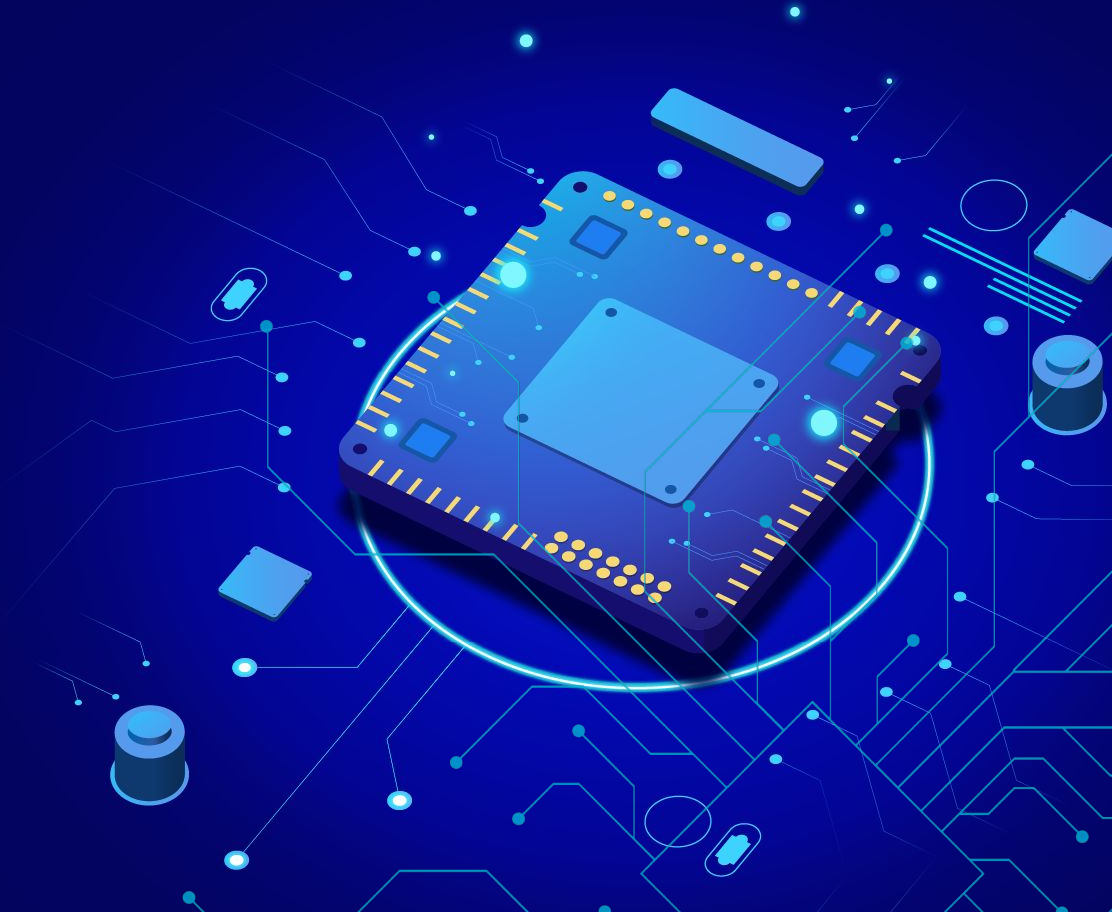
Guilherme Paulino
guilherme.paulino@pitec.co

FPGA Design Workshop

Oct 18, 2022



VI SEnC - USP São Carlos

About the Presenter

2012/13	Unicamp	telecom scholarship / OFDM TX
2014	SIU	ASIC course
2015	Unicamp	telecom/physics scholarship / Digital Picoammeter
2016	LNLS	White Rabbit sub-ns timing networks
2016	UVIC	miniBee
2017+	Pitec	FPGA and Detectors for Sirius (LNLS/CNPEM)
2022+	Unicamp	masters

Technology Park UNICAMP (4 floors)

X-Ray Systems (5y 40p)

Sirius (LNLS/CNPEM)
Brazilian 4th generation synchrotron light source

Outline

- Introduction to FPGAs
- What board do we have?
- Introduction to Hardware Description Language
- Combinational Logic and Finite State Machines
- Soft-Core Processors

Intro to FPGAs

FPGA Applications

Video & Image

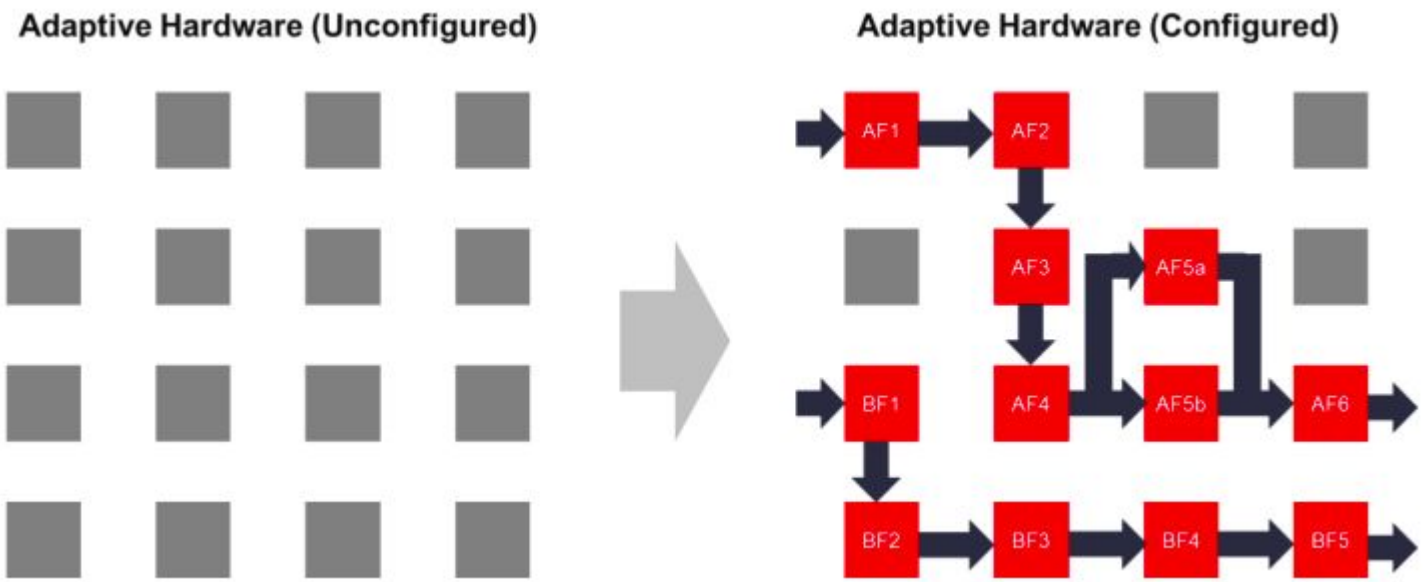
Aerospace & Defense

Server & Cloud

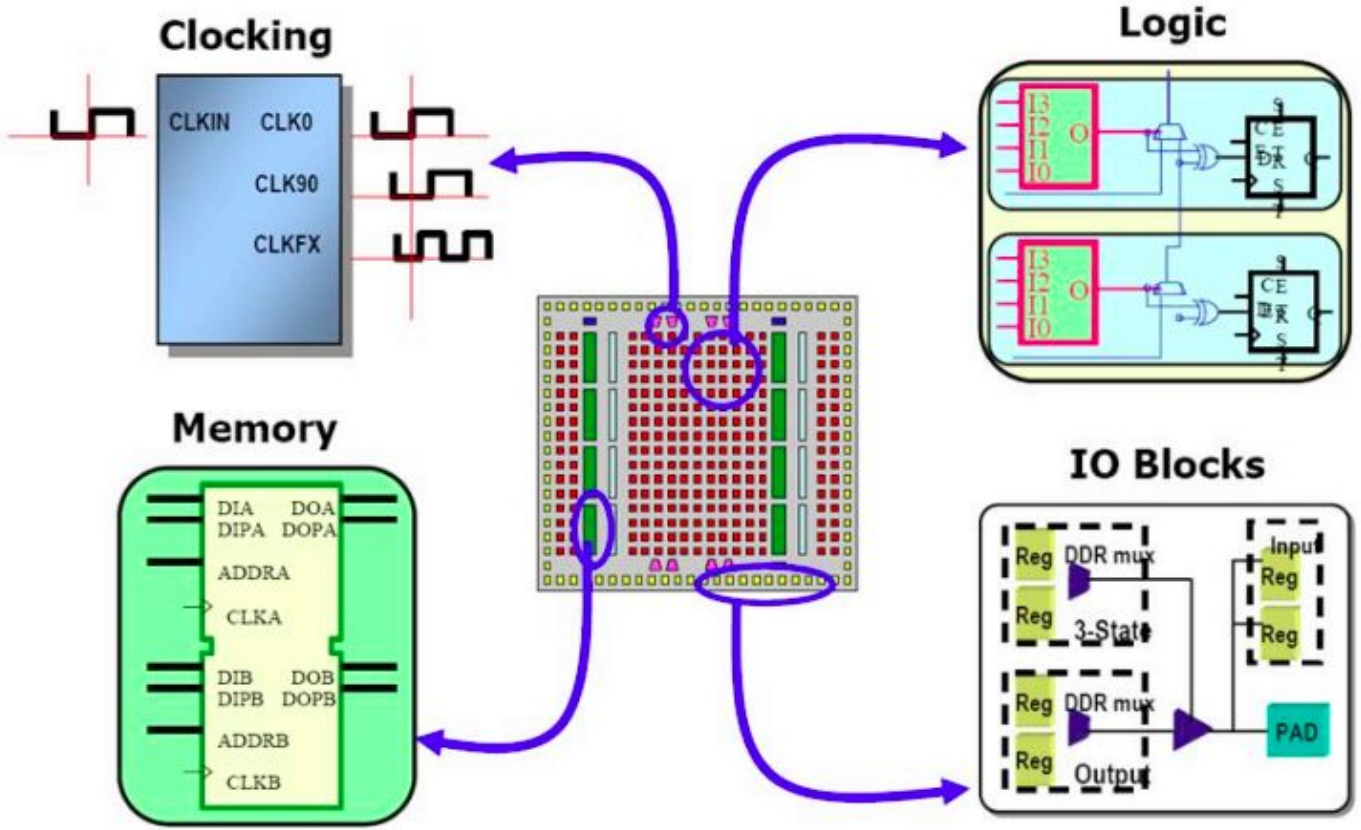
Medical & Scientific

Telecom & Datacom

Introducing Adaptive Computing



FPGA structure



Source: [Introduction to FPGA design]

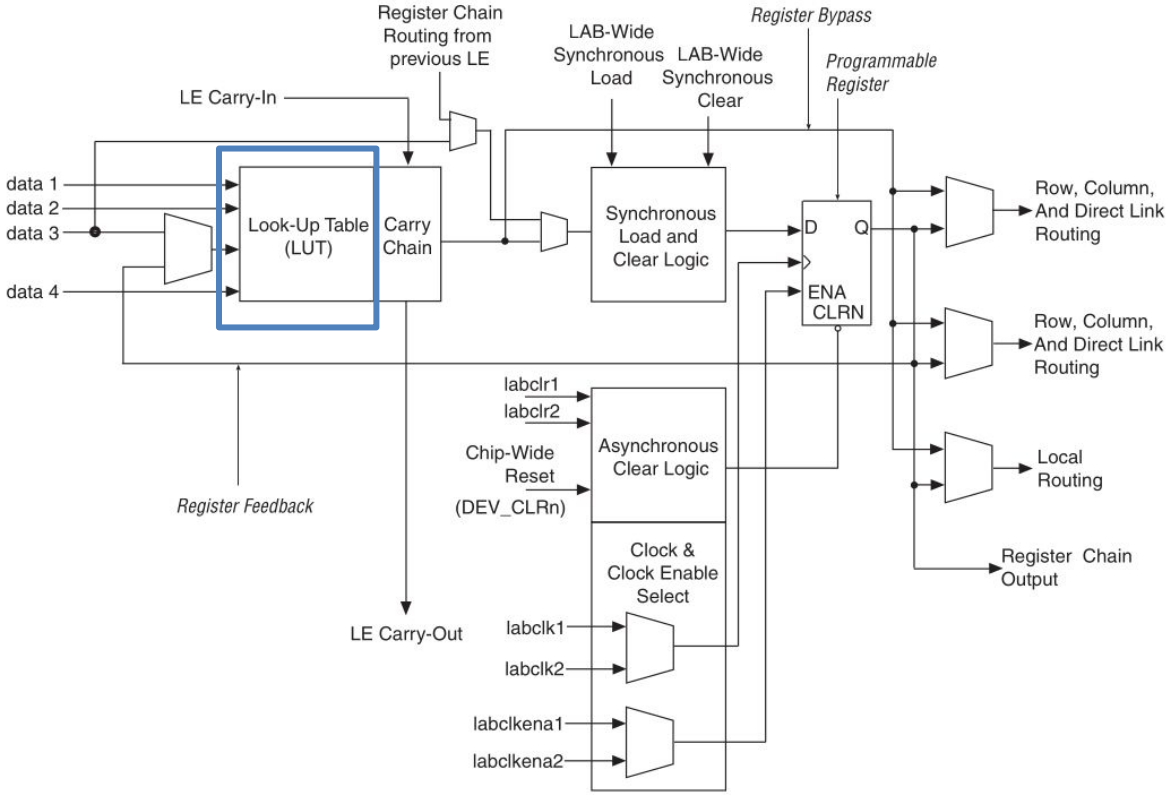
Cyclone core fabric

- **Logic Elements (LEs)**

- **M9K memory block** provides 9-Kbits of embedded SRAM
  - single port, simple dual port, or true dual port RAM, as well as FIFO buffers or ROM

