FYSIKUM

Digital System Construction

Lecture 6

Functions and Procedures in VHDL

Digital clock management

Serial communication

Lab introductions:

- Lab 5: RS232 serial receiver
- Lab 7: Arbitrary function generator

Functions and procedures

Functions in VHDL

- Similar to functions in programming languages
  - Contain sequential code
- Can be declared:
  - In an architecture or process before ‘begin’
  - Or within a package.
- Declaration includes
  - Arguments
    - List of inputs to the function, including type
    - Arguments are not changed by the function
  - return (result_type)

Example: buffer with enable

```
architecture arch1 of buffer is

function enable (d, en: std_logic) return std_logic is
begin
    if en = '0' then return '0';  -- Disable output
    else return d;
    end if;
end enable;

signal data, enb : std_logic;

begin  -- architecture

q <= enable(data, enb);  -- Use function in architecture

end arch1;
```

Example: vector to integer

```
function vec_to_int (x: std_logic_vector) return integer is

variable result: integer;

begin
    result := 0;
    for i in x'range loop
       result := result * 2;          -- Shift output to left
       case x(i) is
          when '1' => result := result + 1;  -- Add 1
          when others => null;
       end case;
    end loop;
    return result;

end vec_to_int;
```

Procedures

- Similar to functions, in that:
  - They also contain sequential code
  - Declared in architecture or process (before begin) or within packages
- No ‘return’ value
  - Instead, procedures can change the arguments
- Procedures declared with an argument list
  - Arguments have both type and direction
  - Arguments that are `out` or `inout` can be changed

D flip-flop with enable

```
architecture arch1 of register is

procedure dff                          -- declare procedure
   (signal d, en, clk: in std_logic;
    signal q: out std_logic) is  -- q is an output
begin
   if en = '0' then                    -- output enable (async)
      q <= '0';
   elsif rising_edge(clk) then
      q <= d;
   end if;
end dff;

signal din, enable, clock, dout: std_logic;

begin                                  -- Begin architecture
   din <= input_port;
   dff(din, enable, clock, dout); -- call procedure here
   output_port <= dout;
end arch1;
```

Using functions and procedures

- Functions and procedures are both subprograms
- Can help in writing compact, readable code.
- Functions commonly used for:
  - Simple expressions for complex functions
    - example: parity/checksum generation
  - Defining operators for custom signal/variable types
  - Only combinatorial code
- Procedures good for:
  - Simple alternative to component declaration/instantiation
  - Sequential code with clocks
  - Useful in writing complex test benches

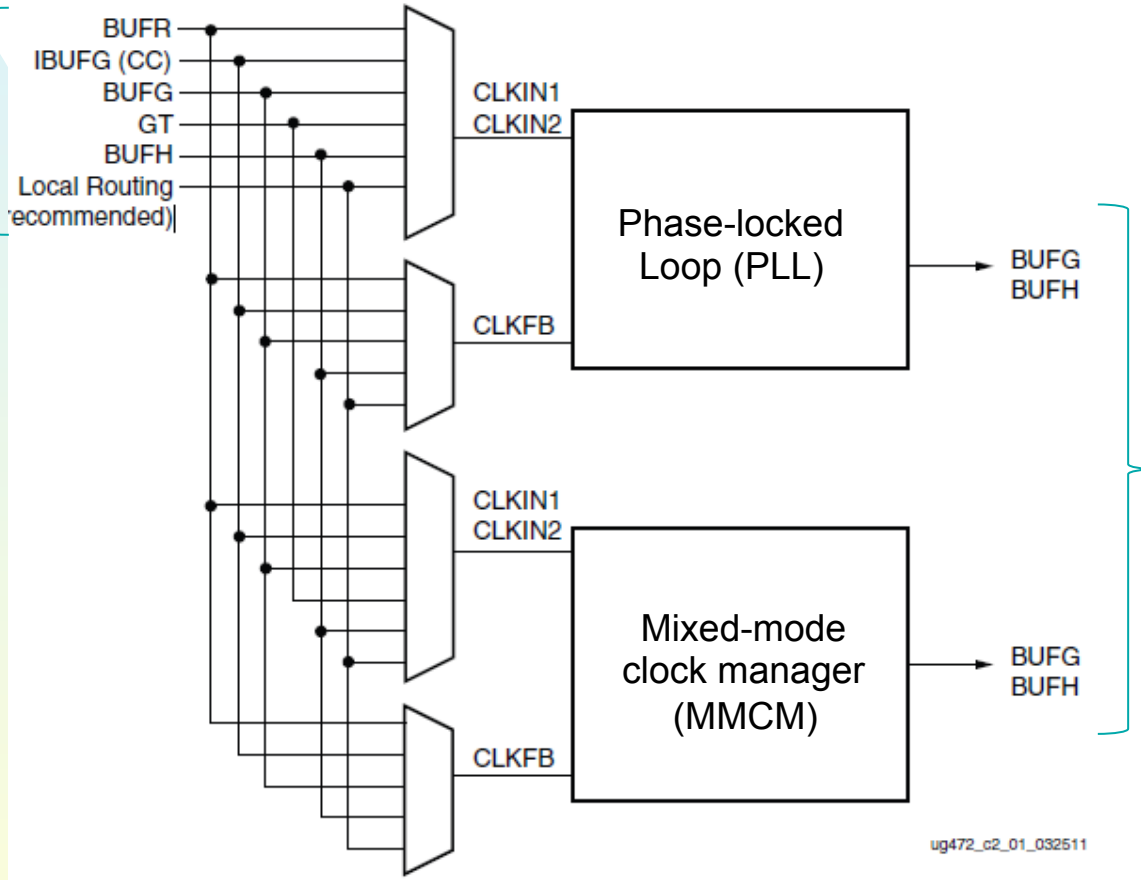
Clock Management

Clock synthesis/conditioning

- Cleaning and distribution:
  - Re-synthesize input clock to 50% duty cycle
  - Jitter removal w/ phase-locked loop (PLL)
  - FPGA input buffer delay compensation
- Synthesizing new frequencies
  - multiply and divide input clock frequency by set values
- Timing alignment
  - Produce clocks with same frequency but shifted in phase by 90º, 180º, 270º, etc.
  - Fine delay setting

Xilinx 7 series clock management tile (CMT)

