
Computer aided Digital System Design

Dr. Mohsen Raji

School of Electrical & Computer Engineering
Shiraz University



Fall 2020

Course Materials

- Based on the slides:
 - R. Sameni, "Digital Systems Design Lecture Notes", School of Electrical & Computer Engineering, Shiraz University, Shiraz, Iran, version 2018.
- Some books and other types of references:
 - Bobda, C. (2007). Introduction to reconfigurable computing: architectures, algorithms, and applications. Springer Science & Business Media.

Evaluation

- Exercises, Quizzes, ... 40%
- Midterm Exam 30%
- Final Exam 30%

Table of Contents

- Part I: Architecture
- Part II: Electronic Design Automation
- Part III: Advanced Topics in Digital Systems
Design and Implementation

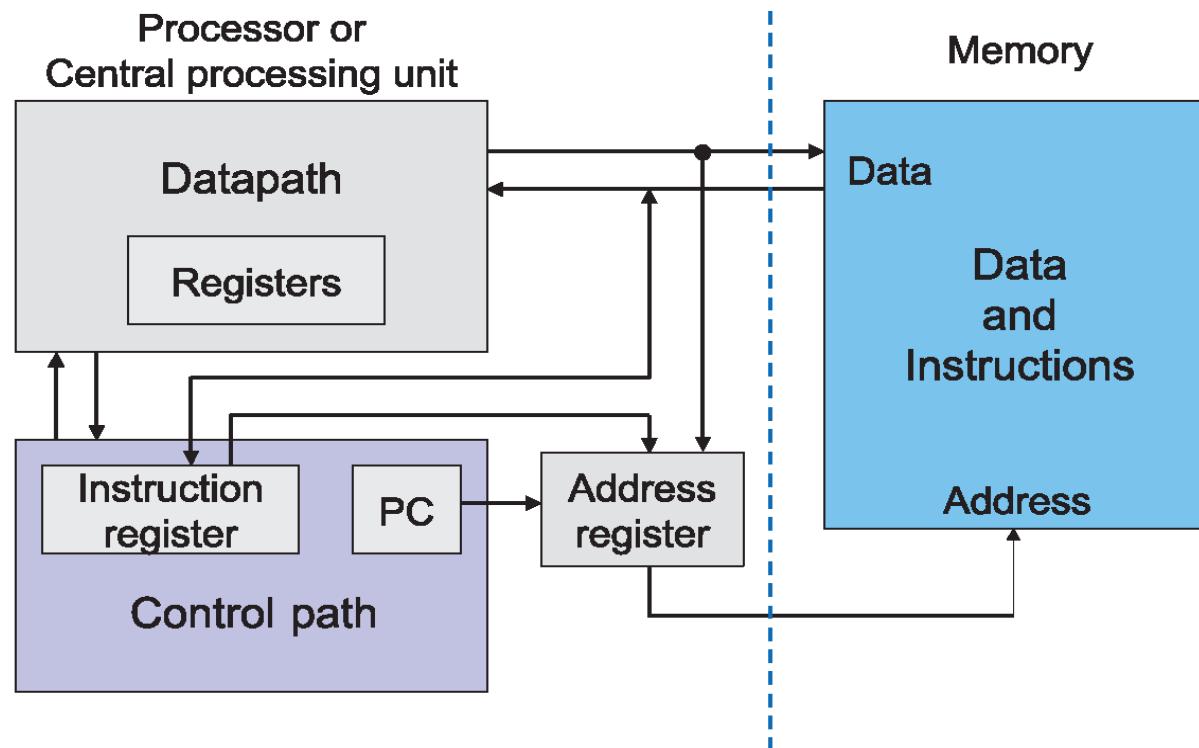
PART I

Architecture

INTRODUCTION

The von Neumann Architecture

The von Neumann Architecture (Bobda 2007)



The von Neumann Architecture

The VN Architecture consists of:

1. A memory for storing program and data (Harvard architectures contain two parallel accessible memories for storing program and data separately)
2. A control unit (also called **control path**) featuring a program counter that holds the address of the next instruction to be executed.
3. An arithmetic and logic unit (also called **data path**) in which instructions are executed.

