** files in E-learning.

AXI4 and AXI4-Lite Channels

- There are five independent channels between an AXI master and slave.
- Address channels: used to send address and control information while performing a basic handshake between master and slave.
- Read/Write channels: A master reads data from and writes data to a slave. Read response information is placed on the read data channel, while write response information has a dedicated channel. This way the master can verify a write transaction has been completed.
- Transaction: Every exchange of data is called a transaction. A transaction includes the address and control information, the data sent, as well as any response information. The actual data is sent in bursts which contain multiple transfers.

AXI4-Lite

- Overview of AXI4-Lite:

> AXI4-Lite is similar to AXI4 with some exceptions: The most notable exception is that bursting is not supported. The AXI4-Lite chapter of the *ARM AMBA AXI Protocol Specification* [Ref 1] describes the AXI4-Lite protocol in more detail.

From *AXI Reference Guide*

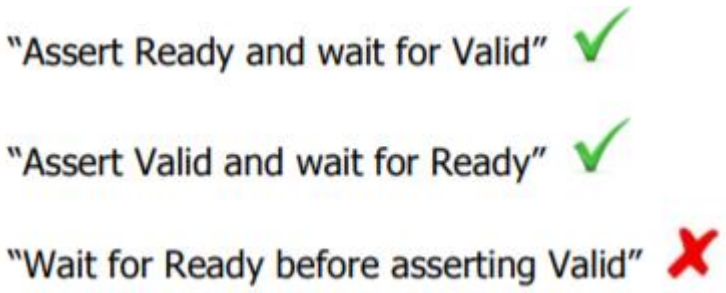
- Notable aspects:
  - All transactions are of burst length 1
  - All data accesses use the full width of the data bus
  - AXI4-Lite supports a data bus width of 32-bit or 64-bit
  - All accesses are non-modifiable, non-bufferable

- Signals sent over the various channels

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

AXI4-lite Handshaking Signals

- Consistent across the five channels.
- Based on a simple “Ready” and “Valid” principle:
  - “Ready” is used by the recipient to indicate that it is ready to accept a transfer of a data or address value.
  - “Valid” is used to clarify that the data (or address) provided on that channel by the sender is valid so that the recipient can then sample it.



HW and SW Project Example: 3 operand adder

Example 1 – Adder with 3 Operands

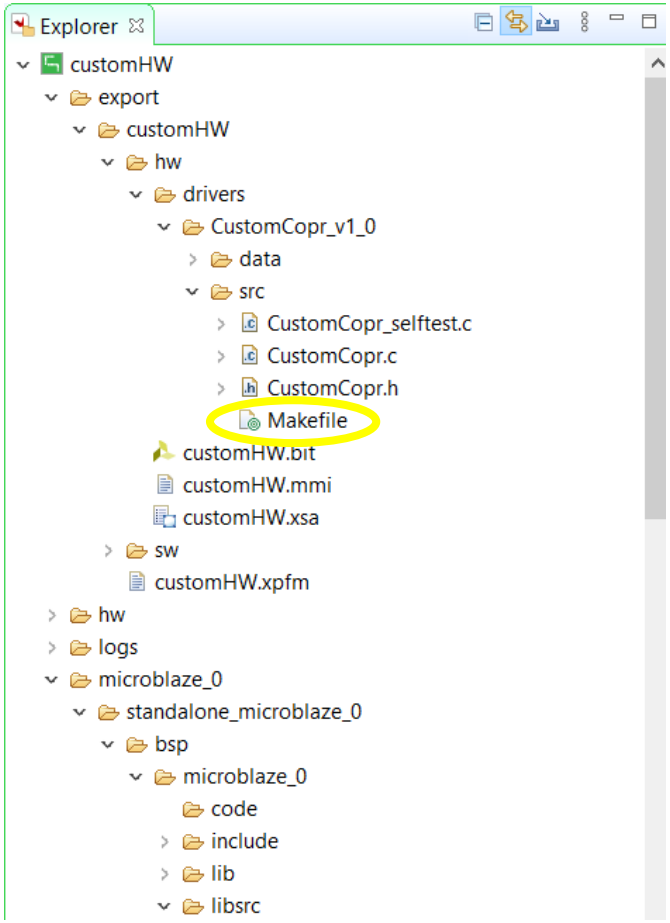
- Support files:
  - **Guião 3 – Atualizar MicroBlaze com Co-processador – Projecto HW**
  - **Guião 4 – Atualizar MicroBlaze com Co-processador – Projecto SW**
  - **Guião 5 – Atualizar MicroBlaze com Periférico – Projecto HW**