- **Embedded multiplier blocks** can implement an 18×18 or two 9×9 multipliers in a single block
  - DSP IPs including finite impulse response (FIR), fast Fourier transform (FFT), and numerically controlled oscillator (NCO) functions for use with the multiplier blocks

Logic Elements

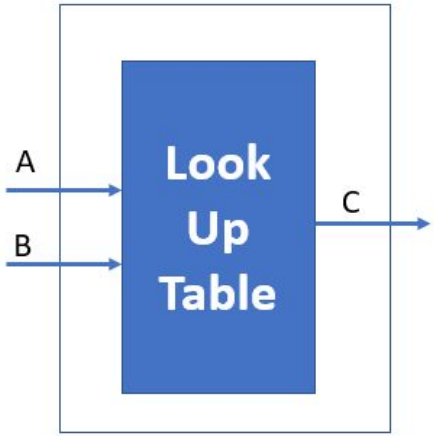


Cyclone IV Device LEs

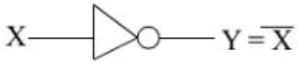
Each LE has the following features:

- A four-input look-up table (LUT), which can implement any function of four variables
- A programmable register
- A carry chain connection
- A register chain connection
- The ability to drive the following interconnects:
  - Local
  - Row
  - Column
  - Register chain
  - Direct link
- Register packing support
- Register feedback support

Look-Up Table (LUT)



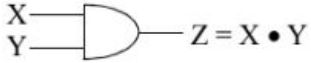
Inverter



X	Y
1	0
0	1

Truth Table of an inverter

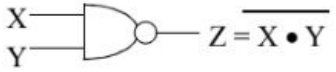
AND



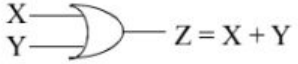
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table of an AND gate

NAND



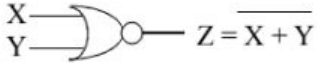
OR



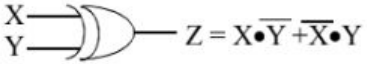
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table of an OR gate

NOR



XOR



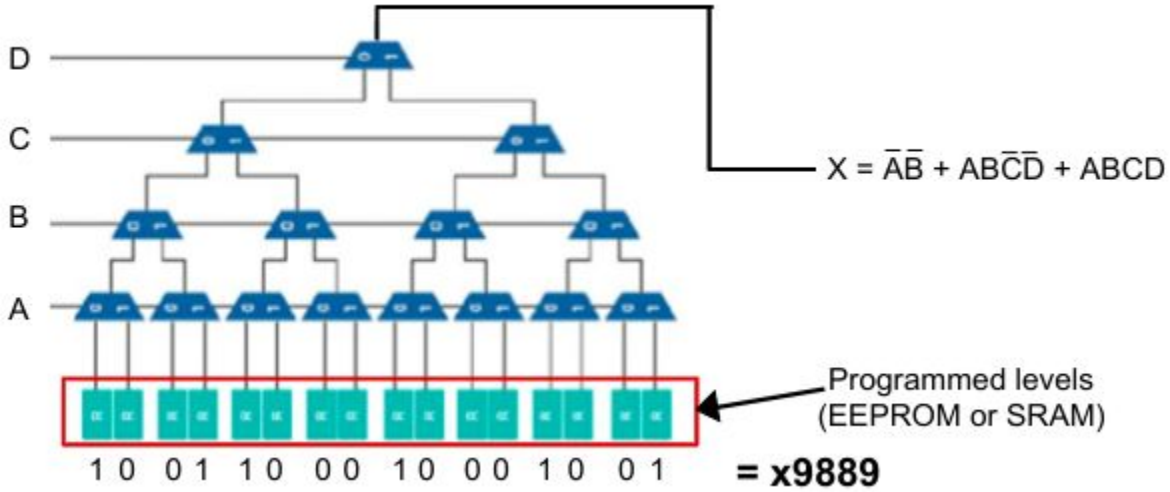
Z = X⊕Y

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

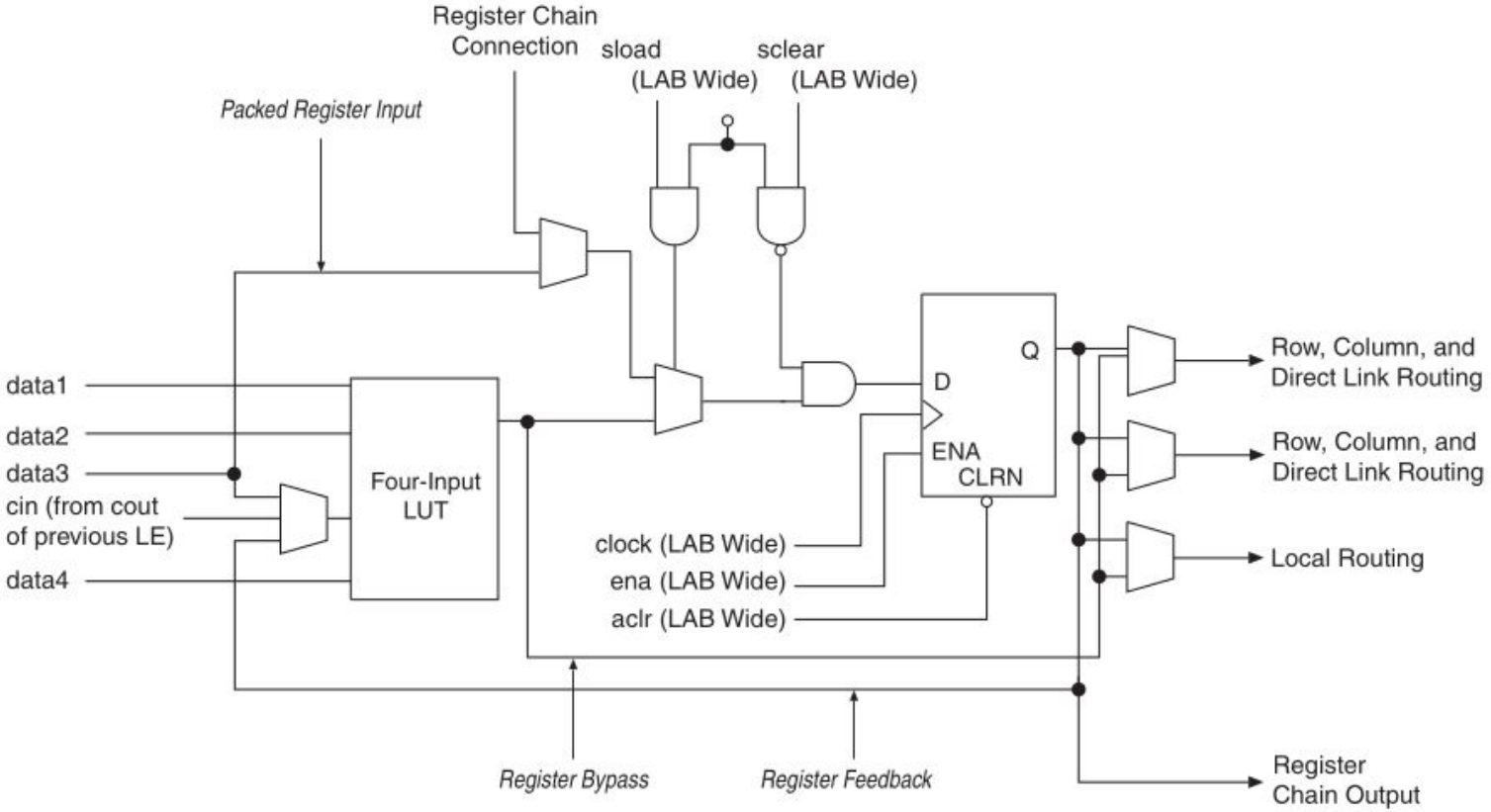
Truth Table of an XOR gate

Building a LUT

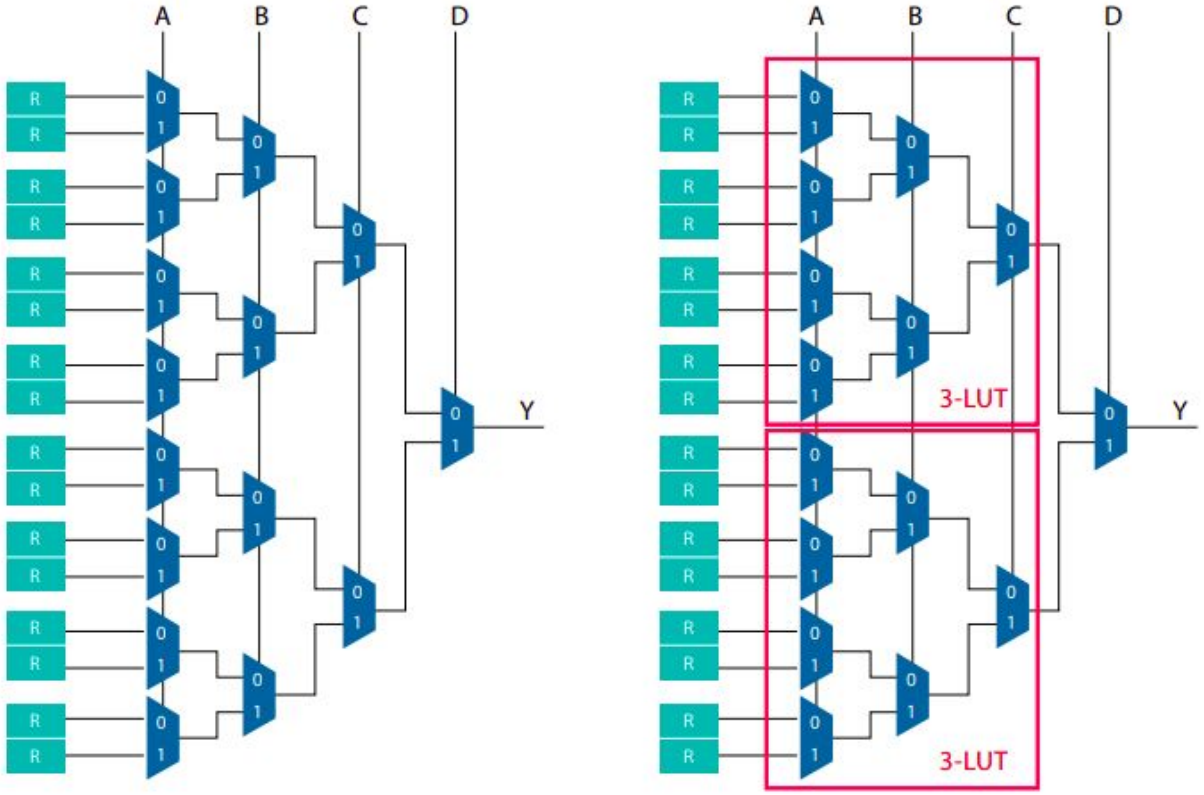
- Combinational functions created with Programmed “tables” (cascaded multiplexers)
- LUT inputs are mux select lines