The von Neumann Architecture

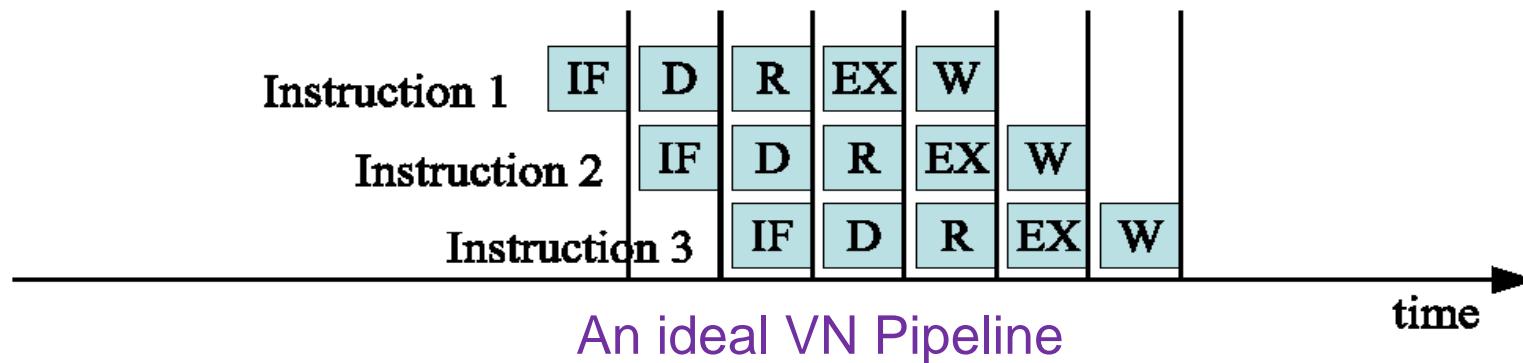
The execution of an instruction on a VN computer can be done in five cycles:

1. Instruction Fetch (IF): An instruction is fetched from the memory
2. Decoding (D): The meaning of the instruction is determined and the operands are localized
3. Read Operands (R): The operands are read from the memory
4. Execute (EX): The instruction is executed with the read operands
5. Write Result (W): The result of the execution is stored back to the memory



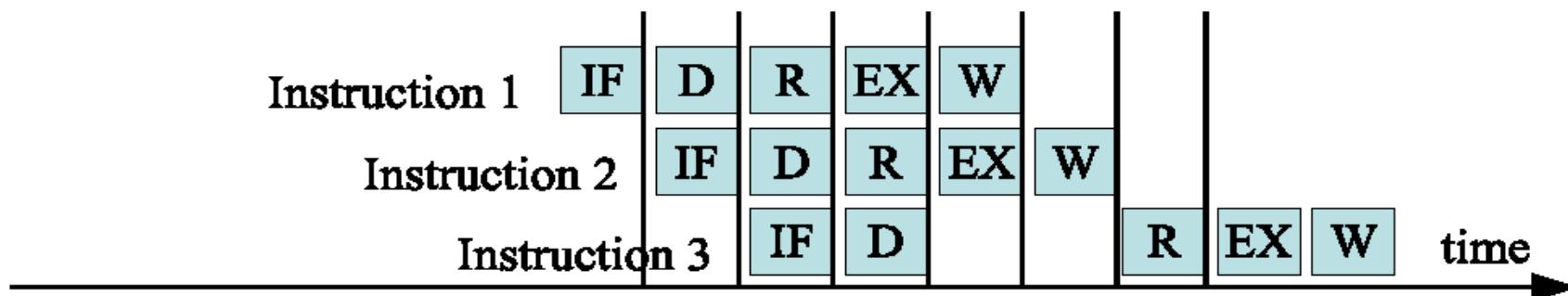
Pipelining in von Neumann Architectures

- Pipelining or *instruction level parallelism (ILP)* can be used to optimize the hardware utilization as well as the performance of programs.
- ILP does not reduce the *execution latency* of a single execution, but increases the *throughput* of a set of instructions.
- The maximum throughput is dictated by the impact of *hazards* in the computation. Hazards can be reduced, e.g., by the use of a Harvard architecture.