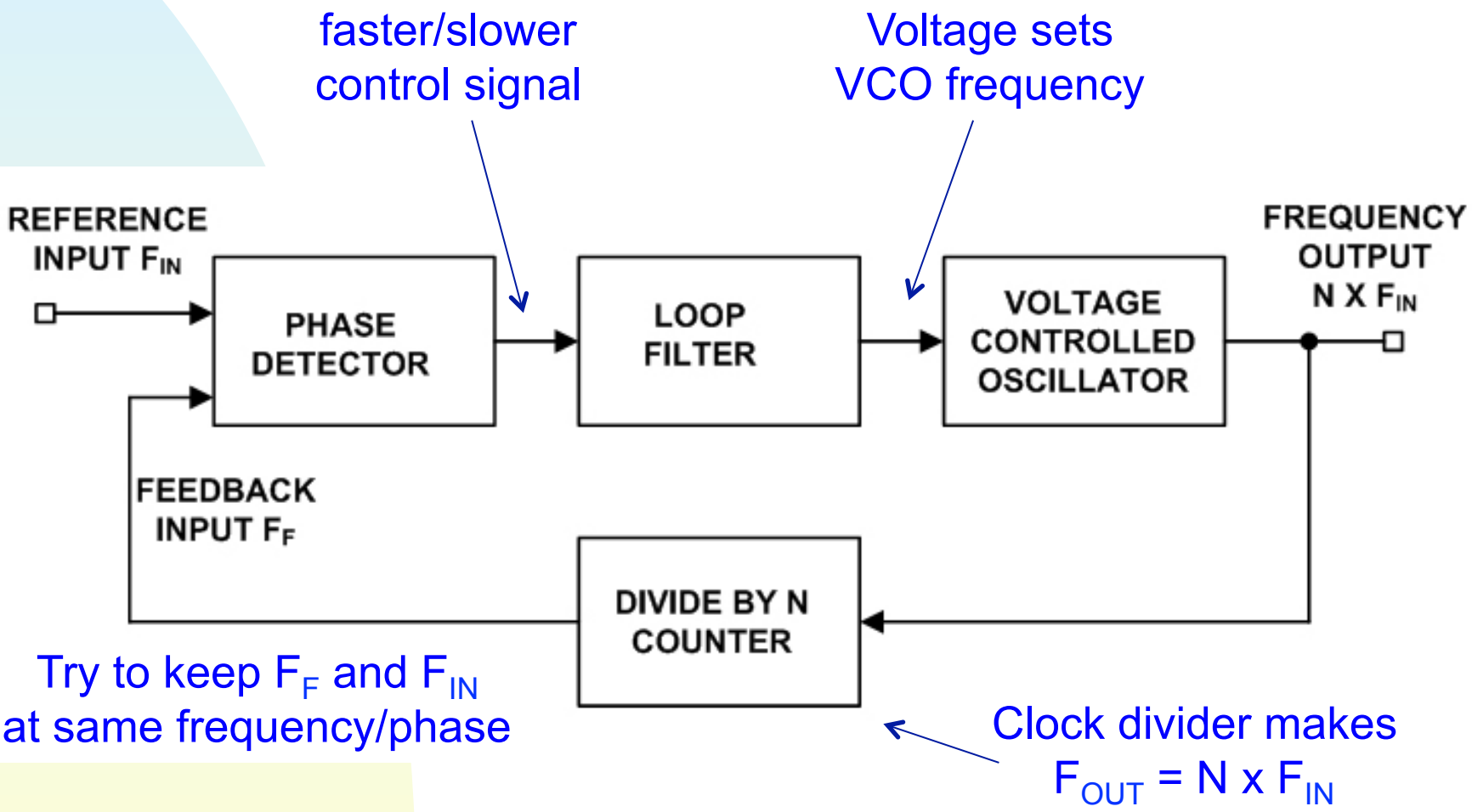
Input distribution



Output buffers

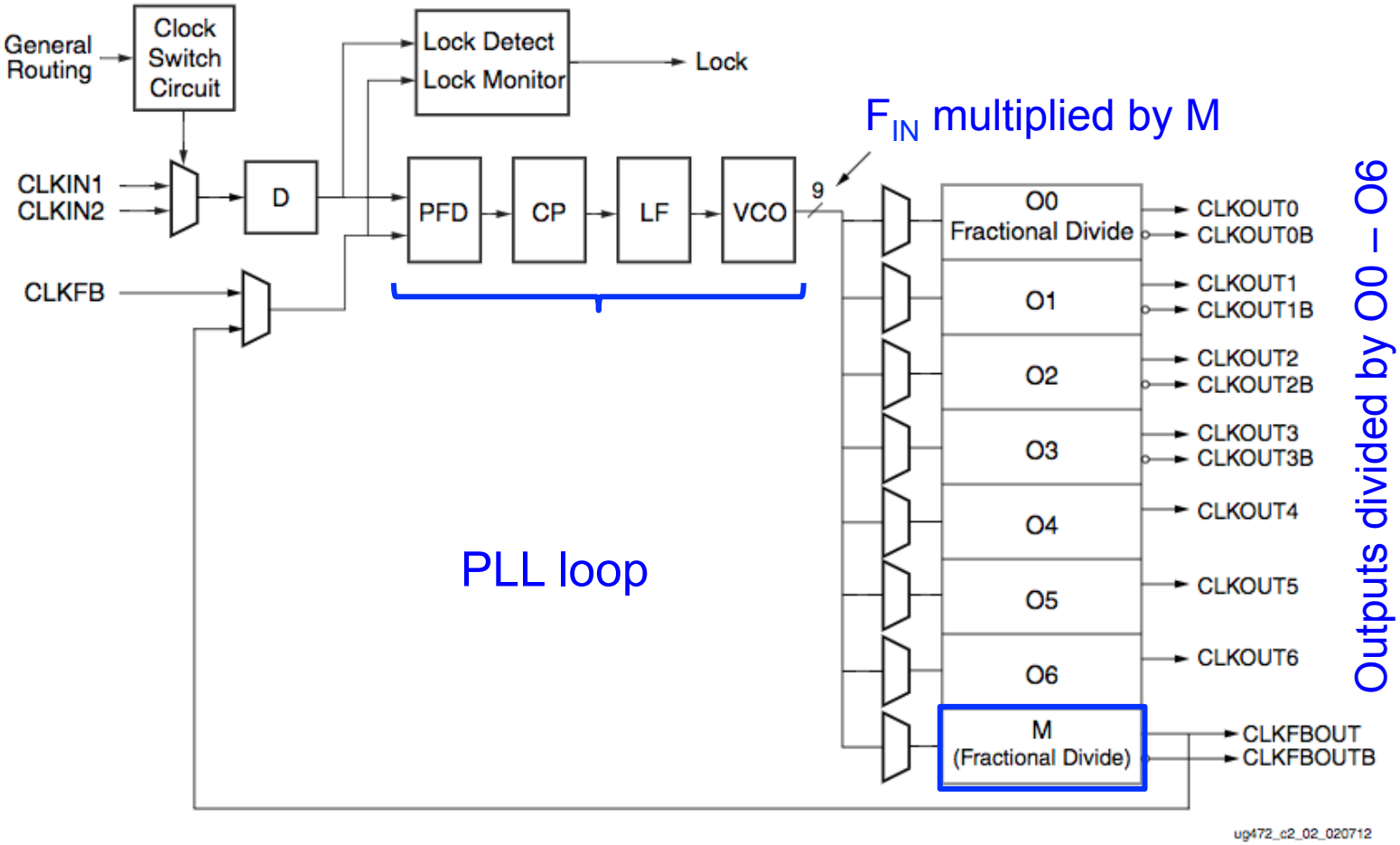
Figure 3-1: **Block Diagram of the 7 Series FPGAs CMT**

What is a PLL?


faster/slower
control signal
Voltage sets
VCO frequency
REFERENCE
INPUT F_{IN}
PHASE
DETECTOR
LOOP
FILTER
VOLTAGE
CONTROLLED
OSCILLATOR
FREQUENCY
OUTPUT
N X F_{IN}
FEEDBACK
INPUT F_F
DIVIDE BY N
COUNTER
Try to keep F_F and F_{IN}
at same frequency/phase
Clock divider makes
$F_{OUT} = N \times F_{IN}$