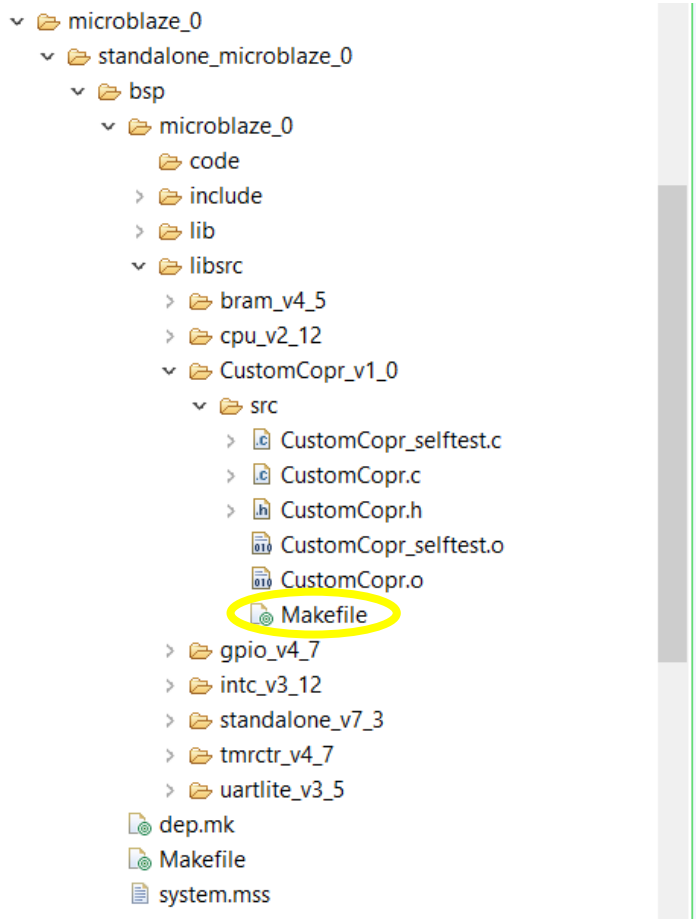
- Sequence:

1. Import Block Design
2. Create and Package new IP (*CustomCopr*)
3. Add IP
4. Edit in IP Packager
5. *CustomCopr*:
   - Adds the contents of 3 registers (written by software)
   - Puts the result in the 4th register (read by software)
6. Apply options from "Aula 6" (Problems and Results – slide 19)
7. Generate output products
8. Create HDL Wrapper
9. Generate Bitstream
10. Export Hardware
11. Launch Vitis

Vitis

If build errors do appear, update **all the makefiles** related to the custom IP:





Correct Makefile

```
COMPILER=
ARCHIVER=
CP=cp
COMPILER_FLAGS=
EXTRA_COMPILER_FLAGS=
LIB=libxil.a

RELEASEDIR=../../../lib
INCLUDEDIR=../../../include
INCLUDES=-I./. -I${INCLUDEDIR}

INCLUDEFILES=$(wildcard *.h)
LIBSOURCES=$(wildcard *.c)

OBJECTS = $(addsuffix .o, $(basename $(wildcard *.c)))
ASSEMBLY_OBJECTS = $(addsuffix .o, $(basename $(wildcard *.S)))

libs:
	echo "Compiling CustomCopr..."
	$(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) $(LIBSOURCES)
	$(ARCHIVER) -r ${RELEASEDIR}/${LIB} ${OBJECTS} ${ASSEMBLY_OBJECTS}
	make clean

include:
	${CP} $(INCLUDEFILES) $(INCLUDEDIR)

clean:
	rm -rf ${OBJECTS} ${ASSEMBLY_OBJECTS}
```

