



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

Wykład 7  
Testbenche, atrybuty

Wrocław 2019



# Plan

- Testbenches
- Attributes vs Constraints
  - Attributes
  - Constraints
- UCF
- Constraints in FPGAs



# Test benches



# Testbench - basics

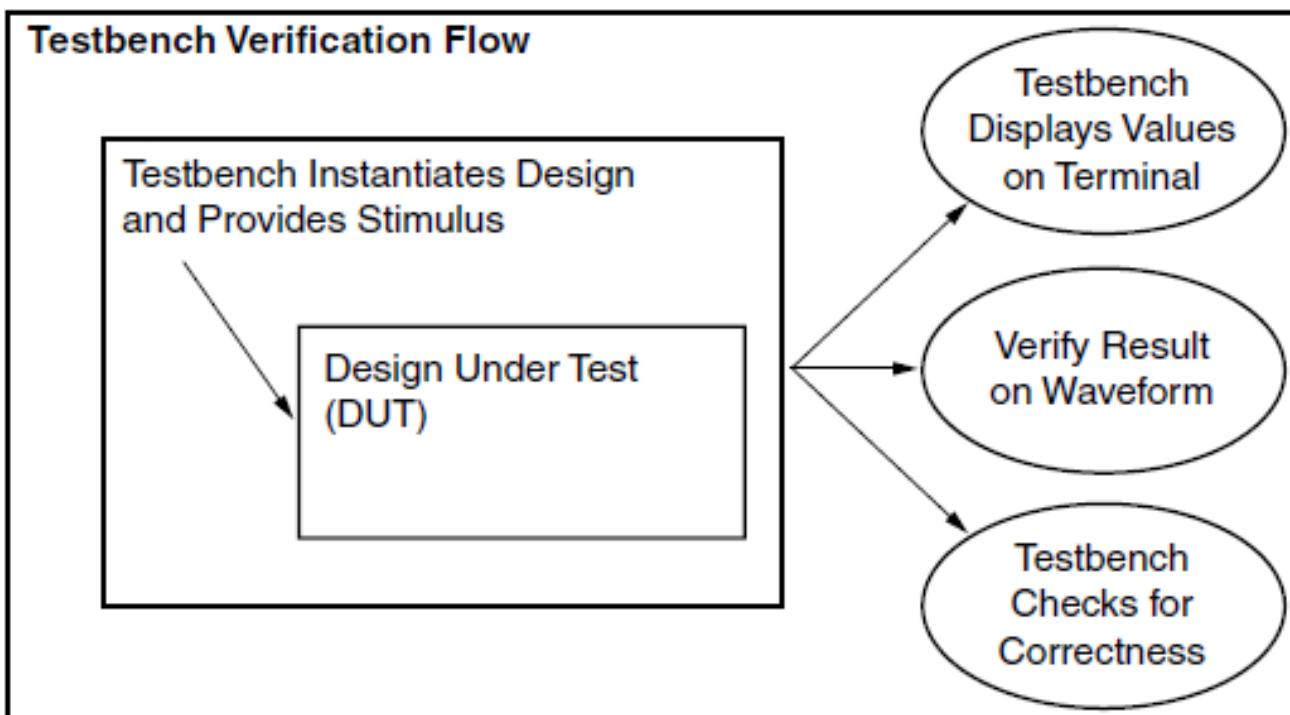
- A testbench is a program that mimics a physical lab bench
- Testbenches have become the standard method to verify HLL (High-Level Language) designs
- There are two main methods of preparing testbenches:
  - By the use of a graphical interface
  - By writing a simulation code



# Testbench - basics

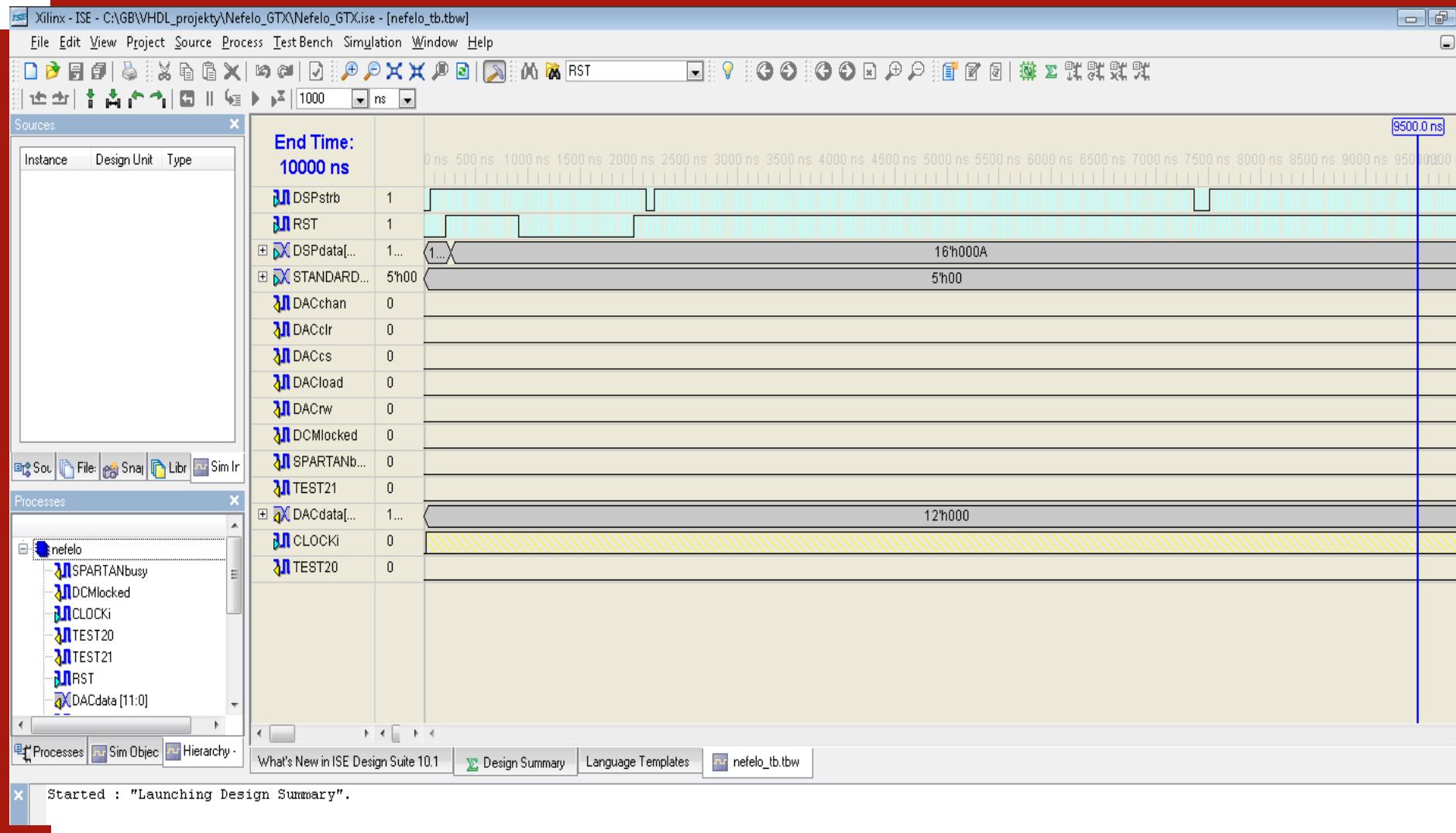
- Typically, testbenches perform the following tasks:
  - Instantiate the design under test (DUT)
  - Stimulate the DUT by applying test vectors to the model
  - Output results to a terminal or waveform window for visual inspection
  - Optionally compare actual results to expected results
- Typically, testbenches are written in the industry-standard VHDL or Verilog hardware description languages.
- Testbenches invoke the functional design, then stimulate it

# Testbench Verification Flow





# Testbench Waveform Editor





# Testbench VHDL code

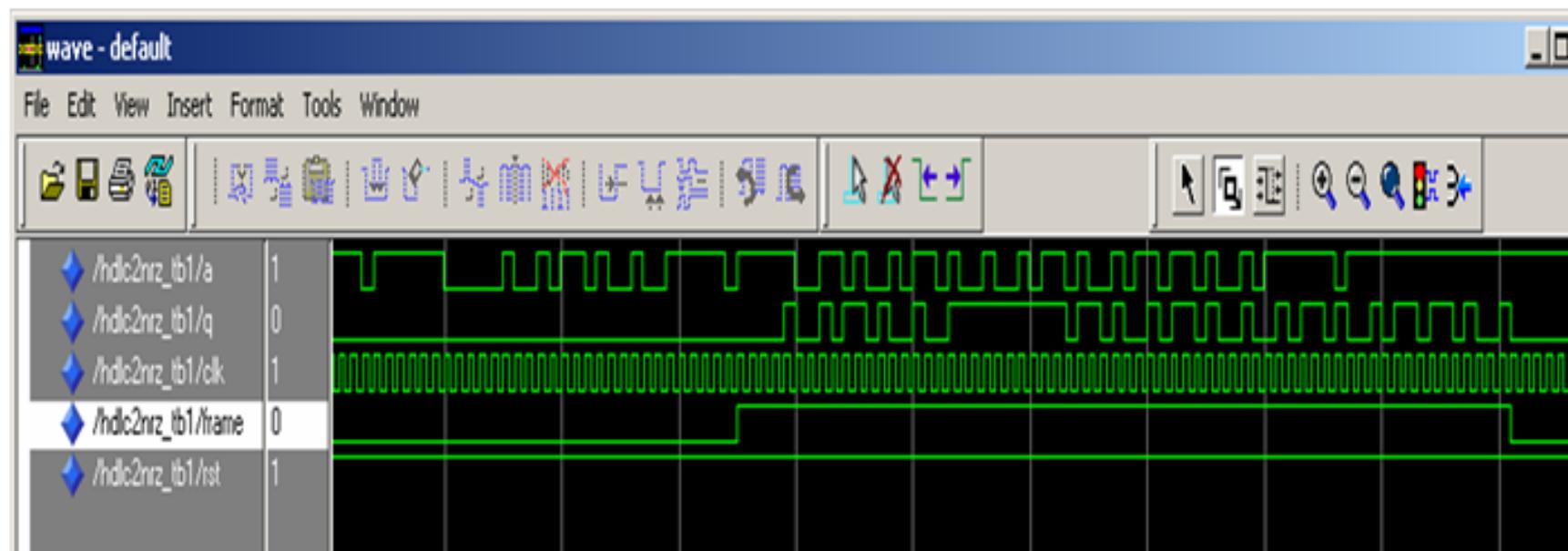
Lister - [c:\GB\VHDL\_projekty\lsp303d\\_LASER\_COUNT\_TB.VHD]

Plik Edytuj Opcje Pomoc 2%

```
PROCEDURE CHECK_data(
    next_data : std_logic_vector (7 DOWNTO 0);
    TX_TIME : INTEGER
) IS
    VARIABLE TX_STR : String(1 to 512);
    VARIABLE TX_LOC : LINE;
BEGIN
    -- If compiler error ("/= is ambiguous) occurs in the next line of code
    -- change compiler settings to use explicit declarations only
    IF (data /= next_data) THEN
        write(TX_LOC,string'("Error at time="));
        write(TX_LOC, TX_TIME);
        write(TX_LOC,string'("ns data="));
        write(TX_LOC, data);
        write(TX_LOC, string'(", Expected = "));
        write(TX_LOC, next_data);
        write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        writeline(results, TX_LOC);
        Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;

BEGIN
-----
lambda_A <= transport '0';
lambda_A2 <= transport '0';
lambda_B <= transport '0';
lambda_B2 <= transport '0';
reset <= transport '0';
addr <= transport std_logic_vector('000'); -->
cpu_trig <= transport '0';
count_a0 <= transport '1';
count_a1 <= transport '1';
cs_count <= transport '1';
-----
WAIT FOR 25 ns; -- Time=25 ns
-----
WAIT FOR 25 ns; -- Time=50 ns
reset <= transport '0';
```

# Simulation results





# Testbench construction



# Testbench - construction

- Typically testbenches written in VHDL contain sections:
  - Entity and Architecture Declaration
  - Signal Declaration
  - Instantiation of Top-level Design



# Testbench - construction

- Generating clock signal:

```
Constant ClockPeriod : TIME := 10 ns;
```

-- Clock Generation method 1:

```
Clock <= not Clock after ClockPeriod / 2;
```

-- Clock Generation method 2:

```
GENERATE_CLOCK: process
begin
    wait for (ClockPeriod / 2)
    Clock <= '1';
    wait for (ClockPeriod / 2)
    Clock <= '0';
end process;
```



# Testbench - construction

- Providing Stimulus (absolute time):

```
AbsoluteTimeStimulus: process begin
    Reset <= '1';
    Load <= '0';
    Count_UpDn <= '0';

    wait for 100 ns;

    Reset <= '0';

    wait for 20 ns;

    Load <= '1';

    wait for 20 ns;

    Count_UpDn <= '1';
end process;
```



# Testbench - construction

- Providing Stimulus (relative time):

```
Process (Clock)
Begin
If rising_edge(Clock) then
    TB_Count <= TB_Count + 1;
end if;
end process;

RealtimeStimulus: process begin
if (TB_Count <= 5) then
    Count_UpDn <= '0';
else
    Count_UpDn <= '1';
end process;
```



# Testbench - construction

- Things to remember:
  - initial blocks are executed concurrently along with other process and initial blocks in the file
  - within each (process or initial) block, events are scheduled sequentially, in the order written
  - stimulus sequences begin in each concurrent block at simulation time zero
  - Multiple blocks should be used to break up complex stimulus sequences into more readable and maintainable code



# Testbench - construction

- Displaying results:
  - Facilitated in Verilog – dedicated keywords
  - In VHDL std\_textio package functions should be used



# Testbench - construction

- Self-checking testbench:
  - Self-checking testbenches are implemented by placing a series of expected vectors in a testbench file
  - Vectors are compared at defined run-time intervals to actual simulation results
  - If actual results match expected results, the simulation succeeds
  - In VHDL vectors are read from a file (`std_textio`)



# Testbench - construction

```
while not endfile(vector_file) loop
    readline(vector_file, l);
    read(l, r, good => good_number);
    next when not good_number;
    vector_time := r * 1 ns;
    if (now < vector_time) then
        wait for vector_time - now;
    end if;
    assert good_val REPORT "bad outvalue";
    wait for 10 ns;
end loop;
```



# Testbench – example (counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity test is
port (clk : in std_logic;
      count : out std_logic_vector(3 downto 0);
      reset :in std_logic
     );
end test;

architecture Behavioral of test is
signal c : std_logic_vector(3 downto 0) :=(others => '0'); --initializing count to zero.
begin
  count <= c;
  process(clk,reset)
  begin
    if(clk'event and clk='1') then
      -- when count reaches its maximum(that is 15) reset it to 0
      if(c = "1111") then
        c <="0000";
      end if;
      c <= c+'1'; --increment count at every positive edge of clk.
    end if;

    if(reset='1') then --when reset equal to '1' make count equal to 0.
      c <=(others => '0'); -- c ="0000"
    end if;
  end process;
```



# Testbench – example (testbench 1/2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;

-- entity declaration for your testbench.Dont declare any ports here
ENTITY test_tb IS
END test_tb;

ARCHITECTURE behavior OF test_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT test  --'test' is the name of the module needed to be tested.
    --just copy and paste the input and output ports of your module as such.
        PORT(
            clk : IN std_logic;
            count : OUT std_logic_vector(3 downto 0);
            reset : IN std_logic
        );
    END COMPONENT;
    --declare inputs and initialize them
    signal clk : std_logic := '0';
    signal reset : std_logic := '0';
    --declare outputs and initialize them
    signal count : std_logic_vector(3 downto 0);
    -- Clock period definitions
    constant clk_period : time := 1 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: test PORT MAP (
        clk => clk,
        count => count,
        reset => reset
    );

```



# Testbench – example (testbench)

2/2

```
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: test PORT MAP (
        clk => clk,
        count => count,
        reset => reset
    );

    -- Clock process definitions( clock with 50% duty cycle is generated here.
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;  --for 0.5 ns signal is '0'.
        clk <= '1';
        wait for clk_period/2;  --for next 0.5 ns signal is '1'.
    end process;
    -- Stimulus process
    stim_proc: process
    begin
        wait for 7 ns;
        reset <='1';
        wait for 3 ns;
        reset <='0';
        wait for 17 ns;
        reset <= '1';
        wait for 1 ns;
        reset <= '0';
        wait;
    end process;
END;
```

# Testbench – reading from a file

- Reading from files is very important for VHDL simulation
- An example entity header:

```
entity FILE_READ is
generic(
    stim_file: string := "sim.dat"
);
port(
    CLK  : in  std_logic;
    RST  : in  std_logic;
    Y    : out std_logic_vector(4 downto 0);
    EOG  : out std_logic
);
end FILE_READ;
```

- Data is read from a file *sim.dat* at every rising clock edge and applied to the output vector *Y*
- Once every line of the file is read the *EOG* (End Of Generation) flag is set



# Testbench – reading from a file

- The main part of the architecture body:

```
architecture read_from_file of FILE_READ is

    file stimulus: TEXT open read_mode is stim_file;

begin
    EOG <= '0';

    -- wait for Reset to complete
    wait until RST='1';
    wait until RST='0';

    while not endfile(stimulus) loop

        -- read digital data from input file
        readline(stimulus, l);
        read(l, s);
        Y <= to_std_logic_vector(s);

        wait until CLK = '1';

    end loop;

    print("I@FILE_READ: reached end of "& stim_file);
    EOG <= '1';

    wait;
```



# Attributes



# Attributes

- Attributes are a feature of VHDL that allow to extract additional information about an object (such as a signal, variable or type) that may not be directly related to the value that the object carries.
- Attributes also allow to assign additional information (such as data related to synthesis) to objects in your design description.



# Attributes

- There are two classes of attributes:
  - predefined as a part of the 1076 standard,
  - introduced outside of the standard:
    - by the designer
    - by the design tool supplier (like Xilinx or Altera)



# Predefined attributes

- There are four kinds of predefined attributes:
  - Attributes on types
  - Attributes on arrays
  - Attributes on signals
  - Attributes on named entities



# Attributes on types 1/5

T'BASE

Gives the base type of a type T. Can only be used as a prefix for another attribute.

Example:

```
SUBTYPE MyNat IS NATURAL RANGE 5 TO 15;  
MyNat'BASE'LOW = -2_147_483_647 (=INTEGER'LOW)
```

T'LEFT

Gives the leftmost value in the type T.

Example:

```
TYPE State IS (reset,start,count);  
State'LEFT = reset
```

T'RIGHT

Gives the rightmost value in the type T.

Example:

```
TYPE State IS (reset,start,count);  
State'RIGHT = count
```



# Attributes on types 2/5

T'HIGH

Gives the largest value in the type T.

Example:

```
SUBTYPE MyNat IS NATURAL RANGE 5 TO 15;  
MyNat'HIGH = 15
```

T'LOW

Gives the smallest value in the type T.

Example:

```
SUBTYPE MyNat IS NATURAL RANGE 5 TO 15;  
MyNat'LOW = 5
```

T'ASCENDING Returns a value of the type BOOLEAN that is TRUE if the type T is defined with an ascending range.

Example:

```
SUBTYPE MyNat IS NATURAL RANGE 5 TO 15;  
MyNat'ASCENDING = TRUE
```



# Attributes on types 3/5

**T'IMAGE(X)\*** Converts the value X, that is of the type or subtype T, to a text string. T must be a scalar type, i.e. an integer, a real, a physical type or an enumerated type.

Example:

```
TYPE State IS (reset,start,count);  
State'IMAGE(start) = "start"
```

**T'VALUE(X)\*** Converts the text string X to a value of the type T. T must be a scalar type, i.e. an integer, a real, a physical type or an enumerated type.

Example:

```
TYPE State IS (reset,start,count);  
State'VALUE("reset") = reset
```

**T'POS(X)** Returns the position number of X within the type T.

Example:

```
TYPE State IS (reset,start,count);  
State'POS(start) = 1
```



# Attributes on types 4/5

T'VAL(X)

Returns the value on position X in the type T.

Example:

```
TYPE State IS (reset,start,count);  
State'VAL(0) = reset
```

T'SUCC(X)

Returns the value, of the type T, with the position number one greater than that of the parameter X.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;  
MyInteger'SUCC(0) = 1
```

T'PRED(X)

Returns the value, of the type T, with the position number one less than that of the parameter X.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;  
MyInteger'PRED(0) = -1
```



# Attributes on types 5/5

**T'LEFTOF(X)** Returns the value to the left of the value X in the range of the type T.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;  
MyInteger'LEFTOF(0) = 1
```

**T'RIGHTOF(X)** Returns the value to the right of the value X in the range of the type T.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;  
MyInteger'RIGHTOF(0) = -1
```



# Attributes on arrays 1/4

**A'HIGH[(N)]** Returns the numerical largest index limit in the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M' HIGH = 3  
matrix'HIGH(2) = 2
```

**A'LOW[(N)]** Returns the numerical smallest index limit in the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M' LOW = 0  
matrix'LOW(2) = 1
```



# Attributes on arrays 2/4

**A'LEFT[(N)]** Returns the left index limit for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M' LEFT = 0
matrix'LEFT(2) = 2
```

**A'RIGHT[(N)]** Returns the right index limit for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M' RIGHT = 3
matrix'RIGHT(2) = 1
```



# Attributes on arrays 3/4

**A'RANGE[(N)]** Returns the index range as a RANGE for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M' RANGE = 0 TO 3  
matrix'RANGE(2) = 2 DOWNTO 1
```

**A'REVERSE\_RANGE[(N)]**

Returns the index range as a RANGE, but with opposite direction, for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M'REVERSE_RANGE = 3 DOWNTO 0  
matrix'REVERSE_RANGE(2) = 1 TO 2
```



# Attributes on arrays 4/4

## A'LENGTH[(N)]

Returns the number of elements in the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M' LENGTH = 4  
matrix' LENGTH(2) = 2
```

## A'ASCENDING[(N)]\*

Returns a value of the type BOOLEAN that is TRUE if the index range N for array A is ascending. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;  
VARIABLE matrix : M;  
M' ASCENDING = TRUE  
matrix' ASCENDING(2) = FALSE
```



# Attributes on signals 1/4

S'STABLE[(T)] Creates a new signal of the type BOOLEAN that returns TRUE as long as the signal S does not change its value. The signal gets FALSE when S changes value and is FALSE during the time T. T may be omitted and its default value is 0 ns.

S'QUIET[(T)] Functions exactly as 'STABLE but reacts on all updates of S from its driver queue, also when the signal is assigned the value it already has.

S'DELAYED[(T)]

Creates a copy of the signal S delayed the time T. T may be omitted and its default value is 0 ns, i.e. exactly S.



# Attributes on signals 2/4

## S' TRANSACTION

Creates a new signal of the type BIT that changes value every time the signal S gets a new value from its driver queue, i.e. also when it gets the same value as it already has. The initial value of the new signal is not specified.

## S'

A function of the type BOOLEAN that returns TRUE during exactly one delta cycle when the signal S gets a new value.

## S' ACTIVE

Functions exactly as 'EVENT' but reacts on all updates of S from its driver queue, also when the signal is assigned the value it already has.



# Attributes on signals 3/4

## S'LAST\_EVENT

A function of the type TIME returning the time since the last change of the value of the signal S.

## S'LAST\_ACTIVE

A function of the type TIME returning the time since the last update of the signal S.

## S'LAST\_VALUE

A function of the same base type as S returning the value the signal S had before its last value change.



# Attributes on signals 4/4

**S'DRIVING\*** A function of the type BOOLEAN returning TRUE if the driver for the signal S is on.

**S'DRIVING\_VALUE\***

A function of the same type as S returning the current value in the driver for the signal S in the current process.



# Attributes on named entities 1/2

## E'SIMPLE\_NAME\*

Returns the name, in a text string with lower-case letters, of a named entity.

## E'INSTANCE\_NAME\*

Returns the hierarchical path including instances higher in the hierarchy, in a string with lower-case letters, to a named entity.



# Attributes on named entities 2/2

## E'PATH\_NAME\*

Returns the hierarchical path not including instances higher in the hierarchy, in a string with lower-case letters, to a named entity.

Example:

```
ENTITY E IS
  ...
END ENTITY E;

ARCHITECTURE A OF E IS
BEGIN
  P: PROCESS(clock)
    VARIABLE inVar : NATURAL RANGE 0 TO 255;
BEGIN
  ...
--  inVar'SIMPLE_NAME = "invar"
--  inVar'INSTANCE_NAME = ":e(a):p:invar"
--  inVar'PATH_NAME = ":e:p:invar"
END PROCESS P;
END ARCHITECTURE A;
```



# Attributes vs Constraints

- There are two classes of attributes:
  - predefined as a part of the 1076 standard,
  - introduced outside of the standard:
    - by the designer
    - by the design tool supplier (like Xilinx or Altera)



# Custom Xilinx attributes

- Custom attributes have to be declared:
  - In the entity – visibility also in the architecture body
  - In the architecture body
- Declaration:
  - attribute ram\_style: string;
- Usage:
  - attribute ram\_style of RAM: signal is "pipe\_distributed";



# Custom Xilinx attributes

- Attributes vs Constraints:
  - Names can be used interchangeably for custom attributes
  - We can define subcategories:
    - Attributes - property associated with a device architecture primitive component that generally affects an instantiated component's functionality or implementation
    - Synthesis constraints - direct the synthesis tool optimization technique for a particular design or piece of HDL code
    - Implementation constraints - instructions given to the FPGA implementation tools to direct the mapping, placement, timing, etc. for the implementation tools

# Spartan 3E attributes (some😊)

- DLL attributes:

Attribute	Description	Values
CLK_FEEDBACK	Chooses either the CLK0 or CLK2X output to drive the CLKFB input	NONE, <u>1X</u> , 2X
CLKIN_DIVIDE_BY_2	Halves the frequency of the CLKIN signal just as it enters the DCM	<u>FALSE</u> , TRUE
CLKDV_DIVIDE	Selects the constant used to divide the CLKIN input frequency to generate the CLKDV output frequency	1.5, <u>2</u> , 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6.0, 6.5, 7.0, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, and 16
CLKIN_PERIOD	Additional information that allows the DLL to operate with the most efficient lock time and the best jitter tolerance	Floating-point value representing the CLKIN period in nanoseconds

# Spartan 3E attributes (some😊)

- Block RAM attributes:

Function	Attribute	Possible Values
Initial Content for Data Memory, Loaded during Configuration	INITxx (INIT_00 through INIT3F)	Each initialization string defines 32 hex values of the 16384-bit data memory of the block RAM.
Initial Content for Parity Memory, Loaded during Configuration	INITPxx (INITP_00 through INITP0F)	Each initialization string defines 32 hex values of the 2048-bit parity data memory of the block RAM.
Data Output Latch Initialization	INIT (single-port) INITA, INITB (dual-port)	Hex value the width of the chosen port.
Data Output Latch Synchronous Set/Reset Value	SRVAL (single-port) SRVAL_A, SRVAL_B (dual-port)	Hex value the width of the chosen port.
Data Output Latch Behavior during Write (see <b>Block RAM Data Operations</b> )	WRITE_MODE	WRITE_FIRST, READ_FIRST, NO_CHANGE



# CPLD attributes

“BUFG (CPLD)”	“Collapse (COLLAPSE)”	“CoolCLOCK (COOL_CLK)”
“Data Gate (DATA_GATE)”	“Fast (FAST)”	“Input Registers (INREG)”
“Input Output Standard (IOSTANDARD)”	“Keep (KEEP)”	“Keeper (KEEPER)”
“Location (LOC)”	“Maximum Product Terms (MAXPT)”	“No Reduce (NOREDUCE)”
“Offset In (OFFSET IN)”	“Open Drain (OPEN_DRAIN)”	“Period (PERIOD)”
“Offset Out (OFFSET OUT)”		
“Prohibit (PROHIBIT)”	“Pullup (PULLUP)”	“Power Mode (PWR_MODE)”
“Registers (REG)”	“Schmitt Trigger (SCHMITT_TRIGGER)”	“Slow (SLOW)”
“Timing Group (TIMEGRP)”	“Timing Specifications (TIMESPEC)”	“Timing Name (TNM)”



# FPGA constraints

- FPGA constraints can be split into:
  - **Grouping Constraints**
  - **Logical Constraints**
  - **Physical Constraints**
  - **Mapping Directives**
  - **Placement Constraints**
  - **Routing Directives**
  - **Synthesis Constraints**
  - **Timing Constraints**
  - **Configuration Constraints**



# Constraints usage in VHDL

- Before a constraint can be used, it must be declared with the following syntax:
  - **attribute *attribute\_name* : string;**
- Example:
  - **attribute RLOC : string;**
- Once the attribute is declared, a VHDL attribute can be specified as:
  - **attribute bufg of my\_clock: signal is “clk”;**



# Constraints usage in VHDL

- An attribute can be declared in an entity or architecture.
  - If the attribute is declared in the entity, it is visible both in the entity and the architecture body.
  - If the attribute is declared in the architecture, it cannot be used in the entity declaration.



# Where to define constraints

- Depending on a attribute they can be defined in different places:
  - Constraints Editor – *Timing Constraints*
  - Floorplanner – *Non-timing placement constraints*
  - PACE – *IO placement and area constraints*
  - Schematic and Symbol Editors - *IO placement and RLOC constraints*



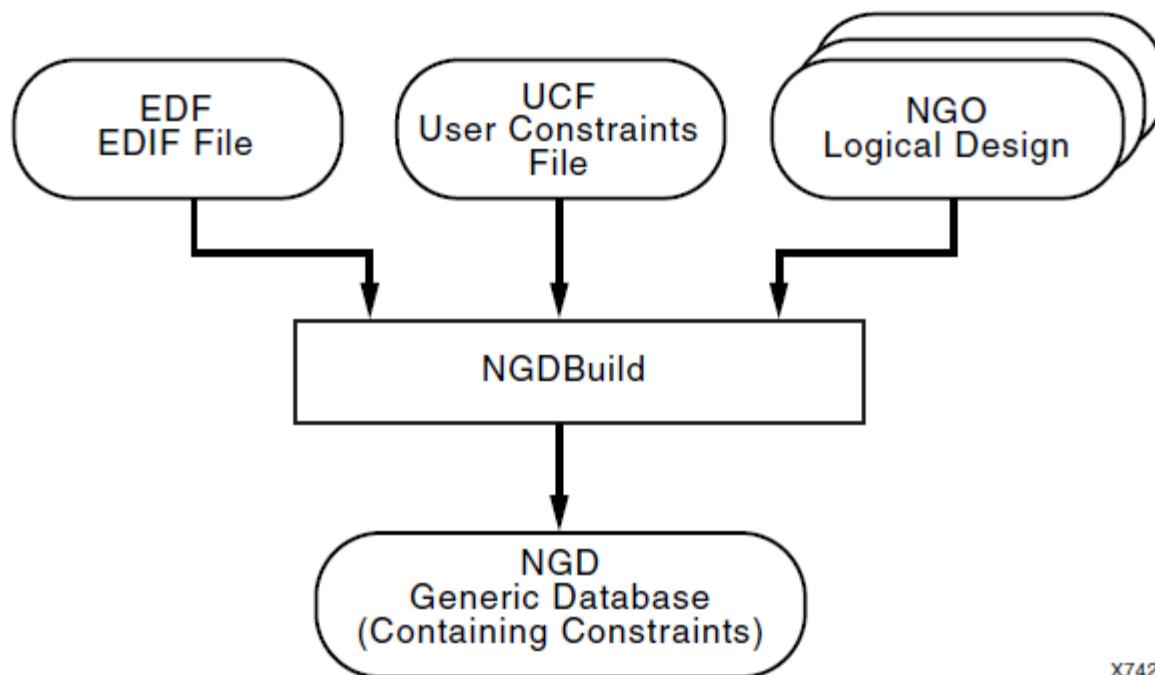
# **UCF / NCF**



# User Constraints File

- The UCF file is an ASCII file specifying constraints on the logical design
- The UCF file can be created with any text editor
- NCF - Netlist Constraint File, an ASCII file generated by synthesis programs

# UCF file flow



X7423



# UCF/NCF rules

- The UCF and NCF files are case sensitive.
- However, any Xilinx constraint keyword (for example, LOC, PERIOD, HIGH, LOW) may be entered in all upper-case, all lower-case, or mixed case.
- Each statement is terminated by a semicolon (;)
- No continuation characters are necessary if a statement exceeds one line, since a semicolon marks the end of the statement



# UCF/NCF rules

- To comment a line a pound sign (#) is used
- C and C++ style comments /\* \*/ and respectively) are supported
- Statements need not be placed in any particular order in the UCF and NCF file
- Enclose inverted signal names that contain a tilde (for example, ~OUTSIG1) in double quotes
- Multiple constraints for a given instance can be entered



# UCF/NCF example

- The UCF file supports a basic syntax that can be expressed as:

- **{NET|INST|PIN}** “*full\_name*” *constraint*;

or as

- **SET** *set\_name* *set\_constraint*;

- Example:

```
INST DCM_INST CLK_FEEDBACK = NONE;  
INST DCM_INST CLKDV_DIVIDE = 2.0;  
INST SDA_pin          LOC=P44;
```



# UCF/NCF addons

- If using Reserved Words (like „net”) double quote is mandatory
- Wildcard characters, asterisk (\*) and question mark (?) can be used in as follows:
  - The asterisk (\*) represents any string of zero or more characters.
  - The question mark (?) indicates a single character
- Example:
  - NET “\*AT?” FAST;



# UCF/NCF addons

- If multiple constraints are necessary then:
  - **INST** *instanceName* *constraintName* = *constraintValue* |  
*constraintName* = *constraintValue*;
- For example:
  - INST myInst LOC = P53 | IOSTANDARD = LVPECL33 | SLEW = FAST;



PCF



# PCF

- The NGD file produced when a design netlist is read into the Xilinx Development System may contain a number of logical constraints from:
  - UCF
  - NCF
  - VHDL code
- Logical constraints in the NGD file are read by MAP. MAP uses some of the constraints to map the design and converts logical constraints to physical constraints.



# PCF

- MAP writes these physical constraints into a Physical Constraints File (PCF)
- The PCF file is an ASCII file containing two separate sections:
  - A section for those physical constraints created by the mapper
  - A section for physical constraints entered by the user



# PCF

- The structure of the PCF file is as follows.
  - schematic start;
  - *translated schematic and UCF and NCF constraints in PCF format*
  - schematic end;
  - *user-entered physical constraints*
- ***Caution!*** All user-entered physical constraints MUST be put after the “schematic end” statement. Any constraints preceding this section or within this section may be overwritten or ignored.



# PCF example

- UCF file:
  - INST LED1 LOC=P65;
  - INST LED2 LOC=P64;
- PCF file
  - COMP "LED1" LOCATE = SITE "P65" LEVEL 1;
  - COMP "LED2" LOCATE = SITE "P64" LEVEL 1;



# Constraints Editor



# Constraints Editor

- Constraints Editor can be found in the ISE environment
- Constraints Editor is used to enter timing constraints
- The user interface simplifies constraint entry by guiding through constraint creation without needing to understand UCF file syntax.



# Constraints Editor

- Constraints Editor requires:
  - A User Constraints File (UCF)
  - A Native Generic Database (NGD) file
- After the constraints are created or modified with Constraints Editor, NGDBuild must be run again, using the new UCF and design source netlist files as input and generating a new NGD file as output



Xilinx - ISE - C:\GBWHDL\_projekty\HS\_counter\_5\HS\_counter\_5.ise - [Timing Constraints\*]

File Edit View Project Source Process Window Help

Sources Constraint Files dd2.ucf  
Show Constraints from Specified File only Show Constraints from All Files

Constraint Type Timing Constraints Global Ports Advanced Group Constraints Miscellaneous

Processes Processes for: HS\_counter\_main - Behavior Add Existing Source Create New Source View Design Summary Design Utilities User Constraints Create Timing Constraints Floorplan IO - Pre-Synthesis Floorplan Area / IO / Logic Synthesize - XST

Timing Constraints Pad to Pad... 20 ns.

Design Summary HS\_counter\_main.vhd Timing Constraints

**Clock To Pad**

Output interface detail:  
 Single data rate  Dual data rate  
Output clock pad net: Clk\_ext  
Output pad timegroup: Create ...

Rising edge constraints:  
Offset out: Units: ns  
Output skew reference pin: <Default>  
Output register timegroup: Create ...

Falling edge constraints:  
Offset out: Units: ns  
Output skew reference pin: <Default>  
Output register timegroup: Create ...

Rising edge comment:  
Falling edge comment:

**Output Interface Detail:**

- The **Single Data Rate** and **Dual Data Rate** determine the output interface type.
- The Output clock Pad Net is the clock net used to trigger the outgoing data.
- The optional Output pad timegroup limits the scope of the OFFSET OUT constraint to only those data pins defined in the PAD timegroup.
- A new Pad Group may be defined by selecting the Create New Pad Group button.

**Rising Constraint Parameters:**

- The optional Rising Clock-to-Output (OFFSET OUT) is the time from the rising clock edge at the input pin of the FPGA until data becomes valid at the output pin of the FPGA. For source-synchronous designs, the OFFSET OUT value can be left blank and only a skew report will be generated.
- The Output Skew Reference Pin is the reference signal in which the skew of all bits in the bus will be reported against.
- The optional Output Register Timegroup is used to limit the scope of the constraint to a subset of registers.
- RISING is a predefined Output Register Group which indicates the constraint applied to all rising edge registers.

**Falling Constraint Parameters:**

OK Cancel Apply Help



# Floorplanner



# Floorplanner

- Floorplanner is a part of the ISE environment
- It is used for defining parameters of input/output signals and for defining the positioning of the logic nets inside the chip
- **Easy** way of making area constraints
- Floorplanner can be used after translation and/or before mapping
- Floorplanner reads the UCF file constraints



Xilinx PACE - C:\GB\HDL\_projekty\filter\_design\main\_filter.ucf

File Edit View IOBs Areas Tools Window Help

Loading device for application Rf\_Device from file '3s200.nph' in environment C:\Xilinx\10.1\ISE.

Compiling vhdl file "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" inLibrary work.

Entity <main\_filter> compiled.

ERROR:HDLParser:164 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 40: parse error, unexpected SIGNAL

ERROR:HDLParser:3312 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 45: Undefined symbol 't'.

ERROR:HDLParser:1209 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 47: t: Undefined symbol (last report in this block)

ERROR:HDLParser:808 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 47: or can not have such operands in this context.

ERROR:HDLParser:3312 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 51: Undefined symbol 't'.

ERROR:HDLParser:1209 - "C:/GB/HDL\_projekty/filter\_design/main\_filter.vhd" Line 51: t: Undefined symbol (last report in this block)

Design Browser

I/O Pins

- A
- Clk
- Reset**
- Q

Global Logic

Logic

Design Object List - I/O Pins

I/O Name	I/O Direction	Loc	Bank	I/O Std.
A	Input			
Clk	Input	BANK3		
Q	Input			
Reset	Input			

Package Pins for xc3s200-5-pq208

Top View

Package View / Architecture View



# Floorplanner

- Floorplanner™ interactive graphical tool can be used to perform the following functions:
  - Doing detailed-level floorplanning
  - Creating an RPM core that can be used in other designs
  - Viewing and editing location constraints
  - Finding logic or nets by name or connectivity
  - Cross probing from the Timing Analyzer to the Floorplanner
  - Placing ports automatically for modular design



# FPGA Editor



# FPGA Editor

- In the FPGA Editor certain constraints can be added or deleted from the PCF (Physical Constraints File)
- In the FPGA Editor, net, site, and component constraints are supported as property fields in the individual nets and components



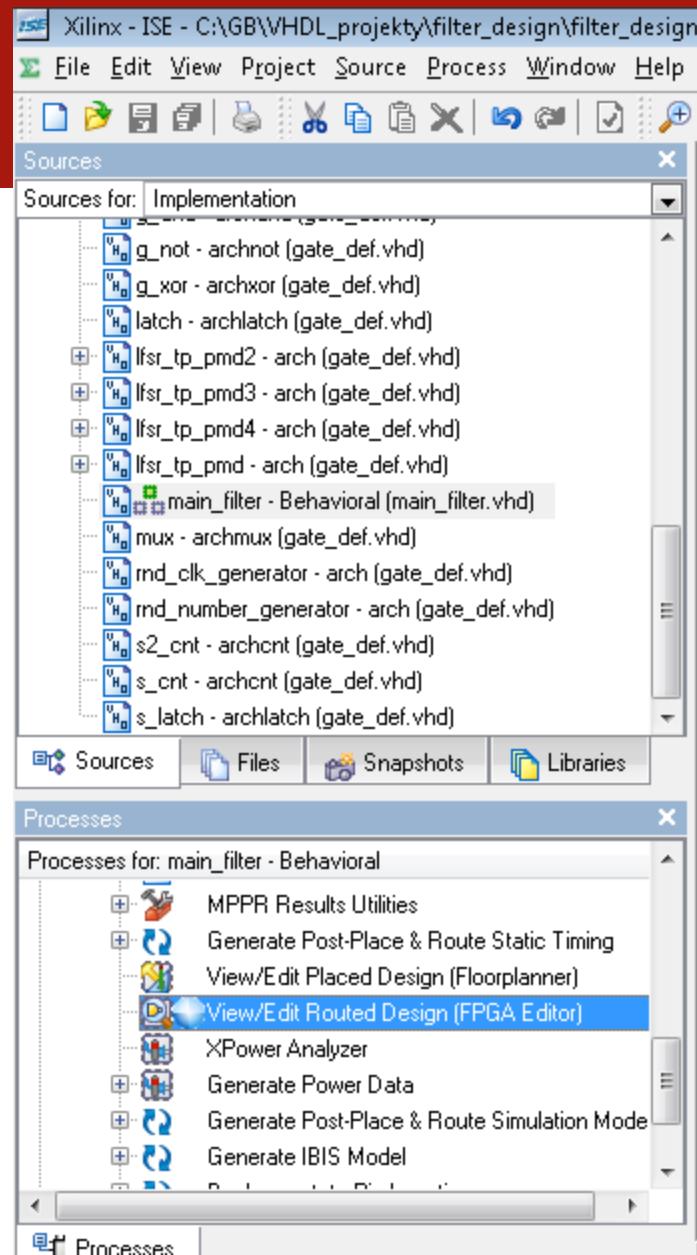
# FPGA Editor function 1/2

- Functions that can be performed in the FPGA Editor:
  - Placing and routing critical components before running the automatic place and route tools.
  - Finishing placement and routing if the routing program does not completely route the design.
  - Adding probes to the design to examine the signal states of the targeted device. Probes are used to route the value of internal nets to an IOB for analysis during the debugging of a device.
  - Cross-probing the design with Timing Analyzer.



# FPGA Editor function 2/2

- Functions that can be performed in the FPGA Editor:
  - Running the BitGen program and downloading the resulting BIT file to the targeted device.
  - Viewing and changing the nets connected to the capture units of an ILA core in your design.
  - Using the ILA command to write a .cdc file
  - Creating an entire design by hand (advanced users)
  - Modifying some constraints





Xilinx FPGA Editor - main\_filter.ncd

File Edit View Tools Window Help

Array1

List1

World1

exit  
add  
attrib  
autoroute  
clear  
delay  
delete  
drc  
editblock  
editmode  
find  
blue  
ila  
info  
probes  
autoprobe  
route  
route-fanout  
swap  
unroute

The screenshot shows the Xilinx FPGA Editor interface. The main window displays a logic array diagram with a grid of small circles representing logic cells. To the right of the array is a component list titled 'List1' showing five entries:

	Name	Site	Type	#Pins	Hilited
1	A	P185	IOB	3	[no cold]
2	Clk	P184	IOB	1	[no cold]
3	Clk_BU	BUFG	BUFG	3	[no cold]
4	Q	P187	IOB	1	blue
5	Reset	P189	IOB	1	[no cold]

The 'Hilited' column indicates which components are currently selected. Below the component list is a 'World1' window showing a single blue rectangle. The right side of the interface features a vertical toolbar with various editing functions.



# Timing Constraint Strategies



# Timing Constraints Strategies

- The easiest way to modify time critical path is to use *global constraints*
- Main timing constraints methods:
  - Global Timing Assignments
  - Specific Timing Assignments
  - Multi-Cycle and Fast or Slow Timing Assignments



# Xilinx Constraint



# Xilinx Constraints

- FPGA constraints can be split into:
  - **Grouping Constraints**
  - **Logical Constraints**
  - **Physical Constraints**
  - **Mapping Directives**
  - **Placement Constraints**
  - **Routing Directives**
  - **Synthesis Constraints**
  - **Timing Constraints**
  - **Configuration Constraints**



# AREA\_GROUP constraint

- AREA\_GROUP is a design implementation constraint that enables partitioning of the design into physical regions for mapping, packing, placement, and routing
- AREA\_GROUP is attached to logical blocks in the design, and the string value of the constraint identifies a named group of logical blocks that are to be packed together by mapper and placed in the ranges if specified by PAR



# AREA\_GROUP constraint

- Applicable to:
  - Logic groups
  - Timing groups
- Syntax example:
  - **INST “X” AREA\_GROUP=groupname;**
  - AREA\_GROUP “groupname” RANGE=range;



# BLKNM constraint

- BLKNM (Block Name) assigns block names to qualifying primitives and logic elements
- If the same BLKNM constraint is assigned to more than one instance, the software attempts to map them into the same block
- Placing BLKNM constraints on instances that do not fit within one block creates an error



# BLKNM constraint

- Applicable to:
  - Flip-flop and latch primitives
  - Any I/O element or pad
  - ROM primitives
  - RAMS and RAMD primitives
  - Carry logic primitives
- Syntax Example:
  - INST “\$1I87/block1” BLKNM=U1358;



# BUFG (CPLD) constraint

- When applied to an input buffer or input pad net, the BUFG attribute maps the tagged signal to a global net.
- When applied to an internal net, the tagged signal is either routed directly to a global net or brought out to a global control pin to drive the global net, as supported by the target device family architecture



# BUFG constraint

- Applicable to (CPLD only):
  - Any input buffer (IBUF),
  - input pad net,
  - internal net that drives a CLK, OE, SR,
  - DATA\_GATE pin
- Syntax Example:
  - NET “fastclk” BUFG=CLK;



# DCI\_VALUE constraint

- DCI\_VALUE determines which buffer behavioral models are associated with the IOBs of a design in the generation of an IBS file using IBISWriter
- Applicable to:
  - IOB
- Syntax Example:
  - INST pin\_name DCI\_VALUE = integer;
    - DCI\_VALUE are integers 25 through 100 with an implied units of ohms (default 50 ohms)



# DRIVE constraint

- DRIVE is a basic mapping directive that selects the output for SPARTAN and VIRTEX devices
- DRIVE selects output drive strength (mA) for the SelectIO buffers that use the:
  - LVTTL, LVCMOS12, LVCMOS15, LVCMOS18, LVCMOS25, or LVCMOS33 interface I/O standard.



# FAST constraint

- FAST is a basic mapping constraint. It increases the speed of an IOB output. While FAST produces a faster output, it may increase noise and power consumption.
- Applicable to:
  - Output primitives
  - Output pads
  - Bidirectional pads



# FROM TO constraint

- FROM-TO defines a timing constraint between two groups.
- It is associated with the PERIOD constraint of the high or low time.
- A group can be user-defined or predefined.
- From synchronous paths, a FROM-TO constraint controls only the setup path, not the hold path.



# FROM TO constraint

- Applicable to:
  - Predefined and user-defined groups
- Syntax Example:
  - **TIMESPEC TS*name*=FROM “*group1*” TO “*group2*”  
*value* {DATAPATHONLY};**
  - **TIMESPEC TS\_MY\_PathA = FROM “my\_src\_grp” TO  
“my\_dst\_grp” 23.5 ns DATAPATHONLY;**



# IBUF\_DELAY\_VALUE constraint

- The IBUF\_DELAY\_VALUE constraint is a mapping constraint that adds additional static delay to the input path of the FPGA array
- This constraint can be applied to any input or bidirectional signal that is not directly driving a clock or IOB (Input Output Block) register
- The IBUF\_DELAY\_VALUE constraint can be set to an integer value from 0-16
- These values do not directly correlate to a unit of time but rather additional buffer delay



# IBUF\_DELAY\_VALUE constraint

- Applicable to:
  - Any top-level I/O port
- Syntax Example:
  - **attribute IBUF\_DELAY\_VALUE : string;**
  - **attribute IBUF\_DELAY\_VALUE of DataIn1: label is "5";**



# IFD\_DELAY\_VALUE constraint

- The IFD\_DELAY\_VALUE constraint is a mapping constraint that adds additional static delay to the input path of the FPGA array
- This constraint can be applied to any input or bidirectional signal which drives an IOB (Input Output Block) register
- The IFD\_DELAY\_VALUE constraint can be set to an integer value from 0-8
- These values do not directly correlate to a unit of time but rather additional buffer delay



# IFD\_DELAY\_VALUE constraint

- Applicable to:
  - Any top-level I/O port
- Syntax Example:
  - **attribute IFD\_DELAY\_VALUE : string;**
  - **attribute IFD\_DELAY\_VALUE of DataIn1: label is "5";**



# IOSTANDARD constraint

- IOSTANDARD is a basic mapping constraint and synthesis constraint
- IOSTANDARD is used to assign an I/O standard to an I/O primitive
- IOSTANDARD works differently for FPGA and CPLD devices



# IOSTANDARD constraint

- Applicable to:
  - IBUF, IBUFG, OBUF, OBUFT
  - IBUFDS, IBUFGDS, OBUFDS, OBUFTDS
  - Output Voltage Banks
- Syntax Example:
  - **INST** “*instance\_name*”  
**IOSTANDARD**=*iostandard\_name*;
  - **NET** “*pad\_net\_name*”  
**IOSTANDARD**=*iostandard\_name*;
    - Where *iostandard\_name* is an IO Standard name as specified in the device data sheet



# KEEP constraint

- When a design is mapped, some nets may be absorbed into logic blocks.
- When a net is absorbed into a block, it can no longer be seen in the physical design database
- The net may then be absorbed into the block containing the components.
- KEEP prevents this from happening



# KEEP constraint

- Applicable to:
  - signals
- Syntax Example:
  - **attribute keep of *signal\_name*: signal is “{TRUE | FALSE}”;**
  - **NET “\$1I3245/\$SIG\_0” KEEP;**



# LOC constraint

- LOC defines where a design element can be placed within an FPGA
- It specifies the absolute placement of a design element on the FPGA die
- It can be a single location, a range of locations, or a list of locations
- LOC can be specified from the design file and also direct placement with statements in a constraints file



# NOREDUCE constraint

- NOREDUCE is a fitter and synthesis constraint for CPLD devices
- It prevents minimization of redundant logic terms that are typically included in a design to avoid logic hazards or race conditions
- When constructing combinatorial feedback latches in a design, always apply NOREDUCE to the latch's output net and include redundant logic terms when necessary to avoid race conditions.



# NOREDUCE constraint

- Applicable to:
  - CPLD devices
  - All nets
- Syntax Example:
  - **attribute NORREDUCE of *signal\_name*: signal is “{TRUE | FALSE}”;**
  - **NET “\$SIG\_12” NORREDUCE;**



# PIN constraint

- The PIN constraint in conjunction with LOC defines a net location
- Applicable to:
  - Nets
- Syntax Example:
  - **PIN “*module.pin*” LOC=*location*;**



# RLOC constraint

- RLOC (Realtive Location) constraints group logic elements into discrete sets and allow to define the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design
- The Blocks are set relative to each other to increase speed and use of die resources efficiently



# RLOC constraint

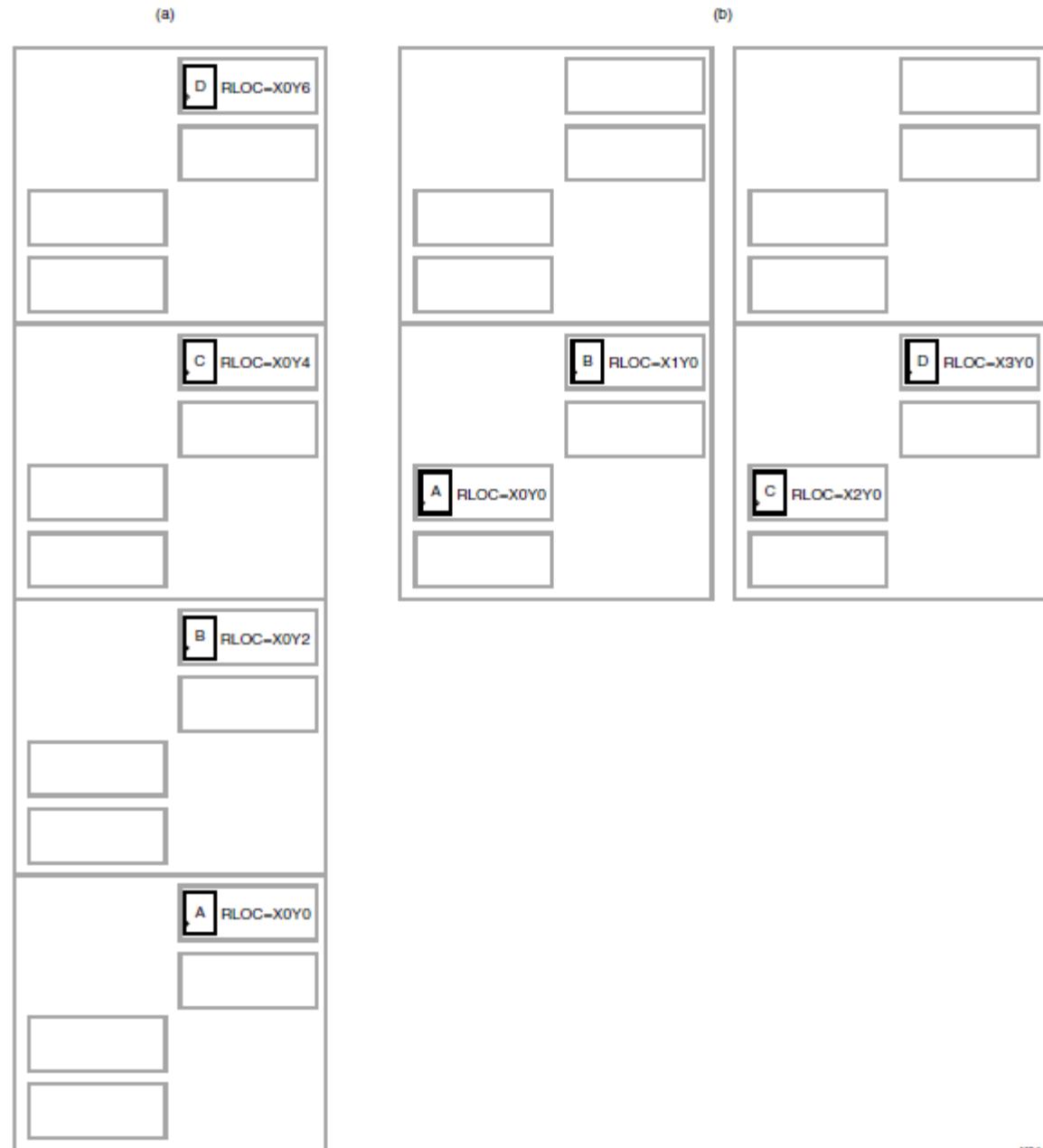
- Applicable to:
  - Registers
  - ROM
  - RAMS, RAMD
  - BUFT
  - LUTs, MUXCY, XORCY, MULT\_AND, SRL16, SRL16E
  - DSP48
  - MULT18x18



# RLOC constraint

- Syntax Example:
  - Syntax depends on the device
  - For Spartan-3 and most Virtex devices:
    - **RLOC=X $m$ Y $n$** 
      - *where*
      - $m$  and  $n$  are the relative X-axis (left/right) value and the relative Y-axis (up/down) value, respectively
      - the X and Y numbers can be any positive or negative integer including zero

- Two RLOC specifications for four flip-flops





**Thank you for your attention**



# References

- [1] „Combinational Circuits”, <http://www.cs.Princeton.EDU/~cos126>
- [2] <http://www.cs.umbc.edu/portal/help/VHDL/>
- [3] <http://ece.wpi.edu/~rjduck/Xilinx%20VHDL%20Test%20Bench%20Tutorial%20.pdf>
- [4] <http://www.seas.upenn.edu/~ese171/vhdl/VHDLTestbench.pdf>
- [5] <http://vhdlguru.blogspot.com/2010/03/how-to-write-testbench.html>
- [6] [http://www.digilentinc.com/Data/Documents/Tutorials/Xilinx%20ISE%20Simulator%20\(I\\$im\)%20VHDL%20Test%20Bench%20Tutorial.pdf](http://www.digilentinc.com/Data/Documents/Tutorials/Xilinx%20ISE%20Simulator%20(I$im)%20VHDL%20Test%20Bench%20Tutorial.pdf)
- [7] [http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx11/ug682.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug682.pdf)



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

### Wykład 9 Arytmetyka w FPGA

Wrocław 2020



# Plan

- FPGA as DSP
- Fixed point multipliers
- CORDIC



# Making DSP out of Spartan 3

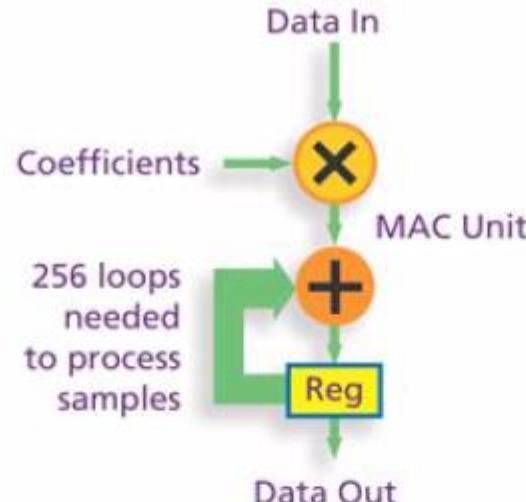


# FPGA as DSP

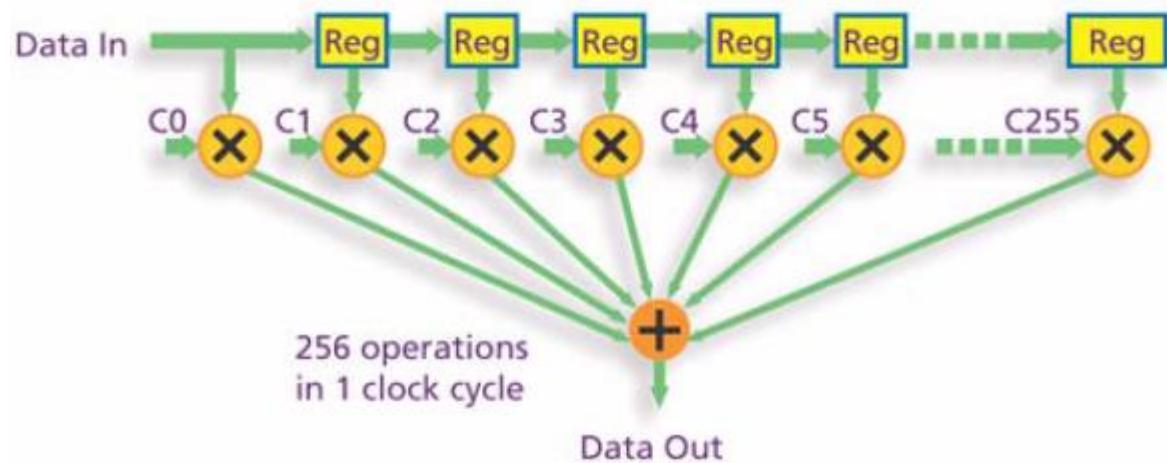
- FPGAs bring two key advantages to digital signal processing:
  - their architectures are well suited for highly parallel implementation of DSP functions (high performance)
  - user programmability allows designers to trade-off device area vs. performance by selecting the appropriate level of parallelism to implement their functions

# FPGA as DSP

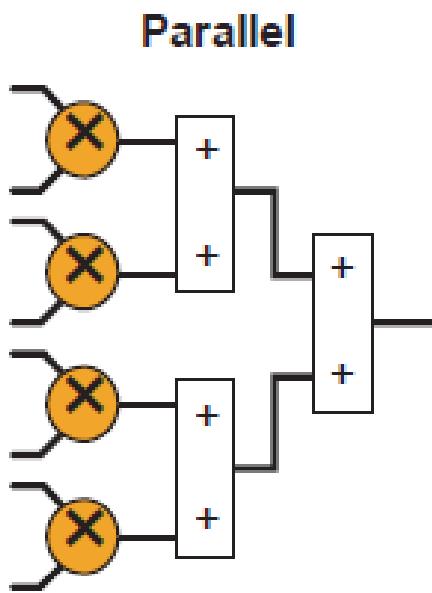
Conventional DSP Processor – Serial implementation



FPGA – Fully parallel implementation



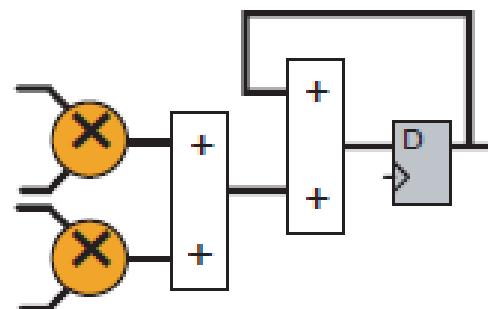
# FPGA as DSP



Device Area: 4A  
MMAC/s: 600

HIGH SPEED

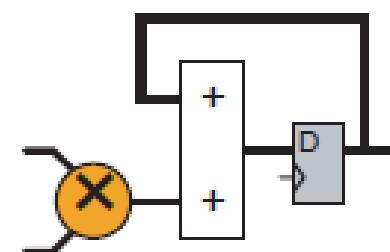
## Semi-Parallel



Device Area: 2A  
MMAC/s: 300

OPTIMIZED FOR

## Serial



Device Area: 1A  
MMAC/s: 150

SMALL AREA





# Spartan 3 resources for DSP

<b>Spartan-3 Silicon Features</b>	<b>Customer Benefits</b>
Embedded 18x18 Multipliers	Area efficient implementation of multiply-accumulate function
Distributed RAM	Local storage for DSP coefficients, small FIFOs
Shift Register Logic	16-bit Shift Register ideal for capturing high-speed or burst mode data and to store data in DSP applications
Up to 104 18 Kb Block RAM	Video line buffers, cache tag memory, scratch-pad memory, packet buffers, large FIFOs

# Virtex 7 resources for DSP

- Even 1800 DSP48E1 slices with 1.33 teraflops performance!!!

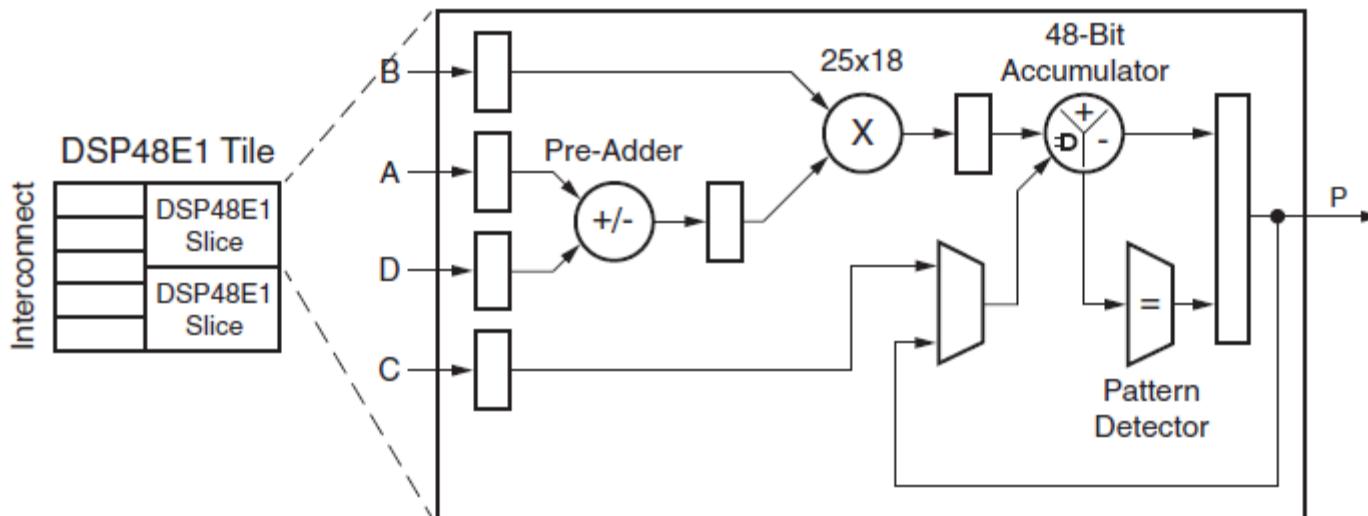


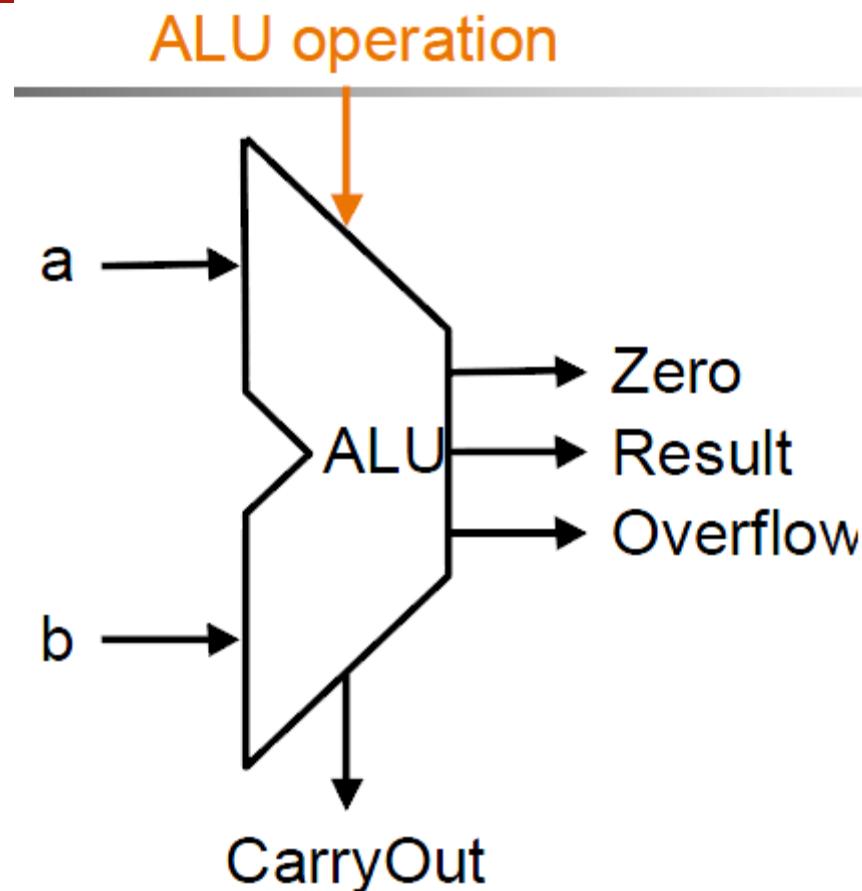
Table 3: DSP Function Effective Costs in Spartan-3 Devices<sup>(1)</sup>

Functions	% of the XC3S1000 Device Utilized	Effective Cost <sup>(2)</sup> (50K Units)	Key Specification	Other Specifications
1024-point complex FFT	24.1%	\$3.23	20 µs transform	20 µs transform, burst I/O, 16-bit input and phase factor
Single channel 64-tap FIR filter	3.0%	\$0.41	8.1 MSPS	16-bit data and co-efficient, MAC implementation, 8.1 MSPS
Digital down converter per channel	18.6%	\$2.49	Sample rate 100 MSPS	
Digital up converter per channel	18.6%	\$2.49	Sample rate 100 MSPS	
Viterbi decoder	37.8%	\$5.06	1.9 MSPS per channel	Parallel mode, trace-back 42, constraint length = 7, 32-channel, 1.9 MSPS per channel
Reed Solomon G.709 encoder	1.3%	\$0.17	120 MHz	
Reed Solomon G.709 decoder	6.9%	\$0.92	60 MHz	



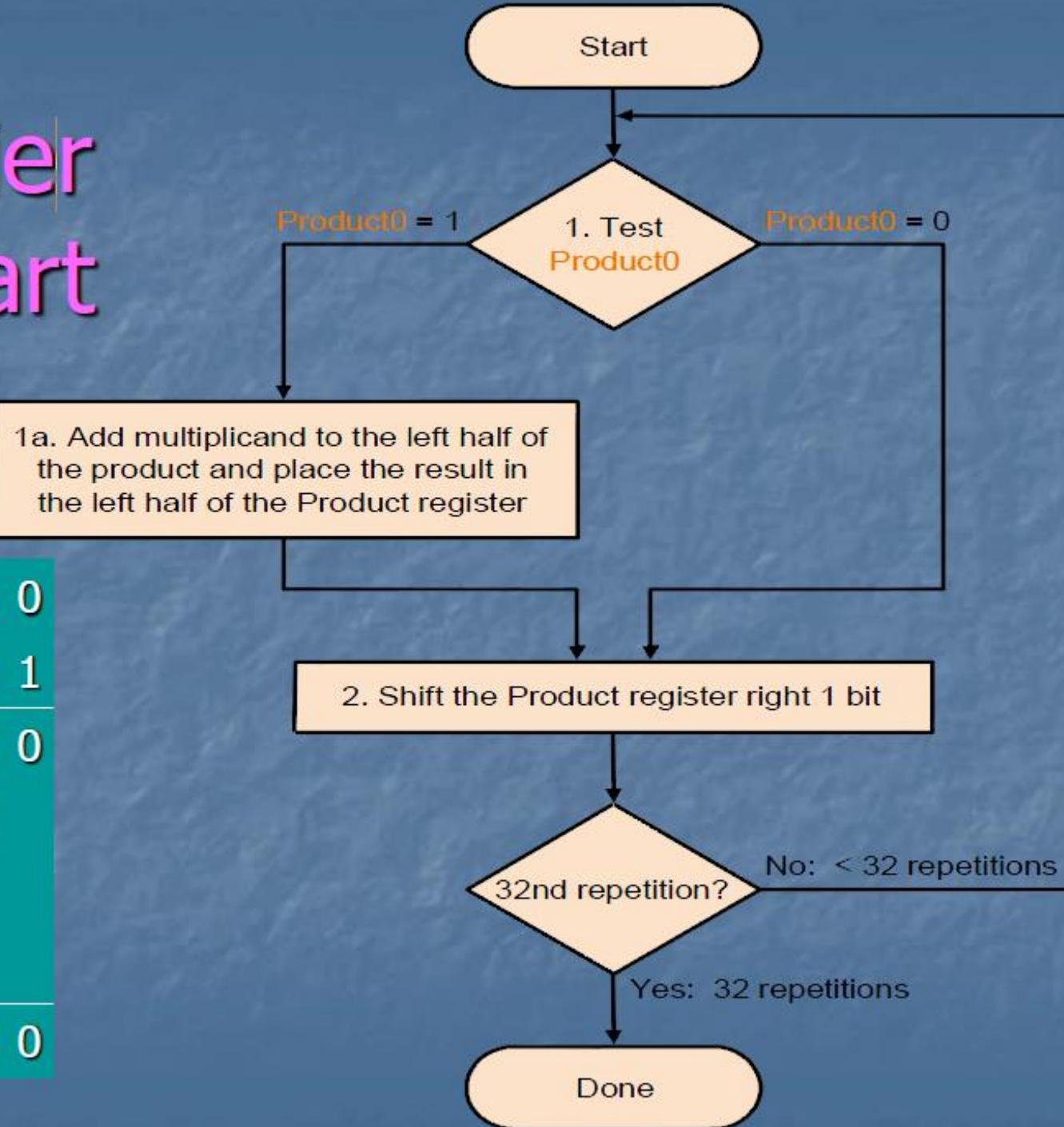
# Fixed point multipliers

# Arithmetic Logic Unit



# Multiplier flowchart

$$\begin{array}{r} 1\ 0\ 0\ 0 \\ \times 1\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0 \\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 0\ 1\ 0\ 0\ 0 \end{array}$$



# MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

Multiplicand

$$\begin{array}{r} 1000 \\ 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ \hline 1000 \\ \hline 01001000 \end{array}$$

Multiplier

## Product

- $m$  bits  $\times$   $n$  bits =  $m+n$  bit product
- Binary makes it easy:
  - 0 => place 0 ( 0  $\times$  multiplicand)
  - 1 => place a copy ( 1  $\times$  multiplicand)
  - successive refinement



# Simultaneous multiplication

	X2	X1	X0	
	Y2	Y1	Y0	
	X2*Y0	X1*Y0	X0*Y0	
	X2*Y1	X1*Y1	X0*Y1	
X2*Y2	X1*Y2	X0*Y2		
P4	P3	P2	P1	P0



# Multipliers architecture

- Serial multipliers
  - Adding series of partial products
  - Successive addition algorithm used
  - Small chip area but low speed due to sequential execution
- Parallel multipliers
- Serial-parallel multipliers



# Parallel multipliers

- Array multipliers:
  - Braun multiplier
  - Booth multiplier
  - Modified Booth multiplier
  - Baugh-Wooley multiplier
- Tree multipliers:
  - Wallace Tree multiplier
  - Hitachi's multiplier



# Parallel multipliers

- Parallel multipliers are high-performance setups
- Main disadvantage is large silicon area occupied and large power consumption
- Two main types:
  - Array type - more regular setup
  - Tree type - generally faster