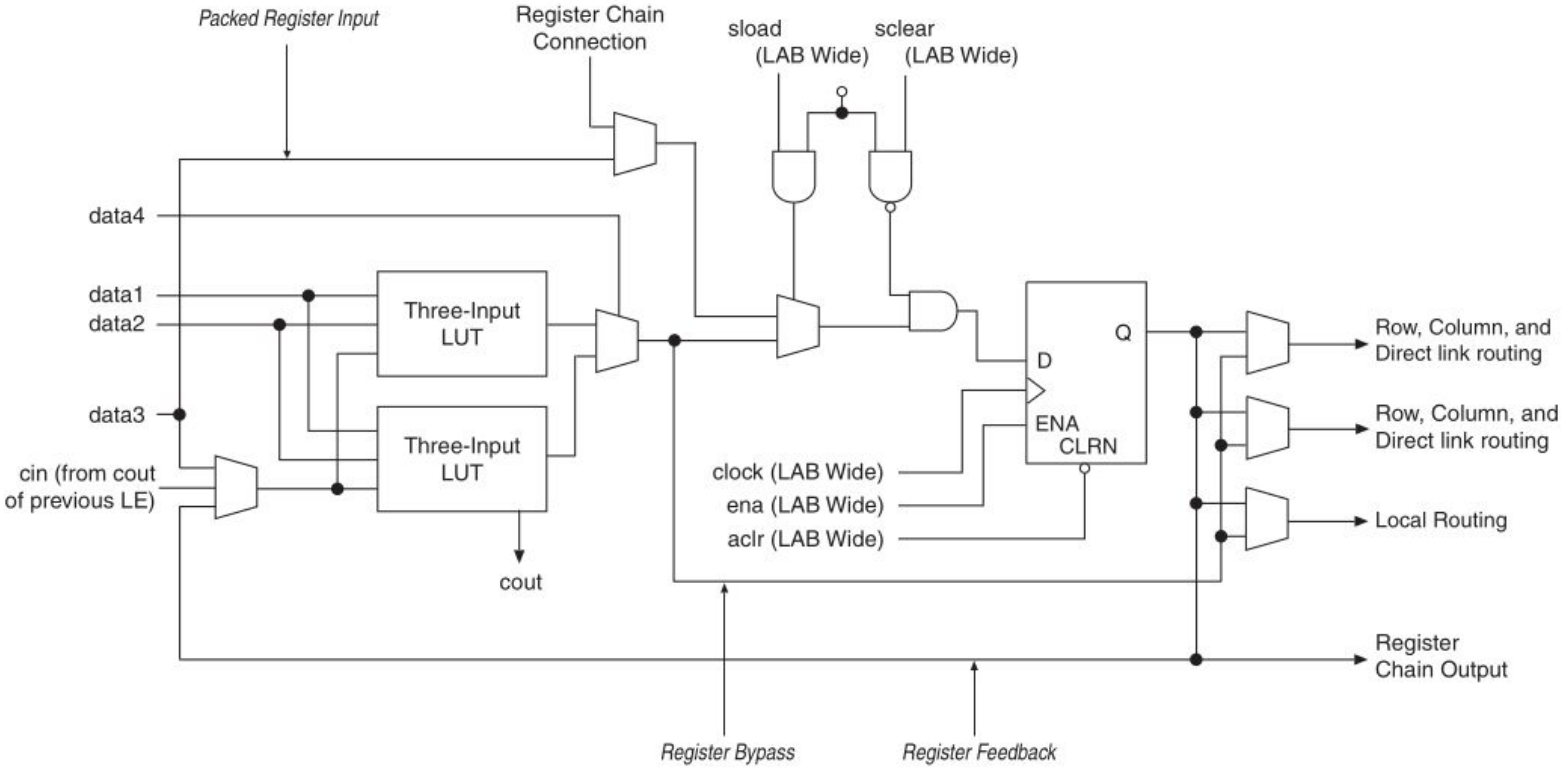
LE in Normal Mode



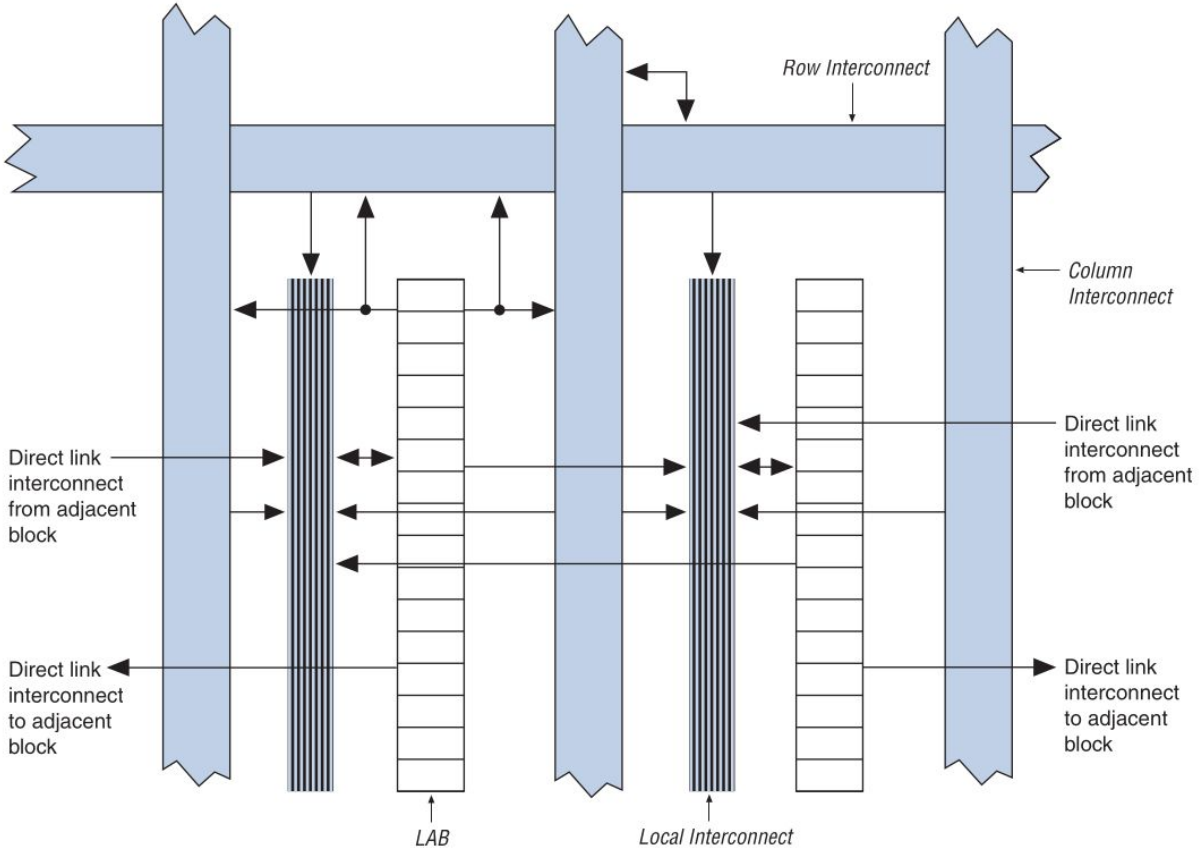
Building a LUT



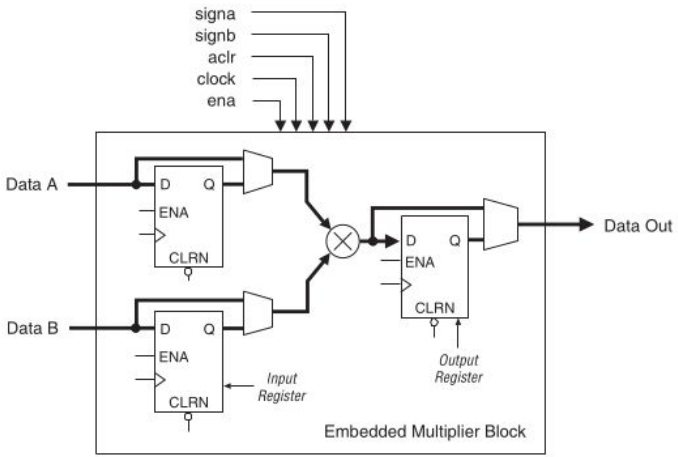
LE in Arithmetic Mode



Logic Array Blocks



Multipliers



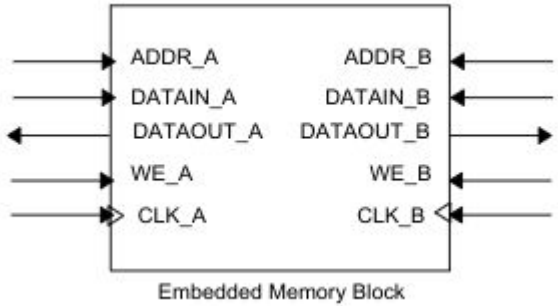
Useful for DSP functions

Table 4–1. Number of Embedded Multipliers in Cyclone IV Devices

Device Family	Device	Embedded Multipliers	9 × 9 Multipliers *(1)*	18 × 18 Multipliers *(1)*
Cyclone IV GX	EP4CGX15	0	0	0
	EP4CGX22	40	80	40
	EP4CGX30	80	160	80
	EP4CGX50	140	280	140
	EP4CGX75	198	396	198
	EP4CGX110	280	560	280
	EP4CGX150	360	720	360
Cyclone IV E	EP4CE6	15	30	15
	EP4CE10	23	46	23
	EP4CE15	56	112	56
	EP4CE22	66	132	66
	EP4CE30	66	132	66
	EP4CE40	116	232	116
	EP4CE55	154	308	154
	EP4CE75	200	400	200
	EP4CE115	266	532	266

M9K memory block

- 9-Kbits of embedded SRAM memory
  - Single/dual port RAM
  - ROM
  - Shift registers or FIFO buffers
- Initialize RAM or ROM contents on Power-on



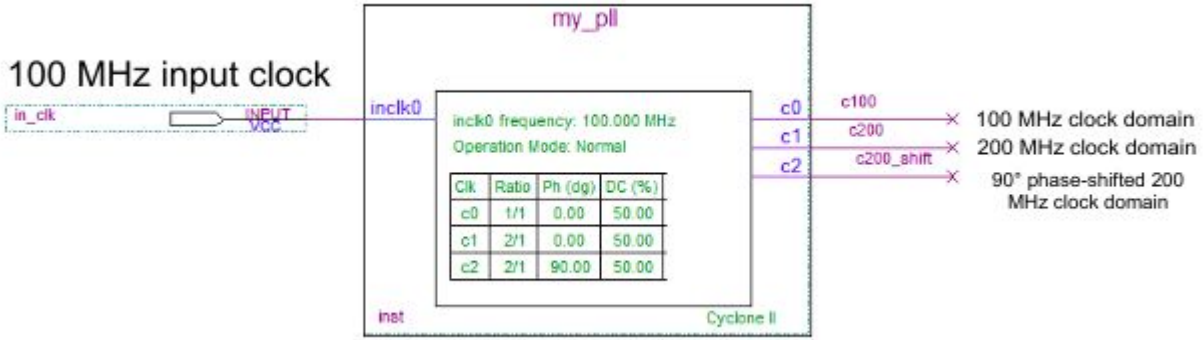
Embedded Memory Block

FPGA Clocking Structures

- Dedicated input clock pins
- Phase Locked Loops (PLLs)
- Delay Locked Loops (DLLs)
  - Dynamically phase-shift strobes for external memory interfaces
- Clock control blocks
  - Select clocks to feed clock routing network
  - Enable/disable clocks for power-up/down and power savings
- Clock routing network
  - Special routing channels reserved for clocks driven by PLLs or clock control blocks
  - Global clock network feeds entire device
  - Regional or hierarchical networks feed certain device areas

PLL

Based on input clock, programmable blocks that generate clock domains for use throughout device with minimal skew



Programmable I/Os

- Input/output/bidirectional
- Multiple I/O standards
- Differential signaling
- Current drive strength
- Slew rate
- On-chip termination/pull-up resistors
- Open drain/tri-state

