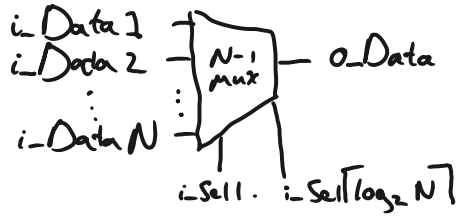
R6 : Common FPGA Modules

RM ch6-8

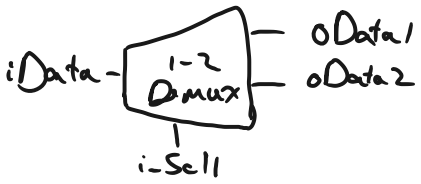
Data Controllers

- **Multiplexers :** Many ins , One out




```
assign oData = !i_Sel1 & !i_Sel2 ... & !i_SelN ? i_Data1 :
               !i_Sel & ...  ? i_DataN-1 : i_DataN
```

- **Demultiplexer :** One in , Many out




```
module Demux1to2 (input iData,
                  input iSel1,
                  output oData1,
                  output oData2
                 );
  assign oData1 = !iSel1 ? iData : 1'b0;
  assign oData2 = iSel1 ? iData : 1'b0;

endmodule
```

The Shift Register : A series of flipflops where their outputs are inputs of others. Used to delay data and for converting parallel ↔ serial

Delaying Data : Register shifting takes 1 clock cycle, therefore used.

```
reg [3:0] r_Shift;

always @ (posedge iClk) begin
    r_Shift[0] <= iDataToDelay;      +1 cc
    r_Shift[3:1] <= r_Shift[2:0];    +3 cc
```

] non-blocking / concurrent operations = 4 cc

Converting Serial and Parallel Data : Using an UART, an 8-bit register can send a byte serially by reading it out or receive it by shifting the bits through the register.

Receiving a Byte from LSB to MSB → **Transmitting** Byte

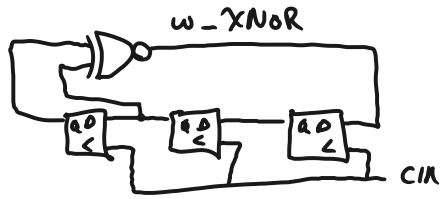
index	received bit	byte contents
0	1	1
1	1	11
2	0	011
3	1	1011
4	0	01011
5	0	001011
6	1	1001011
7	0	01001011

bit index	transmitted bit	byte contents
0	1	1
1	0	01
2	0	001
3	1	1001
4	0	01001
5	0	001001
6	1	1001001
7	0	01001001

Linear Feedback Shift Register

LFSR: Certain flip-flops are tapped into and used as input for an XOR or XNOR gate. The output of the gate is fedback into the input of the beginning.

- Used to generate pseudorandom strings
- XNOR generates true if inputs are even.



- Can be used as a counter — albeit uncommon due to its range being logarithmic and not linear count.

Traditional Counter

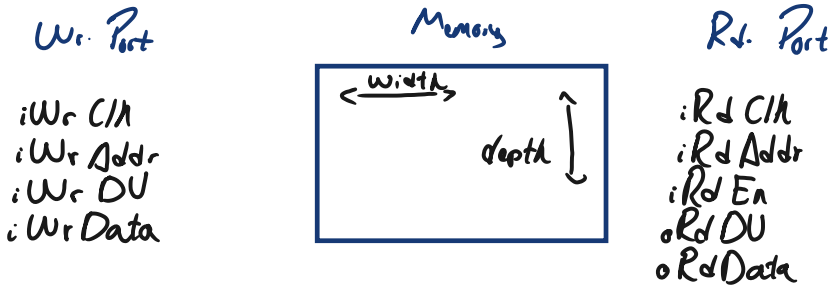
Using a CountLimit and a register `$clog2(CountLimit-1:0) rCounter;`

```
if rCounter == CountLimit begin
    rCounter = 0;
    :
else
  rCounter <= rCounter+1;
end
  :
```

Random Access Memory

RAM: Storing data & reading it later.