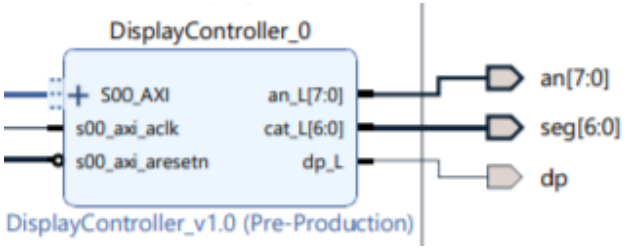
After Correcting the Makefiles

1. Clean the projects
2. Build the hardware platform
3. Analyze the file xparameters.h (correct the main code if necessary)
4. Build the software application
5. Run the application (source code available on eLearning)

Lab. 7

Lab. 7 - Description

- Implementar um contador horário mm.ss (*countdown timer*), com contagem decrescente, com um comportamento do ponto de vista do utilizador semelhante ao do trabalho prático 6, em que:
  - o controlo, a atualização temporal e a interface com o utilizador continuam a ser realizadas por software a executar sobre um SoC baseado no processador MicroBlaze (tal como no trabalho prático 6).
  - mas a **interface com o display passa a ser realizada através de um periférico de hardware especializado**, o qual controla a atualização/refrescamento periódico dos dígitos, tendo o software que interagir com esse periférico apenas para alteração dos valores mostrados.



- Desta forma, pretende-se libertar o software da tarefa de refrescamento periódico do display, passando esta operação para hardware dedicado.
- **O periférico especializado deverá integrar o módulo "Nexys4DisplayDriver"** desenvolvido no trabalho prático 3.

Lab. 7 - Description

- O seu modelo de programação deverá consistir em diversos registos mapeados no espaço de endereçamento do processador MicroBlaze, que disponibilizem ao software os seguintes campos:
  - **Digit Enables** - 8 bits para (des)ativação individual de cada dígito;
  - **Decimal Point Enables** - 8 bits para (des)ativação individual de cada ponto decimal;
  - **Digit Values** - 32 bits (8 x 4 bits) para configuração, em binário, do valor afixado em cada dígito;
- O *display driver* personalizado receberá esses dados pelo interface AXI-Lite.
- O *display driver* personalizado deverá produzir os seguintes outputs:
  - **an** – 8 bits
  - **seg** – 7 bits
  - **dp** – 1 bit
- O módulo *GPIO_Displays* associado ao display já existente tem de ser removido do projeto

Plano & Partes:

- Parte 1 - Adaptação e extensão da plataforma de hardware
- Parte 2 - Desenvolvimento do software de controlo baseado em polling ao registo de estado do timer de hardware (AXI Timer)
- Parte 3 - Desenvolvimento do software de controlo baseado em interrupções do timer de hardware

Final Remarks

At the end of this lecture you should be able to:

- Design custom hardware modules interacting with the MicroBlaze through AXI-Lite interface
- write C programs that make use of custom hardware

To do:

- Construct the considered hardware platforms
- Test the given applications in Vitis
- Do lab. 7