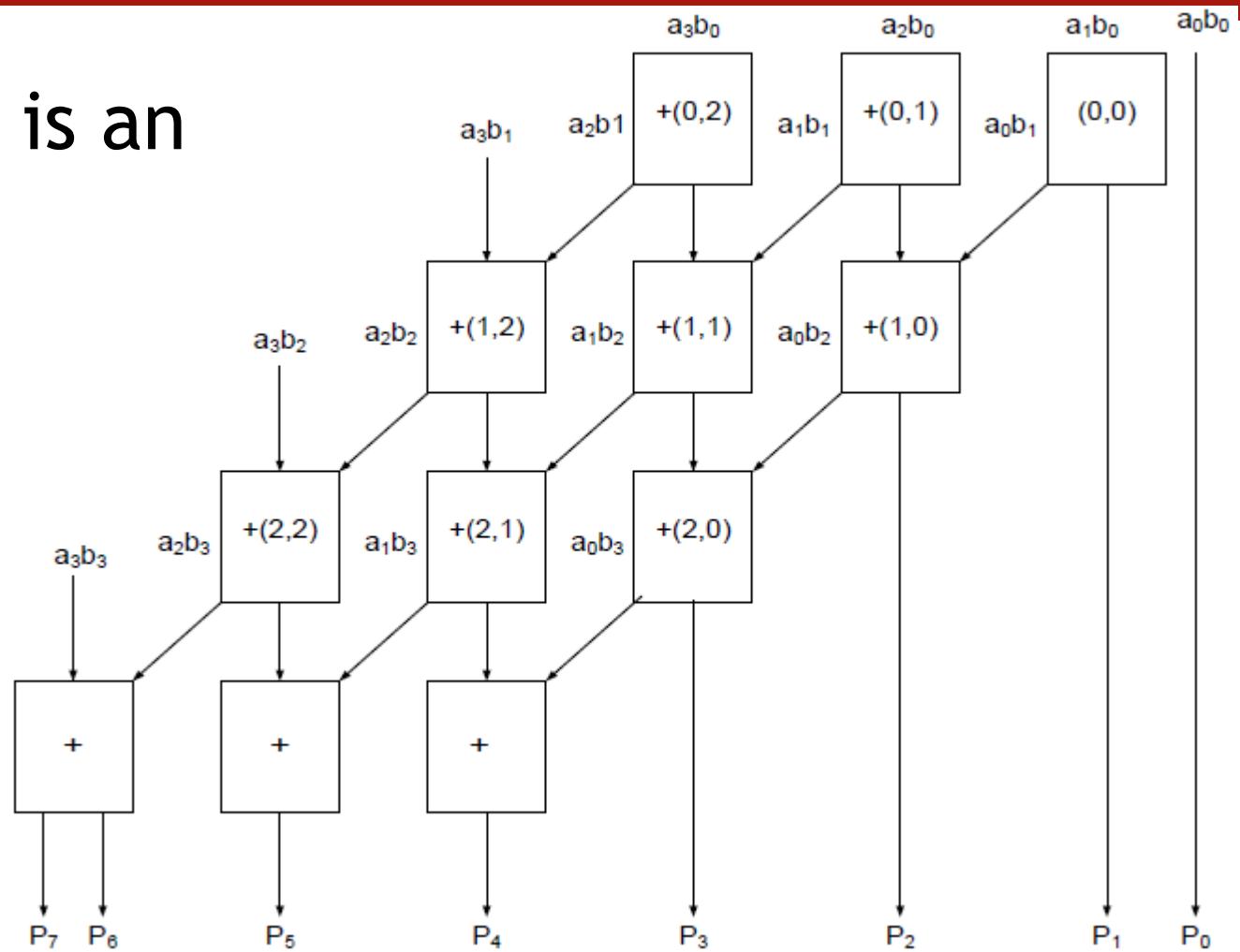


# Braun array multiplier

- Simple parallel multipliers know also as Carry Save Array multiplier
- Consists of array of AND gates and adders arranged in iterative that does not require logic registers
- An  $n \times n$  bit multiplier requires  $n(n-1)$  adders and  $n^2$  AND gates

# Braun array multiplier (4bx4b)

- Each block is an adder





# Braun array multiplier

- Multiplier circuit is based on add and shift algorithm
- Each partial product is generated by the multiplication of the multiplicand with one multiplier bit
- The partial products are shifted according to their bit orders and then added
- Addition is done by Carry Save Algo in which every carry and sum signal is passed to the adders of the next stage



# Braun array multiplier

- Advantages:
  - Regular structure
  - Easy layout and small size
  - Ease of design also for pipelined architecture
- Limitations:
  - Size
  - Arrays grow in size at the rate equal to the square of the operand size



# Booth array multiplier

- Booth multiplier utilises Booth encoding algorithm in order to reduce the number of partial products
- Two bits are considered at a time
- Algorithm is valid for signed and unsigned numbers



# Booth array multiplier

- Booth multiplier utilises Booth encoding algorithm in order to reduce the number of partial products
- Two bits are considered at a time
- Algorithms is valid for signed and unsigned numbers
- Mainly improves version of Modified Booth Algorithm (MBA) is used



# Booth algorithm

<b>X<sub>i</sub></b>	<b>X<sub>i-1</sub></b>	<b>Operations</b>	<b>Comments</b>	<b>Y<sub>i</sub></b>
0	0	Shift only	String of zeros	0
1	0	Sub and shift	Beg of string of ones	1
1	1	Shift only	String of zeros	0
0	1	Add and shift	End of string of ones	1



# Modified Booth Algorithm

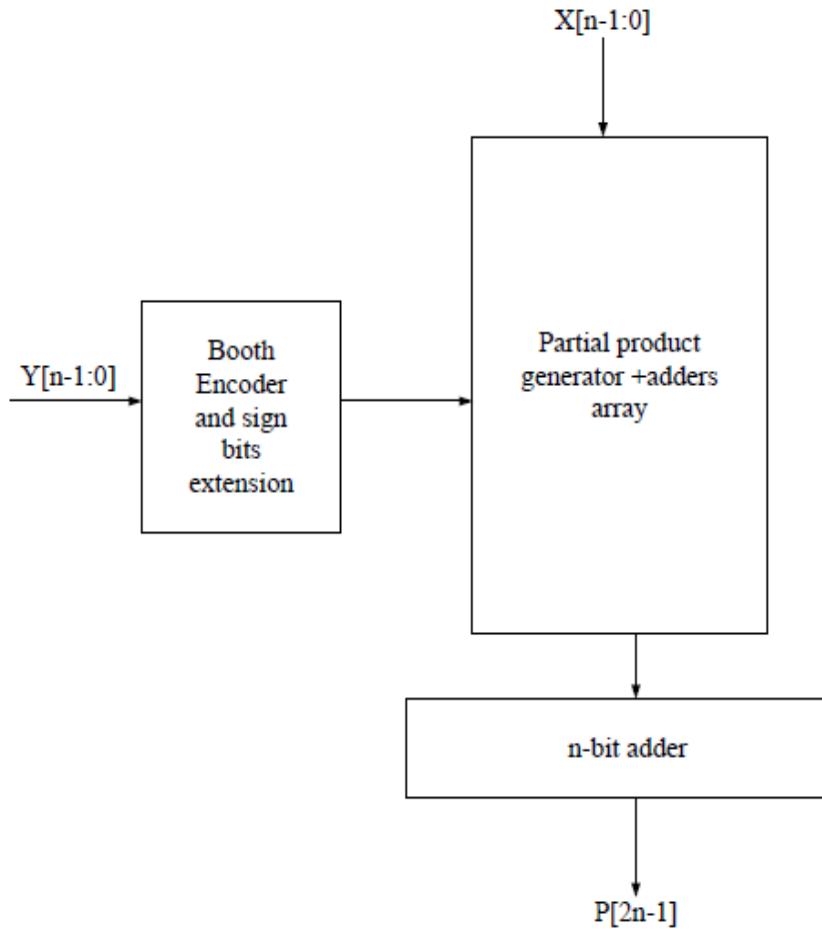
- Original Booth algo has two drawbacks:
  - The number of add/subtract operations becomes variable and becomes inconvenient in designing parallel multipliers
  - The algorithms becomes inefficient when there are isolated 1's (a lot of adding/subtracting)
- In MBA three bits are processed at a time
- Total number of cycles required to obtain a product is reduced



# Modified Booth Algorithm

$Y_{2i+1}$	$Y_{2i}$	$Y_{2i-1}$	Recoded Digit	Operand Multiplication
0	0	0	0	0*Multiplicand
0	0	1	+1	+1*Multiplicand
0	1	0	+1	+1*Multiplicand
0	1	1	+2	+2*Multiplicand
1	0	0	-2	-2*Multiplicand
1	0	1	-1	-1*Multiplicand
1	1	0	-1	-1*Multiplicand
1	1	1	0	0*Multiplicand

# Modified Booth Multiplier



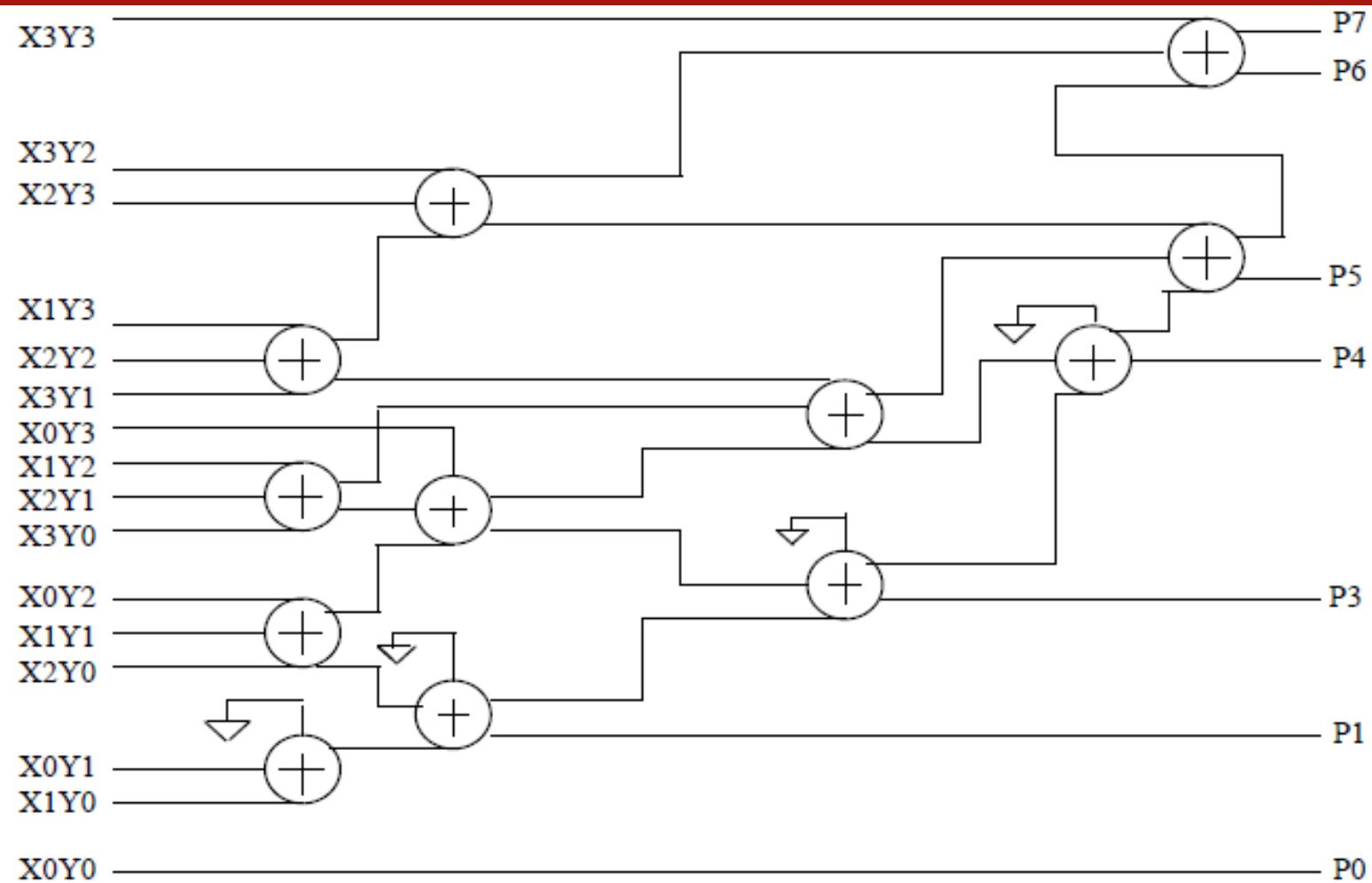
- The Booth multiplier can be designed in many ways and the design will be a trade-off between area and speed
- Most efficient is Wen-Chang version



# Tree Multipliers

- Original Booth algo has two drawbacks:
  - Extremely fast structure for summing partial products
  - Tree structures require only  $\log N$  stages to reduce  $N$  partial products by performing parallel additions
  - The number of partial products can be reduced by employing multiple input compressors capable of accumulating several partial products concurrently

# Tree Multiplier 4x4b (Wallace)





# Tree Multipliers

- Operation basics:
  - The first step for a tree multiplier is to group the summands into threes, and introduce each group into its own 3: 2 compressors, thus reducing the count of numbers by a factor of 1.5
  - The second step is to group the numbers resulting from the first step into threes and again add each group in its own 3:2 compressors
  - By continuing such steps until only two numbers remain, the addition is completed in a time proportional to the *logarithm* of the number of summands



# Multipliers comparison

Multiplier Type	Speed	Circuit Complexity	Layout	Area
Array	Low/Medium	Simple	Regular	Smallest
Booth	High	Complex	Irregular	Medium
Wallace	Higher	Medium	More irregular	Large
Booth Wallace	Highest	More complex	Medium regularity	Largest

	Array multiplier	Booth multiplier	Wallace multiplier	Booth Wallace multiplier
Area(LUTs)	3456	4067	4378	4167
Delay(ns)	36.34	22.45	15.76	13.38
Power(mW)	89	98	108	102



# Fixed point package



# Fixed point package

- Fixed-point math is basically integer math with numbers that can be less than 1.0.
- A fixed-point number has an assigned width and an assigned location for the decimal point.
- Since it is based on integer math it is extremely efficient as long as the data does not vary too much in magnitude



# Fixed point package

- The fixed-point math packages are based on the VHDL 1076.3 numeric\_std package
- This package defines two new types “ufixed” which is unsigned fixed point, and “sfixed” which is signed fixed point



# Fixed point package

- This data type uses a negative index to show where the decimal point is.
- The decimal point is assumed to be between the "0" and "-1" index.
- Thus if we can assume "signal y : ufixed (4 downto 5)" as the data type (unsigned fixed point, 10 bits wide, 5 bits of decimal), then
  - $y = 6.5 = "00110.10000"$ , or simply:
  - $y <= "01011010000"$ ;



# Fixed point package

- Any non-zero index range is valid. Thus:
- signal z : ufixed (-2 downto -3);
- $z \leq "11"; \text{ -- } 0.375 = 0.011$
- signal x : sfixed (4 downto 1);
- $y \leq "111"; \text{ -- } -2 = 1110.0$



# Fixed point package

- Unsigned Example:
  - signal x : ufixed ( 7 downto -3);
  - signal y : ufixed ( 2 downto -9);
  - If we multiply x by y we would get a signal which would be:
    - $x * y = \text{ufixed} (7+2+1 \text{ downto } -3+(-9))$  or  $\text{ufixed} (10 \text{ downto } -12)$ ;
- Signed Example:
  - signal x : sfixed (-1 downto -3);
  - signal y : sfixed (3 downto 1);
  - If we divide x by y we would get a signal which would be:
    - $x/y = \text{sfixed} (-1, 1 \text{ downto } -3, 3)$  or  $\text{sfixed} (-3 \text{ downto } -6)$ ;



# Fixed point package

- The following operations are defined for ufixed/sfixed:
  - +, -, \*, /, rem, mod, =, /=, <, >, >=, <=, sll, srl, rol, ror, sla, sra
- The following functions are defined for ufixed:
  - divide, reciprocal, scalb, maximum, minimum, find\_lsb, find\_msb, resize, To\_01, Is\_X,
- Conversion functions are defined for ufixed:
  - to\_ufixed (natural), to\_ufixed (real), to\_ufixed (unsigned), to\_ufixed(signed), remove\_sign (sfixed), to\_unsigned, to\_real, to\_integer, to\_UFix



# Floating point package



# Floating point package

- The 32 bit floating point package looks like the following:
  - package fphdl32\_pkg is new IEEE.fphdl\_pkg
  - generic map (
    - fp\_fraction\_width => 23; -- 23 bits of fraction
    - fp\_exponent\_width => 8; -- exponent 8 bits
    - fp\_round\_style => round\_nearest; -- round nearest algorithm
    - fp\_denormalize => true; -- Turn on Denormalized numbers
    - fp\_check\_error => true; -- Turn on NAN and overflow processing
    - fp\_guard\_bits => 3); -- number of guard bits



# Floating point package

- Package generics allow to specify any data width or size of floating point number.
- The resulting data type will be called “fp”.
  - signal a, b, c : fp;
  - signal x : unsigned (5 downto 0);
  - constant PI : real := 3.14;
  - begin
  - b <= to\_fp (x);
  - c <= a + PI;



# Floating point package

- The actual floating-point type is defined as follows:
  - type fp is array (fp\_exponent\_width downto - fp\_fraction\_width) of STD\_LOGIC;
- For a 32-bit representation that specification makes the number look as follows:
  - 0 00000000 000000000000000000000000
  - 8 7 0 -1 -23
  - +/- exp. fraction
- where the sign is bit 8, the exponent is contained in bits 7-0 (8 bits) with bit 7 being the MSB, and the mantissa is contained in bits -1 - -23 (32 - 8 - 1 = 23 bits) where bit -1 is the MSB.



# Floating point package

- Defined operations for floating point numbers are:
- Unary -, abs, “+”, “-“, “\*”, “/”, “rem”, “mod”, “=”, “/=”, “<”, “>”, “<=”, “>=”
- The non floating-point type is first converted into floating point and the operation is performed.
- Defined functions for floating point number are:
  - dividbyp2 (divide by a power of 2), reciprocal (1/x), maximum, minimum, to\_unsigned, to\_signed, to\_ufixed, to\_sfixed, to\_real, to\_integer , To\_fp(SIGNED), To\_fp(UNSIGNED), To\_fp(ufixed), To\_fp(sfixed), To\_fp(integer), To\_fp(real)



CORDIC



# CORDIC

- CORDIC ( COordinate Rotation Digital Computer)
- Introduced in 1959 by Jack E. Volder
- Efficient to compute sin, cos, tan, sinh, cosh, tanh
- Its a Hardware Efficient Algorithm
- Iterative Algorithm for Circular Rotation
- ***No Multiplication***
- Delay/Hardware cost comparable to division or square rooting.



# Why CORDIC

- How to evaluate trigonometric functions?
  - Table lookup
  - Polynomial approximations
  - CORDIC
- Compared to other approaches, CORDIC is a clear winner when :
  - Hardware Multiplier is unavailable (eg. microcontroller)
  - Not enough hardware multipliers available (eg. FPGA)

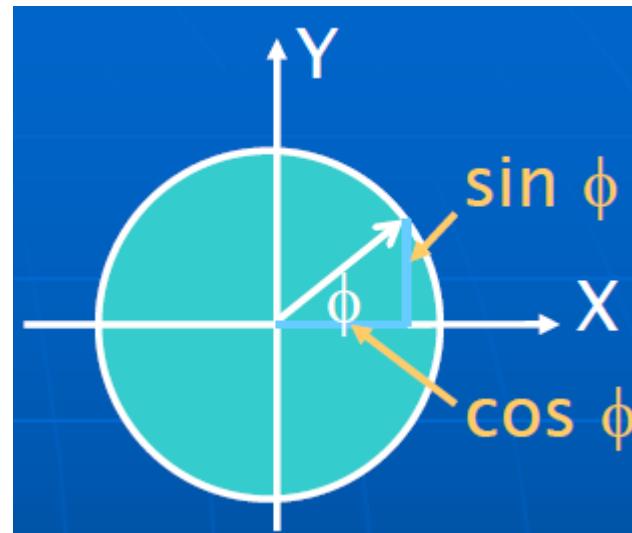


# Basic Ideas

- Embedding of elementary function evaluation as a generalized rotation operation.
- Decompose rotation operation into successive basic rotations.
- Each basic rotation can be realized with *shift-and-add* arithmetic operations.

# CORDIC principles

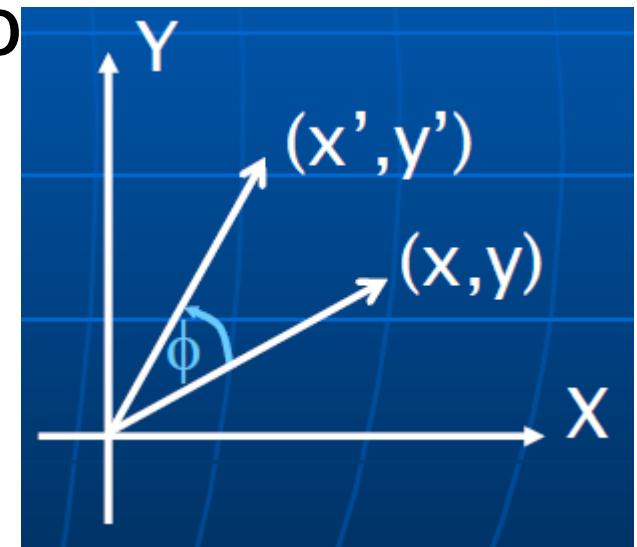
- Basic idea:
  - Rotate  $(1,0)$  by  $\varphi$  degrees to get  $(x,y)$ :
    - $x=\cos(\varphi)$ ,
    - $y=\sin(\varphi)$



# CORDIC principles

- Rotation of any  $(x,y)$  vector

- $x' = x \cdot \cos(\phi) - y \cdot \sin(\phi)$
- $y' = y \cdot \cos(\phi) + x \cdot \sin(\phi)$

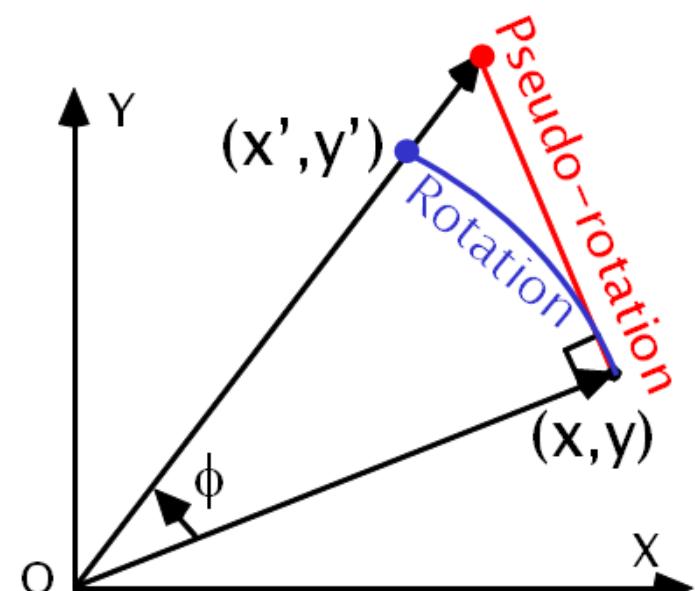


- Hence:

- $x' = \cos(\phi) * [x - y \cdot \tan(\phi)]$
- $y' = \cos(\phi) * [y + x \cdot \tan(\phi)]$

# CORDIC principles

- Two components:
  - $\cos(\varphi)$ :
    - Reduces the magnitude of the vector
    - If not used -> Pseudo-rotation
  - $\tan(\varphi)$ :
    - Rotates the vector
    - Break  $\varphi$  into a series of successively shrinking angles  $\alpha_i$  such that :
      - $\tan(\alpha_i)=2^{-i}$



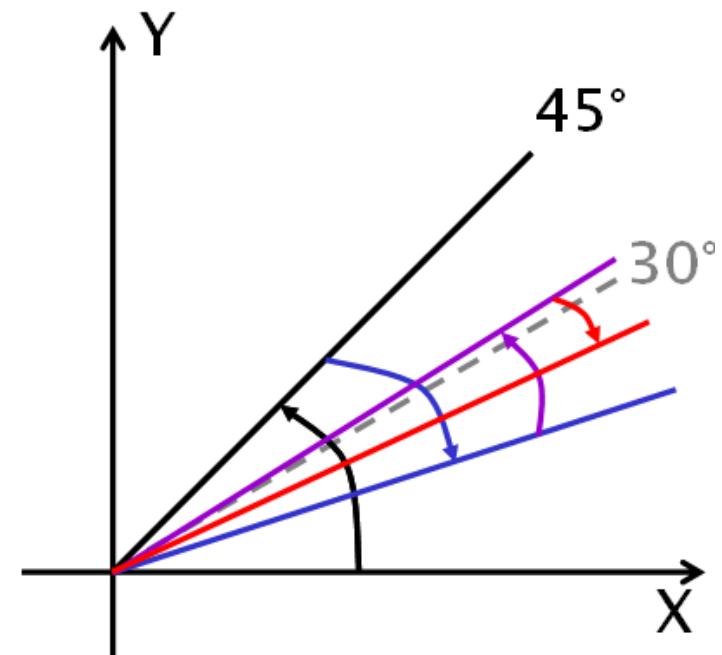
# CORDIC principles

- Precomputation of  $\tan(\alpha_i)$ :

i	$\alpha_i$	$\tan(\alpha_i)$	
0	45.0°	1	$= 2^{-0}$
1	26.6°	0.5	$= 2^{-1}$
2	14.0°	0.25	$= 2^{-2}$
3	7.1°	0.125	$= 2^{-3}$
4	3.6°	0.0625	$= 2^{-4}$
5	1.8°	0.03125	$= 2^{-5}$
6	0.9°	0.015625	$= 2^{-6}$
7	0.4°	0.0078125	$= 2^{-7}$
8	0.2°	0.00390625	$= 2^{-8}$
9	0.1°	0.001953125	$= 2^{-9}$

# Example

- Example:  $\phi = 30.0^\circ$ 
  - Start with  $\alpha_0 = 45.0$  ( $> 30.0$ )
  - $45.0 - 26.6 = 18.4$  ( $< 30.0$ )
  - $18.4 + 14.0 = 32.4$  ( $> 30.0$ )
  - $32.4 - 7.1 = 25.3$  ( $< 30.0$ )
  - $25.3 + 3.6 = 28.9$  ( $< 30.0$ )
  - $28.9 + 1.8 = 30.7$  ( $> 30.0$ )
  - ...
  - $\phi = 30.0$   
 $\approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6$   
 $+ 1.8 - 0.9 + 0.4 - 0.2 + 0.1$   
 $= 30.1$





# Rotation reduction

- Two components:
  - $\cos(\varphi)$ :
    - Reduces the magnitude of the vector
    - If not used -> Pseudo-rotation
  - $\tan(\varphi)$ :
    - Rotates the vector
    - Break  $\varphi$  into a series of successively shrinking angles  $\alpha_i$  such that :
      - $\tan(\alpha_i) = 2^{-i}$

# Rotation Reduction

- Rewrite in terms of  $\alpha_i$ : ( $0 \leq i \leq n$ )

$$x_{i+1} = \cos(\alpha_i)[x_i - y_i d_i \tan(\alpha_i)]$$



$$y_{i+1} = \cos(\alpha_i)[y_i + x_i d_i \tan(\alpha_i)]$$

$$x_{i+1} = K_i[x_i - y_i d_i 2^{-i}]$$

$$y_{i+1} = K_i[y_i + x_i d_i 2^{-i}]$$

- Where:

$$K_i = \cos(\alpha_i) = \cos(\tan^{-1}(2^{-i}))$$

Note :

$$\cos(\alpha_i) = \cos(-\alpha_i)$$

$$d_i = \pm 1$$

- $d_i$  - rotate direction
- K - magintude



# Taking Care of the Magnitude

$$x_{i+1} = K_i \cdot [x_i - y_i \cdot d_i \cdot 2^{-i}]$$

$$y_{i+1} = K_i \cdot [y_i + x_i \cdot d_i \cdot 2^{-i}]$$

- Observations:

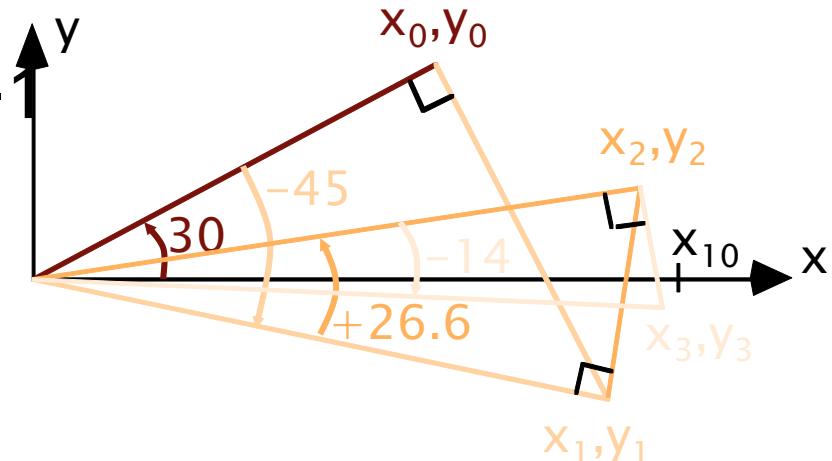
- We choose to always use ALL  $\alpha_j$  terms, with +/- signs
- $K_j = \cos(\alpha_j) = \cos(-\alpha_j)$
- At each step, we multiply by  $\cos(\alpha_j)$  [constant?]

- Let the multiplications aggregate to:

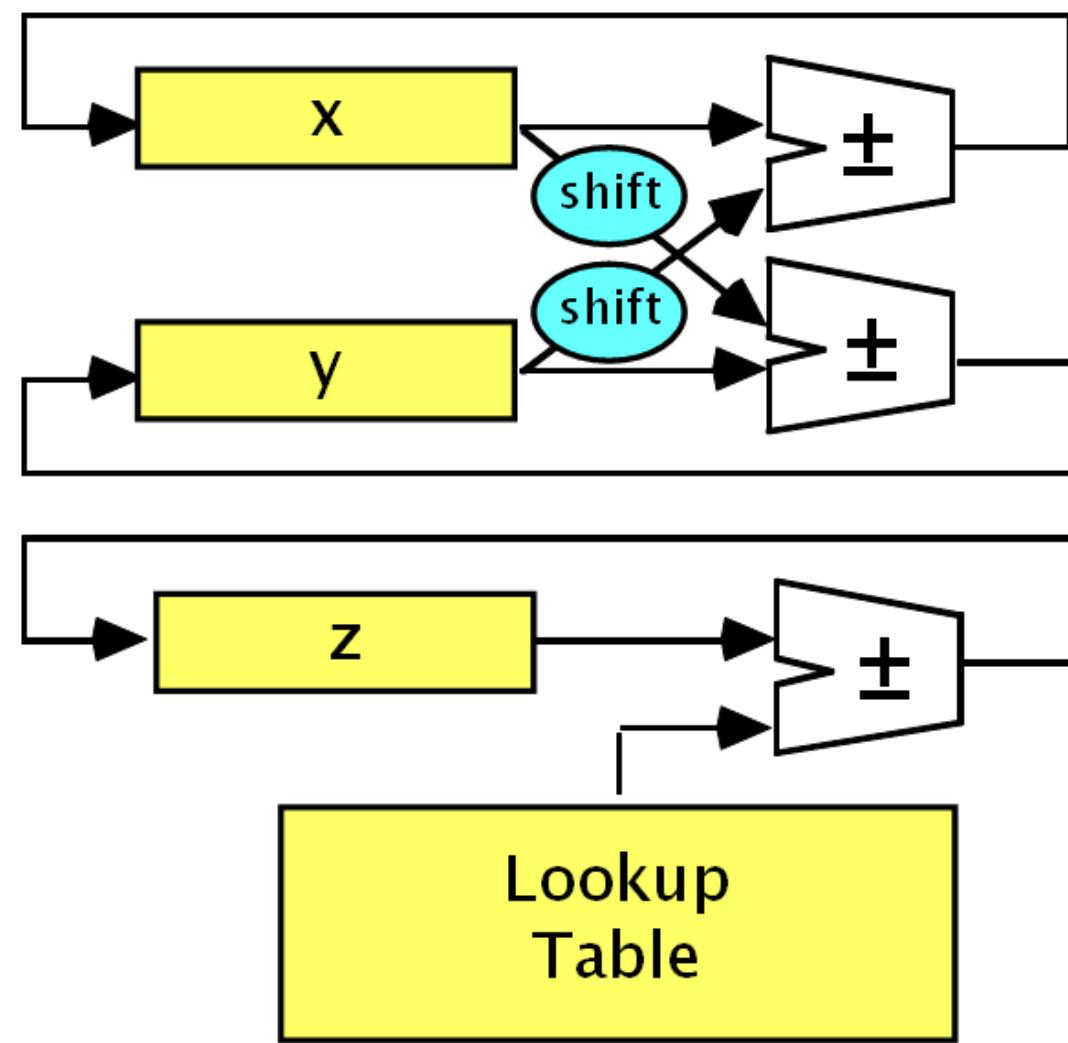
$$K = \prod_{i=0}^n K_i \quad n \rightarrow \infty, K = 0.607\ 252\ 935\dots$$

# Hardware Realization: CORDIC Rotation Mode

- Algorithm: (z is the current angle)
  - Mode: rotation: “at each step, try to make z zero”
  - Initialize  $x=0.607253, y=0, z=\phi$
  - For  $i=0 \rightarrow n$
  - $d_i = 1$  when  $z > 0$ , else -
  - $x_{i+1} = x_i - d_i \cdot 2^{-i} \cdot y_i$
  - $y_{i+1} = y_i + d_i \cdot 2^{-i} \cdot x_i$
  - $z_{i+1} = z_i - d_i \cdot \alpha_i$
  - Result:  $x_n = \cos(\phi), y_n = \sin(\phi)$
  - Precision: n bits ( $\tan^{-1}(2^{-i}) \approx 2^{-i}$ )

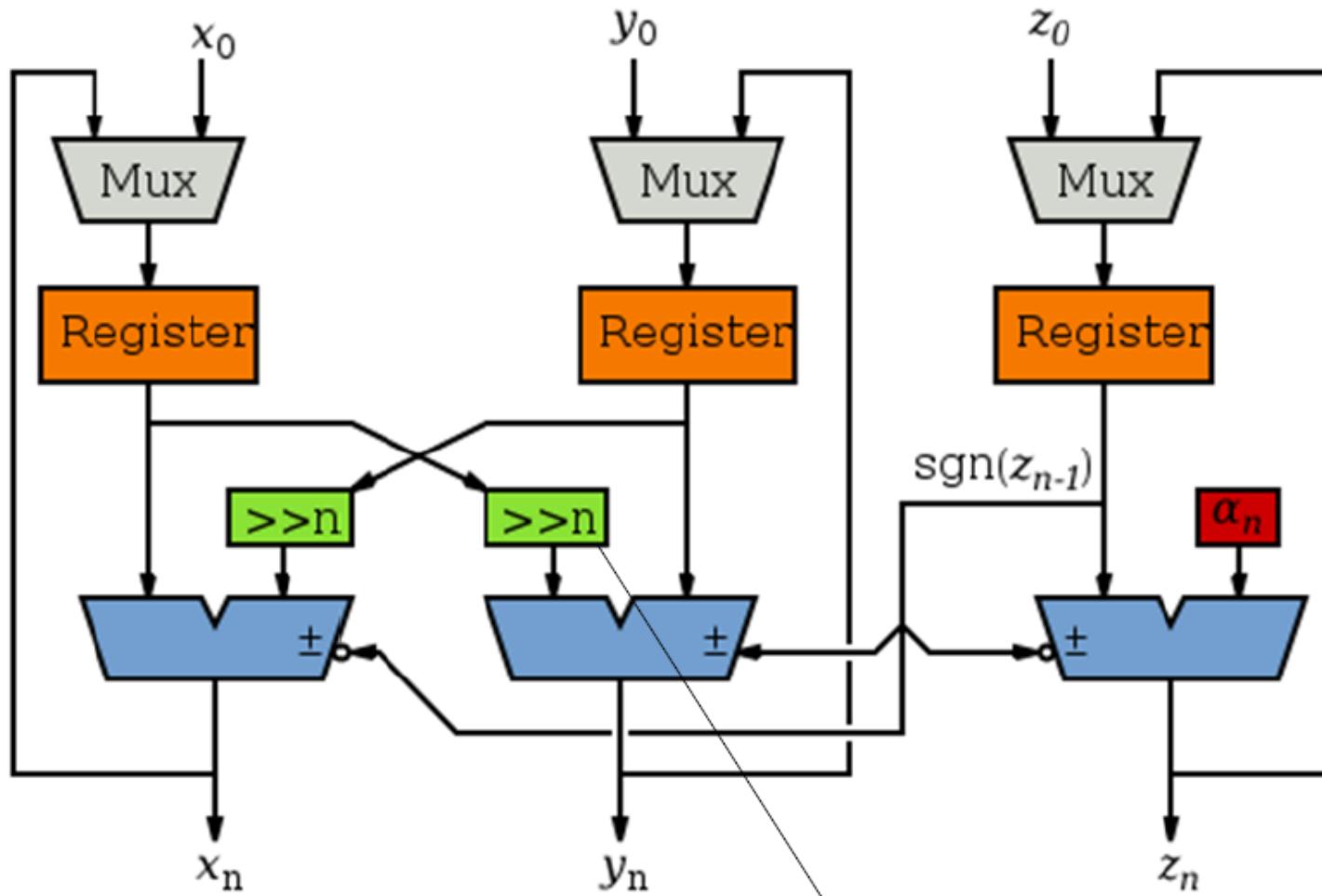


# CORDIC hardware

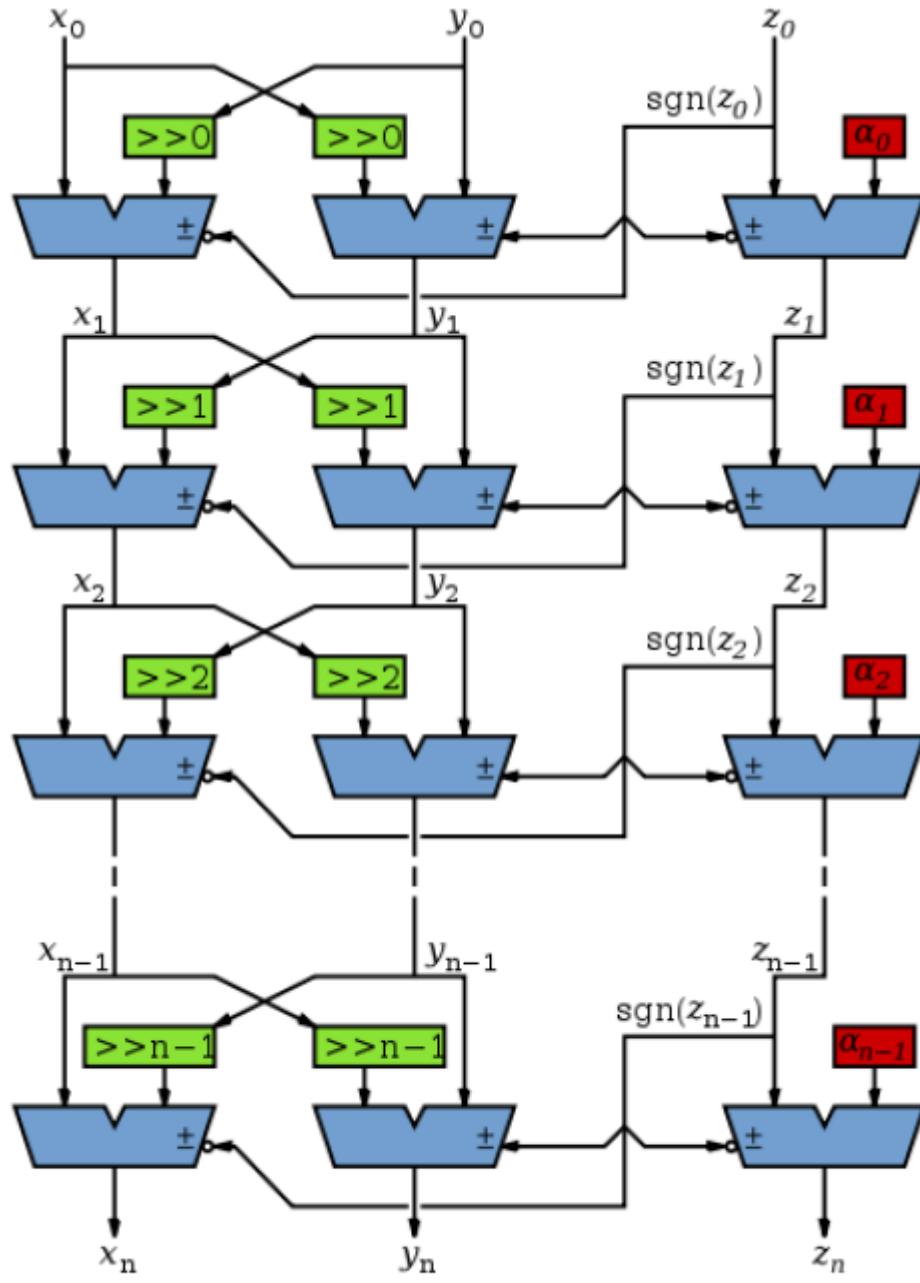




# CORDIC Hardware (Bit-parallel, iterative)



- Lower throughput ( $n$  times less)
- The barrel-shifter is variable and costs logic
- $a_n$  is stored in a small register file



# CORDIC hardware

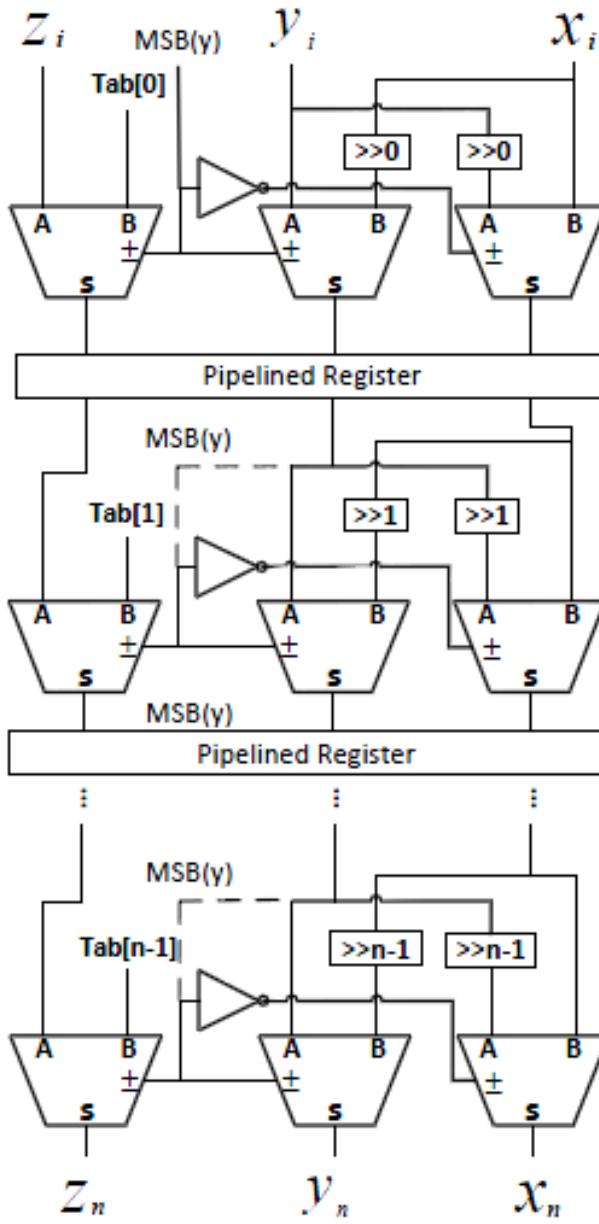
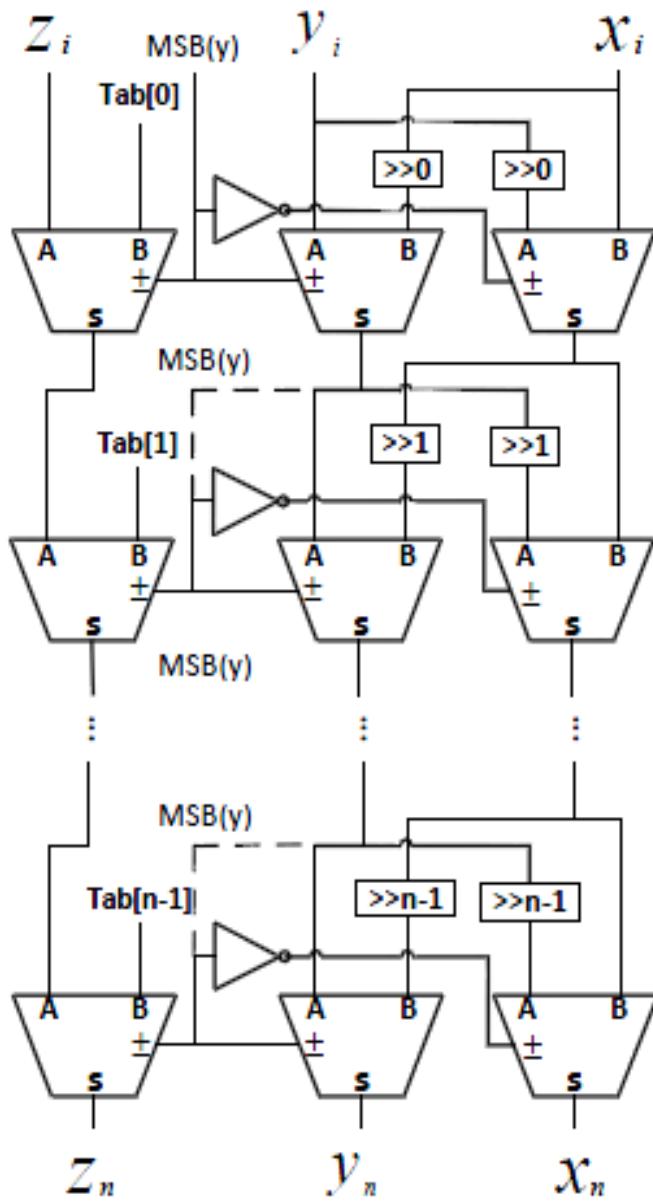
bit-parallel,  
unrolled)



# CORDIC hardware (bit-parallel, unrolled)

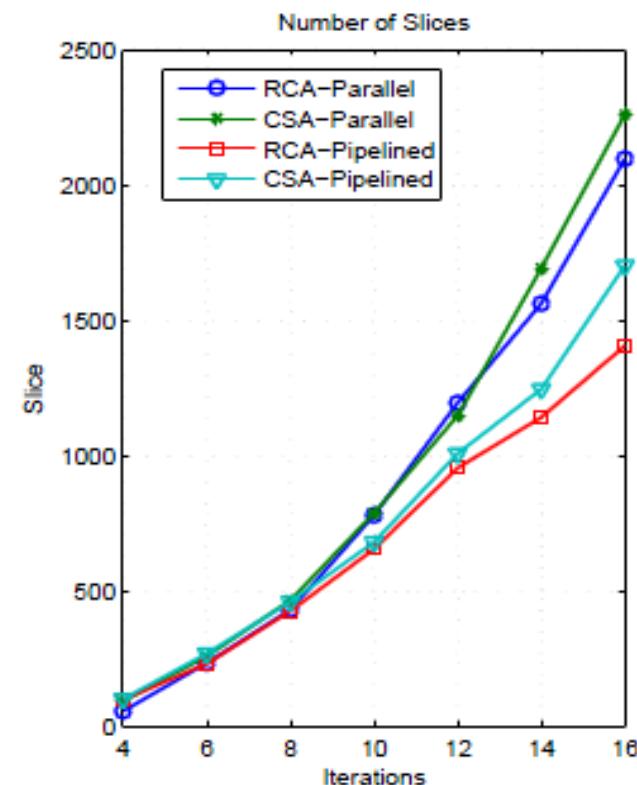
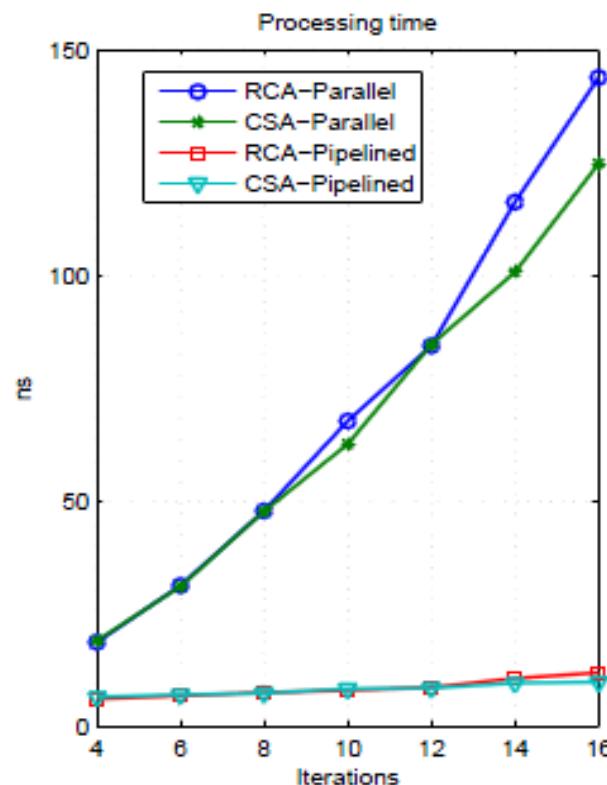
- Implementation cost: three ADD/SUB ALUs per iteration.
- Shift operations: hardwired
- rotate angles ( $a_i$ ) are fit into the logic
- Typically with pipeline register after each iteration (results in very high throughput)
- Improvement of the angle resolution by almost one signal bit iteration.

# Parallel vs pipelined



# CORDIC hardware cost (XC3s500)

- Processing time and slices used (RCA, CSA - different adders)





# CORDIC applications

- Directly computes:
  - $\sin$ ,  $\cos$ ,  $\sinh$ ,  $\cosh$
  - $\tan^{-1}$ ,  $\tanh^{-1}$
  - Division, multiplication
- Also directly computes:
  - $\tan^{-1}(y/x)$
  - $y + x^*z$
  - $\sqrt{x^2+y^2}$
  - $\sqrt{x^2-y^2}$
  - $e^z = \sinh(z) + \cosh(z)$



# CORDIC Applications

- Indirectly computes:

$$\tan z = \frac{\sin z}{\cos z}$$

$$\tanh z = \frac{\sinh z}{\cosh z}$$

$$\ln w = 2 \tanh^{-1} \left| \frac{w-1}{w+1} \right|$$

$$\log_b w = K \cdot \ln w$$

$$w^t = e^{t \ln w}$$

$$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1-w^2}}{w}$$

$$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1-w^2}}$$

$$\cosh^{-1} w = \ln \left( w + \sqrt{1-w^2} \right)$$

$$\sinh^{-1} w = \ln \left( w + \sqrt{1+w^2} \right)$$

$$\sqrt{w} = \sqrt{(w+1/4)^2 - (w-1/4)^2}$$



Thank you for your attention



# References

- [1] [http://dspace.thapar.edu:8080/dspace/bitstream/10266/1683/1/Simran\\_thesis.pdf](http://dspace.thapar.edu:8080/dspace/bitstream/10266/1683/1/Simran_thesis.pdf)
- [2] <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.45.6568&rep=rep1&type=pdf>
- [3] [http://teal.gmu.edu/courses/ECE645/projects\\_S06/talks/CORDIC.pdf](http://teal.gmu.edu/courses/ECE645/projects_S06/talks/CORDIC.pdf)
- [4] <http://www.eecs.tufts.edu/~ryun01/vlsi/design.htm>
- [5] <http://casdc.ee.ncku.edu.tw/class/CA/CH11.pdf>
- [6] <http://mountains.ece.umn.edu/~kia/Courses/EE5324/09-CORDIC/EE5324-CORDIC.ppt#262,7,Example: Rewriting Angles in Terms of ai>
- [7] [http://en.wikibooks.org/wiki/Digital\\_Circuits/CORDIC](http://en.wikibooks.org/wiki/Digital_Circuits/CORDIC)
- [8] [http://www.cs.ucla.edu/digital\\_arithmetic/files/ch11.pdf](http://www.cs.ucla.edu/digital_arithmetic/files/ch11.pdf)
- [9] Guerrero D., Meloni L., „Comparison of Parallel and Pipelined CORDIC



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

### Wykład 10 Mikrokontrolery w FPGA

Wrocław 2020



# Plan

- Introduction
- PicoBlaze
- MicroBlaze
- Cortex-M1



# Introduction



# Introduction

- There are dozens of 8-bit microcontroller architectures and instruction sets
- Modern FPGAs can efficiently implement practically any 8-bit microcontroller
- Available FPGA soft cores support popular instruction sets such as the PIC, 8051, AVR, 6502, 8080, and Z80 microcontrollers
- Each significant FPGA producer offers their own soft core solution



# Introduction

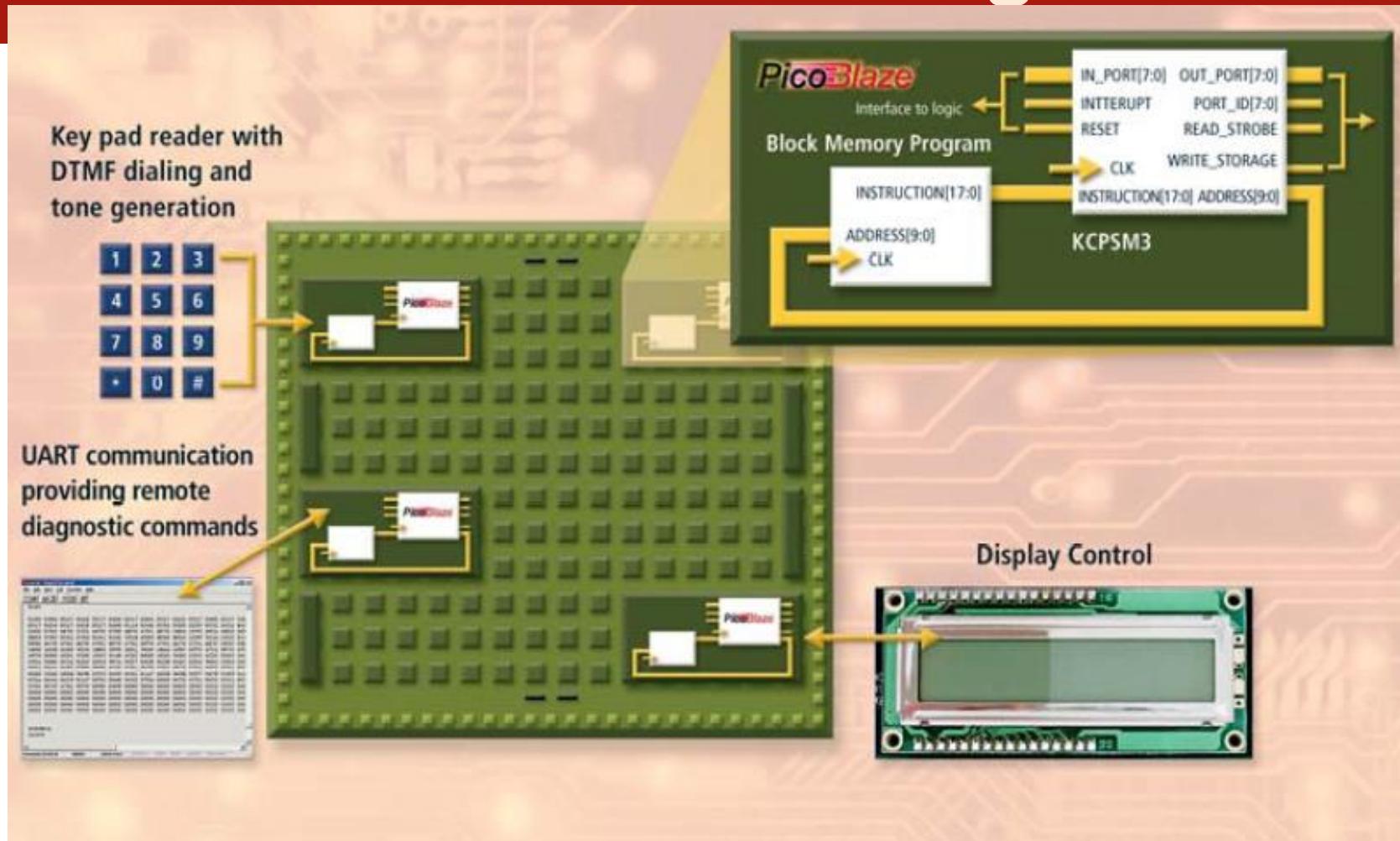
- Microcontrollers and FPGAs both successfully implement practically any digital logic function.
- Each solution has unique advantages in cost, performance, and ease of use.
- Microcontrollers are well suited to control applications, especially with widely changing requirements.
- The FPGA resources required to implement the microcontroller are relatively constant.



# Introduction

- The same FPGA logic is re-used by the various microcontroller instructions, conserving resources.
- The program memory requirements grow with increasing complexity
- Programming control sequences or state machines in assembly code is often easier than creating similar structures in FPGA logic
- Microcontrollers are typically limited by performance. Each instruction executes sequentially.

# Introduction – block diagram





# FPGA vs Soft Core

## Microcontroller:

- Easy to program, excellent for control and state machine applications
- Resource requirements remain constant with increasing complexity
- Re-uses logic resources, excellent for lower-performance functions
- Executes sequentially
- Performance degrades with increasing complexity
- Program memory requirements increase with increasing complexity
- Slower response to simultaneous inputs



# FPGA vs Soft Core

## FPGA:

- Significantly higher performance
- Excellent at parallel operations
- Sequential vs. parallel implementation tradeoffs optimize performance or cost
- Fast response to multiple, simultaneous inputs
- Control and state machine applications more difficult to program
- Logic resources grow with increasing complexity





# PicoBlaze – Key Features

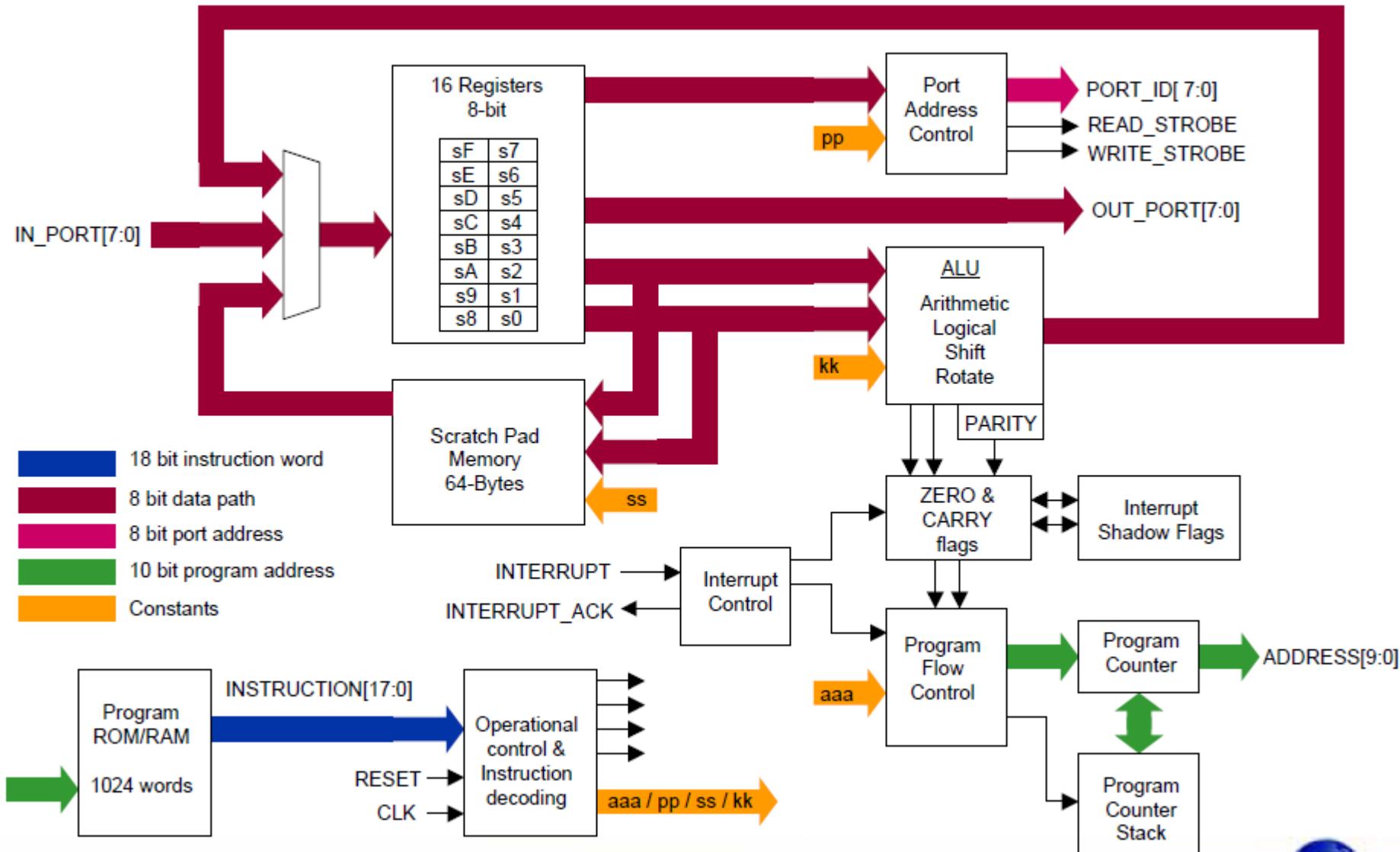
- 16 x byte-wide general-purpose data registers
- 1K instructions of programmable on-chip program store, automatically loaded during FPGA configuration
- Byte-wide Arithmetic Logic Unit (ALU) with CARRY and ZERO indicator flags
- 64-byte internal scratchpad RAM
- 256 input and 256 output ports for easy expansion and enhancement



# PicoBlaze – Key Features

- Automatic 31-location CALL/RETURN stack
- Predictable performance, always two clock cycles per instruction, up to 200 MHz or 100 MIPS in a Virtex-4™ FPGA and 88 MHz or 44 MIPS in a Spartan-3 FPGA
- Fast interrupt response; worst-case 5 clock cycles
- Assembler, instruction-set simulator support

# PicoBlaze – Block Diagram





# PicoBlaze – Performance

Feature	PicoBlaze for Spartan-3, Virtex-II/Pro and Virtex-4	PicoBlaze for Virtex-E and Spartan-II/E	PicoBlaze for CoolRunner-II
Program Space	1024	256	256
Instruction Size	18-bit	16-bit	16-bit
Internal Program	Yes	Yes	Yes
8-Bit Registers	16	16	8
Stack Depth	31	15	4
Assembler	KCPSM3	KCPSM	ASM
Size	96 Spartan-3 slices	76 Spartan-II/E slices	212 macrocells in XC2C256
Performance	44 MIPS (Spartan-3) 76 MIPS (Virtex-II) 100 MIPS (Virtex-II Pro) 100 MIPS (Virtex-4 LX, SX) 102 MIPS (Virtex-4 FX)	37 MIPS (Spartan-II/E)	21 MIPS

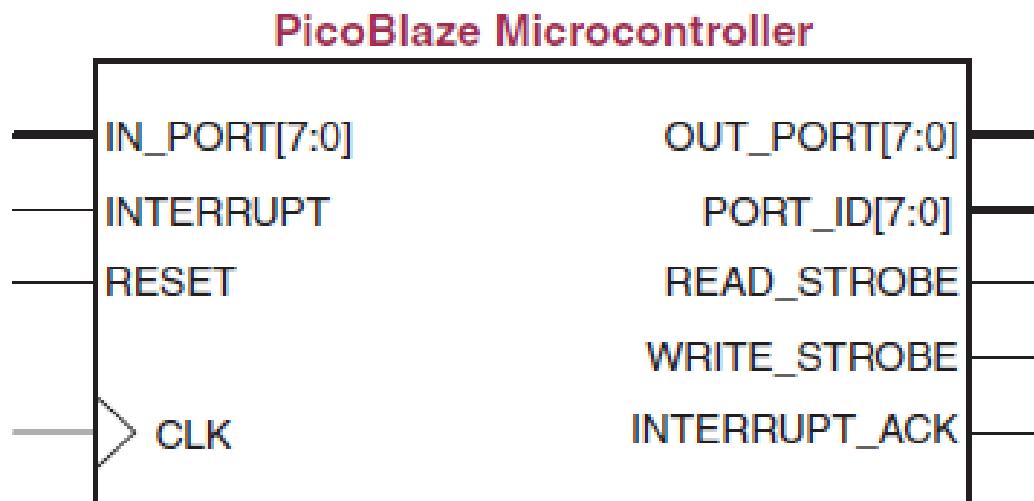
Source: [1]



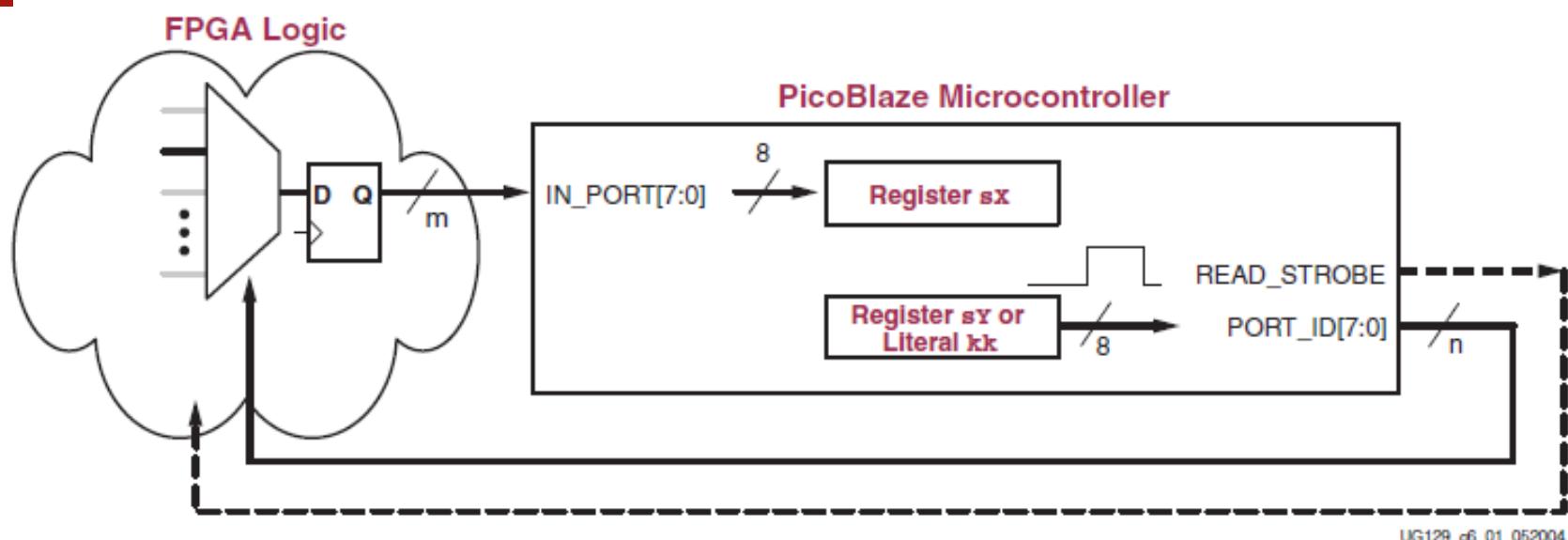
# PicoBlaze – Instruction Set

<u>Program Control</u>	<u>Logical</u>	<u>Arithmetic</u>
JUMP aaa	LOAD sX,kk	ADD sX,kk
JUMP Z,aaa	AND sX,kk	ADDCY sX,kk
JUMP NZ,aaa	OR sX,kk	SUB sX,kk
JUMP C,aaa	XOR sX,kk	SUBCY sX,kk
JUMP NC,aaa	TEST sX,kk	COMPARE sX,kk
CALL aaa	LOAD sX,sY	ADD sX,sY
CALL Z,aaa	AND sX,sY	ADDCY sX,sY
CALL NZ,aaa	OR sX,sY	SUB sX,sY
CALL C,aaa	XOR sX,sY	SUBCY sX,sY
CALL NC,aaa	TEST sX,sY	COMPARE sX,sY
	<u>Shift and Rotate</u>	<u>Storage</u>
RETURN		FETCH sX,ss
RETURN Z	SR0 sX	FETCH sX,(sY)
RETURN NZ	SR1 sX	STORE sX,ss
RETURN C	SRX sX	STORE sX,(sY)
RETURN NC	SRA sX	
	RR sX	
	SL0 sX	
	SL1 sX	
	SLX sX	
	SLA sX	
	RL sX	
<u>Input/Output</u>		<u>Interrupt</u>
INPUT sX,pp		RETURNI ENABLE
INPUT sX,(sY)		RETURNI DISABLE
OUTPUT sX,pp		ENABLE INTERRUPT
OUTPUT sX,(sY)		DISABLE INTERRUPT
		<i>All instructions execute in 2 clock cycles</i>

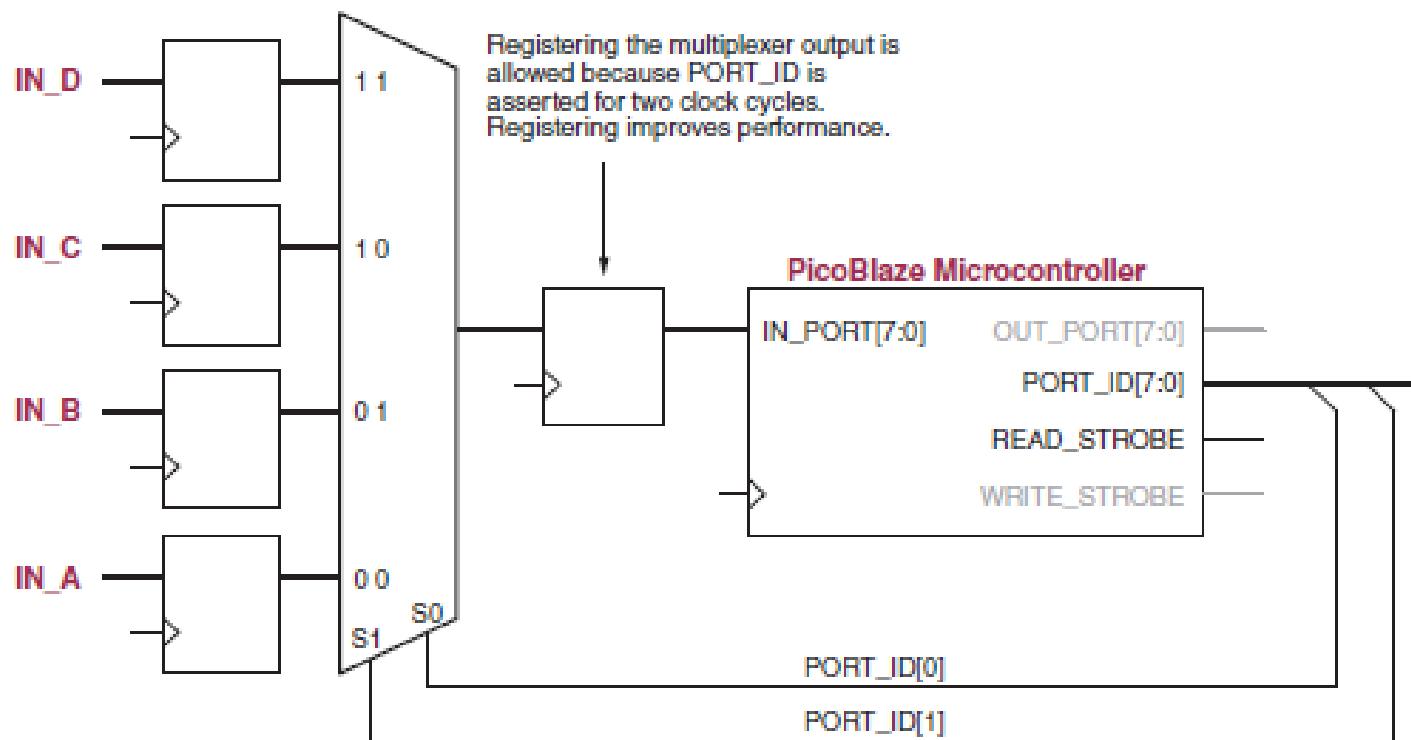
# PicoBlaze – Interface Signals



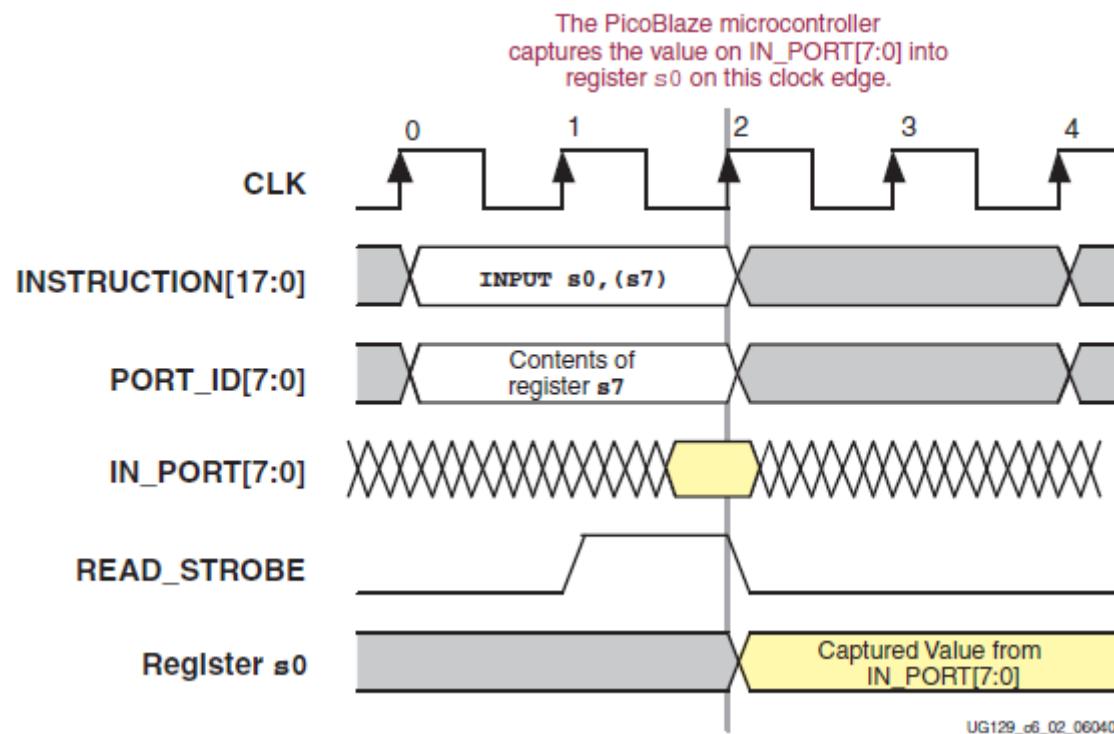
# PicoBlaze – Ports



# PicoBlaze – Ports



# PicoBlaze – Ports





# PicoBlaze – Ports

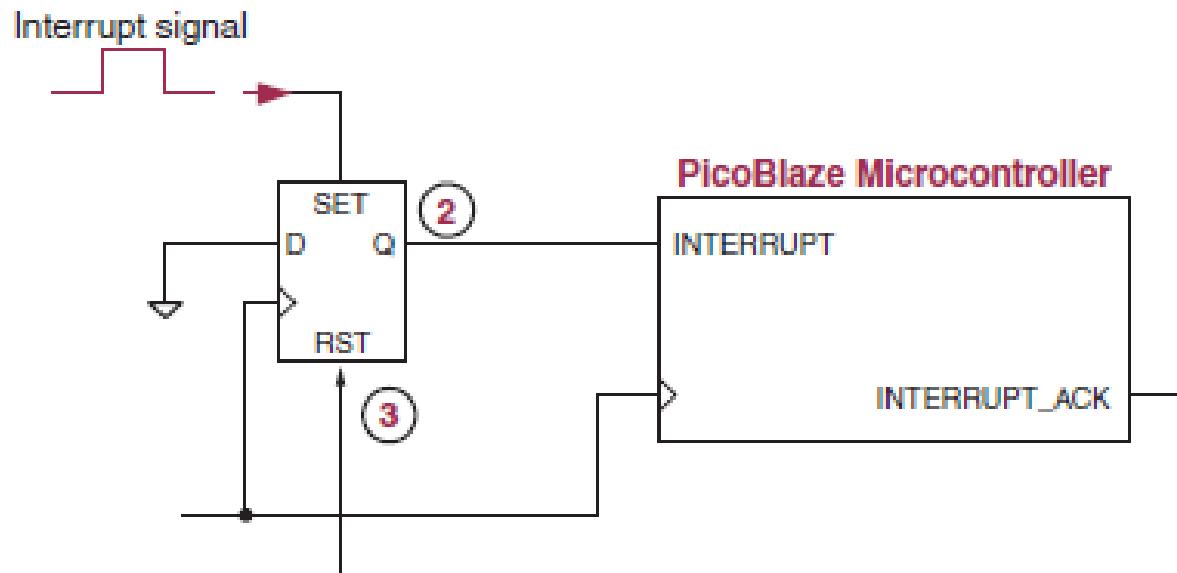
- The PicoBlaze processor provides two instructions to access input and output ports
  - **INPUT sX, sY:**
    - Set PORT\_ID to the contents of sY,
    - Read the value on IN\_PORT into register sX
  - **OUTPUT sX, sY:**
    - Set PORT\_ID to the contents of sY,
    - Write register sX to OUT\_PORT