Table 1–8. I/O Standards Support for the Cyclone IV Device Family

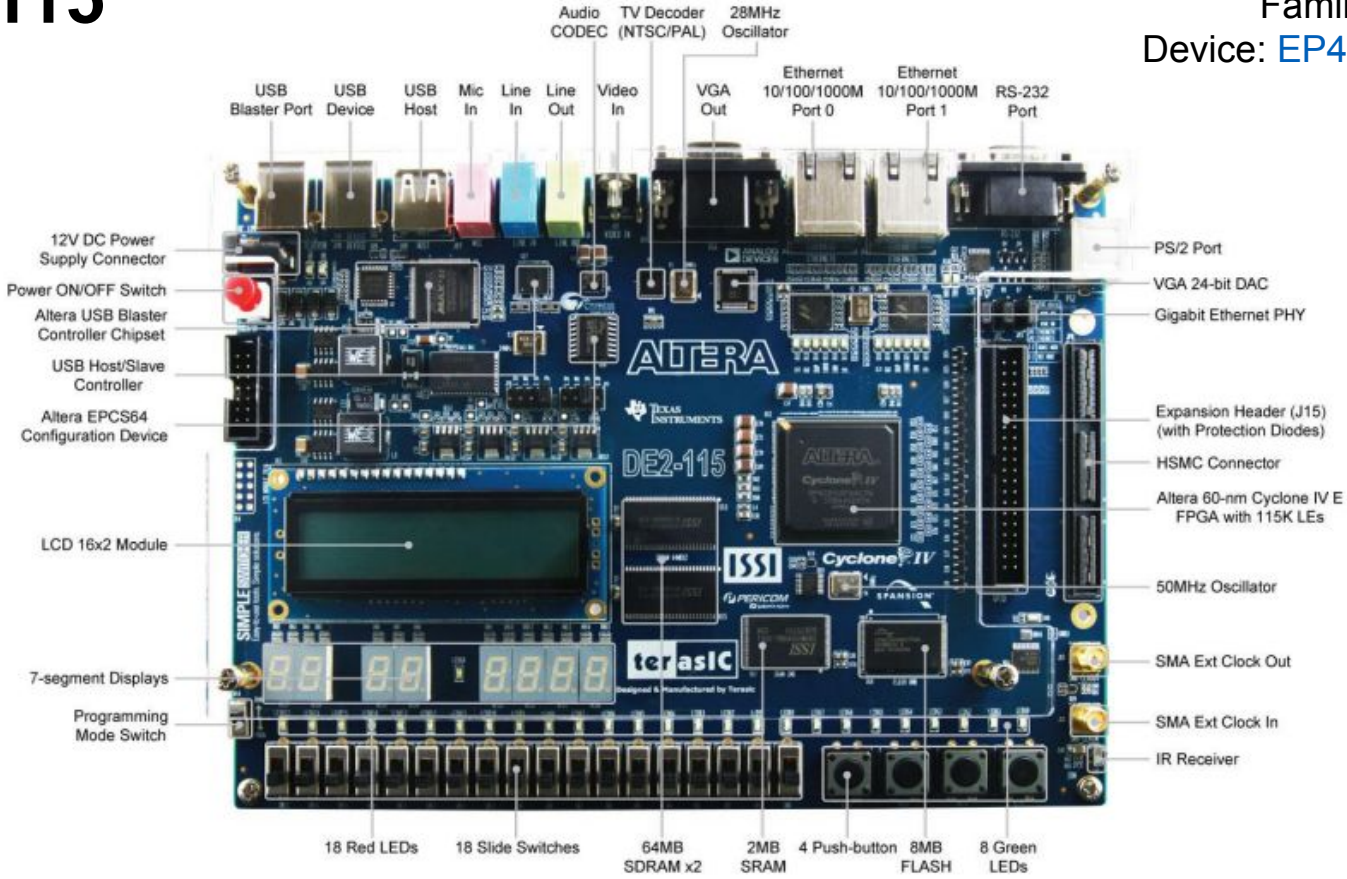
Type	I/O Standard
Single-Ended I/O	LVTTL, LVCMOS, SSTL, HSTL, PCI, and PCI-X
Differential I/O	SSTL, HSTL, LVPECL, BLVDS, LVDS, mini-LVDS, RSDS, and PPDS

The LVDS SERDES is implemented in the core of the device using logic elements.

What board do we have?

DE2-115

Family: Cyclone IV
Device: EP4CE115F29C7



Cyclone IV Device Family Features

The Cyclone IV device family offers the following features:

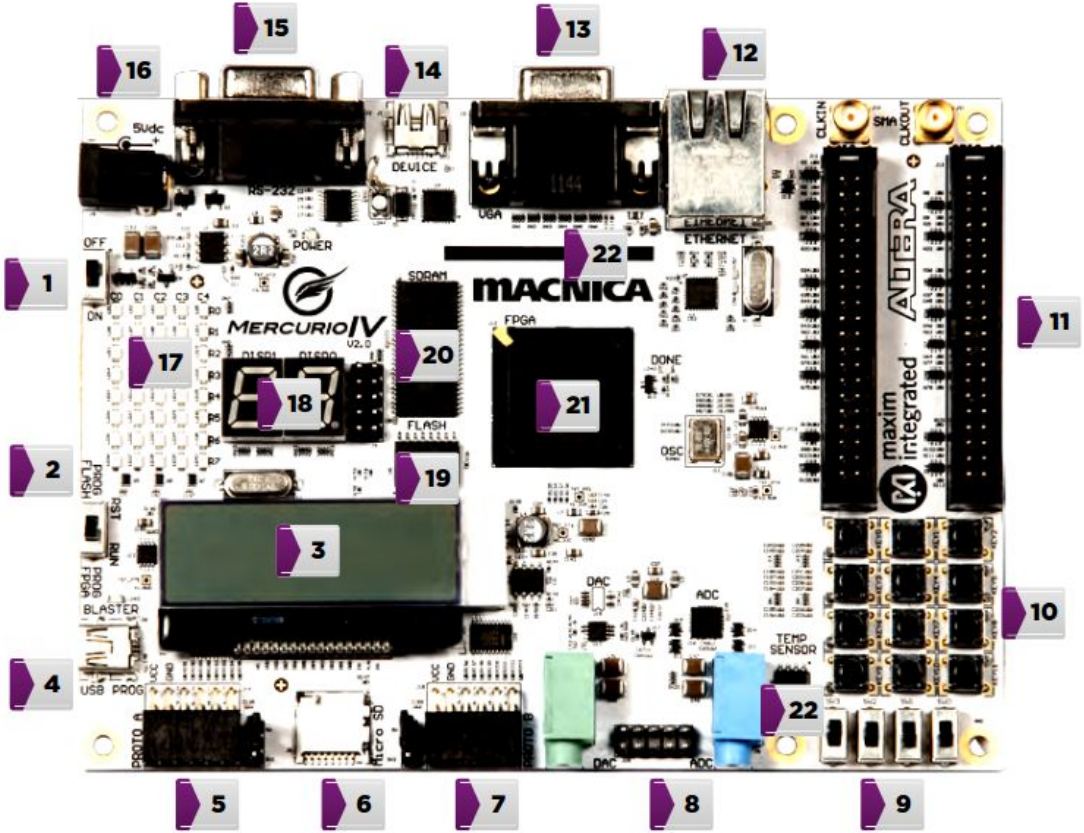
- Low-cost, low-power FPGA fabric:
  - 6K to 150K logic elements
  - Up to 6.3 Mb of embedded memory
  - Up to 360 18 × 18 multipliers for DSP processing intensive applications
  - Protocol bridging applications for under 1.5 W total power

Cyclone IV EP4CE115

Logic Elements	114,480
Embedded memory	3,888 Kbits
Embedded 18x18 multipliers	266
General-Purpose PLLs	4
Global Clock Networks	20
User I/O Banks	8
Maximum user I/O	528

Mercúrio IV

Family: Cyclone IV
Device: EP4CE30F23C7



Mercúrio IV


ALIMENTAÇÃO EXTERNA OU VIA USB
CLOCK 50MHZ
ENTRADA E SAÍDA DE CLOCK SMA
USB BLASTER
CONVERSOR AD COM 4 ENTRADAS
CONVERSOR DA COM 2 SAÍDAS
TECLADO 12 BOTÕES
CARTÃO MICROSD
SDRAM 512 MBITS
FLASH DE 64 MBITS CONFIGURAÇÃO + USUÁRIO
CYCLONE IV - 30K EP4CE30F23
LED RGB ALTO BRILHO
DISPLAY LED 7 SEGMENTOS
LCD ALFANUMÉRICO 2 X 16 COM BACKLIGHT
MATRIZ LED 8 LINHAS X 5 COLUNAS
4 CHAVES TIPO SWITCH
SENSOR TEMPERATURA
PORTA VGA
PORTA RS-232
PORTA ETHERNET
CONEXÃO USB NO MODO DEVICE
DUAS INTERFACES GPIO DE 32 PINOS IO E 4 PINOS DE CLOCK
DUAS INTERFACES PMOD® DE 8 PINOS IO

Cyclone IV EP4CE30

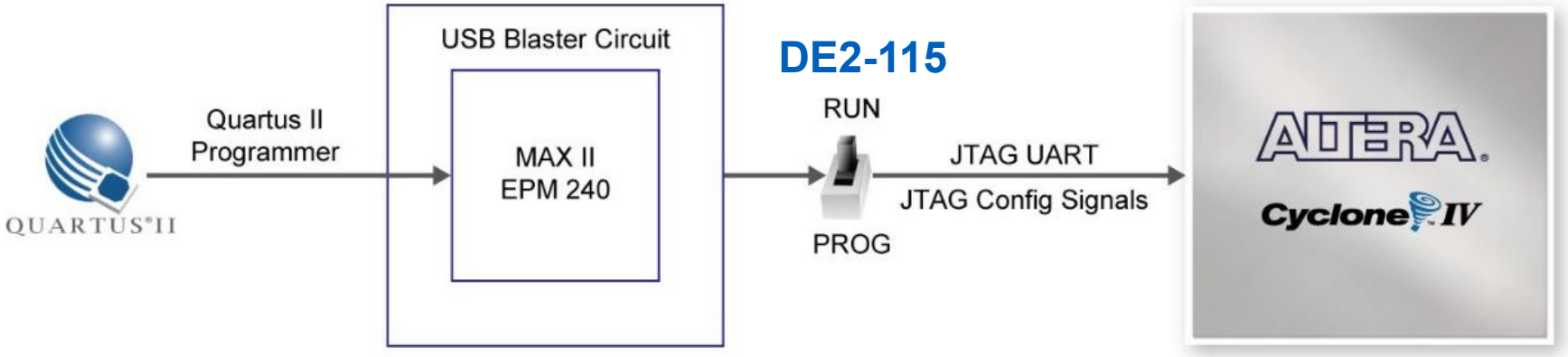
Logic Elements	28,848
Embedded memory	594 Kbits
Embedded 18x18 multipliers	66
General-Purpose PLLs	4
Global Clock Networks	20
User I/O Banks	8
Maximum user I/O	532

Lab Practice 0

Lab Practice 0

1. Open project on Quartus
    a. DE2_115_Lab_0_Programming
    b. Mercurio_IV_Lab_0_Programming
2. Compile
3. Program Device using JTAG

DE2-115 Configuring the FPGA in JTAG Mode



See:

- DE2-115 User manual - 4.1 Configuring the Cyclone IV E FPGA
- DIP Switch - **RUN**

- Manual Mercurio IV - 2.3.1 Configurando o FPGA
- DIP Switch - **PROG FPGA**

Mercurio IV

Lab Practice 0 - Questions

- Show the Synthesis resources usage?
- Show the Resource Utilization by Entity?
- How is the percentage status?
  - See Fitter summary
- What does the switch **SW**[0] do?

Intro to HDL

Part 1

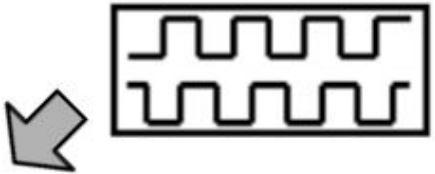
Overview

learning.intel.com/ Verilog HDL Basics

Logic Design workflow

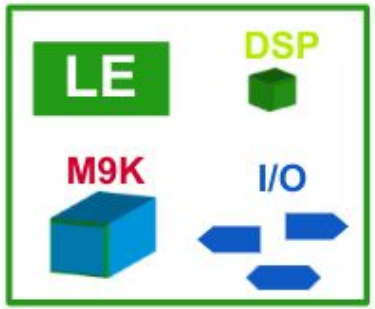
Design entry/RTL coding

- Behavioral or structural description of design
- Possibly with the help of high level tools



RTL functional simulation

- Mentor Graphics ModelSim* - Intel® FPGA Edition or other 3rd party simulators
- Verify logic model & data flow (no timing delays)



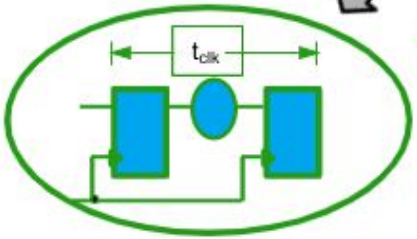
Synthesis (Mapping)

- Translate design into device specific primitives
- Optimization to meet required area & performance constraints
- Intel® Quartus® Prime Software synthesis or those available from 3rd party vendors
- *Result*: **Post-synthesis netlist**

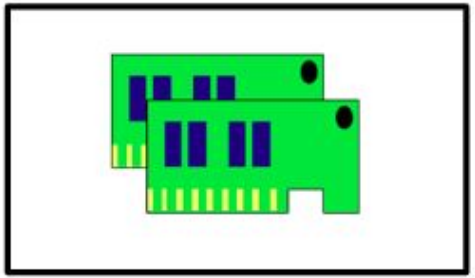
Logic Design workflow

Intel® Quartus® Prime
Design Software

Place & route (Fitting)
- Map primitives to specific locations inside target technology with reference to area & performance constraints
- Specify routing resources to be used
- *Result:* **Post-fit netlist**



Timing analysis
- Verify performance specifications were met
- Static timing analysis