Pipelining in von Neumann Architectures

Ideal pipelining is commonly unachievable. For example, the Harvard architecture pipeline is as follows:



Harvard Architecture Pipeling

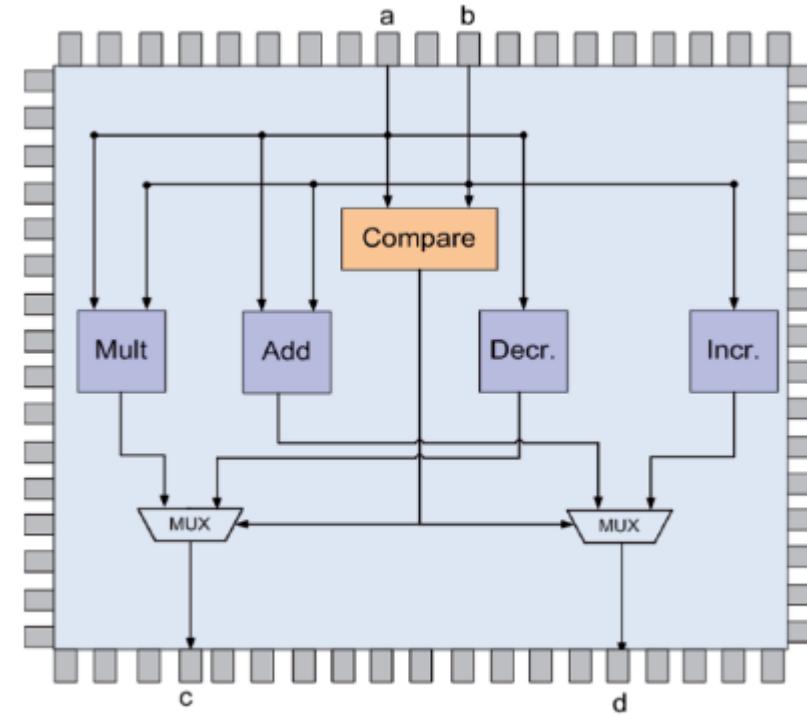
Application Specific Hardware versus von Neumann Architectures