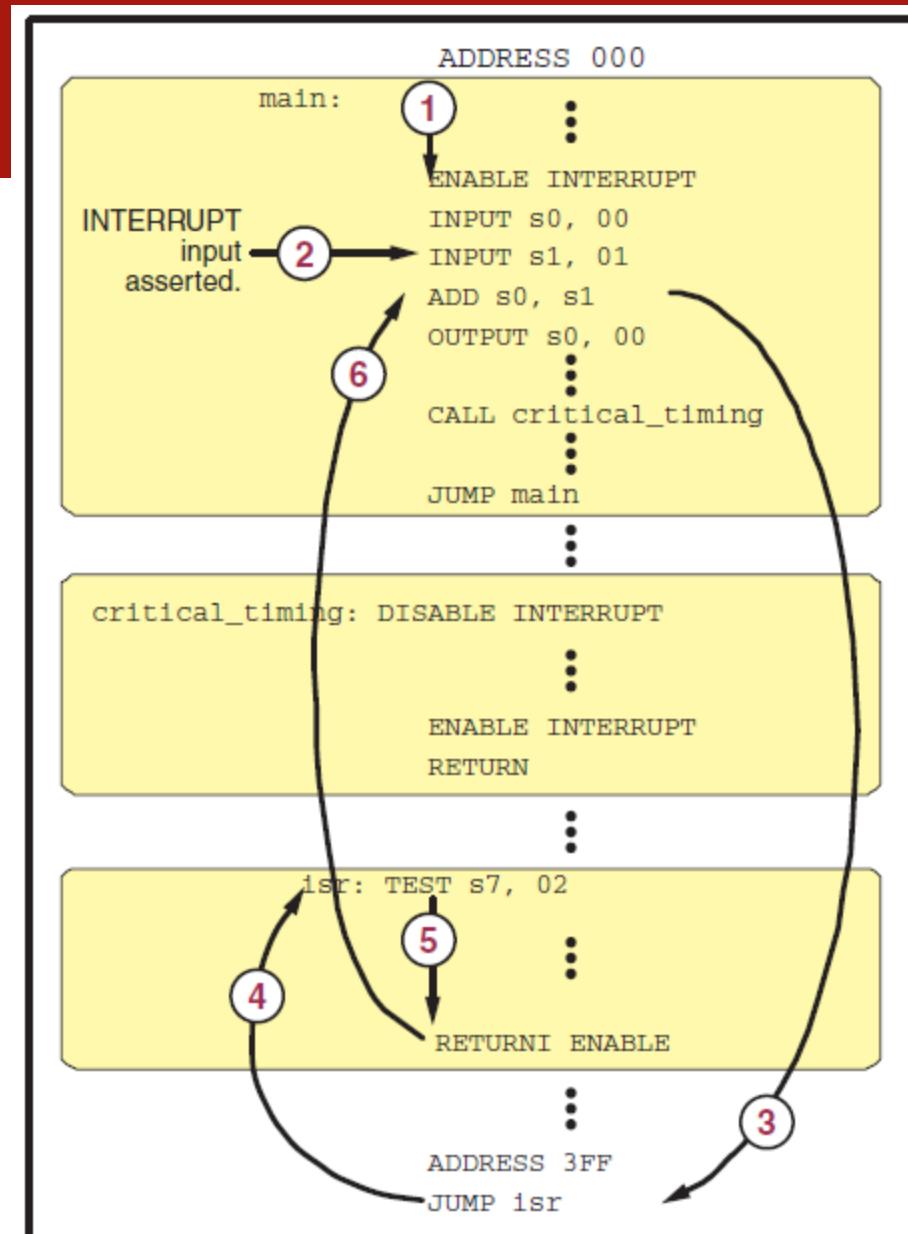
# PicoBlaze – Interrupts

- The PicoBlaze processor provides a single interrupt input signal
- If the application requires multiple interrupt signals, combine the signals using simple FPGA logic to form a single INTERRUPT input signal.
- An active interrupt forces the PicoBlaze processor to execute the CALL 3FF instruction immediately after completing the instruction currently executing

# PicoBlaze – Simple interrupts logic

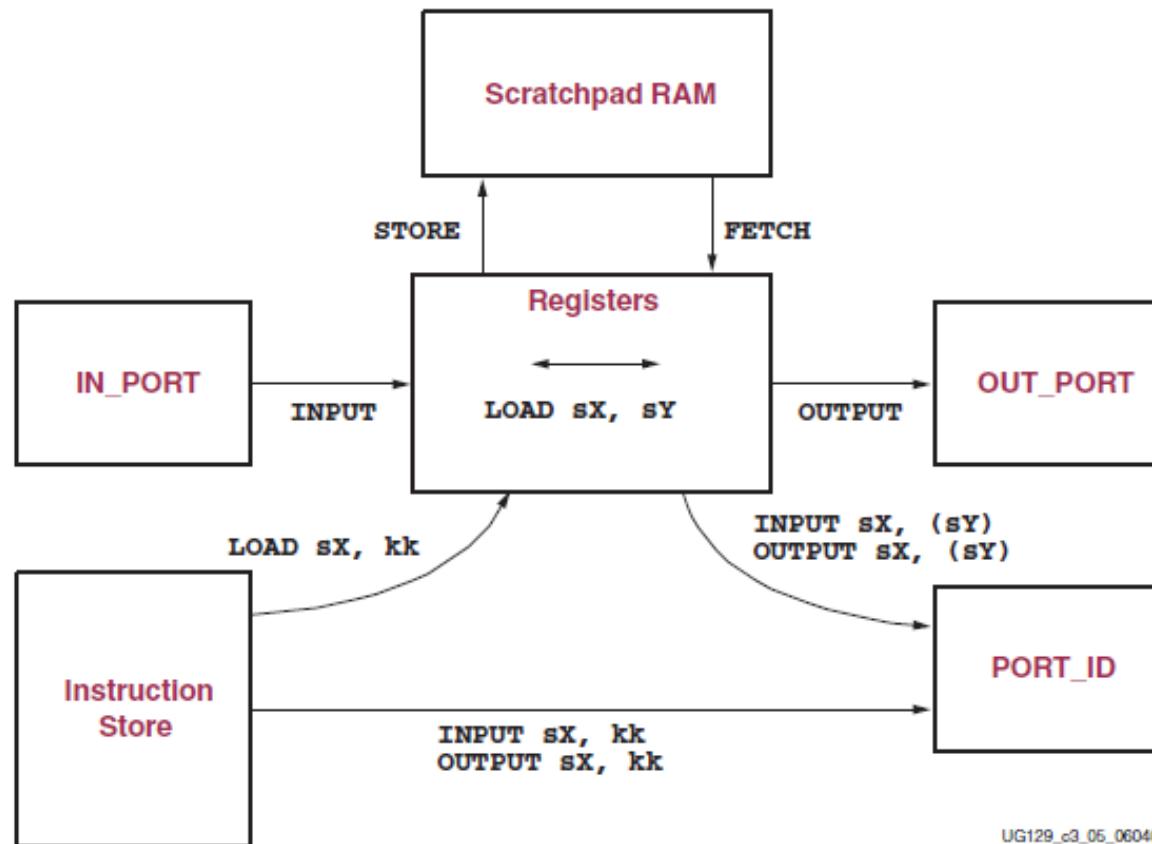


# PicoBlaze – Interrupt Flow



Source: [1]

# PicoBlaze – data movement





# PicoBlaze in FPGA design

- The PicoBlaze microcontroller is primarily designed for use in a VHDL design flow
- The PicoBlaze microcontroller is supplied as a VHDL source file, called *KCPSM3.vhd*
- The KCPSM3 module contains the PicoBlaze ALU, register file, scratchpad, RAM etc,
- Up to 1k PROM uploaded during FPGA startup sequence has to be additionally added (done automatically using tools delivered by Xilinx – *kcpsm3.exe*)



# PicoBlaze in FPGA design

```
component KCPSM3
port (
    address      : out std_logic_vector( 9 downto 0);
    instruction   : in  std_logic_vector(17 downto 0);
    port_id       : out std_logic_vector( 7 downto 0);
    write_strobe   : out std_logic;
    out_port      : out std_logic_vector( 7 downto 0);
    read_strobe   : out std_logic;
    in_port        : in  std_logic_vector( 7 downto 0);
    interrupt      : in  std_logic;
    interrupt_ack  : out std_logic;
    reset          : in  std_logic;
    clk            : in  std_logic
);
end component;
```

Declaration

```
processor: kcpsm3
port map(
    address => address_signal,
    instruction => instruction_signal,
    port_id => port_id_signal,
    write_strobe => write_strobe_signal,
    out_port => out_port_signal,
    read_strobe => read_strobe_signal,
    in_port => in_port_signal,
    interrupt => interrupt_signal,
    interrupt_ack => interrupt_ack_signal,
    reset => reset_signal,
    clk => clk_signal
);
```

Instantiation



# PicoBlaze in FPGA design

## Declaration

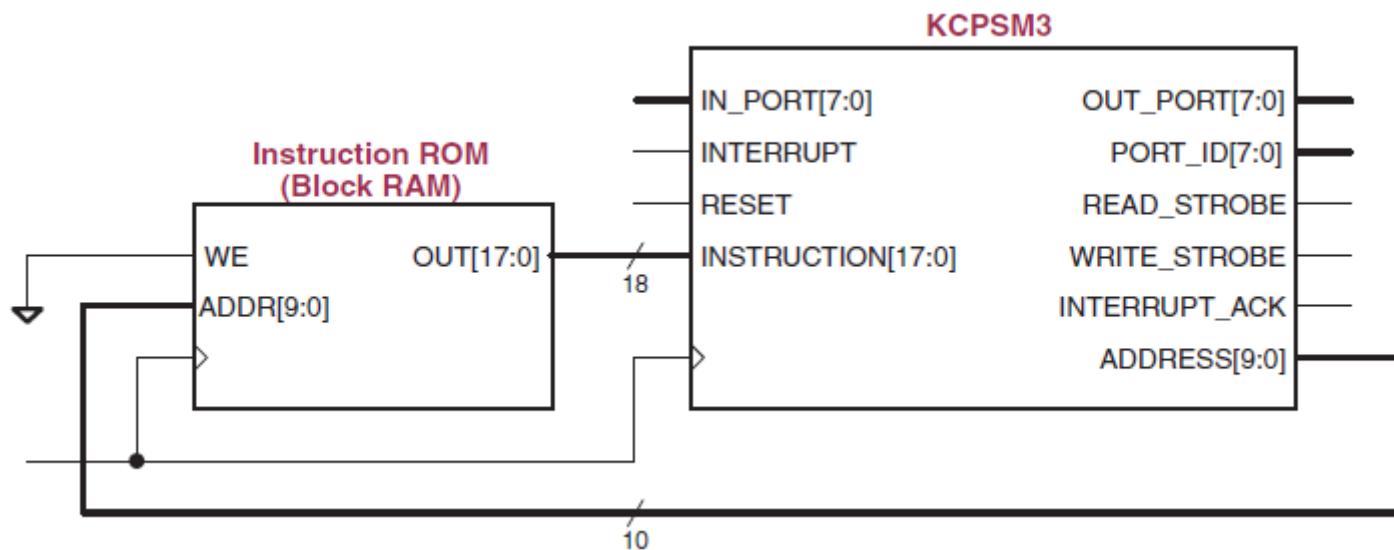
```
component prog_rom
port (
    address      : in  std_logic_vector( 9 downto 0);
    instruction  : out std_logic_vector(17 downto 0);
    clk          : in  std_logic
);
end component;
```

## Instantiation

```
program: prog_rom
port map(
    address => address_signal,
    instruction => instruction_signal,
    clk => clk_signal
);
```

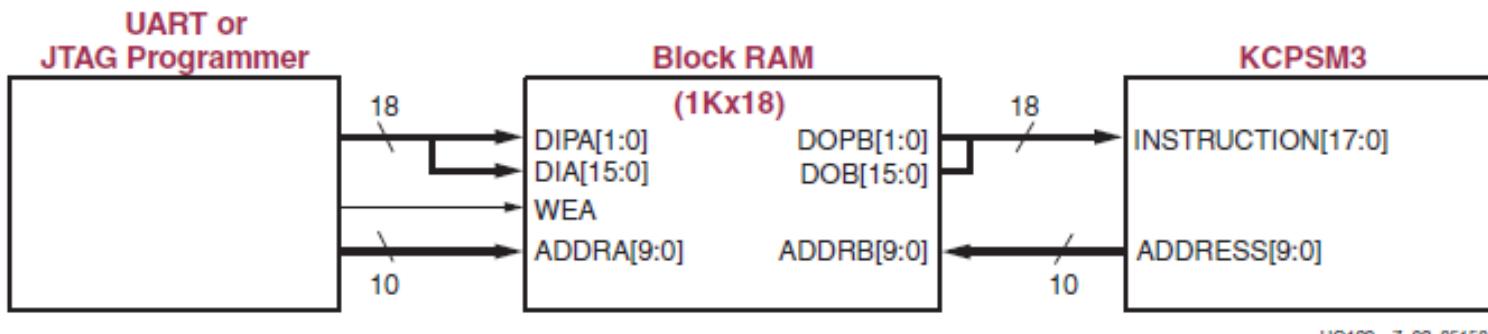
# PicoBlaze – instruction storage

- Standard: Single 1Kx18 Block RAM



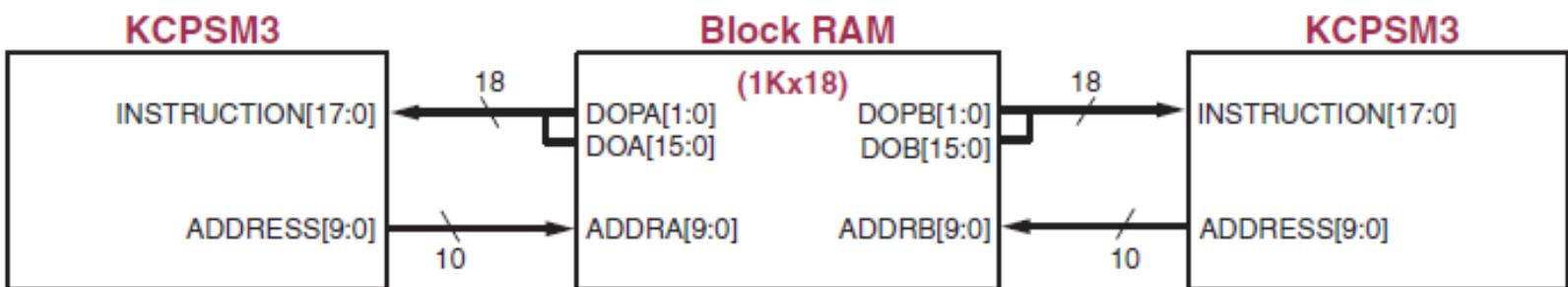
# PicoBlaze – instruction storage

- Standard: Single 1Kx18 Block RAM - reconfigurable through JTAG or UART



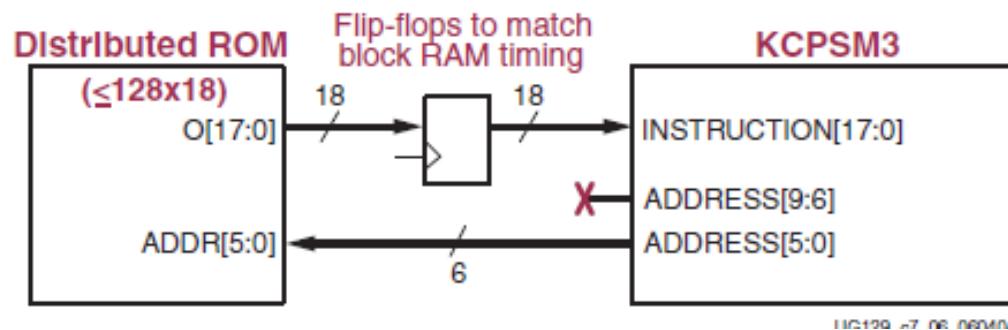
# PicoBlaze – instruction storage

- Two PicoBlaze Microcontrollers Share a 1Kx18 Code Image



# PicoBlaze – instruction storage

- Distributed ROM Instead of Block RAM

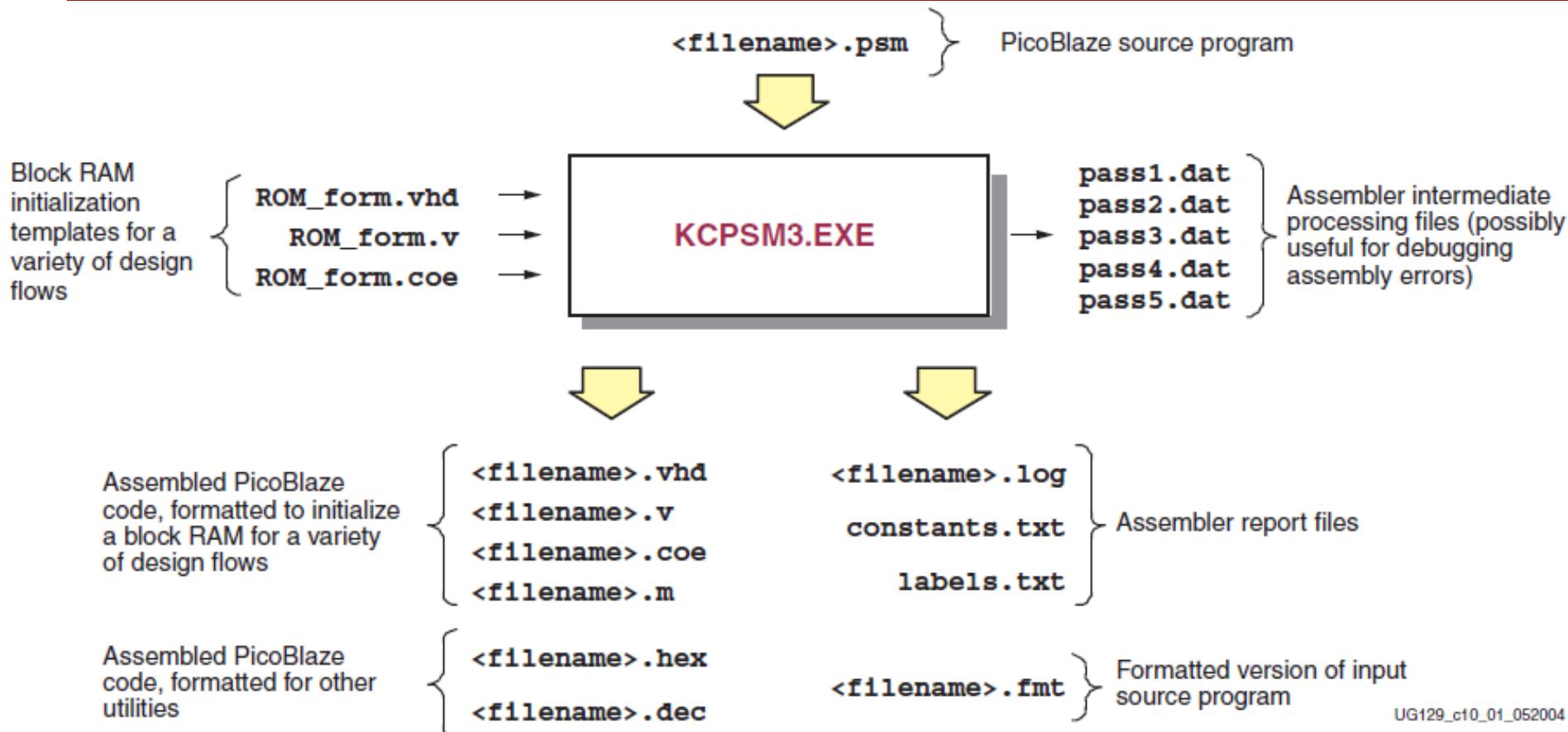


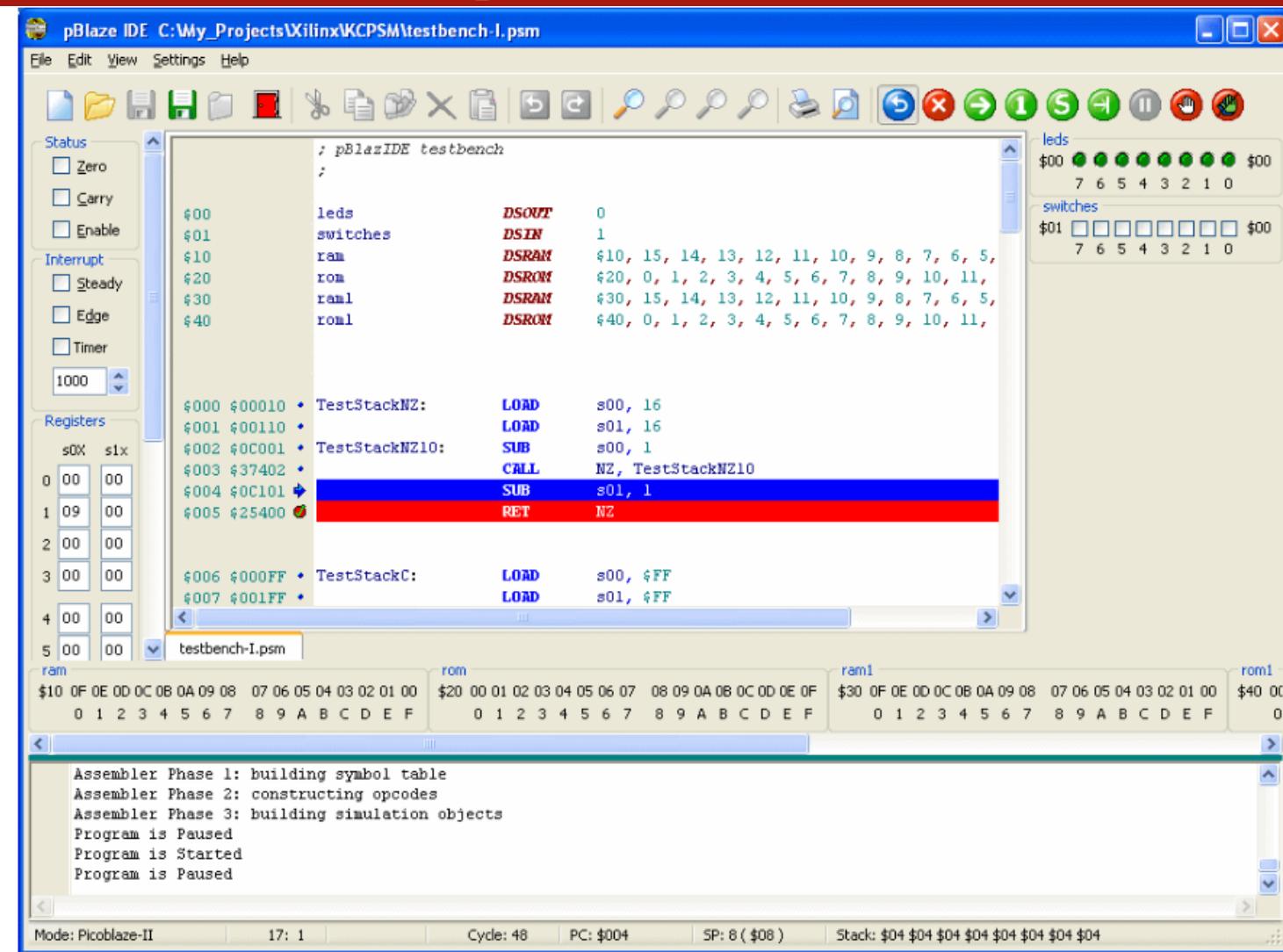


# PicoBlaze – development tools

	Xilinx KCPSM3	Mediatronix pBlazIDE	Xilinx System Generator
Platform Support	Windows	Windows 98, Windows 2000, Windows NT, Windows ME, Windows XP	Windows 2000, Windows XP
Assembler	Command-line in DOS window	Graphical	Command-line within System Generator
Instruction Syntax	KCPSM3	PBlazIDE	KCPSM3
Instruction Set Simulator	Facilities provided for VHDL simulation	Graphical/Interactive	Graphical/Interactive
Simulator Breakpoints	N/A	Yes	Yes
Register Viewer	N/A	Yes	Yes
Memory Viewer	N/A	Yes	Yes

# PicoBlaze – KCPSM3.exe







# PicoBlaze – development tools

KCPSM3 Instruction	pBlazIDE Instruction
RETURN	RET
RETURN C	RET C
RETURN NC	RET NC
RETURN Z	RET Z
RETURN NZ	RET NZ
RETURNI ENABLE	RETI ENABLE
RETURNI DISABLE	RETI DISABLE
ADDCY	ADDC
SUBCY	SUBC
INPUT sX, (sY)	IN sX, sY (no parentheses)
INPUT sX, kk	IN sX, kk
OUTPUT sX, (sY)	OUT sX, sY (no parentheses)
OUTPUT sX, kk	OUT sX, kk
ENABLE INTERRUPT	EINT
DISABLE INTERRUPT	DINT
COMPARE	COMP
STORE sX, (sY)	STORE sX, sY (no parentheses)
FETCH sX, (sY)	FETCH sX, sY (no parentheses)



# PicoBlaze - multiplication

- The PicoBlaze microcontroller core does not have a dedicated hardware multiplier
- The multiplication is available with the use of arithmetic and shift instructions
- An 8-bit by 8-bit multiply routine that produces a 16-bit multiplier product in 50 to 57 instruction cycles, or 100 to 114 clock cycles (96 clock cycles in 8051 CPU)



# PicoBlaze - multiplication

```
; Multiplier Routine (8-bit x 8-bit = 16-bit product)
; =====
; Shift and add algorithm
;

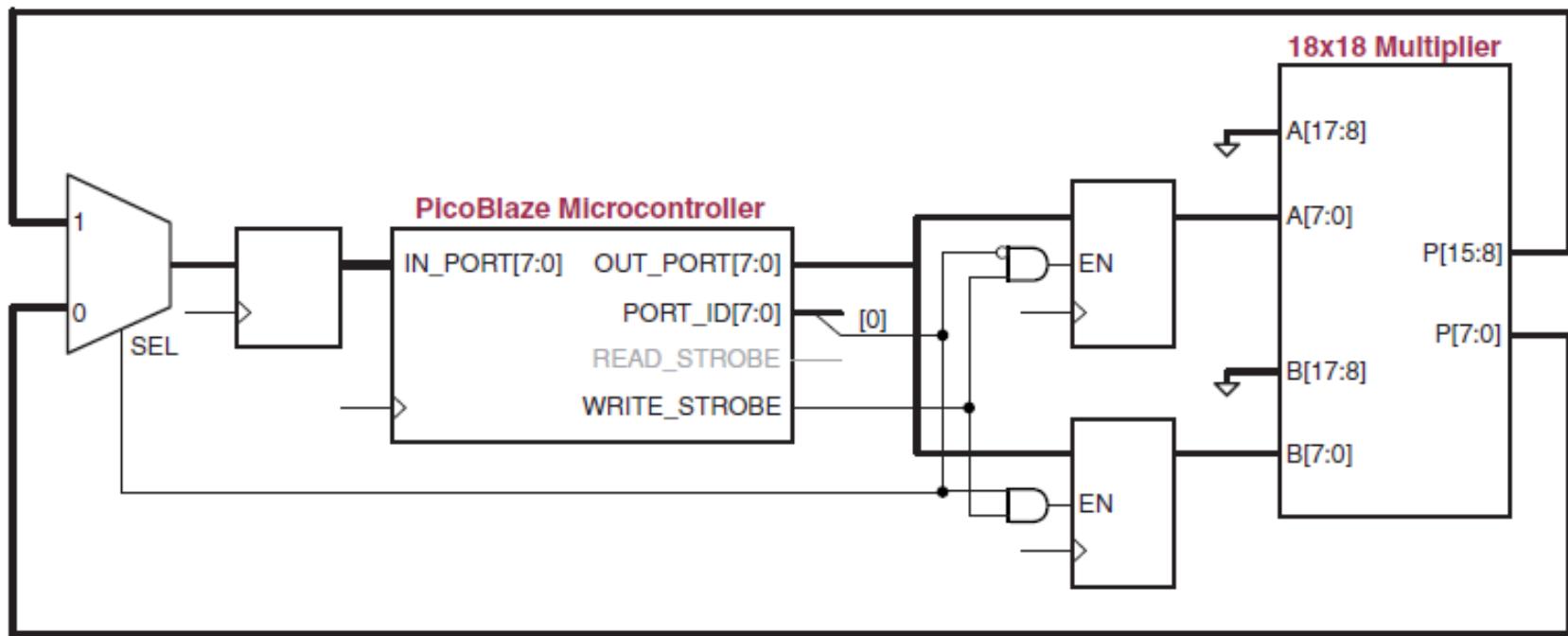
mult_8x8:
    NAMEREG s0, multiplicand      ; preserved
    NAMEREG s1, multiplier       ; preserved
    NAMEREG s2, bit_mask         ; modified
    NAMEREG s3, result_msb      ; most-significant byte (MSB) of result,
                                ;     modified
    NAMEREG s4, result_lsb      ; least-significant byte (LSB) of result,
                                ;     modified
    ;
    LOAD bit_mask, 01           ; start with least-significant bit (lsb)
    LOAD result_msb, 00          ; clear product MSB
    LOAD result_lsb, 00          ; clear product LSB (not required)
    ;
    ; loop through all bits in multiplier
mult_loop: TEST multiplier, bit_mask ; check if bit is set
    JUMP Z, no_add              ; if bit is not set, skip addition
    ;
    ADD result_msb, multiplicand ; addition only occurs in MSB
    ;
no_add:   SRA result_msb        ; shift MSB right, CARRY into bit 7,
                                ;     lsb into CARRY
    SRA result_lsb             ; shift LSB right,
                                ;     lsb from result_msb into bit 7
    ;
    SLO bit_mask               ; shift bit_mask left to examine
                                ;     next bit in multiplier
    ;
    JUMP NZ, mult_loop         ; if all bit examined, then bit_mask = 0,
```



# PicoBlaze - multiplication

- If multiplication performance is important to the application, then one of a 18x18 hardware multipliers the PicoBlaze I/O ports can be used
- The hardware multiplier computes the 16-bit result in less than one instruction cycle
- This same technique can be expanded to multiply two 16-bit values to produce a 32-bit result

# PicoBlaze - multiplication





# PicoBlaze - multiplication

```
; Multiplier Routine (8-bit x 8-bit = 16-bit product)
; =====
; Connects to embedded 18x18 Hardware Multiplier via ports
;
mult_8x8io:
    NAMEREG s0, multiplicand      ; preserved
    NAMEREG s1, multiplier       ; preserved
    NAMEREG s3, result_msb      ; most-significant byte (MSB) of result, modified
    NAMEREG s4, result_lsb       ; least-significant byte (LSB) of result, modified
;
    ; Define the port ID numbers as constants for better clarity
    CONSTANT multiplier_lsb, 00
    CONSTANT multiplier_msb, 01
;
    ; Output multiplicand and multiplier to FPGA registers connected to the
    ; inputs of
    ;   the embedded multiplier.
    OUTPUT multiplicand, multiplier_lsb
    OUTPUT multiplier, multiplier_msb
;
    ; Input the resulting product from the embedded multiplier.
    INPUT result_lsb, multiplier_lsb
    INPUT result_msb, multiplier_msb
```



# PicoBlaze - multiplication

- The PicoBlaze microcontroller core does not have a dedicated hardware multiplier
- The multiplication is available with the use of arithmetic and shift instructions
- An 8-bit by 8-bit multiply routine that produces a 16-bit multiplier product in 50 to 57 instruction cycles, or 100 to 114 clock cycles (96 clock cycles in 8051 CPU)



# PicoBlaze - simulation

- There are four may ways to simulate PicoBlaze:
  - In-system on FPGA (free ISE)
  - With free tool: pBlazIDE
  - With Xilinx System Generator (not free)
  - With ModelSim (not free)
- Each method has its pros and cons



# PicoBlaze - simulation

Verification Tool	Strengths	Weaknesses
Xilinx System Generator	<ul style="list-style-type: none"><li>• Full PicoBlaze system-level simulation with the System Generator environment</li><li>• Cycle-accurate Instruction Set Simulation (ISS)</li><li>• Single-step</li><li>• Breakpoints</li><li>• Register and memory viewer</li></ul>	<ul style="list-style-type: none"><li>• Primarily only useful if already using Xilinx System Generator</li></ul>
In-system on FPGA	<ul style="list-style-type: none"><li>• Fast, real-time performance</li><li>• Ideal for complex interactions</li><li>• Integrated with peripherals, displays, UARTs, etc.</li></ul>	<ul style="list-style-type: none"><li>• Poor visibility of register contents</li></ul>



# PicoBlaze - simulation

Verification Tool	Strengths	Weaknesses
pBlazIDE	<ul style="list-style-type: none"><li>• Ideal for rapid code development</li><li>• Cycle-accurate Instruction Set Simulation (ISS)</li><li>• Single-step</li><li>• Breakpoints</li><li>• Intimate interaction with PicoBlaze registers, flags, and memory values</li><li>• Code coverage indicator</li><li>• Software timing</li><li>• Basic system-level simulation via file I/O functions</li><li>• Free!</li></ul>	<ul style="list-style-type: none"><li>• No modeling of custom logic attached to the PicoBlaze microcontroller</li></ul>
ModelSim	<ul style="list-style-type: none"><li>• Holistic simulation of PicoBlaze microcontroller with associated FPGA logic</li><li>• VHDL and Verilog logic and timing simulation</li><li>• Cycle and timing accurate with FPGA hardware</li></ul>	<ul style="list-style-type: none"><li>• Requires simulator setup and stimulus files</li><li>• Digital simulator, not an ideal Instruction Set Simulation environment</li></ul>





# MicroBlaze – Key Features

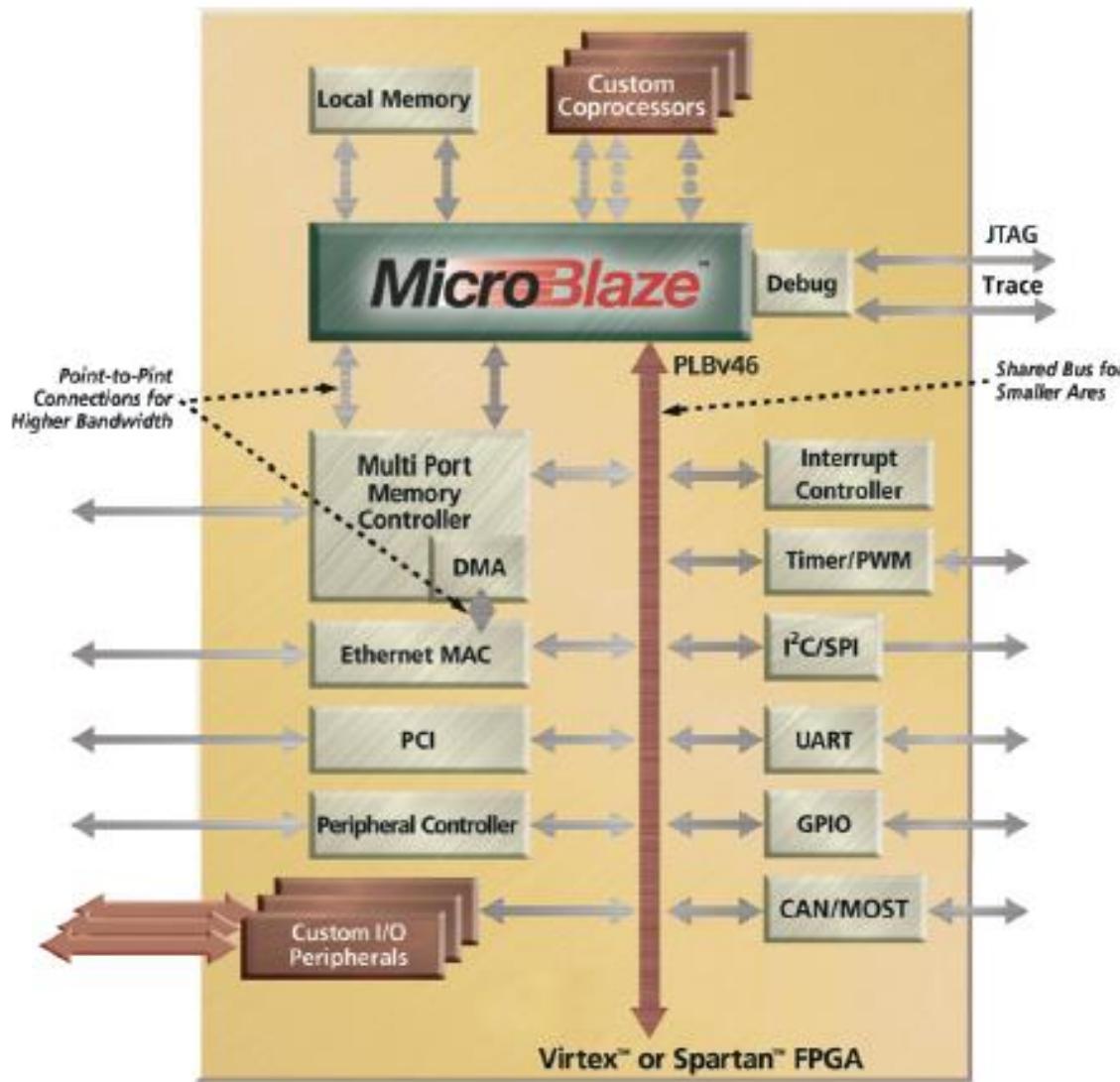
- 32-bit RISC Harvard architecture soft processor core
- 32 general purpose 32b registers
- ALU
- Rich instruction set optimized for embedded applications
- Three levels of MMU/MPU support:
  - MMU (privileged mode, memory protection and virtual address translation)
  - MPU (privileged mode and memory protection)
  - Privileged mode only



# MicroBlaze – Key Features

- An integrated single precision, IEEE-754 compatible Floating Point Unit (FPU) option optimized for embedded applications such as industrial control, automotive, and office automation
- Highly configurable features such as:
  - barrel shifter,
  - divider,
  - multiplier,
  - instruction and data caches,
  - FPU

# MicroBlaze – Block Diagram



Source: [2]



# MicroBlaze – Hardware functions

- Hardware Barrel Shifter
- Hardware Divider
- Machine Status Set and Clear Instructions
- Hardware Exception Support
- Processor Version Register
- Floating-Point Unit (FPU)
- MMU/MPU
- Hardware Multiplier
- Hardware Debug Logic



# MicroBlaze – Cache options

- Configurable size 2 KB - 64 KB
- Configurable micro-cache size 64B –1024B
- 4 or 8 word cache lines



# MicroBlaze – Bus infrastructure

- PLB v4.6 for main processor interface
- Local Memory Bus (LMB) for fast local access memory
- Fast Simplex Link (FSL) for interfacing to co-processors



# MicroBlaze – Performance

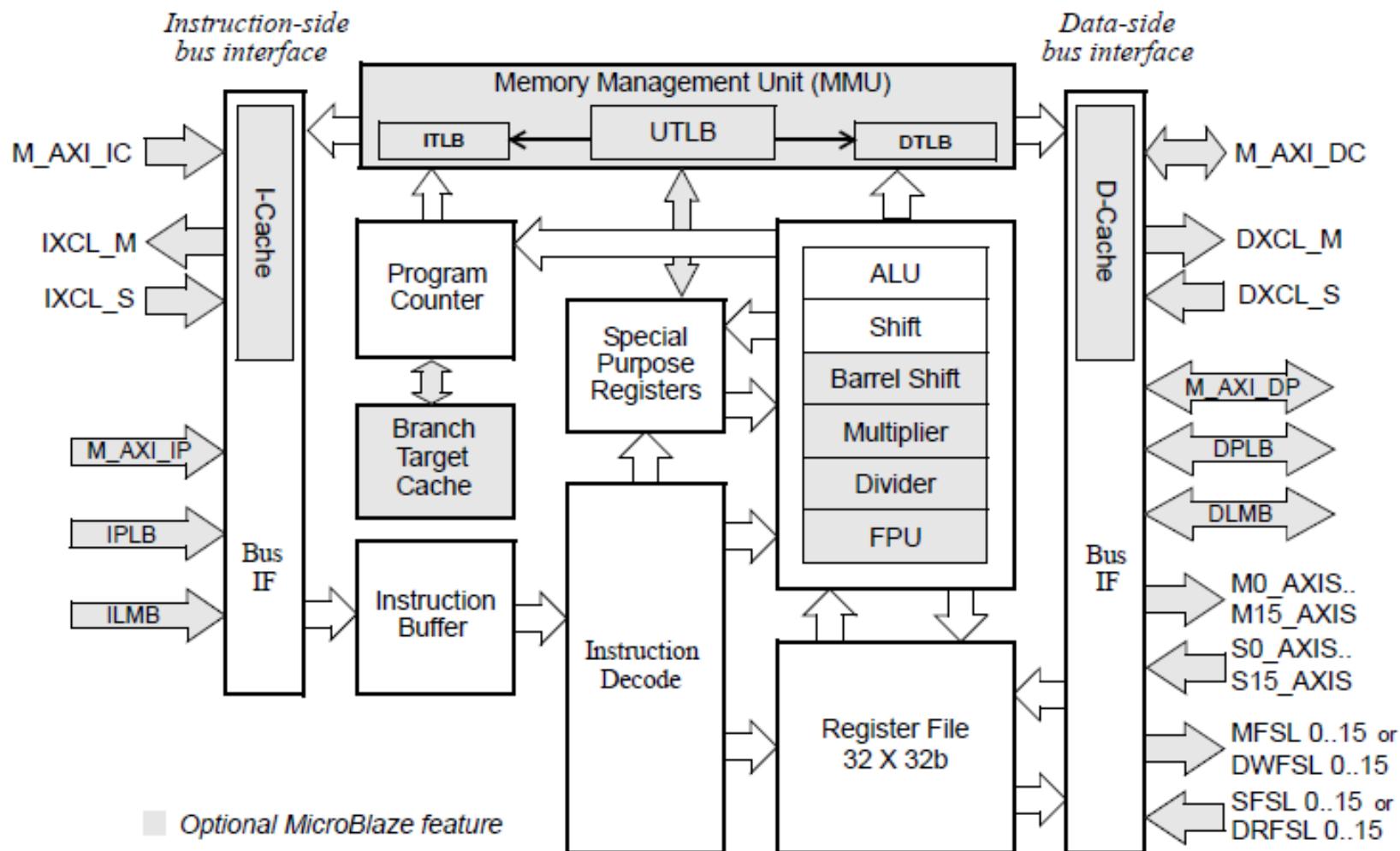
## MicroBlaze Processor v7.2 Performance Levels

Architecture	Performance	Maximum Clock Frequency	Maximum Dhrystone 2.1 Performance
5-Stage Pipeline	1.19 DMIPS/MHz	235 MHz in Virtex-5 FXT	280 DMIPS
3-Stage Pipeline	0.95 DMIPS/MHz	106 MHz in Spartan-3A DSP	100 DMIPS

## MicroBlaze Processor v7.2 FPU: Single Precision

FPGA	Size	Maximum Clock Frequency	Peak Floating Point Throughput
Virtex-5	<1650 LUTs (850 FPU + 800MB)	200MHz	50 MFLOPS

# MicroBlaze core – Block Diagram



Source: [2]



# MicroBlaze – some instructions

ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	000000000000	Rd := Rb + Ra
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	000000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	000000000000	Rd := Rb + Ra + C
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	000000000000	Rd := Rb + $\overline{Ra}$ + C
ADDK Rd,Ra,Rb	000100	Rd	Ra	Rb	000000000000	Rd := Rb + Ra
RSUBK Rd,Ra,Rb	000101	Rd	Ra	Rb	000000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDKC Rd,Ra,Rb	000110	Rd	Ra	Rb	000000000000	Rd := Rb + Ra + C
RSUBKC Rd,Ra,Rb	000111	Rd	Ra	Rb	000000000000	Rd := Rb + $\overline{Ra}$ + C
CMP Rd,Ra,Rb	000101	Rd	Ra	Rb	000000000001	Rd := Rb + $\overline{Ra}$ + 1(signed)
CMPU Rd,Ra,Rb	000101	Rd	Ra	Rb	000000000011	Rd := Rb + $\overline{Ra}$ + 1(unsigned)
ADDI Rd,Ra,Imm	001000	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBI Rd,Ra,Imm	001001	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIC Rd,Ra,Imm	001010	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIC Rd,Ra,Imm	001011	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
ADDIK Rd,Ra,Imm	001100	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBIK Rd,Ra,Imm	001101	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIKC Rd,Ra,Imm	001110	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIKC Rd,Ra,Imm	001111	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
MUL Rd,Ra,Rb	010000	Rd	Ra	Rb	000000000000	Rd := Ra * Rb



# MicroBlaze – Special registers

- Up to 18 Special Purpose Registers available:
  - Program Counter (PC)
  - Machine Status Register (MSR)
  - Exception Address Register (EAR)
  - Exception Status Register (ESR)
  - Branch Target Register (BTR)
  - Stack Low Register (SLR)
  - Stack High Register (SHR)
  - Zone Protection Register (ZPR)
  - Processor Version Register (PVR)
  - ....

# MicroBlaze – pipeline

- MicroBlaze uses a pipelined instruction execution. The pipeline is divided into three or five stages:
  - Fetch
  - Decode
  - Execute

	cycle 1	cycle 2	cycle 3	cycle4	cycle5	cycle6	cycle7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

# MicroBlaze – pipeline

- Five stage pipeline (better performance):
  - Fetch (IF)
  - Decode (OF)
  - Execute (EX)
  - Access Memory (MEM)
  - Writeback (WB)

	cycle 1	cycle 2	cycle 3	cycle4	cycle5	cycle6	cycle7	cycle8	cycle9
instruction 1	IF	OF	EX	MEM	WB				
instruction 2		IF	OF	EX	MEM	MEM	MEM	WB	
instruction 3			IF	OF	EX	Stall	Stall	MEM	WB



# MicroBlaze – pipeline

- For most instructions, each stage takes one clock cycle to complete
- When executing from slower memory, instruction fetches may take multiple cycles
- MicroBlaze implements an instruction prefetch buffer that reduces the impact of such multi-cycle instruction memory latency



# MicroBlaze – memory access

- MicroBlaze is implemented with a Harvard memory architecture; instruction and data accesses are done in separate address spaces
- Each address space has a 32-bit range
- The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory



# MicroBlaze – memory access

- Both instruction and data interfaces of MicroBlaze are default 32 bits wide
- MicroBlaze supports word, halfword, and byte accesses to data memory
- MicroBlaze prefetches instructions to improve performance, using the instruction prefetch buffer and (if enabled) instruction cache streams



# MicroBlaze – memory access

- MicroBlaze does not separate data accesses to I/O and memory (it uses memory mapped I/O)
- The processor has up to three interfaces for memory accesses:
  - Local Memory Bus (LMB)
  - Advanced eXtensible Interface (AXI4) or Processor Local Bus (PLB)
  - Advanced eXtensible Interface (AXI4) or Xilinx CacheLink (XCL)



# MicroBlaze – virtual memory

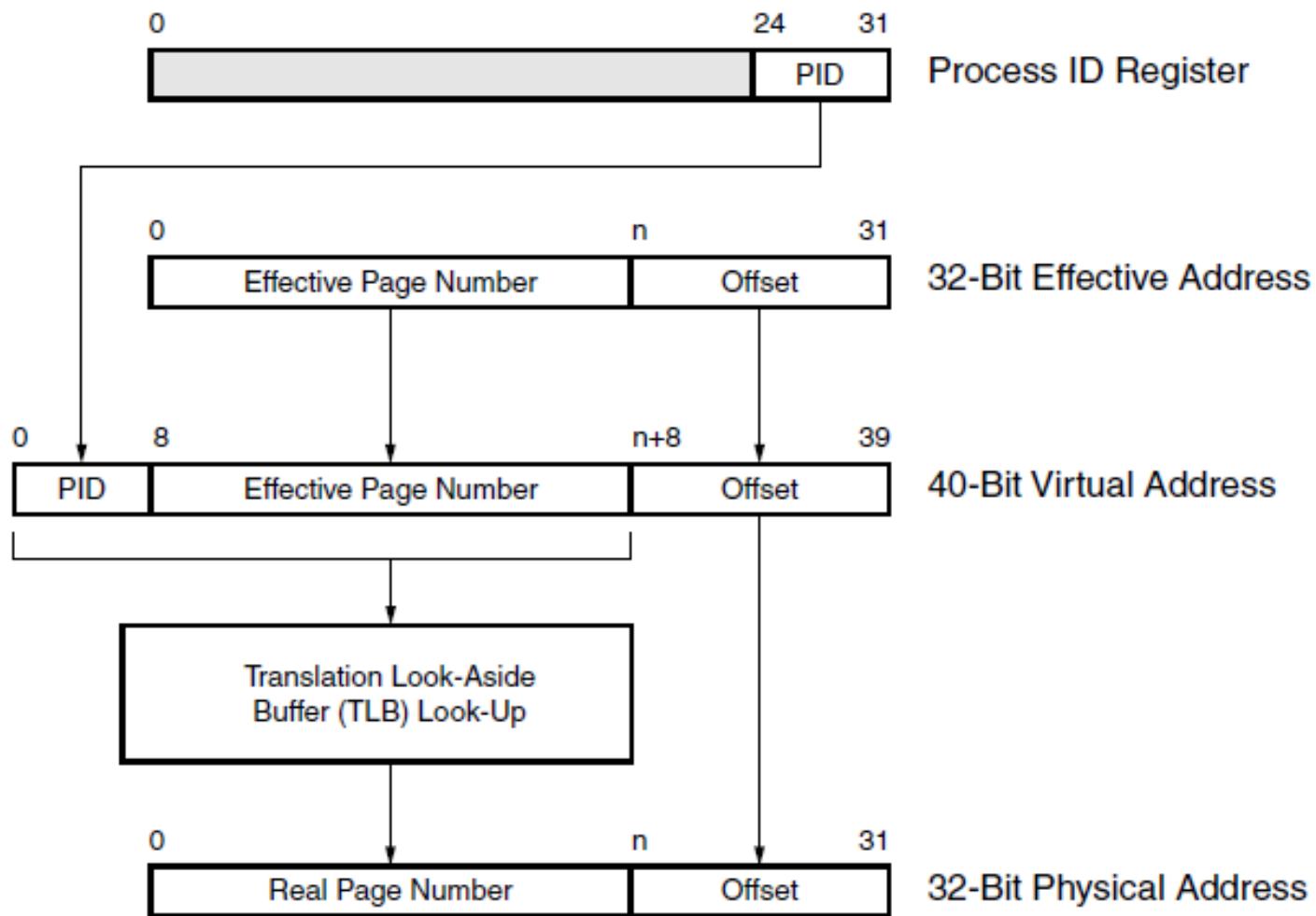
- Programs running on MicroBlaze use effective addresses to access a flat 4 GB address space in two modes:
  - Real
  - Virtual (translated by memory management hardware)



# MicroBlaze – virtual memory

- The MMU features:
  - Translates effective addresses into physical addresses
  - Controls page-level access during address translation
  - Provides additional virtual-mode protection control through the use of zones
  - Provides independent control over instruction-address and data-address translation and protection
  - Supports eight page sizes: 1 kB, 4 kB, 16 kB, 64 kB, 256 kB, 1 MB, 4 MB, and 16 MB. Any combination of page sizes can be used by system software
  - Software controls the page-replacement strategy

# MicroBlaze – virtual memory





# MicroBlaze – interrupts

- All versions of MicroBlaze supports reset, interrupt, user exception and break
- Newer versions support also hardware interrupts
- MicroBlaze supports one external interrupt source
- The processor will only react to interrupts if the interrupt enable (IE) bit in the machine status register (MSR) is set to 1
- On an interrupt the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the interrupt vector

# MicroBlaze – interrupts latency

- The time it will take MicroBlaze to enter an Interrupt Service Routine (ISR) from the time an interrupt occurs, depends on the configuration of the processor

Scenario	LMB Memory Vector
Normally	4 cycles
Worst case without hardware divider	6 cycles
Worst case with hardware divider <sup>1</sup>	37 cycles



# MicroBlaze – interrupts table

Event	Vector Address	Register File Return Address
Reset	0x00000000 - 0x00000004	-
User Vector (Exception)	0x00000008 - 0x0000000C	Rx
Interrupt	0x00000010 - 0x00000014	R14
Break: Non-maskable hardware	0x00000018 - 0x0000001C	R16
Break: Hardware		
Break: Software		
Hardware Exception	0x00000020 - 0x00000024	R17 or BTR
Reserved by Xilinx for future use	0x00000028 - 0x0000004F	-



# MicroBlaze – Signal interfaces

- The main bus interfaces are:
  - On-chip Peripheral Bus (OPB) - interface with byte-enable support
  - Local Memory Bus (LMB) - simple synchronous protocol for efficient block RAM transfers
  - up to 8 Fast Simplex Link (FSL) ports - a fast non-arbitrated streaming communication mechanism
  - Xilinx CacheLink - a fast slave-side arbitrated streaming interface between caches and specialized external memory controller



# MicroBlaze – FPU

- Uses IEEE 754 single precision floating point format
- Supported operations:
  - addition, fadd
  - subtraction, fsub
  - multiplication, fmul
  - division, fdiv
  - square root, fsqrt
  - compare less-than, fcmp.lt
  - compare equal, fcmp.eq

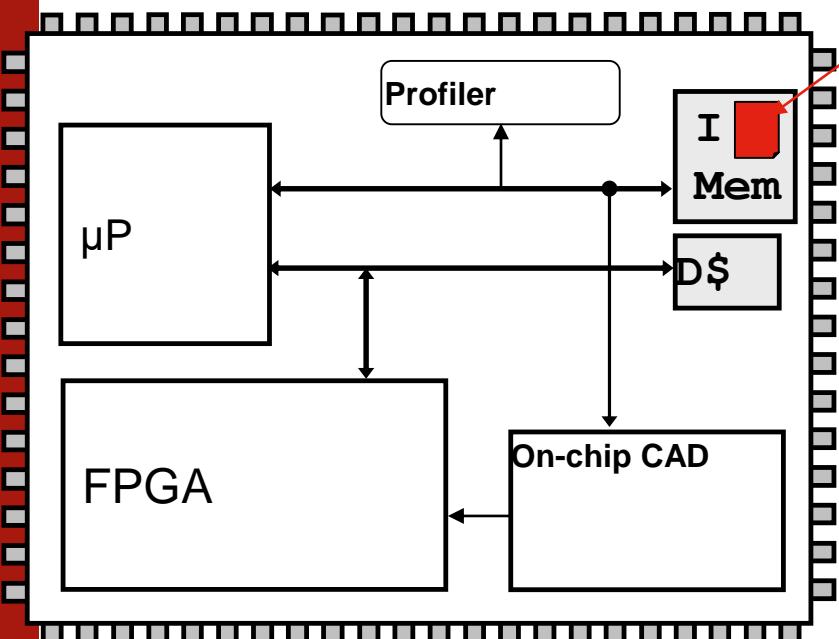


# MicroBlaze – FPU

- Supported operations:
  - compare less-or-equal, fcmp.le
  - compare greater-than, fcmp.gt
  - compare not-equal, fcmp.ne
  - compare greater-or-equal, fcmp.ge
  - compare unordered, fcmp.un (used for NaN)
  - convert from signed integer to floating point, flt
  - convert from floating point to signed integer, fint

# Warp Processing Idea

- 1 Initially, software binary loaded into instruction memory



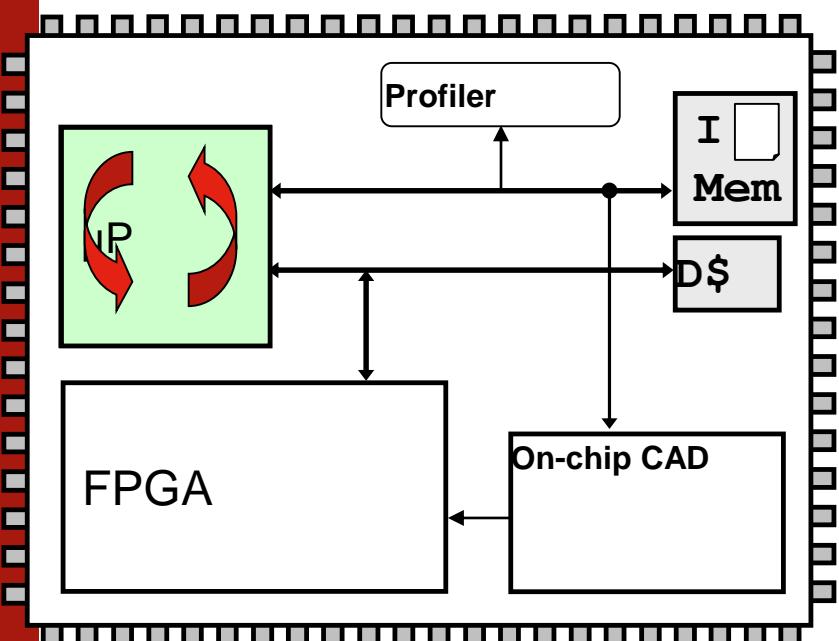
## Software

### Binary

```
Mov reg3, 0
Mov reg4, 0
loop:
Shl reg1, reg3, 1
Add reg5, reg2,
reg1
Ld reg6, 0(reg5)
Add reg4, reg4,
reg6
Add reg3, reg3, 1
Beq reg3, 10, -5
Ret reg4
```

# Warp Processing Idea

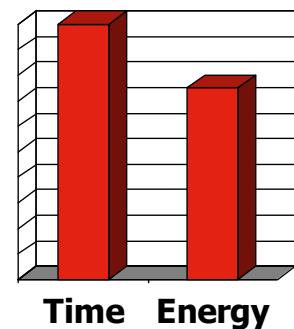
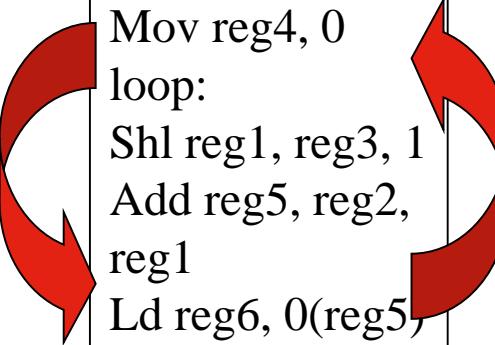
- 2 Microprocessor executes instructions in software binary



## Software

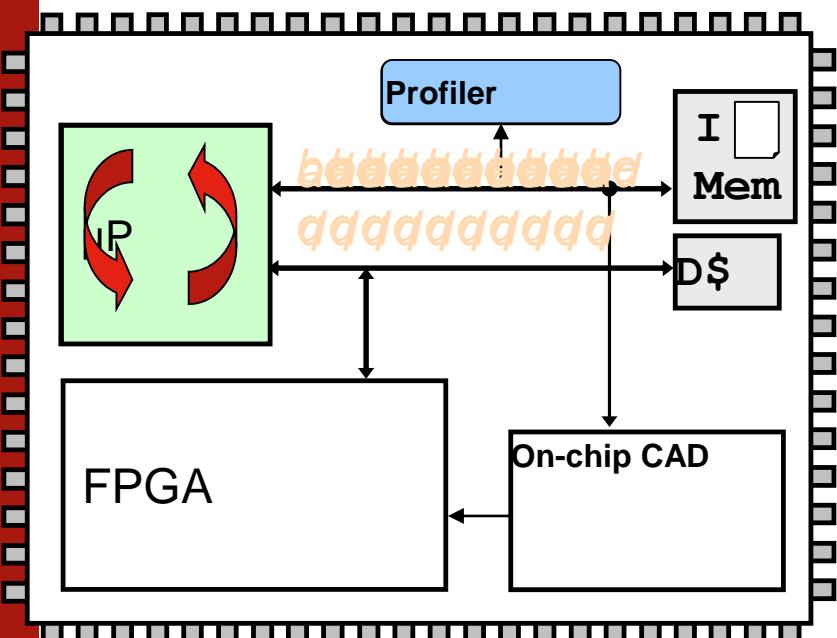
### Binary

```
Mov reg3, 0
Mov reg4, 0
loop:
Shl reg1, reg3, 1
Add reg5, reg2,
reg1
Ld reg6, 0(reg5,
Add reg4, reg4,
reg6
Add reg3, reg3, 1
Beq reg3, 10, -5
Ret reg4
```



# Warp Processing Idea

- 3 Profiler monitors instructions and detects critical regions in binary



## Software

### Binary

Mov reg3, 0

Mov reg4, 0

loop:

Shl reg1, reg3, 1

Add reg5, reg2,

reg1

Ld reg6, 0(reg5)

Add reg4, reg4,

reg6

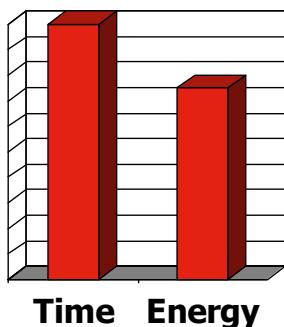
Critical

Loop

Detected

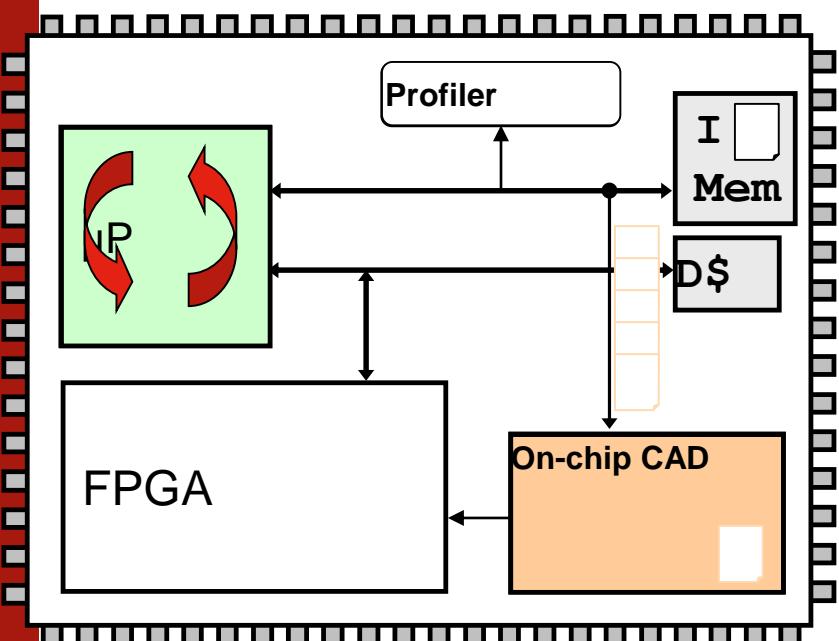
Reg reg3, 10, -5

Ret reg4



# Warp Processing Idea

- 4 On-chip CAD reads in critical region



## Software

### Binary

Mov reg3, 0

Mov reg4, 0

loop:

Shl reg1, reg3, 1

Add reg5, reg2,  
reg1

Ld reg6, 0(reg5)

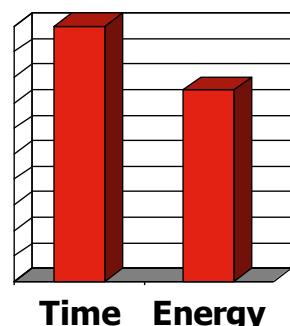
Add reg4, reg4,

reg6

Add reg3, reg3, 1

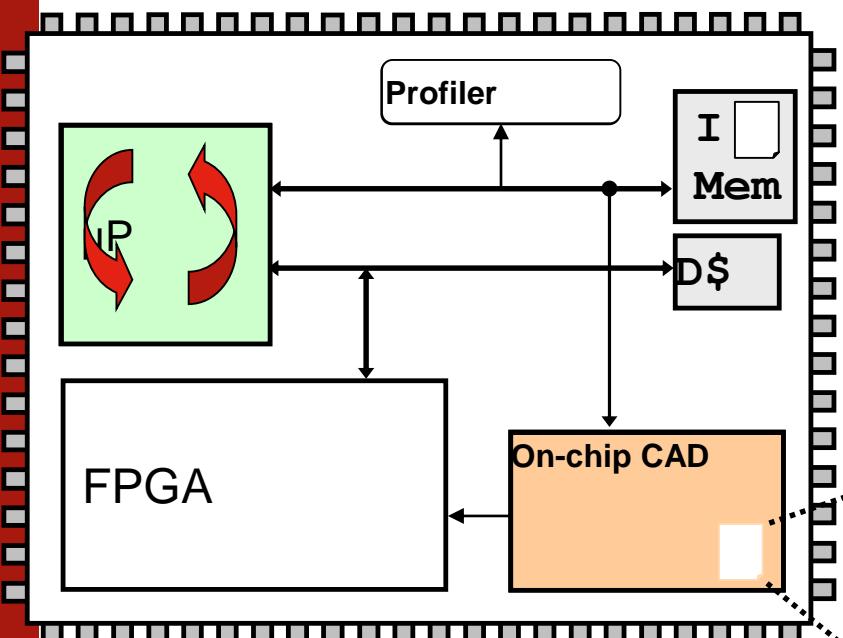
Beq reg3, 10, -5

Ret reg4



# Warp Processing Idea

- 5 On-chip CAD decompiles critical region into control data flow graph (CDFG)



## Software

### Binary

```
Mov reg3, 0
```

```
Mov reg4, 0
```

```
loop:
```

```
Shl reg1, reg3, 1
```

```
Add reg5, reg2,
```

```
reg1
```

```
Ld reg6, 0(reg5),
```

```
Add reg4, reg4,
```

```
reg6
```

```
Add reg3..r: reg3 :=
```

```
Beq reg3, 100..5
```

```
Ret reg4 .. loop :=
```

```
0reg4 := reg4 +
```

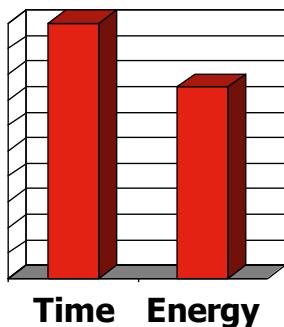
```
mem[
```

```
reg2 + (reg3 <<
```

```
1..1
```

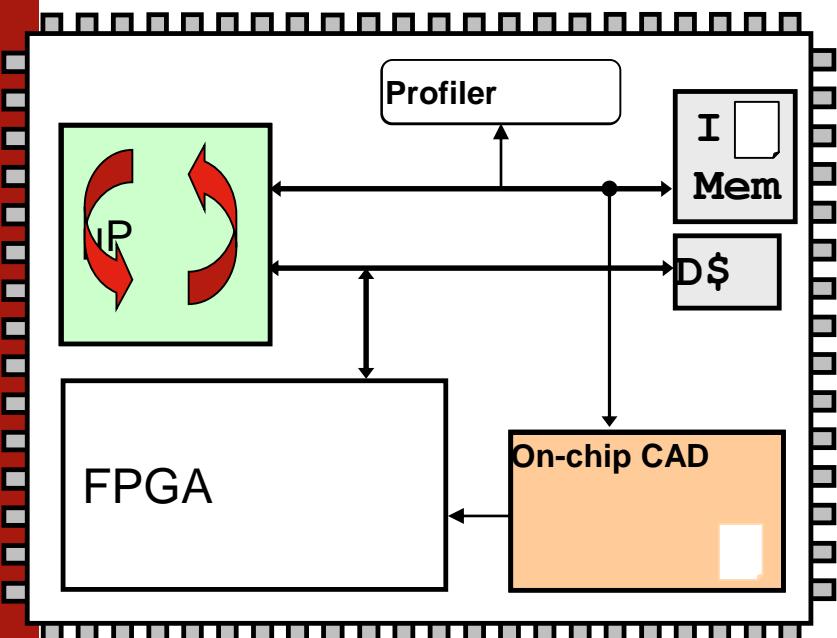
```
ret reg4, reg3 + 1
```

```
if (reg3 < 10) goto
```



# Warp Processing Idea

- 6 On-chip CAD synthesizes decompiled CDFG to a custom (parallel) circuit



## Software

### Binary

Mov reg3, 0

Mov reg4, 0

loop:

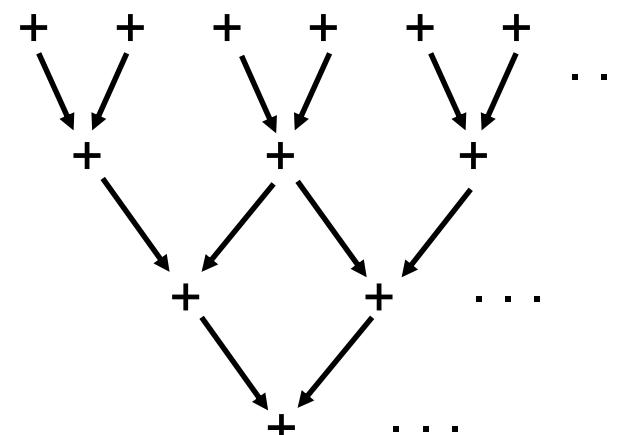
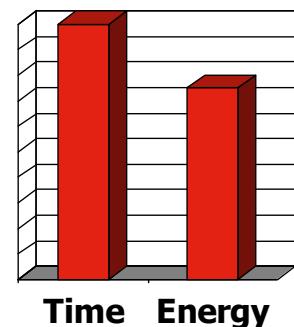
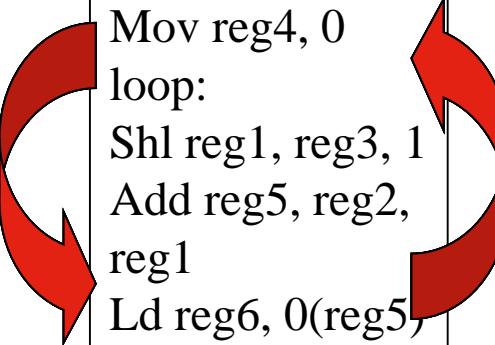
Shl reg1, reg3, 1

Add reg5, reg2,

reg1

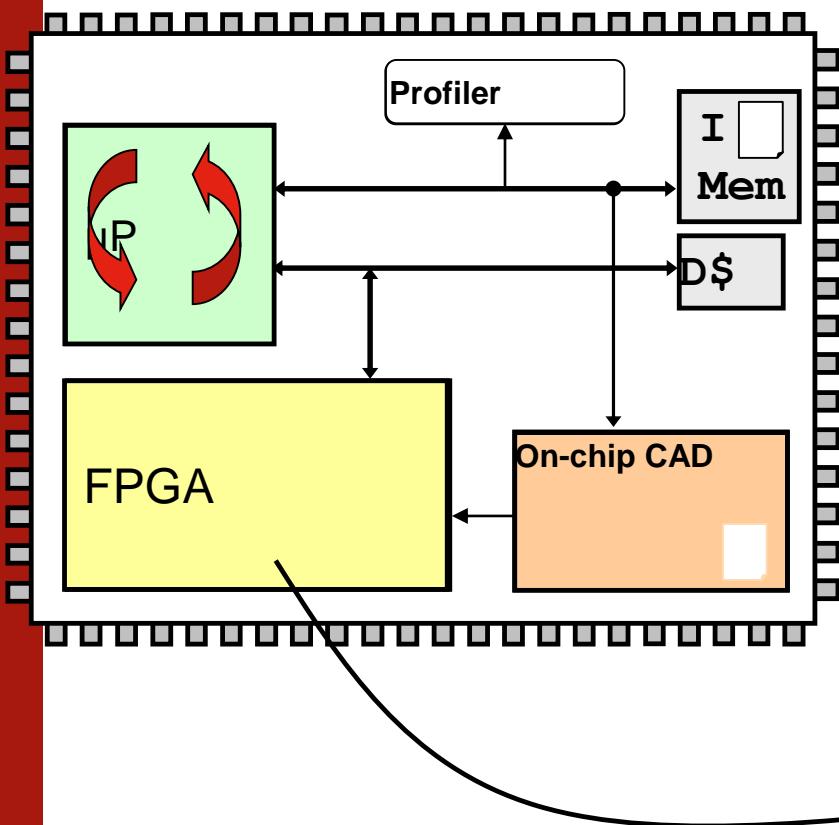
Ld reg6, 0(reg5,

Add reg4, reg4,



# Warp Processing Idea

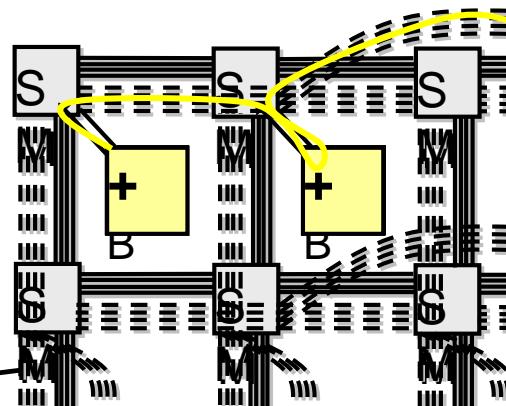
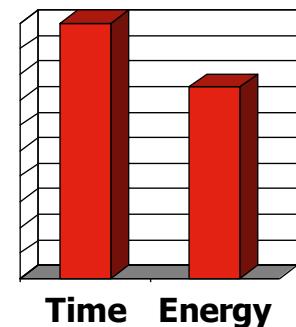
7 On-chip CAD maps circuit onto FPGA



## Software

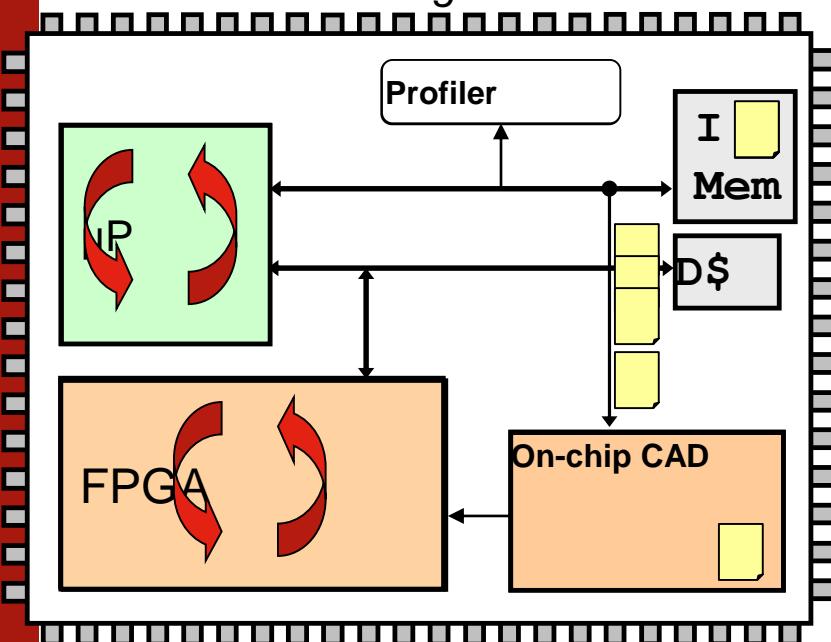
### Binary

```
Mov reg3, 0  
Mov reg4, 0  
loop:  
Shl reg1, reg3, 1  
Add reg5, reg2,  
reg1  
Ld reg6, 0(reg5),  
Add reg4, reg4,
```