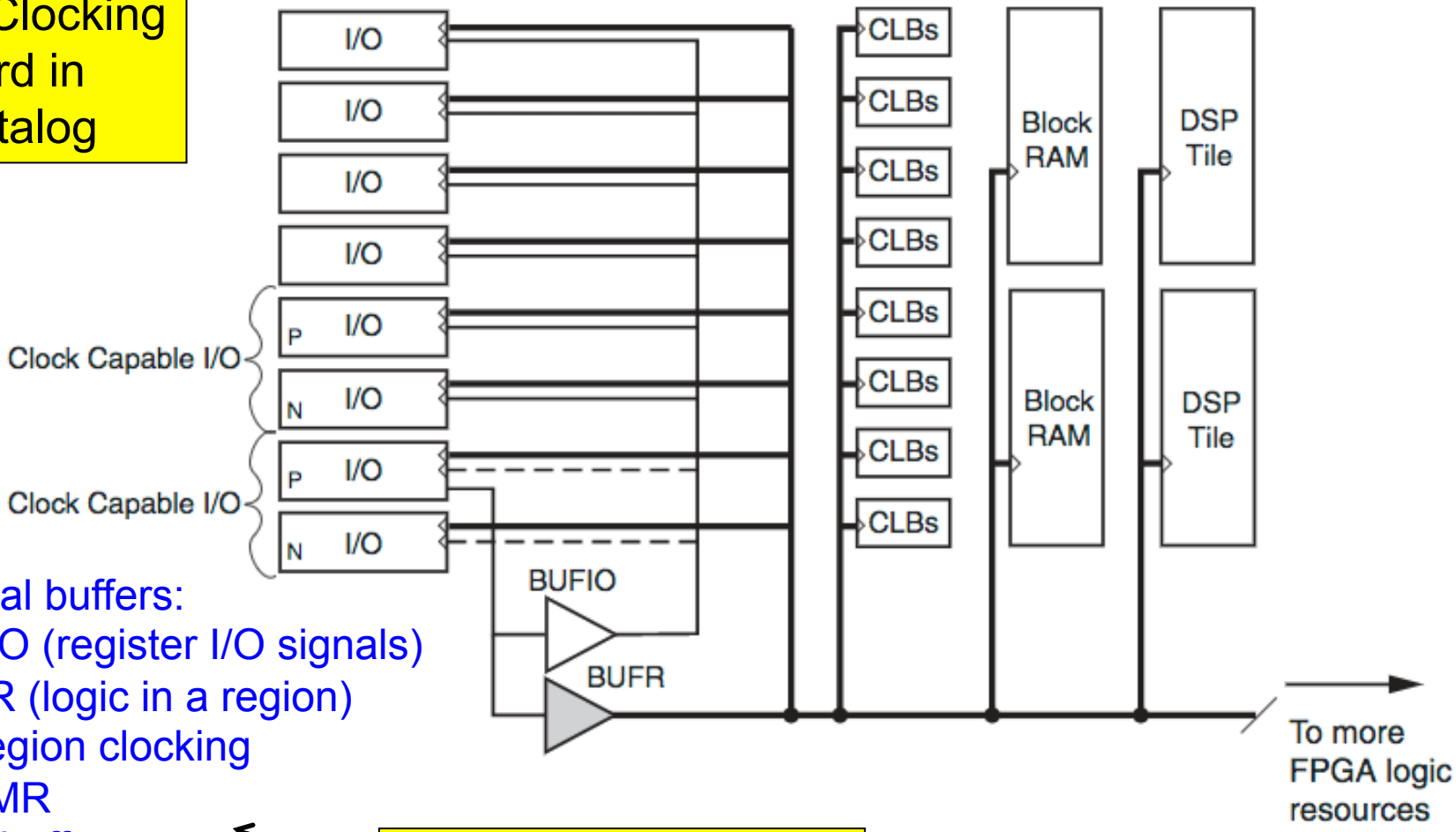
MMCM block diagram

(PLL block is similar)



Clock buffers/distribution

See Clocking Wizard in IP catalog



Regional buffers:
- BUFIO (register I/O signals)
- BUFR (logic in a region)

Multi-region clocking
- BUFMR

Global buffers:
- BUFG

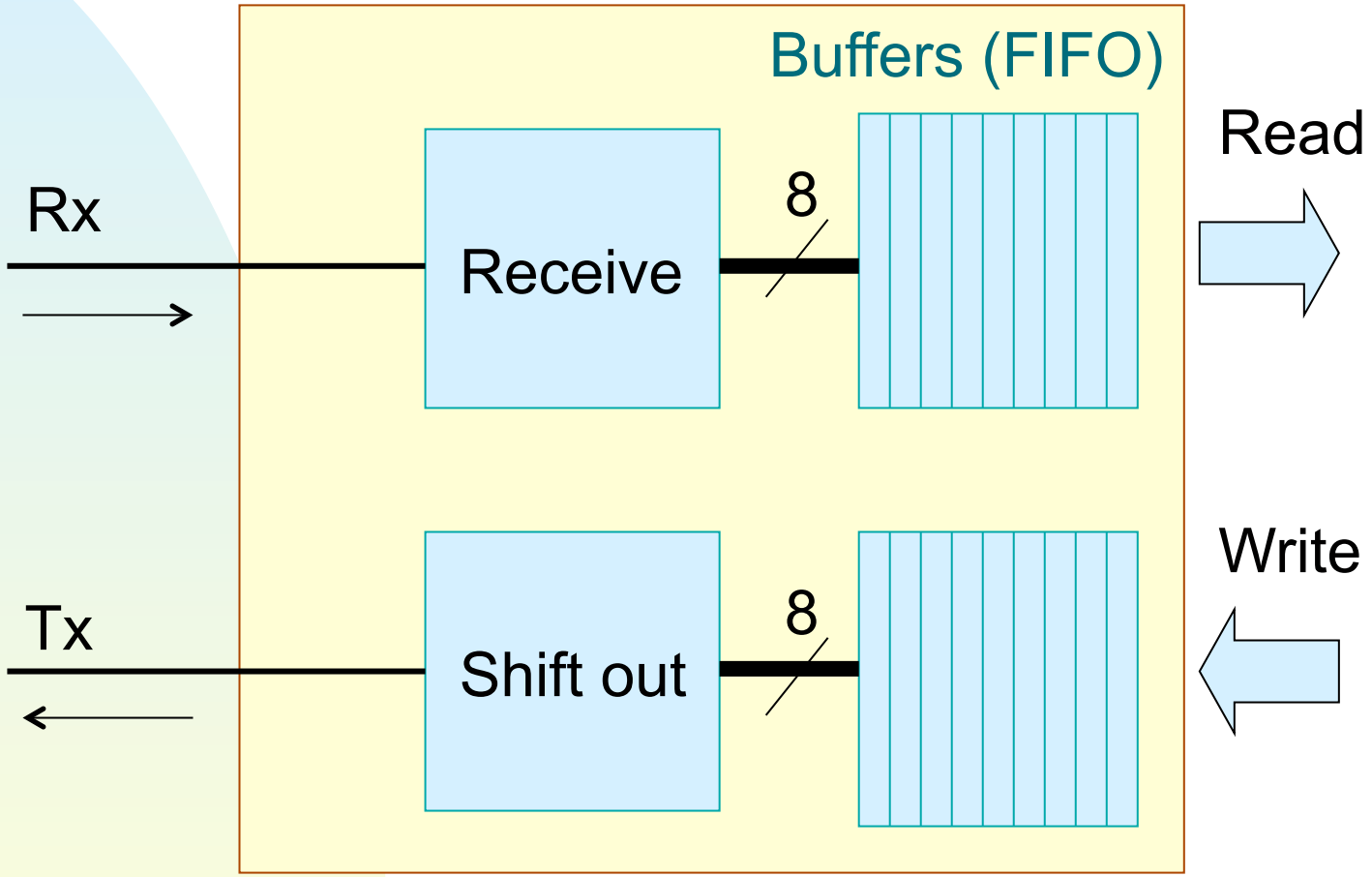
Not an exhaustive list

Lab 5: UART receiver
(serial data receiver/decoder)

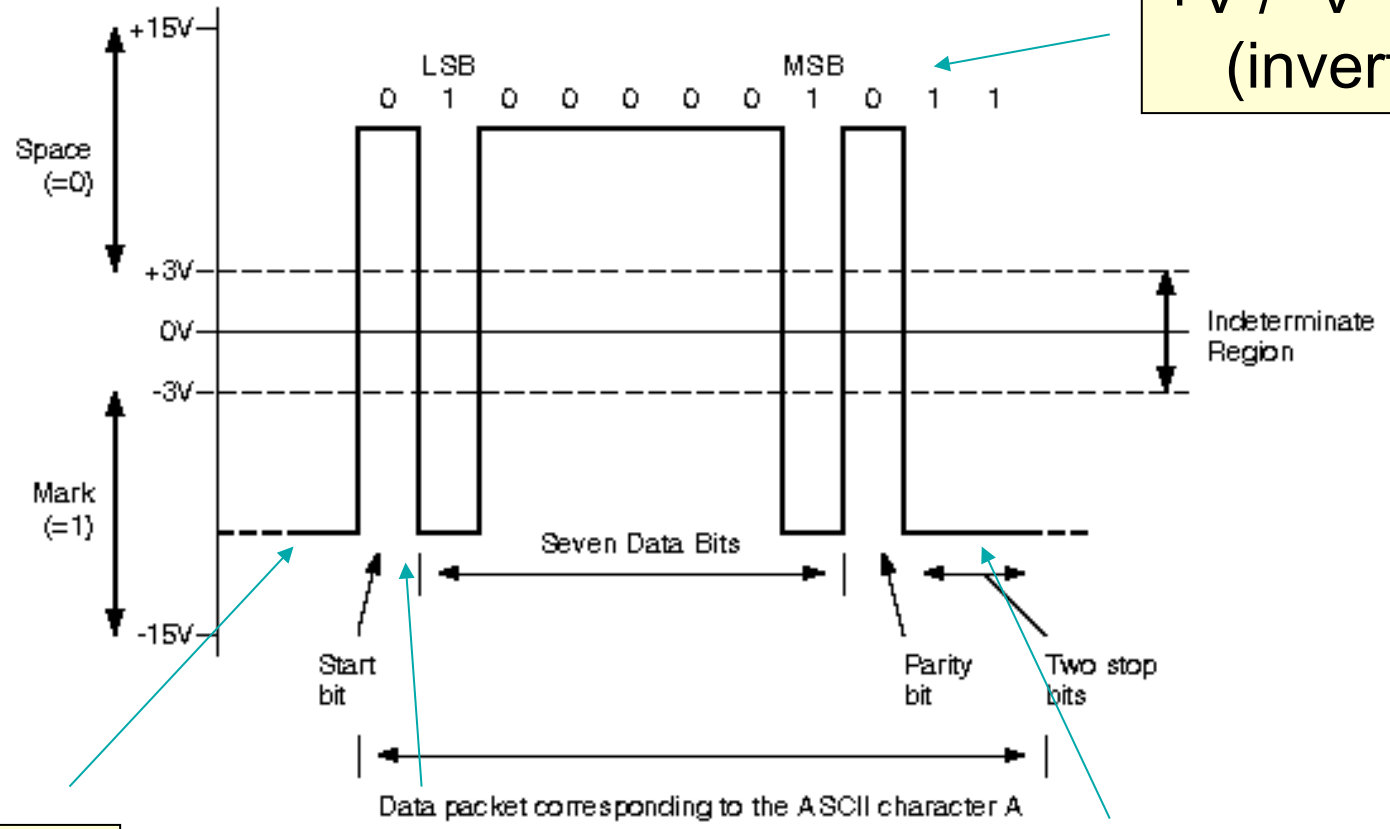
What is a UART?