Algorithm 1

```

if  $a < b$  then
     $d = a + b$ 
     $c = a \cdot b$ 
else
     $d = b + 1$ 
     $c = a - 1$ 
end if

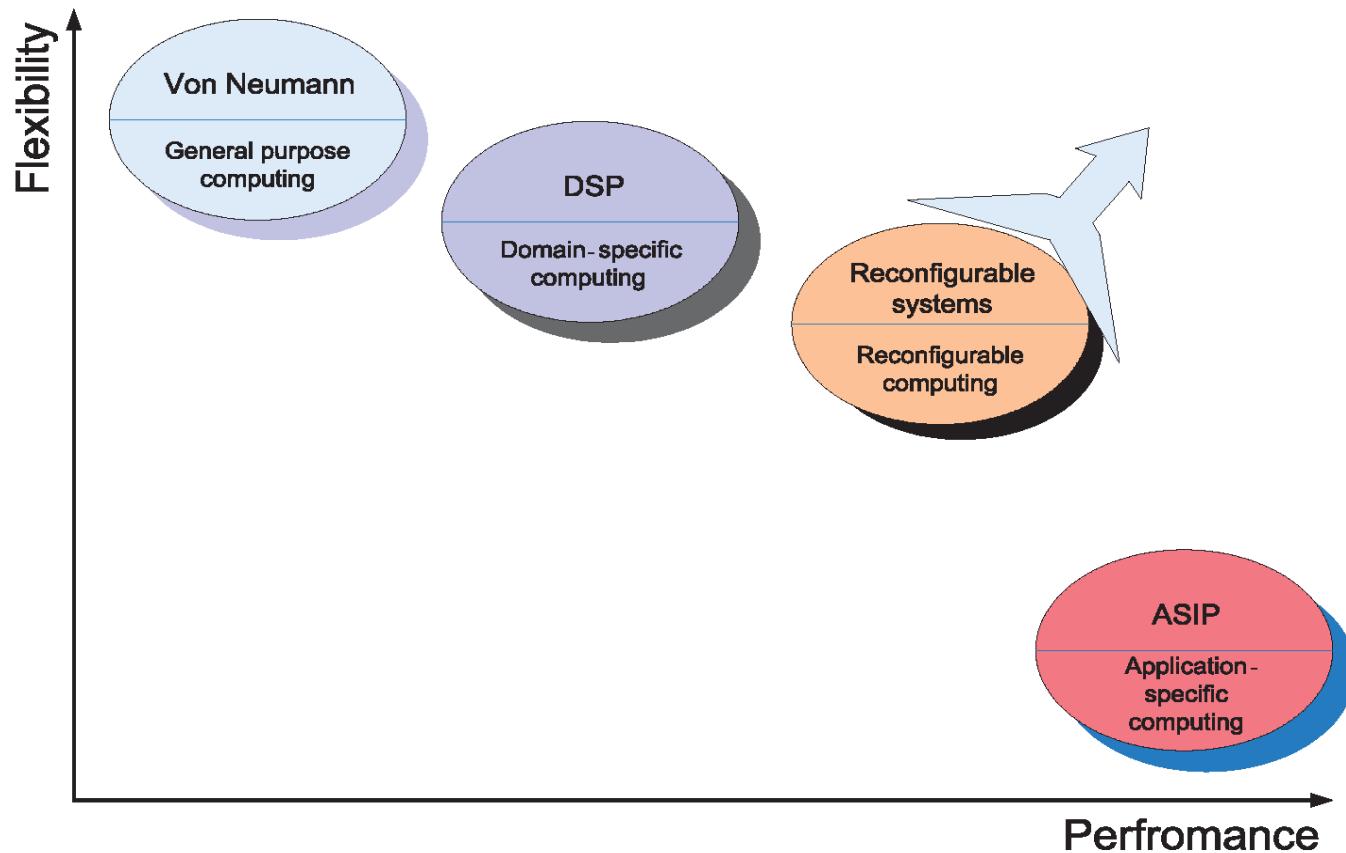
```



VN versus ASIP: Considering 5 cycles per instruction, the VN should be 15 times faster than the ASIP to outperform its speed ([Bobda 2007](#))

✓ Application specific hardware have higher performance, at a cost of lower flexibility

Flexibility vs. Performance



Flexibility vs. performance of different architectures (Bobda 2007)

Applications of Reconfigurable Architectures

- Rapid prototyping
 - reduced time-to-market
- In-system customization
 - hardware updates and patches
- Remote reconfiguration
 - via RF links for telecommunication BTS, spacecrafts, satellites,...
- Multi-modal computation
 - Environment aware hardware
- Adaptive computing systems
 - Machine learning applications

References

- Bobda, C. (2007). *Introduction to reconfigurable computing: architectures, algorithms, and applications.* Springer Science & Business Media.

PROGRAMMABLE LOGIC DEVICES & TECHNOLOGIES

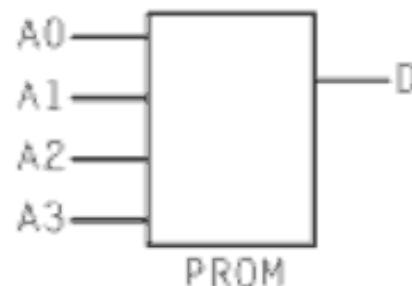
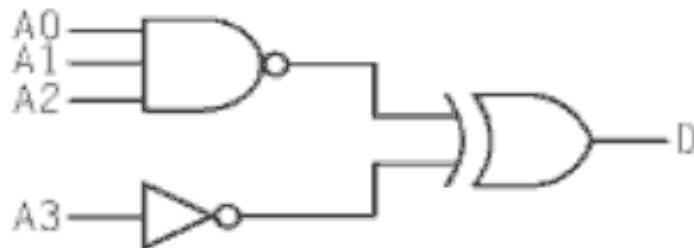
PLD Technologies