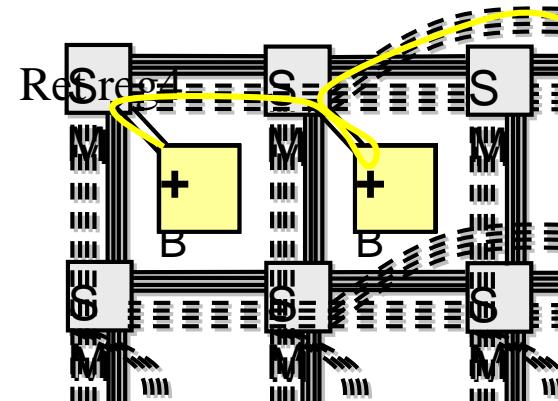
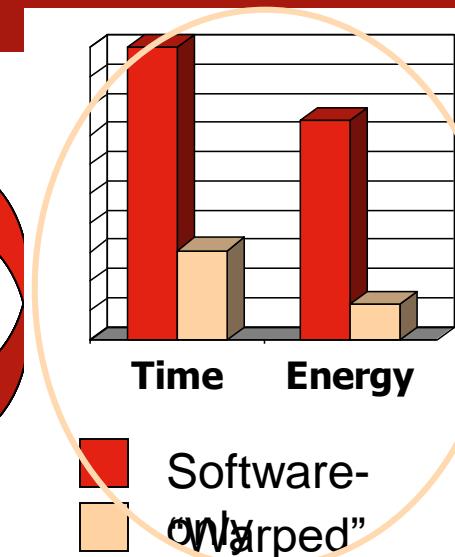
# Warp Processing Idea

8 On-chip CAD replaces instructions in binary to use hardware, causing performance and energy to “warp” by an order of magnitude or more



## Software

Mov reg3, 0  
Mov reg4, 0  
loop:  
*// instructions that interact with FPGA*



DAC'03, DAC'04, DATE'04,  
ISSS/CODES'04, FPGA'04, DATE'05,  
FCCM'05, ICCAD'05, ISSS/CODES'05,  
TECS'06, U.S. Patent Pending

# Warp Processors

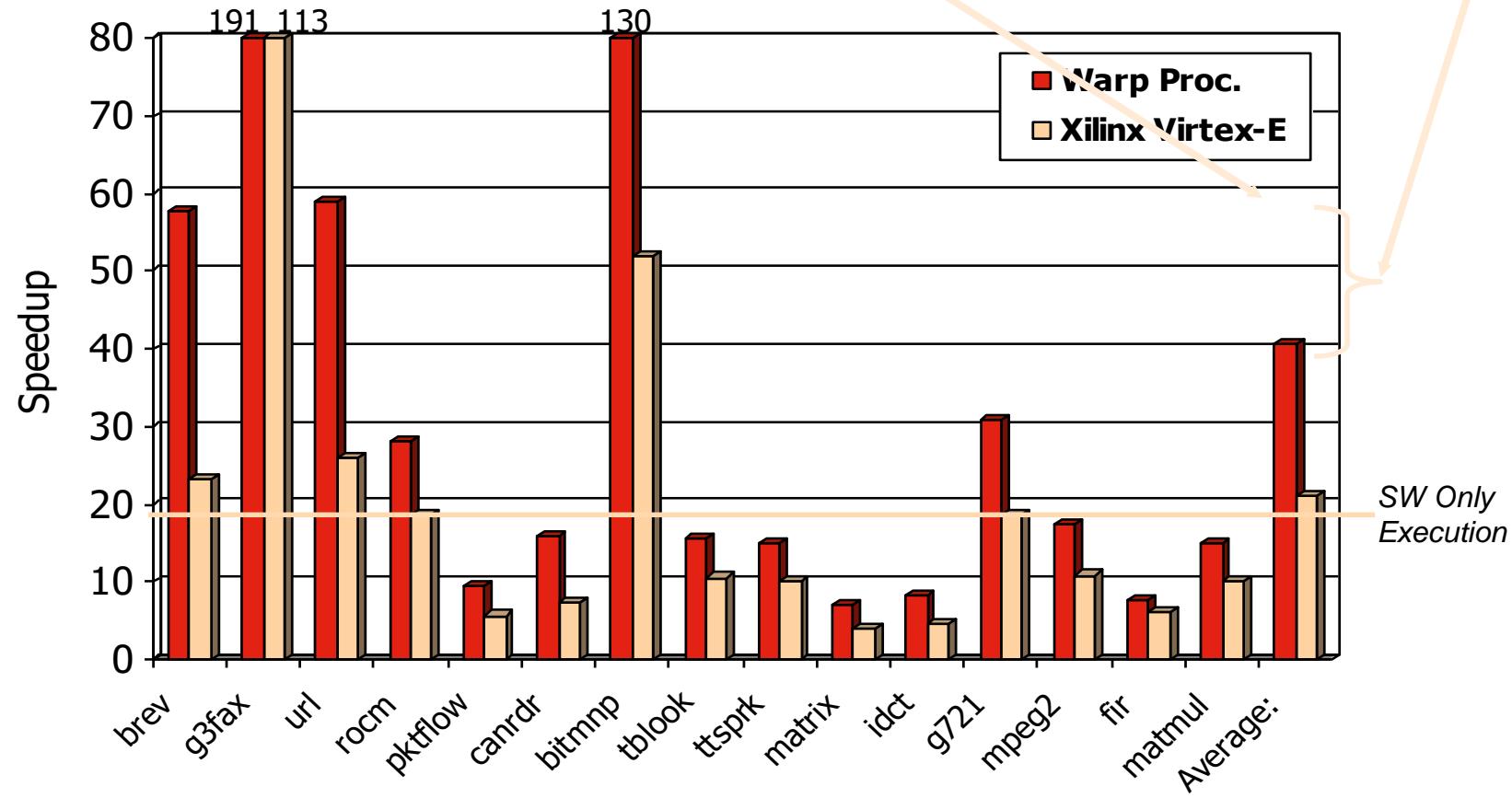
*Performance Speedup (Most Frequent Kernel Only)*

Average kernel speedup of 41,

vs. 21 for Virtex-E

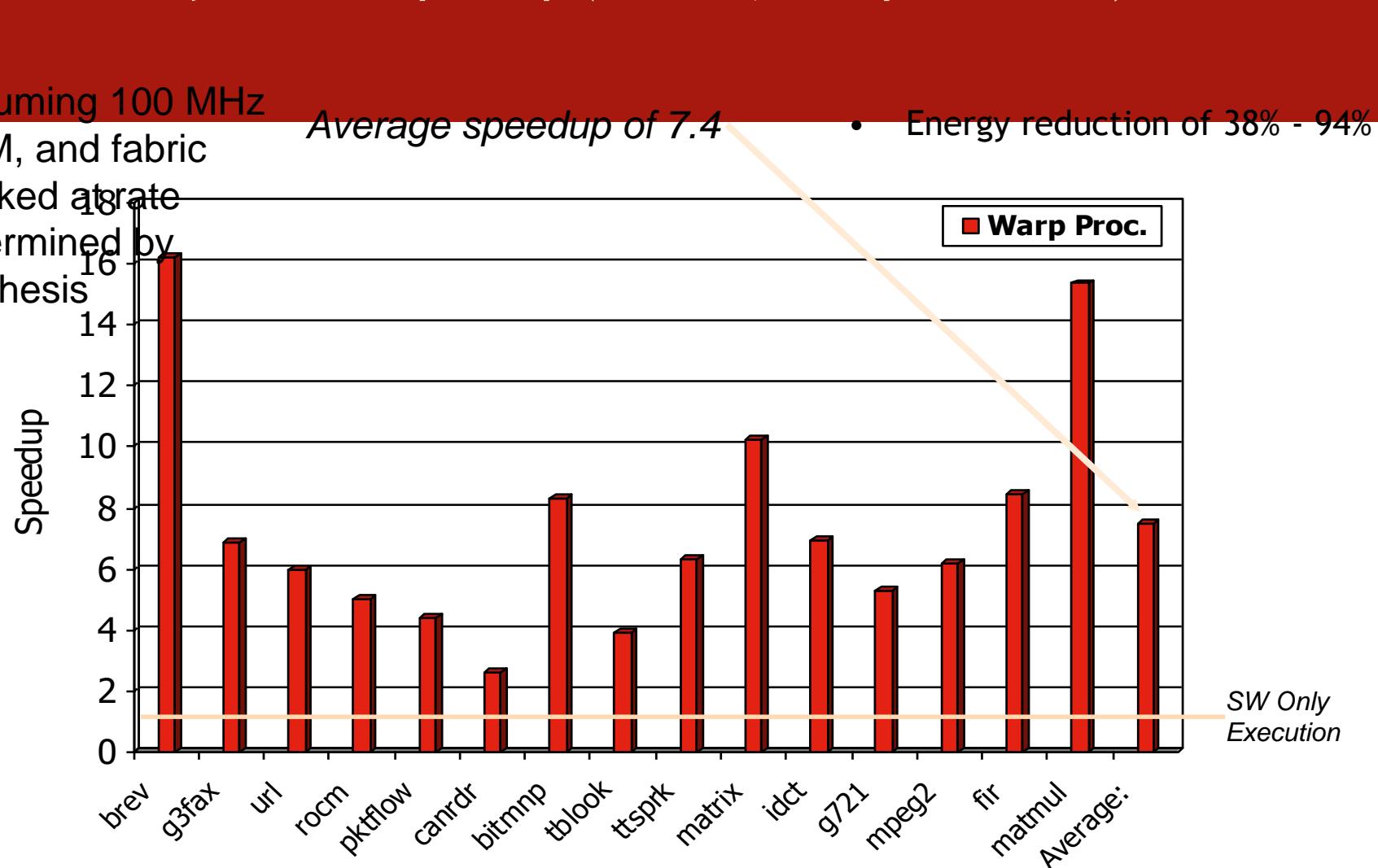
WCLA simplicity results

in faster HW circuits



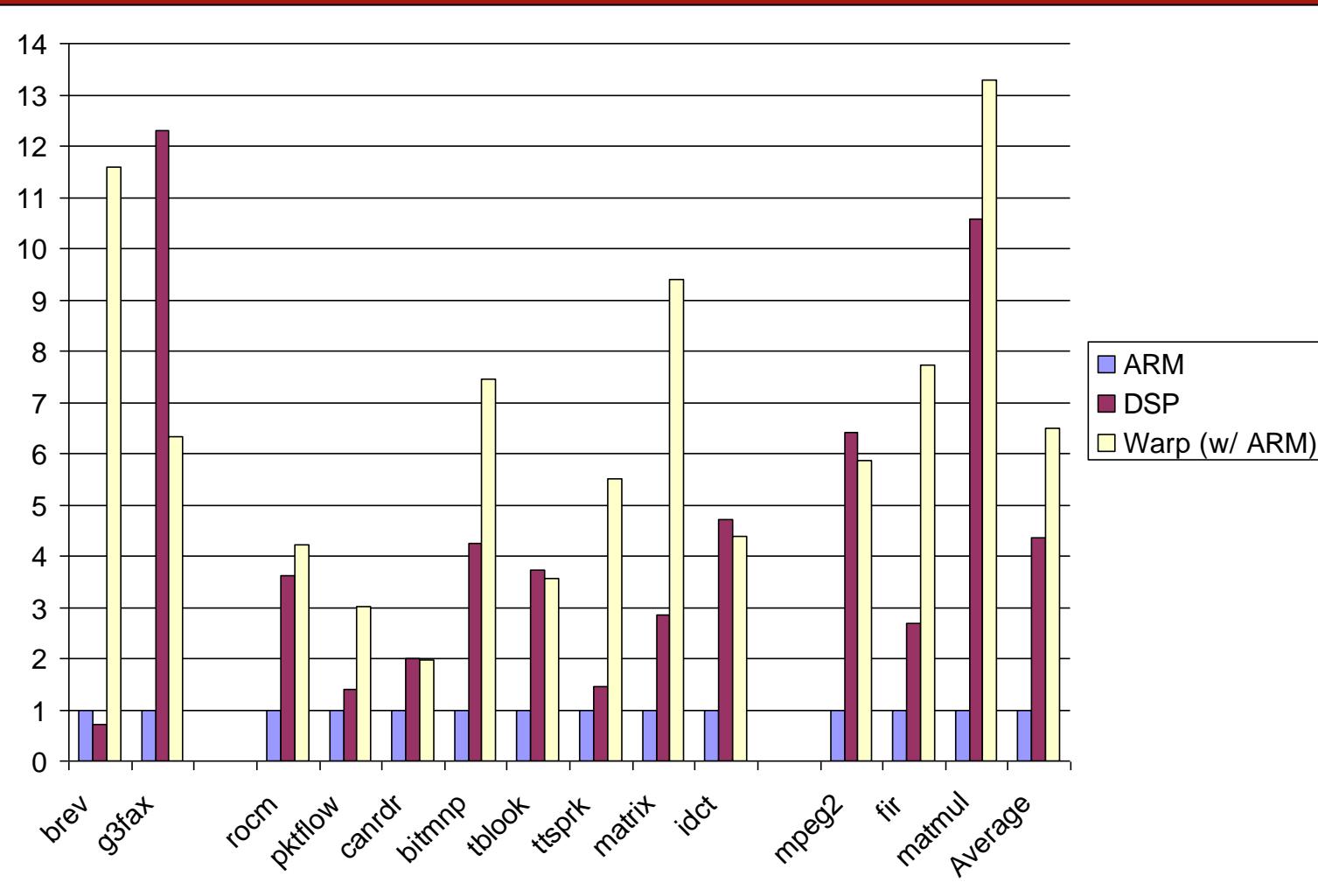
# Warp Processors

## Performance Speedup (Overall, Multiple Kernels)



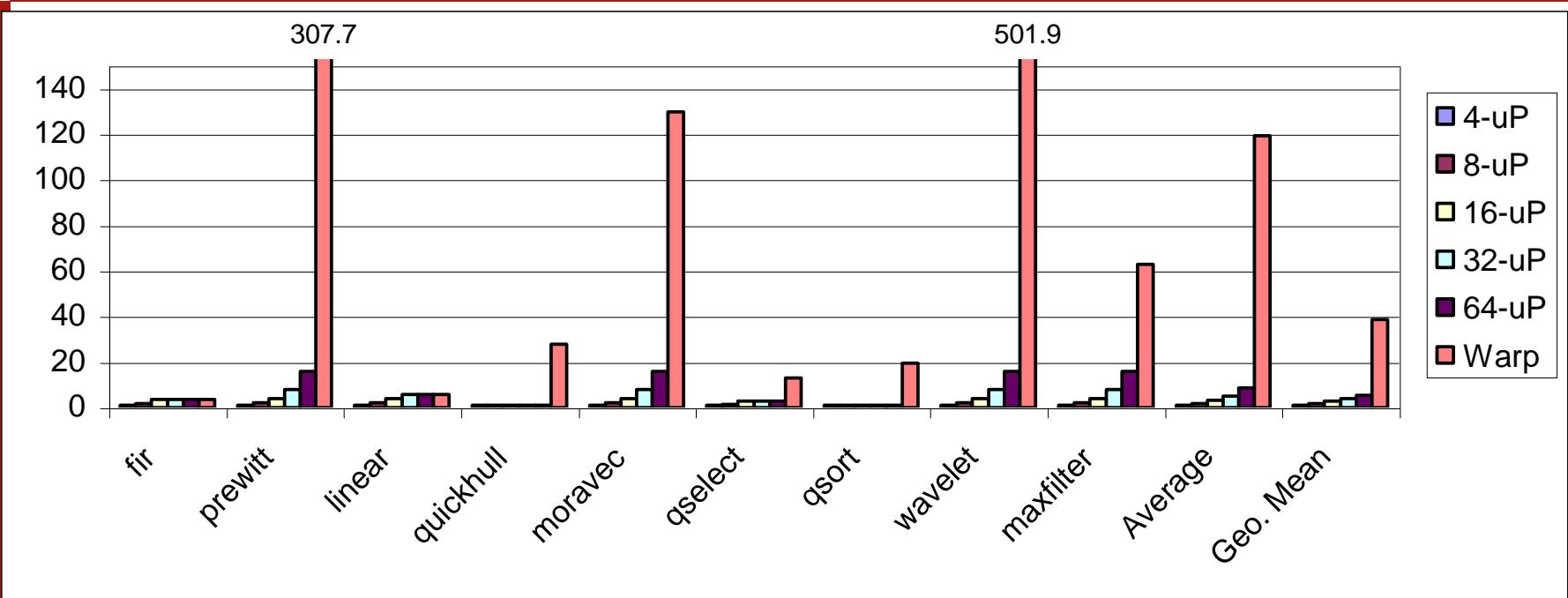
# Warp Processors

*Speedups Compared with Digital Signal Processor*



# Warp Processors

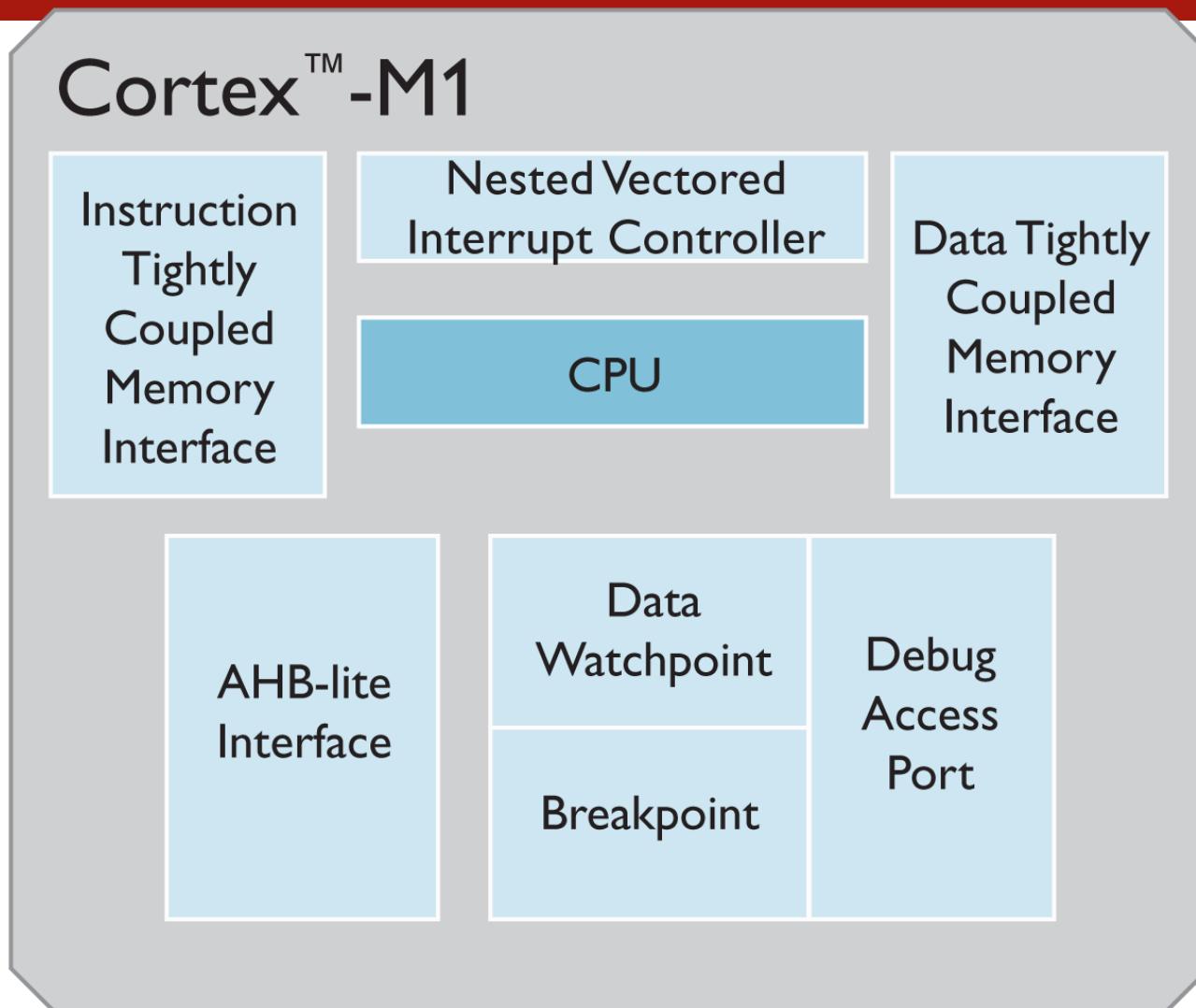
*Speedups for Multi-Threaded Application Benchmarks*



Compelling computing advantage of FPGAs:

*Parallelism from bit level up to processor level, and everywhere in between*

# Cortex-M1





# Cortex-M1

- Developed for the usage with FPGAs
- It is targeting low-cost applications in which costs, ease of use and low interrupt latency are critical



# Cortex-M1

- Features 1/2:
  - Streamlined three stage 32 bit RISC processor
  - High frequency, low area design
  - Configurable instruction and data tightly coupled memories (0K - 1024K)
  - Integrated interrupt controller
  - 1 to 32 interrupts supported
  - 4 priority levels per interrupt
  - 0.8 DMIPS/MHz



# Cortex-M1

- Features 2/2:
  - Highly configurable debug logic
  - Removable debug, breakpoint and watchpoint
  - Big or Little endian configurability
  - Fast or small multiplier configuration options supported
  - AMBA<sup>®</sup> AHB-Lite 32-bit bus interface
  - Executes a subset of Thumb-2 instruction set



# Cortex-M1

FPGA Type	Example	Frequency (MHz)	Area (LUTS)
65 nm	<a href="#">Altera</a> Stratix-III, Xilinx Virtex-5	200	1900
90 nm	Altera Stratix-II, Xilinx Virtex-4	150	2300
65 nm	Altera Cyclone-III	100	2900
90 nm	Altera Cyclone-II, Xilinx Spartan-3	80	2600
130 nm	<a href="#">Actel</a> ProASIC3, Actel Fusion	70	4300 Tiles



# NVIC

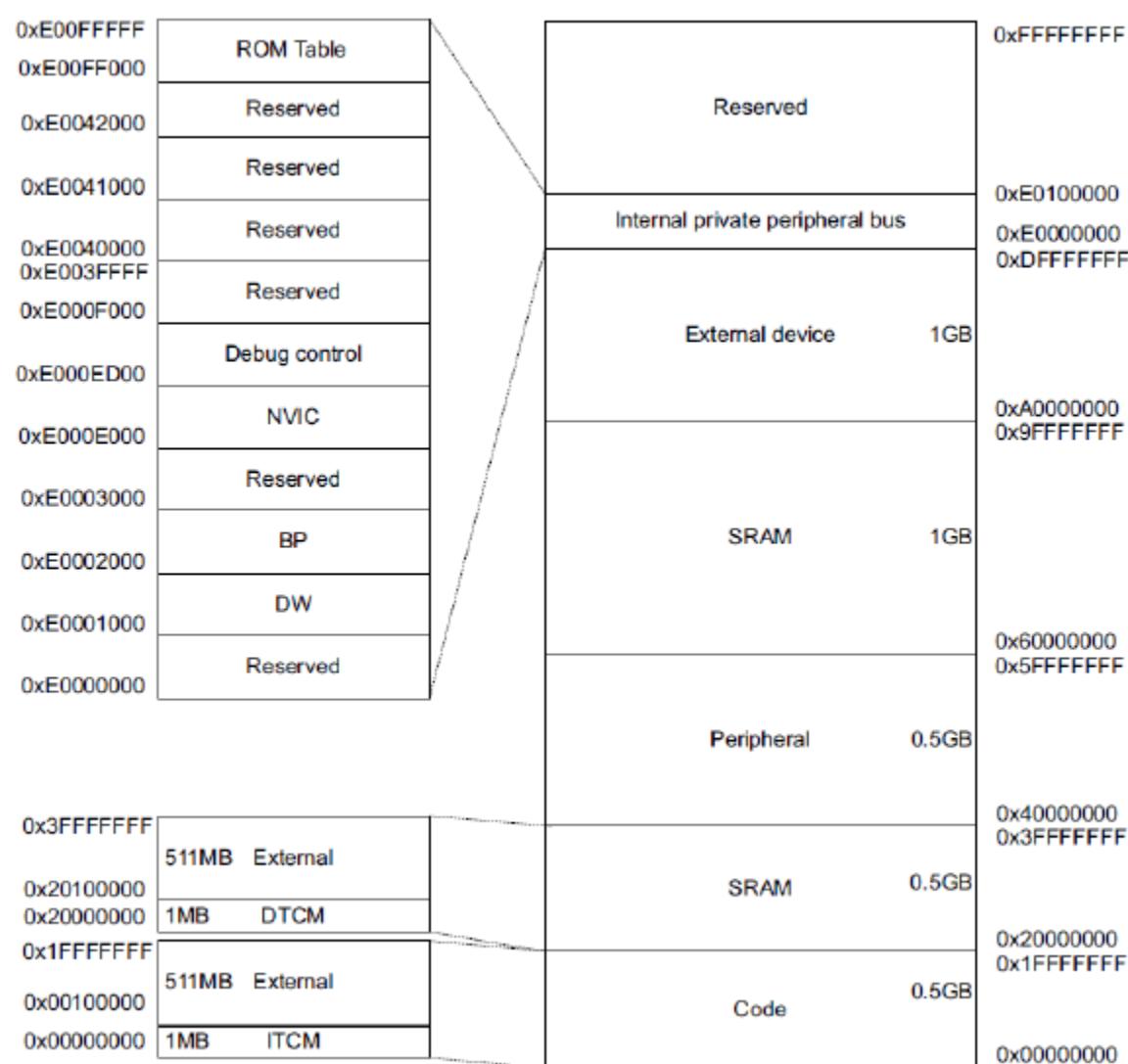
- Depending on the implementation, the NVIC can support 1, 8, 16 or 32 external interrupts with 4 different priority levels
- It supports both level and pulse interrupt sources
- The processor state is automatically saved by hardware on interrupt entry and is restored on interrupt exit
- The vector table for a Cortex-M1 is similar to the Cortex-M3



# Memory Map

- The overall layout of the memory map of a device based around the Cortex-M1 is fixed.
- This allows easy porting of software between different systems based on the Cortex-M1.
- The Cortex-M1 has two Tightly Coupled Memories (TCM) one for instructions and one for data.
- Each of these can be up to 1 megabyte in size
- As the TCMs are implemented in FPGA on-chip RAM, in many cases their contents can be programmed from the FPGA's configuration flash

# Memory Map





# Execution Modes

- The processor supports two operation modes:
  - Thread mode
  - Handler mode.
- Thread mode is entered on reset and is used to execute system initialization and application code.
- Handler mode is entered as a result of an exception. On return from the exception the core will return to Thread mode



# Forward compatibility with Cortex-M3 processor

- The Cortex-M1 processor implements a forward binary compatible subset of the instruction set and features provided by Cortex-M3 processor.
- Software, including system level code, can be easily moved from Cortex-M1 processors to Cortex-M3 processors without the need for recompilation.



**Thank you for your attention**



# References

- [1] Tim Behne, „FPGA Clock Schemes”
- [2] Spartan-6 family documentation; [www.xilinx.com](http://www.xilinx.com)
- [3] [http://www.actel.com/documents/Clock\\_Skew\\_AN.pdf](http://www.actel.com/documents/Clock_Skew_AN.pdf)
- [4] <http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf>
- [5] [http://www.xilinx.com/support/documentation/application\\_notes/xapp094.pdf](http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf)
- [6] [https://nepg.nasa.gov/mapld\\_2009/talks/Posters/Landoll\\_David\\_mapld09\\_pres\\_1.ppt#1019,12,Clock Domain Crossings Guaranteed to Cause Metastability](https://nepg.nasa.gov/mapld_2009/talks/Posters/Landoll_David_mapld09_pres_1.ppt#1019,12,Clock Domain Crossings Guaranteed to Cause Metastability)
- [7] [https://arco.esi.uclm.es/public/doc/tutoriales/Xilinx/old\\_training/FPGA%20Design/achieving-timing-closure.ppt#300,3,Timing Closure](https://arco.esi.uclm.es/public/doc/tutoriales/Xilinx/old_training/FPGA%20Design/achieving-timing-closure.ppt#300,3,Timing Closure)



# References

[8] Tim



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

### Wykład 11 Verilog

Wrocław 2020



# Plan

- Introduction
- Verilog in brief
- VHDL/Verilog comparison
- Examples
- Summary



# Introduction



# Introduction

- At present there are two industry standard hardware description languages, VHDL and Verilog.
- The complexity of ASIC and FPGA designs has meant an increase in the number of specific tools and libraries of macro and mega cells written in either VHDL or Verilog.
- As a result, it is important that designers know both VHDL and Verilog and that EDA tools vendors provide tools that provide an environment allowing both languages to be used in unison.



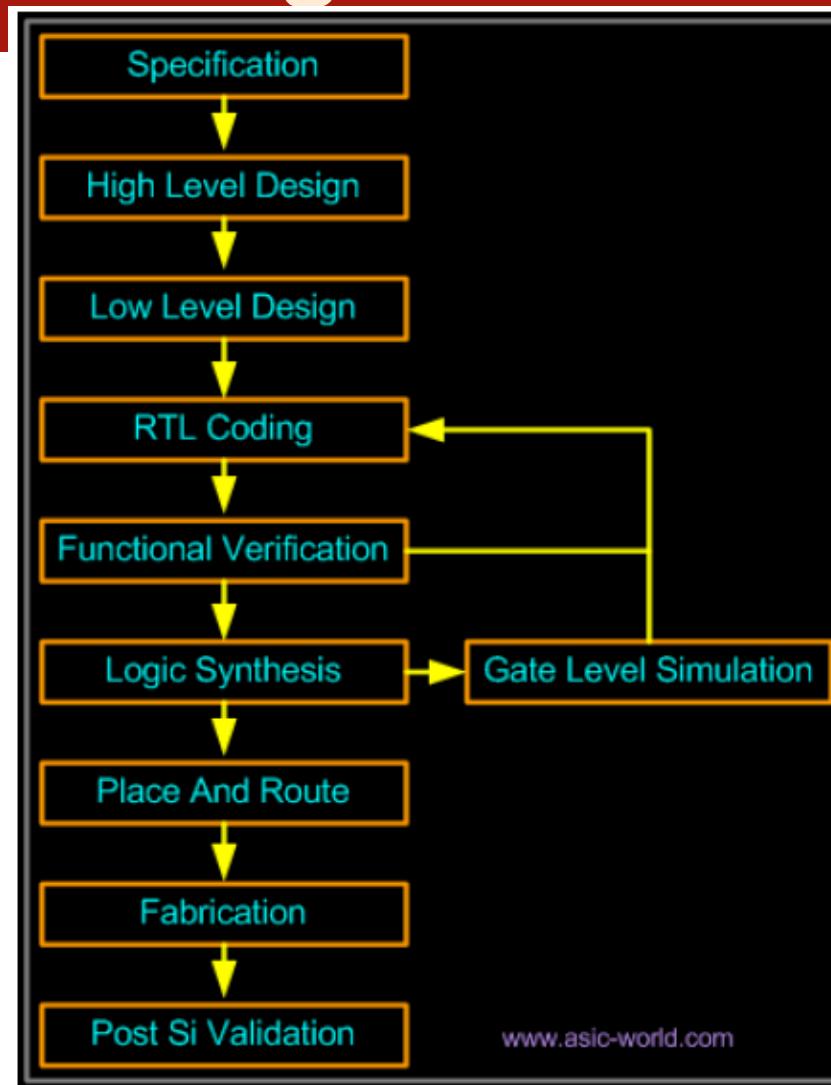
# Introduction

- VHDL (Very high speed integrated circuit Hardware Description Language) became IEEE standard 1076 in 1987.
- The Verilog hardware description language has been used far longer than VHDL and has been used extensively since it was launched by Gateway in 1983.
- Cadence bought Gateway in 1989 and opened Verilog to the public domain in 1990.
- Verilog became IEEE standard 1364 in December 1995



# Verilog in brief

# Verilog – design flow





# Verilog – abstraction levels

- Verilog supports three main abstraction levels:
  - Behavioral level – a system is described by concurrent algorithm
  - Register-transfer level – a system is characterised by operations and transfer of data between registers according to an explicit clock
  - Gate level – a system is described by logical links and their timing characteristics



# Verilog – „Hello world”

```
//////////.
module HelloWorld;

initial begin
    $display("Hello World");
    #10 $finish;
end

endmodule

//////////.
module test(
    input clk,
    input rst,
    output q
);

endmodule
```



# Verilog – 8-bit counter example

```
//-----
module counter(
clk,
rst,
cnt_out
);
//End of port list
//-----Input ports-----
input clk;
input rst;
//-----Output ports-----
output [7:0] cnt_out;
//----Input port types-----
//by rule all the input ports should be wires
wire clk;
wire rst;
//----Output port types-----
//output port types can be a wire or a storage element (reg)
reg [7:0] cnt_out;

//----Code starts here-----
always @ (posedge clk)
begin : COUNTER //block name
    if (reset == 1'b1) begin
        cnt <= #1 8b'00000000;
    end
    else
        cnt <= #1 cnt + 1;
    end
end

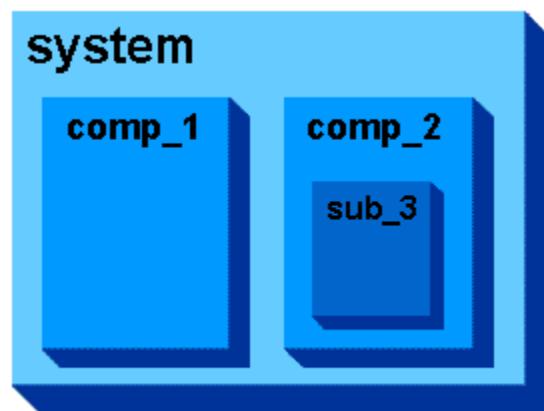
endmodule
```



# Basic constructs

# Verilog - hierarchy

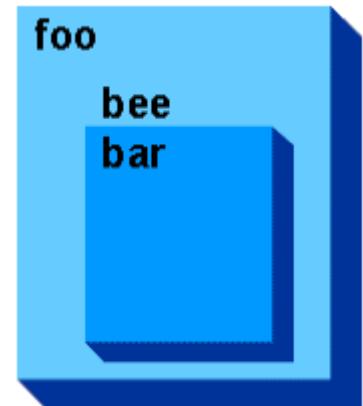
- Verilog structures which build the hierarchy are:
  - modules
  - ports
- A module is the basic unit of the model, and it may be composed of **instances** of other modules



# Verilog - hierarchy

- the top level module is not instantiated by any other module
- Example:

```
module foo;  
    bar bee (port1, port2);  
endmodule  
module bar (port1, port2);  
    ...  
endmodule
```





# Verilog

- Port types in Verilog:

- Input
  - Output
  - Inout

- Matching ports by names:

*foo f1 (.bidi(bus), .out1(sink1), .in1(source1));*

versus normal way

*foo f1 (source1, , sink1, , bus);*



# Verilog - modules

- Verilog models are made up of modules
- Modules are made of different types of components:
  - Parameters
  - Nets
  - Registers
  - Primitives and Instances
  - Continuous Assignments
  - Procedural Blocks
  - Task/Function definitions



# Verilog - parameters

- *Parameters* are constants whose values are determined at compile-time
- They are defined with the parameter statement:

```
parameter identifier = expression;
```

Example:

```
parameter width = 8, msb = 7, lsb = 0;
```



# Verilog - nets

- Nets are the things that connect model components together – like signals in VHDL
- Nets are declared in statements like this:

```
net_type [range] [delay3] list_of_net_identifiers
```

- Example:

```
wire w1, w2;
```

```
tri [31:0] bus32;
```

# Verilog – types of nets

Net Data Type	Functionality
wire, tri	Interconnecting wire – no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand,triand	Wired outputs AND together (models open-collector)
tri0,tri1	Net pulls-down or pulls-up when not driven
supply0,supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	

- Each net type has functionality that is used to model different types of hardware such as CMOS, NMOS, TTL etc



# Verilog – net drivers

- Nets are driven by net drivers.
- Drivers may be:
  - output port of a primitive instance
  - output port of a module instance
  - left-hand side of a continuous assignment
- There may be more than one driver on a net
- If there is more than one driver, the value of the net is determined by a **built-in** resolution function



# Verilog - registers

- *Registers* are storage elements
- Values are stored in registers in procedural assignment statements
- Registers can be used as the source for a primitive or module instance (i.e. registers can be connected to input ports), but they cannot be driven in the same way a net can
- Examples:
  - `reg r1, r2;`  
`reg [31:0] bus32;`  
`integer i;`



# Verilog - registers

- There are four types of registers:
  - **Reg:**
    - This is the generic register data type. A reg declaration can specify registers which are 1 bit wide to 1 million bits wide
  - **Integer**
    - Integers are 32 bit signed values
  - **Time**
    - Registers declared with the time keyword are 64-bit unsigned integers
  - **Real (and Realtime)**
    - Real registers are 64-bit IEEE floating point



# Verilog - memories

- Verilog allows arrays of registers, called memories
- Memories are static, single-dimension arrays
- The format of a memory declaration is:
  - reg [range] identifier range ;
- Example:
  - reg [0:31] temp, mem[1:1024];
  - ...
  - temp = mem[10]; --extract 10th element
  - bit = temp[3]; --extarct 3rd bit

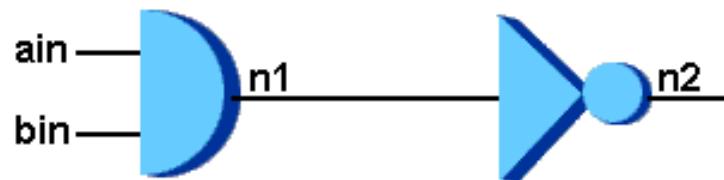


# Verilog - primitives

- Primitives are pre-defined module types
- The Verilog primitives are sometimes called gates, because for the most part, they are simple logical primitives
- Examples:
  - And, nand, or, nor, xor, xnor
  - Buf, not
  - Pullup, pulldown
  - bufif0, notif0

# Verilog - primitives

- Examples:



```
module test;
    wire n1, n2;
    reg ain, bin;

    and and_prim(n1, ain, bin);
    not not_prim(n2, n1);

endmodule
```



# Procedural blocks



# Verilog - Continuous assignments

- Continuous assignments are known as data flow statements
- They describe how data moves from one place, either a net or register, to another
- They are usually thought of as representing combinational logic
- Examples:

assign w1 = w2 & w3;

assign (strong1, pull0) mynet = enable;



# Verilog - Procedural blocks

- Procedural blocks are the part of the language which represents sequential behavior
- A module can have as many procedural blocks as necessary
- These blocks are sequences of executable statements
- The statements in each block are executed sequentially, but the blocks themselves are concurrent and asynchronous to other blocks



# Verilog - Procedural blocks

- There are two types of procedural blocks, *initial blocks* and *always blocks*
- All initial and always blocks contain a single statement, which may be a compound statement, e.g.:

initial

```
begin statement1 ; statement2 ; ... end
```



# Verilog - Initial blocks

- All initial blocks begin at time 0 and execute the initial statement
- Because the statement may be a compound statement, this may entail executing lots of statements
- An initial block may cause activity to occur throughout the entire simulation of the model
- When the initial statement finishes execution, the initial block terminates



# Verilog - Initial blocks

- Examples:

```
initial x = 0; // a simple initialization
```

```
initial begin
    x = 1;      // an initialization
    y = f(x);
```

```
#1 x = 0;      // a value change 1 time unit later
    y = f(x);
```

```
end
```



# Verilog – Always block

- Always blocks also begin at time 0
- The only difference between an always block and an initial block is that when the always statement finishes execution, it starts executing again



# Tasks and functions



# Verilog – Tasks/functions

- Tasks and functions are declared within modules
- Tasks may only be used in procedural blocks
- A task invocation is a statement by itself. It may not be used as an operand in an expression
  
- Functions are used as operands in expressions
- A function may be used in either a procedural block or a continuous assignment, or indeed, any place where an expression may appear



# Verilog – Tasks/functions

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value.



# Verilog – Tasks

- Tasks may have zero or more arguments, and they may be input, output, or inout arguments
- Time can elapse during the execution of a task, according to time and event controls in the task definition



# Verilog – Tasks

```
module this_task;
    task my_task;
        input a, b;
        inout c;
        output d, e;
        reg foo1, foo2, foo3;
        begin
            <statements>          // the set of statements that
                                   // performs the work of the task
            c = foo1;              // the assignments that initialize
            d = foo2;              // the results variables
            e = foo3;
        end
    endtask
endmodule
```



# Verilog – Functions

- In contrast to tasks, functions must execute in a single instant of simulated time
- That is, not time or delay controls are allowed in a function
- Function arguments are also restricted to inputs only.
- Output and inout arguments are not allowed.
- The output of a function is indicated by an assignment to the function name



# Verilog – Functions

- Example:

```
function [15:0] relocate;  
    input [11:0] addr;  
    input [3:0] relocation_factor;  
    begin  
        relocate = addr + (relocation_factor<<12);  
        count = count + 1;  
    endfunction  
  
assign absolute_address = relocate(relative_address,  
rf);
```



# VHDL/Verilog comparison



# Capability

- VHDL – like Pascal or Ada programming languages
- Verilog – like C programming language
- It is important to remember that both are **Hardware Description Languages** and not programming languages
- For synthesis only a subset of languages is used

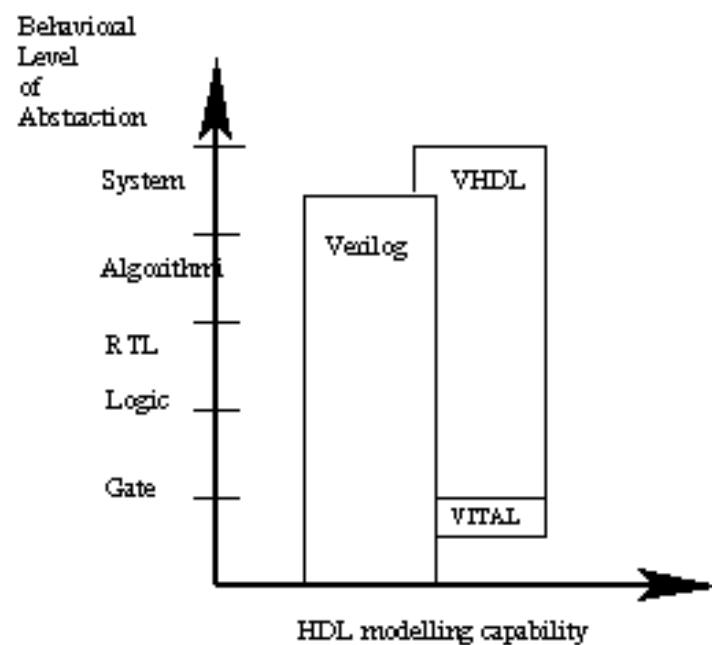


# Capability

- Hardware structure can be modeled equally effectively in both VHDL and Verilog.
- When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the Programming Language Interface (PLI) – Verilog addon .
- The choice of which to use is not therefore based solely on technical capability but on:
  - personal preferences
  - EDA tool availability
  - commercial, business and marketing issues

# Capability

- The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction



Source: [1]



# Compilation

- VHDL:
  - Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired.
  - It is good design practice to keep each design unit in its own system file in which case separate compilation should not be an issue.



# Compilation

- Verilog:
  - The Verilog language is still rooted in its native interpretative mode.
  - Compilation is a mean of speeding up simulation, but has not changed the original nature of the language.
  - Care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files.
  - Simulation results can change by simply changing the order of compilation.



# Data types

- VHDL:
  - A multitude of language or user defined data types can be used.
  - This may mean dedicated conversion functions are needed to convert objects from one type to another.
  - The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types.
  - VHDL may be preferred because it allows a multitude of language or user defined data types to be used.



# Data types

- Verilog:
  - Compared to VHDL, Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling.
  - Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user.
  - There are net data types, for example wire, and a register data type called reg.
  - A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit.
  - Verilog may be preferred because of its simplicity.



# Design reusability

- VHDL:
  - Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them
- Verilog:
  - There is no concept of packages in Verilog.
  - Functions and procedures used within a model must be defined in the module.
  - To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive.



# Ease of learning

- Starting with zero knowledge of either language, Verilog is probably the easiest to grasp and understand.
- VHDL may seem less intuitive at first for two primary reasons:
  - First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase.
  - Second, there are many ways to model the same circuit, specially those with large hierarchical structures



# High level constructs

- VHDL:
  - There are more constructs and features for high-level modeling in VHDL than there are in Verilog.
  - Abstract data types can be used along with the following statements:
    - package statements for model reuse,
    - configuration statements for configuring design structure,
    - generate statements for replicating structure,
    - generic statements for generic models that can be individually characterized, for example, bit width.
  - All these language statements are useful in synthesizable models.



# High level constructs

- Verilog:
  - Except for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modeling statements in Verilog



# Libraries

- VHDL:
  - A library is a store for compiled entities, architectures, packages and configurations. Useful for managing multiple design projects.
- Verilog:
  - There is no concept of a library in Verilog. This is due to it's origins as an interpretive language.



# Low level constructs

- VHDL:
  - Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR.
  - Any timing must be separately specified using the after clause.
  - Separate constructs defined under the VITAL language must be used to define the cell primitives of ASIC and FPGA libraries.



# Low level constructs

- Verilog:
  - The Verilog language was originally developed with gate level modeling in mind, and so has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries.
  - Examples include User Defined Primitives (UDP), truth tables and the specify block for specifying timing delays across a module.



# Managing large designs

- VHDL:
  - Configuration, generate, generic and package statements all help manage large design structures.
- Verilog:
  - There are no statements in Verilog that help manage large designs



# Operators

- The majority of operators are the same between the two languages.
- Verilog does have very useful unary reduction operators that are not in VHDL.
- A loop statement can be used in VHDL to perform the same operation as a Verilog unary reduction operator.
- VHDL has the mod operator that is not found in Verilog.



# Procedures and tasks

- VHDL:
  - concurrent procedure calls are allowed
- Verilog:
  - concurrent procedure calls are not allowed



# Readability

- This is more a matter of coding style and experience than language feature.
- VHDL is a concise and verbose language;
- Verilog is more like C because it's constructs are based approximately 50% on C and 50% on Ada.
- For this reason an existing C programmer may prefer Verilog over VHDL.
- Whatever HDL is used, when writing or reading an HDL model to be synthesized it is important to think about **hardware intent**.



# Structural replication

- VHDL:
  - The generate statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately.
- Verilog:
  - There is no equivalent to the generate statement in Verilog.



# Verboseness

- VHDL:
  - Because VHDL is a very strongly typed language models must be coded precisely with defined and matching data types.
  - This may be considered an advantage or disadvantage.
  - It does mean models are often more verbose, and the code often longer, than it's Verilog equivalent.



# Verboseness

- Verilog:
  - Signals representing objects of different bits widths may be assigned to each other.
  - The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits, and is independent of whether it is the assigned signal or not.
  - Unused bits will be automatically optimized away during the synthesis process.
  - This has the advantage of not needing to model quite so explicitly as in VHDL, but does mean unintended modeling errors will not be identified by an analyzer.



# Examples



# Binary up counter

- VHDL:

```
process (clock)
begin
    if clock='1' and clock'event then
        counter <= counter + 1;
    end if;
end process;
```



# Binary up counter

- Verilog:

```
reg [upper:0] counter;  
  
always @ (posedge clock)  
    counter <= counter + 1;
```



# D FlipFlop

- VHDL:

```
process (<clock>)
begin
    if <clock>'event and <clock>='1'
then
    <output> <= <input>;
end if;
end process;
```



# D FlipFlop

- Verilog:

```
always @ (posedge <clock>) begin  
    <reg> <= <signal>;  
end
```



# Synchronous multiplier

- VHDL:

```
process (<clock>)
begin
    if <clock>='1' and <clock>'event
then
        <output> <= <input1> * <input2>;
    end if;
end process;
```



# Synchronous multiplier

- Verilog:

```
wire [17:0] <a_input>;
wire [17:0] <b_input>;
reg    [35:0] <product>;

always @ (posedge <clock>)
    <product> <= <a_input> *
<b_input>;
```



# Summary



	VHDL	Verilog
Strong typing	Yes	No
User-defined types	Yes	No
Dynamic memory allocation	Yes	No
Physical types	Yes	No
Enumerated types	Yes	No
Records/structs	Yes	No



	VHDL	Verilog
Bit (vector) / integer equivalence	Partial (by libraries)	Yes
Subprograms	Yes	Yes
Separate packaging	Yes Packages	Yes Include files
Gate level modeling	Yes VITAL	Yes Builtin primitives



	VHDL	Verilog
Conditional statements	<p>Yes</p> <ul style="list-style-type: none"><li>• If-then-else/elsif (priority)</li><li>• Case (mux)</li><li>• Selected assign (mux)</li><li>• Conditional assign (priority)</li><li>• No “don’t care” matching capability</li></ul>	<p>Yes</p> <ul style="list-style-type: none"><li>• if-else (priority)</li><li>• case (mux)</li><li>• casex (mux)</li><li>• ?: (conditional used in concurrent assignments)</li></ul>



	VHDL	Verilog
Iteration	<p>Yes</p> <ul style="list-style-type: none"><li>• Loop</li><li>• while-loop</li><li>• for-loop</li><li>• exit</li><li>• next</li></ul>	<p>Yes</p> <ul style="list-style-type: none"><li>• repeat</li><li>• for</li><li>• while</li></ul>



**Thank you for your attention**



# References

- [1] Douglas J. Smith, „VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and C”
- [2] [http://www.stanford.edu/class/ee183/handouts\\_win2003/  
VerilogQuickRef.pdf](http://www.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf)
- [3] [www.asic-world.com](http://www.asic-world.com)
- [4] [http://www.ece.umd.edu/courses/enee359a/verilog\\_tutorial.pdf](http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf)
- [5] [http://www.eecis.udel.edu/~elias/verilog/verilog\\_manuals/chap\\_9  
.pdf](http://www.eecis.udel.edu/~elias/verilog/verilog_manuals/chap_9.pdf)



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

### Wykład 8 FPGA - taktowanie

Wrocław 2015



# Plan

- Introduction
- Definitions
  - Clock skew
  - Metastability
- FPGA clocking resources
  - DCM
  - PLL
- Clocking constraints



# Introduction

- One of the most important steps in the design process is to identify how many different clocks to use and how to route them



# Introduction

- As larger designs are implemented in FPGAs, it is likely that many of them will have multiple data paths running on multiple clocks
- An FPGA design that contains multiple clocks requires special attention
- Issues to focus on are:
  - maximum clock rates and skew,
  - maximum number of clocks,
  - asynchronous clock design,
  - clock/data relationships.



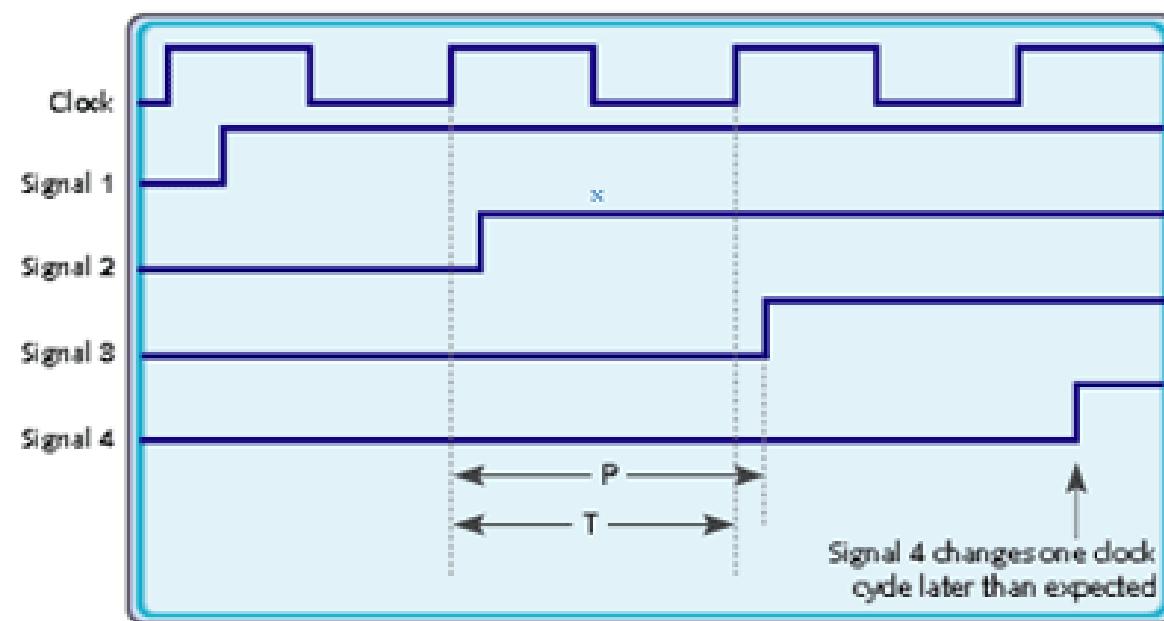
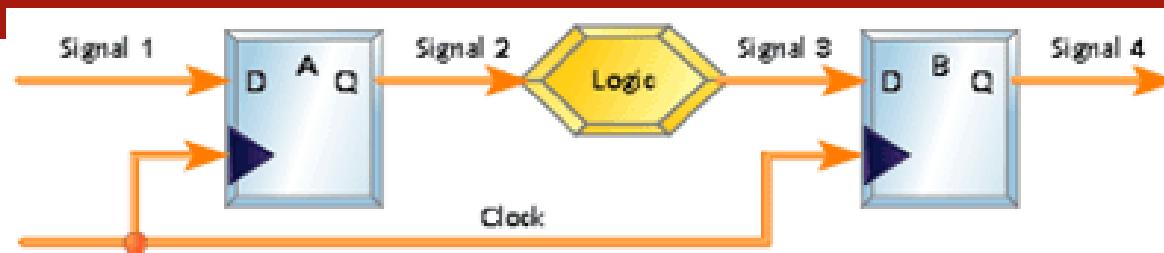
# Definitions



# Definitions

- The first step in any FPGA design is to decide what clock speed is needed within the FPGA
- The fastest clock in the design will determine the clock rate that the FPGA must be able to handle
- The maximum clock rate is determined by the propagation time,  $P$ , of a signal between two flip-flops in the design.
- If  $P$  is greater than the clock period,  $T$ , then when the signal changes at one flip-flop, it doesn't change at the next stage of logic until two clock cycles later.

# Definitions



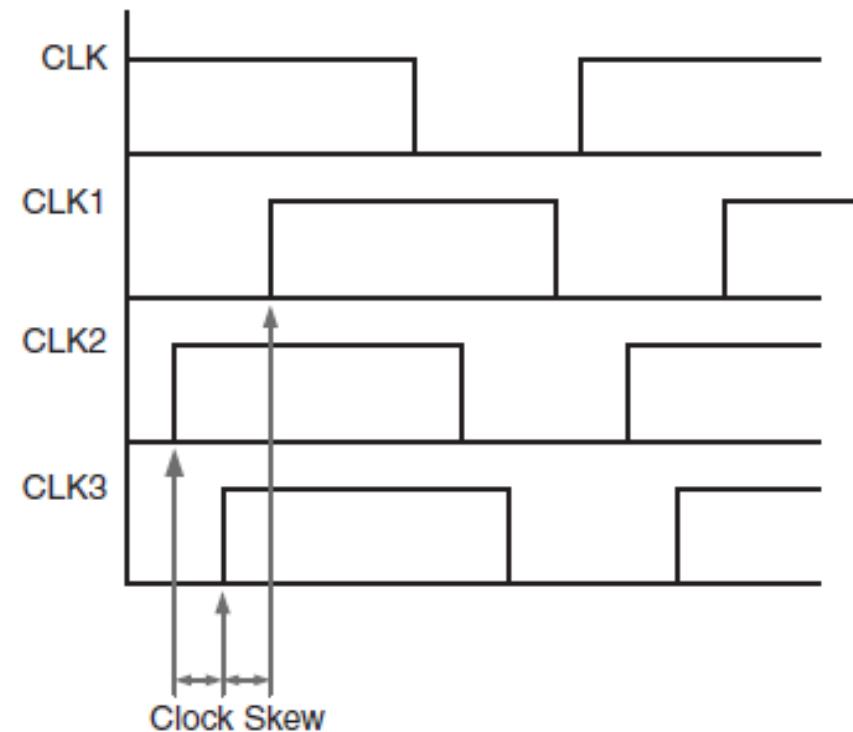
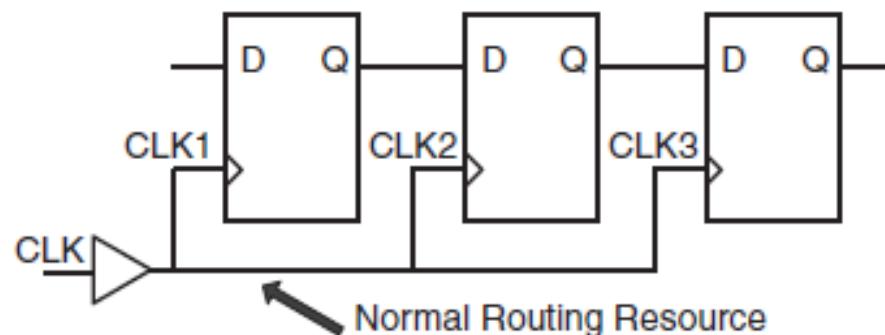


# Definitions

- The ***propagation time*** is the sum of:
  - the hold time required for the signal to change at the output of the first flip-flop,
  - the delay of any combinatorial logic between stages,
  - the routing delay between stages,
  - the set-up time for the signal going into the flip-flop at the second stage.

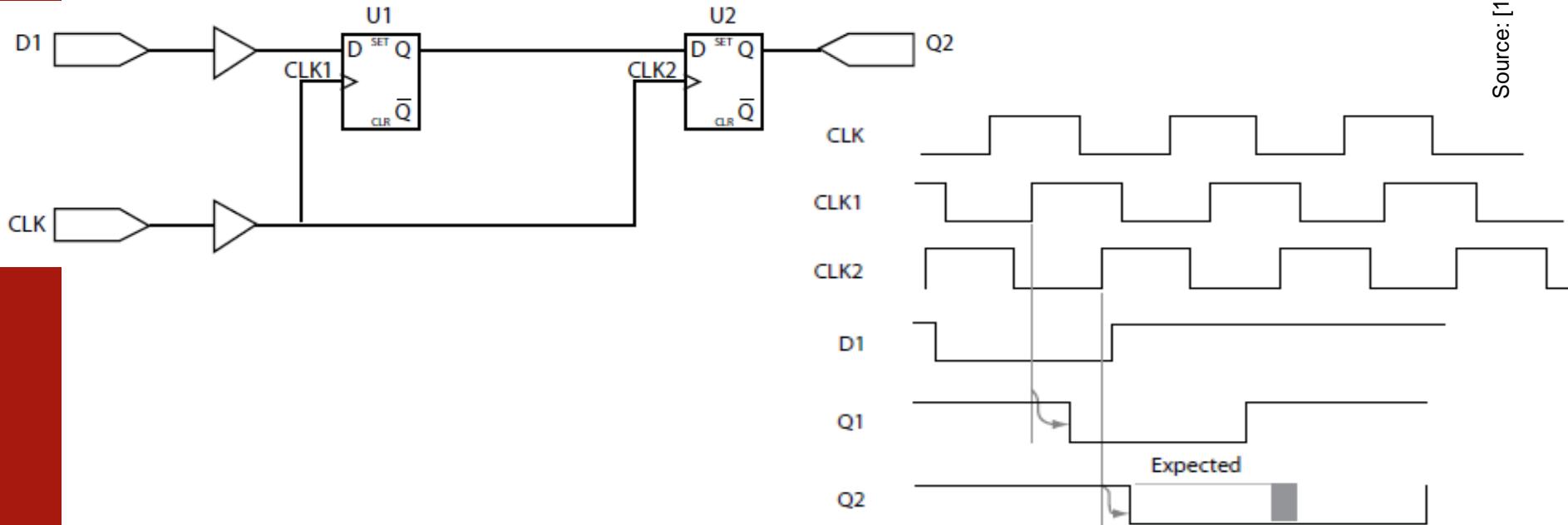
# Clock Skew

- The clock *skew*,  $S$ , is the maximum delay from the clock input of one flip-flop to the clock input of another flip-flop.



# Clock Skew

- For the circuit to work properly, the skew must be less than the propagation time between the two flip-flops. Otherwise we have ***short-path problem***
- Each clock used in an FPGA design, no matter the rate of the clock, must have low skew.





# Clock Skew - reduction

- The short-path problem is created by the existence of an unacceptably large clock skew
- Minimizing the clock skew is the best approach to reduce the risk of short-path problems
- The best way to minimize the clock skew is to use hardware resources like Delay Locked Loop or global clock networks

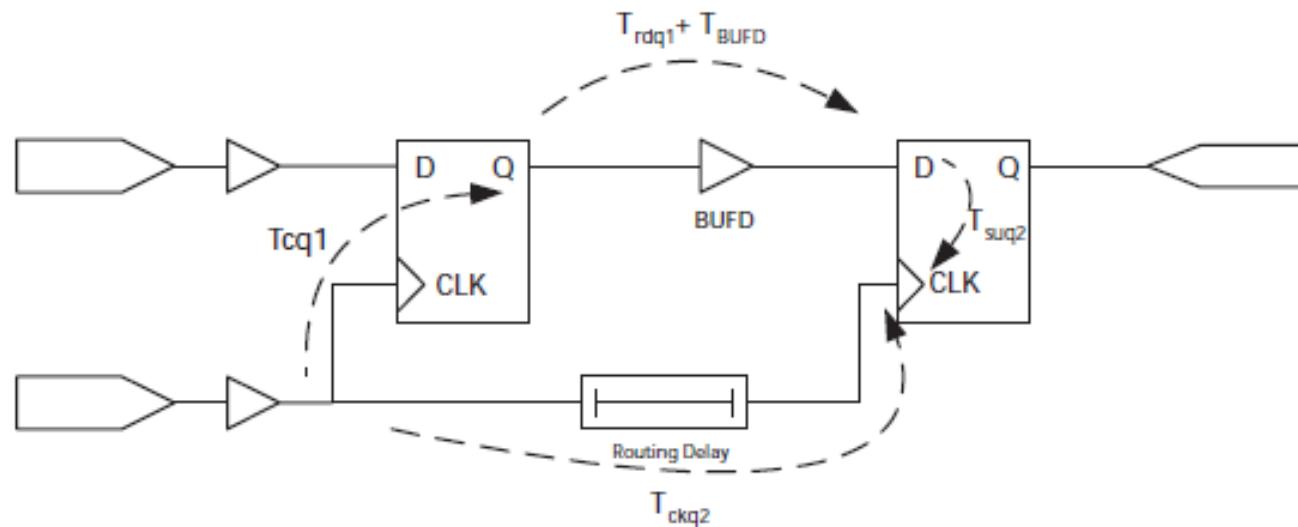


# Clock Skew - reduction

- In cases where designs include multiple clock domains, there may not be enough low-skew global resources in the targeted FPGA for all the external/internal clock signals
- Therefore, regular routing resources and buffers are used to build clock trees for the clock network – possibility of a noticeable clock skew

# Clock Skew - reduction

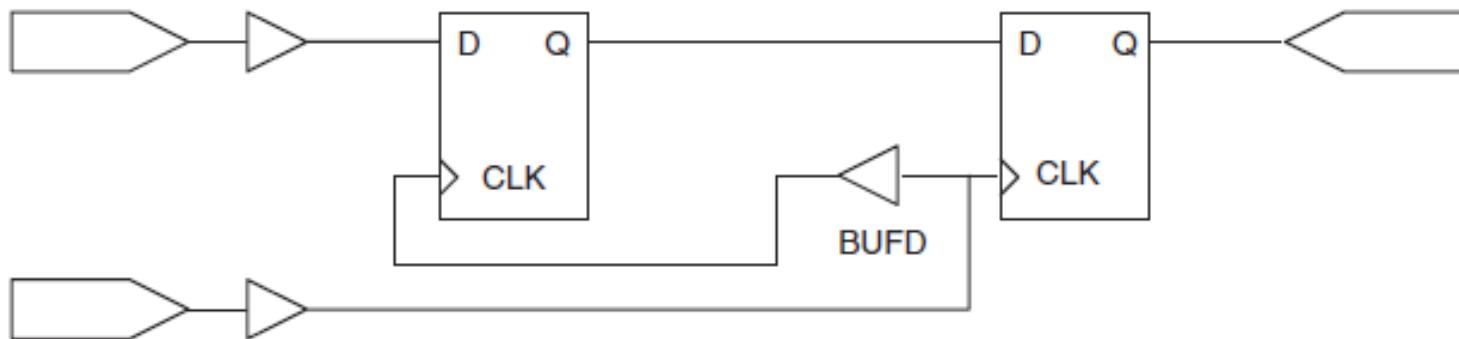
- Methods of manual clock skew reduction:
  - *Adding Delay in Data Path:*
    - the data path propagation time must be greater than the clock skew



$$T_{cq1} + T_{rdq1} + (n \times T_{BUFDO}) + T_{suq2} > T_{ckq2}$$

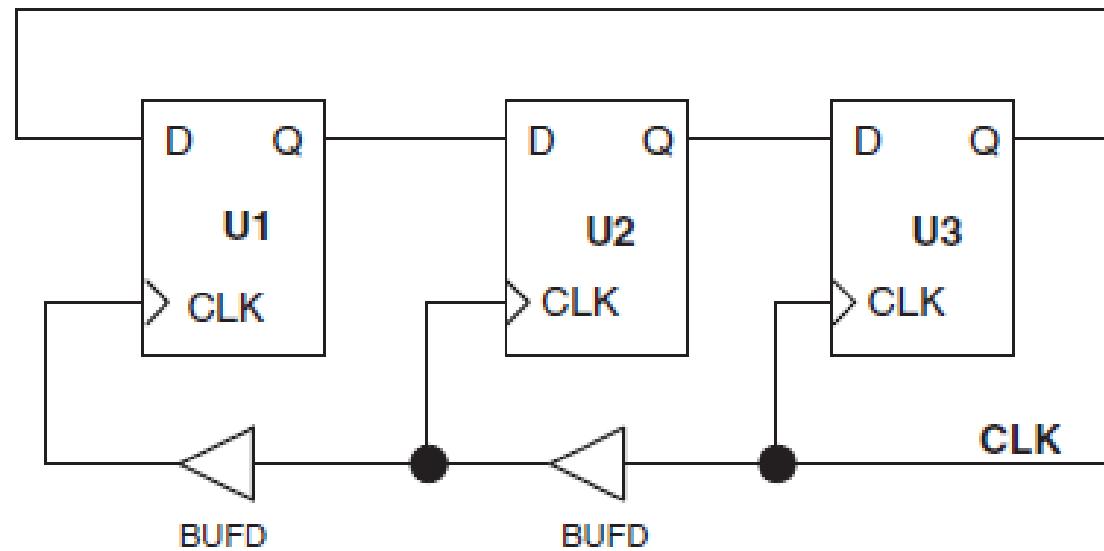
# Clock Skew - reduction

- Methods of manual clock skew reduction:
  - *Clock Reversing*:
    - the clock signal arrives at the clock port of the sink (receiving) register sooner than the source (transmitting) register



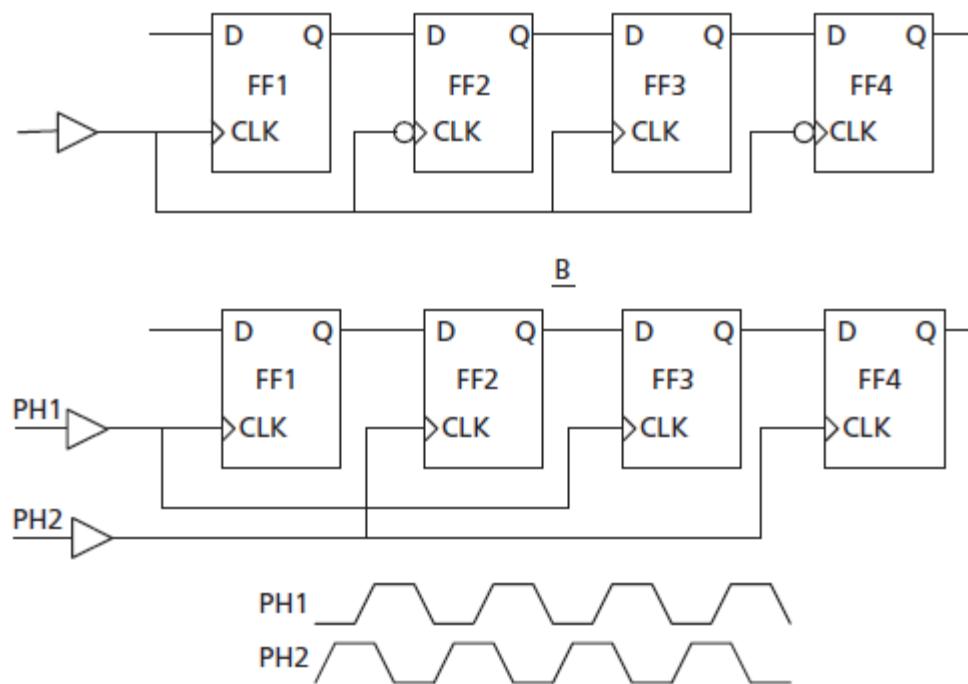
# Clock Skew - reduction

- Methods of manual clock skew reduction:
  - *Alternate Phase Clocking* - Clocking on alternate edges:
    - sequentially adjacent registers are clocked on the opposite edges of the clock



# Clock Skew - reduction

- Methods of manual clock skew reduction:
  - *Alternate Phase Clocking* - Clocking with two phases:
    - a set of adjacent registers are alternately clocked on two different phases of the same clock





# Metastability

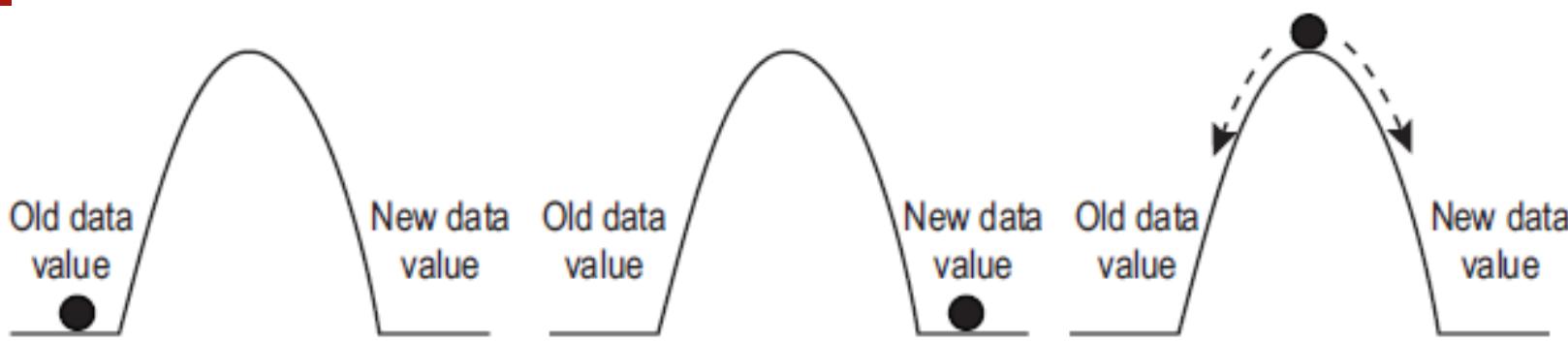
- One of the most serious problems associated with multiple clock designs is when two stages of logic are combined using asynchronous clocks
- Asynchronous logic can create metastable states that can seriously degrade the performance of the design or completely destroy the functionality.



# Metastability

- A metastable state is created when the flip-flop's timing requirements (setup and hold times) are violated.
- The resulting output of the flip-flop is unknown, and can make the entire design nondeterministic.
- If one stage of logic asynchronously feeds data to another, it is difficult, if not impossible to meet the set-up and hold-time requirements of the flip-flop.

# Metastability

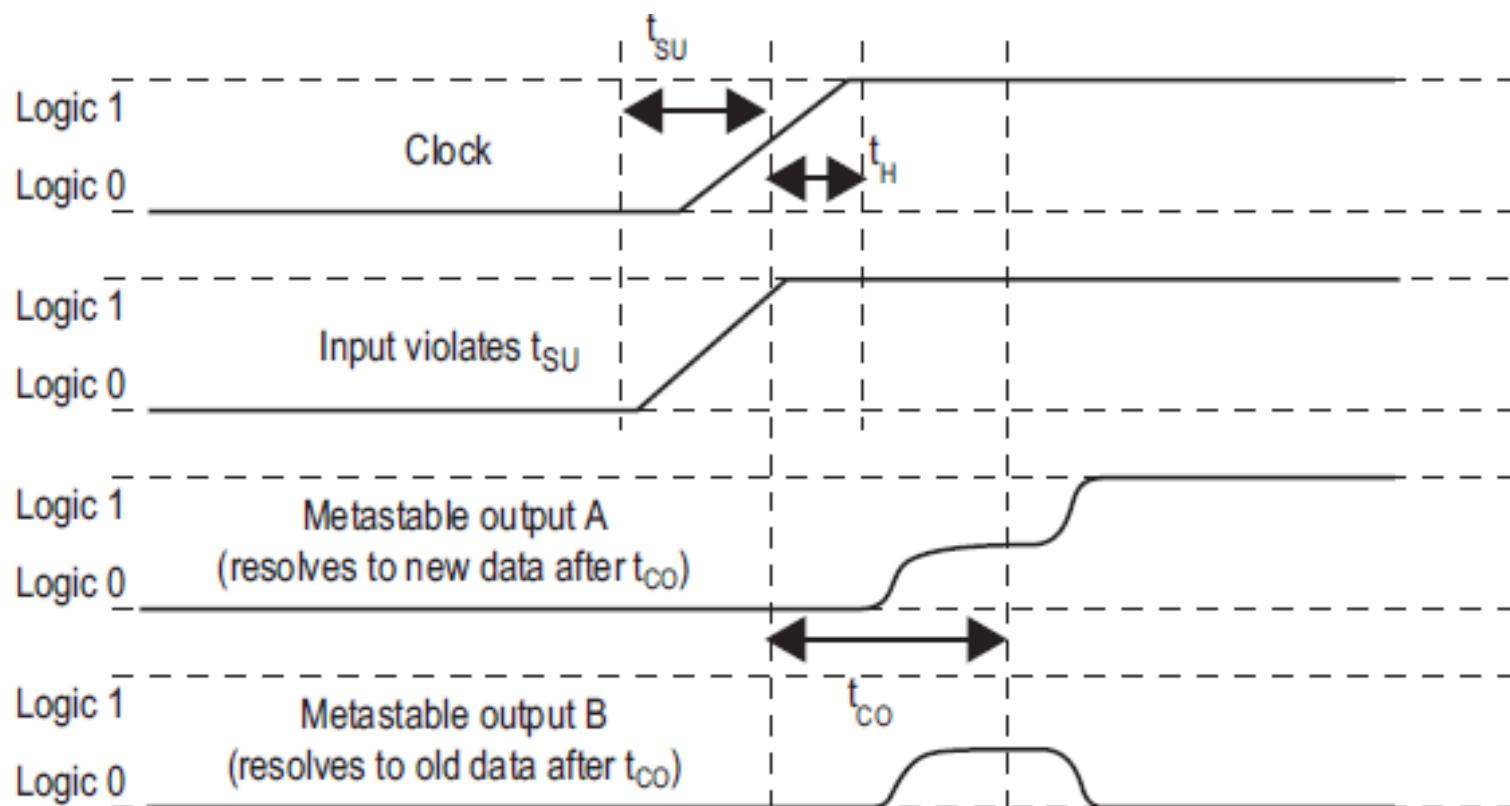


Signal transition occurs after clock edge and minimum  $t_H$ :  
Ball lands on the old data side.