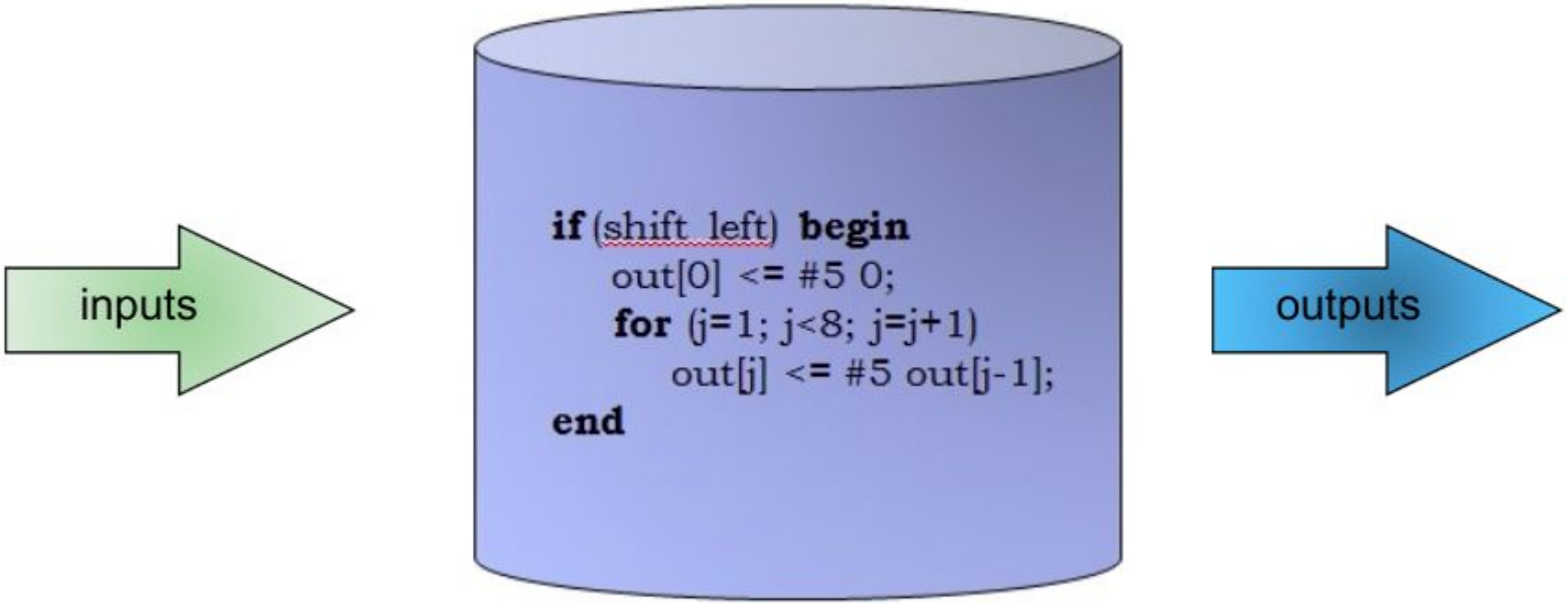
PC board simulation & test
- Simulate board design
- Program & test device on board
- Use on-chip tools for debugging

HDL Code

- HDL
  - Text based programming language that is used to model hardware
- Behavior Modeling
  - A component is described by its input/output response
- Structural Modeling:
  - A component is described by interconnecting lower-level components/primitives

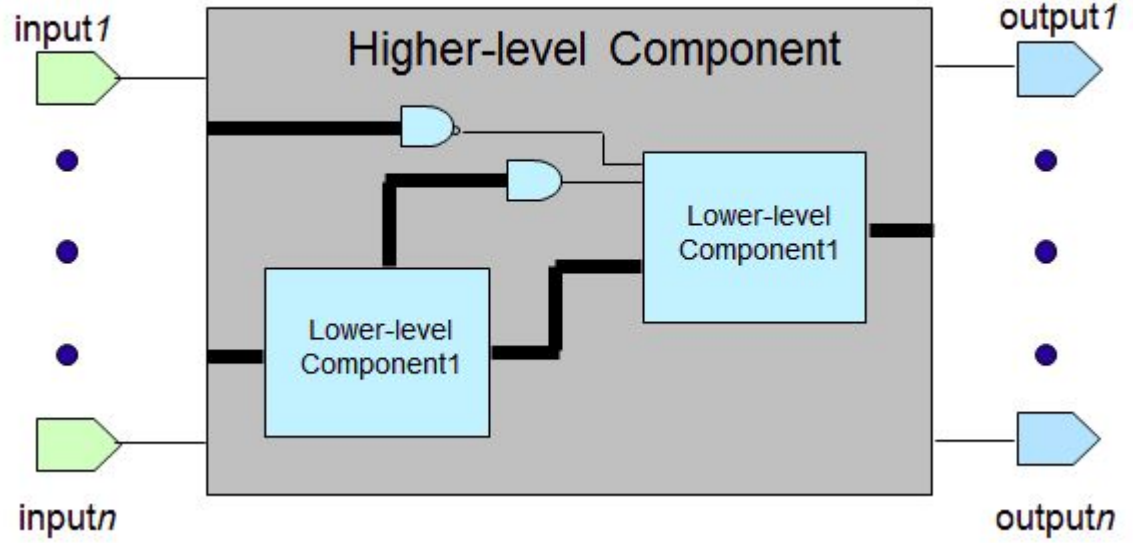
Behavior Modeling

Only the functionality of the circuit, no structure



Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware

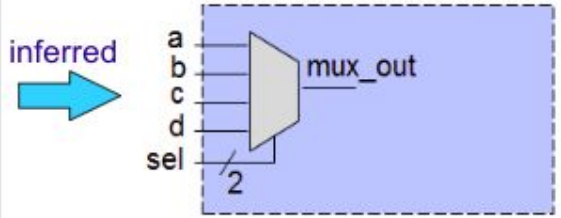


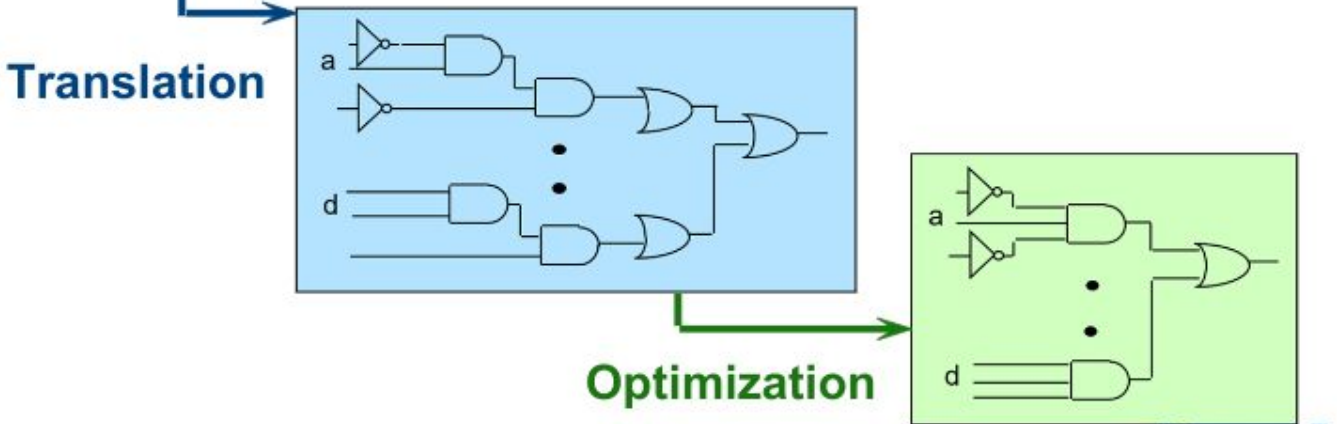
HDL Terminology

- Register Transfer Level (RTL)
  - A type of behavioral modeling, for the purpose of synthesis
    - Hardware is implied or inferred
    - Synthesizable
- Synthesis
  - Translading HDL to a circuit and then optimizing the represented circuit
- RTL Synthesis
  - Translating a RTL model of hardware into an optimized technology specific gate level implementation

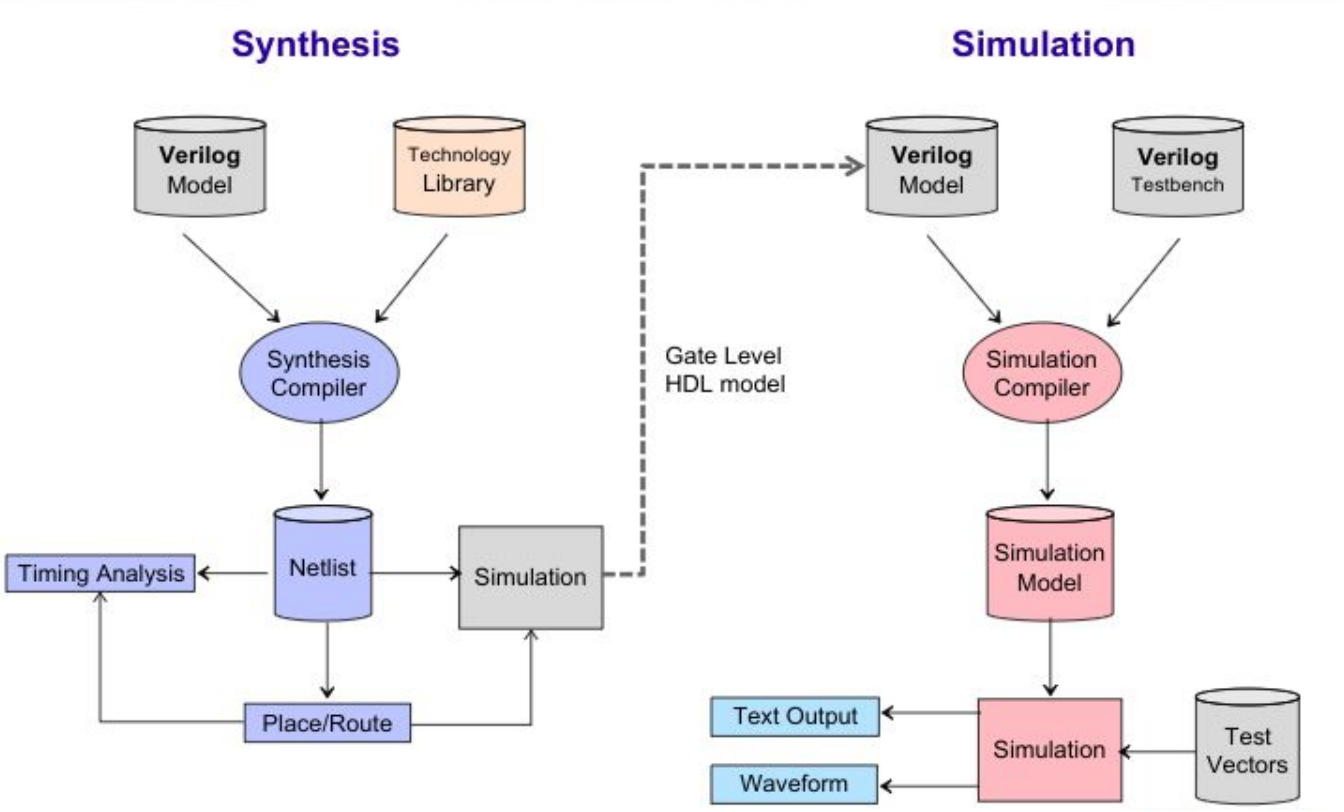
RTL Synthesis

```
always @(a, b, c, d, sel)
    case (sel)
        2'b00: mux_out = a;
        2b'01: mux_out = b;
        2b'10: mux_out = c;
        2'b11: mux_out = d;
    endcase
```





RTL Synthesis & Simulation



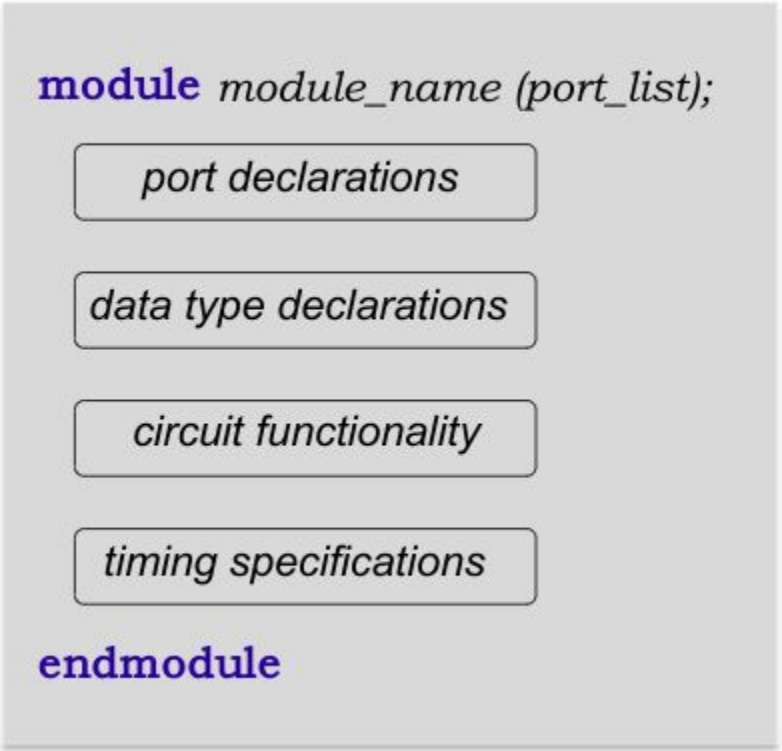
Intro to HDL

Part 2

Verilog Basics

learning.intel.com/ Verilog HDL Basics

Verilog - Modeling Structure

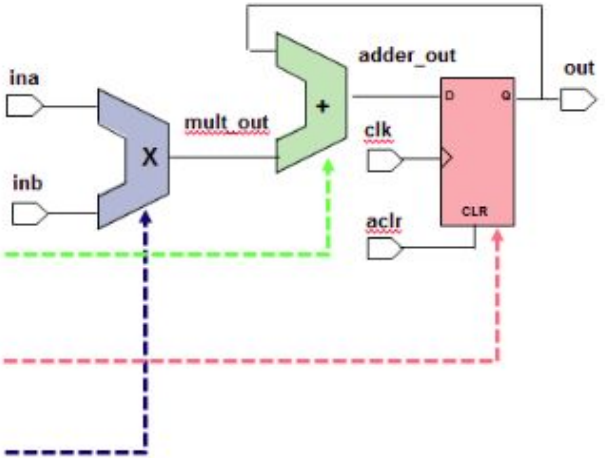


- Begins with keyword **module** & ends with keyword **endmodule**
- Case-sensitve
- All keywords are lowercase
- Whitespace is used for readability
- Semicolon is the statement terminator
- // : Single line comment
- /* */ : Multi-line comment
- Timing specification is for simulation (not discussed)