Programmable Logic Devices (PLD) have a long history (longer than conventional VN architecture CPUs):

- PROM
- Logic Chips
- SPLD: PLA & PAL
- CPLD
- FPGA
- ASIC

Programmable ROM (PROM)

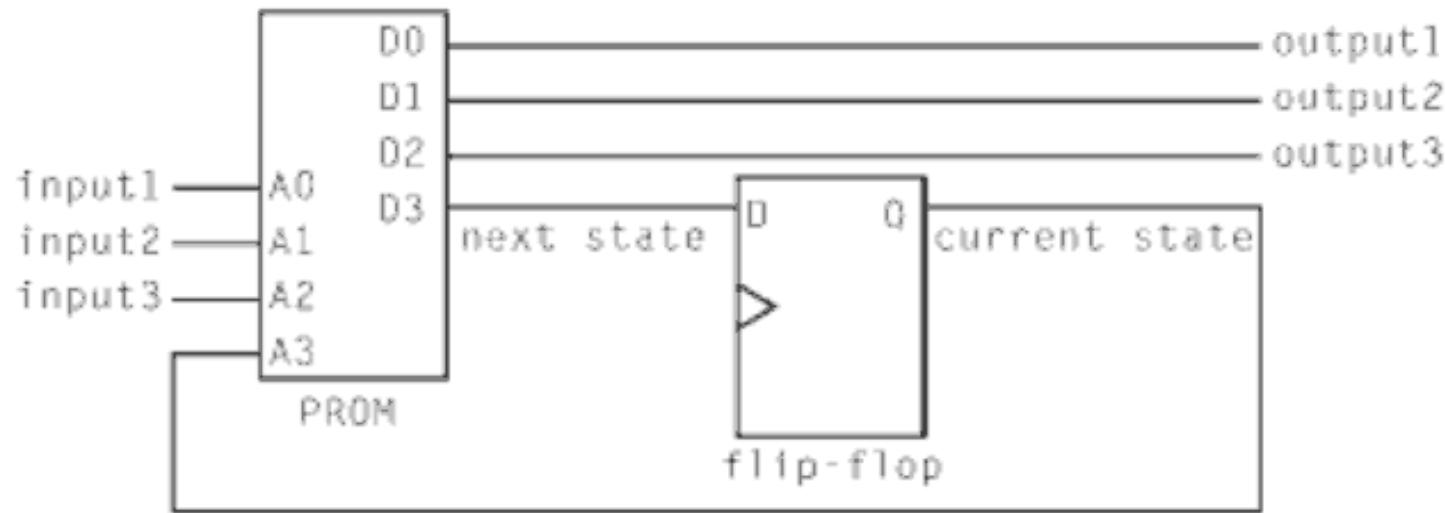
Example 1: PROM-based Combinatorial logic



Inputs A[3:0]	Output D
0000	0
0001	0
0010	0
0011	0
0100	0
0101	0
0110	0
0111	1
1000	1
1001	1
1010	1
1011	1
1100	1
1101	1
1110	1
1111	0

Programmable ROM (PROM)