Signal transition meets register  $t_{SU}$  and  $t_H$ :  
Ball lands on the new data side.

Signal violates register  $t_{SU}$  or  $t_H$ :  
Ball balances at top of hill or takes too long to reach the bottom. Output is metastable and violates  $t_{CO}$ .

# Metastability



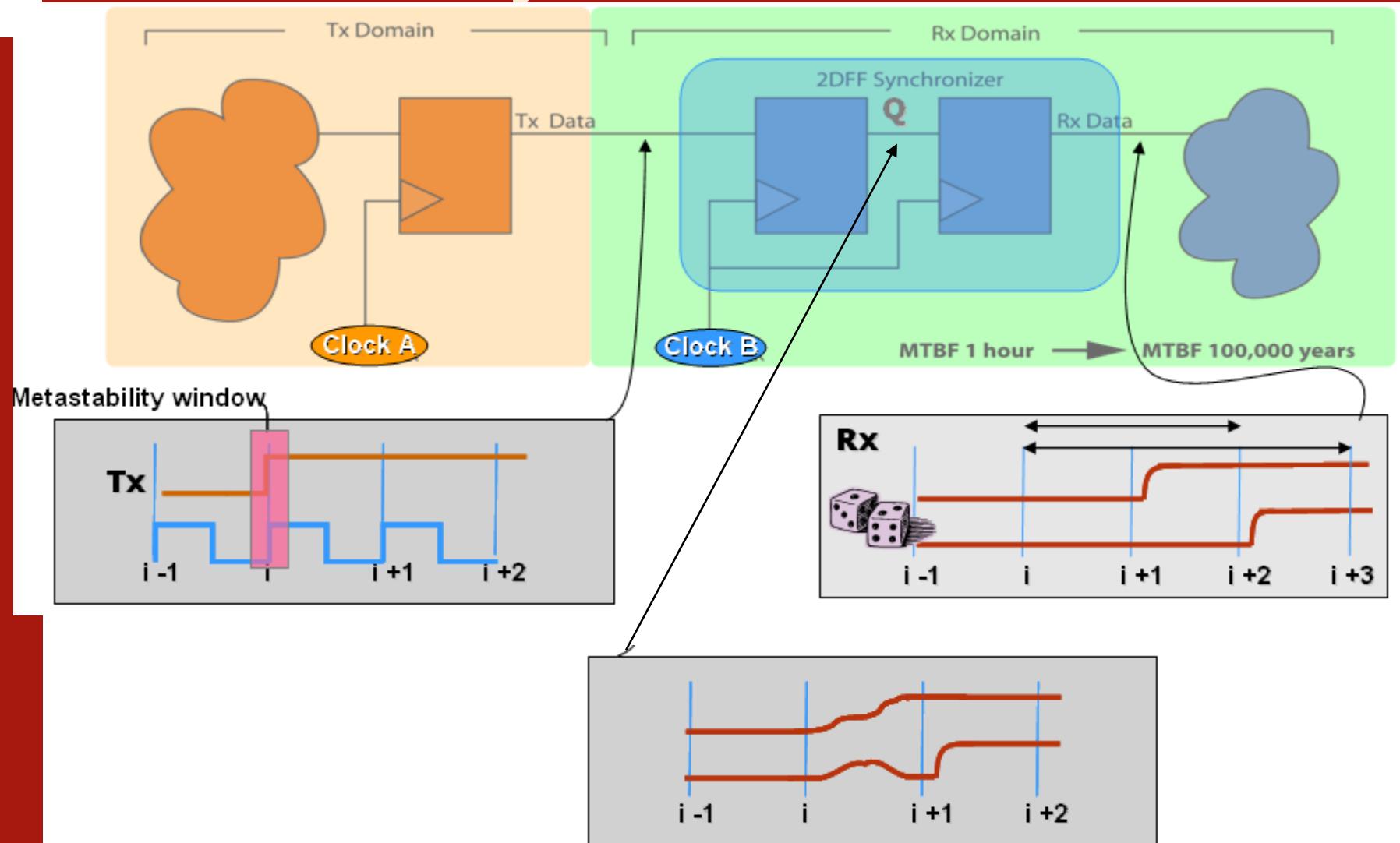
Source: [6]



# Metastability

- One of the metastability solution is the double-registering technique.
- Data coming into the first flip-flop is asynchronous with the clock, so the first flip-flop will almost certainly go metastable.
- However, the second flip-flop will never go metastable as long as the length of metastability is less than the period of the clock. (Unfortunately, FPGA vendors rarely publish metastability times, though they are typically less than the sum of the set-up and hold time of the flip-flop.)

# Metastability



# Metastability

- If the clock is not too fast to meet normal timing constraints, it is probably not going to propagate metastable states in a circuit shown.
- Even though the output of the first flip-flop can be used as long as all of the paths out go to flip-flops clocked by the same clock, it is generally good practice to use a circuit such as shown to isolate metastability to one short line.
- That way, it is less likely that a future change to the circuit will unintentionally use the metastable line in nonclocked logic.



# FPGA clocking resources (Spartan 6)



# Clocking resources

- Each Spartan-6 FPGA device includes 16 high-speed, low-skew global clock resources
- Each Spartan-6 FPGA also provides 40 ultra high-speed, low-skew I/O regional clock resources
- These resources are used automatically by the Xilinx tools

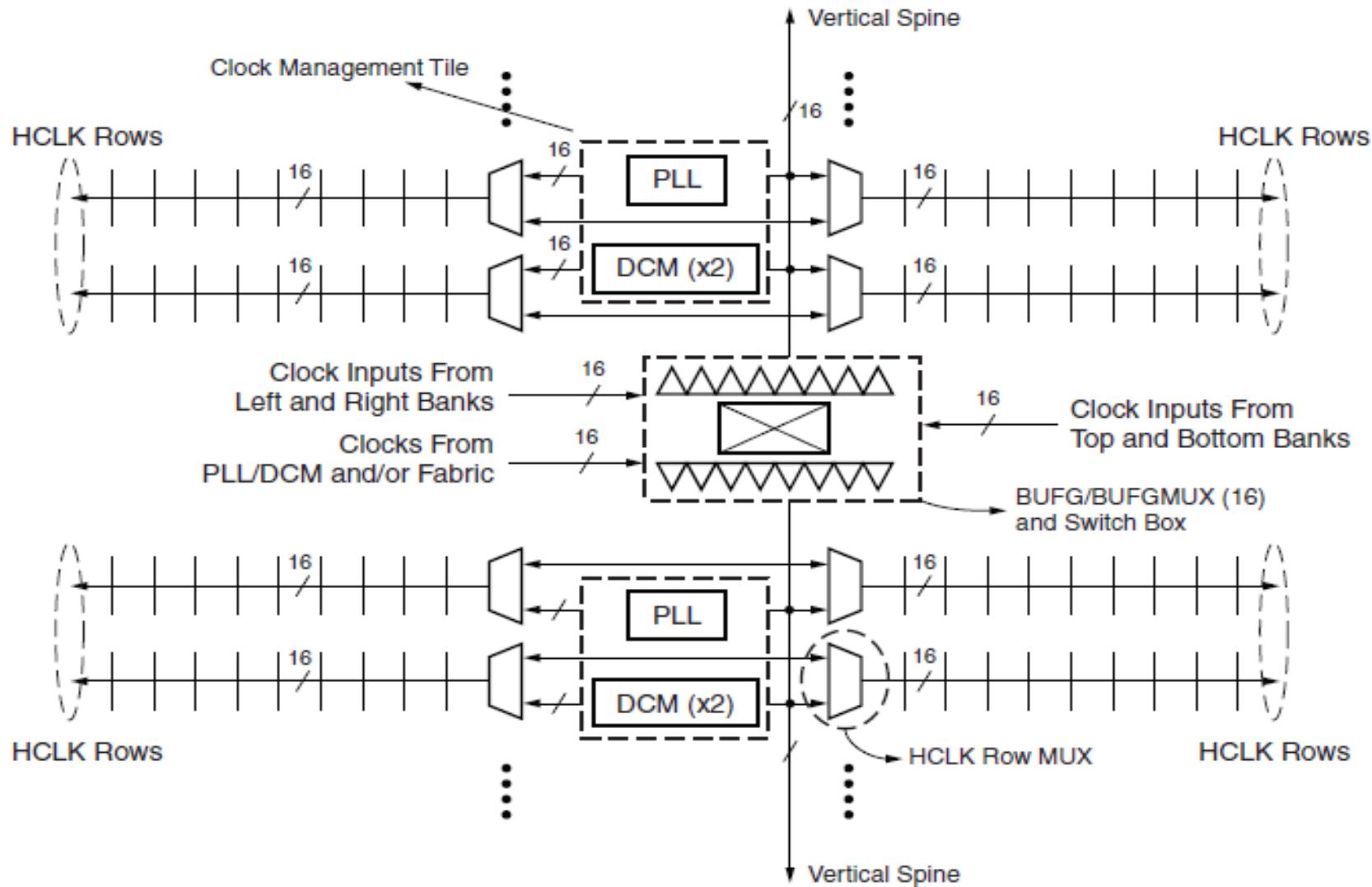
# Clocking resources

- The Spartan-6 FPGA clock resources consist of:
  - Clock networks:
    - Global clock network driven by global clock multiplexers (BUFGMUX):
      - can multiplex between two global clock sources or be used as a simple BUFG clock buffer.
    - I/O regional clock networks driven by I/O clock buffers (BUFIO2) as well as PLL clock buffers (BUFPLL):
      - used to drive clocks routed only on the I/O regional clock network with much higher performance than the global clock network

# Clocking resources

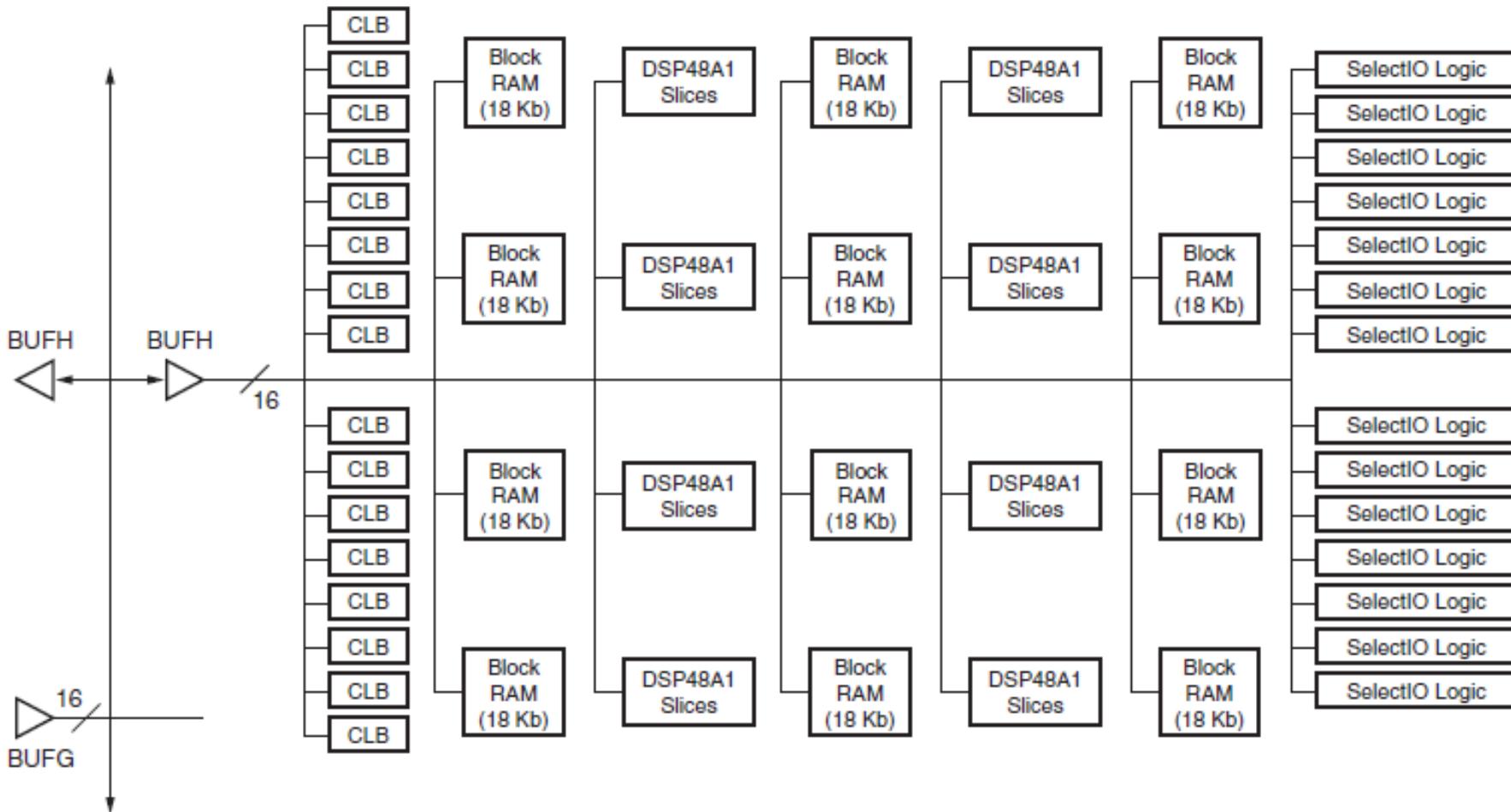
- The Spartan-6 FPGA clock resources consist of:
  - Connections:
    - Global clock input pads (GCLK)
    - Global clock multiplexers (BUFG, BUFGMUX)
    - I/O clock buffers (BUFIO2, BUFIO2\_2CLK, BUFPLL)
    - Clock routing buffers (BUFH)
  - Auxiliary blocks:
    - CMT – Clock Management Tiles
    - DCM – Digital Clock Management
      - DLL – Delay Locked Loop
      - DFS – Digital Frequency Synthesiser
      - DPS – Digital Phase Synthesiser
    - PLL – Phase Locked Loop

# Global Clocking Infrastructure



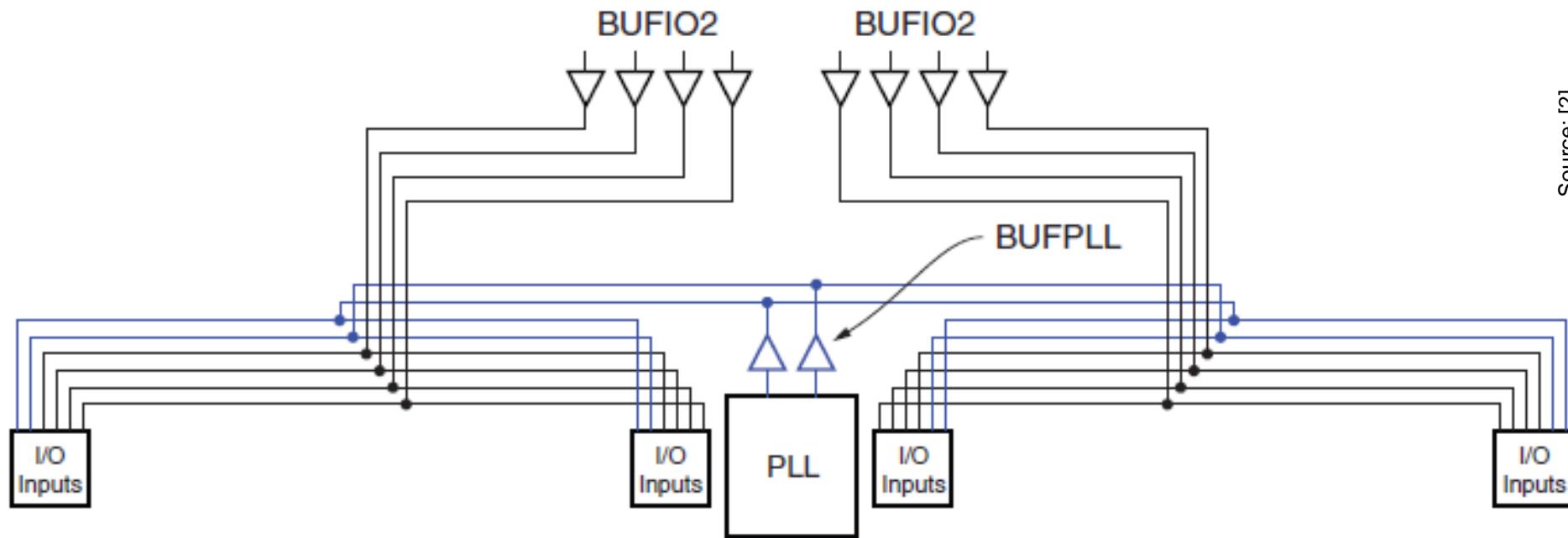
Source: [2]

# Clock routing buffers (BUFH)



Source: [2]

# I/O Clocking infrastructure



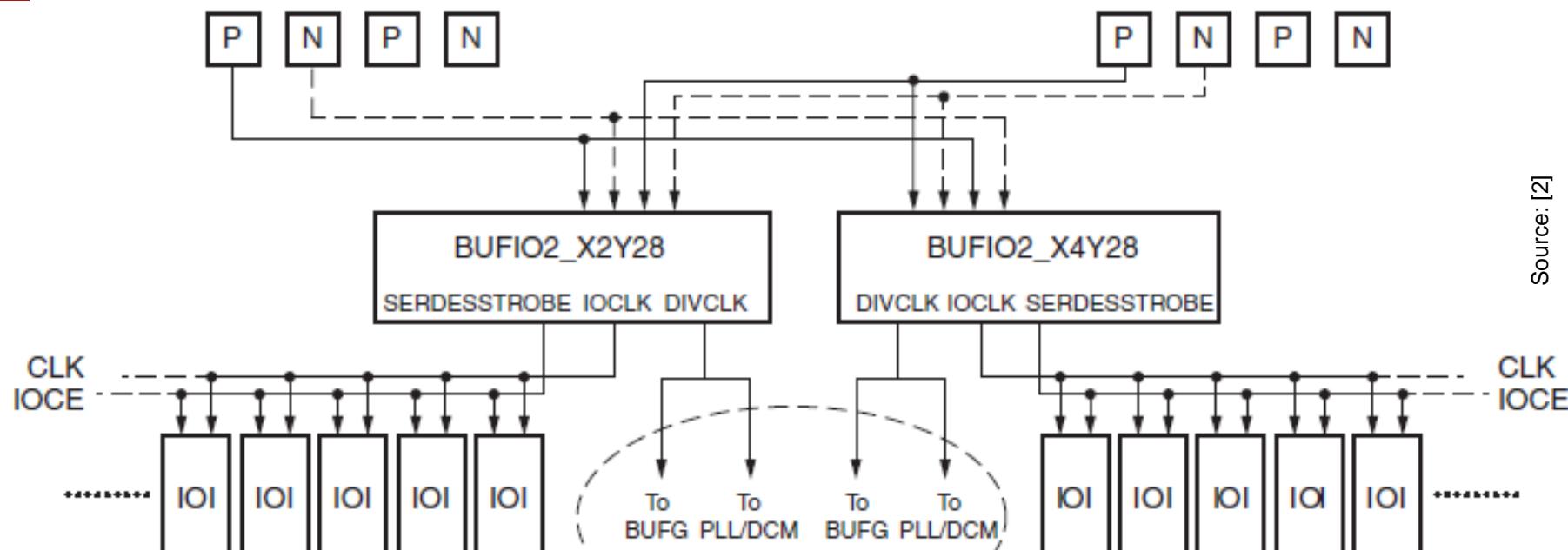
Source: [2]



# Clock inputs (GCLK)

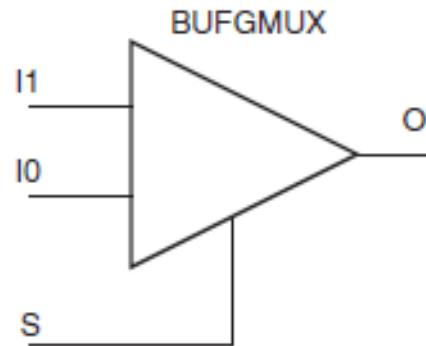
- Each Spartan-6 FPGA has:
  - Up to 32 global clock inputs located along four edges of the FPGA.
  - Eight dedicated clock inputs in the middle of each edge of the device.
  - Eight BUFI02 clocking regions

# Clock inputs (GCLK)



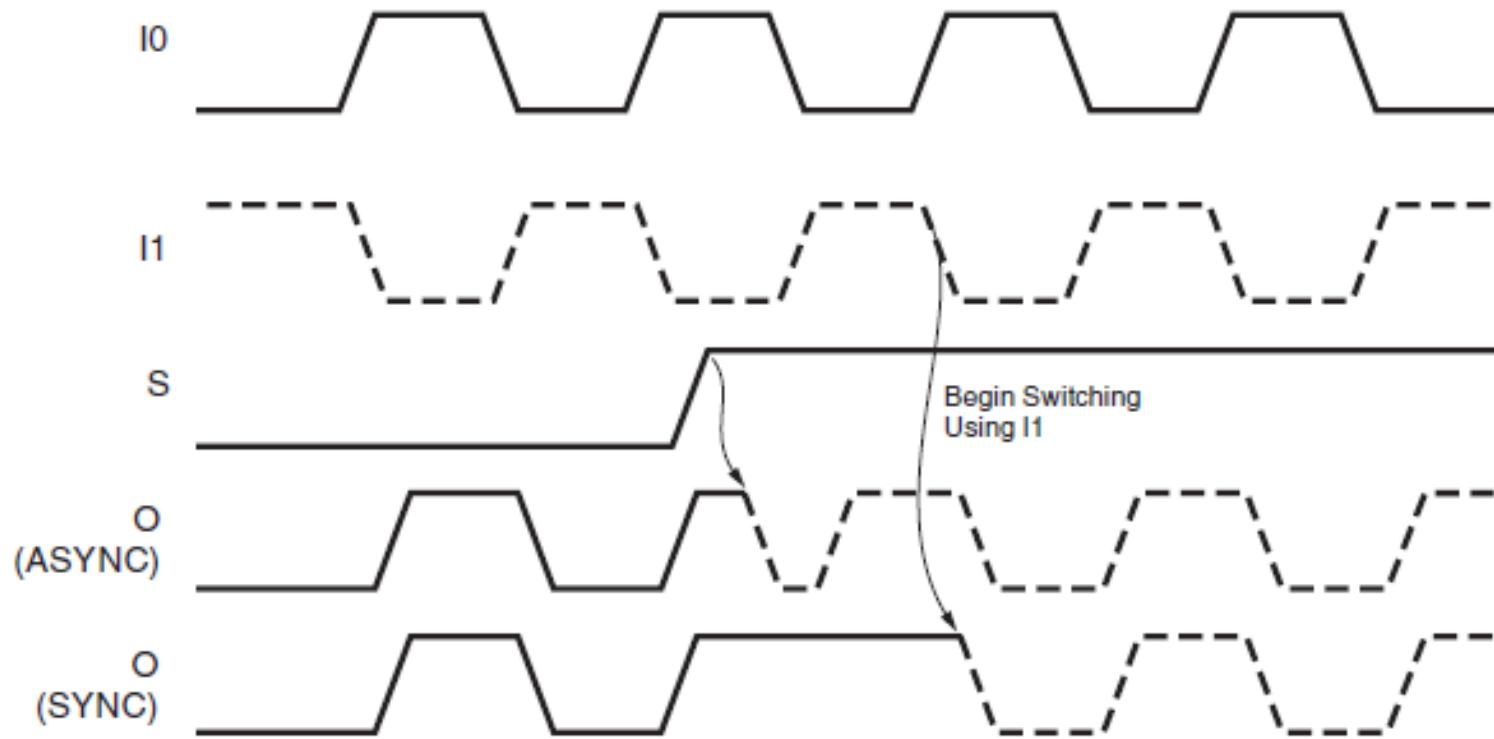
# Global clock multiplexers BUFGMUX

- Each BUFGMUX primitive is a 2-to-1 multiplexer



- The BUFGMUX multiplexes two clock signals while eliminating any timing hazards by switching from one clock source to a completely asynchronous clock source without glitches

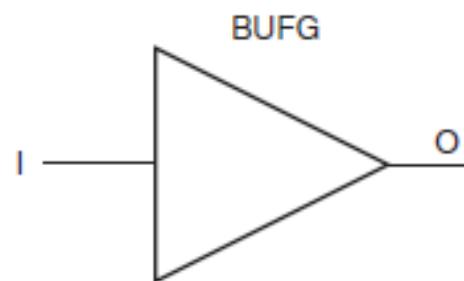
# Global clock multiplexers BUFGMUX



UG382\_c1\_22\_020811

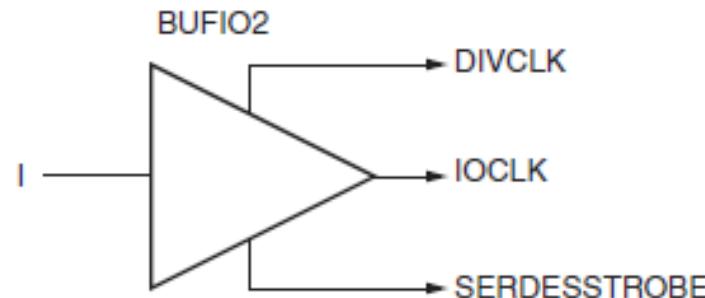
# Global clock multiplexers BUFG

- The BUFG clock buffer primitive drives a single clock signal onto the clock network
- The BUFG is essentially the same as a BUFGMUX, without the clock select mechanism



# I/O clock buffers - BUFI02

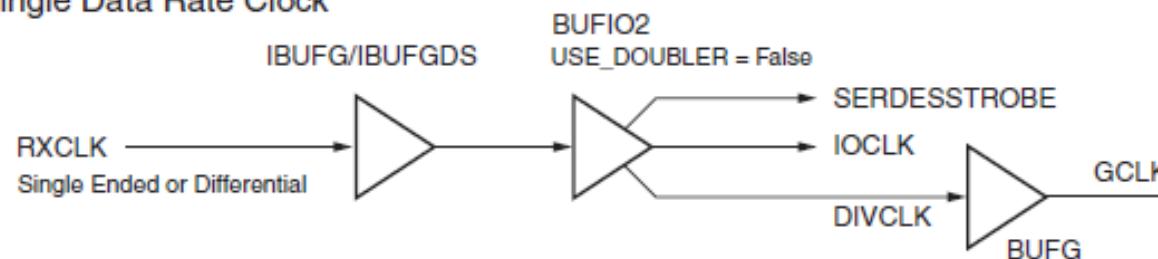
- The BUFI02 takes a GCLK clock input and generates two clock outputs and a strobe pulse



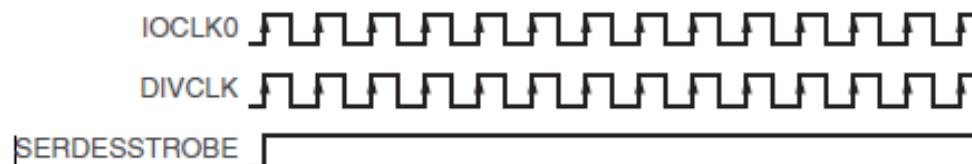
- IOCLK – normal output
- DIVCLK – divided output (by 1,2,3,4,5,6,7,8)
- SERDESSTROBE - clock network output used to drive IOSERDES2

# I/O clock buffers - BUFI02

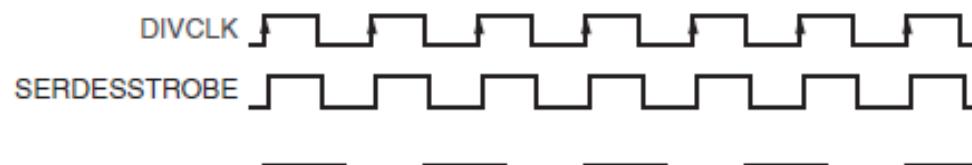
Single Data Rate Clock



DIVIDE = 1



DIVIDE = 2



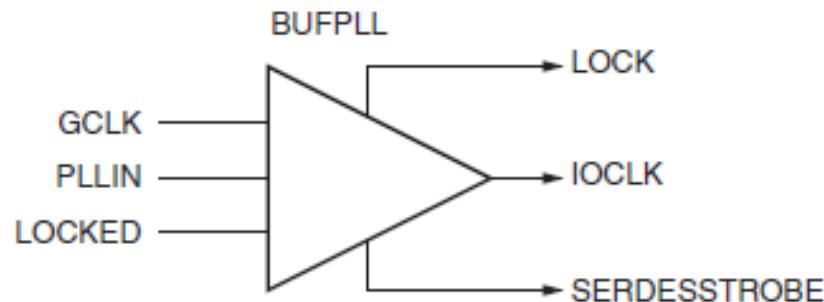
DIVIDE = 3



DIVIDE = 4

# I/O clock buffers - BUFPLL

- The BUFPLL is intended for high-speed I/O routing to generate clocks and strobe pulses for the SERDES primitives



- The BUFPLL aligns the SERDESSTROBE to the IOCLK
- SERDESSTROBE is a divided PLLIN signal



# Available CMT, DCM, and PLL Resources

Device	Number of CMTs	Number of DCMs	Number of PLLs
XC6SLX4	2	4	2
XC6SLX9	2	4	2
XC6SLX16	2	4	2
XC6SLX25	2	4	2
XC6SLX25T	2	4	2
XC6SLX45	4	8	4
XC6SLX45T	4	8	4
XC6SLX75	6	12	6
XC6SLX75T	6	12	6
XC6SLX100	6	12	6
XC6SLX100T	6	12	6
XC6SLX150	6	12	6
XC6SLX150T	6	12	6



**DCM**



# DCM Summary

- Digital Clock Managers (DCMs) provide advanced clocking capabilities to Spartan FPGA applications
- DCMs integrate advanced clocking capabilities directly into the global clock distribution network.



# DCM Summary

- DCM solves a variety of common clocking issues, especially in high-performance, high-frequency applications:
  - Eliminates clock skew, either within the device or to external components, to improve overall system performance and to eliminate clock distribution delays.
  - Phase shifts a clock signal, either by a fixed fraction of a clock period or by incremental amounts.



# DCM Summary

- Multiplies or divides an incoming clock frequency or synthesizes a completely new frequency by a mixture of static or dynamic clock multiplication and division
- Conditions a clock, ensuring a clean output clock with a 50% duty cycle.
- Filters clock input jitter



# DCM Summary

- Mirrors, forwards or rebuffers a clock signal, often to deskew and converts the incoming clock signal to a different I/O standard. For example, forwarding and converting an incoming LVTTL clock to LVDS.
- Free-running oscillator
- Spread-spectrum clock generation



# DCM Features

Feature	Description	DCM Signals
DCMs per device	Four to 12 DCMs, depending on device size. See <a href="#">Table 2-1</a> .	All
Clock input sources	GCLK input, BUFG output, cascaded DCM or PLL output (within the same CMT)	CLKIN
Frequency synthesizer output	Multiply CLKIN by the fraction (M/D) where M = {2..256}, D = {1..256} when using the DCM_CLKGEN primitive	CLKFX, CLKFX180
Clock divider output	Divide CLKIN by 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, or 16	CLKDV
Clock doubler output	Multiply CLKIN frequency by 2	CLK2X, CLK2X180



# DCM Features

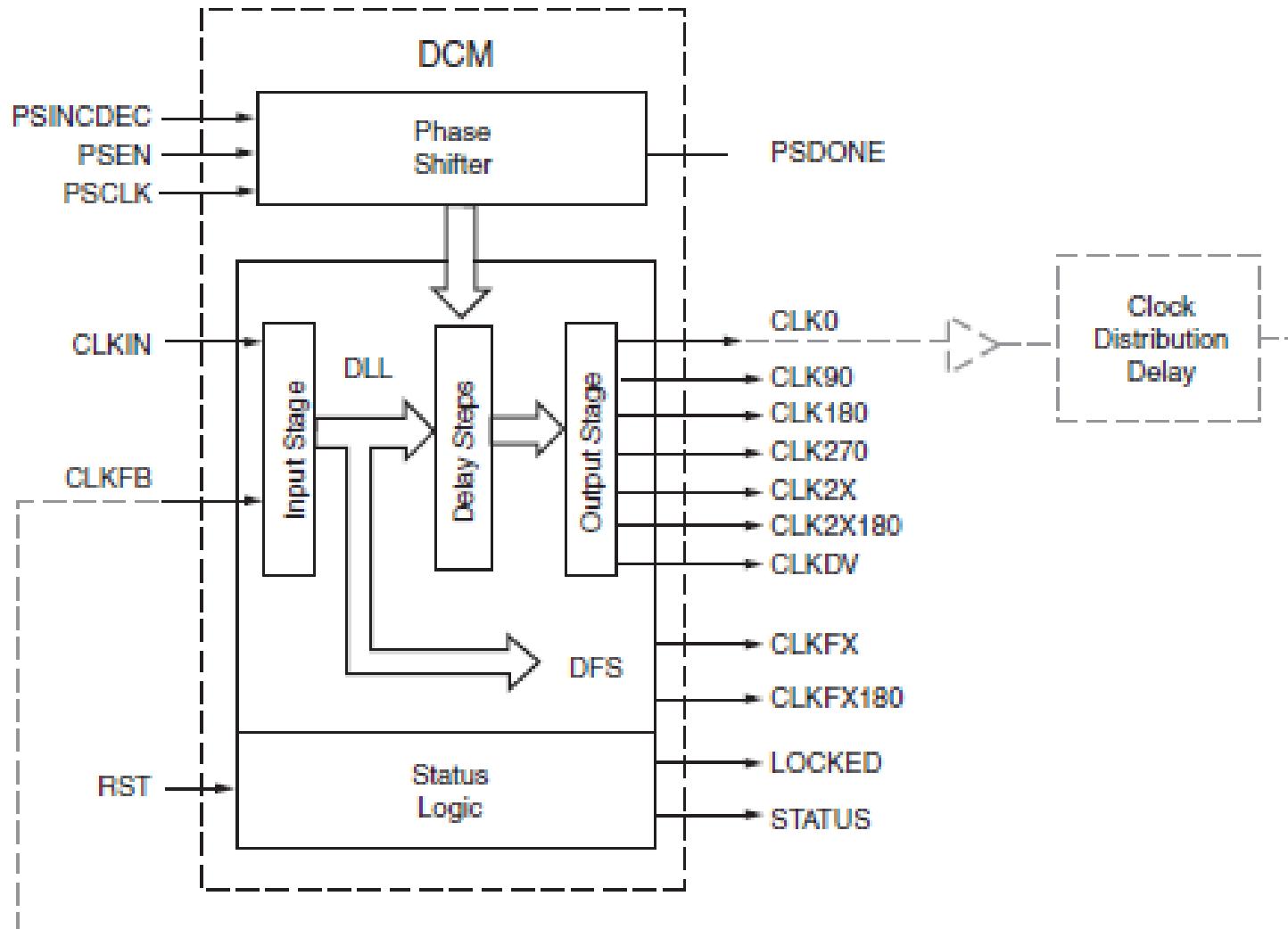
Clock conditioning, duty-cycle correction	Always provided on most outputs.	All
Quadrant phase-shift outputs	$0^\circ$ (no phase shift), $90^\circ$ ( $\frac{1}{4}$ period), $180^\circ$ ( $\frac{1}{2}$ period), $270^\circ$ ( $\frac{3}{4}$ period)	CLK0, CLK90, CLK180, CLK270
Half-period phase-shift outputs	Output pairs with $0^\circ$ and $180^\circ$ phase shift, ideal for DDR applications	CLK0, CLK180, CLK2X, CLK2X180, CLKFX, CLKFX180
Variable phase-shifting	Allows DCM clock outputs to adjust phase shift during operation	PSEN, PSINCDEC, PSCLK, PSDONE
General purpose DCM operation indicators	Number of DCM clock outputs connected to general-purpose interconnect	STATUS, LOCKED



# DCM in Xilinx families

Function	Virtex-5 FPGAs <sup>(1)</sup>	Spartan-3E and Extended Spartan-3A FPGAs	Spartan-6 FPGAs
Design primitive	DCM_BASE DCM_ADV	DCM_SP	DCM_SP DCM_CLKGEN
Distinct DLL operating frequency ranges	Two: Low and High	One	One
Distinct DFS operating frequency ranges	Two: Low and High	One	One
Variable phase-shift increment or decrement unit	$\frac{1}{256}$ th of CLKIN period (degrees)	DCM_DELAY_STEP between 15 to 35 ps (time)	DCM_DELAY_STEP See the Spartan-6 FPGA data sheet
DCM V <sub>CCAUX</sub> voltage supply	2.5V	2.5V or 3.3V	2.5V or 3.3V
Jitter reduction with expense of phase alignment	No	No	Yes
Dynamic programming of frequency multiplication and division	No	No	Yes
Generation of spread-spectrum clocks	No	No	Yes

# DCM Block Diagram



Source: [1]

# Delay-Locked Loop

- The Delay-Locked Loop (DLL) provides an on-chip digital deskew circuit that effectively generates clock output signals with a net zero delay.
- The deskew circuit compensates for the delay on the clock routing network by monitoring an output clock, from either the CLK0 or the CLK2X outputs



# Delay-Locked Loop

- The DLL effectively eliminates delay from the external clock input port to the individual clock loads within the device.
- The well-buffered global network minimizes the clock skew on the network caused by loading differences.

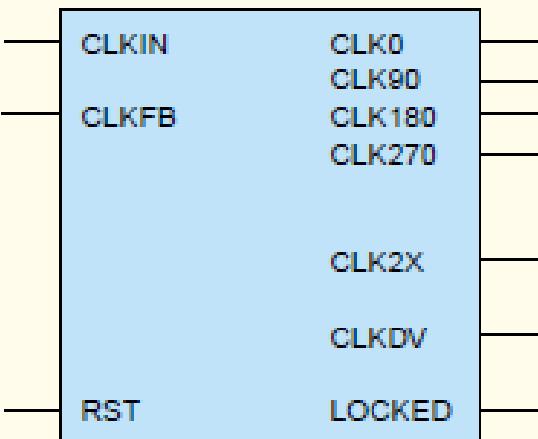


# Delay-Locked Loop - functioning

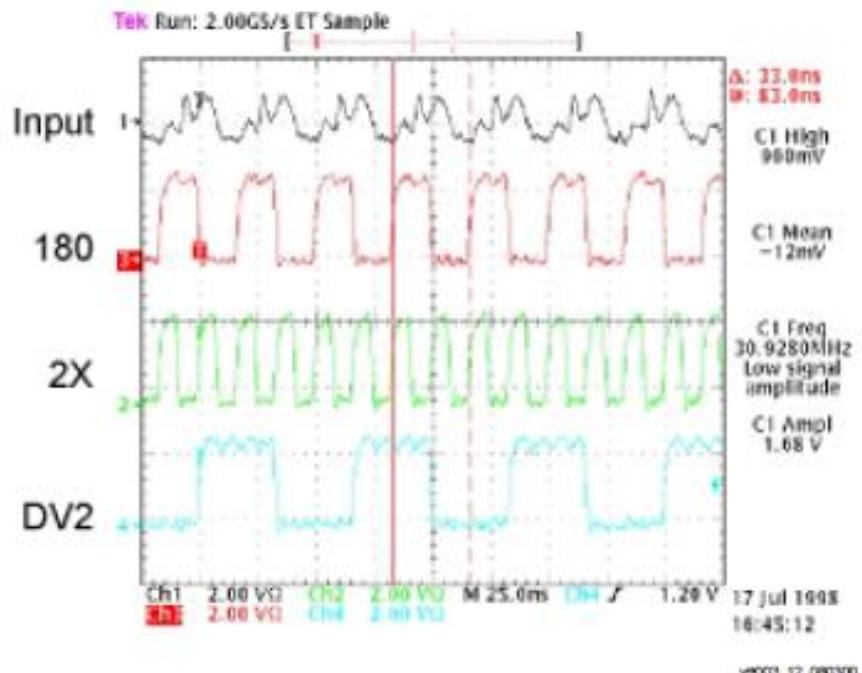
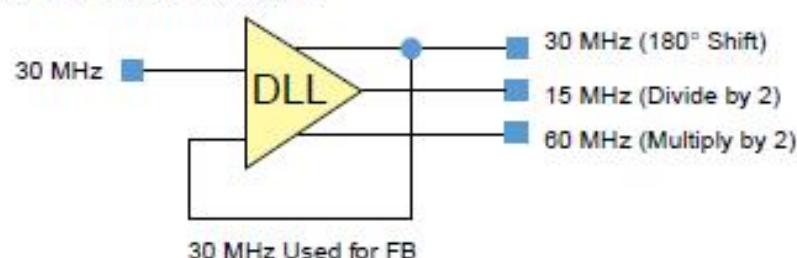
- A DLL in its simplest form inserts a variable delay line between the external clock and the internal clock
- The clock tree distributes the clock to all registers and then back to the feedback pin of the DLL
- The control circuit of the DLL adjusts the delays so that the rising edges of the feedback clock align with the input clock

# DLL Block Diagram - functioning

Delay-Locked Loop



30 MHz - 180° Phase Shift





# Digital Frequency Synthesizer

- The Digital Frequency Synthesizer (DFS) provides a wide and flexible range of output frequencies based on the ratio of two user-defined integers, a multiplier (CLKFX\_MULTIPLY) and a divisor (CLKFX\_DIVIDE).
- The output frequency is derived from the input clock (CLKIN) by simultaneous frequency division and multiplication.
- The DFS feature can be used in conjunction with or separately from the DLL feature of the DCM

# Phase Shift

- The Phase Shift (PS) controls the phase relations of the DCM clock outputs to the CLKIN input.
- PS can be used in either fixed phase-shift or variable phase shift modes
- The phase of all nine DCM clock output signals are shifted by a fixed fraction of the input clock period when using the fixed phase shift.
- The PS also provides a digital interface for the FPGA application to dynamically advance or retard the current shift value, called variable phase shift.



# Status Logic

- The status logic indicates the current state of the DCM via the LOCKED and STATUS[0] output signals.
- The LOCKED output signal indicates whether the DCM outputs are in phase with the CLKIN input.
- The STATUS output signals indicate the state of the DLL and PS operations.



# DCM config

Xilinx Clocking Wizard - General Setup

CLKIN CLKFB DCM CLK0 CLK90 CLK180 CLK270 CLKDV CLK2X CLK2X180 CLKFX CLKFX180 RST PSEN PSINCDEC PSCLK STATUS LOCKED PSDONE

Input Clock Frequency: 100 MHz ns

Phase Shift: Type: NONE Value: 0

CLKIN Source: External Single

Feedback Source: External Single

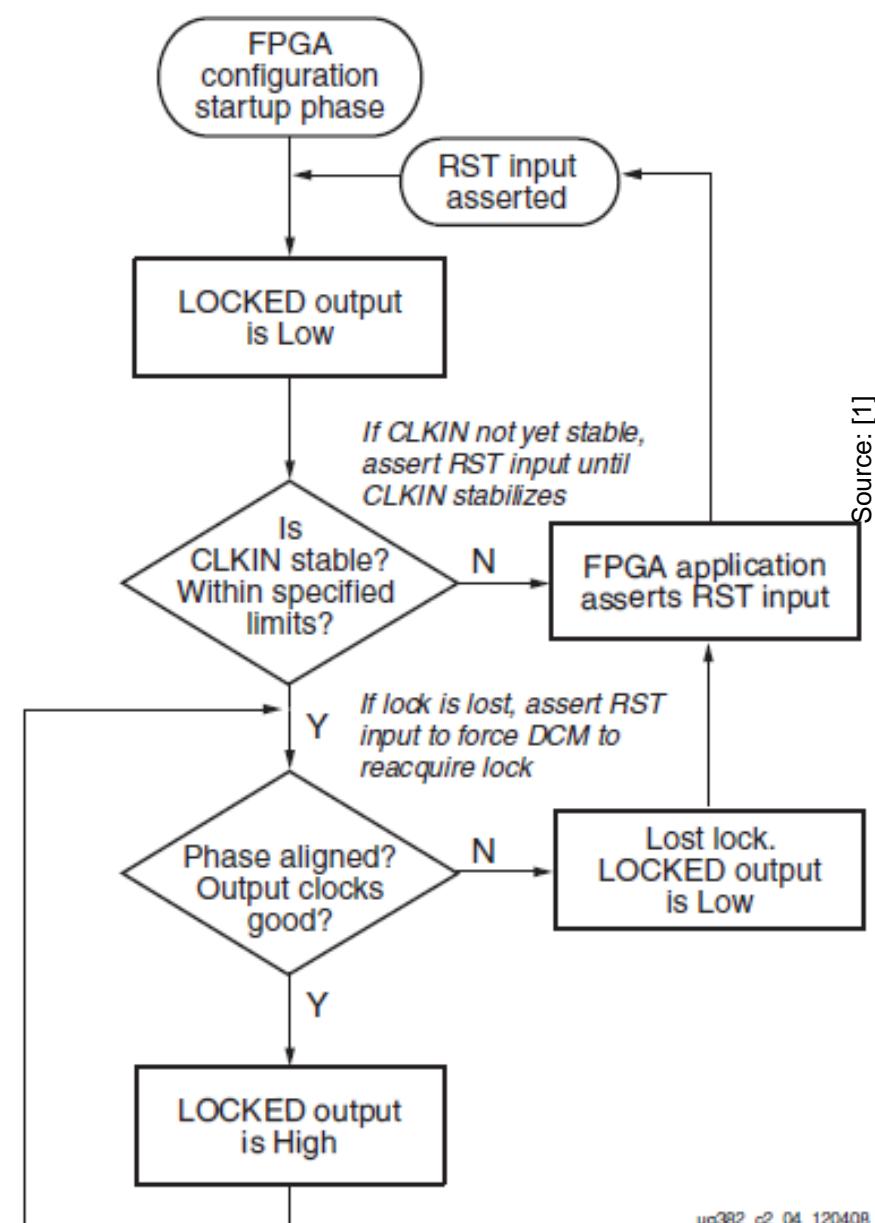
Divide By Value: 2

Feedback Value: 1X

Use Duty Cycle Correction:

More Info Advanced < Back Next > Cancel

# LOCKED output behaviour



Source: [1]



# Spread Spectrum Generation

- Spread-spectrum clock generation (SSCG) is widely used by manufacturers of electronic devices to reduce the spectral density of the electromagnetic interference (EMI) generated by these devices.
- Typical (but expensive!) solutions for meeting EMC requirements involve adding expensive shielding, ferrite beads, or chokes



# Spread Spectrum Generation

- SSCG spreads the electromagnetic energy over a large frequency band to effectively reduce the electrical and magnetic field strengths measured within a narrow window of frequencies.
- The peak electromagnetic energy at any one frequency is reduced by modulating the SSCG output.

# Spread Spectrum Generation

- The SSCG commonly defines the following parameters:
  - Frequency deviation, that is the percentage of the input frequency
  - Modulation frequency
  - Spread type: up/down/center
  - Modulation profile, for example, the shape of the triangle
- Typically: 75 MHz (input frequency),  $\pm 2.0\%$  center spread, and 75 KHz triangular modulation.



# Spread Spectrum Generation

Spread Spectrum Values	Fixed Spread-Spectrum Modes	Soft Spread-Spectrum Modes
SPREAD_SPECTRUM values	CENTER_LOW_SPREAD CENTER_HIGH_SPREAD	VIDEO_LINK_M0, VIDEO_LINK_M1, VIDEO_LINK_M2
Additional logic	None	SOFT_SS
Modulation profile	Triangular	Triangular
Spread direction	Center	Down
Spread range	Fixed	See Figure
$F_{MOD}$	$F_{IN}/1024$	See Figure
CLKFX_MULTIPLY	2–32	7
CLKFX_DIVIDE	1–4	2, 4
DCM_CLKGEN programming ports	N/A	PROGCLK, PROGEN, PROGDATA, PROGDONE



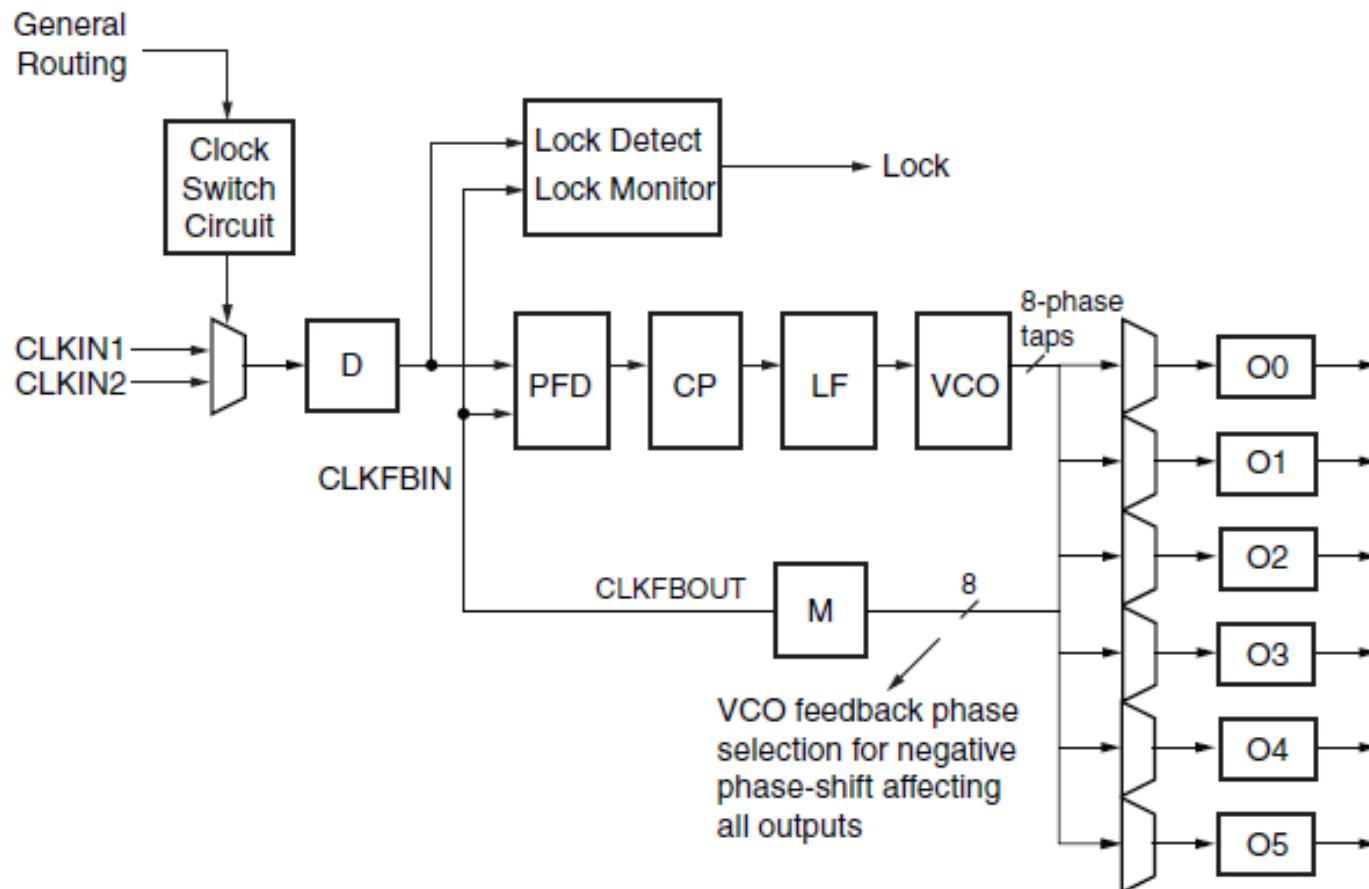
PLL



# PLL Introduction

- The main purpose of PLLs is:
  - to serve as a frequency synthesizer for a wide range of frequencies
  - to serve as a jitter filter for either external or internal clocks in conjunction with the DCMs of the CMT (Clock Management Tile).

# PLL Block Diagram



Source: [1]



# PLL Block Diagram

- D is a programmable counter for each clock input
- Phase-Frequency Detector (PFD) compares both phase and frequency of the input (reference) clock and the feedback clock
  - The PFD is used to generate a signal proportional to the phase and frequency between the two clocks.
- Charge Pump (CP) and Loop Filter (LF) are used to generate a reference voltage to the Voltage Controlled Oscillator (VCO).

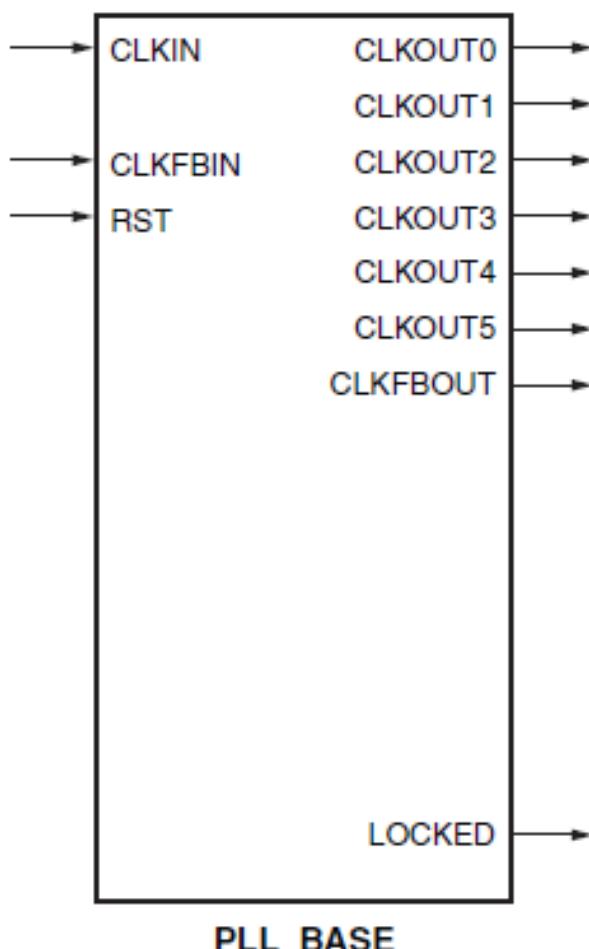
# PLL Block Diagram

- The PFD produces an up or down signal to the charge pump and loop filter to determine whether the VCO should operate at a higher or lower frequency
- When VCO operates at too high of a frequency, the PFD activates a down signal, causing the control voltage to be reduced decreasing the VCO operating frequency
- When the VCO operates at too low of a frequency, an up signal will increase voltage

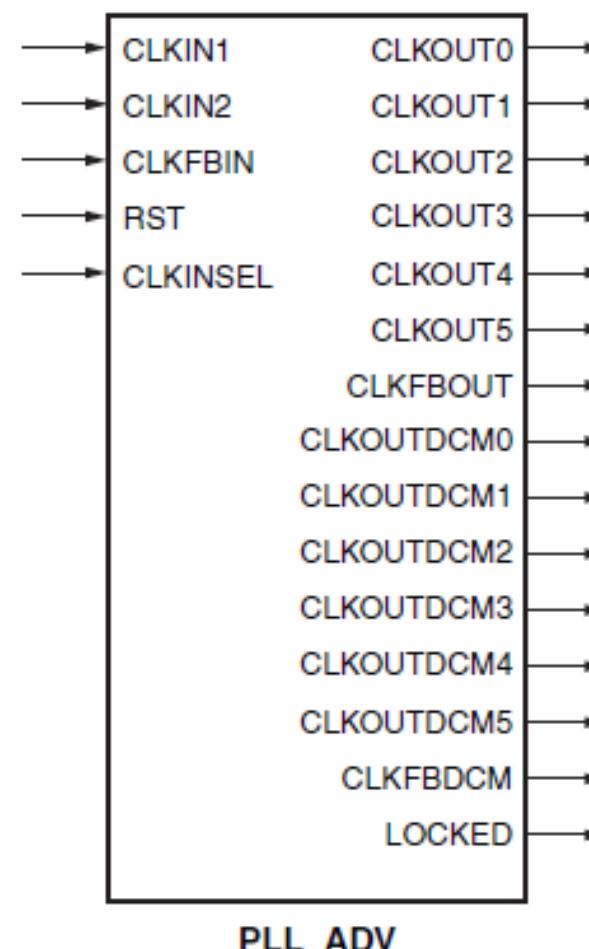
# PLL Block Diagram

- The VCO produces eight output phases.
- Each output phase can be selected as the reference clock to the output counters
- The counter M controls the feedback clock of the PLL allowing a wide range of frequency synthesis

# PLL Primitives



**PLL\_BASE**



**PLL\_ADV**

Source: [1]



# Timing Constraints



# About Timing Constraints

- Timing constraints provide a basis for the design of timing goals
- Creating global constraints for a design is the easiest way to provide coverage of constrainable connections in a design
- Global constraints constrain the entire design



# Specifying Timing Constraints

- To specify timing constraints before synthesis:
  - Specify the timing constraints into your design:
    - HDL
      - VHDL
      - Verilog
    - Schematic
- OR
  - Specify the timing constraints in an XCF (XST Constraints File).



# XST Timing Constraints

- Asynchronous Register (ASYNC\_REG)
- Clock Signal (CLOCK\_SIGNAL)
- Multi-Cycle Path
- Maximum Delay (MAXDELAY)
- Maximum Skew (MAXSKEW)
- Offset (OFFSET)
- Period (PERIOD)
- System Jitter (SYSTEM\_JITTER)
- Timing Ignore (TIG)
- Time Group (TIMEGRP)
- Timing Specifications (TIMESPEC)
- Timing Name (TNM)
- Timing Name Net (TNM\_NET)



# XST Timing Constraints

- **Asynchronous Register (ASYNC\_REG):**
  - can be attached only on registers or latches with asynchronous input (D input or the CE input)
- **Clock Signal (CLOCK\_SIGNAL)**
  - If a clock signal goes through combinatorial logic before being connected to the clock input of a flip-flop, XST cannot identify which input pin or internal signal is the real clock signal.
  - Clock Signal (CLOCK\_SIGNAL) allows to define



# XST Timing Constraints

- Multi-Cycle Path (FROM:TO):
  - The Multi-Cycle Path constraint specifies a timing constraint between two groups
- Maximum Delay (MAXDELAY)
  - defines the maximum allowable delay on a net
- Maximum Skew (MAXSKEW)
  - controls the amount of skew on a net



# XST Timing Constraints

- Offset (OFFSET)
  - specifies the timing relationship between an external clock and its associated data-in or data-out pin.
  - OFFSET is used only for pad related signals, and cannot be used to extend the arrival time specification method to the internal signals in a design



# XST Timing Constraints

- Period (PERIOD)

- Checks timing between all synchronous elements within the clock domain as defined in the destination element group
- The period specification is attached to the clock net
- Checks for hold violations



# XST Timing Constraints

- System Jitter (SYSTEM\_JITTER)
  - Specifies the system jitter of the design.
  - System Jitter depends on various design conditions, such as the number of flip-flops changing at one time and the number of I/Os changing.
  - System Jitter applies to all clocks within a design
  - Used within the clock uncertainty calculation for all constraints that analyze a clock in the design



# XST Timing Constraints

- System Jitter (SYSTEM\_JITTER) VHDL example

```
entity top_yann_mem is
port ( cntrl0_DDR2_DQ : inout
      std_logic_vector(71 downto 0);
      SYS_CLK_P : in std_logic;
      SYS_CLK_N : in std_logic;
      CLK200_P : in std_logic;
      CLK200_N : in std_logic
);
attribute SYSTEM_JITTER : string;
attribute SYSTEM_JITTER of top_yann_mem:
entity is "10 ps";
```



# XST Timing Constraints

- **Timing Ignore (TIG)**
  - Causes paths that fan forward from the point of application (of TIG) to be treated as if they do not exist (for the purposes of the timing model) during implementation
- **Time Group (TIMEGRP)**
  - basic grouping constraint



# XST Timing Constraints

- **Timing Specifications (TIMESPEC)**
  - Serves as a placeholder for timing specifications, which are called TS attribute definitions
  - A TS attribute defines the allowable delay for paths in the design
- **Timing Name (TNM)**
  - Use TNM to identify the elements that make up a group which can then be used in a timing specification



# XST Timing Constraints

- Timing Name Net (TNM\_NET)
  - identifies the elements that make up a group, which can then be used in a timing specification.
  - TNM\_NET is essentially equivalent to TNM on a net except for input pad nets.

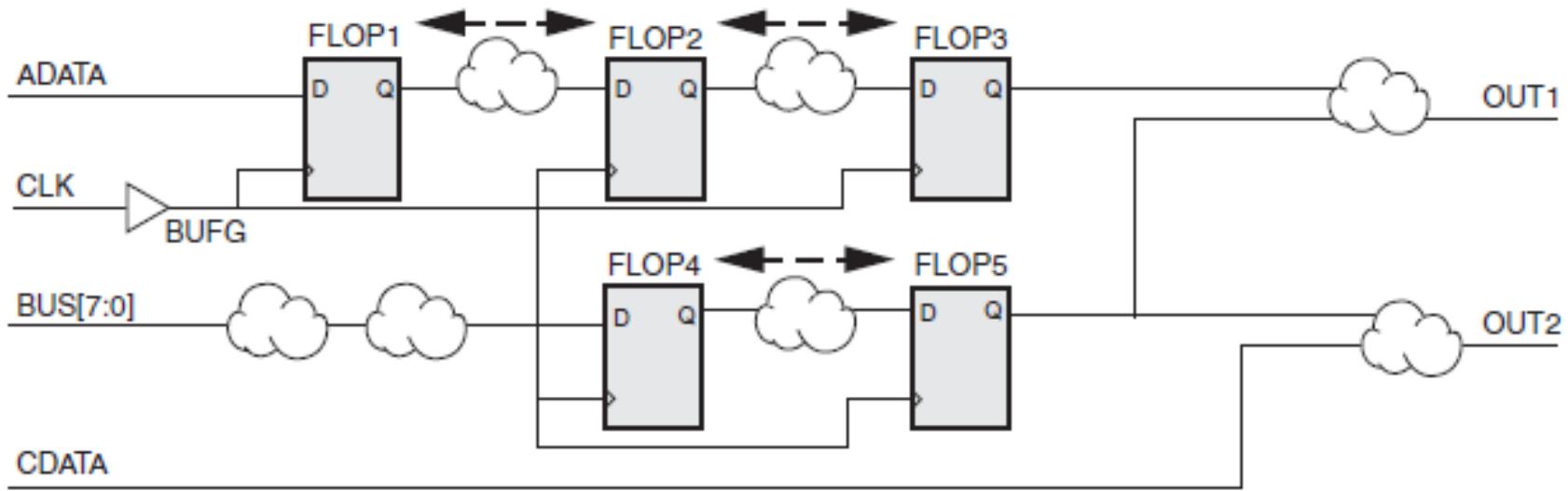


# PERIOD Constraint

- The PERIOD constraint is a fundamental timing and synthesis constraint.
- PERIOD:
  - Defines each clock within the design
  - Covers all synchronous paths within each clock domain
  - Cross checks clock domain paths between related clock domains
  - Defines the duration of the clock
  - Can be configured to have different duty cycles

# PERIOD Constraint

- The PERIOD defines the timing between synchronous elements (FFS, RAMS, LATCHES, HSIOs, CPUs and DSPS) clocked by a specific clock net that is terminated at a registered clock pin





# PERIOD Constraint

- The PERIOD constraint on a clock net analyzes all delays on all paths that terminate at a pin with a setup and hold analysis relative to the clock net.



# PERIOD Constraint

- The PERIOD constraint includes analysis of:
  - Clock path delay in the clock skew analysis for global and local clocks
  - Local clock inversion
  - Setup and hold time analysis
  - Phase relationship between related clocks
  - DCM Jitter, Duty-Cycle Distortion User-Defined System and Clock Input Jitter as Clock Uncertainty
  - Unequal clock duty cycles (non 50%)
  - Clock phase including DCM phase and negative edge clocking

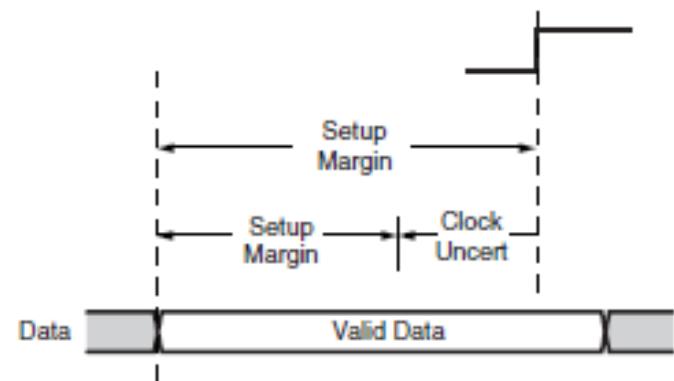
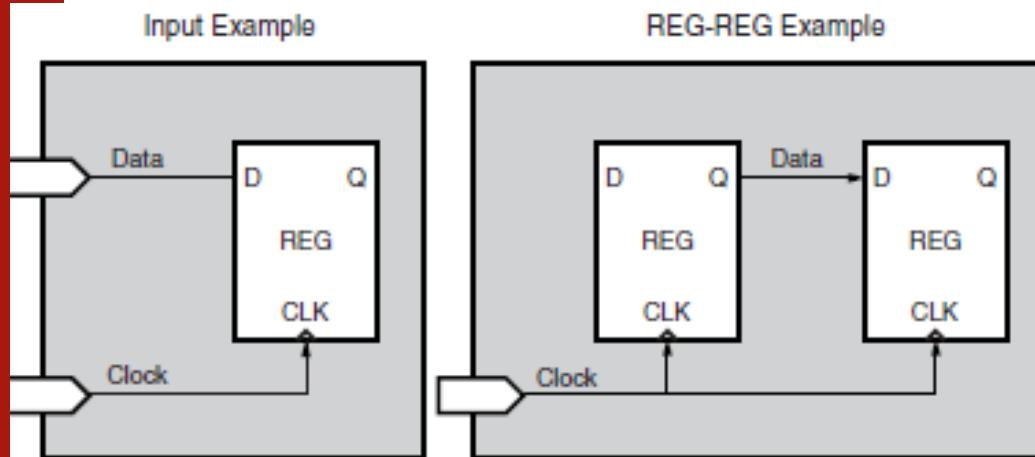


# PERIOD Constraint

- The PERIOD constraint covers paths only between synchronous elements
- Pads are not included in the analysis
- Analysis between unrelated or asynchronous clock domains is also not included
- The PERIOD constraint analysis includes the setup and hold analysis on synchronous elements

# PERIOD Constraint - Setup time

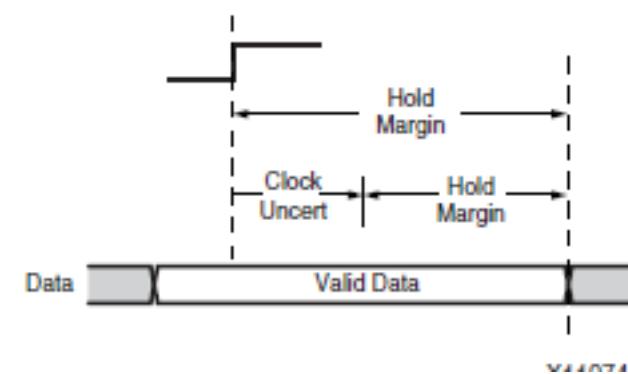
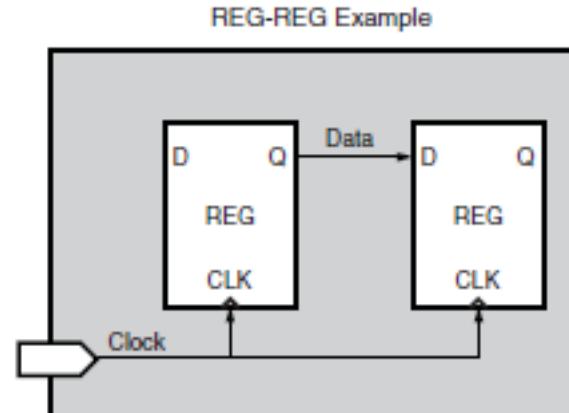
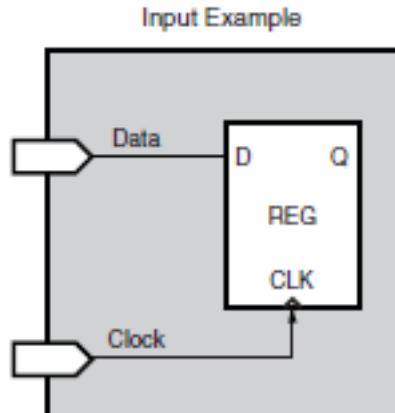
- Setup Time = Data Path Delay + Synchronous Element Setup Time – Clock Path Skew
- Slack = Requirement – (Data Path – Clock Path Skew + Clock Uncertainty)



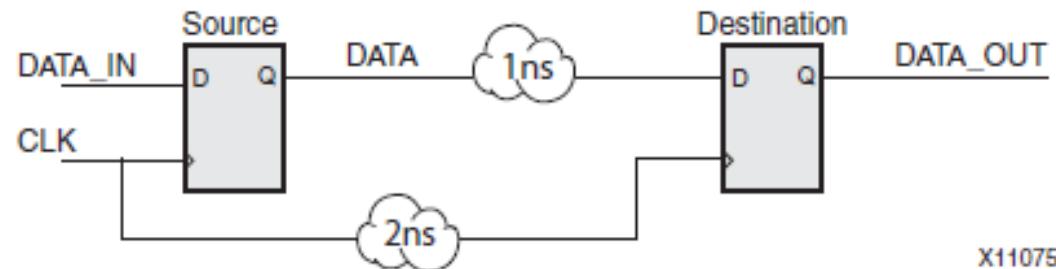
# PERIOD Constraint - Hold time

A hold time violation occurs when the positive clock skew is greater than the data path delay

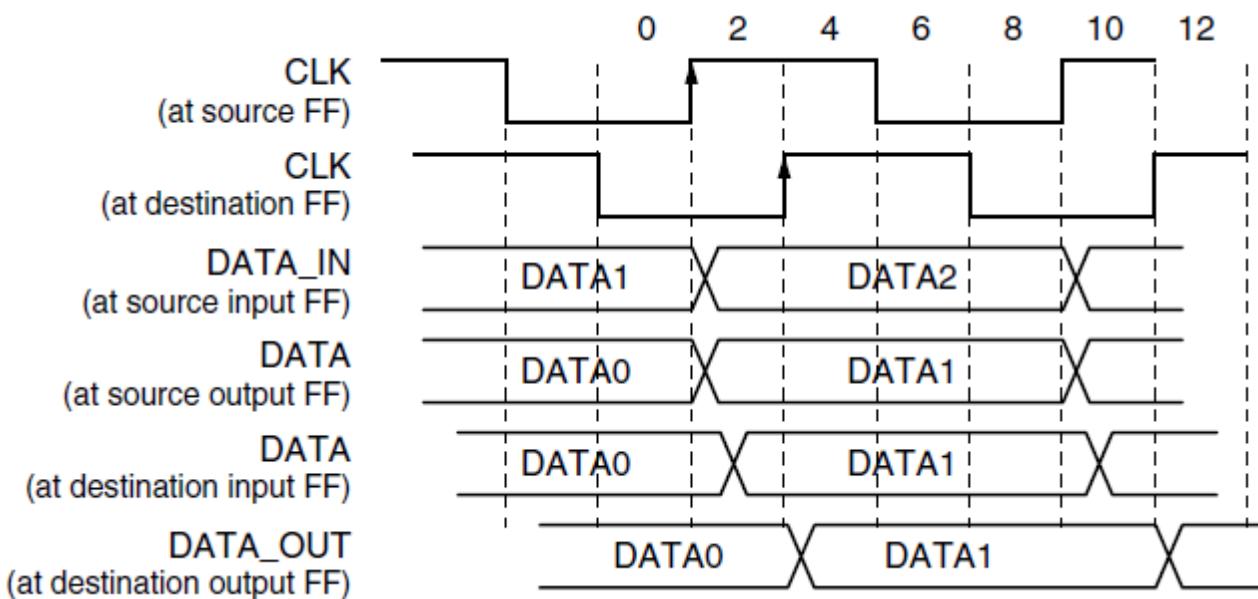
- Hold Time = Clock Path Skew + Synchronous Element Hold Time - Data Path Delay
- Slack = Requirement - (Clock Path Skew +



# PERIOD Constraint - Hold violation



X11075



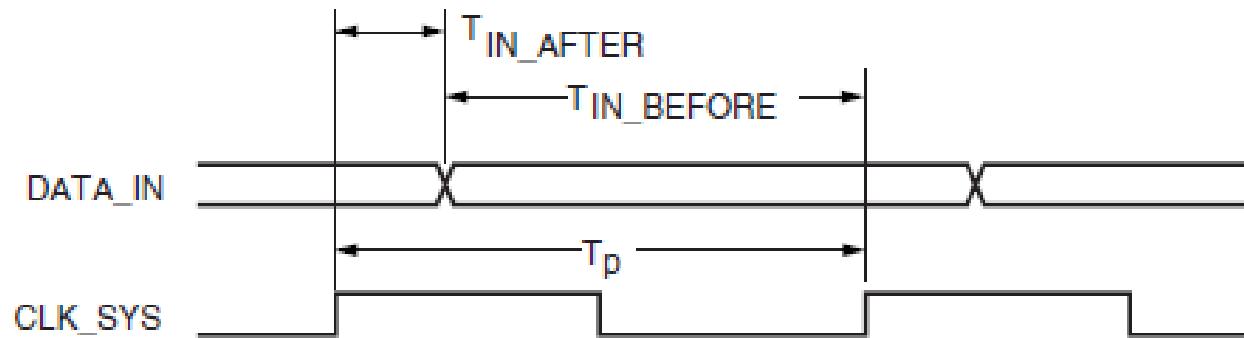


# OFFSET Constraint

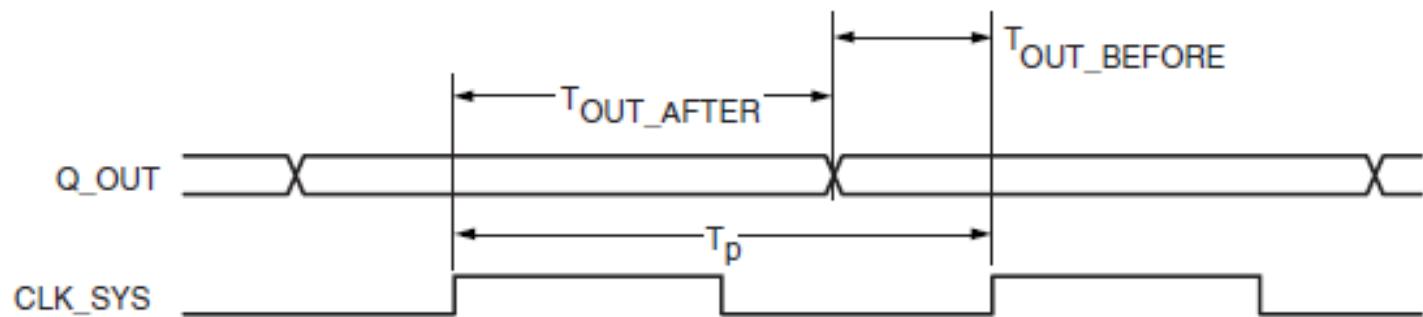
- The OFFSET constraints are used to define the timing relationship between an external clock pad and its associated data-in or data-out pad
- This relationship is also known as constraining the Pad to Setup or Clock to Out paths on the device

# OFFSET Constraint

- OFFSET IN constraint



- OFFSET OUT constraint

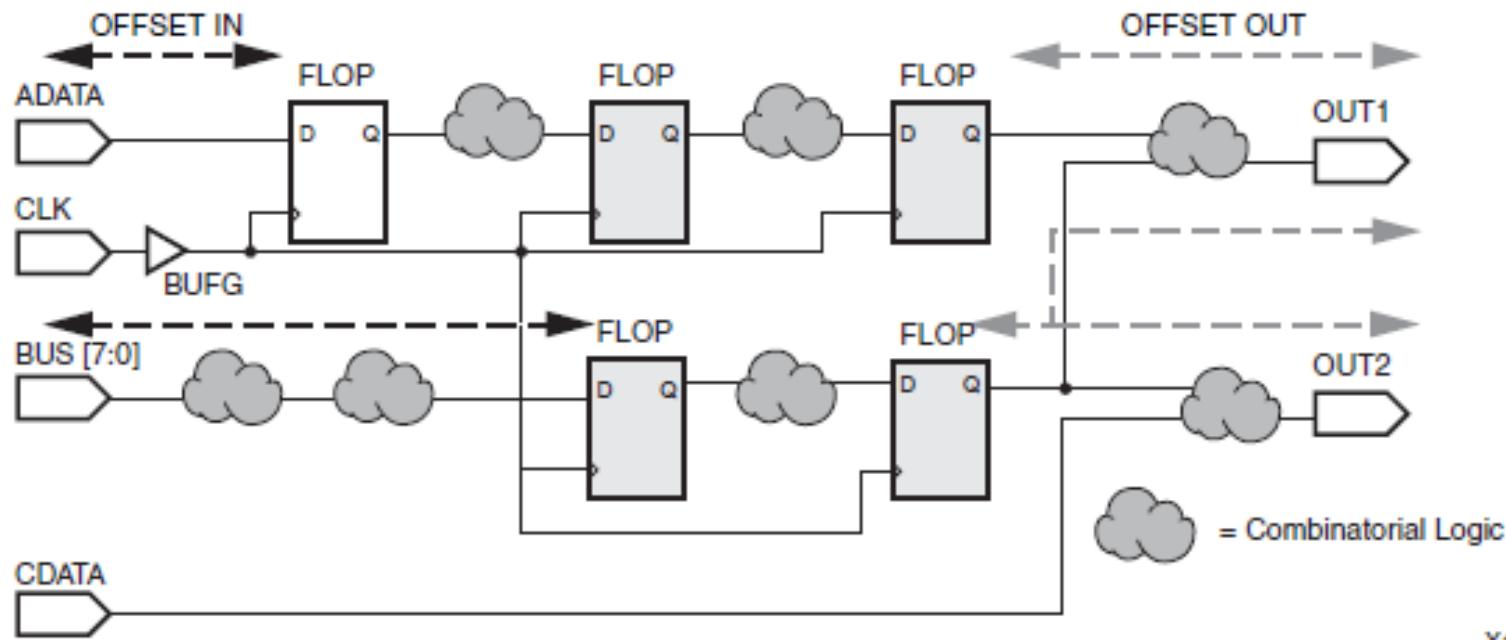




# OFFSET Constraint

- The OFFSET constraint:
  - Includes clock path delay in the analysis for each individual synchronous element
  - Includes paths for all synchronous element types (FFS, RAMS, LATCHES, etc.)
  - Allows a global syntax that allows all inputs or outputs to be constrained with respect to an external clock
  - Analyzes setup and hold time violation on inputs

# OFFSET Constraint



- The **OFFSET** constraint is analyzed with respect to only a single clock edge!