Verilog Model - Example

```
module mult_acc (out, ina, inb, clk, aclr);

input [7:0] ina, inb;
input clk, aclr;
output [15:0] out;

wire [15:0] mult_out, adder_out;
reg [15:0] out;

assign adder_out = mult_out + out;

always @ (posedge clk or posedge aclr)
        if (aclr)    out = 16'h0000;
        else         out = adder_out;

multa u1(.in_a(ina), .in_b(inb),
                     .m_out(mult_out));

endmodule
```

Ports and data types



Module and Port Declaration

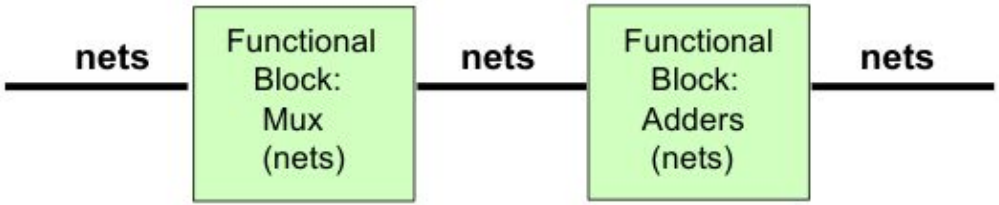
- **Module Declaration:**
  - Begins with keyword **module**
  - Provides module name
  - Includes port list, if any
- **Port Types:**
  - **input** ⇒ input port
  - **output** ⇒ output port
  - **inout** ⇒ bidirectional port
- **Port Declarations:**

<port_type> <port_name>;

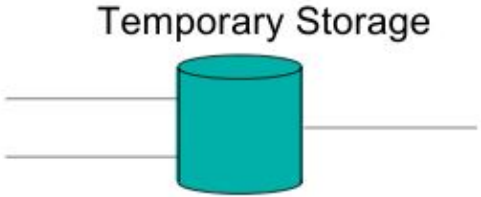
```
module mult_acc (out,
                 ina, inb,
                 clk, aclr);
input [7:0] ina, inb;
input clk, aclr;
output [15:0] out;

…

endmodule
```

Data Types

- **Net data type - represents physical interconnect between structures (activity flows)**



- **Variable data type - represents element to store data temporarily**



Net Data Type

Type	Definition
wire	Represents a node or connection
tri	Represents a tri-state node
supply0 supply1	Logic 0 Logic1

- **Bus Declarations:**

  <data_type> [*MSB* : *LSB*] <*signal name*>;
  <data_type> [*LSB* : *MSB*] <*signal name*>;

- **Examples:**
  - **wire** [7 : 0] out ;
  - **tri** enable;

Variable Data Types

- **Variables can be any one of the following:**
  - **reg** - unsigned variable of any bit size
    - Use **reg signed** for signed implementation
  - **integer** : signed 32-bit variable
  - **real, time, realtime** : no synthesis support
- **Can be assigned only within a procedure, a task or a function**
- **Bus Declarations:**

  reg [*MSB* : *LSB*] <*signal name*>;

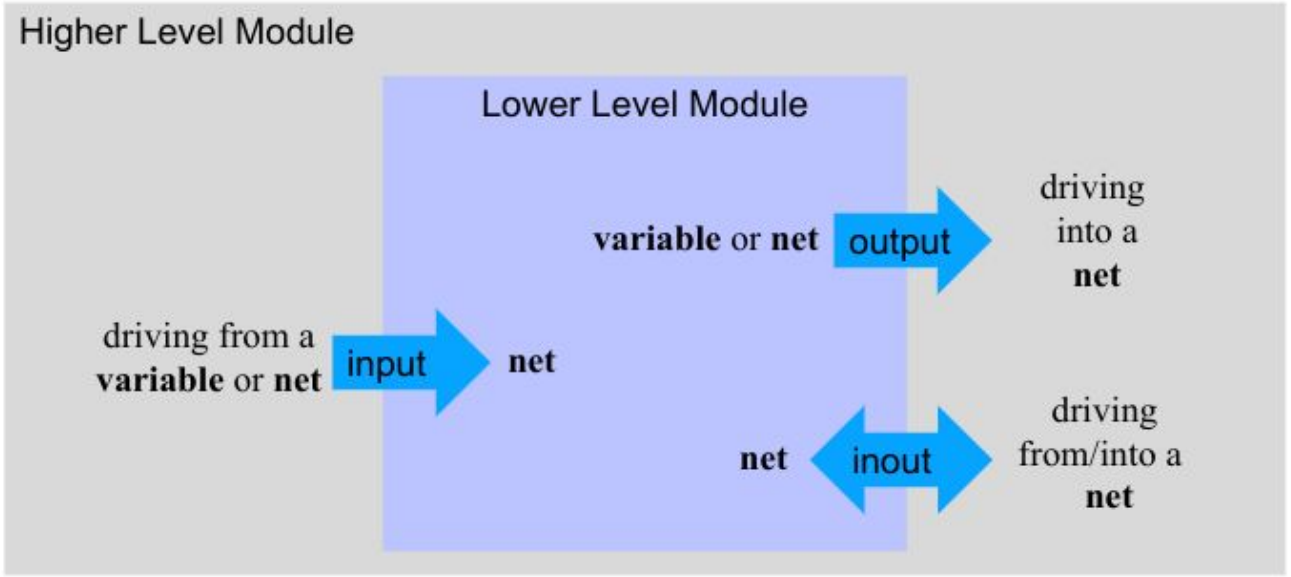
  reg [*LSB* : *MSB*] <*signal name*>;
- **Examples:**

  reg [7 : 0] out ;

  integer count;

Port Connection Rules

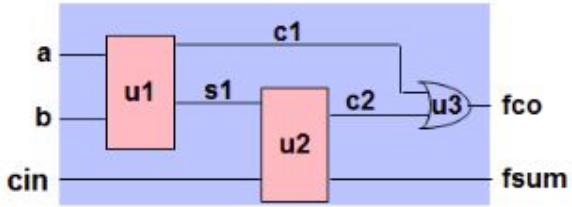
- **The diagram shows the type requirements for port connection when modules are instantiated**



Connecting Module Instantiations

- **Two methods to define port connections**
  - By ordered list
  - By name
- **By ordered list (1st half adder*)**
  - Port connections defined by the order of the port list in the lower-level module declaration
    - `module half_adder (co, sum, a, b);`
  - Order of the port connections does matter
    - co -> c1, sum -> s1, a -> a, b -> b
- **By name (2nd half_adder*)**
  - Port connections defined by name
  - Recommended method
  - Order of the port connections does not matter
    - a -> s1, b -> cin, sum -> fsum, co ->c2

```
module full_adder (
    output fco, fsum,
    input cin, a, b
);
    wire c1, s1, c2;

    half_adder  u1 (c1, s1, a, b);
    half_adder  u2 (.a(s1),  .b(cin),
        .sum(fsum), .co(c2));
    or  u3(fco, c1, c2);

endmodule
```

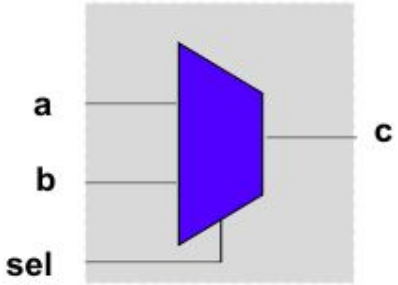


RTL Processes

- **Combinatorial Process**
  - Sensitive to all inputs used in the combinatorial logic

```
always @ (a, b, sel)
always @ *
```

Sensitivity list includes all inputs used In the combinatorial logic

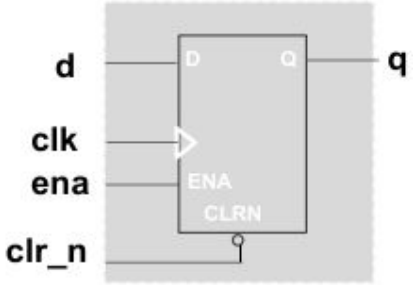


** is a Verilog shortcut to manually having to add all inputs*

- **Clocked Process**
  - Sensitive to a clock or/and control signals

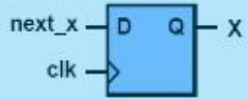
```
always @(posedge clk, negedge clr_n)
```

Sensitivity list does not include the **d** *or* **ena** *inputs, only the clock and asynchronous control signals*

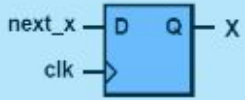


Blocking vs. Non-Blocking Assignments

```
always @( posedge clk )
begin
  x = next_x;
end
```

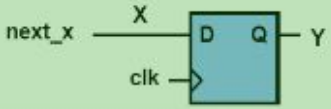



```
always @( posedge clk )
begin
  x <= next_x;
end
```

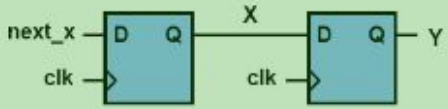


Same Behavior

```
always @( posedge clk )
begin
  x = next_x;
  y = x;
end
```




```
always @( posedge clk )
begin
  x <= next_x;
  y <= x;
end
```



Different Behavior

Best Practices:

Blocking operator (=) for combinational logic

Non-Blocking operator (<=) for sequential logic

Testbench Code

```
module clk_gen
    #(parameter period = 50)
(
    output reg clk
);

    initial clk = 1'b0;

    always
        #(period/2) clk = ~clk;

    initial #100  $finish;

endmodule
```

Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	$finish;

Lab Practice 1

Lab Practice 1 - Questions

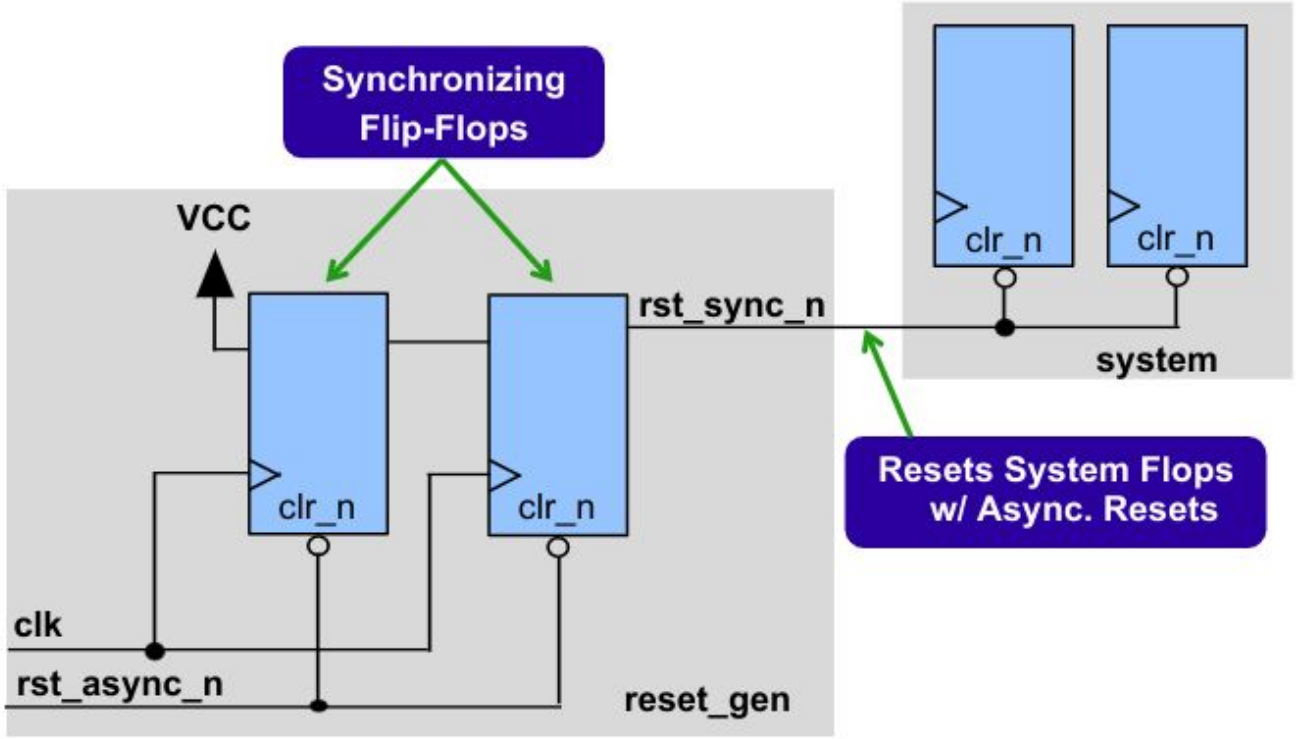
- See the resource utilization by Entity.
- See the resources usage percentage status.
  - See Fitter Resource Usage summary
- Logic Elements by mode?
  - Normal or Arithmetic
- Show the RTL?
  - Expand the Counter module
- What are the types of design optimization for a programmable logic device?