- Universal Asynchronous Receiver and Transmitter
- Typically used for serial data transmission (e.g. RS232)
- Common implementations:
  - Discrete component, or
  - Embedded in FPGA or ASIC

UART diagram (simplified)



RS232 data format (physical)



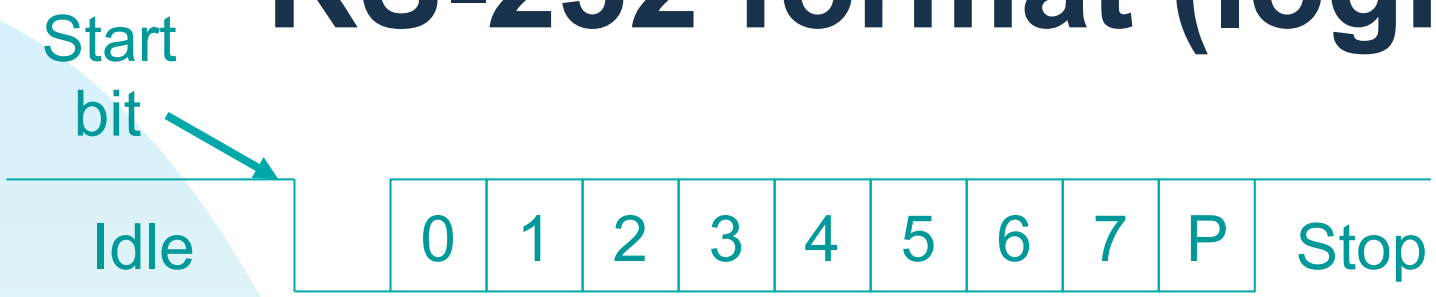
+V / -V = 0 / 1
(inverted)

Idle: 1

Start bit: 0

Stop bit(s): 1

RS-232 format (logical)

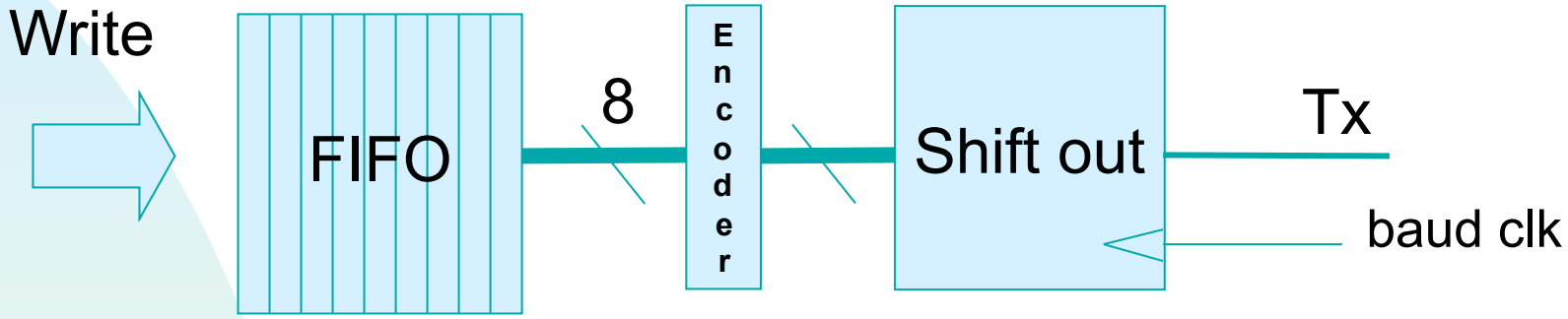


- RS-232 communication is asynchronous:
  - Clock signal not sent with the data.
  - Each word synchronized using its start bit
  - Receiver reads data with local internal clock (defined by baud rate)
- Signal is logical '1' while idle, packet ends with '1'
- Start bit ('0') signals that data is about to be sent.
- Up to 8 bits of data sent. Optional parity bit can be added. Finally, stop bit ('1') is sent.

RS232 formats (options)

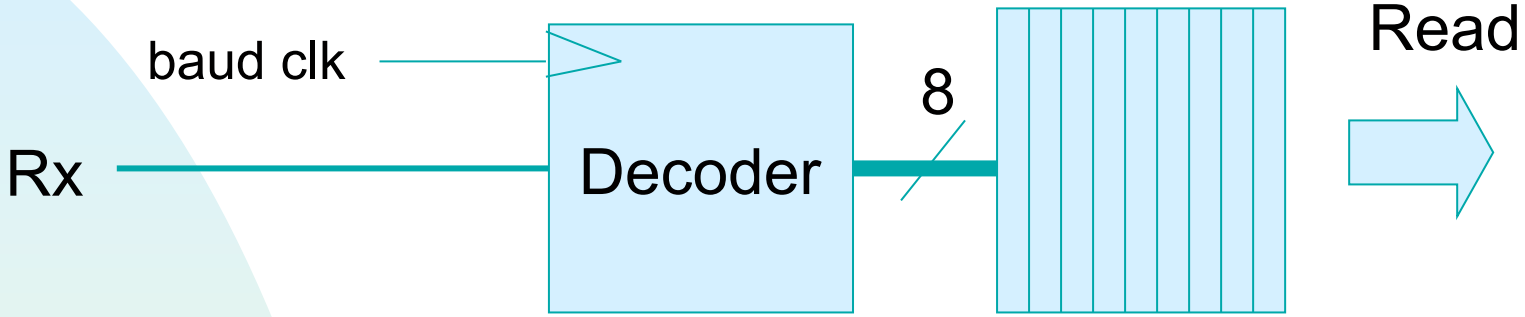
- Baud rates (bits/second):
  - 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 115200
- Data bits
  - 5, 6, 7, or 8
- Parity bit
  - odd/even/none
- Stop bits (minimum time between words)
  - 1, 1.5 or 2

Simple UART transmitter



- Data written to asynchronous pipeline buffer (FIFO)
- A state machine reads data bytes one-by-one
  - Adding start/stop bits, and parity (if used)
- Data words are then sent to a shift register
  - Bits transmitted one-by one at baud rate
  - Correct order: start, data (LSB first), parity, stop