Single-Port can only write or read in the same clock cycle
Dual-Port can do both simultaneously.

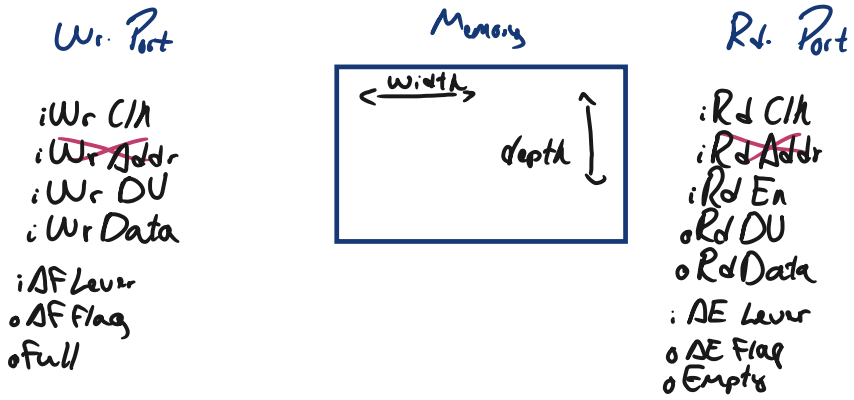


FIFO: First In, First Out

- Crossing Clock domains

Useful in FPGAs for buffering data between a producer and consumer.

A FIFO is a Ram w/ extra flags.



Addresses not requered — FIFO cycles data through memory registers.

The flags are useful to indicate when to write/read in bursts.

If AE - write, If Af - read.

R7

Synthesis

The equivalent of compilation, for FPGAs. Converting HW descriptions into simple components (LUTs, flip-flops, block RAMs, ...) that exists physically.

The physical components are the constraints the synthesis tool must know.

Non-Synthesizable Code

- Time - Printing - Files - Looping (Not as traditional for-loops) → pg 129 pg 130

Place & Route

Placing synthesized design & routing to physical components. Routing is the most time consuming.

Timing Errors

Setup and Hold Time: Setup is the time a flipflop's input must be stable before a clock edge in order for a flipflop reliably register data for output.

Hold time is for a flipflop holding a data reliable after the clock edge.

The unstable state is referred as 'metastable'.

Propagation Delay

Time for a signal to travel from src → dst.

$t_{clk(min)} = t_{su} + t_p$: min. clock required = setup time + worst propagation delay

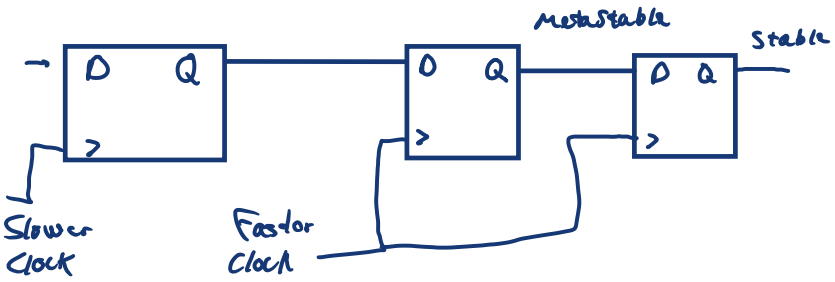
4 Fixing Timing Errors <pg 136>

- Slow down clock frequency
- Reduce propagation delay by breaking up logic into parts

Crossing Clock Domains

Slower → Faster

Double flop the data



Faster → Slower

Slightly difficult due to slower clock can miss the faster's pulse whilst changing.

Fixed by stretching faster's signals by a factor of 2 slower clock cycles.

* Using a FIFO

- write according to one clock
- read according to the other

These FIFOs are the FPGAs primitive, a dedicated component designed by manufacturer.

* Don't read from an empty & don't write to a full

On implementation — Use $\overline{AE}$, $\overline{AF}$ flags!

R8

The State Machine

The Finite state machine containing state, events, transitions, utilized as a controller.