Extra slides: LED Brightness Control via PWM

Nexys-4 Tri-color LEDs

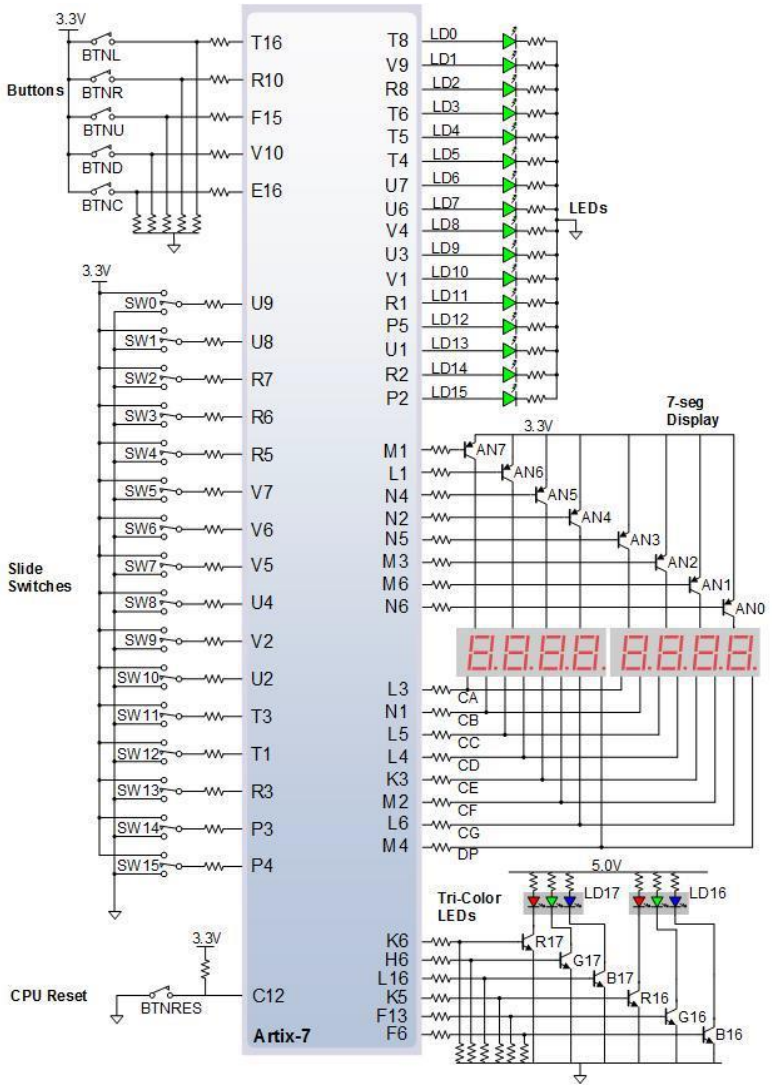
Each tri-color LED has three input signals that drive the cathodes of three smaller internal LEDs: one red, one blue, and one green.

Driving the signal corresponding to one of these colors high will illuminate the internal LED.

The input signals are driven by the FPGA through a transistor, which inverts the signals. Therefore, to light up the tri-color LED, the corresponding signals need to be driven high.