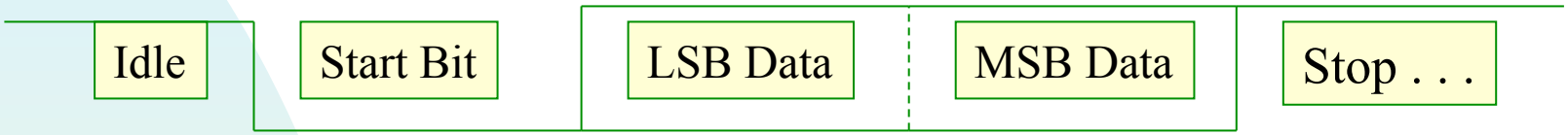
"Simple" UART receiver



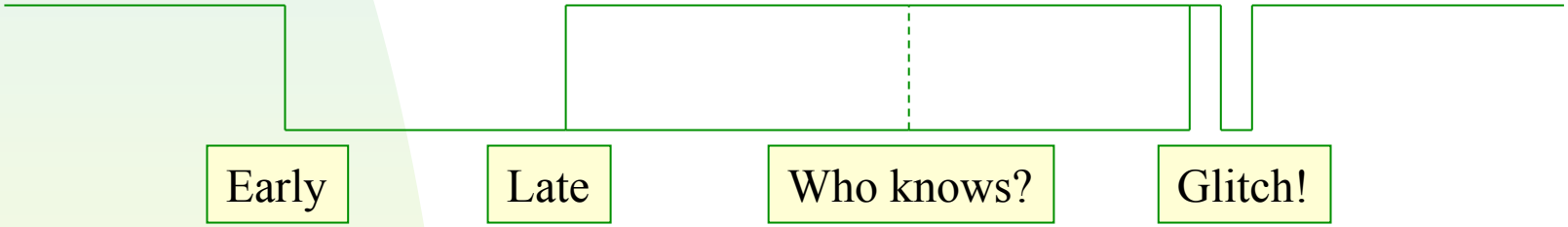
- Naively:
  - A decoder receives and processes bits one by one into a shift register at the baud clock rate
  - Decoded output written in parallel to a readout FIFO
- Problems with this approach:
  - Data arrives asynchronously,
    - Don't know when data will arrive!
  - Transmitter/receiver clocks have slightly different frequencies, out of phase

UART Receiver Sampling

- "Ideal" serial data waveform:



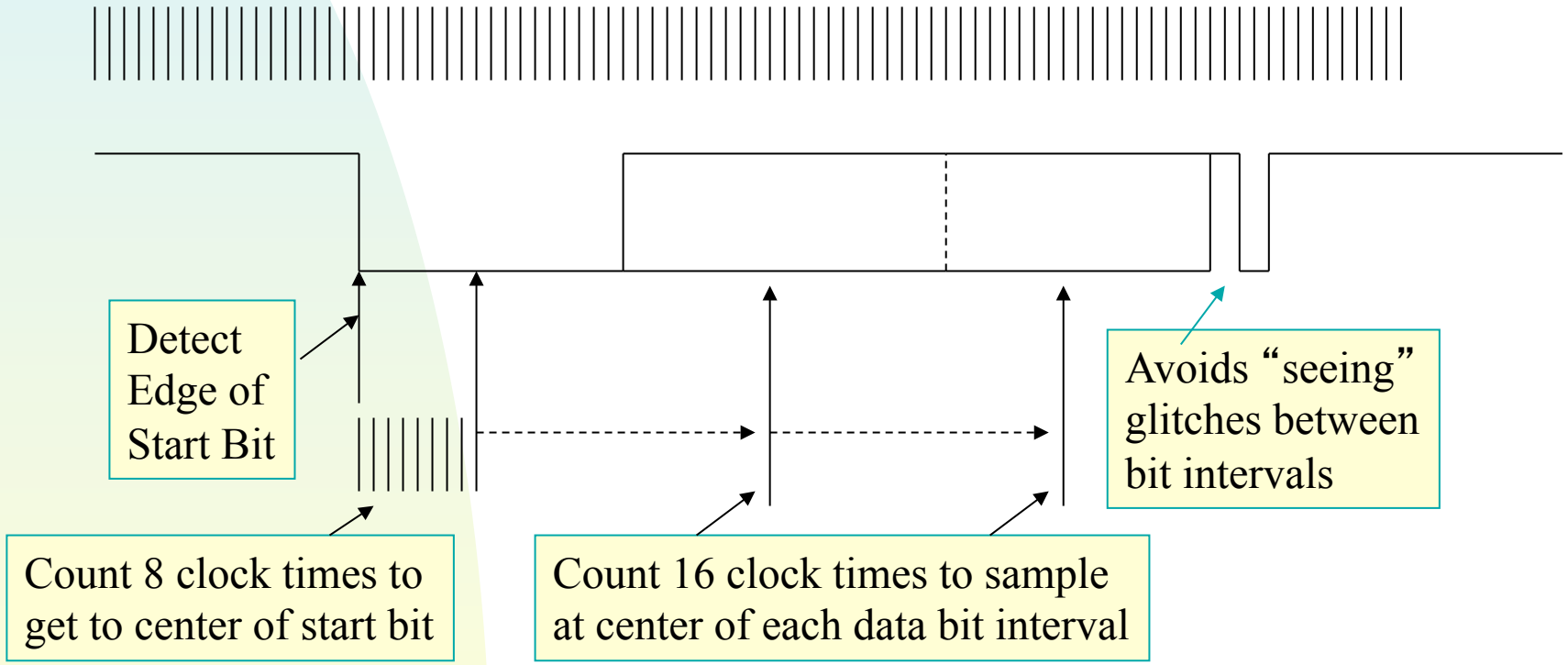
- What the receiver "sees":



- Receiver needs to "center-sample" the data bits to assure proper reception (or even take several samples)

UART Receiver Sampling

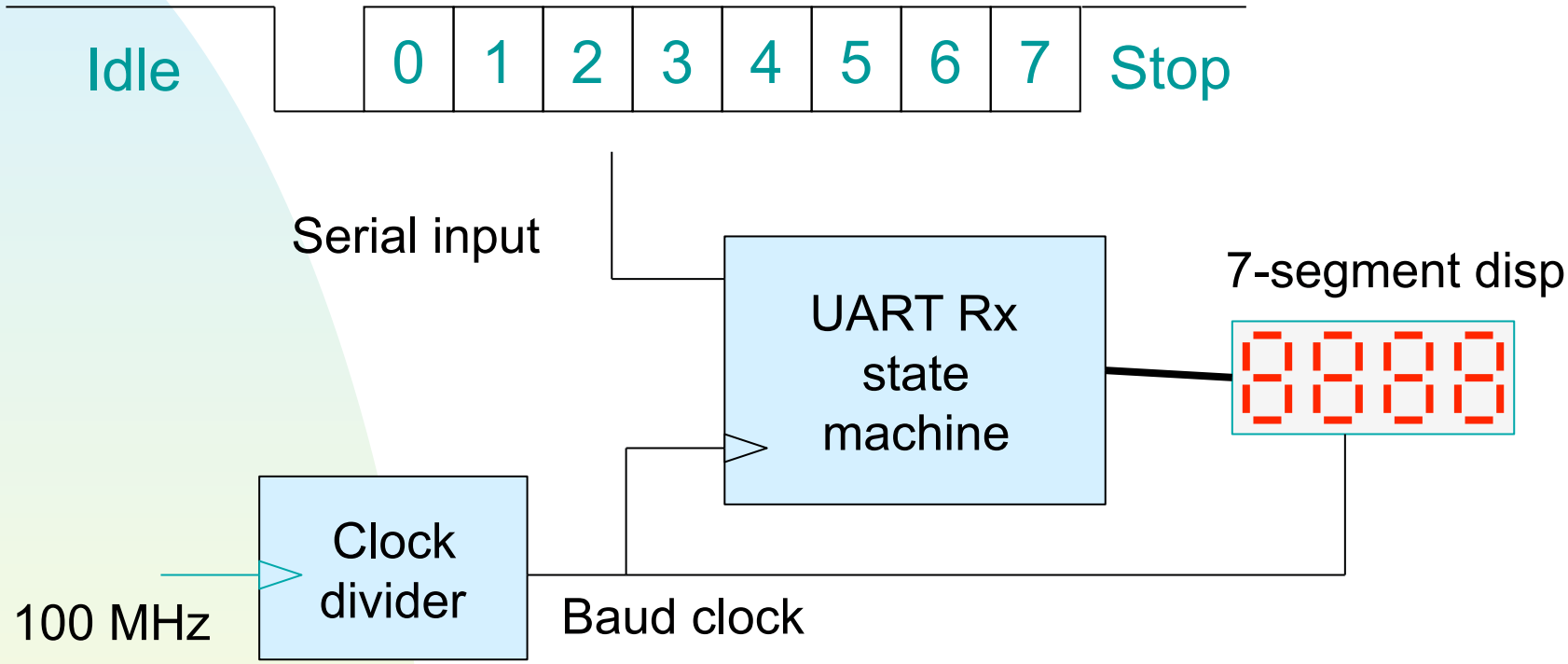
- Receiver samples the input (Rx) signal at (for example)16 times the baud rate:



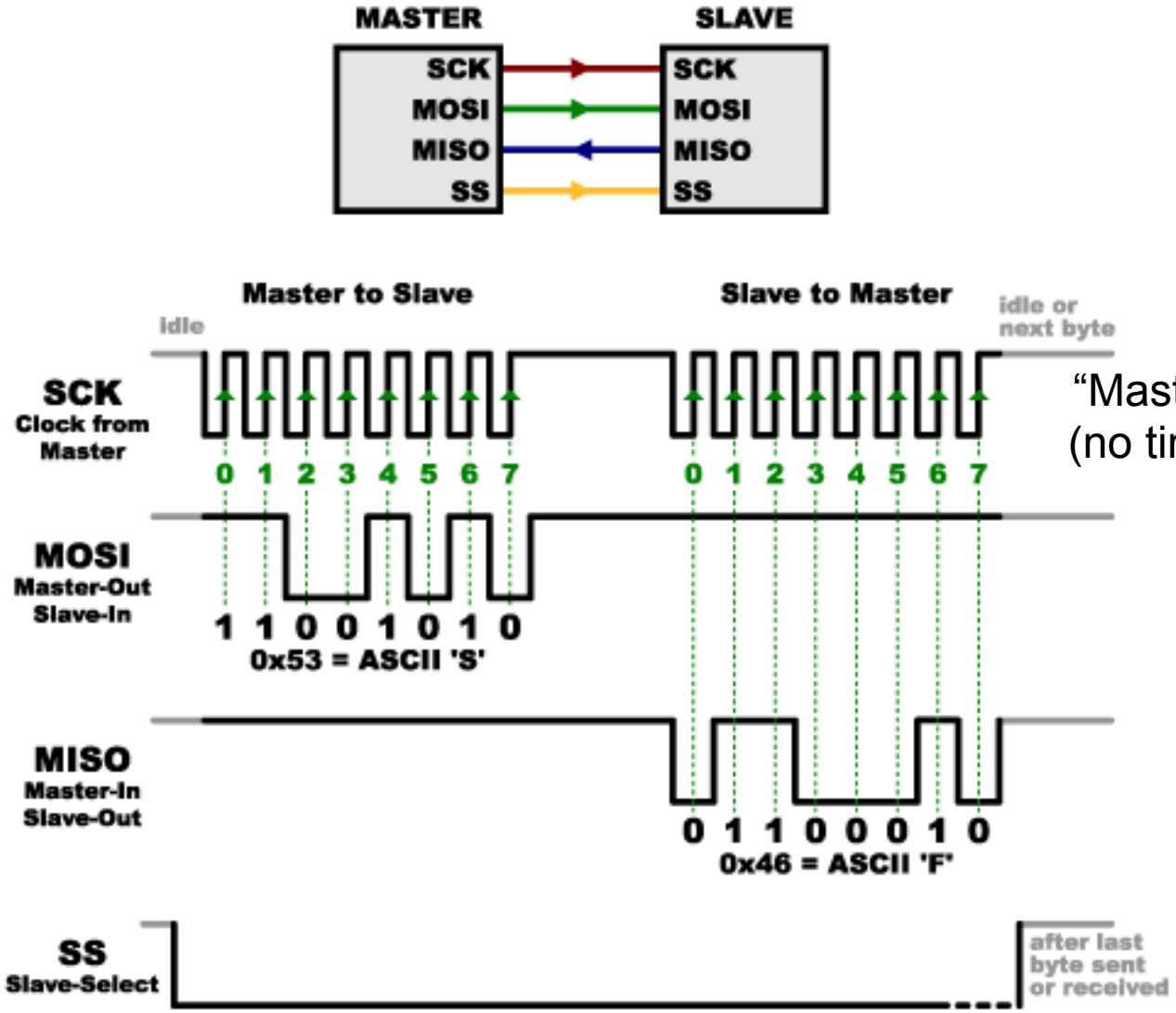
Receiver timing

- UART receiver clock derived from a high-frequency local oscillator (e.g. 100 MHz)
- Can use a counter to divide the clock
  - Maximum counter value (before resetting to zero) is a divisor
    - Divisor = freq / (num_samples * baud_rate)
  - Match output clock to baud rate by (much) better than 5% to avoid data misalignment
- Best to generate a symmetric baud clock
- MMCM not suitable here (too slow!)
  - But essential for (e.g.) high-speed links!
- Distribute with a GBUF for best results

UART Rx block diagram



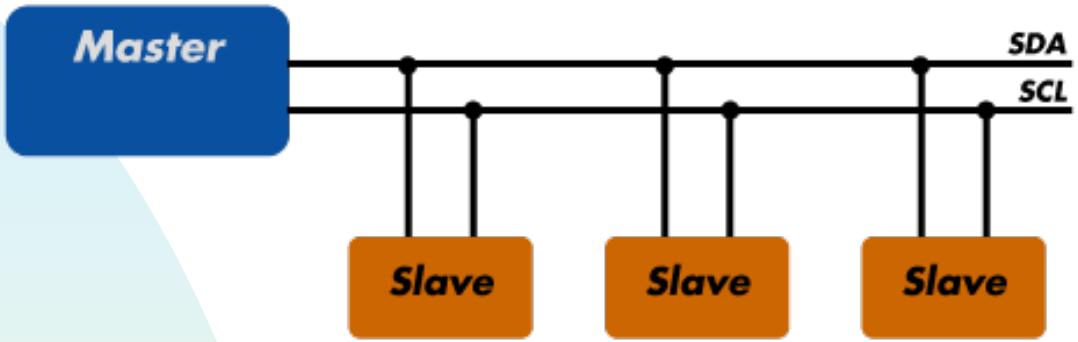
Other protocols: SPI



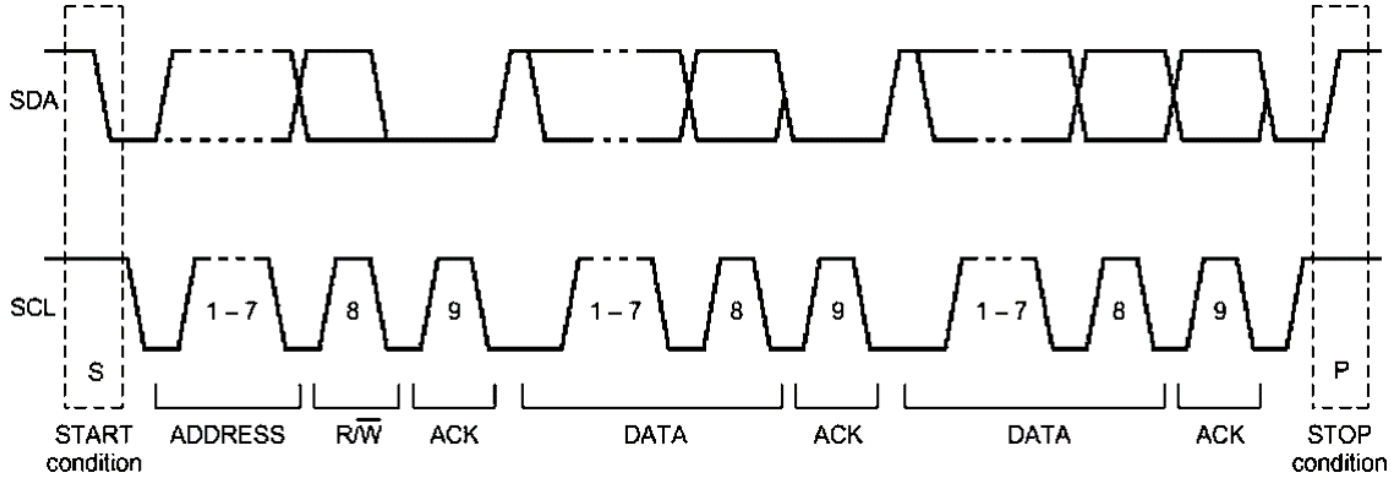
"Master" provides the clock (no time-alignment needed)