Example 2: PROM-based state machine

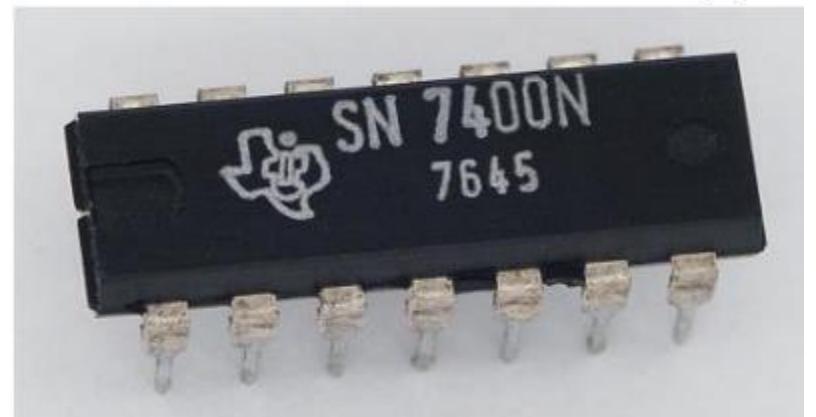
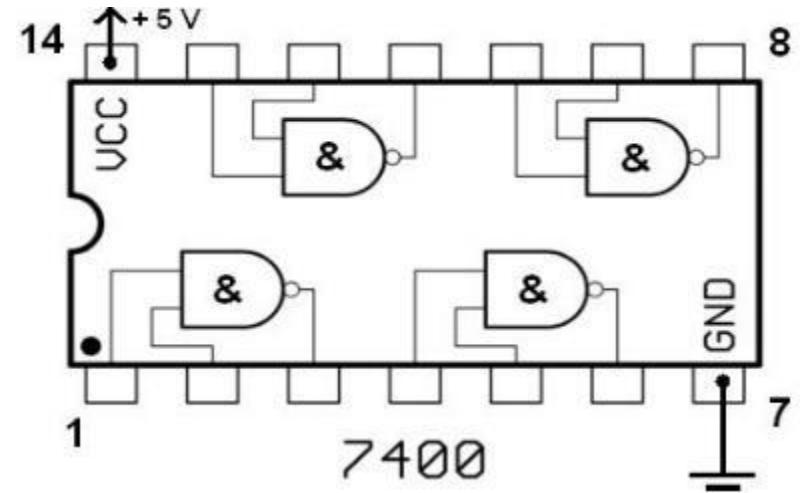


Benefit: Any logic circuit may be implemented

Drawback: Low speed

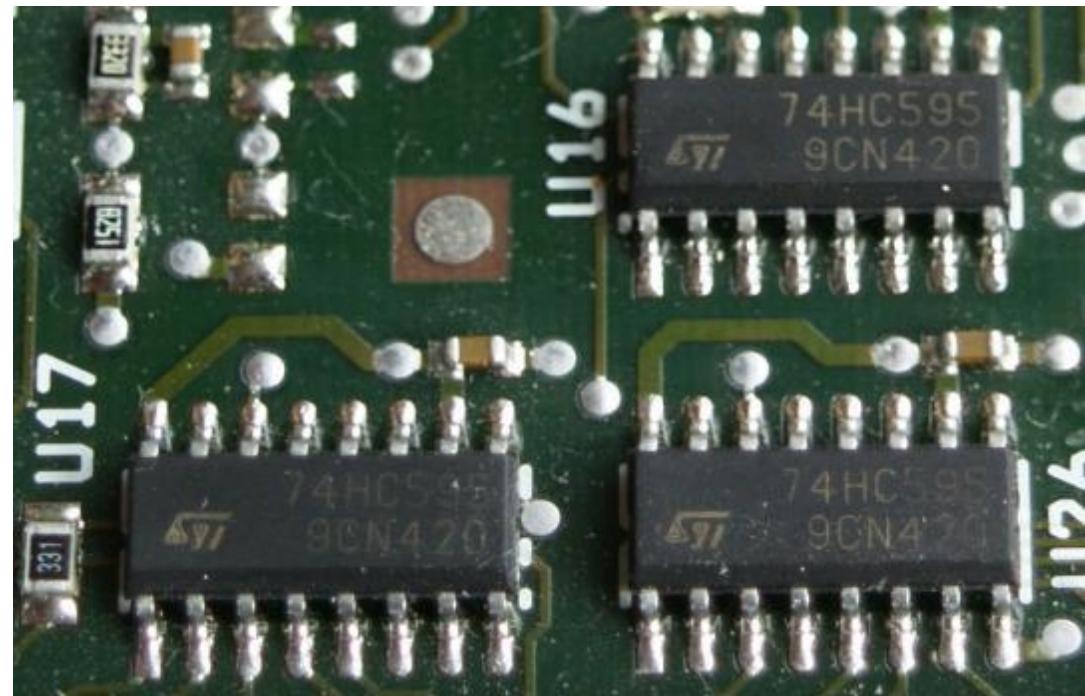
Standard Logic Chips

- TTL (Transistor-Transistor Logic) Technology:
 - The 74xxx-series
- CMOS Technology:
 - The 4xxx-series