# FROM:TO (Multi-Cycle) Constraint

- A multi-cycle path is path that is allowed to take multiple clock cycles
- A multi-cycle constraint is applied by using a FROM:TO constraint
- FROM:TO constraint have a higher priority than a PERIOD constraint,
- It removes the specified paths from the PERIOD to the FROM:TO constraint
- A FROM:TO constraint begins at a synchronous element and ends at a synchronous element



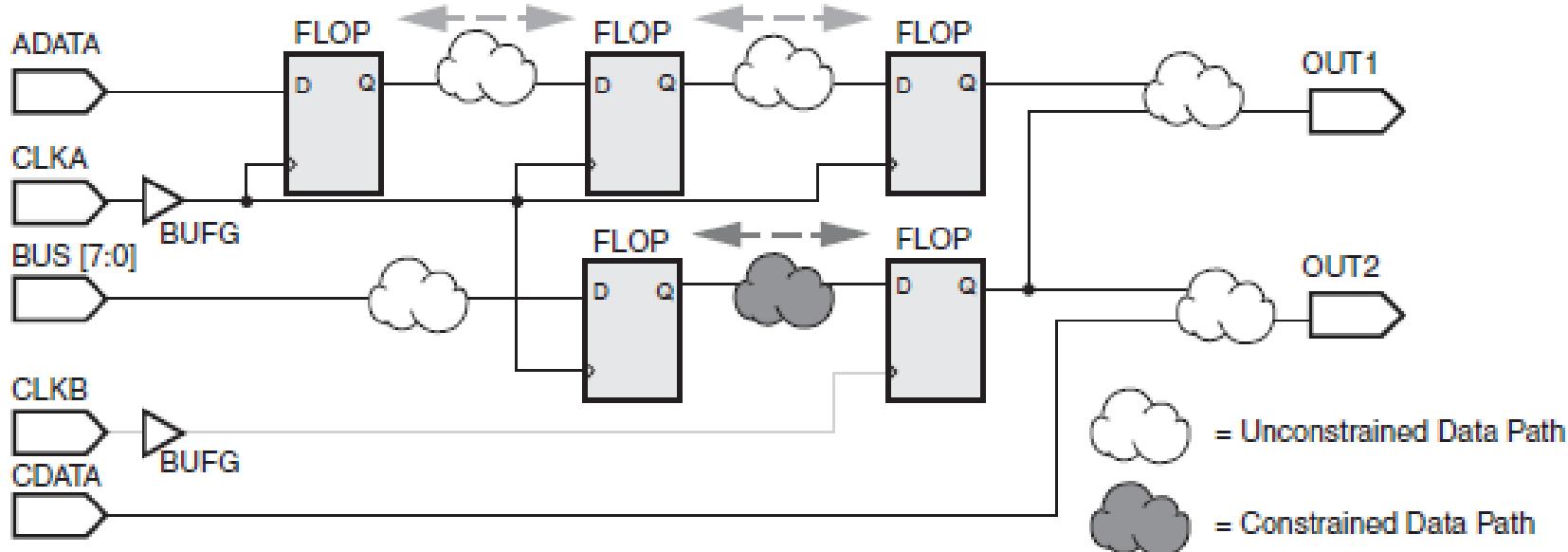
# FROM:TO (Multi-Cycle) Constraint

- Possible combinations:

- FROM:TO
- FROM:THRU:TO
- THRU:TO
- FROM:THRU
- FROM
- TO
- FROM:THRU:THRU:THRU:TO

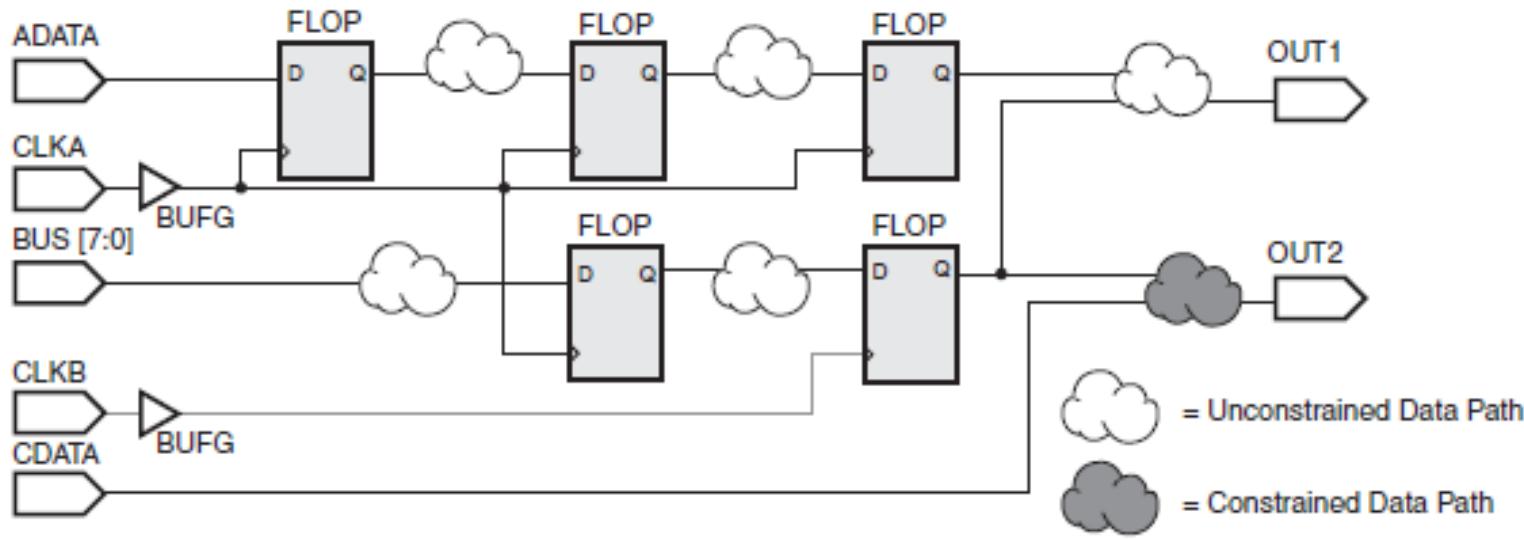
# FROM:TO (Multi-Cycle) Constraint

- A FROM:TO constraint can cover the multi-cycle paths that cover the path between clock domains



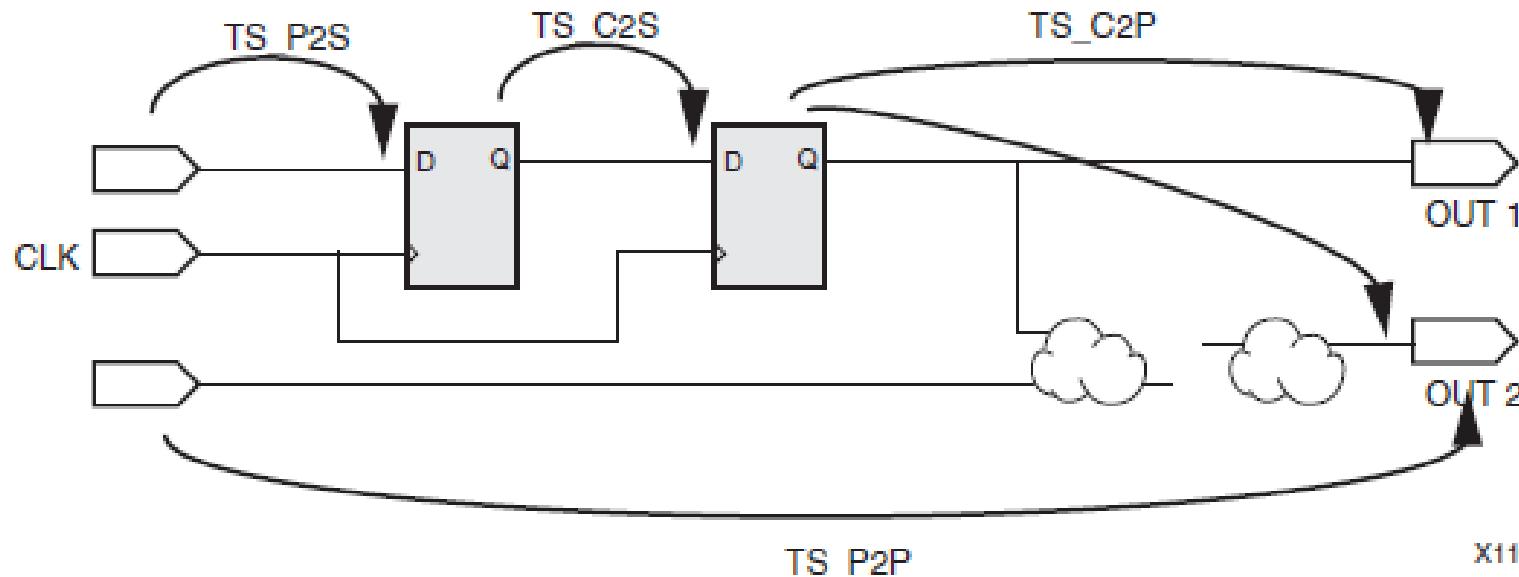
# FROM:TO - Pad2Pad

- One of the fundamental FROM:TO constraints is the Pad to Pads path or asynchronous paths of the design
- It constraints purely combinatorial paths with the start and endpoints at the Pads of the



# FROM:TO - Example

- TIMESPEC TS\_C2S = FROM FFS TO FFS 12 ns;
- TIMESPEC TS\_P2S = FROM PADS TO FFS 10 ns;
- TIMESPEC TS\_P2P = FROM PADS TO PADS 13 ns;
- TIMESPEC TS\_C2P = FROM FFS TO PADS 8 ns;





# Timing groups

- All design elements with same TNM/TNM\_NET attribute are considered a timing group.
- A design element may be in multiple timing groups (TNM/TNM\_NET).
- The TNM/TNM\_NET attributes can be applied to:
  - Net Connectivity (NET)
  - Instance/Module - INST
  - Instance Pin - PIN

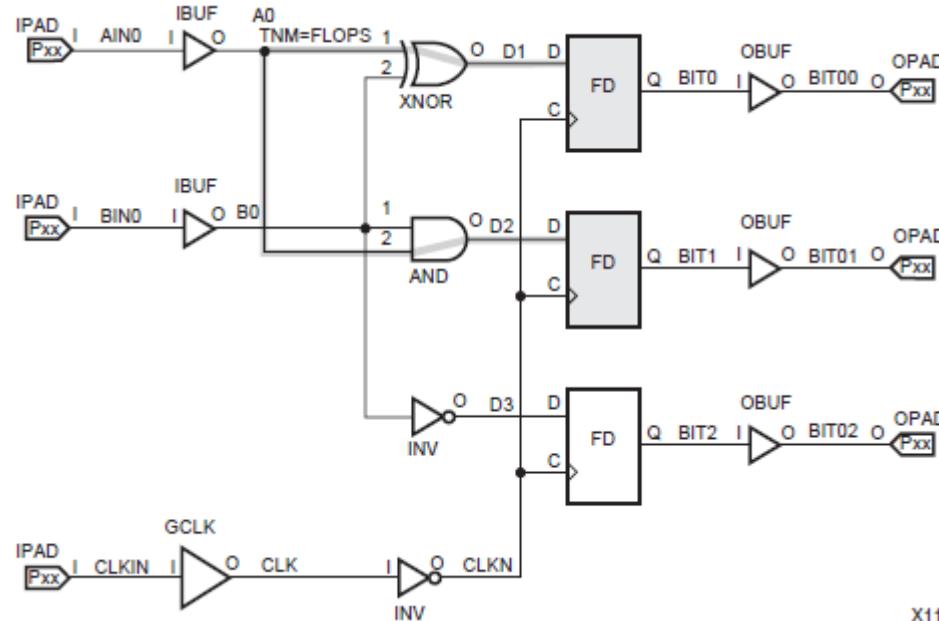


# Timing groups - NET

- Identifying groups by net connectivity allows the grouping of elements by specifying a net or signal that eventually drives synchronous elements and pads
- Those synchronous elements are then tagged with the same timing name attribute

# Timing groups - NET

- The TNM is traced forward along the path (through any number of gates, buffers, or combinatorial logic) until it reaches a flip-flop, input latch, or synchronous element





# Predefined Timing groups

- FFS - All SLICE and IOB edge-triggered flip-flops and shift registers
- PADS - All I/O pads
- DSPS
- RAMS - All single-port and dual-port SLICE LUT RAMs and block Rams
- MULTS
- LATCHES - All SLICE level-sensitive latches



# Timing groups - INST, PIN

- Identifying groups by pin connectivity allows you to group elements by specifying a pin that eventually drives synchronous elements and pads
- If a TNM attribute is placed on a pin, the constraints parser traces the pin downstream to the synchronous elements



# Pattern matching

- Pattern matching is as follows:
  - Asterisk (\*)
    - Matches any string of zero or more characters
  - Question Mark (?)
    - Matches a single character



# Timing groups - Examples

- **INST my\_core TNM = RAMS my\_rams;**
  - This time group (my\_rams) is the RAM components of the hierarchical block my\_core
- **TIMSPEC TS01 = FROM FFS TO my\_rams 14.24ns;**



# Timing groups - Examples

- **.NET** `clock_enable TNM_NET = RAMS(address*) fast_rams;`
  - This time group (`fast_rams`) is the RAM components driven by net name of `clock_enable` with an output net name of `address*`
- **TIMSPEC** `TS01 = FROM FFS TO fast_rams 12.48ns;`



# Timing groups - Examples

- TIMEGRP "larger\_grp" = "small\_grp"  
"medium\_grp";
  - Combines small\_grp and medium\_grp into a larger group called larger\_grp
- TIMEGRP new\_time\_group =  
Original\_time\_group EXCEPT  
a\_few\_items\_time\_grp;
  - Removes the elements of  
a\_few\_items\_time\_grp from  
Original\_time\_group.



# Constraint priorities

- Each constraint type has different priority levels
- Constraints from highest to lowest priorities
  - Timing Ignore (TIG)
  - FROM:THRU:TO
  - FROM:TO
  - OFFSET
  - PERIOD



# Timing Constraints Methodology



# Basic Constraints Methodology

- Timing requirements depend on the type of path to be covered:
  - Input paths
  - Synchronous element to synchronous element paths
  - Output Paths
  - Path specific exceptions
- The most efficient way to specify these constraints is to begin with global constraints and add path specific



# Input Timing Constraints

- Input timing covers the data path from the external pin of the FPGA device to the internal register that captures that data
- The constraint used to specify the input timing is the OFFSET IN constraint
- The OFFSET IN constraint defines the relationship between the data and the clock edge used to capture that data at the pins of the FPGA device

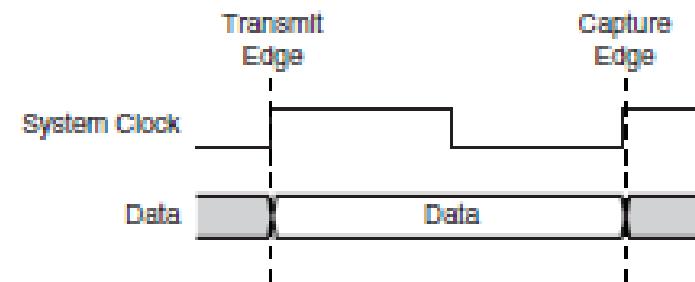
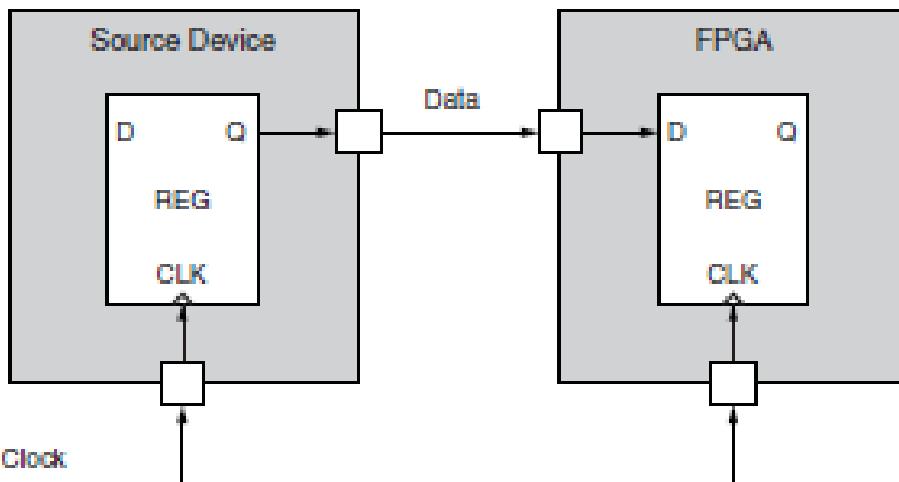


# Input Timing Constraints

- Factors affecting the delay of the input clock and data:
  - Frequency and phase transformations of the clock
  - Clock uncertainties
  - Data delay adjustments
- By default, the OFFSET IN constraint covers all paths from the input pads of the FPGA device to the internal synchronous elements that capture that data and are triggered by the specified OFFSET IN clock.

# System Synchronous Inputs

- In a system synchronous interface, a common system clock both transfers and captures the data.
- The board trace delays and clock skew limit the operating frequency of the





# System Synchronous Inputs

- The input timing for a system synchronous interface should be defined by OFFSET IN constraint
- To specify the input timing:
  - Define the clock PERIOD constraint for the input clock associated with the interface
  - Define the global OFFSET IN constraint for the interface



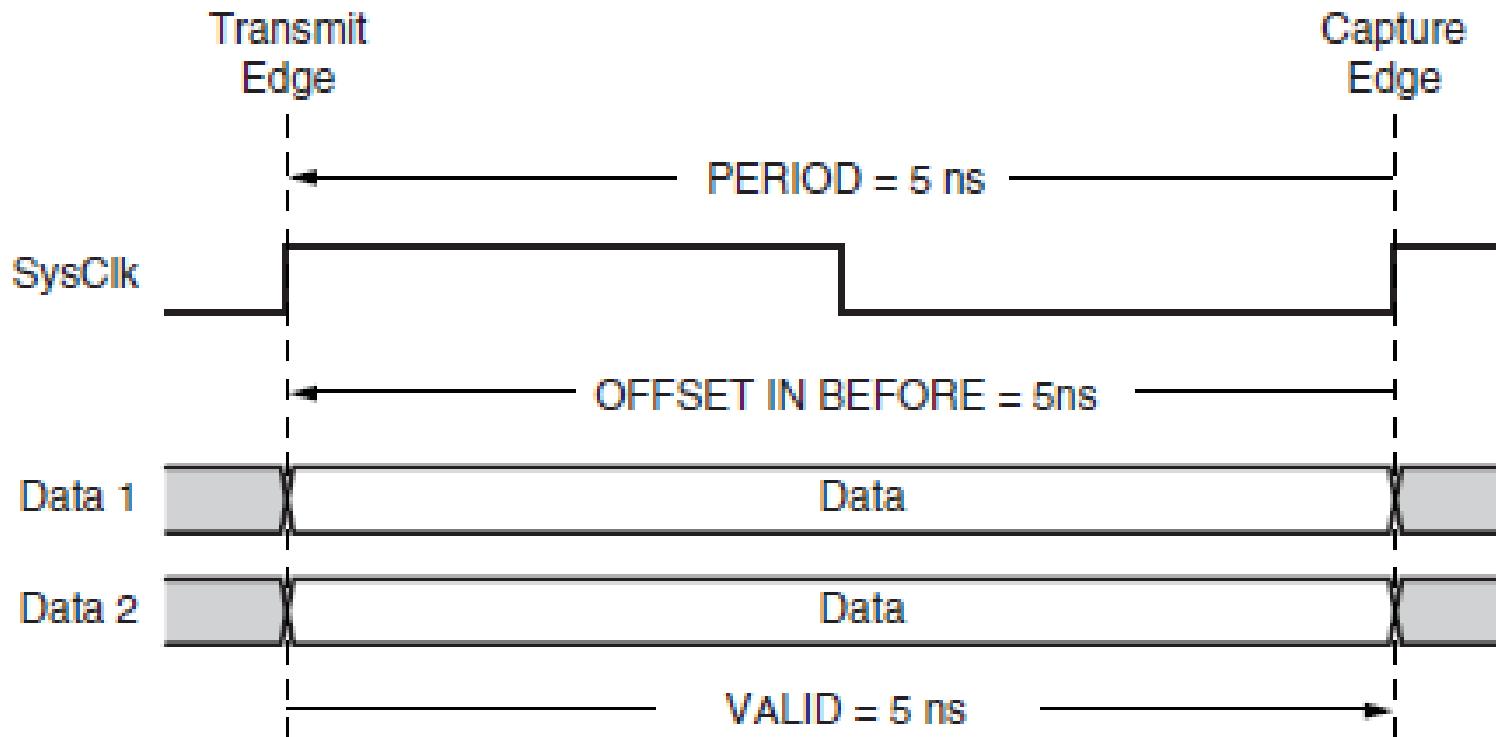
# OFFSET IN constraint

- The global OFFSET IN constraint is:

OFFSET = IN value VALID value BEFORE clock;

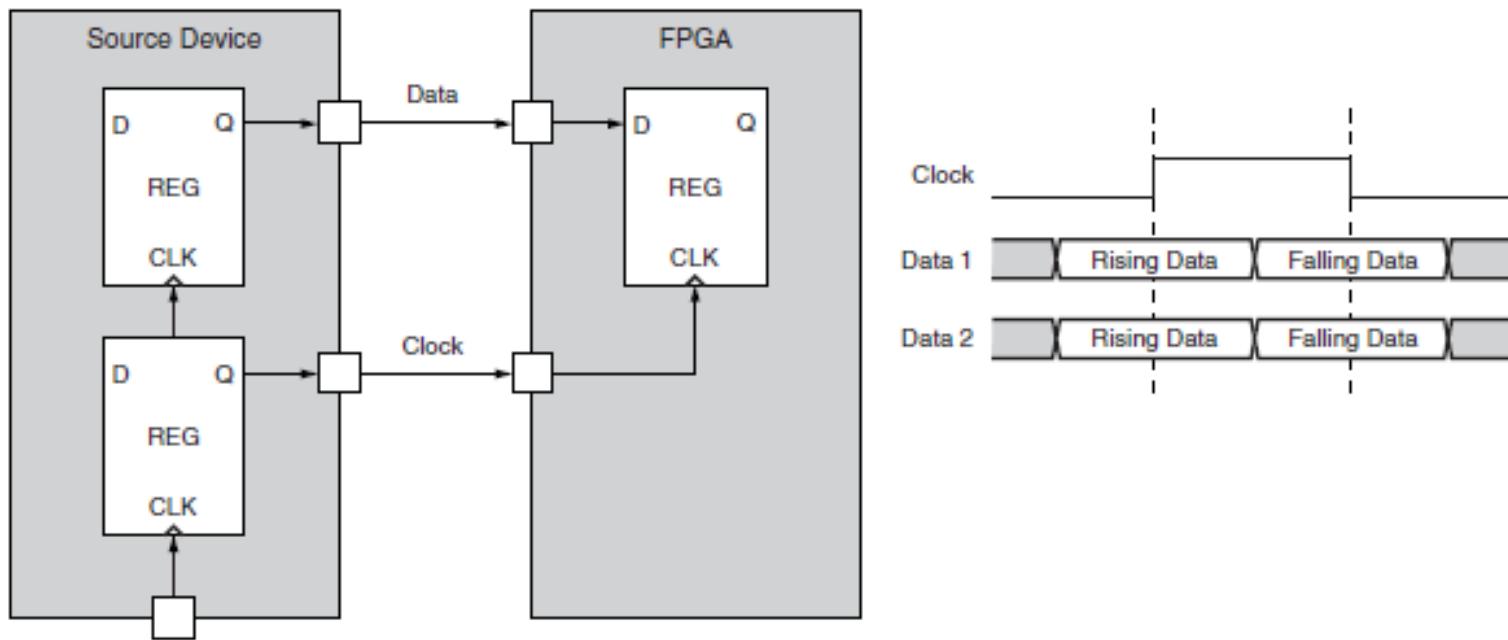
- The OFFSET=IN <value> determines the time from the capturing clock edge to the time in which data first becomes valid
- The VALID <value> determines the duration in which data remains valid

# OFFSET IN constraint



# Source Synchronous Inputs

- In a source synchronous input interface, a clock is regenerated and transmitted along with the data from the source device along similar board traces

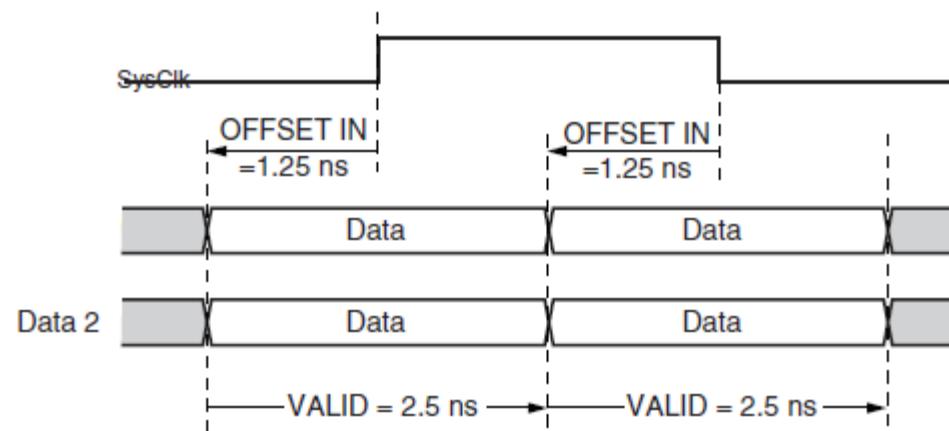




# Source Synchronous Inputs

- Source Synchronous Inputs are faster thus are often used in DDR applications
- To specify the input timing for DDR application:
  - Define the clock PERIOD constraint for the input clock associated with the interface
  - Define the global OFFSET IN constraint for the rising edge (RISING) of the interface
  - Define the global OFFSET IN constraint for the falling edge (FALLING) of the interface

# Source Synchronous Inputs



- NET "SysClk" TNM\_NET = "SysClk";
- TIMESPEC "TS\_SysClk" = PERIOD "SysClk" 5 ns HIGH 50%;
- OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE "SysClk" RISING;
- OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE "SysClk" FALLING;



# Register-To-Register Timing Constraints

- Register-to-register or synchronous element to synchronous element path constraints cover the synchronous data paths between internal registers



# PERIOD Timing Constraints

- The PERIOD constraint:
  - Defines the timing requirements of the clock domains
  - Analyzes the paths within a single clock domain
  - Analyzes all paths between related clock domains
  - Takes into account all frequency, phase, and uncertainty differences between the clock domains during analysis



# Constraining synchronous clock domains

- Categories:
  - Automatically Related Synchronous DCM/PLL Clock Domains
  - Manually Related Synchronous Clock Domains
  - Asynchronous Clock Domains



# Automatically Related Synchronous DCM/PLL Clock Domains

- The most common type of clock circuit is one in which:
  - The input clock is fed into a DLL/DCM/PLL
  - The outputs are used to clock the synchronous paths in the device
- The PERIOD constraint should be used



# Automatically Related Synchronous DCM/PLL Clock Domains

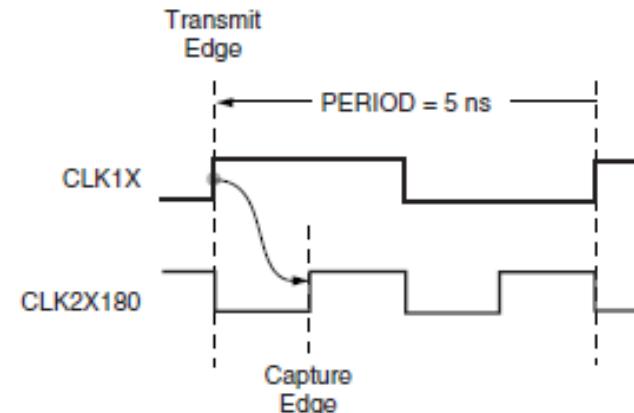
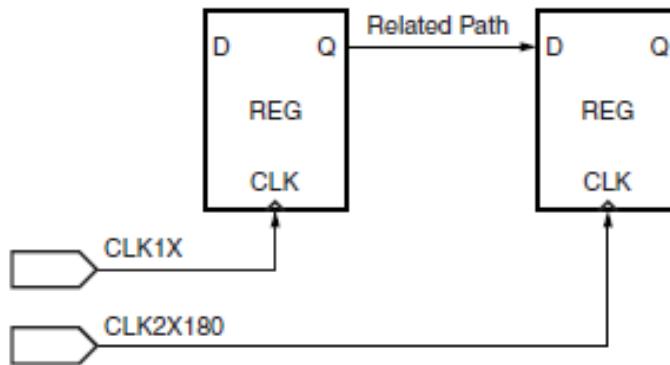
- By placing the PERIOD constraint on the input clock, the Xilinx tools automatically:
  - Derive a new PERIOD constraint for each of the DLL/DCM/PLL output clocks
  - Determine the clock relationships between the output clock domains, and automatically perform an analysis for any paths between these clock domains



# Manually Related Synchronous Clock Domains

- In some cases the relationship between synchronous clock domains can not be automatically determined by the tools - for example, when related clocks enter the FPGA device on separate pins.
- In this case it is recommended to:
  - Define a separate PERIOD constraint for each input clock
  - Define a manual relationship between the clocks

# Manually Related Synchronous Clock Domains



- NET "Clk1X" TNM\_NET = "Clk1X";
- NET "Clk2X180" TNM\_NET = "Clk2X180";
- TIMESPEC "TS\_Clk1X" = PERIOD "Clk1X" 5 ns;
- TIMESPEC "TS\_Clk2X180" = PERIOD "Clk2X180" TS\_Clk1X/2 PHASE + 1.25 ns ;



# Asynchronous Clock Domains

- Asynchronous clock domains are those in which the source and destination clocks do not have a frequency or phase relationship
- Since the clocks are not related, it is not possible to determine the final relationship for setup and hold time analysis
- Proper asynchronous design techniques have to be used!



# Asynchronous Clock Domains

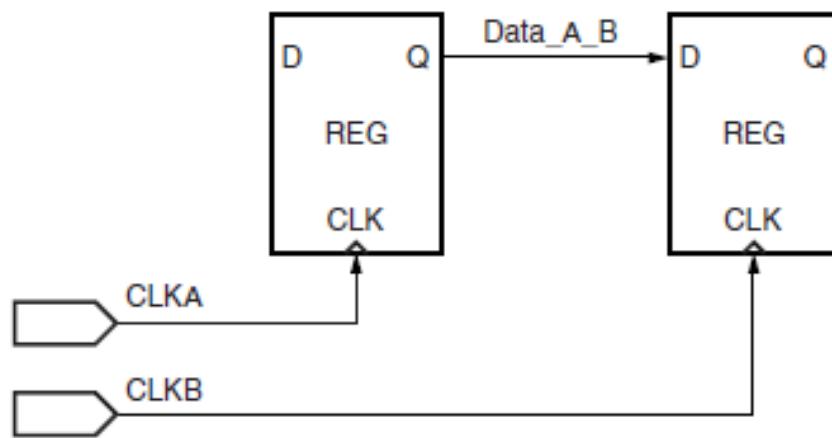
- It is possible to constrain the maximum data path delay without regard to source and destination clock frequency and phase relationship.
- This requirement is specified using the FROM-TO constraint with the DATAPATHONLY keyword.



# Asynchronous Clock Domains

- To constrain of the maximum data path delay without regard to source and destination clock frequency and phase relationship:
  - Define a time group for the source synchronous elements
  - Define a time group for the destination synchronous elements
  - Define the maximum delay of the data paths using the FROM-TO constraint between the two time groups with DATAPATHONLY keyword

# Asynchronous Clock Domains



- NET "CLKA" TNM\_NET = FFS "GRP\_A";
- NET "CLKB" TNM\_NET = FFS "GRP\_B";
- TIMESPEC TS\_Example = FROM "GRP\_A" TO "GRP\_B" 5 ns DATAPATHONLY;

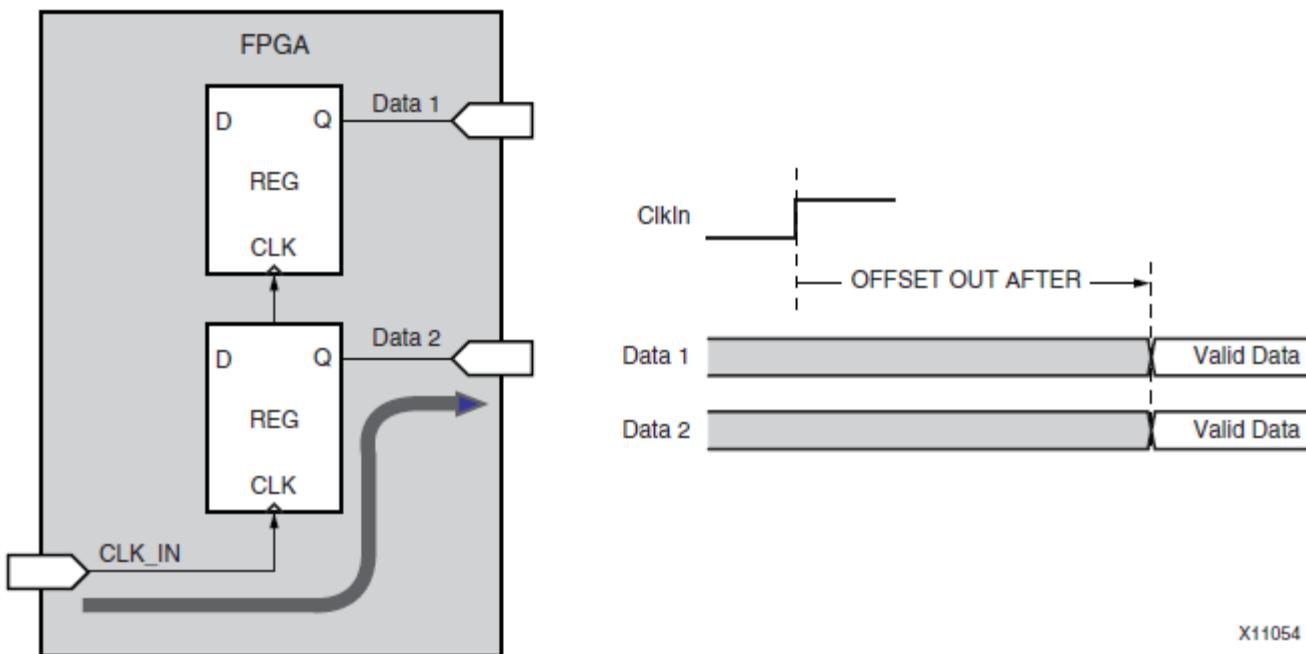


# Output Timing Constraints

- Output timing covers the data path from a register inside the FPGA device to the external pin of the FPGA device - OFFSET OUT constraint should be used.
- The OFFSET OUT constraint defines the maximum time allowed for data to be transmitted from the FPGA device

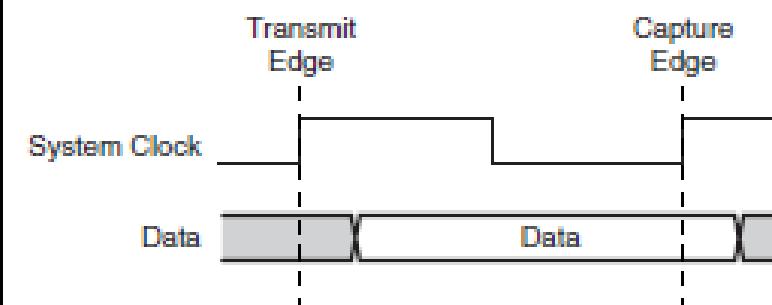
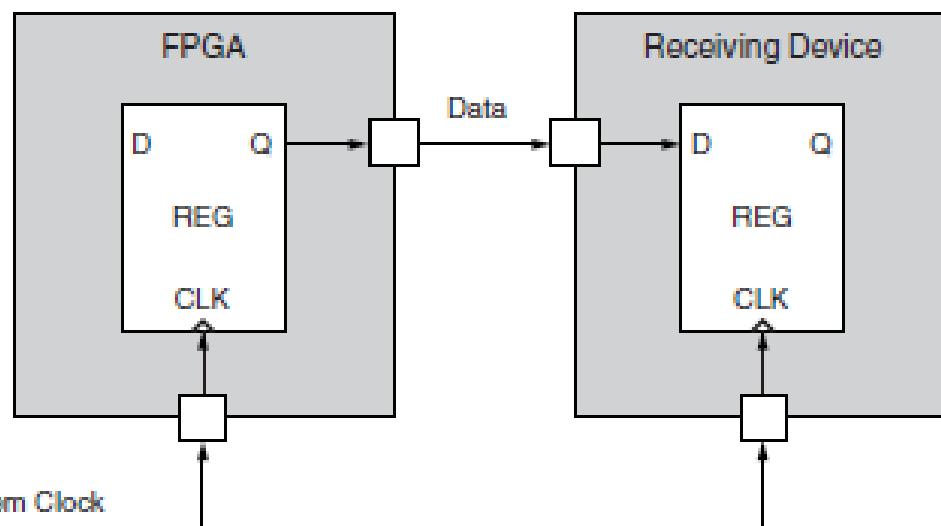
# Output Timing Constraints

- The output delay path begins at the input clock pin of the FPGA device and continues through the output register to the data pins of the FPGA device



# System Synchronous Output

- The system synchronous output interface is an interface in which a common system clock is used to both transfer and capture

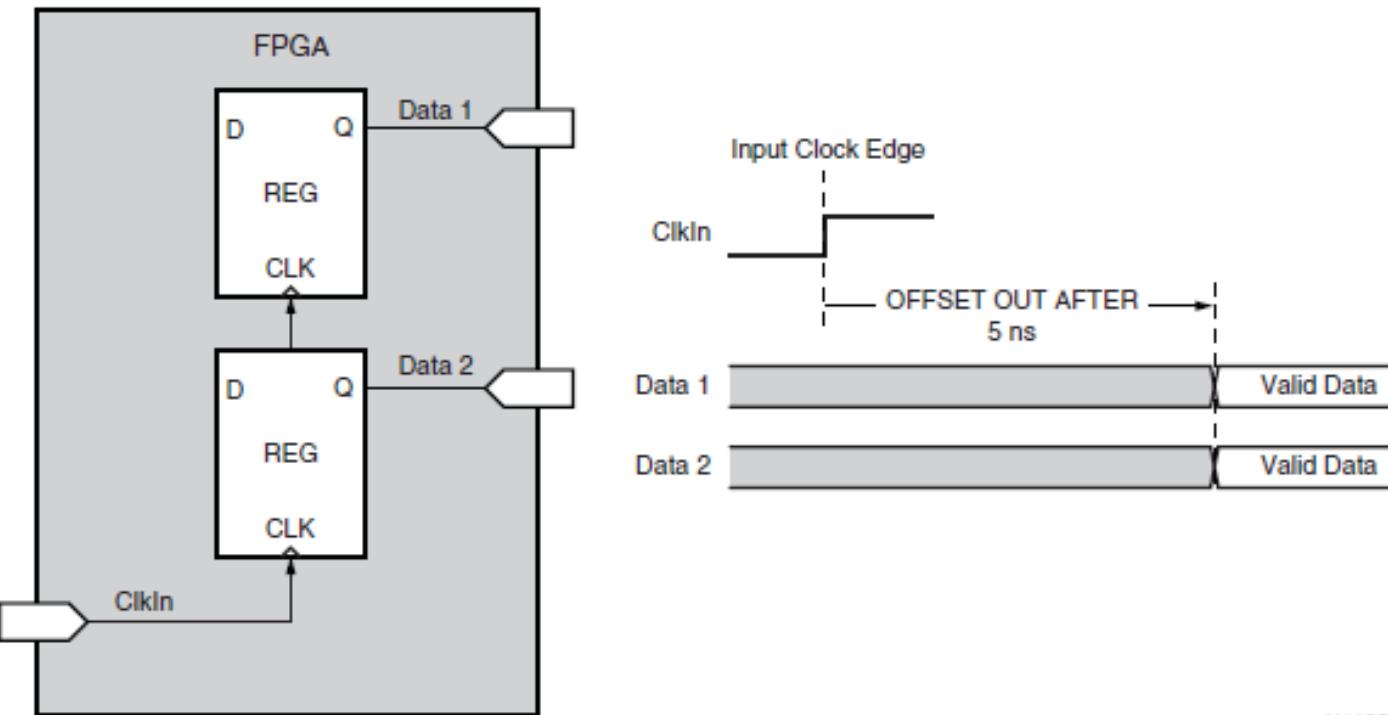




# System Synchronous Output

- One OFFSET OUT constraint should be defined for each system synchronous output interface clock
- To specify the output timing:
  - Define a time name (TNM) for the output clock to create a time group, which contains all output registers triggered by the input clock
  - Define the global OFFSET OUT constraint for the interface

# System Synchronous Output



X11056

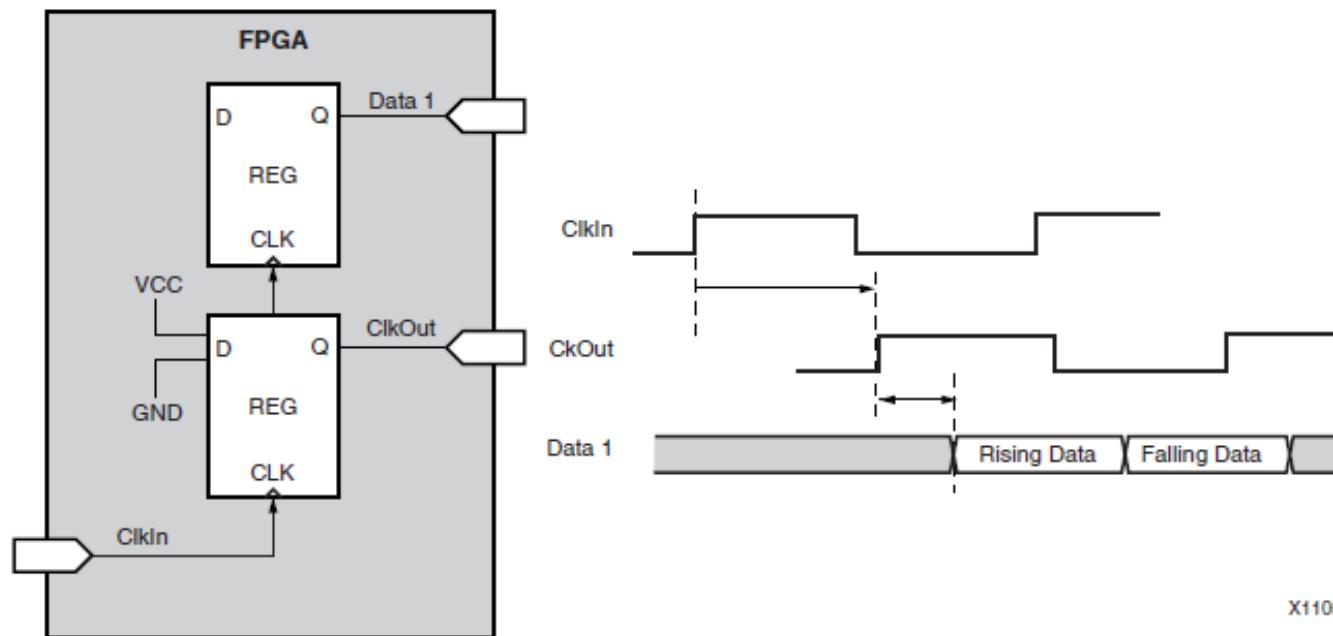
- NET "ClkIn" TNM\_NET = "ClkIn";
- OFFSET = OUT 5 ns AFTER "ClkIn";



# Source Synchronous Output

- The source synchronous output interface is an interface in which a clock is regenerated and transmitted along with the data from the FPGA device.
- The regenerated clock is transmitted along with the data

# Source Synchronous Output



- In this interface, the time from the input clock edge to the output data becoming valid is not as important as the skew between the output data bits



# Source Synchronous Output

- To specify the input timing:
  - Define a time name (TNM) for the output clock to create a time group which contains all output registers triggered by the output clock
  - Define the global OFFSET OUT constraint for the rising edge (RISING) of the interface
  - Define the global OFFSET OUT constraint for the falling edge (FALLING) of the interface



# Timing Exceptions - False Paths

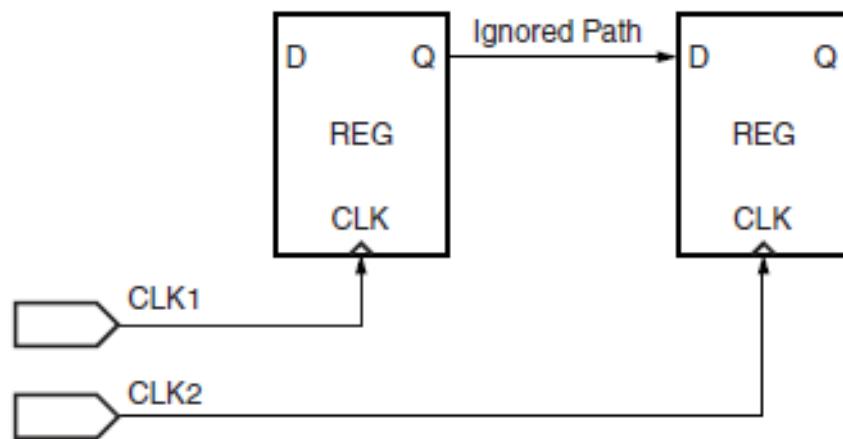
- In some cases, a set of paths may be removed from timing analysis (if these paths do not affect timing performance)
- One common way to specify the set of paths to be removed from timing analysis is to use the FROM-TO constraint with the timing ignore (TIG) keyword



# Timing Exceptions - False Paths

- Using TIG allows to:
  - Specify a set of registers in a source time group
  - Specify a set of registers in a destination time group
  - Automatically remove all paths between those time groups from analysis.
- To specify the timing ignore (TIG) define:
  - A set of registers for the source time group
  - A set of registers for the destination time group
  - A FROM-TO constraint with a TIG keyword to remove the paths between the groups

# Timing Exceptions - False Paths



- NET "CLK1" TNM\_NET = FFS "GRP\_1";
- NET "CLK2" TNM\_NET = FFS "GRP\_2";
- TIMESPEC TS\_Example = FROM "GRP\_1" TO "GRP\_2" TIG;



# Timing Constraints Analysis



# Timing Constraints Analysis

- The traditional timing report contains the following sections:
  - 1. Constraint Details - Path details per constraint
  - 2. Data Sheet Section - General Setup, Hold, and Clock to Out times
  - 3. Summary - Timing Errors/Score, Constraint Coverage, and Design Statistics



# Types of Timing Reports

- Analysis Against Design Timing Constraints
  - Compares design performance with timing constraints
  - Most commonly used report format
    - Used for Post-Map and Post-Place & Route Static Timing Reports if the design contains constraints
- Analysis Against Auto-Generated Design Constraints
  - Determines the longest paths in each clock domain
  - Use with designs that have no constraints defined



# Types of Timing Reports

- Analysis Against User Specified Paths by Defining Endpoints
  - Custom report for selecting sources and destinations
- Analysis Against User Specified Paths by Defining Clock and I/O Timing
  - Allows you to define PERIOD and OFFSET constraints on-the-fly
  - Use with designs that have no constraints defined



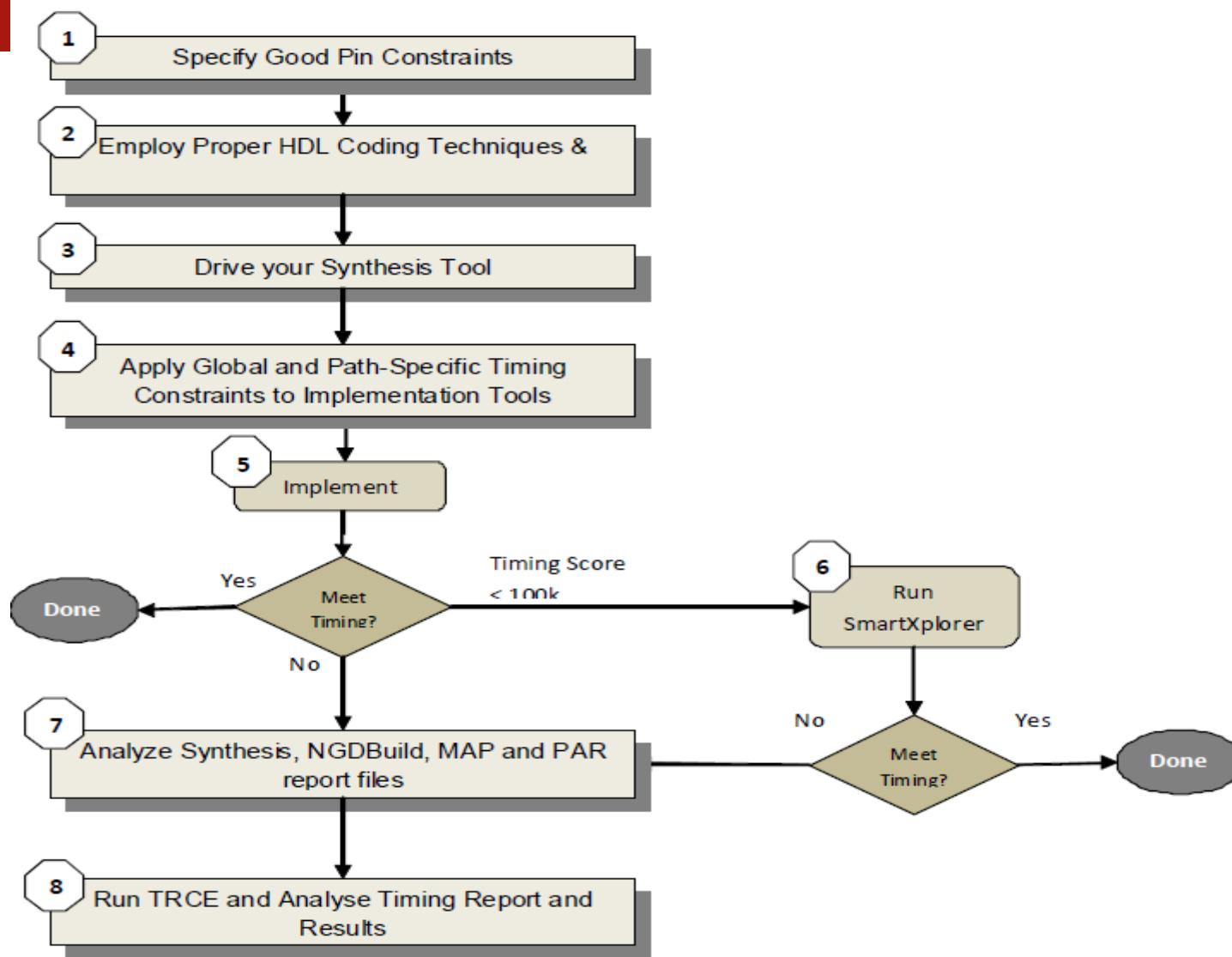
# Timing Closure



# Timing Closure

- Timing closure is achieved when the timing score for a given design is zero. The timing score:
  - Is the total value representing the timing analysis for all constraints, and the amount by which the constraints are failing
  - Is the *sum* in picoseconds of all timing constraints that have not been met
  - Shows the total amount of error (in picoseconds) for all timing constraints in the design

# Achieving Timing Closures





# Achieving Timing Closures

- **Timing constraints define timing objectives**
  - Over-constraining gets nothing, but costs extra PAR time
  - Always use timing constraints, even when your timing objective is modest
- **Unrealistic timing constraints will cause the tools to stop**
  - Your synthesis tool's timing report and the Post-Map Static Timing Report contain performance estimates
  - Both will tell you if your constraints are realistic
- **After implementing, review the Post-Place & Route Static Timing Report to determine if your objectives were met**
  - If your constraints failed, use the Timing Report to determine the cause



# Analyzing Post-Place & Route Timing

- There are many factors that contribute to timing errors, including
  - Poor micro-architecture
  - Neglecting synchronous design rules or using incorrect HDL coding style
  - Poor synthesis results (too many logic levels in the path)
  - Inaccurate or incomplete timing constraints
  - Poor logic mapping or placement



# Analyzing Post-Place & Route Timing

- Each root cause has a different solution
  - Rewrite HDL code
  - Ensure that synthesis constraints are correct and use proper synthesis options
  - Add path-specific timing constraints
  - Resynthesize or reimplement with different software options
- Correct interpretation of timing reports can reveal the most likely cause
  - Therefore, the most likely solution



**Thank you for your attention**



# References

- [1] Tim Behne, „FPGA Clock Schemes”
- [2] Spartan-6 family documentation; [www.xilinx.com](http://www.xilinx.com)
- [3] [http://www.actel.com/documents/Clock\\_Skew\\_AN.pdf](http://www.actel.com/documents/Clock_Skew_AN.pdf)
- [4] <http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf>
- [5] [http://www.xilinx.com/support/documentation/application\\_notes/xapp094.pdf](http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf)
- [6] [https://nepg.nasa.gov/mapld\\_2009/talks/Posters/Landoll\\_David\\_mapld09\\_pres\\_1.ppt#1019,12,Clock Domain Crossings Guaranteed to Cause Metastability](https://nepg.nasa.gov/mapld_2009/talks/Posters/Landoll_David_mapld09_pres_1.ppt#1019,12,Clock Domain Crossings Guaranteed to Cause Metastability)
- [7] [https://arco.esi.uclm.es/public/doc/tutoriales/Xilinx/old\\_training/FPGA%20Design/achieving-timing-closure.ppt#300,3,Timing Closure](https://arco.esi.uclm.es/public/doc/tutoriales/Xilinx/old_training/FPGA%20Design/achieving-timing-closure.ppt#300,3,Timing Closure)



# References

[8] Tim



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



# Politechnika Wrocławska

## Układy programowalne w technologii FPGA

Wykład 13  
Przegląd rynku CPLD, FPGA i ASIC

Wrocław 2020



# Agenda

- Altera family
  - MAX series
  - Cyclone series
  - Aria series
  - Stratix series
- Xilinx family
  - CoolRunner II series
  - EasyPath-6 series
  - Spartan-6 series
  - Virtex-6 series
  - 7-th series
  - SoCs



**ALTERA.**®

Altera family



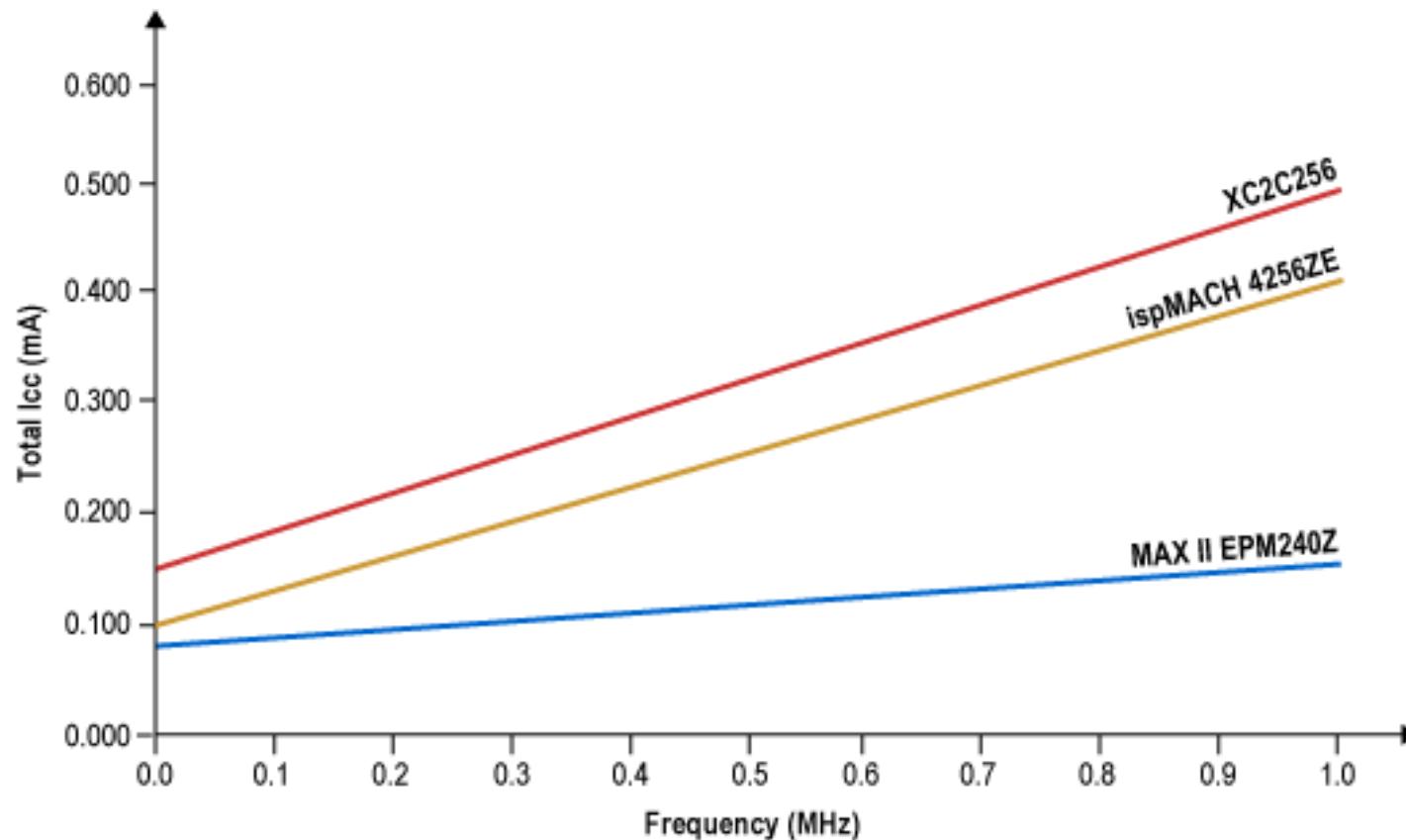
# Altera family

- Altera family of FPD consists of:
  - CPLD - MAX II series, MAX V series
  - ASIC - HardCopy series
  - FPGA:
    - Cyclone IV, Cyclone V (low end)
    - Aria II, Aria V
    - Stratix IV, Stratix V (high end)



# MAX II series - features

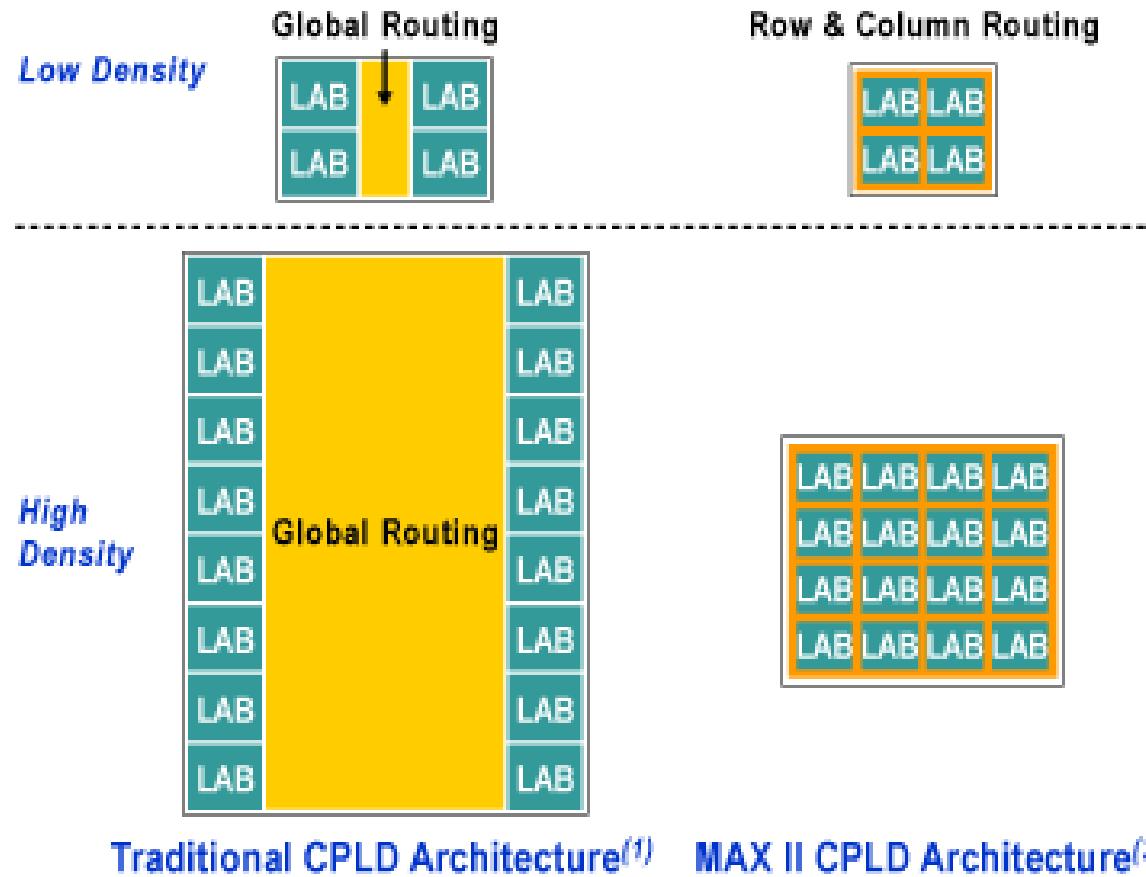
- Low power consumption



Source: [1]

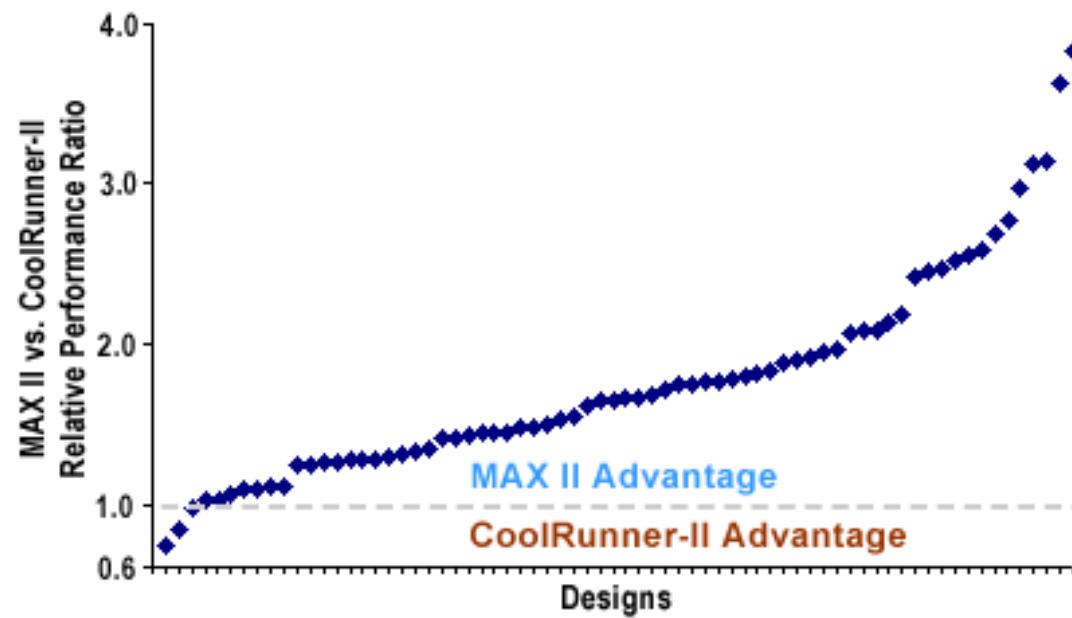
# MAX II series - features

- Modified, low-cost architecture



# MAX II series - features

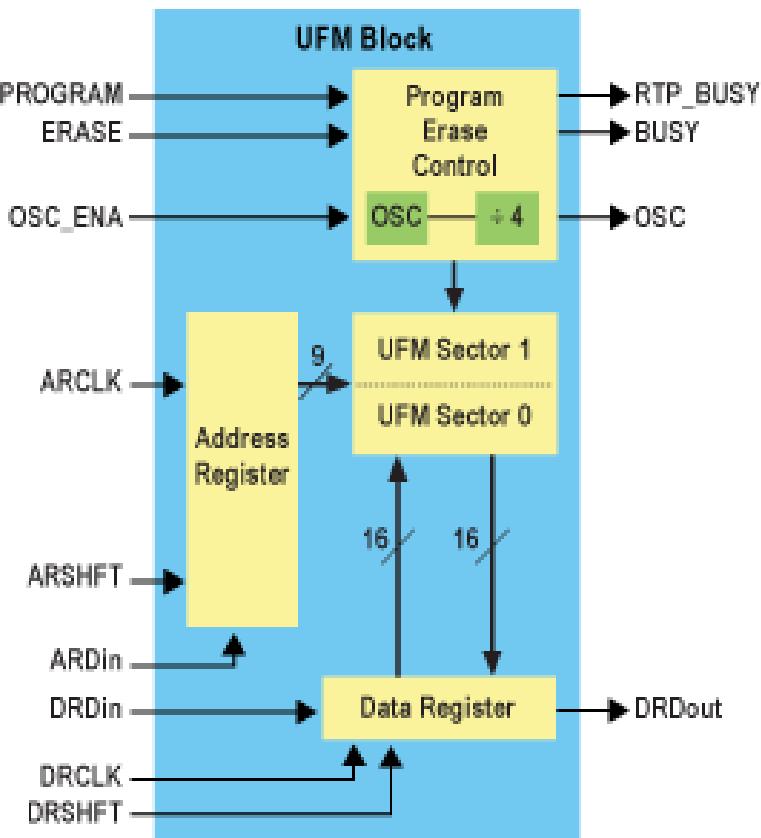
- High performance
  - Clock up to 300MHz



Source: [2]

# MAX II series - features

- Interesting add-ons on board:
  - Oscillator
  - User flash memory (8kb)



Source: [2]



# MAX II series - features

- Many I/O standards available:

I/O Standard	Performance
3.3-V LVTT/LVCMOS	300 MHz
2.5-V LVTT/LVCMOS	220 MHz
1.8-V LVTT/LVCMOS	200 MHz
1.5-V LVCMOS	150 MHz
3.3-V PCI <a href="#">(1)</a>	66 MHz



# MAX II series - features

- Many I/O standards available:

Feature	Benefit
3.3-, 2.5-, 1.8- & 1.5-V LVTTL/LVCMSOS	Enables broad application support and compatibility with other devices on board
MultiVolt™ I/O With Multiple I/O Banks	Up to four I/O banks seamlessly interface to other devices at 3.3-, 2.5-, 1.8-, and 1.5-V voltage levels
PCI Support <a href="#">(1)</a>	Enables support for the 32-bit, 66-MHz PCI standard
Schmitt Triggers	Aids in noise tolerance on inputs with up to 300 mV of hysteresis on 3.3-V inputs and 160 mV of hysteresis on 2.5-V inputs
Programmable Drive Strength and Slew Rate	Allows you to control and improve signal integrity
Single Output Enable (OE) per I/O Pin	Numerous OEs allow you to use smaller devices, reducing cost
Hot-Socketing Support	Enables safe insertion or removal of device from powered systems
Fast I/O Connection	Enables fast $t_{PD}$ and $t_{CO}$ timing



# MAX V family

- Low-cost, low-power, and non-volatile CPLD architecture
- Instant-on (0.5 ms or less) configuration time
- Standby current as low as 25 µA and fast power-down/reset operation
- Fast propagation delay and clock-to-output times
- Internal oscillator
- Emulated LVDS output support with a data rate of up to 304 Mbps



# MAX V family

- Four global clocks with two clocks available per logic array block (LAB)
- User flash memory block up to 8 Kbits for non-volatile storage with up to 1000 read/write cycles
- Single 1.8-V external supply for device core
- MultiVolt I/O interface supporting 3.3-V, 2.5-V, 1.8-V, 1.5-V, and 1.2-V logic levels
- Bus-friendly architecture including programmable slew rate, drive strength, bus-hold, and programmable pull-up resistors
- Schmitt triggers enabling noise tolerant inputs (programmable per pin)

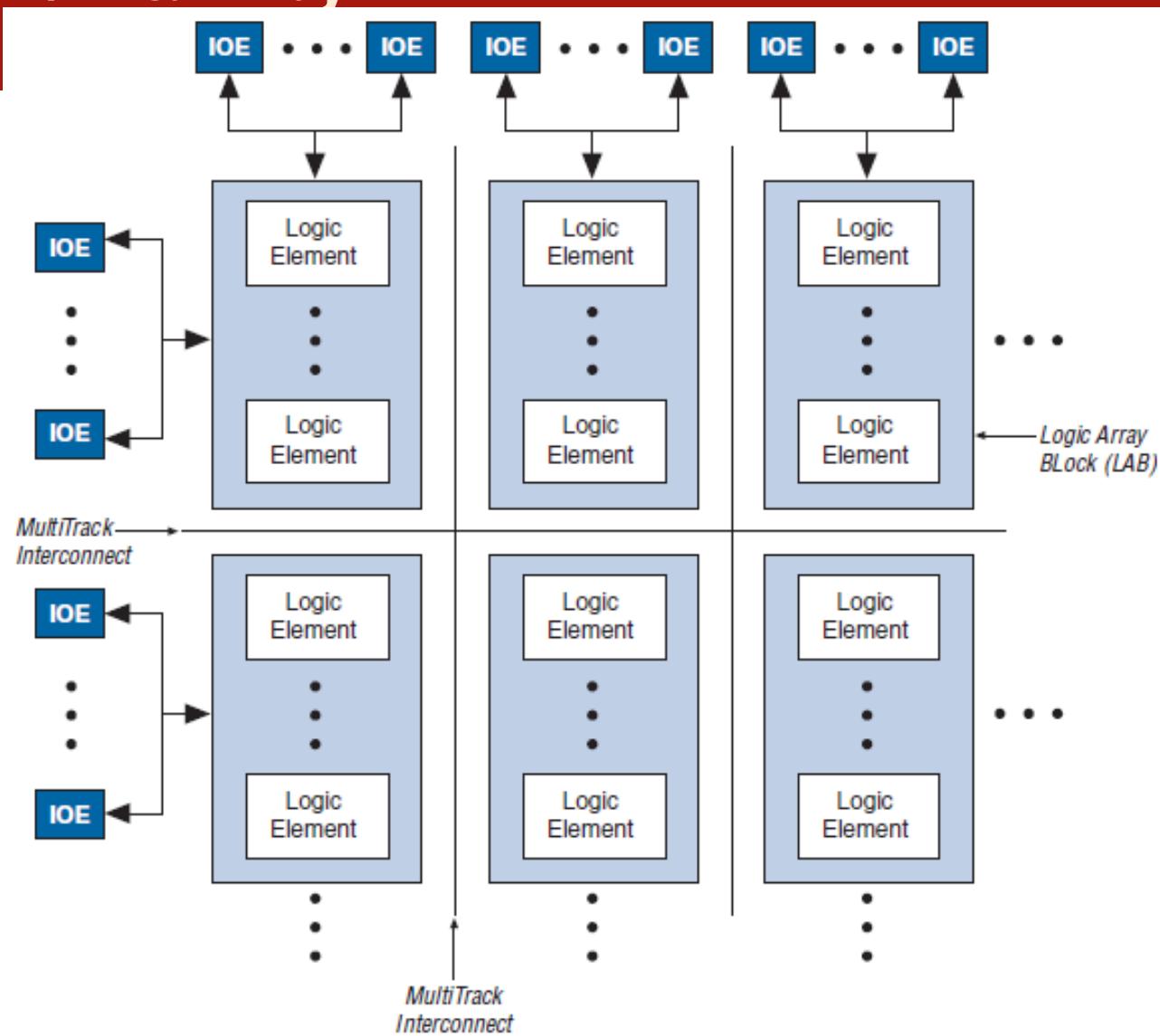
# MAX V family

Feature	5M40Z	5M80Z	5M160Z	5M240Z	5M570Z	5M1270Z	5M2210Z
LEs	40	80	160	240	570	1,270	2,210
Typical Equivalent Macrocells	32	64	128	192	440	980	1,700
User Flash Memory Size (bits)	8,192	8,192	8,192	8,192	8,192	8,192	8,192
Global Clocks	4	4	4	4	4	4	4
Internal Oscillator	1	1	1	1	1	1	1
Maximum User I/O pins	54	79	79	114	159	271	271
$t_{PD1}$ (ns) (1)	7.5	7.5	7.5	7.5	9.0	6.2	7.0
$f_{CNT}$ (MHz) (2)	152	152	152	152	152	304	304
$t_{SU}$ (ns)	2.3	2.3	2.3	2.3	2.2	1.2	1.2
$t_{CO}$ (ns)	6.5	6.5	6.5	6.5	6.7	4.6	4.6