"Slave" response is pre-defined

Other protocols: I2C



Shared tri-state data bus
Master provides clock



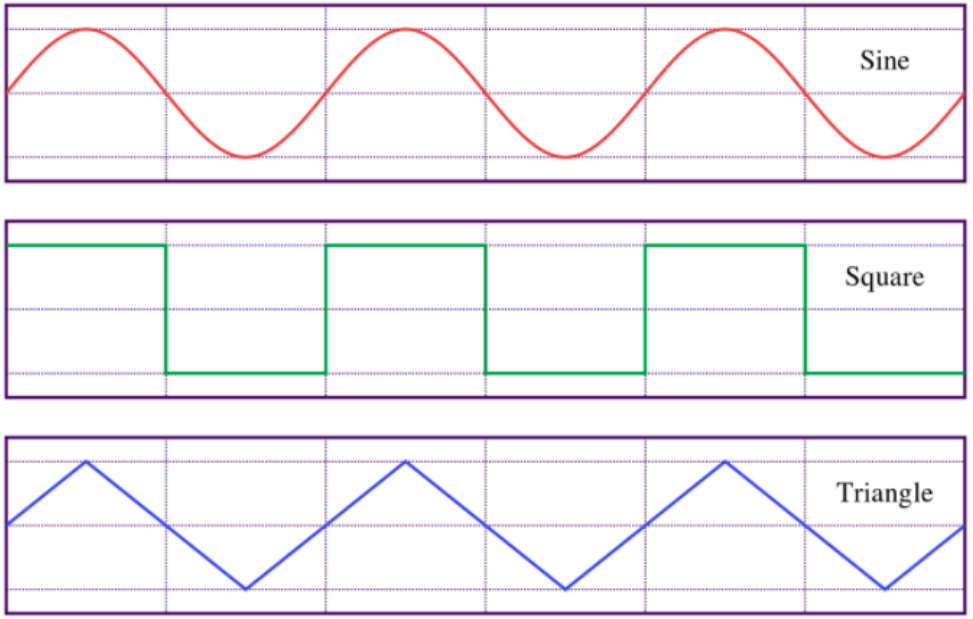
Protocol example

Lab 5

- Design the receiving part of a UART circuit:
  - Receive RS232 words
    - 8 bits of data, no parity, at 9600 Baud
  - Extract and store the 8 data bits in a register
  - Display data on the 7-segment hexadecimal LED display
- Write a test bench and simulate
- Synthesize for the FPGA board and test with real serial data.

Lab 7 : Arbitrary waveform generator

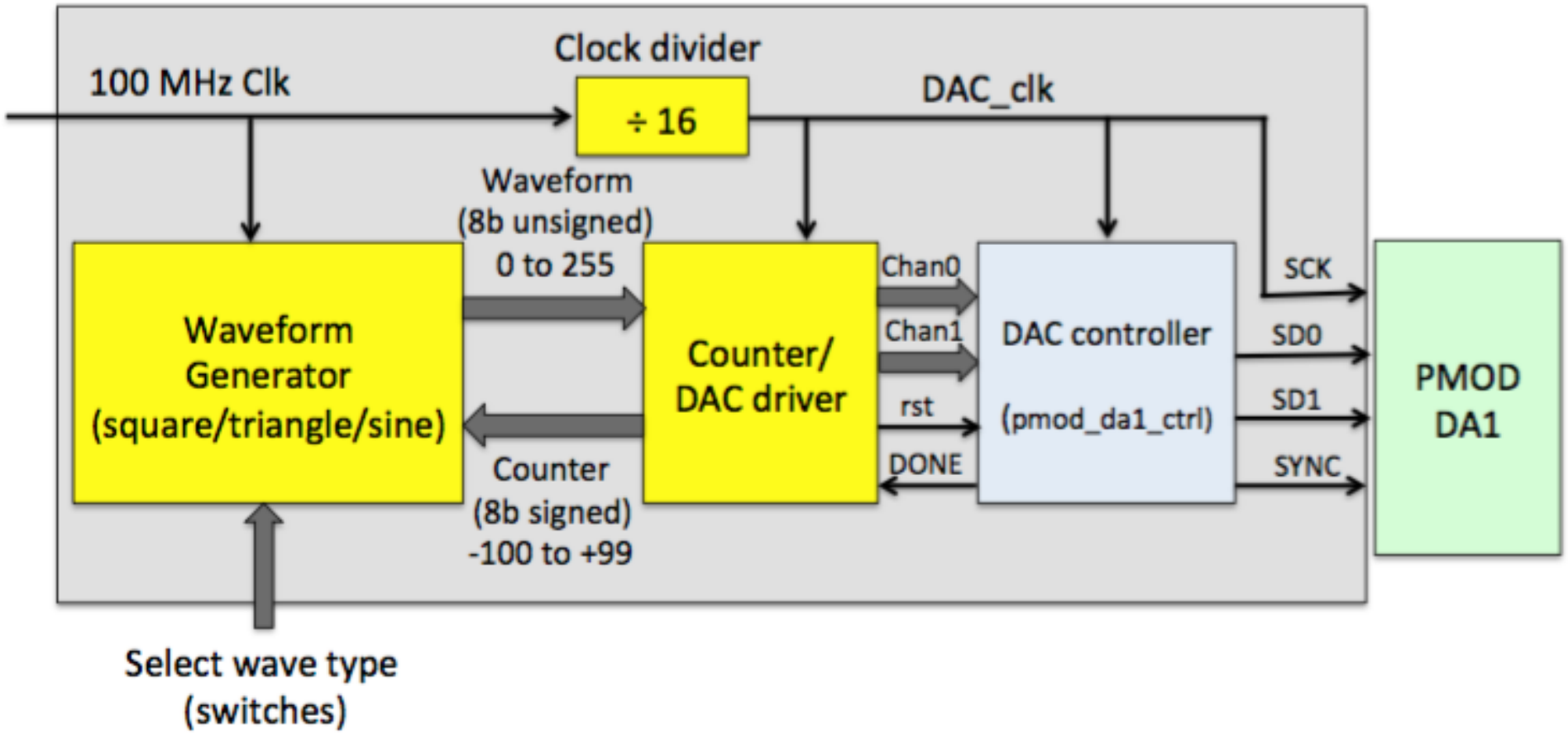
Typical waveform generator outputs:



Project goals

- Build a simplified digital function generator
  - Square, triangle and sine waves
    - User-selectable with switches/buttons
  - Fixed frequency and amplitude
    - For simplicity
    - Can try to make it adjustable if you like…
- Produce an analog output
  - Use PMOD-DA module
    - Driver IP available on course web page