Finite State Machines

State Machine Guidelines

- Assign default values to outputs derived from the state machine
- Separate the state machine logic from
  - arithmetic functions
  - data paths
  - output values
- If your design contains an operation that is used by more than one state, define the operation outside the state machine then use the value in the output logic of the state machine

State Machine Guidelines - Resets



Reset Example in SystemVerilog

```
module reset_gen (
    output rst_sync_n,
    input  clk, rst_async_n);

logic rst_s1, rst_s2;

always_ff @ (posedge clk, negedge rst_async_n)
    if (~rst_async_n) begin
        rst_s1 <= 1'b0;
        rst_s2 <= 1'b0;
        end
    else begin
        rst_s1 <= 1'b1;
        rst_s2 <= rst_s1;
        end
assign rst_sync_n = rst_s2;
endmodule
```

State Machine in SystemVerilog

```
enum logic [2:0] {IDLE, SOP, DATA_PYLD, CRC, EOP} pckt_state,
    nxt_pckt_state;

always_ff @ (posedge clk, posedge rst)
    if (rst)
        pckt_state <= IDLE;
    else
        pckt_state <= nxt_pckt_state;

always_comb begin
    nxt_pckt_state = pckt_state;
    unique case (pckt_state)
        IDLE      : if (pkt_rdy)      nxt_pckt_state = SOP;
        SOP       :                   nxt_pckt_state = DATA_PYLD;
        DATA_PYLD : if (end_of_data)  nxt_pckt_state = CRC;
        CRC       : if (crc_done)     nxt_pckt_state = EOP;
        EOP       :                   nxt_pckt_state = IDLE;
    endcase
end
// output assignments follow
assign sop_state = (pckt_state == SOP);
```

Lab Practice 2

Lab Practice 2 - Questions

- See the resource utilization summary.
- See the RTL.
- Show the State Machine Viewer.

Soft-Core Processors

Nios II Processor

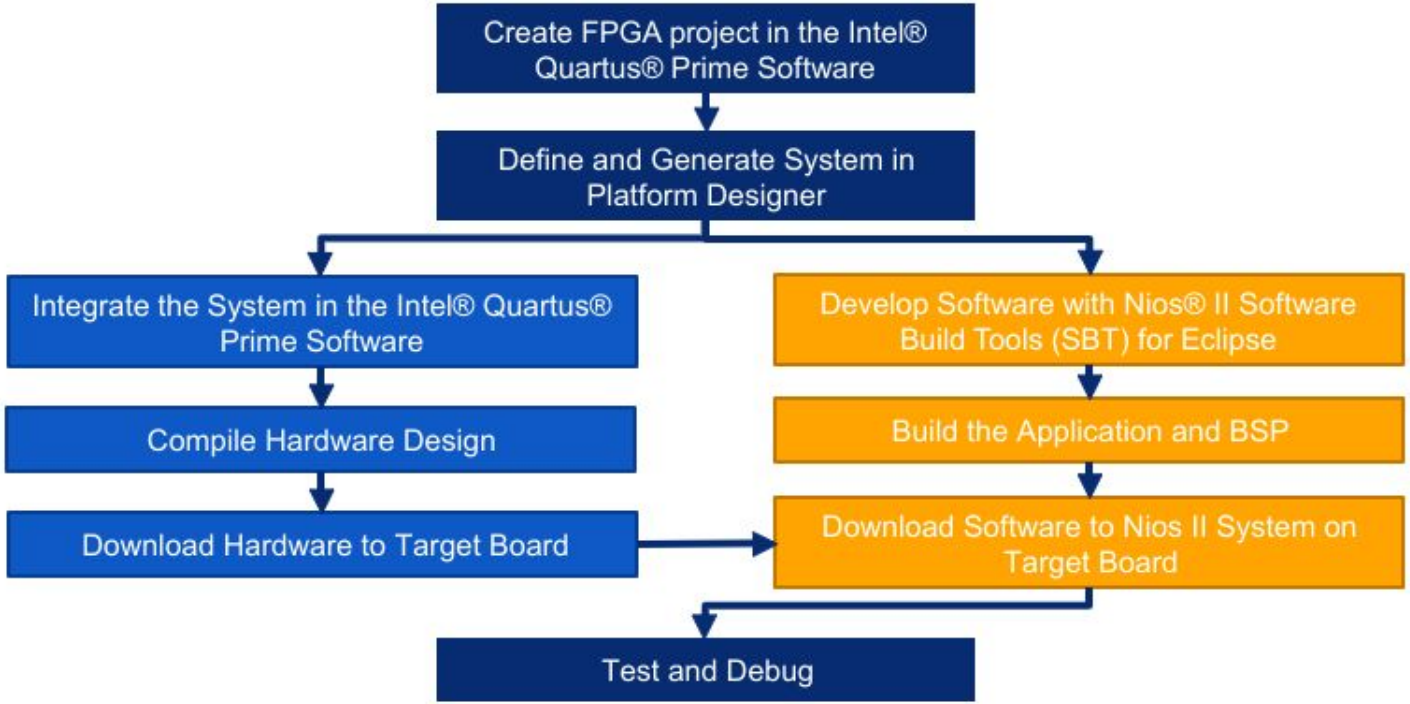
Intel® FPGAs second generation 32-Bit RISC soft microprocessor

- Royalty-free, can be targeted for all Intel® FPGAs
- Harvard architecture
- Processor and all peripherals written in VHDL or Verilog
- Can be synthesized into all modern Intel® FPGA devices using the Intel® Quartus® Prime software

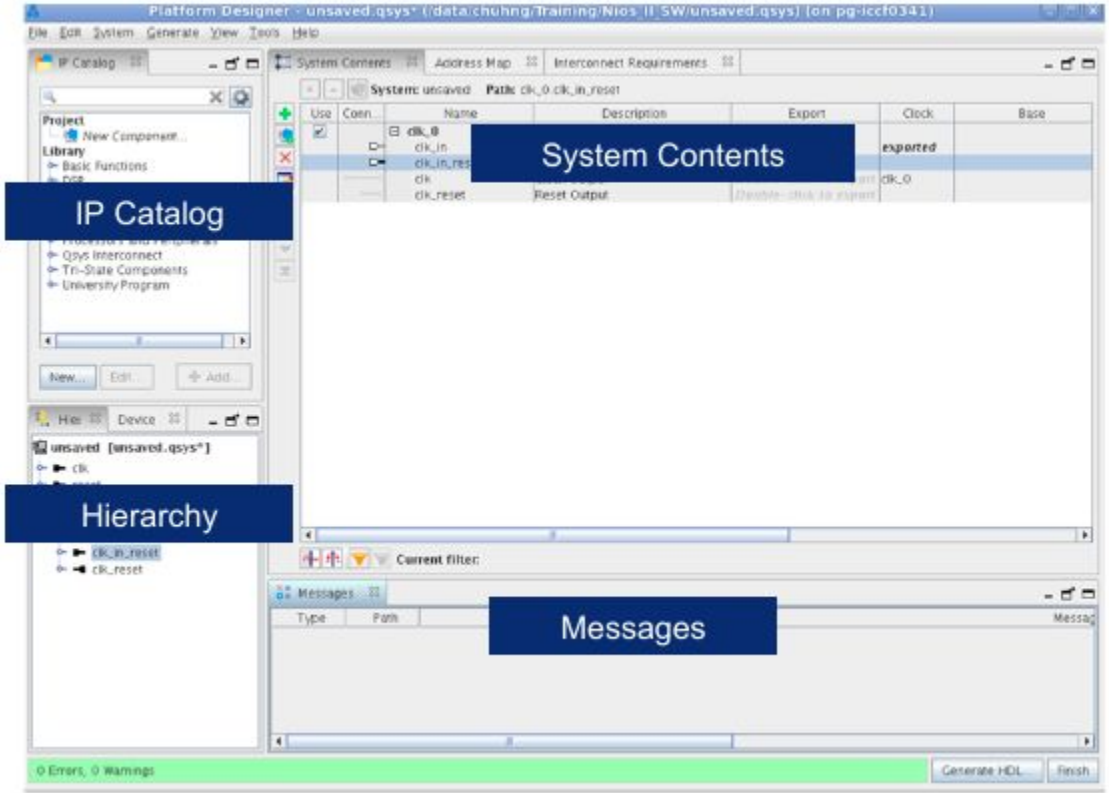
Two ISA compatible variants

- Nios II/f core (Fast): Optimized for performance
- Nios II/e core (Economy): Optimized for size

Nios II System Design workflow



Platform Designer



Platform Designer - System Generation

Generates interconnect structure to connect components together

- Avalon®-MM (Memory Mapped) interfaces
- Avalon®-ST (Streaming) interfaces
- Arm* AMBA* interfaces

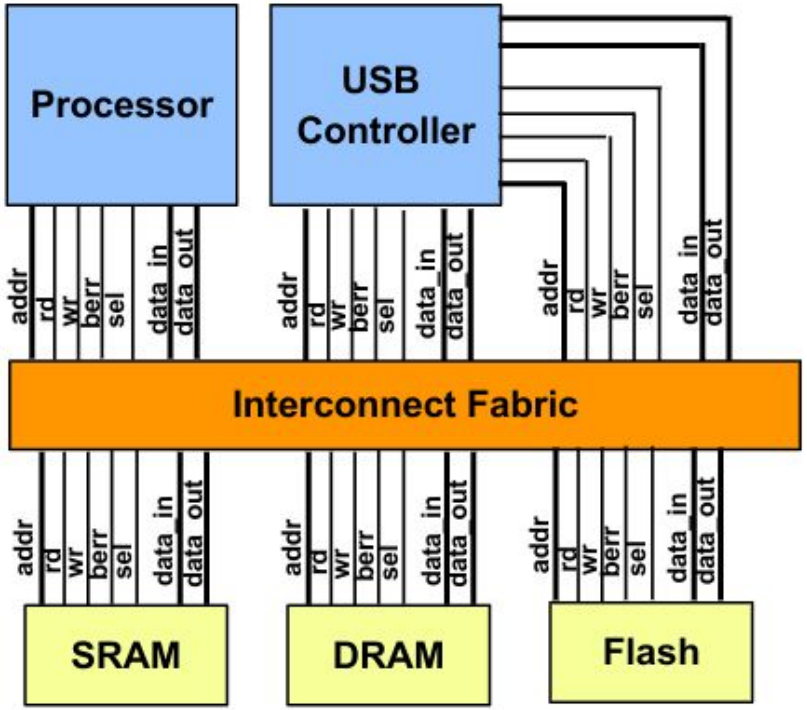
Generates synthesizable files of the hardware system

- Compiled with the Intel® Quartus® Prime software

Generates .sopcinfo file for software development

- XML file describing the system hardware

Verilog Bus Module Connections



First step - Program FPGA Logic!

Intel® Quartus® Prime software compilation generates FPGA configuration bitstream

- File formats (.sof, .pof, etc.)

Used to configure the FPGA

FPGA must be configured with the Nios® II processor system before you can run software

- Many configuration methods
  - JTAG
  - Serial or quad-serial configuration device
  - From flash - use CPLD or external processor as controller

Lab Practice 3

Lab Practice 3 - Questions

- See the resource utilization summary.
- See the RTL.
- Show the State Machine Viewer.