Its implementation may be:

- Two Always Blocks
  - combinational always
  - sequential always
- One Always Block
  - combinational
  - sequential

R9 # Useful FPGA Primitives < Hard IP / Cores >

Refer to platform's vendor for Primitive instantiations, AMD's Artix → Libraries Guide

- **Primitive Creation**
  - Instantiation: Similar to using APIs.
  - GUI Approach: Using the dev. platform.

- **Block RAM**

They're sliced up into blocks on an FPGA.

Usually comes as a fixed sized block, e.g. a block of 16 Mb, if we only need 4 Mb, it'll still instantiate a 16 Mb block.

They've built-in error detection & correction, automatically.

- **Digital Signal Processing Block**

Used for performing math-based operations on signals within a digital system, which must be fast & parallel :).

Analog	Digital
Continuous signal representing Physical measurement.	Not continuous, consists of discrete measurements at individual points in time.
The information is always analog no conversion needed.	

ADCs are utilized to convert these signals, by sampling the analog at discrete points in time and convert to digital.

The discrete sampling time is referred as 'sampling rate' or 'sampling frequency'.

Filters: A system that enhances or reduces certain features of the input signals,

eg. A Low-Pass-Filter

Implementation: Recommended by inference using the synthesis tool & report.

However, for our case in an encryption engine, we'll create a primitive using the GUI.

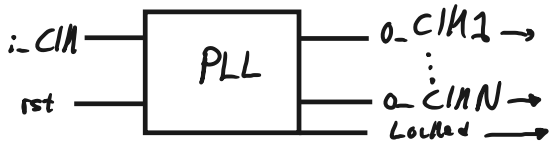
Phase-Locked Loop

The primitive used as the clock generator for the entire FPGA.
It allows us to change the frequency by modifying code — not swapping physical clock chip(s).

Frequency:

Makes it easy to have multiple clock domains in the same FPGA, since it can generate multiple clock frequencies simultaneously.

The input clock is an external, physical clock.



The frequencies vary by multiplying and dividing the input frequency, assigning to the desired clock as output.

Phase Shift: A signal's phase is its current position along the repeating waveform of the signal, measured as an angle from 0° to 360°.
<pg 200>

Shifting the phase of a clock signal moves the location of its rising edges.

Original Clock (NO PS)	+ 90°	+ 180°	+ 270°	+ 360°
Rising edge delayed by:	1/4	2/4	3/4	original rising edge position

- Phase Shifts can be negative, going "backwards in time".

- Locked tells downstream modules the PLL is operating & its "trustworthy". The locked signal is commonly used as a reset to other modules' relying on PLL clocks.

Creation: Recommended by GUI.

R10

Numbers and Math

Signal width in FPGAs: [9b, 15b, 23b, >b]

Unsigned vs Signed

Range: Unsigned $[0, 2^N - 1]$ Signed $[-2^{N-1}, 2^{N-1} - 1]$

* Reminder: Signed uses MSB as its negative sign, the rest are offsets.

Two's Complement

Another way of knowing the signed representation of a number, invert the bits then add 2: ~N+1 <for us humans>.

* Dynamic Sizing via Parameters most recommended

Type Conversion: Verilog is loosely type, no worries — UHDL is strongly typed, consult <pg 210>.

Blocking vs Non-Blocking Assignment

It's essential when to use which assignment, if we want to display a data, using a blocking assignment is wiser, especially in testbenches.

Performing Mathematical Operations

Addition:

Rule #1: The result should at least 1 bit bigger than the biggest input, before sign extension.

For unsigned values, sign extension means adding a '0' as the MSB.

UHDL requires resize() Function, to do manually.

Rule #2: Match input & output types.

Subtraction:

Rule 1: Use signed output in case a<b in a-b. Also at least 1-bit bigger for the output size.

⋮

Multiplication:

Rule #1: The output's bit width must be the size of the sum of the input's bit width, and an additional bit for sign extension.

Multiplication by Powers of 2:

Remember the shift operators ☺.