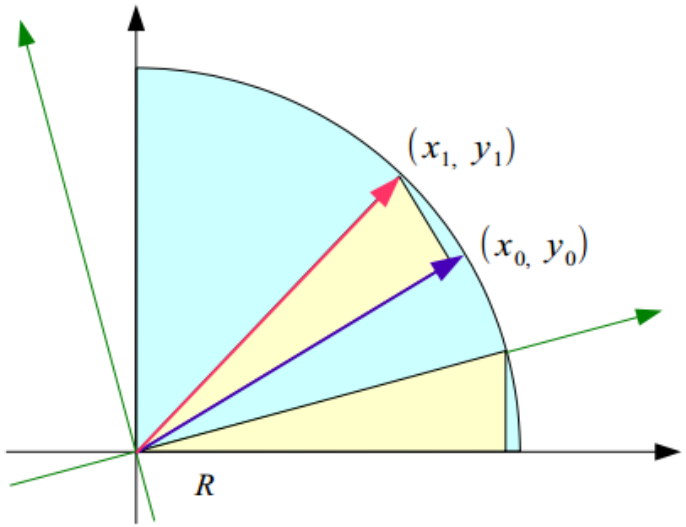
Design overview

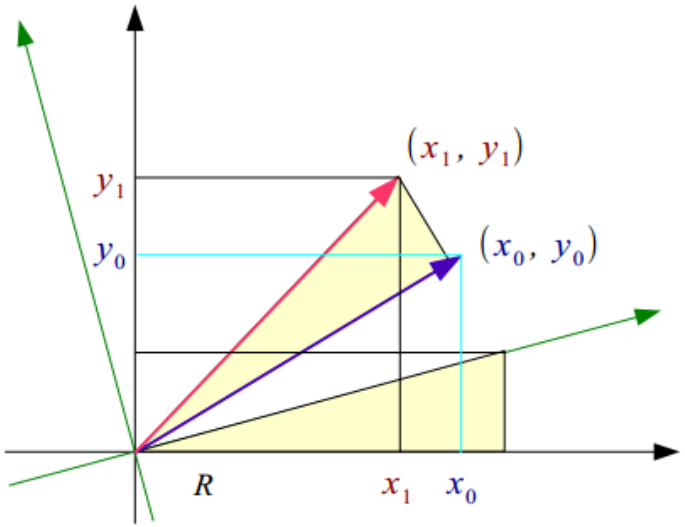


Waveform generator

- Input: signed 8b vector from the counter/DAC driver module. Range -100 to +99.
- Output: unsigned 8b vector (0 to 255)
  - Square wave (simple)
    - Could just use the sign of the input, for instance
  - Triangle wave (slightly less simple)
    - Several ways to do this, including counting up/down, sign-dependent adding/subtracting from a constant, etc.
  - Sine wave (challenging)
    - Not directly supported operation in VHDL
    - One “brute force” method is a RAM lookup table
    - Less known, but small footprint: CORDIC algorithm

Basic CORDIC concept (sin + cos)





rotate by θ

(x_0, y_0) → (x_1, y_1)

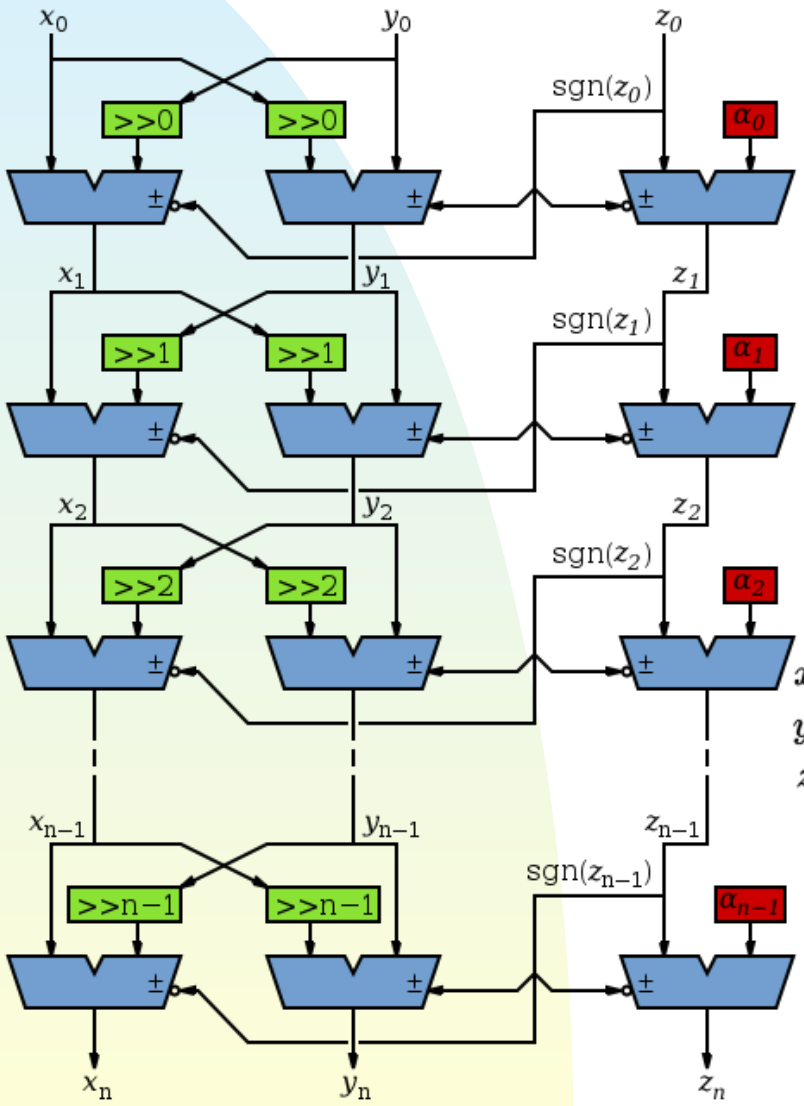
$$x_1 = x_0 \cos\theta - y_0 \sin\theta$$

$$y_1 = x_0 \sin\theta + y_0 \cos\theta$$

Originally developed for sine/cosine.
Later extended to include:
tan, hyperbolics, square root, etc.

Can configure and generate in Vivado IP Catalog

Cordic algorithm (pipelined)



- Can also implement in iterative(loop) architecture.
- A small LUT provides the values of α_n
- n-bit CORDIC calculation requires n steps/iterations

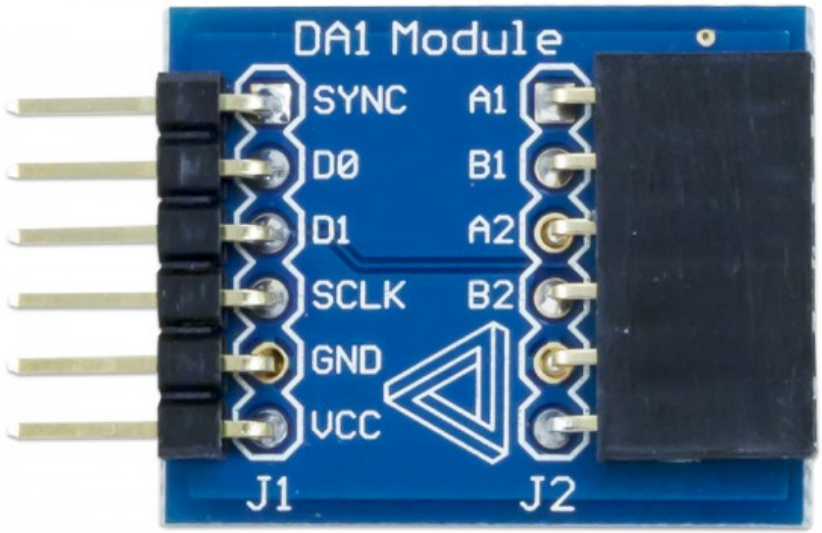
CORDIC in rotation mode:

$$x_{i+1} = x_i - d_i y_i 2^{-i}$$
$$y_{i+1} = y_i + d_i x_i 2^{-i}$$
$$z_{i+1} = z_i - d_i \alpha_i$$

$$x_n = K\left(x_0 \cos\theta_0 - y_0 \sin\theta_0\right)$$
$$y_n = K\left(y_0 \cos\theta_0 + x_0 \sin\theta_0\right)$$
$$z_n = 0$$

Can find good CORDIC articles and resources on the net…

PMOD DA1 module



- Standard form factor, use any PMOD port on the BASYS3
- Dual DAC channels (1-2)
  - 8 bit dynamic range
  - 0 – 3.3V
- Serial interface (SPI-like)
  - Two output channels D0, D1
  - Serial clock (SCLK) max 25 MHz
  - The SYNC signal enables a new write sequence while it is low
- VHDL IP module provides a simplified interface
  - Start write sequence with “rst”
  - Returns “DONE” when finished

Lab 5

- Design an arbitrary waveform generator that provides the following:
  - Square, triangle and sine wave functions
  - Wave function user selectable (switches/buttons)
  - Fixed frequency and amplitude
  - Drive analog output through PMOD-DA1 module
- Write a test bench and simulate
- Implement on BASYS3 board and test output with an oscilloscope.