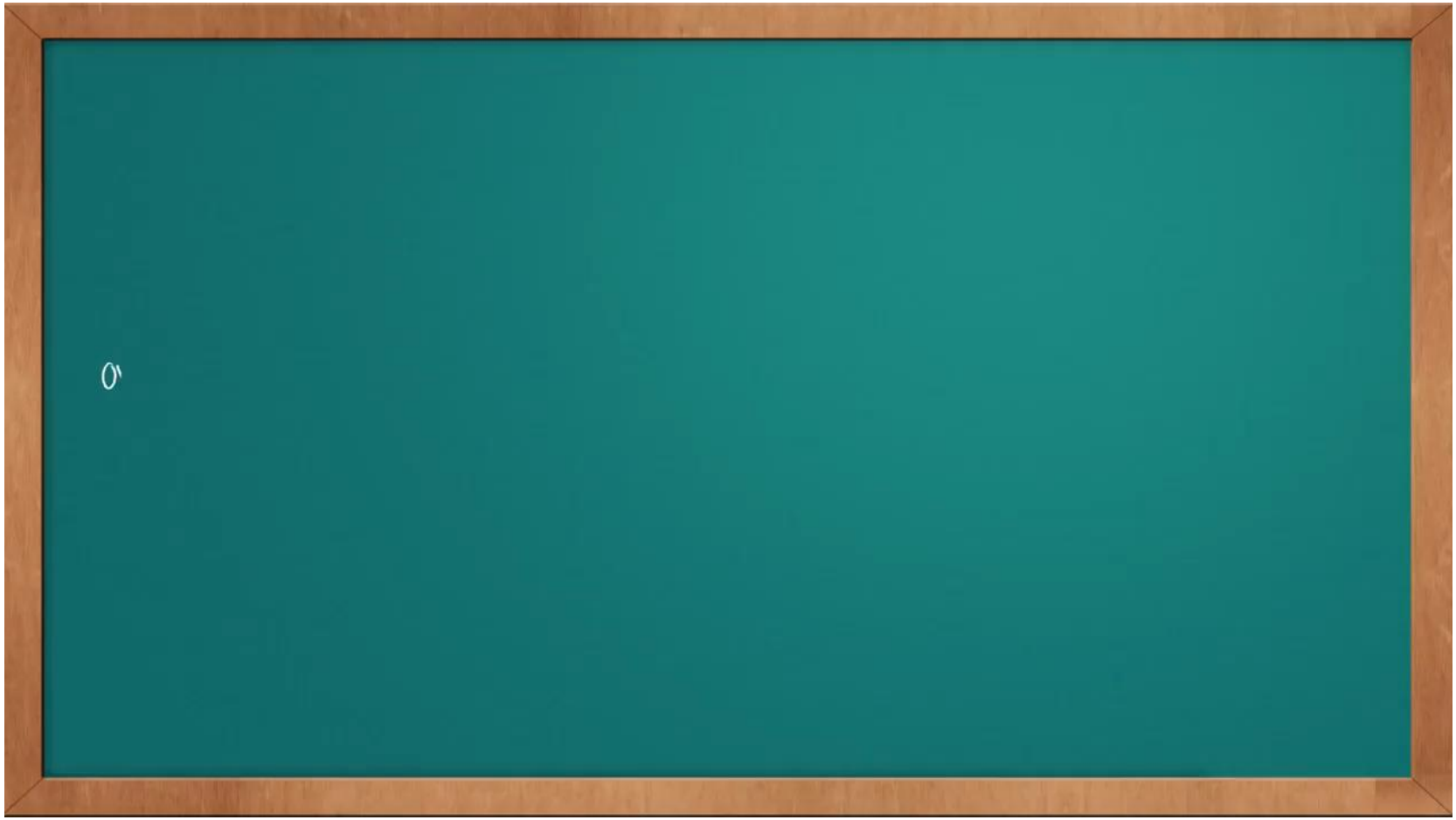
The tri-color LED will emit a color dependent on the combination of internal LEDs that are currently being illuminated.

Note: Driving any of the inputs to a steady logic '1' will result in the LED being illuminated at an uncomfortably bright level. You can avoid this by ensuring that none of the tri-color signals are driven with more than a 50% duty cycle.



Example 2 – triple PWM Generator

PWM – Pulse Width Modulation



PWM

Frequency – how many times per second the LED is switched on and off:

- Cannot be very slow to avoid flicker

Resolution - indicates how many intermediate steps (also called „units“) a PWM cycle has:

- In 8 bit mode, there are 256 levels of brightness

One easy way to build a PWM generator is to use two counters:

- Counter 1 (*s_clkEnbCnt*) limits the frequency of PWM pulses
- Counter 2 (*s_pwmCounter*) controls duty-cycle of PWM pulses (pulse width) within the chosen resolution
- PWM frequency =
- = system clock / (resolution * counter_1_MaxValue)