References

References

1. Intel Developer Training
   a. Basics of Programmable Logic: FPGA Architecture
   b. Verilog HDL Basics
   c. The Nios® II Processor: Introduction to Developing Software

2. Terasic Altera DE2-115 Development and Education Board

3. Macnica Mercurio IV Kit

4. Cornell ECE 5760 Advanced Microcontroller Design and system-on-chip

guilherme.paulino@pitec.co

Extra Slides

Verilog HDL - Basics

learning.intel.com/ Verilog HDL Basics

Parameters

- Value assigned to a symbolic name
- Must resolve to a constant at compile time
- Can be overwritten at compile time
- localparam – same as parameter but cannot be overwritten

```
parameter size = 8;
localparam outsize = 16;
reg [size-1:0] dataa, datab;
reg [outsize-1:0] out
```

- Verilog-2001 style, include with module declaration

```
module mult_acc
    #(parameter size = 8)
    (…);
```

Assigning Values - Numbers

- **Are sized or unsized: <size>'<base format><number>**
  - **Sized** example: `3'b010` = 3-bit wide binary number
    - The prefix (3) indicates the size of number
  - **Unsized** example: `123` = 32-bit wide decimal number by default
    - **Defaults**
      - No specified <base format> defaults to **decimal**
      - No specified <size> defaults to **32-bit** wide number

- **Base Formats**
  - Decimal ('d or 'D) `16'd255` = 16-bit wide decimal number
  - Hexadecimal ('h or 'H) `8'h9a` = 8-bit wide hexadecimal number
  - Binary ('b or 'B) `'b1010` = 32-bit wide binary numer
  - Octal ('o or 'O) `'o21` = 32-bit wide octal number
  - Signed ('s' or 'S') `16'shFA` = signed 16-bit hex value

Numbers

- **Negative numbers - specified by putting a minus sign before the <size>**
  - **Legal**: **-8'd3** = 8-bit negative number stored as 2's complement of 3
  - **Illegal**: **4'd-2** = **ERROR!!**

- **Special Number Characters**
  - '_' (underscore): used for readability
    - Example: **32'h21_65_bc_fe** = 32-bit hexadecimal number
  - 'x' or 'X' (unknown value)
    - Example: **12'h12x** = 12-bit hexadecimal number; LSBs unknown
  - 'z' or 'Z' (high impedance value)
    - Example: **1'bz** = 1-bit high impedance number

Arithmetic Operators

Operator Symbol	Functionality	Examples ain = 5 ; bin = 10 ; cin = 2'b01 ; din = 2'b0z		
+	Add, Positive	bin + cin ⇒ 11	+bin ⇒ 10	ain + din ⇒ x
–	Subtract, Negate	bin – cin ⇒ 9	-bin ⇒ -10	ain – din ⇒ x
*	Multiply	ain * bin ⇒ 50		
/	Divide	bin / ain ⇒ 2		
%	Modulus	bin % ain ⇒ 0		
**	Exponent*	ain ** 2 ⇒ 25		

- **Treats vectors as a whole value**
- **Results unknown if any operand is Z or X**
- **Carry bit(s) handled automatically if result wider than operands**
- **Carry bit lost if operands and results are same size**

Check synthesis tool for support

Bitwise Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
~	Invert each bit	~ain ⇒ 3b'010	~cin ⇒ 3'b10x
&	**AND** each bit	ain & bin ⇒ 3'b100	bin & cin ⇒ 3'b010
|	**OR** each bit	ain | bin ⇒ 3'b111	bin | cin ⇒ 3'b11x
^	**XOR** each bit	ain ^ bin ⇒ 3'b011	bin ^ cin ⇒ 3'b10x
^~ or ~^	**XNOR** each bit	ain ^~ bin ⇒ 3'b100	bin ~^ cin ⇒ 3'b01x

- **Operates on each bit or bit pairing of the operand(s)**
- **Result is the size of the largest operand**
- **X or Z are both considered unknown in operands, but result maybe a known value**
- **Operands are left-extended if sizes are different**

Reduction Operators

Operator Symbol	Functionality	Examples ain = 4'b1010 ; bin = 4'b10xz ; cin = 4'b111z		
&	**AND** all bits	&ain ⇒ 1'b0	&bin ⇒ 1'b0	&cin ⇒ 1'bx
~&	**NAND** all bits	~&ain ⇒ 1'b1	~&bin ⇒ 1'b1	~&cin ⇒ 1'bx
|	**OR** all bits	|ain ⇒ 1'b1	|bin ⇒ 1'b1	|cin ⇒ 1'b1
~|	**NOR** all bits	~|ain ⇒ 1'b0	~|bin ⇒ 1'b0	~|cin ⇒ 1'b0
^	**XOR** all bits	^ain ⇒ 1'b0	^bin ⇒ 1'bx	^cin ⇒ 1'bx
^~ or ~^	**XNOR** all bits	~^ain ⇒ 1'b1	~^bin ⇒ 1'bx	~^cin ⇒ 1'bx

- **Reduces a vector to a single bit value**
- **X or Z are both considered unknown in operands, but result maybe a known value**

Relational Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
>	Greater than	ain > bin ⇒ 1'b0	bin > cin ⇒ 1'bx
<	Less than	ain < bin ⇒ 1'b1	bin < cin ⇒ 1'bx
>=	Greater than or equal to	ain >= bin ⇒ 1'b0	bin >= cin ⇒ 1'bx
<=	Less than or equal to	ain <= bin ⇒ 1'b1	bin <= cin ⇒ 1'bx

- **Used to compare values**
- **Returns a 1 bit scalar value of Boolean true (1) / false (0)**
- **X or Z are both considered unknown in operands and result is always unknown**

Equality Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
==	Equality	ain == bin ⇒ 1'b0	cin == cin ⇒ 1'bx
!=	Inequality	ain != bin ⇒ 1'b1	cin != cin ⇒ 1'bx
===	Case equality	ain === bin ⇒ 1'b0	cin === cin ⇒ 1'b1
!==	Case inequality	ain !== bin ⇒ 1'b1	cin !== cin ⇒ 1'b0

- **Used to compare values**
- **Returns a 1 bit scalar value of Boolean true (1) / false (0)**
- **For equality/inequality, X or Z are both considered unknown in operands and result is always unknown**
- **For case equality/case inequality, X or Z are both considered distinct values and operands must match completely**

Logical Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b000 ; cin = 3'b01x		
!	Expression not true	!ain ⇒ 1'b0	!bin ⇒ 1'b1	!cin ⇒ 1'bx
&&	**AND** of two expressions	ain && bin ⇒ 1'b0	bin && cin ⇒ 1'bx	
||	**OR** of two expressions	ain || bin ⇒ 1'b1	bin || cin ⇒ 1'bx	

- **Used to evaluate single expression or compare multiple expressions**
  - Each operand is considered a single expression
  - Expressions with a zero value are viewed as false (0)
  - Expressions with a non-zero value are viewed as true (1)
- **Returns a 1 bit scalar value of Boolean true (1) / false (0)**
- **X or Z are both considered unknown in operands and result is always unknown**

Shift Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b01x	
<<	Logical shift left	ain << 2 ⇒ 3'b100	bin << 2 ⇒ 3'bx00
>>	Logical shift right	ain >> 2 ⇒ 3'b001	bin >> 2 ⇒ 3'b000
<<<	Arithmetic shift left	ain <<< 2 ⇒ 3'b100	bin <<< 2 ⇒ 3'bx00
>>>	Arithmetic shift right	ain >>> 2 ⇒ 3'b111 (signed)	bin >>> 2 ⇒3'b000 (signed)

- **Shifts a vector left or right some defined number of bits**
- **Left shifts (logical or arithmetic): Vacated positions always filled with zero**
- **Right shifts**
  - Logical: Vacated positions always filled with zero
  - Arithmetic (unsigned): Vacated positions filled with zero
  - Arithmetic (signed): Vacated position filled with sign bit value (MSB value)
- **Shifted bits are lost**
- **Shifts by values of X or Z (right operand) return unknown**

Miscellaneous Operators

Operator Symbol	Functionality	Format & Examples
?:	Conditional test	(condition) ? true_value : false_value sig_out = (sel == 2'b01) ? a : b
{ }	Concatenate	ain = 3'b010 ; bin = 3'110 {ain,bin} ⇒ 6'b010110
{ { } }	Replicate	{3 {3'b101} } ⇒ 9'b101101101

Operators Precedence

Operator(s)	Priority
+ - ! ~ & ~& etc. (unary* operators)	High
**	
* / %	
+ - (binary operators)	
<< >> <<< >>>	
< > <= >=	
== != === !==	
& (binary operator)	
^ ~^ ^~ (binary operators)	
| (binary operator)	
&&	
||	
?:	
{} {{}}	Low

- ( ) used to override default and provide clarity

** Unary operators have only one operand*

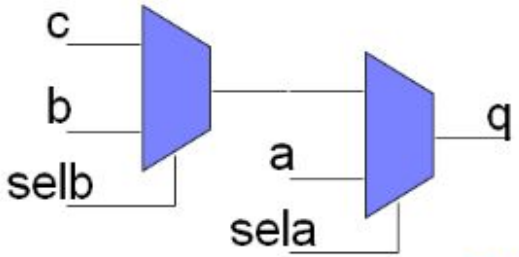
If-else Statements

- Format:

```
if <condition1>
    {sequence of statement(s)}
else if <condition2>
    {sequence of statement(s)}
        ...
else
    {sequence of statement(s)}
```

- Example:

```
always @ * begin
    if (sela)
        q = a;
    else if (selb)
        q = b;
    else
        q = c;
end
```



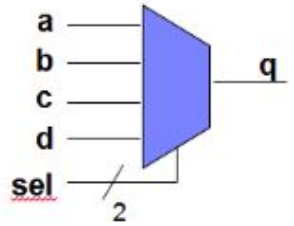
Case Statement

- Format:

```
case {expression}
    <condition1> :
        {sequence of statements}
    <condition2> :
        {sequence of statements}
    …
    default : -- (optional)
        {sequence of statements}
endcase
```

- Example:

```
always @ * begin
    case (sel)
        2'b00 : q = a;
        2'b01 : q = b;
        2'b10 : q = c;
        default : q = d;
    endcase
end
```



Two Other Forms of case Statement

- **casez**
  - Treats both **Z** and **?** in the case conditions as don't cares

```
casez (encoder)
    4'b1??? : high_lvl = 3;
    4'b01?? : high_lvl = 2;
    4'b001? : high_lvl = 1;
    4'b0001 : high_lvl = 0;
    default : high_lvl = 0;
endcase
```

if **encoder = 4'b1z0x**, then **high_lvl = 3**

- **casex**
  - Treats **X, Z, and ?** in the case conditions as don't cares, instead of logic values

```
casex (encoder)
    4'b1xxx : high_lvl = 3;
    4'b01xx : high_lvl = 2;
    4'b001x : high_lvl = 1;
    4'b0001: high_lvl = 0;
    default : high_lvl = 0;
endcase
```

if **encoder = 4'b1z0x**, then **high_lvl = 3**

Forever and Repeat Loops

- **forever** loop - executes continually

```
initial begin
    clk = 0;
    forever  #25  clk = ~clk;
end
```

Clock with period of 50 time units

Not synthesizable!

- **repeat** loop - executes a fixed number of times

```
if (rotate == 1)
    repeat (8) begin
        tmp = data[15];
        data = {data << 1, tmp};
    end
```

Repeats a rotate operation 8 times

While Loop

- **while** loop - executes if expression is true

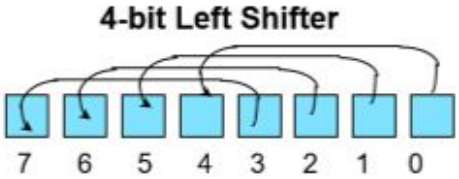
```
initial begin
    count = 0;
    while (count < 101) begin
        $display ("Count = %d", count);
        count = count + 1;
    end
end
```

Counts from 0 to 100
Exits loop at count 101

Not synthesizable!

For Loop

- **for** loop - executes initial assignment at the start of the loop and then executes loop body if expression is true




```
// declare the index for the FOR loop
integer  i;

always @(inp, cnt) begin
    result[7:4] = 0;
    result[3:0] = inp;
    if (cnt == 1) begin
        for (i = 4; i <= 7; i = i + 1) begin
            result[i] = result[i-4];
        end
        result[3:0] = 0;
    end
end
```

Synchronous vs. Asynchronous

Synchronous Preset & Clear

```
module dff_sync (
    input d, clk, sclr, spre,
    output reg q
);

    always @(posedge clk)
    begin
        if (sclr)
            q <= 1'b0;
        else if (spre)
            q <= 1'b1;
        else
            q <= d;
    end

endmodule
```

Asynchronous Clear

```
module dff_async (
    input d, clk, aclr,
    output reg q
);

    always @(posedge clk,
                posedge aclr)
    begin
        if (aclr)
            q <= 1'b0;
        else
            q <= d;
    end

endmodule
```