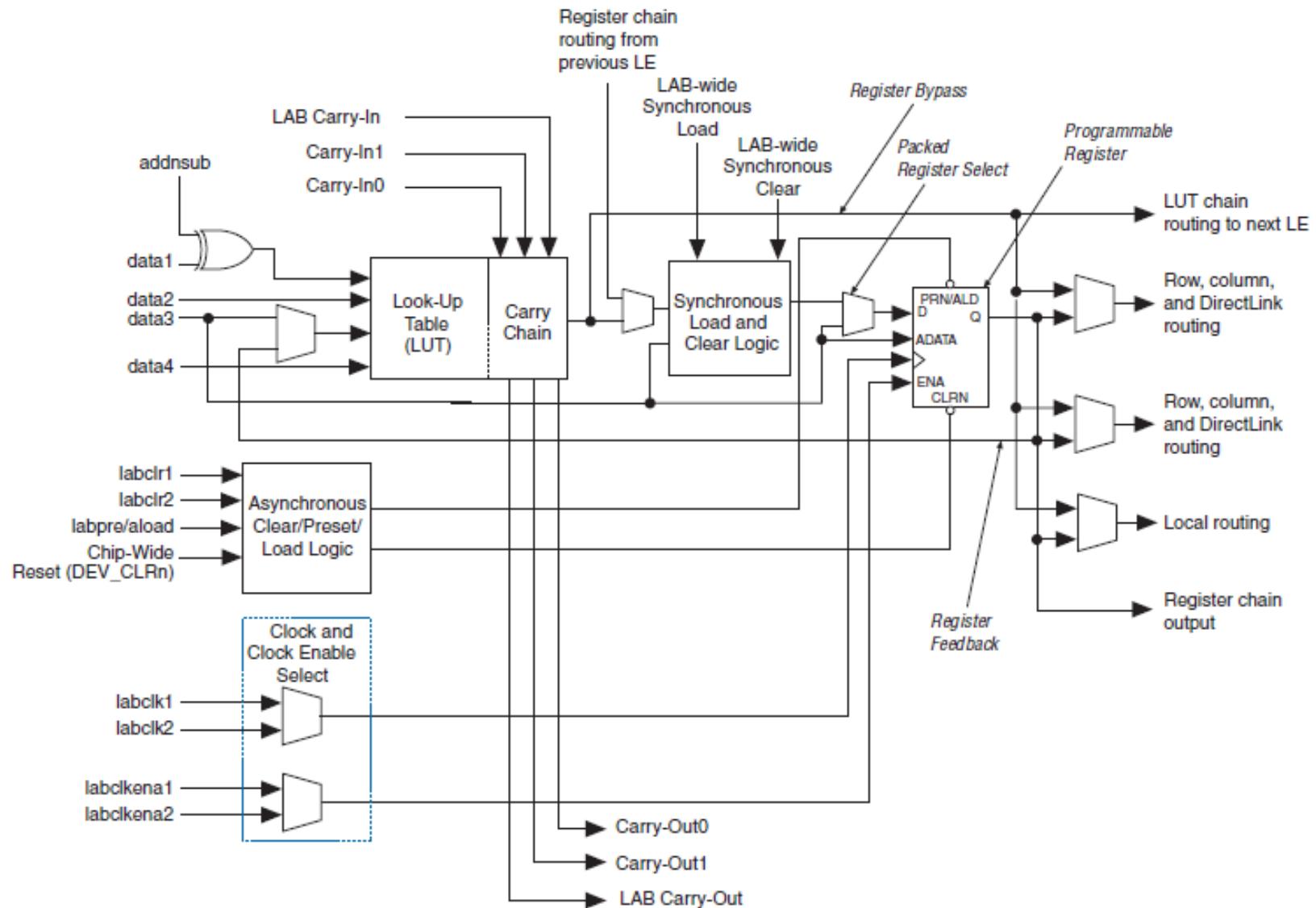
Table 1–2. MAX V Packages and User I/O Pins (Note 1)

Device	64-Pin MBGA	64-Pin EQFP	68-Pin MBGA	100-Pin TQFP	100-Pin MBGA	144-Pin TQFP	256-Pin FBGA	324-Pin FBGA
5M40Z	↑ 30	↑ 54	—	—	—	—	—	—
5M80Z	↓ 30	54	↑ 52	↑ 79	—	—	—	—
5M160Z	—	↓ 54	52	79	↑ 79	—	—	—
5M240Z	—	—	↓ 52	79	79	↑ 114	—	—
5M570Z	—	—	—	↓ 74	74	114	↑ 159	—
5M1270Z	—	—	—	—	—	↓ 114	211	↑ 271
5M2210Z	—	—	—	—	—	—	↓ 203	↓ 271

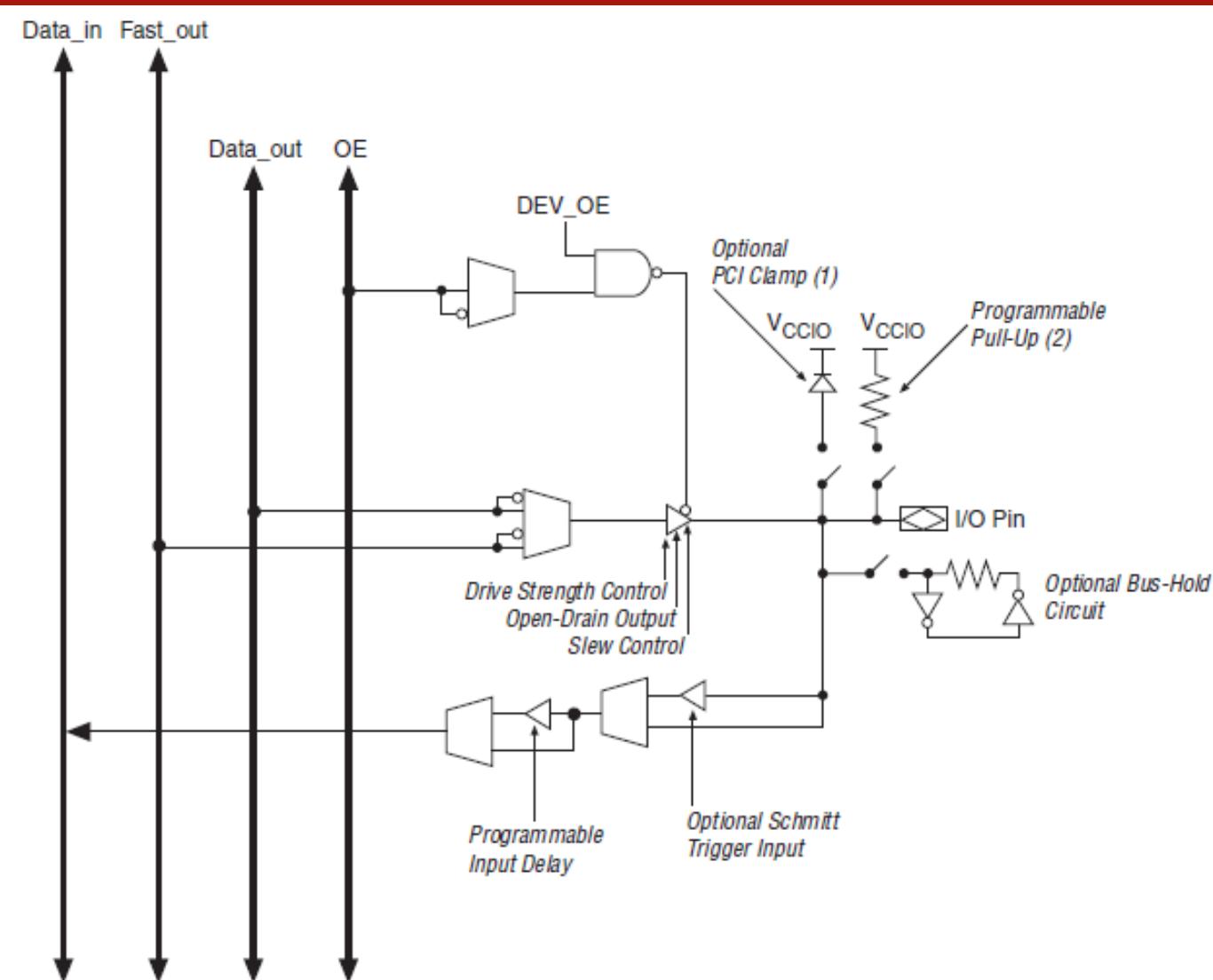
# MAX V family



# MAX V family - logic element



# MAX V family - IOE structure



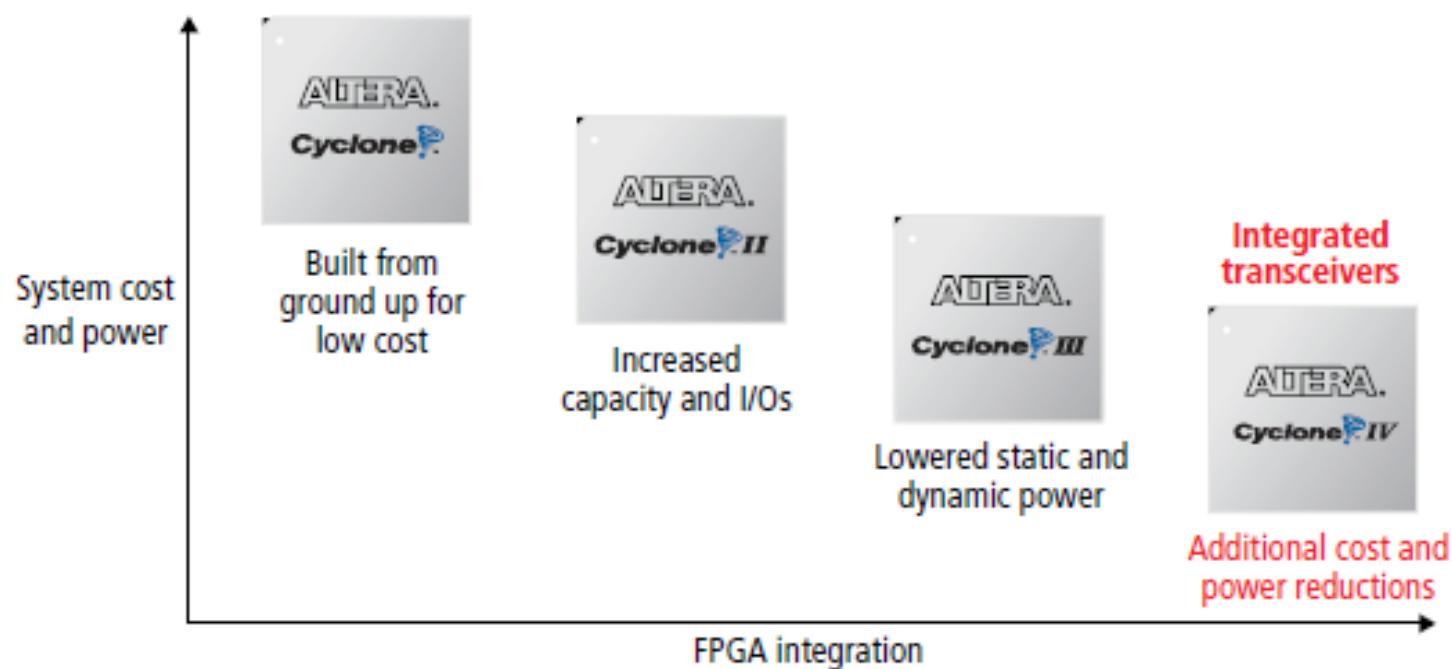
# MAX V family - drive strength

I/O Standard	IOH/IOL Current Strength Setting (mA)
3.3-V LVTTL	16
	8
3.3-V LVCMOS	8
	4
2.5-V LVTTL/LVCMOS	14
	7
1.8-V LVTTL/LVCMOS	6
	3
1.5-V LVCMOS	4
	2
1.2-V LVCMOS	3



# Cyclone IV series - features

Integration lowers cost and power



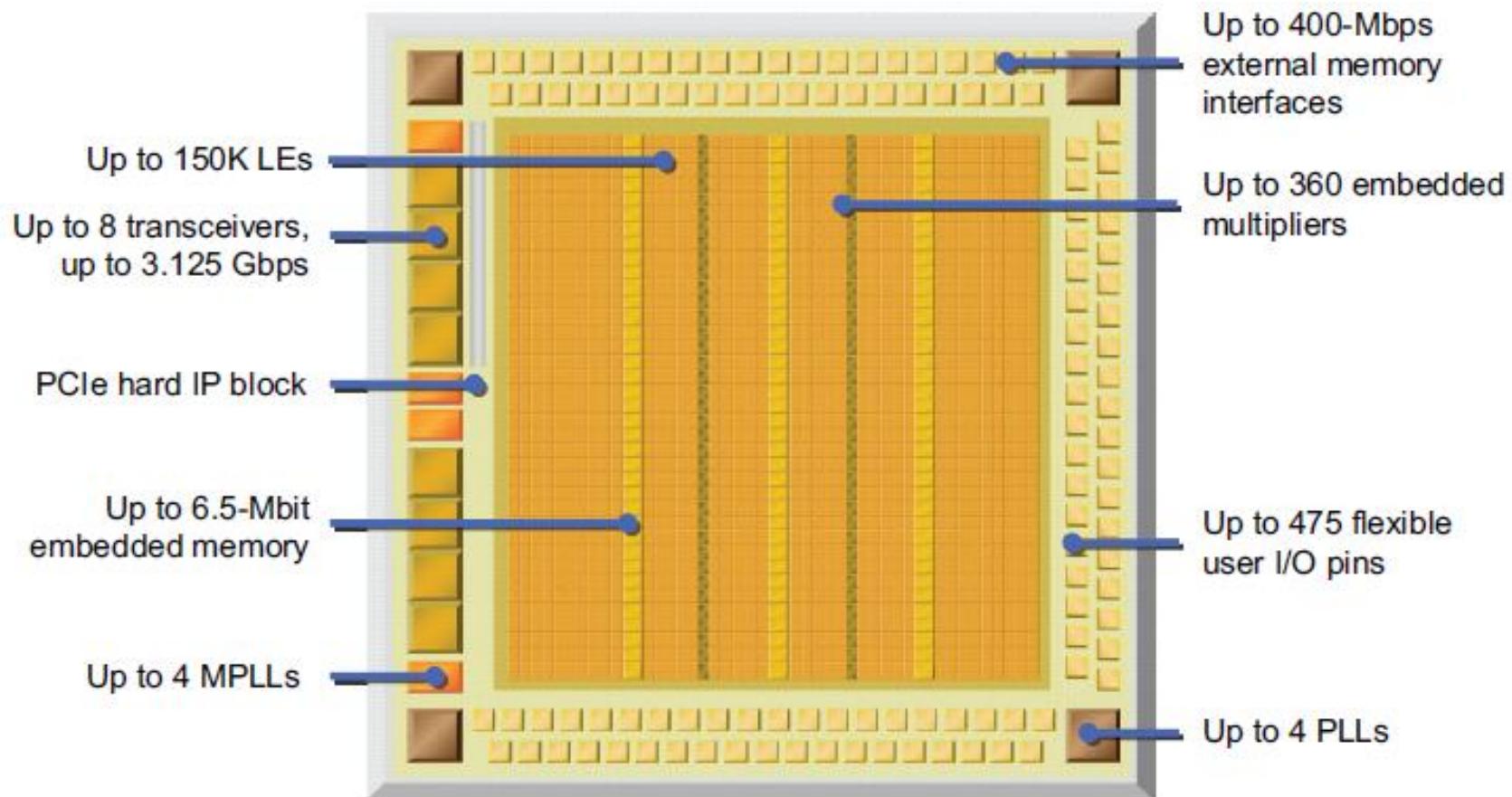


# Cyclone IV series - features

Device	Logic elements	Total memory (Kbits)	18x18 multipliers	Transceiver I/Os	PCI Express hard IP block	User I/Os
Cyclone IV E FPGAs (1.0V)	6,272-114,480	270-3,888	15-266	N/A	N/A	94-535
Cyclone IV GX FPGAs (1.2V)	14,400-149,760	50-6,480	0-360	2-8	1	72-475

- Two subseries available:
  - Simpler - Cyclone IV E
  - More complex - Cyclone IV GX

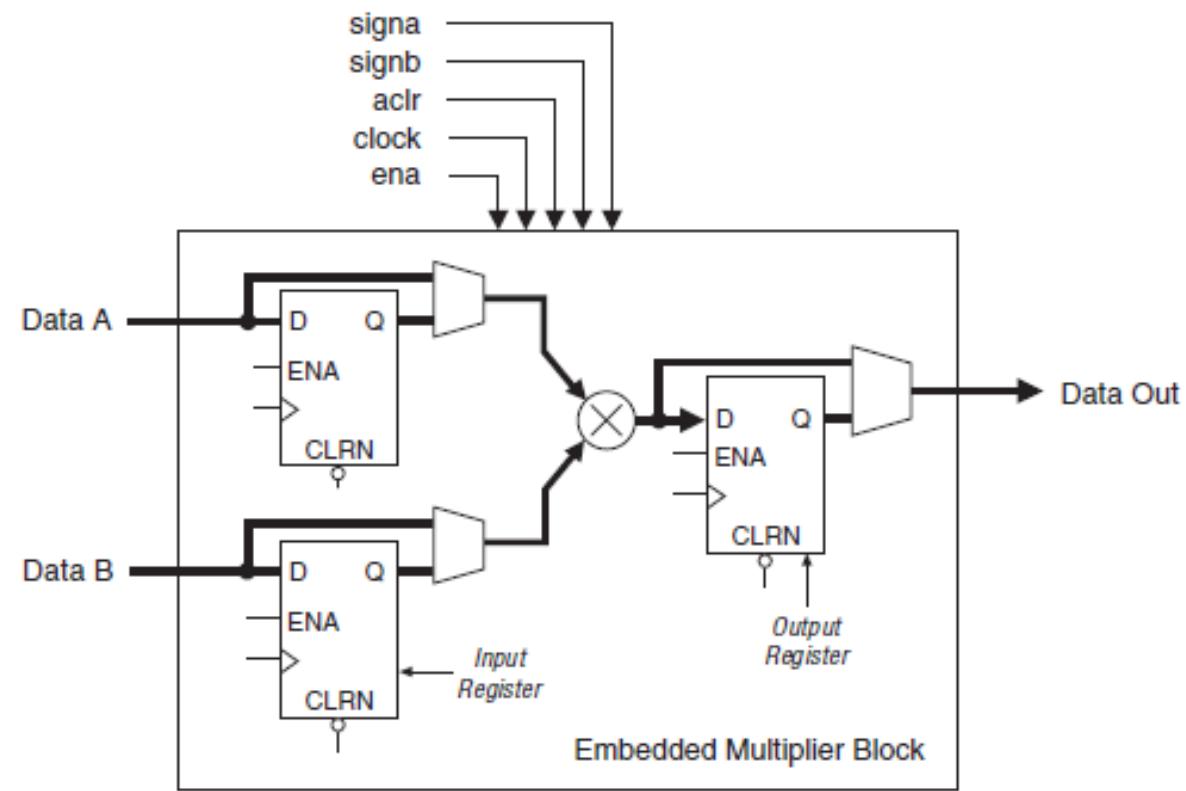
# Cyclone IV series - features



Source: [3]

# Cyclone IV series - features

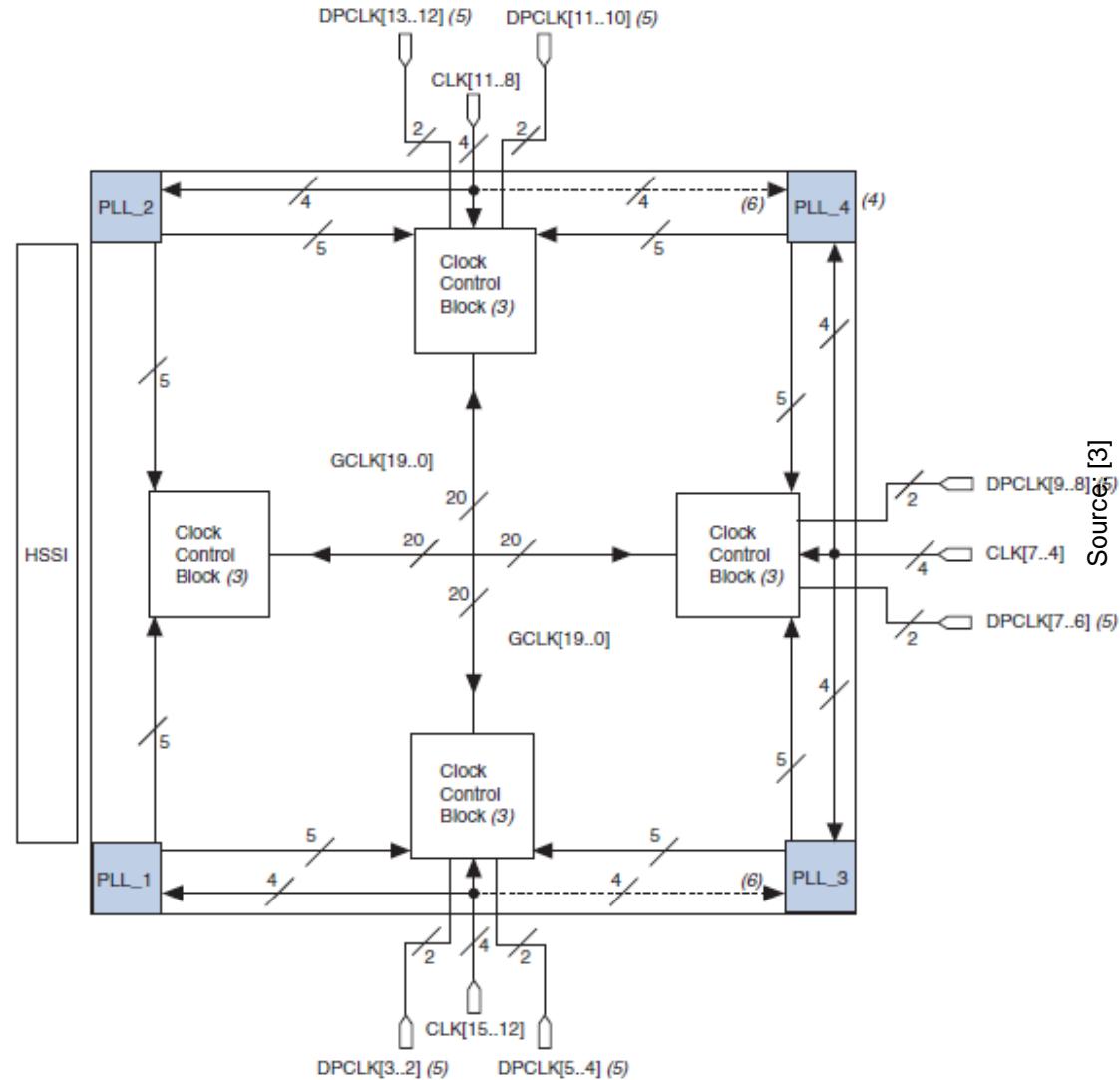
- Embedded multipliers:
  - Huge number of multipliers available (<360)!



Source: [3]

# Cyclone IV series - features

- Clock network



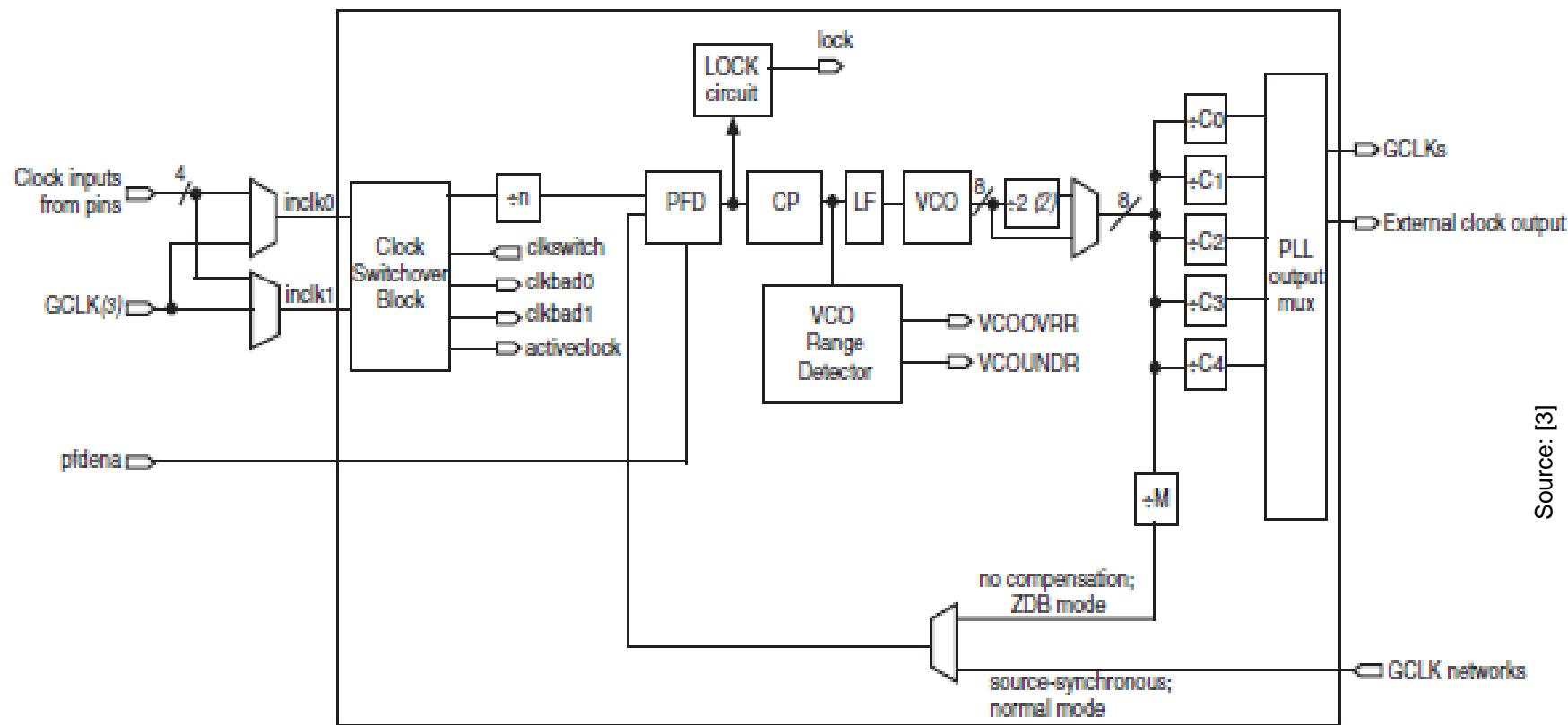
# Cyclone IV series - features

- PLL features

Hardware Features	Availability
C (output counters)	5
M, N, C counter sizes	1 to 512 (1)
Dedicated clock outputs	1 single-ended or 1 differential pair
Clock input pins	4 single-ended or 2 differential pairs
Spread-spectrum input clock tracking	✓ (2)
PLL cascading	Through GCLK
Compensation modes	Source-Synchronous Mode, No Compensation Mode, Normal Mode, and Zero Delay Buffer Mode
Phase shift resolution	Down to 96-ps increments (3)
Programmable duty cycle	✓
Output counter cascading	✓
Input clock switchover	✓
User mode reconfiguration	✓
Loss of lock detection	✓

# Cyclone IV series - features

- PLL features

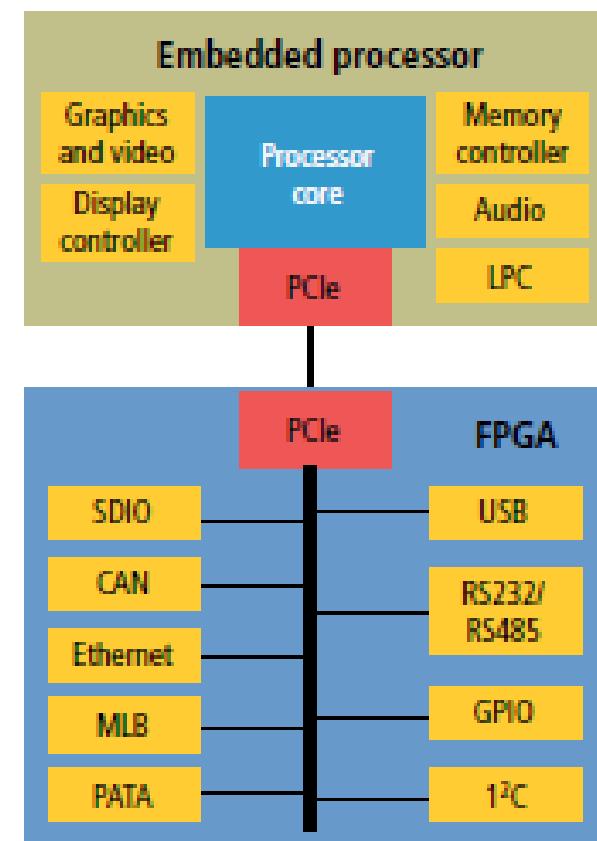


Source: [3]

# Cyclone IV series - features

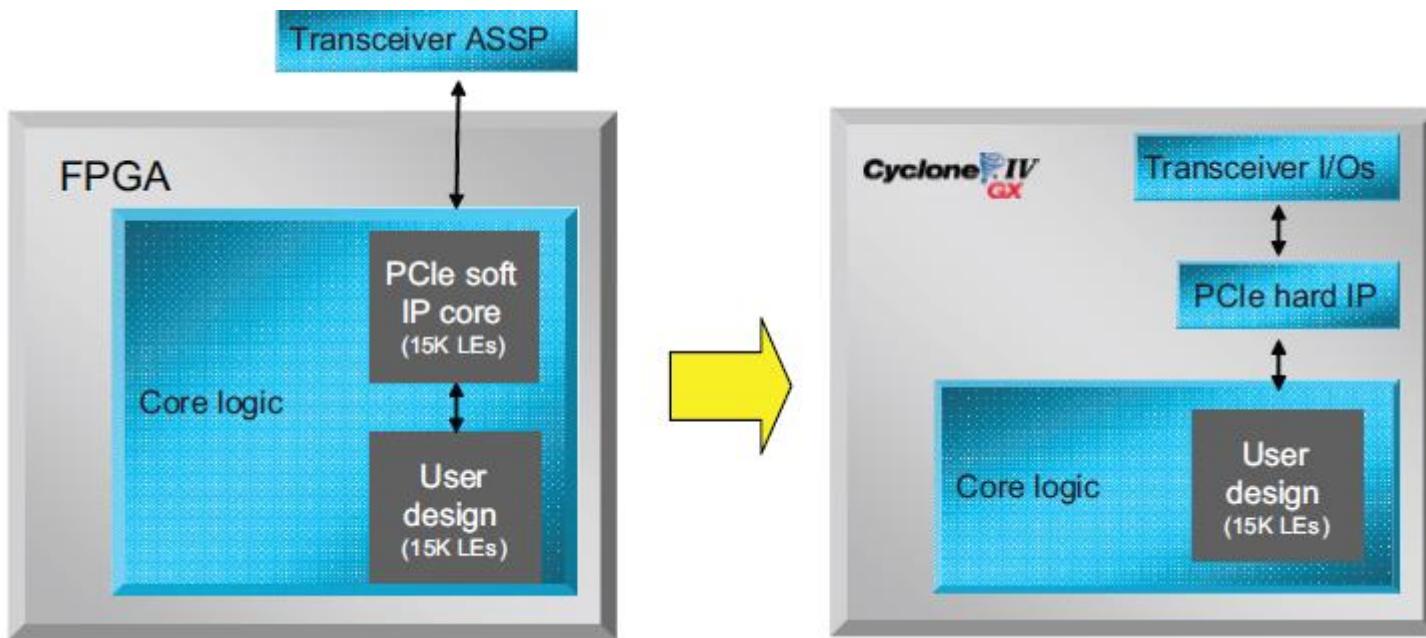
- Available I/O protocols

Protocol	Data rate (Gbps)
Basic (proprietary)	2.5 – 3.125
CPRI	3.072
DisplayPort	2.7
Gigabit Ethernet	1.25
PCI Express Gen1.1	2.5
SATA	3.0
Serial RapidIO®	3.125
V-by-One	3.0
XAU	3.125
3G SDI	2.97



# Cyclone IV series - features

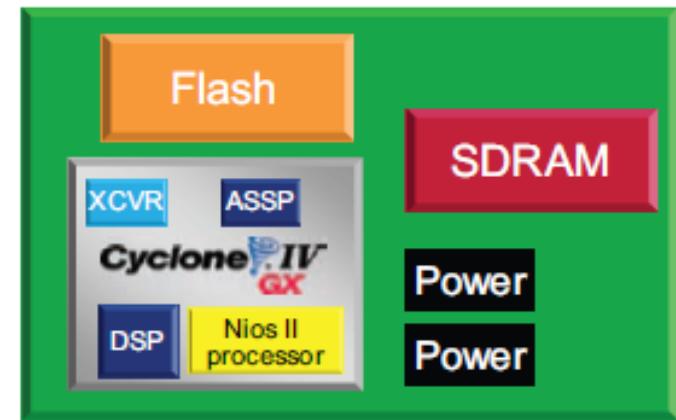
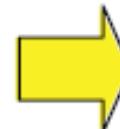
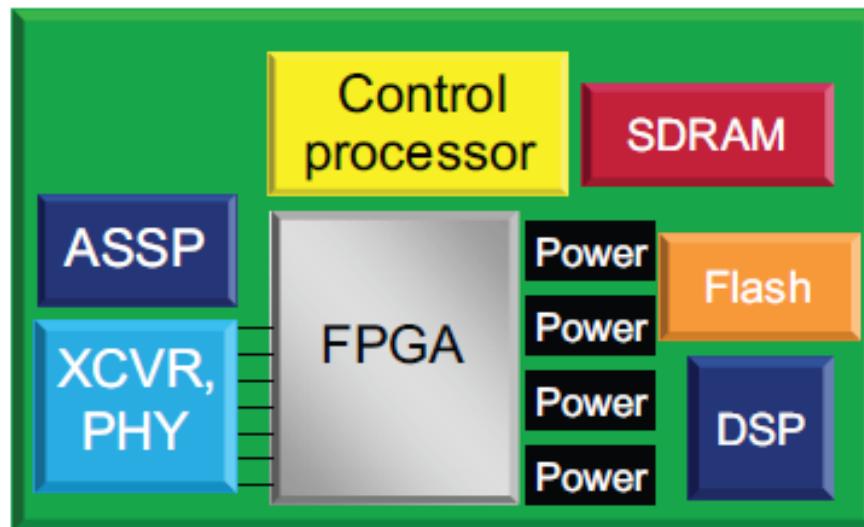
- PCIe Hard IP:
  - x1, x2, x4 lane support
  - Endpoint and rootport functionality



Source: [3]

# Cyclone IV series - features

- Embedded system integration:
  - 18x18 b multipliers as DSP can be used or
  - 32b Altera NIOS-II Soft Core



Source: [3]





# Cyclone V series - features

- Low cost FPGA
- Six versions:
  - Cyclone V E with logic only
  - Cyclone V GX FPGAs with 3.125-Gbps transceivers
  - Cyclone V GT FPGAs with 5-Gbps transceivers
  - Cyclone V SE SoC FPGA with ARM-based hard processor system (HPS) and logic
  - Cyclone V SX SoC FPGA with ARM-based HPS and 3.125-Gbps transceivers
  - Cyclone V ST SoC FPGA with ARM-based HPS and 5-Gbps transceivers



# Cyclone V - E family

Device	5CEA2	5CEA4	5CEA5	5CEA7	5CEA9
<b>Logic elements (LEs)</b>	25,000	48,000	76,500	149,500	301,000
<b>M10K memory blocks</b>	170	270	380	650	1,160
<b>M10K memory (Kb)</b>	1,700	2,700	3,800	6,500	11,600
<b>Memory logic array blocks (MLABs) (Kb)</b>	196	270	440	836	1,717
<b>18-bit x 19-bit multipliers</b>	50	144	248	312	684
<b>Variable-precision digital signal processing (DSP) blocks <a href="#">(1)</a></b>	25	72	124	156	342
<b>Phase-locked loops (PLLs)</b>	4	4	6	6	6
<b>Maximum user I/Os</b>	304	304	360	488	488
<b>Memory controllers</b>	1	1	2	2	2

# Cyclone V - GX family

Device	5CGXC3	5CGXC4	5CGXC5	5CGXC7	5CGXC9
<b>LEs</b>	31,000	50,000	76,500	149,500	301,000
<b>M10K memory blocks</b>	140	250	380	650	1,160
<b>M10K memory (Kb)</b>	1,400	2,500	3,800	6,500	11,600
<b>MLABs (Kb)</b>	188	295	440	836	1,717
<b>18-bit x 19-bit multipliers</b>	84	140	248	312	684
<b>Variable-precision DSP blocks</b>	42	70	124	156	342
<b>PCI Express® hard intellectual property (IP) block</b>	1	2	2	2	2
<b>PLLs</b>	4	6	6	7	8
<b>Maximum user I/Os</b>	224	368	368	480	560
<b>Memory controllers</b>	1	2	2	2	2



# Cyclone V - GT family

Device	5CGTD5	5CGTD7	5CGTD9
LEs	76,500	149,500	301,000
M10K memory blocks	380	650	1,160
M10K memory (Kb)	3,800	6,500	11,600
MLABs (Kb)	440	836	1,717
18-bit x 19-bit multipliers	248	312	684
Variable-precision DSP blocks	124	156	342
PCI Express hard IP block	2	2	2
PLLs	6	7	8
Maximum user I/Os	368	480	560
Memory controllers	2	2	2

# Cyclone V - SE SoC family

Device	5CSEA2	5CSEA4	5CSEA5	5CSEA6
<b>LEs</b>	25,000	40,000	85,000	110,000
<b>Adaptive logic modules (ALMs)</b>	9,434	15,094	32,075	41,509
<b>M10K memory blocks</b>	140	224	397	514
<b>M10K memory (Kb)</b>	1,400	2,240	3,972	5,140
<b>MLABs (Kb)</b>	138	220	480	621
<b>18-bit x 19-bit multipliers</b>	72	116	174	224
<b>Variable-precision DSP blocks (1)</b>	36	58	87	112
<b>FPGA PLLs</b>	4	5	6	6
<b>HPS PLLs</b>	3	3	3	3
<b>Maximum FPGA user I/Os</b>	124	124	288	288
<b>Maximum HPS I/Os</b>	188	188	188	188
<b>FPGA hard memory controllers</b>	-	1	1	1
<b>HPS hard memory controllers</b>	1	1	1	1
<b>Processor cores (ARM® Cortex™-A9)</b>	Single or dual	Single or dual	Single or dual	Single or dual



# Cyclone V - SX SoC family

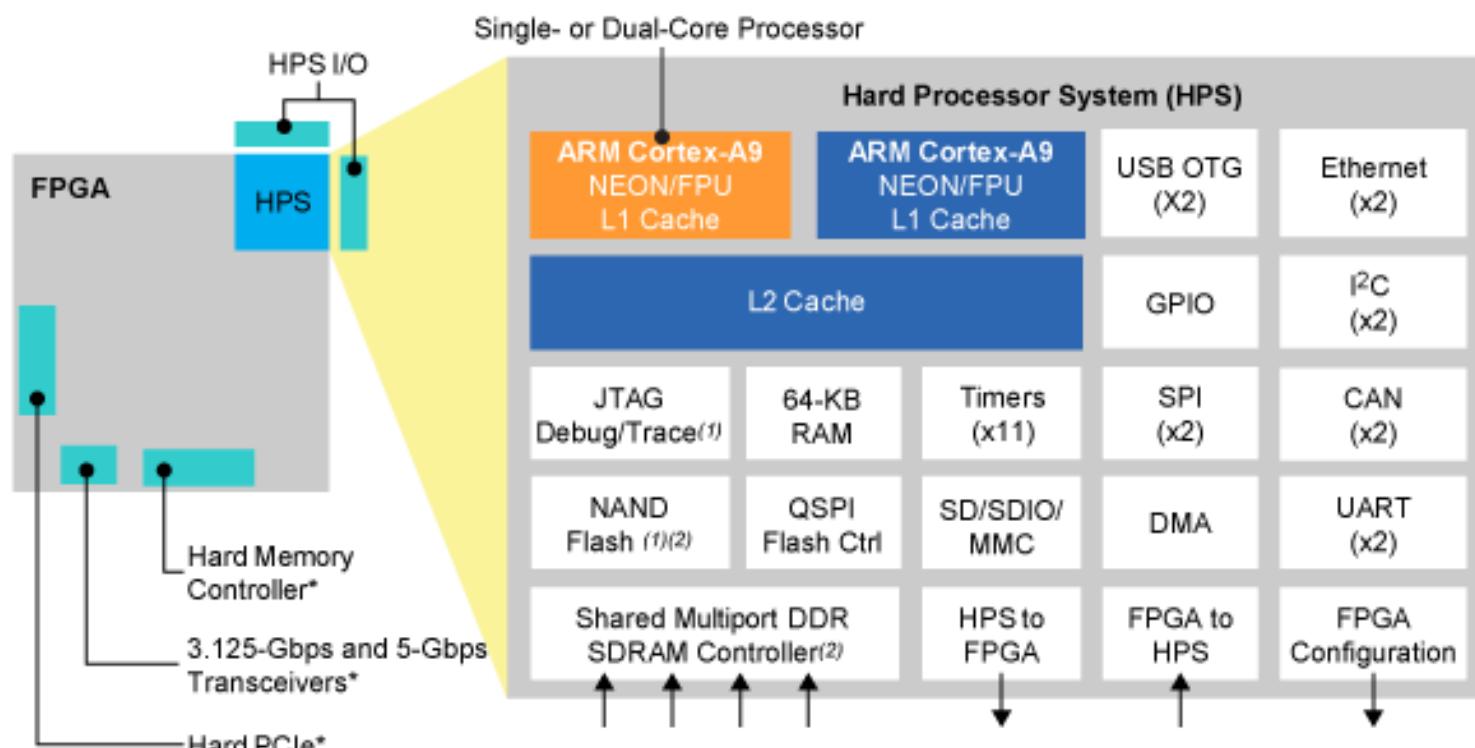
Device	5CSXC4	5CSXC5	5CSXC6
LEs	40,000	85,000	110,000
ALMs	15,094	32,075	41,509
M10K memory blocks	224	397	514
M10K memory (Kb)	2,240	3,972	5,140
MLABs (Kb)	220	480	621
18-bit x 19-bit multipliers	116	174	224
Variable-precision DSP blocks	58	87	112
Maximum transceivers	6	9	9
PCI Express hard IP block	2	2	2
FPGA PLLs	5	6	6
HPS PLLs	3	3	3
Maximum FPGA user I/Os	124	288	288
Maximum HPS I/Os	188	188	188
FPGA hard memory controllers	1	1	1
HPS hard memory controllers	1	1	1
Processor cores (ARM Cortex-A9)	Dual	Dual	Dual



# Cyclone V - ST SoC family

Device	5CSTD5	5CSTD6
LEs	85,000	110,000
ALMs	32,075	41,509
M10K memory blocks	397	514
M10K memory (Kb)	3,972	5,140
MLAB (Kb)	480	621
18-bit x 19-bit multipliers	174	224
Variable-precision DSP blocks	87	112
Maximum transceivers	9	9
PCI Express hard IP block	2	2
FPGA PLLs	6	6
HPS PLLs	3	3
Maximum FPGA user I/Os	288	288
Maximum HPS I/Os	188	188
FPGA hard memory controllers	1	1
HPS hard memory controllers	1	1
Processor cores (ARM Cortex-A9)	Dual	Dual

# Cyclone V - Hard Processor System



Source: [3]



# Cyclone V- Hard Processor System

- Features:
  - 800-MHz, dual-core ARM® Cortex™-A9 MPCore™ processor
  - Each processor core includes:
    - 32 KB of L1 instruction cache, 32 KB of L1 data cache
    - Single- and double-precision floating-point unit and NEON™ media engine
    - CoreSight™ debug and trace technology
  - 512 KB of shared L2 cache
  - 64 KB of scratch RAM
  - Multiport SDRAM controller with support for DDR2, DDR3, LPDDR1, and LPDDR2
  - 8-channel direct memory access (DMA) controller



# Cyclone V- Hard Processor System

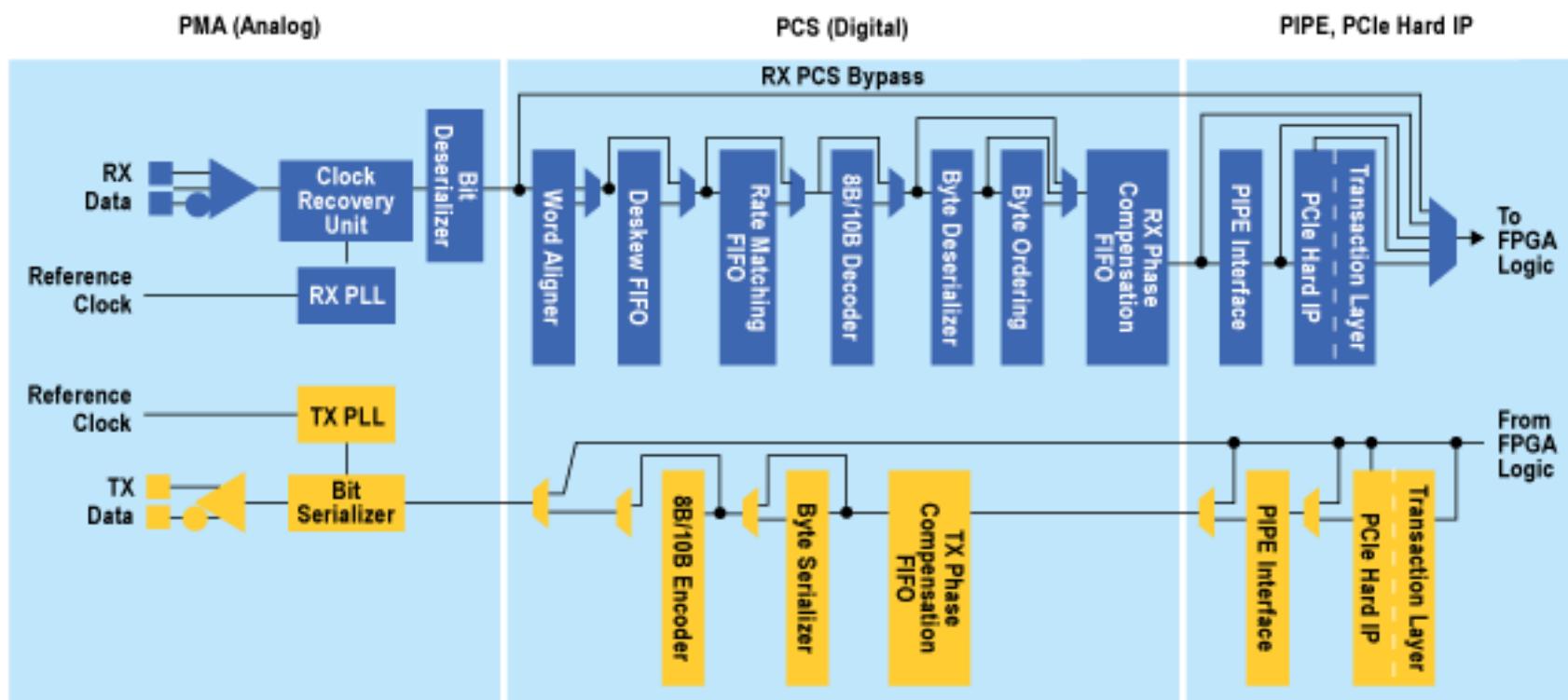
- Features:
  - QSPI flash controller
  - NAND flash controller with DMA
  - SD/SDIO/MMC controller with DMA
  - 2x 10/100/1000 Ethernet media access control (MAC) with DMA
  - 2x USB On-The-Go (OTG) controller with DMA
  - 2x I2C controller
  - 2x UART
  - 2x serial peripheral interface (SPI)
  - Up to 134 general-purpose I/O (GPIO)
  - 7x general-purpose timers
  - 4x watchdog timers

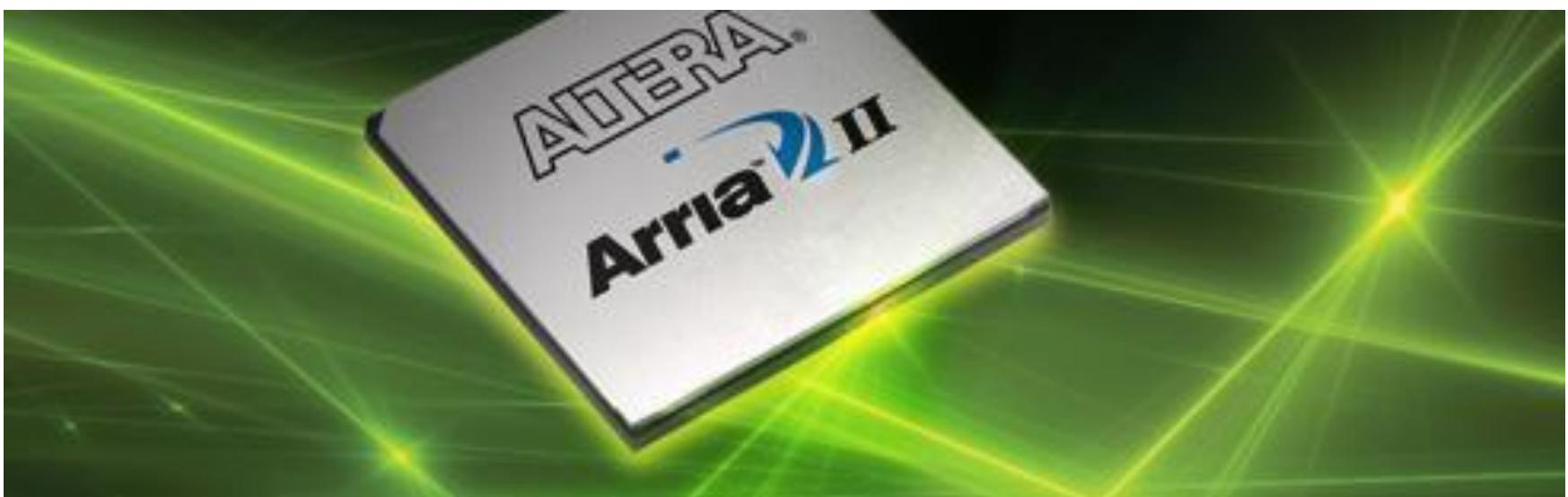


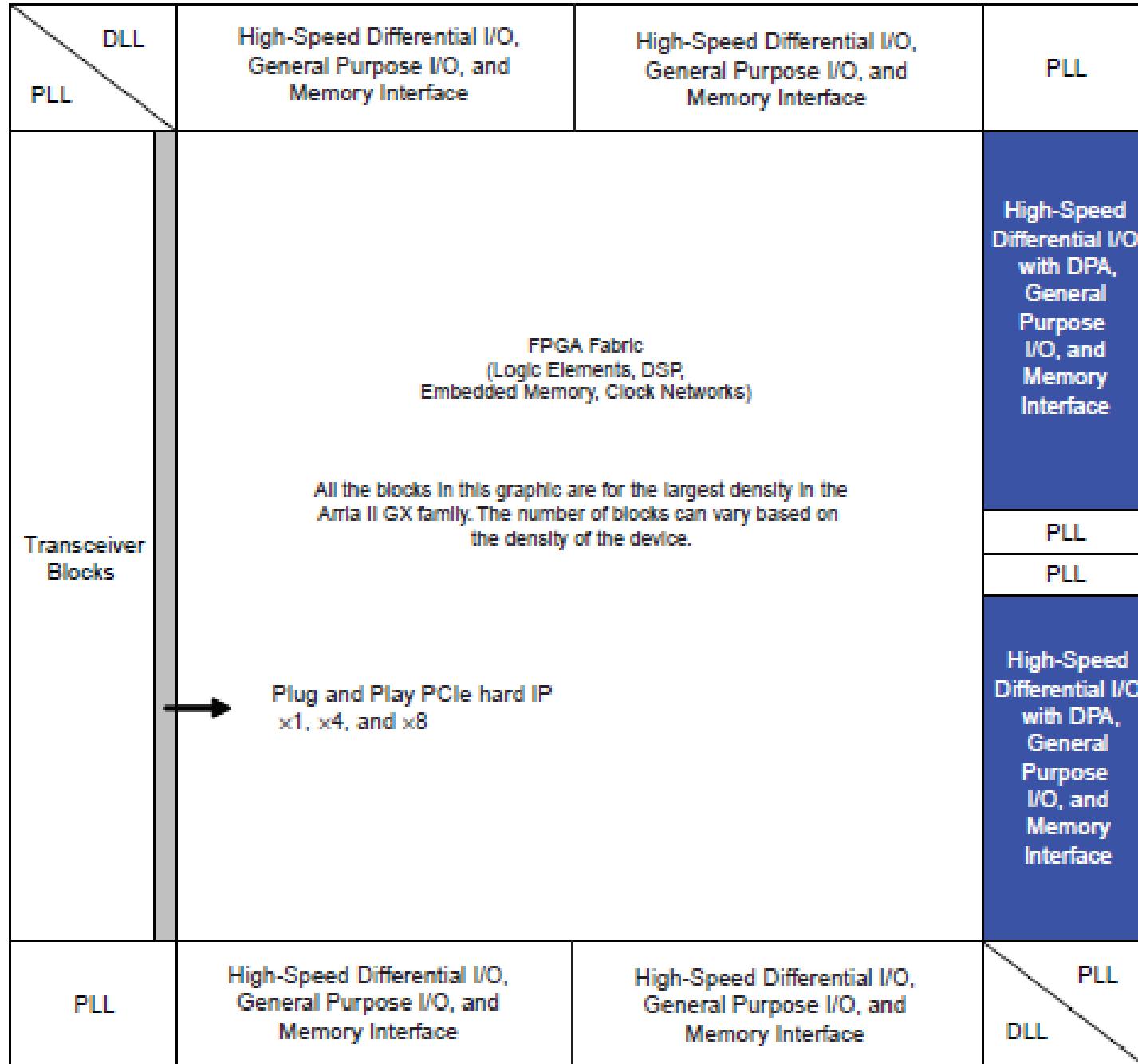
# Cyclone V- Transceivers

- Main features:
  - Up to twelve transceivers supporting data rates from 600 Mbps to 3.75 Gbps or 5.0 Gbps
  - Flexible and easy-to-configure transceiver datapath to implement industry-standard and proprietary protocols
  - Dynamic reconfiguration of the transceiver to support multiple protocols and data rates on the same channel without reprogramming the FPGA
  - Support for protocol features such as spread-spectrum clocking in PCI Express® (PCIe®) DisplayPort, V-by-One, and SATA configurations
  - 8B/10B encoder and decoder that performs 8-bit to 10-bit encoding and 10-bit to 8-bit decoding
  - Dedicated circuitry compliant with the physical interface for PCIe, XAUI, and Gbps Ethernet (GbE)

# Cyclone V- Transceivers









# Aria II GZ series - features

Features	EP2AGZ225	EP2AGZ300	EP2AGZ350
<b>Adaptive Logic Modules (ALMs)</b>	89,600	119,200	139,400
<b>Equivalent LEs</b>	224,000	298,000	348,500
<b>M9K Memory Blocks/Mb</b>	1,235/11.1	1,248/11.2	1,248/11.2
<b>M144K Memory Blocks / Mb</b>	-	24 / 3.4	36 / 5.2
<b>Total Memory (M9K+M144K in Mb)</b>	11.1	14.6	16.4
<b>18-Bit x 18-Bit Embedded Multipliers</b>	800	920	1,040
<b>Maximum Transceivers</b>	24	24	24
<b>PCI Express Hard IP Blocks</b>	1	1	1
<b>Maximum User I/O Pins</b>	726	726	726



# Aria II GX series - features

Features	EP2AGX45	EP2AGX65	EP2AGX95	EP2AGX125	EP2AGX190	EP2AGX260
<b>Adaptive Logic Modules (ALMs)</b>	18,050	25,300	37,470	49,640	76,120	102,600
<b>Equivalent LEs</b>	45,125	63,250	93,674	124,100	190,300	256,500
<b>M9K Memory Blocks/Mb</b>	319/2.9	495/4.5	612/5.5	730/6.6	840/7.6	950/8.5
<b>Total Memory (M9K + MLAB in Mb)</b>	3.4	5.3	6.7	8.1	9.9	11.8
<b>18-Bit x 18-Bit Embedded Multipliers</b>	232	312	448	576	656	736
<b>PLLs</b>	4	4	6	6	6	6
<b>Maximum Transceivers</b>	8	8	12	12	16	16
<b>PCI Express Hard IP Blocks</b>	1	1	1	1	1	1
<b>Maximum User I/O Pins</b>	364	364	452	452	612	612



# Aria II series - features

- 40-nm, low-power FPGA engine
  - Adaptive logic module (ALM) offers the highest logic efficiency
  - Eight-input look-up tables (LUT)
  - Memory logic array blocks (MLABs) for efficient implementation of small FIFOs



## Aria II series - features

- High-performance digital signal processing (DSP) blocks up to 380 MHz
  - Configurable as  $9 \times 9$ -bit,  $12 \times 12$ -bit,  $18 \times 18$ -bit, and  $36 \times 36$ -bit full-precision multipliers as well as  $18 \times 36$ -bit high-precision multiplier
  - Hardcoded adders, subtractors, accumulators, and summation functions
  - Fully-integrated design flow with the MATLAB and DSP Builder software from Altera



## Aria II series - features

- Maximum system bandwidth
  - Up to 16 full-duplex clock data recovery (CDR)-based transceivers supporting rates between 155 Mbps and 6.375 Gbps
  - Dedicated circuitry to support physical layer functionality for popular serial protocols, including PCIe Gen1, Gbps Ethernet, Serial RapidIO® (SRIIO), Common Public Radio Interface (CPRI), OBSAI, SD/HD/3G/ASI Serial Digital Interface (SDI), XAUI, HiGig/HiGig+, SATA/Serial Attached SCSI (SAS), GPON, SerialLite II, Fiber Channel, and SONET/SDH



## Aria II series - features

- Advanced usability and security features
  - Parallel and serial configuration options
  - On-chip series termination (RS OCT) and differential I/O termination
  - 256-bit advanced encryption standard (AES) programming file encryption for design security with volatile and non-volatile key storage options
  - Robust portfolio of IP for processing, serial protocols, and memory interfaces
  - Low cost, easy-to-use





# Arria V family

- Medium size FPGA suitable for various advanced applications like:
  - Power sensitive wireless infrastructure equipment
  - 20G/40G bridging, switching, and packet processing applications
  - High-definition video processing and image manipulation
  - Intensive digital signal processing (DSP) applications



# Arria V family

- Family variants:
  - Arria V GX—integrated 6-Gbps transceivers:
    - optimized for high-volume data and signal-processing applications
  - Arria V GT—integrated 10-Gbps transceivers:
    - high-speed serial I/O bandwidth for cost-sensitive data and signal processing applications
  - Arria V SX—system-on-a-chip (SoC) FPGA with integrated ARM®-based hard processor system (HPS)
  - Arria V ST—SoC FPGA with integrated ARM-based HPS, and 10-Gbps transceivers



# Arria V family

- Features:
  - 28-nm process technology
  - 1.1-V core nominal voltage
  - 611-Mbps to 10.3125-Gbps integrated transceivers
  - Transmit pre-emphasis and receiver equalization
  - 1.25-Gbps LVDS
  - 667-MHz/1.333-Gbps external memory interface
  - On-chip termination (OCT)



# Arria V family

- Features:
  - PCI Express® (PCIe®) Gen1 and Gen2
  - Gbps Ethernet (GbE) and XAUI physical coding sublayer (PCS)
  - Gigabit-capable passive optical network (GPON) PCS
  - Dual-core ARM Cortex-A9 MPCore processor.  
Up to 800 MHz maximum frequency that supports symmetric and asymmetric multiprocessing



# Arria V family

- Features:
  - Interface peripherals—10/100/1000 Ethernet media access control (MAC), USB 2.0 On-The-Go (OTG) controller, Quad SPI flash controller, NAND flash controller, and SD/MMC/SDIO controller, UART, serial peripheral interface (SPI), I2C interfaces, and up to 86 GPIO interfaces
  - System peripherals—general-purpose and watchdog timers, direct memory access (DMA) controller, FPGA configuration manager, and clock and reset managers



# Arria V family

- Features:
  - 10GBASE-R
  - 9.8304-Gbps CPRI
  - Natively supports three-signal processing precision ranging from  $9 \times 9$ ,  $18 \times 19$ , or  $27 \times 27$  in the same DSP block
  - 64-bit accumulator and cascade for systolic finite impulse responses (FIRs)
  - Embedded internal coefficient memory



# Arria V family

- Features:
  - Hardened double data rate3 (DDR3) and DDR2 memory controllers
  - Integer mode and fractional mode PLL
  - 625-MHz global clock network
  - Global, quadrant, and peripheral clock networks
  - Partial and dynamic reconfigurations
  - Configuration via HPS
  - Serial and parallel flash interface

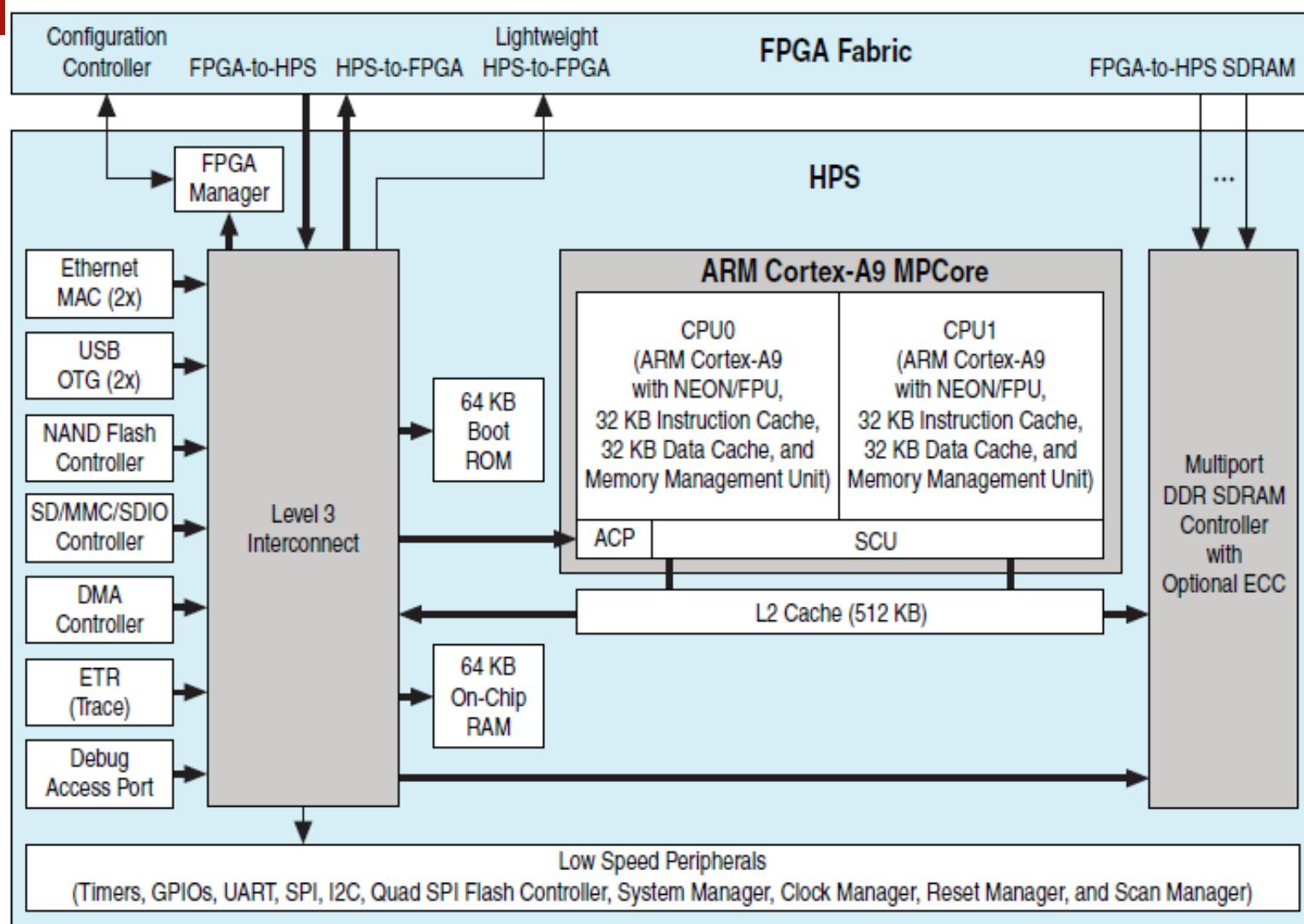
# Arria V family - GX devices

Feature	Arria V GX Device							
	5AGXA1	5AGXA3	5AGXA5	5AGXA7	5AGXB1	5AGXB3	5AGXB5	5AGXB7
ALMs	28,302	56,100	71,698	91,680	113,208	136,880	158,491	190,240
LE (K)	75	148.67	190	242.95	300	362.73	420	504.14
M10K memory blocks	800	1,051	1,180	1,366	1,510	1,726	2,054	2,414
MLAB memory (Kbit)	463	873	1,173	1,448	1,852	2,098	2,532	2,906
Block memory (KByte)	8,000	10,510	11,800	13,660	15,100	17,260	20,540	24,140
Variable-precision DSP blocks	240	396	600	800	920	1,045	1,092	1,156
18 x 19 multipliers	480	792	1,200	1,600	1,840	2,090	2,184	2,312
Fractional PLLs <sup>(1)</sup>	10	10	12	12	12	12	16	16
GPIO	480	480	544	544	704	704	704	704
LVDS transmitter (TX) <sup>(2)</sup>	68	68	120	120	160	160	156	160
LVDS receiver (RX) <sup>(2)</sup>	80	80	136	136	176	176	172	176
PCIe hard IP blocks	1	1	2	2	2	2	2	2
Hard memory controllers	2	2	4	4	4	4	4	4

# Arria V family - GT, SX and ST devices

Feature	Arria V GT Device		Arria V SX Device		Arria V ST Device	
	5AGTD3	5AGTD7	5ASXB3	5ASXB5	5ASTD3	5ASTD5
ALMs	136,880	190,240	132,075	174,340	132,075	174,340
LE (K)	362.73	504.14	350	462	350	462
M10K memory blocks	1,726	2,414	1,729	2,282	1,729	2,282
MLAB memory (Kb)	2,098	2,906	2,014	2,658	2,014	2,658
Block memory (KB)	17,260	24,140	17,288	22,820	17,288	22,820
Variable-precision DSP blocks	1,045	1,156	809	1,068	809	1,068
18 x 19 multipliers	2,090	2,312	1,618	2,186	1,618	2,186
FPGA Fractional PLLs <sup>(2)</sup>	12	16	TBD	TBD	TBD	TBD
HPS PLLs <sup>(2)</sup>	—	—	TBD	TBD	TBD	TBD
FPGA GPIO	704	704	528	528	528	528
HPS I/O	—	—	216	216	216	216
LVDS TX <sup>(1)</sup>	160	160	120	120	120	120
LVDS RX <sup>(1)</sup>	176	176	120	120	120	120
PCIe hard IP blocks	2	2	2	2	2	2
Hard memory controllers	4	4	3	3	3	3
HPS memory controllers	—	—	1	1	1	1
ARM Cortex-A9 MPCore processor	—	—	Dual-core	Dual-core	Dual-core	Dual-core

# Arria V family - HPS block





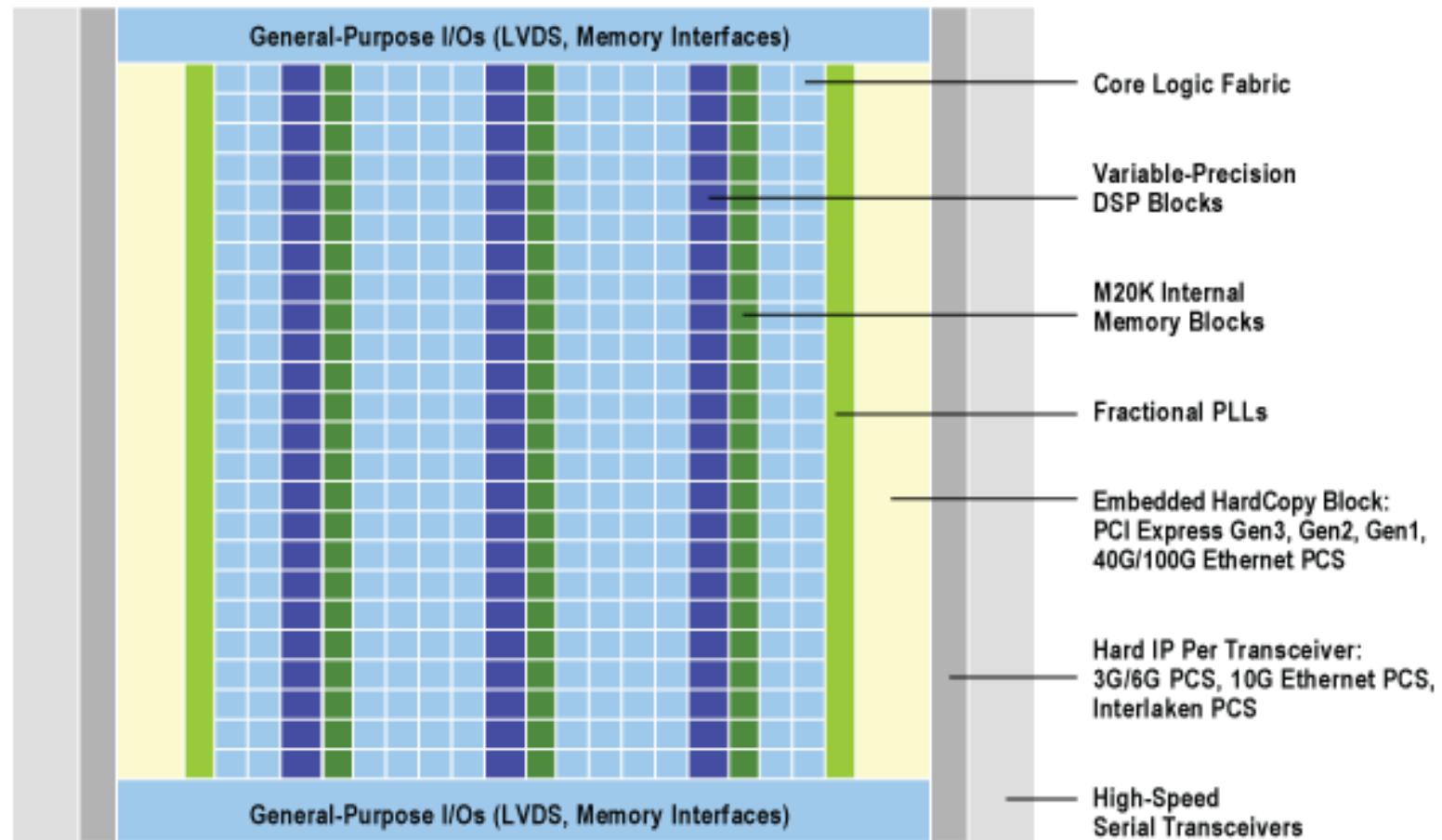


# Stratix V family

- Family Variants:

- **Stratix V GT** - Optimized for applications with 28G transceivers requiring ultra-high bandwidth and performance, such as 40G/100G/400G applications
- **Stratix V GX** - Optimized for high-performance, high-bandwidth applications with integrated transceivers supporting backplane, chip-to-chip, and chip-to-module operation at up to 14.1 Gbps
- **Stratix V GS** - Optimized for high-performance, variable-precision digital signal processing (DSP) applications with integrated transceivers supporting backplane, chip-to-chip, and chip-to-module operation at up to 14.1 Gbps
- **Stratix V E** - Optimized for ASIC prototyping with over 1 million logic elements on the highest performance logic fabric