Division: The dreaded operation... Some tricks to manage it:

- Using powers of 2 (right shift).
- Using a precalculated table: Storing the results of possible inputs in a 2D array, with the inputs as the indexes.
- Using multiple clock cycles: Make a loop that see how many additions required to add up to a quotient (manually - division algorithm).

Working with Decimals:

CPE 2307

Floating-Point

Most electronic devices can handle these operations.

Allow to 'dynamically' change its precision. Dynamic radix.

Fixed-Point

Common in FPGAs,

Fixed radix (decimal separator).

Rule #1: Decimal's widths must match, for <+,-> operations.

Conversion can easily be done with shifts.

Rule #2: Multiplying doesn't require decimal point equality, just output width must be >= width of inputs sum.

For example U2.2 * U3.1 = U5.3 => 5+3 = 8,
which is just U4 + U4 = U8.

R11 Getting data with In/Out and SERDES

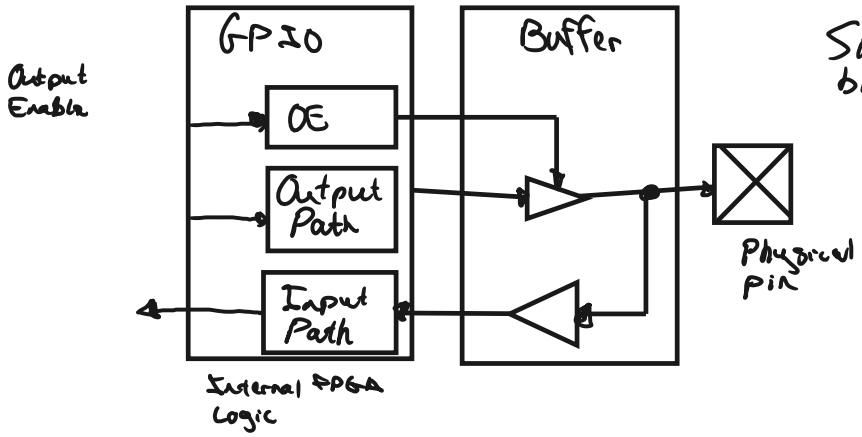
'Working with I/O is where being a "software person" and a "hardware person" lies.'

We must consider the details of the electrical signals such as operating voltage, are signals single-ended or differential?, using double data rate or a serializer/deserializer and such...

Working with GPIO Pins

These are general purpose input/output and can work either direction.

I/O Buffers: GPIO pins interfaces w/ electronic buffers, it allows us to reconfigure the pin's functionalities.



Simplified GPIO block diagram

When OE is low, the pin is configured as input, output buffer stops passing its input to its output.

The OE becomes high impendance (hi-Z or tri-state), it accepts little current. It means that the output buffer no longer affects the pin, it's entirely governed by incoming external signals.

Bidirectional Data for Half-Duplex Communications

Allows a pin to switch between I/O in the same design. It toggles the OE signal accordingly. The pin is a transceiver.

- A full-duplex allows simultaneous data transmission and receiving. This requires 2 transmission lines (pins), for sending and receiving.

To avoid data collision on a wire, devices agree on a <protocol>.

I^2C: Used as a half-duplex protocol. Uses 2 pins, clock and data.

Implementation:

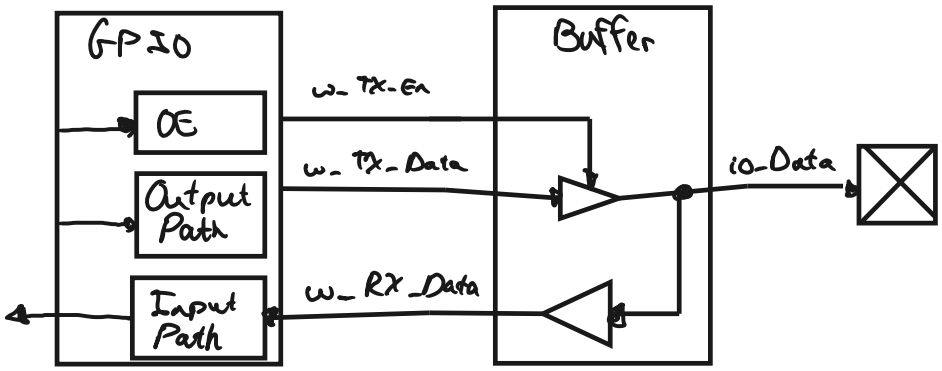
```
module bidirectional ( inout io_Data, ... );

        assign w_RX_Data = io_Data;
        assign io_Data   = w_TX_En ? w_TX_Data : 1'bZ;
    :
```

io_Data is to be mapped to a physical pin in the constraints file.

For input functionality, use assignment to drive w_RX_Data w/ data from pin.

For output, selectively toggle output buffer using w_TX_En.



* We'll implement code to ignore data driven out on w_TX_Data received on w_RX_Data of io_Data when w_TX_En.

<pg 242> Electrical Characteristics

Some of chars. we can specify to GPIO pins and electrical differences.

- Operating Voltage: Specifies what voltage the pin will be driven to for a logic 1 output and an expected volt. for a logic 1 input.

Most commonly, FPGA pins use a 0 V to represent a logic 0 and a 3.3 V as a 1. This standard is called LVCMOS33.

These standards are <single ended> I/O standards, in which the signals are referenced to ground.

- Differential standards are signals not referenced to ground.
- A bank is a group of pins operating on same voltage <VCCIO>.

- Drive Strength: Determines how much current (milliamps <mA>) can be driven in/out of pin. Allows to sink or source mA of current.

- Slew Rate: Change of rate for a signal's output, <slow/medium/fast>. Useful for interfacing with external components.

Differential Signaling: Signals 'grounded'/referenced to other signals. <fig. 11-3> These signals are more immune to <noise> or <electromagnetic interference (EMI)>.

<pg 246> Faster Data Transmission w/ Double Data Rate: With DDR, signals can be synchronized to rising and falling edges alike, allowing to send data twice in the same clock frequency!

We're to create double data rate output (ODDR) buffers anywhere we want to use DDR for data transfer.

Recommend to instantiate the buffers directly, ease of use.

We'll connect its output pin; .Q(Q) to our nets.

Serializer Deserializer (SerDes)

Is a primitive for some FPGAs allowing gigabit transmissions. It converts a parallel data stream onto a serial data stream, the receiving end converts serial to parallel. They're full-duplex.

Useful for networking w/ ethernet packets :).

Parallel vs Serial Communication

- Parallel comms. is using multiple comms. channels to send data, data split amongst multiple channels.
- Serial comms. sending data on a single channel.

Fixed parallel limitations by sending clock signal alongside data — serially as part of a single combined signal.

Self-Clocking Speeds: Here's where we combine clock & data into one signal. Embedding the clock in the data, these are encoding schemes.

- **Manchester Encoding Scheme:** Take the XOR of clock and data signals. Allows to send clock and data in a single wire.

The recipient simply use an XOR gate and some logic to retrieve the clock and data, this clock will serve as our flip-flop's clk input as well as the data to ff's data in.

How SerDes works: We've a serializer as transmitter and deserializer as receiver. A clock and data into Ser and out of Des.

It uses a Phase-Locked Loop as a clock input.

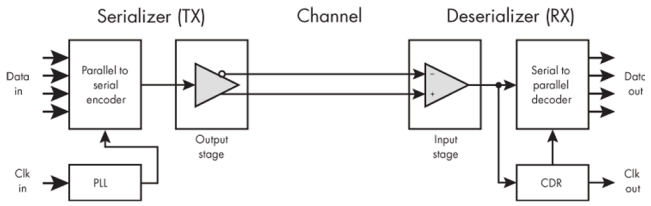


Figure 11-7: A simplified SerDes block diagram

The data in comes in parallel, it'll encode alongside the clock our data, then it'll send over to receiver, generated as a serialized data stream.

The output contains a Differential Output Buffer, maintaining DC balance. The CDR recovers the clock signal from data stream, the deserializer uses the extracted clock to sample the data.

Then at output, the data is again converted to parallel.