# Stratix V series - architecture



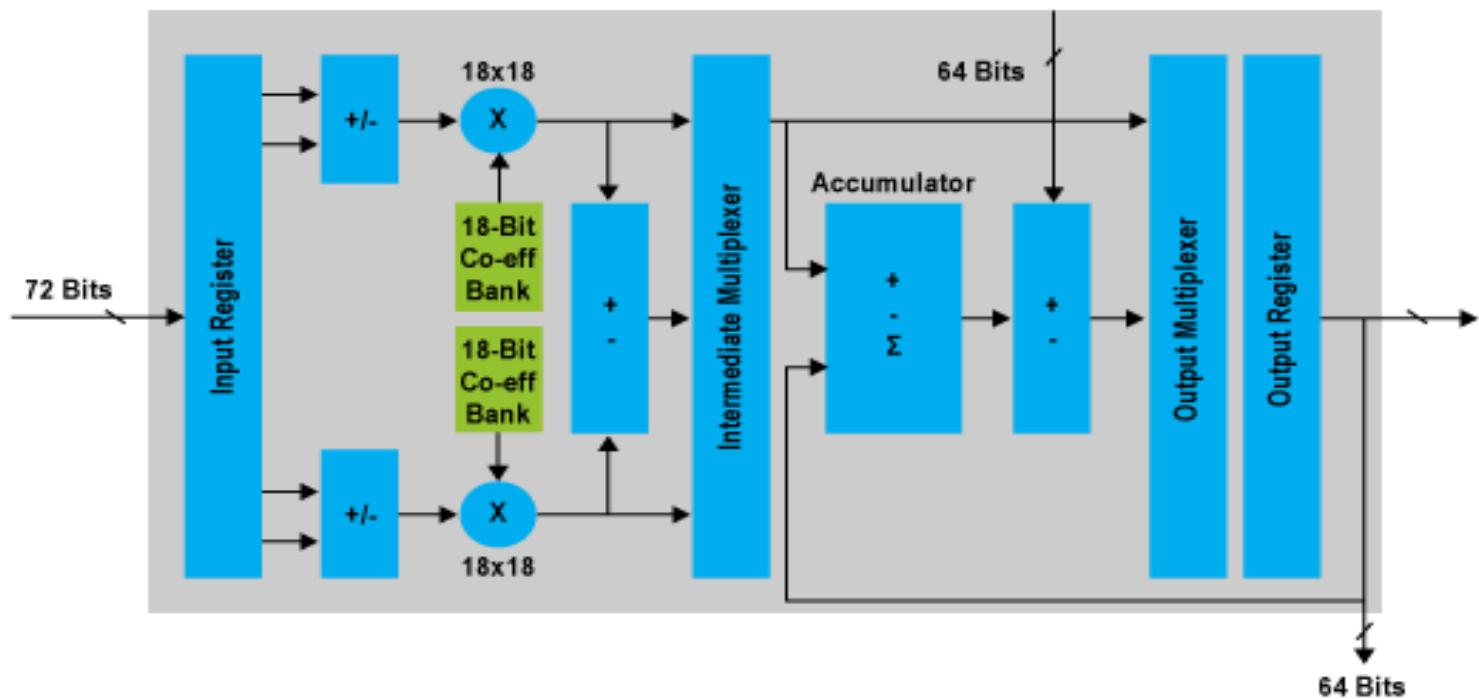
Source: [5]

# Stratix V series - architecture

Feature	Stratix V E	Stratix V GS	Stratix V GX	Stratix V GT
<a href="#"><u>High-Performance Adaptive Logic Module</u></a>	✓	✓	✓	✓
<a href="#"><u>Variable-Precision DSP Blocks (18x18)</u></a>	1,100	3,510	612	512
<a href="#"><u>M20K Memory Blocks</u></a>	2,100	1,755	2,560	2,560
<a href="#"><u>External Memory Interface</u></a>	✓	✓	✓	✓
<a href="#"><u>Partial Reconfiguration</u></a>	✓	✓	✓	✓
<a href="#"><u>fPLL</u></a>	✓	✓	✓	✓
<a href="#"><u>Design Security</u></a>	✓	✓	✓	✓
<a href="#"><u>SEU Mitigation</u></a>	✓	✓	✓	✓
<a href="#"><u>PCIe Gen 3, Gen2, Gen1 Hard IP blocks</u></a>	-	Up to 2	Up to 4	1
<a href="#"><u>Embedded HardCopy Blocks and Hard IP</u></a>	-	✓	✓	✓
<a href="#"><u>Transceivers (1)</u></a>	-	12.5 Gbps / 27	12.5 Gbps / 66	28 Gbps / 4 12.5 Gbps / 32

# Stratix V series - features

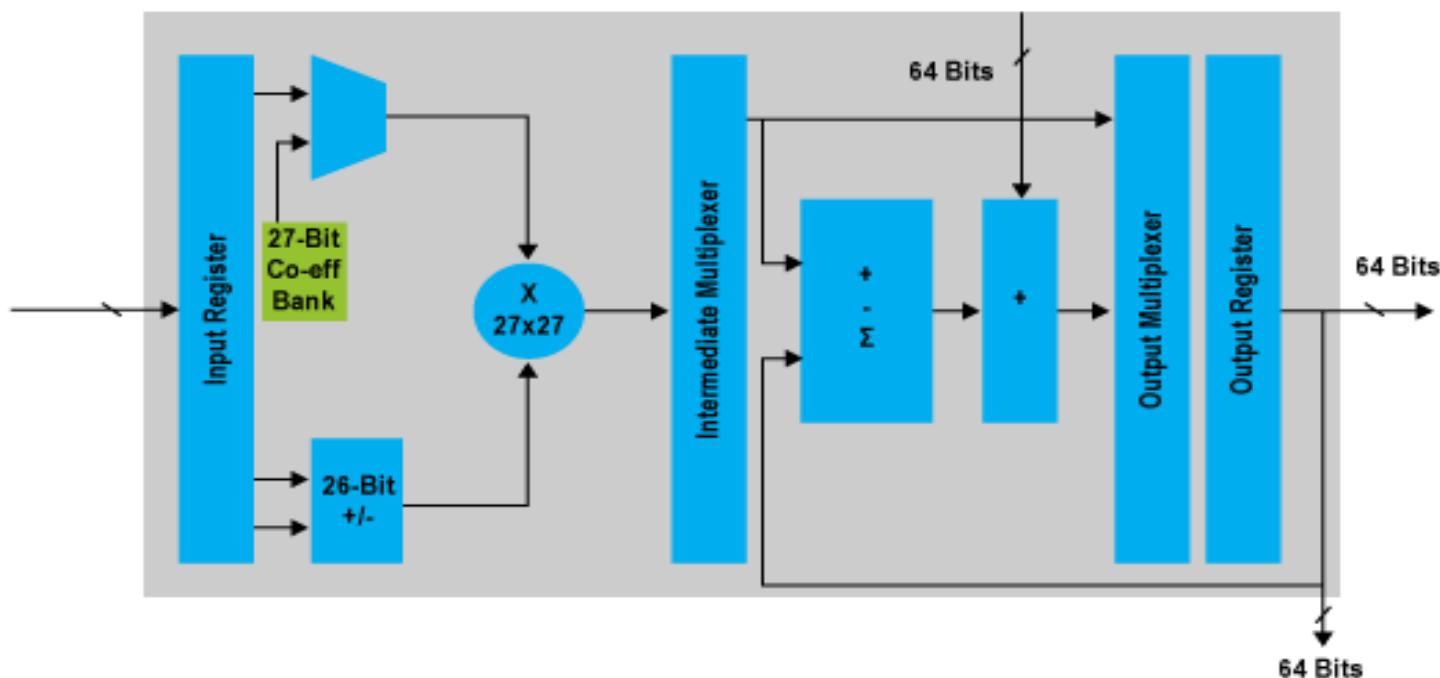
- Variable-Precision DSP Block Architecture  
18-Bit Precision Mode



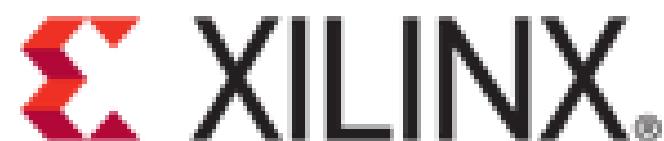
Source: [5]

# Stratix V series - features

- Variable-Precision DSP Block Architecture  
High-Precision Mode



Source: [5]



Xilinx family



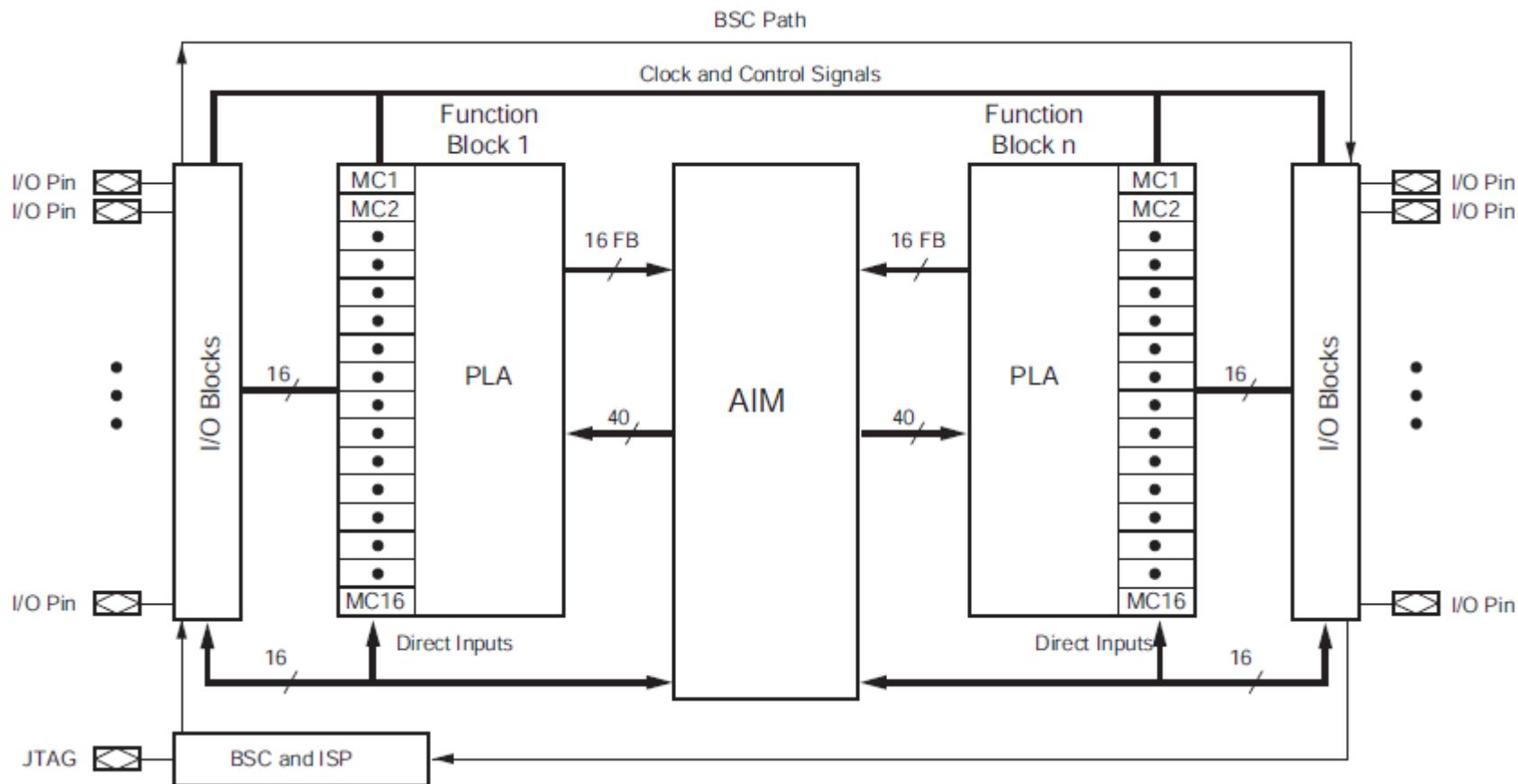
# Xilinx family

- Xilinx family of FPD consists of:
  - CPLD - CoolRunner II series
  - FPGA:
    - EasyPath-6 (low cost)
    - Spartan-6 (low end)
    - Virtex-6 (high end)



# CoolRunner II series

# CoolRunner II series - architecture





# CoolRunner II series - features

- Multiple I/O banking (2-4):
  - Including different bus interface I/O voltage levels
  - Voltage translation of peripheral devices
  - Memory to microcontrollers
  - Communication between wired interfaces

# CoolRunner II series - features

- Input hysteresis (500mV):
  - Improved noise immunity
  - Reduced power consumption
  - Superior signal integrity

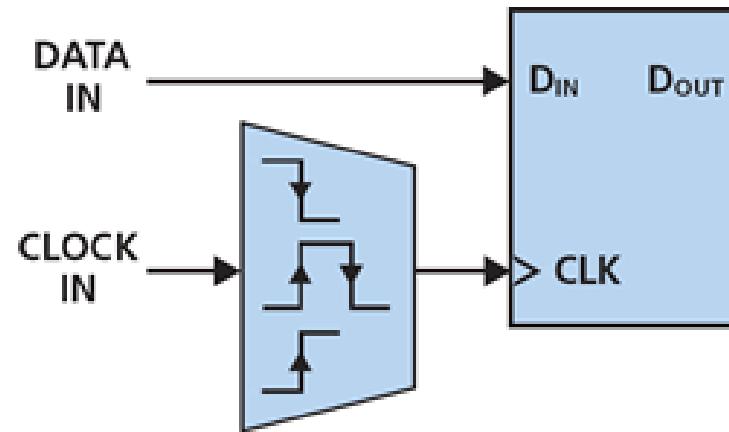
Source: [1]



# CoolRunner II series - features

- DualEDGE flipflops:
  - Distributes divided clock globally then double locally at macrocell
  - Use 2x clocking for double data rate applications
  - No additional insertion delay
  - Available in all CoolRunner-II CPLDs

Source: [1]







# EasyPath-6 series - features

- EasyPath-6™ FPGAs offer a fast, simple and risk-free solution for cost reducing Virtex-6 FPGA designs.
- EasyPath-6 FPGAs deliver production volume devices in just six weeks with the lowest total product cost of any FPGA cost reduction solution.
- The cost savings are also available free of additional design constraints, re-engineering requirements, or re-layout of boards.



# EasyPath-6 series - features

- EasyPath-6 FPGAs are architecturally equivalent to Virtex-6 FPGAs and match the FPGA datasheet specifications for functionality and timing.
- Unlike ASICs, the EasyPath-6 FPGA silicon requires no-requalification.
- Additionally, all Xilinx and third-party Intellectual Property (IP) for Virtex-6 FPGAs is supported by EasyPath-6 FPGAs without change or re-verification, and with no additional royalty, conversion or licensing fees.

# EasyPath-6 series - features

## TIME-TO-MARKET COMPARISON

### FPGA TO STRUCTURED ASIC<sup>1</sup>



### VIRTEX-6 TO EASYPATH-6 FPGA<sup>2</sup>



INTERCHANGEABLE  
IN PRODUCTION







# Spartan-6 series - features

- The Spartan®-6 family provides leading system integration capabilities with the lowest total cost for high-volume applications
- Spartan-6 is a lowcost programmable alternative to custom ASIC products with ease of use
- Good solution for high-volume logic designs, consumer-oriented DSP designs, and cost-sensitive embedded applications



# Spartan-6 series - features

- Spartan-6 Family:
  - Spartan-6 LX FPGA: Logic optimized
  - Spartan-6 LXT FPGA: High-speed serial connectivity
- Designed for low cost:
  - Multiple efficient integrated blocks
  - Optimized selection of I/O standards
  - Staggered pads
  - High-volume plastic wire-bonded packages



## Spartan-6 series - features

- Multi-voltage, multi-standard SelectIO™ interface banks
  - Up to 1,080 Mb/s data transfer rate per differential I/O
  - Selectable output drive, up to 24 mA per pin
  - 3.3V to 1.2V I/O standards and protocols
  - Low-cost HSTL and SSTL memory interfaces
  - Hot swap compliance
  - Adjustable I/O slew rates to improve signal integrity



# Spartan-6 series - features

- High-speed GTP serial transceivers in the LXT FPGAs
  - Up to 3.125 Gb/s
  - High-speed interfaces including: Serial ATA, Aurora,
  - 1G Ethernet, PCI Express, OBSAI, CPRI, EPON, GPON, DisplayPort, and XAUI
- Integrated Endpoint block for PCI Express designs (LXT)
- Low-cost PCI® technology support compatible with the 33 MHz, 32- and 64-bit specification.



# Spartan-6 series - features

- Efficient DSP48A1 slices
  - High-performance arithmetic and signal processing
  - Fast  $18 \times 18$  multiplier and 48-bit accumulator
  - Pipelining and cascading capability
  - Pre-adder to assist filter applications
- Integrated Memory Controller blocks
  - DDR, DDR2, DDR3, and LPDDR support
  - Data rates up to 800 Mb/s (12.8 Gb/s peak bandwidth)
  - Multi-port bus structure with independent FIFO to reduce design timing issues



# Spartan-6 series - features

- Abundant logic resources with increased logic capacity
  - Optional shift register or distributed RAM support
  - Efficient 6-input LUTs improve performance and minimize power
  - LUT with dual flip-flops for pipeline centric applications
- Block RAM with a wide range of granularity
  - Fast block RAM with byte write enable
  - 18 Kb blocks that can be optionally programmed as two independent 9 Kb block RAMs



# Spartan-6 series - features

- Clock Management Tile (CMT) for enhanced performance
  - Low noise, flexible clocking
  - Digital Clock Managers (DCMs) eliminate clock skew and duty cycle distortion
  - Phase-Locked Loops (PLLs) for low-jitter clocking
  - Frequency synthesis with simultaneous multiplication, division, and phase shifting
  - Sixteen low-skew global clock networks



# Spartan-6 series - devices

Device	Logic Cells <sup>(1)</sup>	Configurable Logic Blocks (CLBs)			DSP48A1 Slices <sup>(3)</sup>	Block RAM Blocks		CMTs <sup>(5)</sup>	Memory Controller Blocks (Max) <sup>(6)</sup>	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices <sup>(2)</sup>	Flip-Flops	Max Distributed RAM (Kb)		18 Kb <sup>(4)</sup>	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232
XC6SLX25	24,051	3,758	30,064	229	38	52	936	2	2	0	0	4	266
XC6SLX45	43,661	6,822	54,576	401	58	116	2,088	4	2	0	0	4	358
XC6SLX75	74,637	11,662	93,296	692	132	172	3,096	6	4	0	0	6	408
XC6SLX100	101,261	15,822	126,576	976	180	268	4,824	6	4	0	0	6	480
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,824	6	4	0	0	6	576
XC6SLX25T	24,051	3,758	30,064	229	38	52	936	2	2	1	2	4	250
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,088	4	2	1	4	4	296
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,096	6	4	1	8	6	348
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,824	6	4	1	8	6	498
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,824	6	4	1	8	6	540



# Spartan-6 series - DSP48A1

- Spartan-6 FPGAs contain dedicated, full-custom, low-power DSP slices, combining high speed with small size, while retaining system design flexibility:
  - Up to 180 efficient, second generation DSP48A1 slices
  - High-performance arithmetic and signal processing
  - Each slice containing a fast  $18 \times 18$  multiplier and a 48-bit accumulator capable of operating at 250 MHz speed
  - Pipelining and cascading capability
  - Pre-adder to assist filter applications
  - 40% lower power consumption: 1.38mW/ 100MHz at a 38% toggle rate





# Virtex-6 series - features

- Virtex®-6 family provides the newest, most advanced features in the FPGA market
- Virtex-6 FPGAs contain many built-in system-level blocks
- Virtex-6 FPGAs offer the best solution for addressing the needs of:
  - high-performance logic designers,
  - high-performance DSP designers,
  - high-performance embedded systems designers with unprecedented logic, DSP, connectivity, and soft microprocessor capabilities.



# Virtex-6 series - features

- Three sub-families:
  - Virtex-6 LXT FPGAs: High-performance logic with advanced serial connectivity
  - Virtex-6 SXT FPGAs: Highest signal processing capability with advanced serial connectivity
  - Virtex-6 HXT FPGAs: Highest bandwidth serial connectivity
- Compatibility across sub-families
  - LXT and SXT devices are footprint compatible in the same package



# Virtex-6 series - features

- Advanced, high-performance FPGA Logic
  - Real 6-input look-up table (LUT) technology
  - Dual LUT5 (5-input LUT) option
  - LUT/dual flip-flop pair for applications requiring rich register mix
  - Improved routing efficiency
  - 64-bit (or two 32-bit) distributed LUT RAM option per 6-input LUT
  - SRL32/dual SRL16 with registered outputs option



# Virtex-6 series - features

- 36-Kb block RAM/FIFOs
  - Dual-port RAM blocks
  - Programmable
    - Dual-port widths up to 36 bits
    - Simple dual-port widths up to 72 bits
  - Enhanced programmable FIFO logic
  - Built-in optional error-correction circuitry
  - Optionally use each block as two independent 18 Kb blocks



## Virtex-6 series - features

- High-performance parallel SelectIO™ technology
  - 1.2 to 2.5V I/O operation
  - Source-synchronous interfacing using ChipSync™ technology
  - Digitally controlled impedance (DCI) active termination
  - Flexible fine-grained I/O banking
  - High-speed memory interface support with integrated write-leveling capability



# Virtex-6 series - features

- Advanced DSP48E1 slices
  - 25 x 18, two's complement multiplier/accumulator
  - Optional pipelining
  - New optional pre-adder to assist filtering applications
  - Optional bitwise logic functionality
  - Dedicated cascade connections
- GTX transceivers: up to 6.6 Gb/s
  - Data rates below 480 Mb/s supported by oversampling in FPGA logic.
- GTH transceivers: 2.488 Gb/s to beyond 11 Gb/s



# Virtex-6 series - features

- Integrated 10/100/1000 Mb/s Ethernet MAC block
  - Supports 1000BASE-X PCS/PMA and SGMII using GTX transceivers
  - Supports MII, GMII, and RGMII using SelectIO technology resources
  - 2500Mb/s support available
- 1.0V core voltage (-1, -2, -3 speed grades only)



# Virtex-6 series - devices

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices <sup>(2)</sup>	Block RAM Blocks			MMCMs <sup>(4)</sup>	Interface Blocks for PCI Express	Ethernet MACs <sup>(5)</sup>	Maximum Transceivers		Total I/O Banks <sup>(6)</sup>	Max User I/O <sup>(7)</sup>
		Slices <sup>(1)</sup>	Max Distributed RAM (Kb)		18 Kb <sup>(3)</sup>	36 Kb	Max (Kb)				GTX	GTH		
XC6VLX75T	74,496	11,640	1,045	288	312	156	5,616	6	1	4	12	0	9	360
XC6VLX130T	128,000	20,000	1,740	480	528	264	9,504	10	2	4	20	0	15	600
XC6VLX195T	199,680	31,200	3,040	640	688	344	12,384	10	2	4	20	0	15	600
XC6VLX240T	241,152	37,680	3,650	768	832	416	14,976	12	2	4	24	0	18	720
XC6VLX365T	364,032	56,880	4,130	576	832	416	14,976	12	2	4	24	0	18	720
XC6VLX550T	549,888	85,920	6,200	864	1,264	632	22,752	18	2	4	36	0	30	1200
XC6VLX760	758,784	118,560	8,280	864	1,440	720	25,920	18	0	0	0	0	30	1200
XC6VSX315T	314,880	49,200	5,090	1,344	1,408	704	25,344	12	2	4	24	0	18	720
XC6VSX475T	476,160	74,400	7,640	2,016	2,128	1,064	38,304	18	2	4	36	0	21	840
XC6VHX250T	251,904	39,360	3,040	576	1,008	504	18,144	12	4	4	48	0	8	320
XC6VHX255T	253,440	39,600	3,050	576	1,032	516	18,576	12	2	2	24	24	12	480
XC6VHX380T	382,464	59,760	4,570	864	1,536	768	27,648	18	4	4	48	24	18	720
XC6VHX565T	566,784	88,560	6,370	864	1,824	912	32,832	18	4	4	48	24	18	720



# New breed of FPGAs



# New Xilinx FPGA devices

Features	Artix-7	Kintex-7	Virtex-7	Spartan-6	Virtex-6
Logic Cells	352,000	480,000	2,000,000	150,000	760,000
BlockRAM	19Mb	34Mb	68Mb	4.8Mb	38Mb
DSP Slices	1,040	1,920	3,600	180	2,016
DSP Performance (symmetric FIR)	1,248GMACS	2,845GMACS	5,335GMACS	140GMACS	2,419GMACS
Transceiver Count	16	32	96	8	72
Transceiver Speed	6.6Gb/s	12.5Gb/s	28.05Gb/s	3.2Gb/s	11.18Gb/s
Total Transceiver Bandwidth (full duplex)	211Gb/s	800Gb/s	2,784Gb/s	50Gb/s	536Gb/s
Memory Interface (DDR3)	1,066Mb/s	1,866Mb/s	1,866Mb/s	800Mb/s	1,066Mb/s
PCI Express® Interface	Gen2x4	Gen2x8	Gen3x8	Gen1x1	Gen2x8
Agile Mixed Signal (AMS)/XADC	Yes	Yes	Yes		Yes
Configuration AES	Yes	Yes	Yes	Yes	Yes
I/O Pins	600	500	1,200	576	1,200
I/O Voltage	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.5V, 1.8V, 2.5V
EasyPath Cost Reduction Solution	-	-	Yes	-	Yes



**Artix-7  
Kintex-7  
Vitrex-7**



# Artix-7 series - features

- Low power and low cost family for high volume applications
- 28nm architecture:
  - Higher capacity
  - Higher performance
  - Lower power consumption
- Agile Mixed Signal technology
- Unified architecture with Kintex-7 and Virtex-7

# Artix-7 series - features

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices <sup>(2)</sup>	Block RAM Blocks <sup>(3)</sup>			Clock Mgmt Tiles (CMTs) <sup>(4)</sup>	PCIe <sup>(5)</sup>	GTPs	XADC Blocks	Total I/O Banks <sup>(6)</sup>	Max User I/O <sup>(7)</sup>
		Slices <sup>(1)</sup>	Max Distributed RAM (Kb)		18Kb	36Kb	Max (Kb)						
XC7A8	8,000	1,250	100	20	40	20	720	2	0	0	1	4	200
XC7A15	15,360	2,400	200	40	80	40	1,440	2	0	0	1	4	200
XC7A30T	33,600	5,250	400	80	104	52	1,872	5	1	4	1	5	250
XC7A50T	52,160	8,150	600	120	150	75	2,700	5	1	4	1	5	250
XC7A100T	101,440	15,850	1,188	240	270	135	4,860	6	1	8	1	6	300
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500
XC7A350T	360,000	56,250	4,638	1,040	1,030	515	18,540	12	1	16	1	12	600



# Kintex-7 series - features

- High-performance for cost-sensitive applications
- 2x better price-performance than previous generation high-performance FPGAs
- 50% lower power consumption than previous generation high-performance FPGAs

# Kintex-7 series - features

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices <sup>(2)</sup>	Block RAM Blocks <sup>(3)</sup>			CMTs <sup>(4)</sup>	PCIe <sup>(5)</sup>	GTx <sup>s</sup>	XADC Blocks	Total I/O Banks <sup>(6)</sup>	Max User I/O <sup>(7)</sup>
		Slices <sup>(1)</sup>	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7K70T	65,600	10,250	838	240	270	135	4,860	6	1	8	1	6	300
XC7K160T	162,240	25,350	2,188	600	650	325	11,700	8	1	8	1	8	400
XC7K325T	326,080	50,950	4,000	840	890	445	16,020	10	1	16	1	10	500
XC7K355T	356,160	55,650	5,088	1,440	1,430	715	25,740	6	1	24	1	6	300
XC7K410T	406,720	63,550	5,663	1,540	1,590	795	28,620	10	1	16	1	10	500
XC7K420T	416,960	65,150	5,938	1,680	1,670	835	30,060	7	1	28	1	7	350
XC7K480T	477,760	74,650	6,788	1,920	1,910	955	34,380	8	1	32	1	8	400



# Virtex-7 series - features

- Highest Bandwidth and System Performance
- 2x higher system performance than Virtex®-6 FPGAs
- Industry's only single-FPGA solution for 400G applications
- Industry's only 2 million logic-cell FPGA



# Virtex-7 series - features

Device <sup>(1)</sup>	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices <sup>(3)</sup>	Block RAM Blocks <sup>(4)</sup>			CMTs <sup>(5)</sup>	PCIe <sup>(6)</sup>	GTX	GTH	GTZ	XADC Blocks	Total I/O Banks <sup>(7)</sup>	Max User I/O <sup>(8)</sup>	SLRs <sup>(9)</sup>
		Slices <sup>(2)</sup>	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)									
XC7V585T	582,720	91,050	6,938	1,260	1,590	795	28,620	18	3	36	0	0	1	17	850	N/A
XC7V1500T	1,465,920	229,050	16,163	1,620	1,938	969	34,884	18	3	36	0	0	1	17	850	3
XC7V2000T	1,954,560	305,400	21,550	2,160	2,584	1,292	46,512	24	4	36	0	0	1	24	1,200	4
XC7VX330T	326,400	51,000	4,388	1,120	1,500	750	27,000	14	2	0	28	0	1	14	700	N/A
XC7VX415T	412,160	64,400	6,525	2,160	1,760	880	31,680	12	2	0	48	0	1	12	600	N/A
XC7VX485T	485,760	75,900	8,175	2,800	2,060	1,030	37,080	14	4	56	0	0	1	14	700	N/A
XC7VX550T	554,240	86,600	8,725	2,880	2,360	1,180	42,480	16	2	0	64	0	1	16	600	N/A
XC7VX690T	693,120	108,300	10,888	3,600	2,940	1,470	52,920	20	3	0	80	0	1	20	1,000	N/A
XC7VX980T	979,200	153,000	13,838	3,600	3,000	1,500	54,000	18	3	0	72	0	1	18	880	N/A
XC7VX1140T	1,139,200	178,000	17,700	3,360	3,760	1,880	67,680	24	4	0	96	0	1	22	1,100	4
XC7VH290T	284,000	44,375	4,425	840	940	470	16,920	6	1	0	24	8	1	6	300	1
XC7VH580T	580,480	90,700	8,850	1,680	1,880	940	33,840	12	2	0	48	8	1	12	600	2
XC7VH870T	876,160	136,900	13,275	2,520	2,820	1,410	50,760	18	3	0	72	16	1	13	650	3



# Artix, Kintex, Virtex-7 series - features

Maximum Capability	Artix-7 Family	Kintex-7 Family	Virtex-7 Family
Logic Cells	360K	478K	1,955K
Block RAM <sup>(1)</sup>	19 Mb	34 Mb	68 Mb
DSP Slices	1,040	1,920	3,600
Peak DSP Performance <sup>(2)</sup>	1,248 GMACS	2,845 GMACS	5,335 GMACS
Transceivers	16	32	96
Peak Transceiver Speed	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Peak Serial Bandwidth (Full Duplex)	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	600	500	1,200
I/O Voltage	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V
Package Options	Low-Cost, Wire-Bond, Lidless Flip-Chip	Low-Cost, Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip



# DSP slice - features

- $25 \times 18$  two's complement multiplier/accumulator high-resolution (48 bit) signal processor
- Power saving pre-adder to optimize symmetrical filter applications
- Advanced features:
  - optional pipelining,
  - optional ALU,
  - dedicated buses for cascading



# DSP slice - operation

- Each DSP slice can operate up to 638 MHz
- The multiplier can be dynamically bypassed, and two 48-bit inputs can feed a single instruction-multiple-data (SIMD) arithmetic unit:
  - dual 24-bit add/subtract/accumulate
  - quad 12-bit add/subtract/accumulate



# Low-Power Gigabit Transceivers-features

- High-performance transceivers capable of up to 6.6 Gb/s (GTP), 12.5 Gb/s (GTX), 13.1 Gb/s (GTH), or 28.05 Gb/s(GTZ) line rates depending on the family
- Low-power mode optimized for chip-to-chip interfaces
- Advanced Transmit pre and post emphasis, and receiver linear (CTLE) and decision feedback equalization (DFE), including adaptive equalization for additional margin.



# PCIe features

- Compliant to the PCI Express Base Specification 2.1 or 3.0 (depending of family) with Endpoint and Root Port capability
- Supports Gen1 (2.5 Gb/s), Gen2 (5 Gb/s), and Gen3 (8 Gb/s) depending on device family
- Advanced configuration options, Advanced Error Reporting (AER), and End-to-End CRC (ECRC)  
Advanced Error Reporting and ECRC features



# XADC features

- Dual 12-bit 1 MSPS analog-to-digital converters (ADCs)
- Up to 17 flexible and user-configurable analog inputs
- On-chip or external reference option
- On-chip temperature ( $\pm 4^\circ\text{C}$  max error) and power supply ( $\pm 1\%$  max error) sensors
- Continuous JTAG access to ADC measurements



# Extensible Processing Platforms

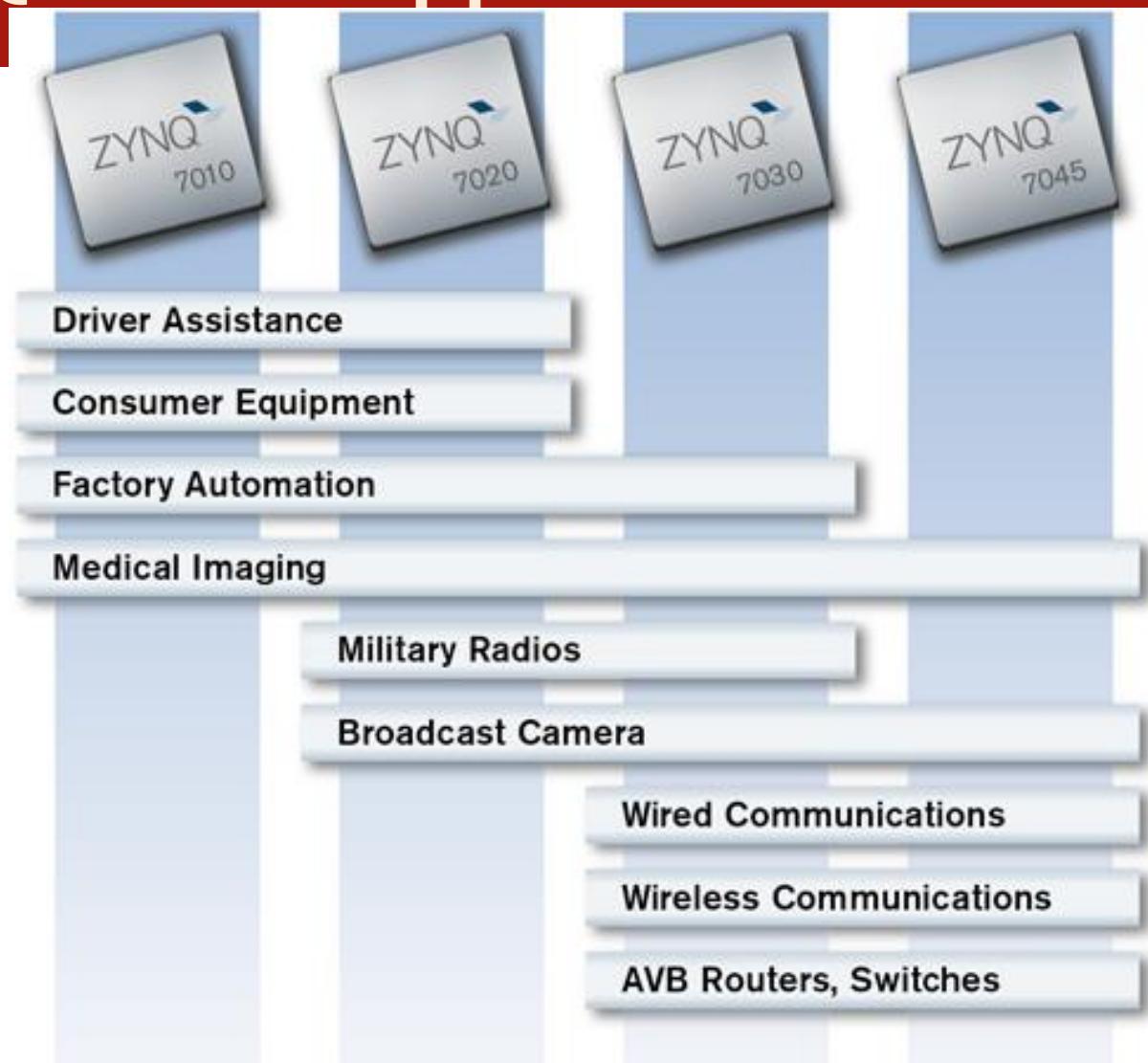
## SoCs - ZYNQs



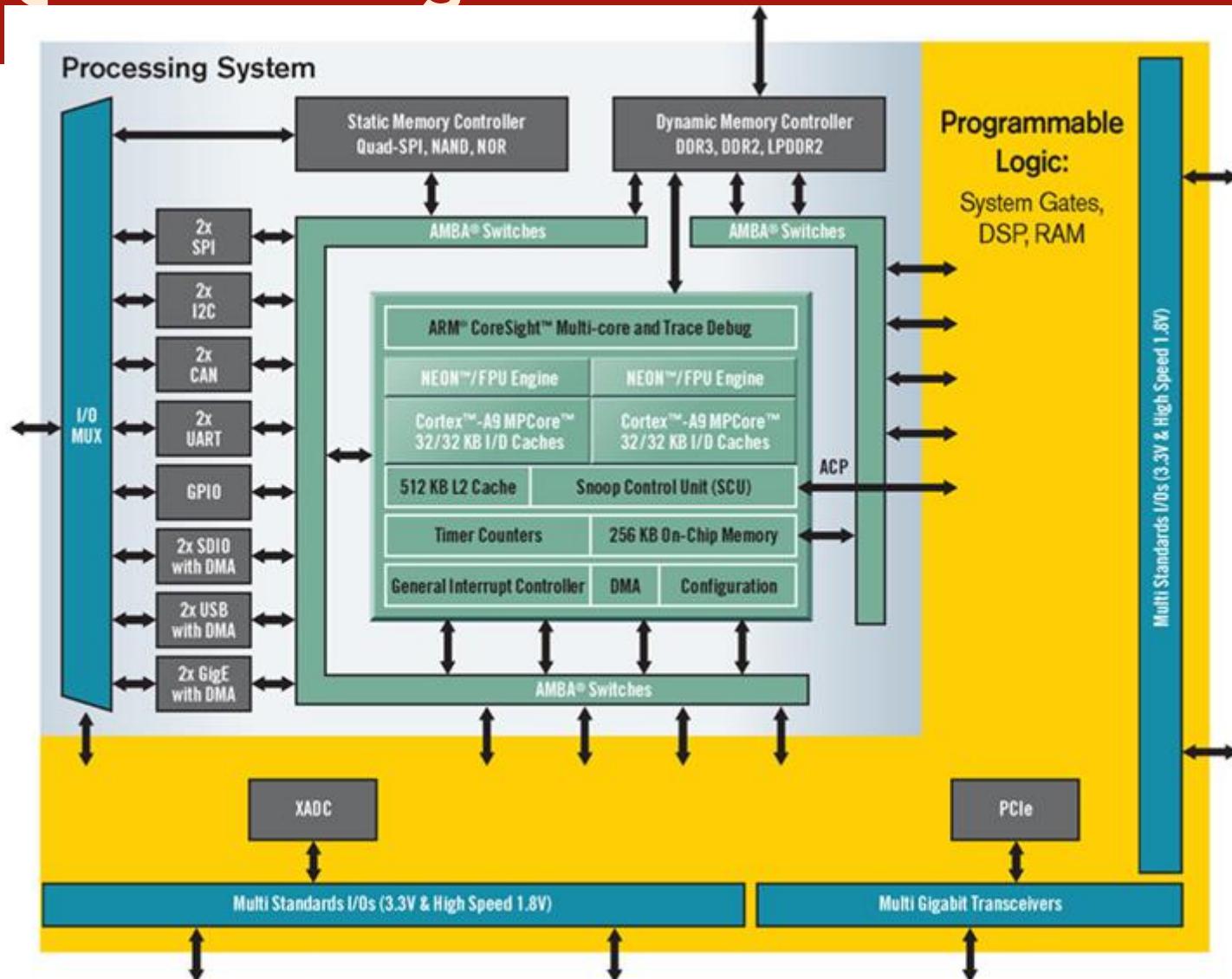
## EPP features (ZYNQ-7000)

- A new class of product
- Combines an industry-standard ARM® dual-core Cortex™-A9 MPCore™ processing system with Xilinx 28nm programmable logic unified architecture
- This processor-centric architecture offers the flexibility and scalability of an FPGA combined with ASIC-like performance and power and the ease of use of an ASSP

# ZYNQ-7000 - applications



# ZYNQ block diagram





# ZYNQ features

- Dual ARM Cortex™-A9 MPCore
  - Up to 800MHz
  - Enhanced with NEON Extension and Single & Double Precision Floating point unit
  - 32kB Instruction & 32kB Data L1 Cache
- Unified 512kB L2 Cache
- 256kB on-chip Memory
- DDR3, DDR2 and LPDDR2 Dynamic Memory Controller



# ZYNQ features

- 2x QSPI, NAND Flash and NOR Flash Memory Controller
- 2x USB2.0 (OTG), 2x GbE, 2x CAN2,0B 2x SD/SDIO, 2x UART, 2x SPI, 2x I2C, 4x 32b GPIO
- AES & SHA 256b encryption engine for secure boot and secure configuration
- Dual 12bit 1Msps Analog-to-Digital converter
- Up to 17 Differential Inputs



# ZYNQ features

- Advanced Low Power 28nm Programmable Logic:
  - 28k to 350k Logic Cells (approximately 430k to 5.2M of equivalent ASIC Gates)
  - 240KB to 2180KB of Extensible Block RAM
  - 80 to 900 18x25 DSP Slices (58 to 1080 GMACS peak DSP performance)
- PCI Express® Gen2x8 (in largest devices)
- 154 to 404 User IOs (Multiplexed + SelectIO™)
- 4 to 16 12.5Gbps Transceivers (in largest devices)



# ZYNQ features



## Zynq-7000 SoC Artix Devices

Dual ARM® Cortex™-A9



## Zynq UltraScale+ MPSoC CG Devices

Dual ARM Cortex-A53  
Dual ARM Cortex-R5



## Zynq UltraScale+ MPSoC EV Devices

Quad ARM Cortex-A53  
Dual Cortex-R5 + GPU + Video  
Codec



## Zynq UltraScale+ RFSoC with RF Data Converters

Quad ARM Cortex-A53  
Dual Cortex-R5

## Zynq-7000S SoC Artix Devices

Single ARM Cortex-A9

## Zynq-7000 SoC Kintex Devices

Dual ARM Cortex-A9

## Zynq UltraScale+ MPSoC EG Devices

Quad ARM Cortex-A53  
Dual Cortex-R5 + GPU

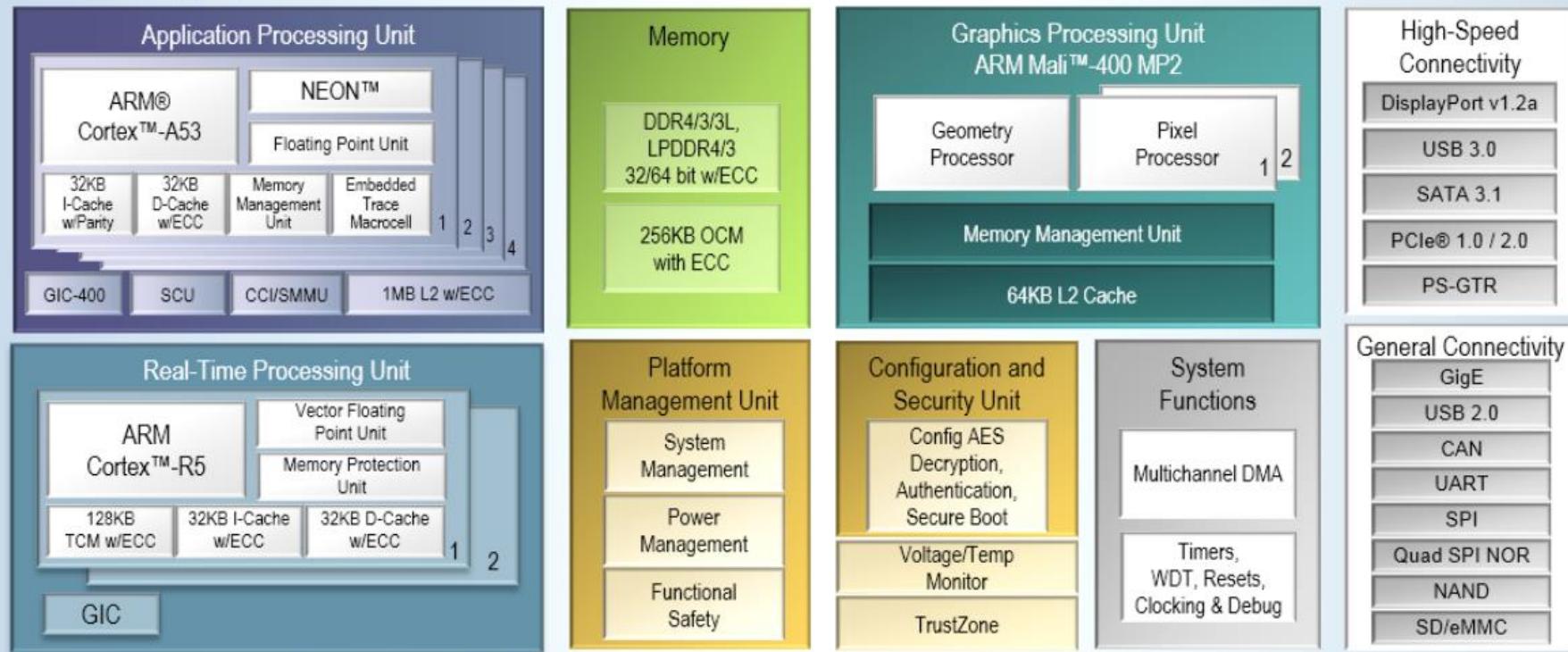
## Zynq UltraScale+ RFSoC with SD-FEC Cores

Quad ARM Cortex-A53  
Dual ARM Cortex-R5

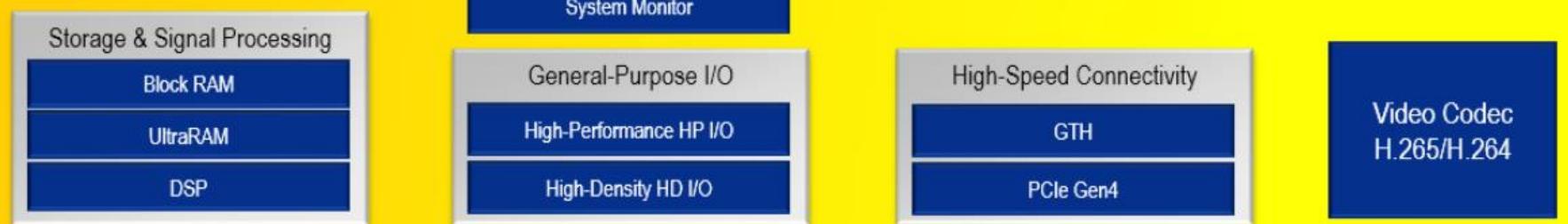
## Zynq UltraScale+ RFSoC with RF Data Converters & SD-FEC Cores

Quad ARM Cortex-A53  
Dual ARM Cortex-R5

## Processing System



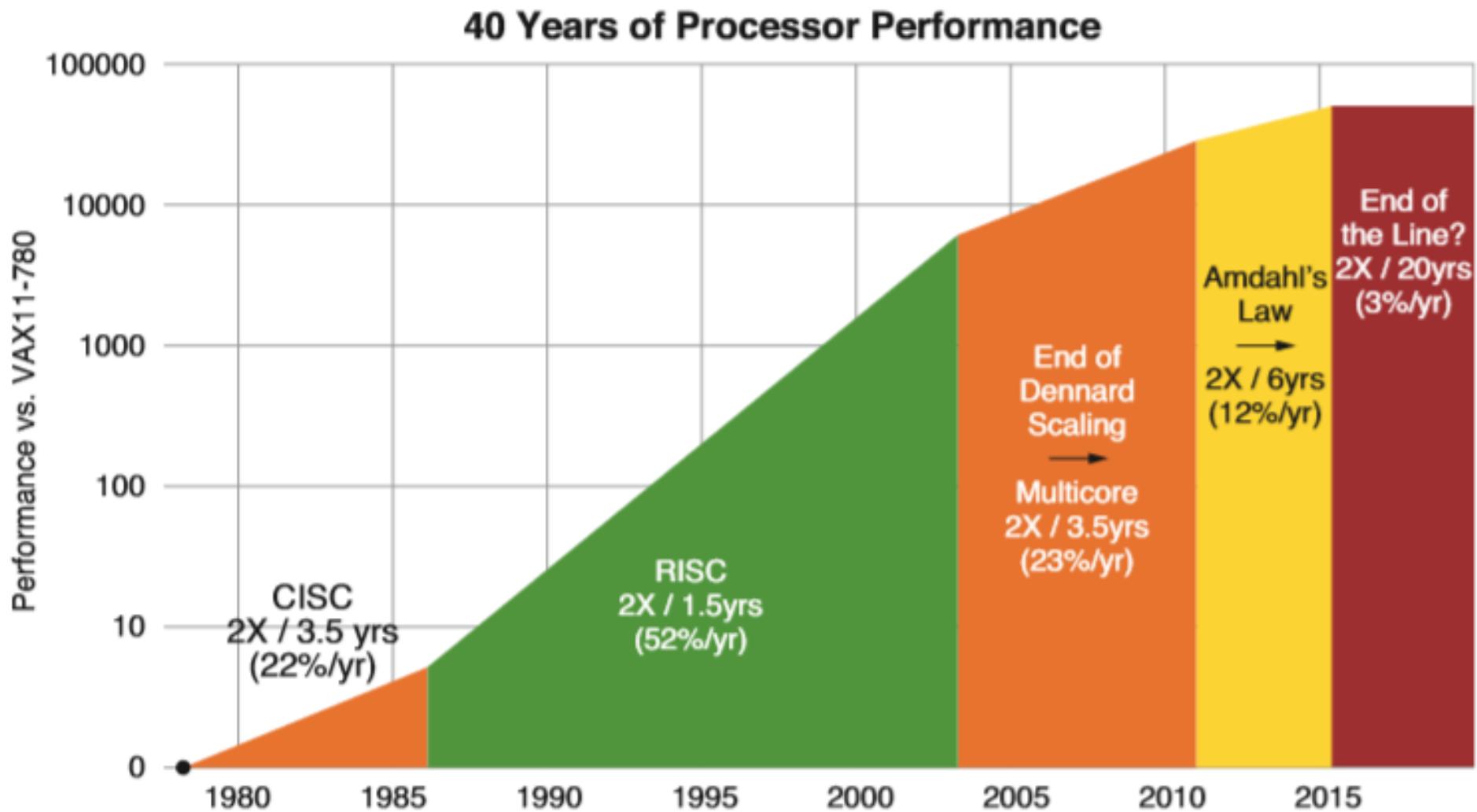
## Programmable Logic





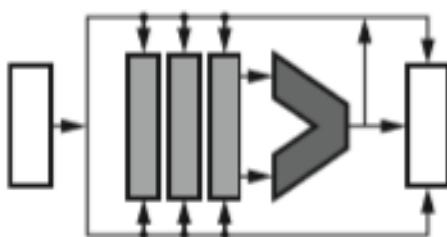
# Future

# Future



# Computing engines

## Scalar Processing



Complex Algorithms  
and Decision Making

## Adaptable Hardware

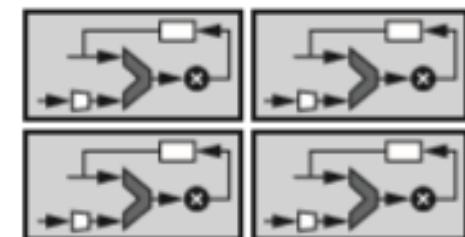


Processing of  
Irregular Data Structures  
*Genomic Sequencing*

Latency  
Critical Workloads  
*Real-Time Control*

Sensor Fusion  
*Pre-processing, Programmable I/O*

## Vector Processing (e.g., GPU, DSP)

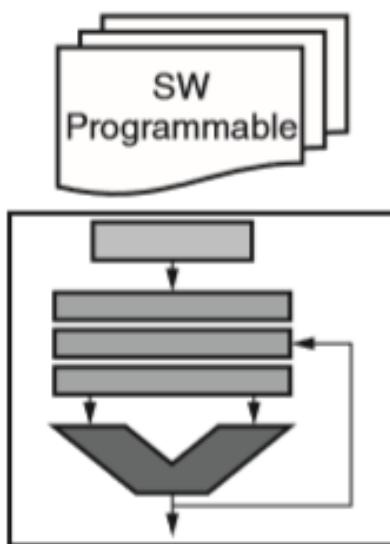
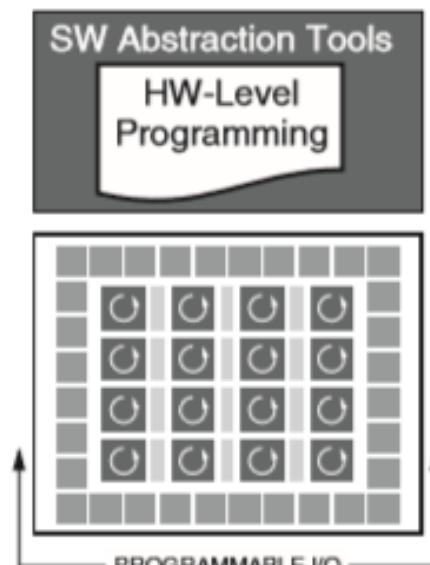
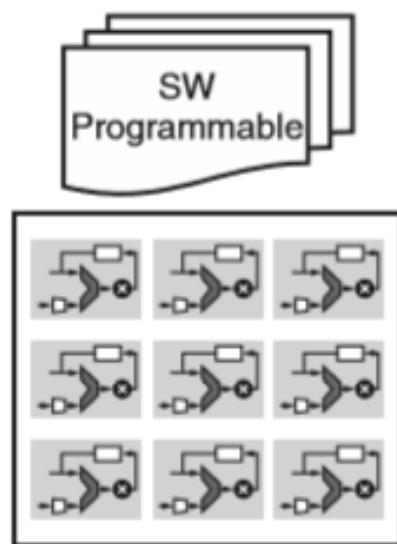


Domain-specific  
Parallelism

Signal Processing  
*Complex Math, Convolutions*

Video and  
Image Processing

# Integration of Computing Engines

**CPU****FPGA****Vector Processor**

- Scalar, sequential processing
- Memory bandwidth limited
- Fixed pipeline, fixed I/O

- Flexible parallel compute
- Fast local memory
- Custom I/O

- Domain-specific parallelism
- High compute efficiency
- Fixed I/O and memory bandwidth

Scalar Engines

Adaptable Engines

Intelligent Engines

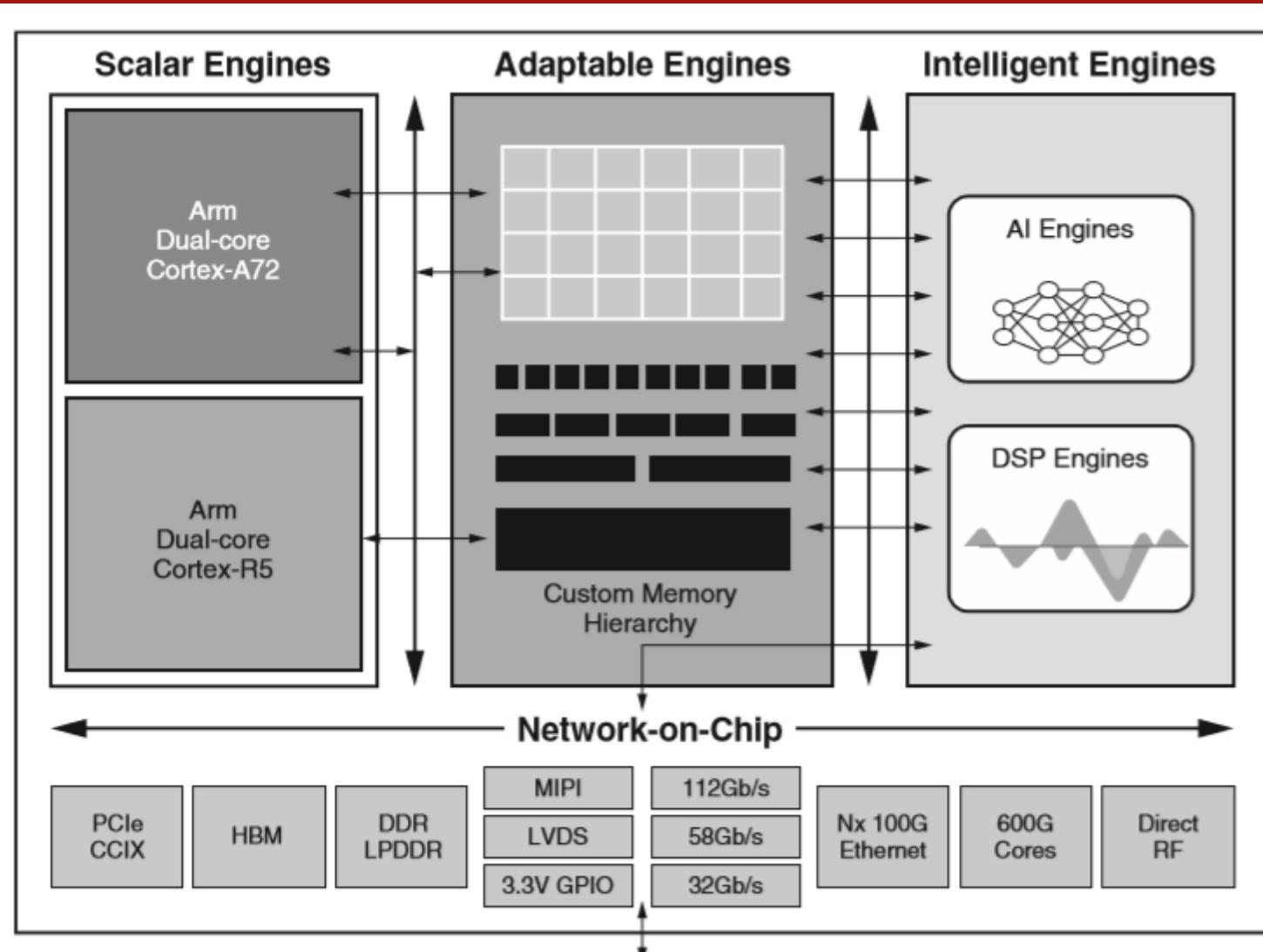
Integrated Software Programmable Interface



# Xilinx ACAP

- ACAP = Adaptive Compute Acceleration Platform
- ACAP = Hardware and Software Optimized Platform for Parallel Heterogeneous Computation
- ACAP = a mix of next-generation Scalar Engines, Adaptable Engines, and Intelligent Engines

# ACAP functional diagram

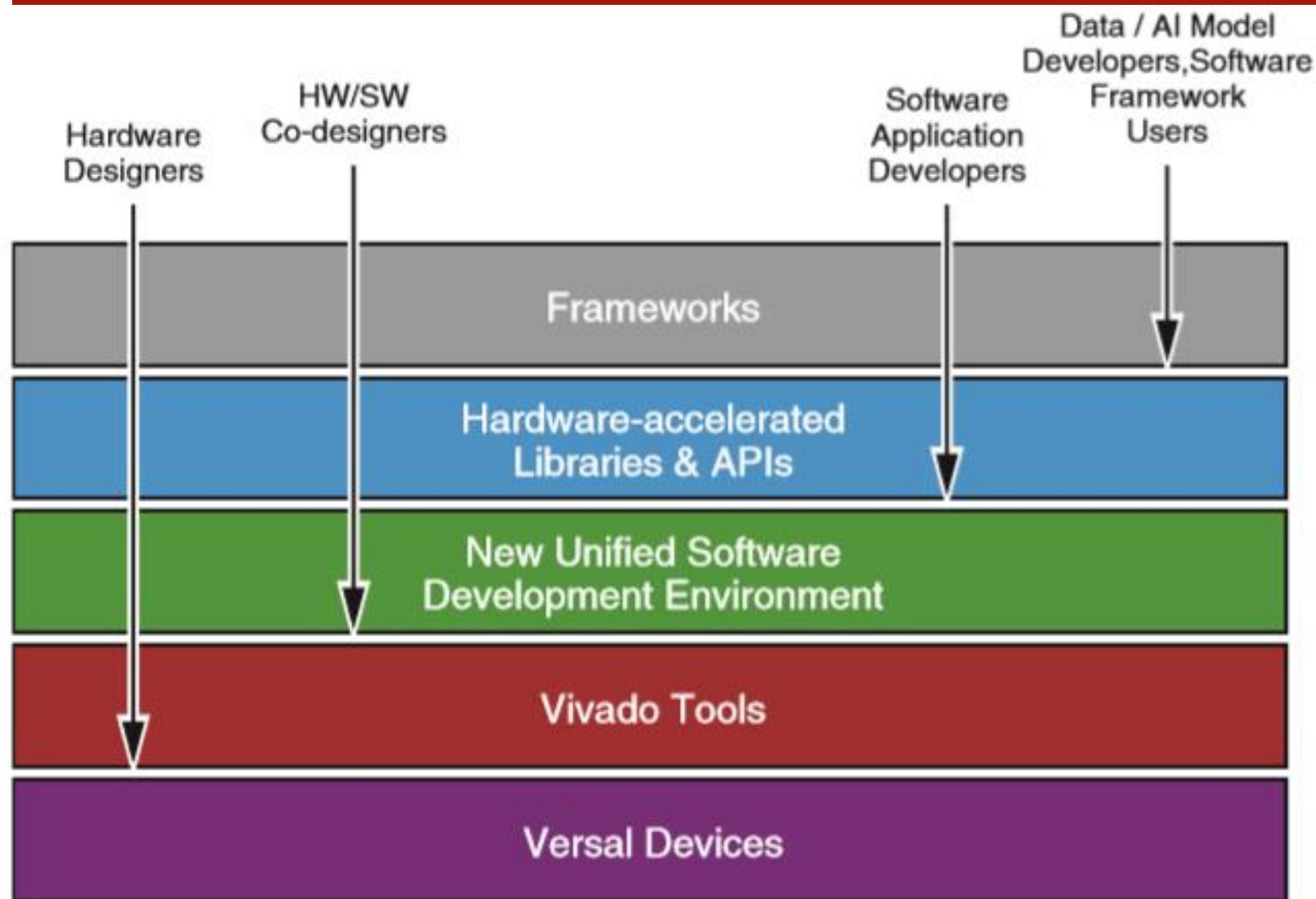




# Xilinx ACAP - advantages

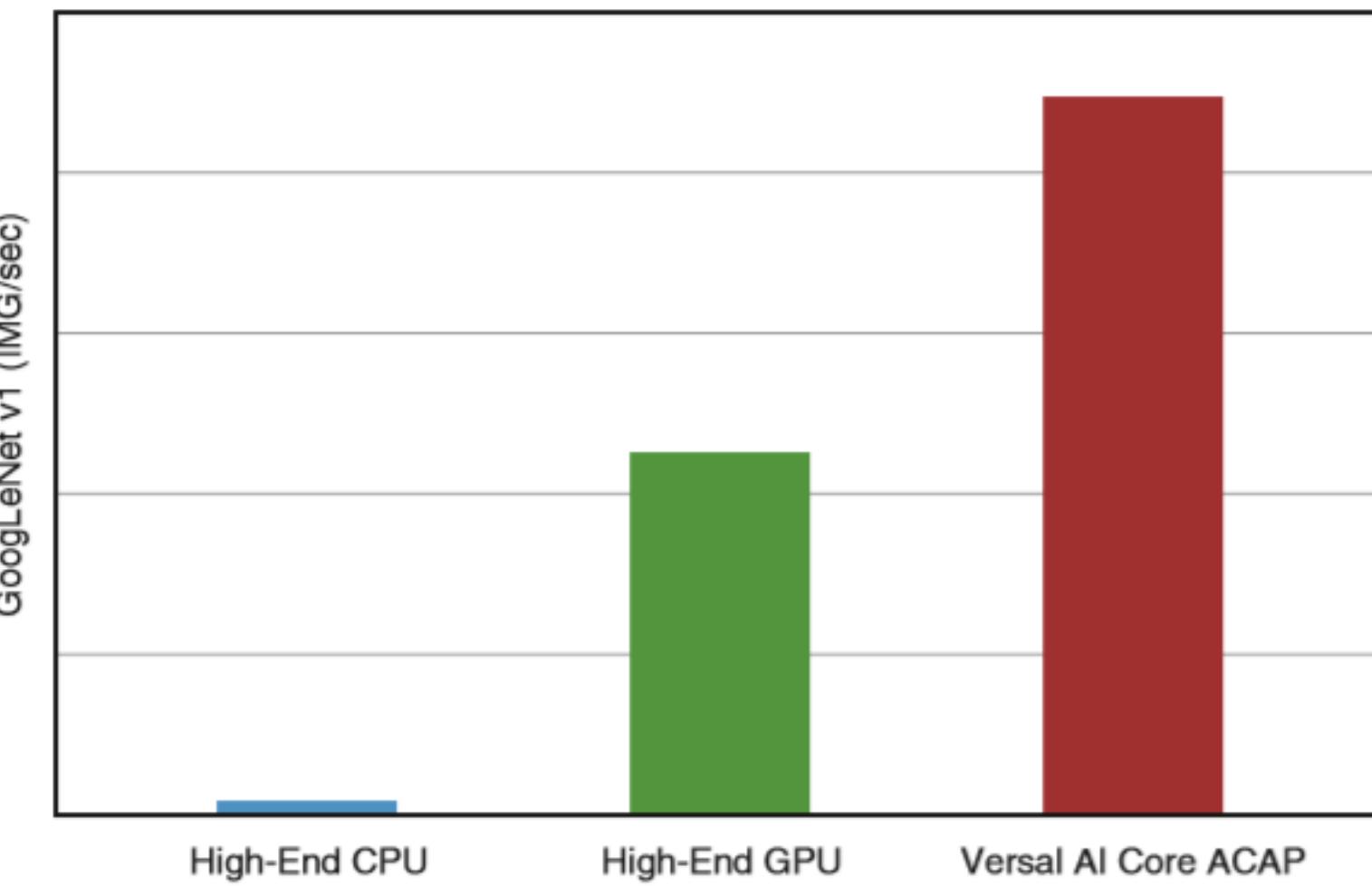
- Software Programmability
  - The ability to quickly develop optimized applications through software-abstraction toolchains.
- Acceleration
  - Metrics for a wide range of applications from artificial intelligence, smart network interface cards, high density storage, 5G wireless, self-driving cars, advanced modular radar, and terabit optical networks.
- Dynamically Adaptable Reconfiguration
  - The ability to reconfigure the hardware to accelerate new loads within milliseconds.

# Xilinx ACAP - software concept



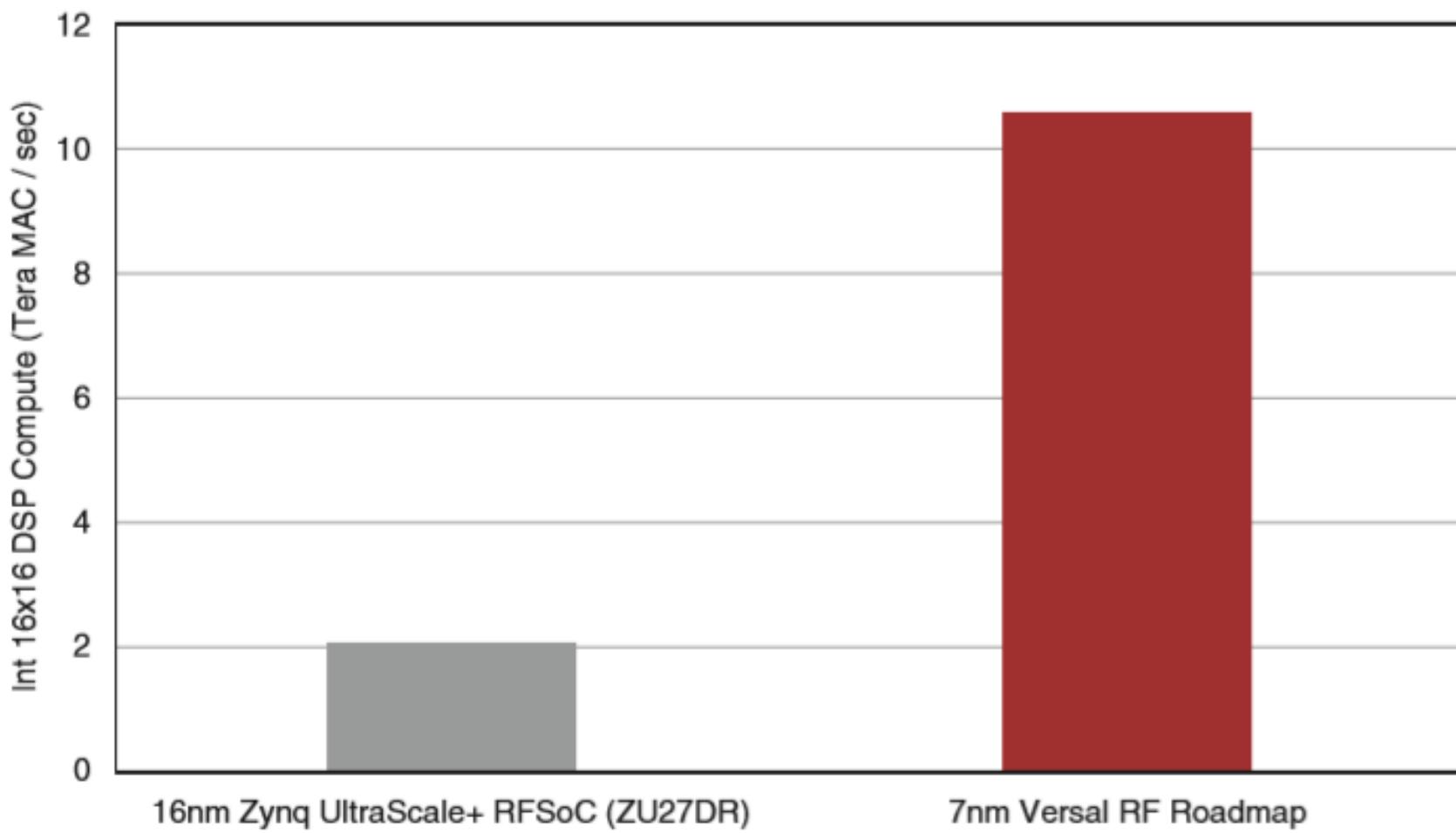
# Xilinx ACAP - gains

Machine Learning Inference  
Latency Insensitive (High Batch)



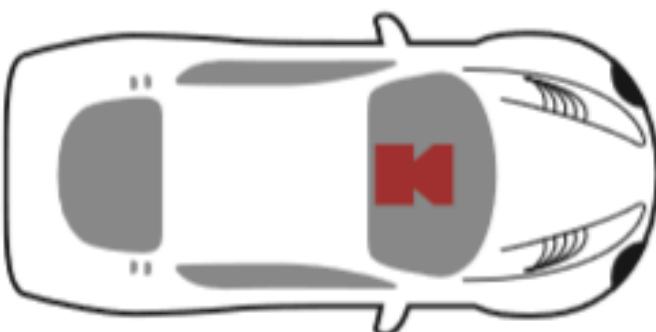
# Xilinx ACAP - gains

**Xilinx 5G and Radar DSP Compute Enhancements  
(in 16x16 Tera Multiply-accumulates / sec)**



# Xilinx ACAP - gains

- Xilinx ACAP Devices Enable Sensor Fusion in Small Power Envelopes



1x HD Camera

~10W



Sensor Fusion

4x HD Cameras

Radar

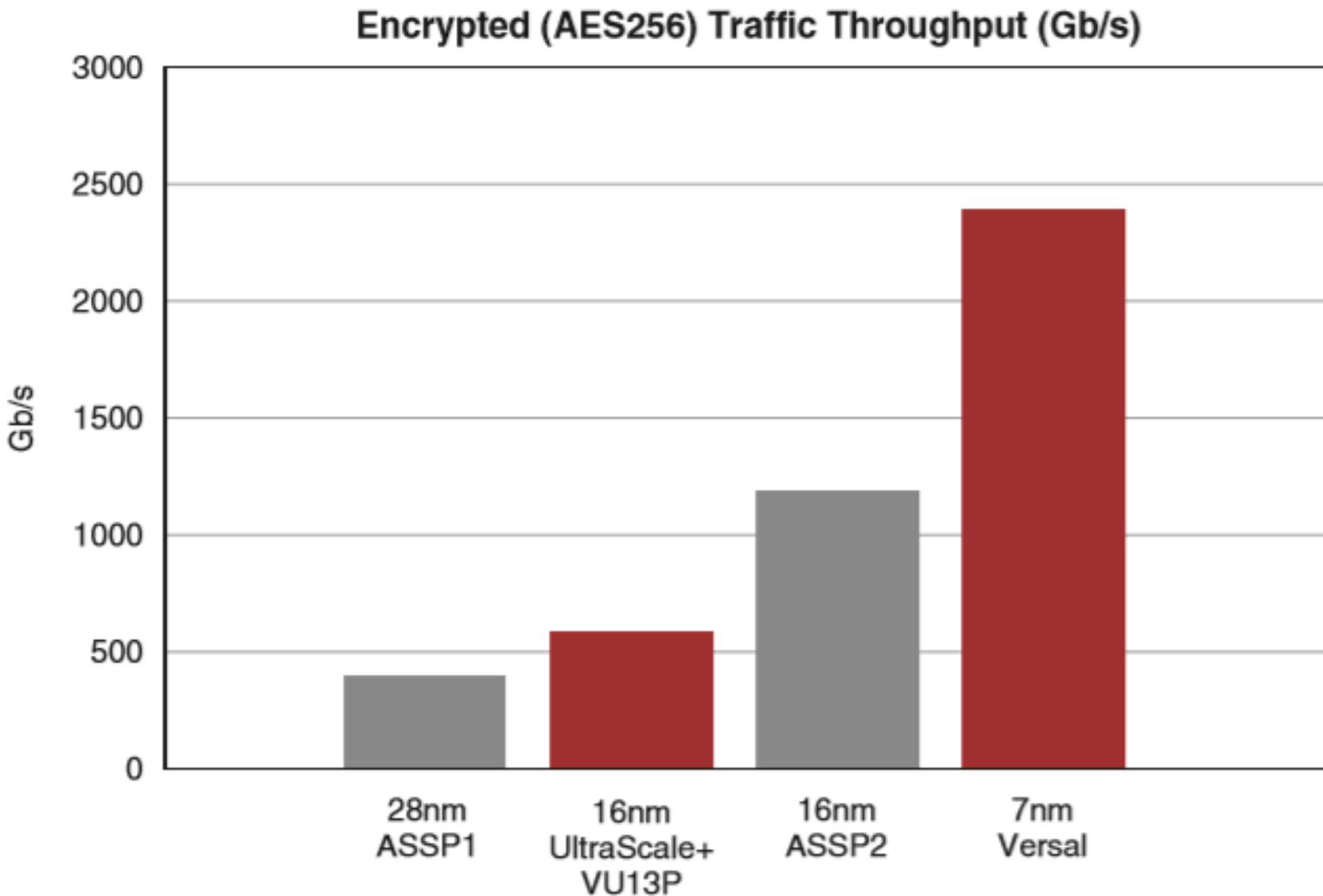
Ultrasound

LIDAR

Machine Learning

~10W

# Xilinx ACAP





**Thank you for your attention**



# References

- [1] CoolRunner II family documentation; [www.xilinx.com](http://www.xilinx.com)
- [2] EasyPath-6 family documentation; [www.xilinx.com](http://www.xilinx.com)
- [3] 7-th series documentation; [www.xilinx.com](http://www.xilinx.com)
- [4]  
[http://www.xilinx.com/publications/prod\\_mktg/Generation-Ahead-Backgrounder.pdf](http://www.xilinx.com/publications/prod_mktg/Generation-Ahead-Backgrounder.pdf)



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.



Unless otherwise noted, the pictures in this presentation were taken from free resources of Internet for the sole purpose of education.