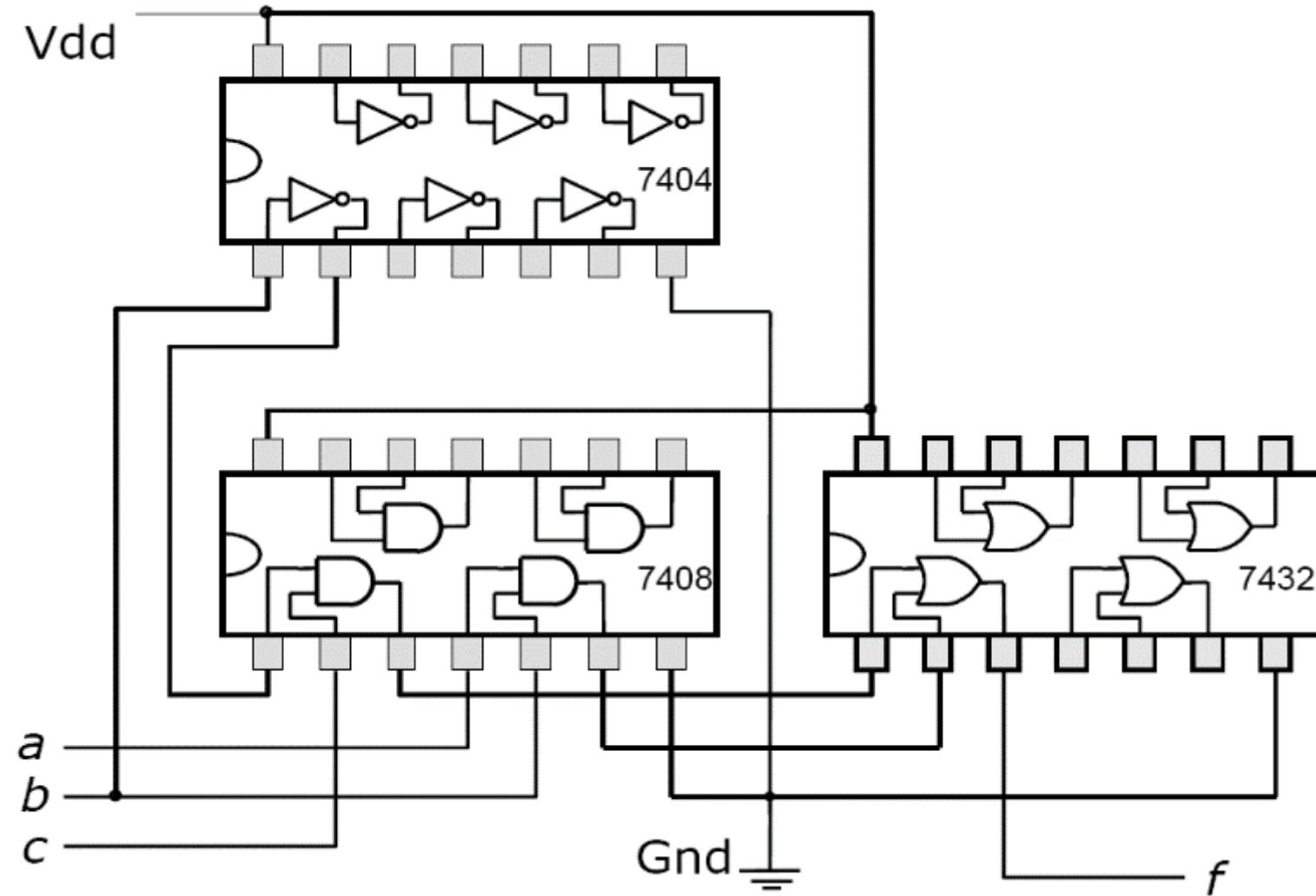
74000 Sub-series

- 74LS74: Low-power Schottky
- 74HCT74: High-speed CMOS
- 74HCT: 74LS TTL-compatible inputs
- SN74F00: Fast logic

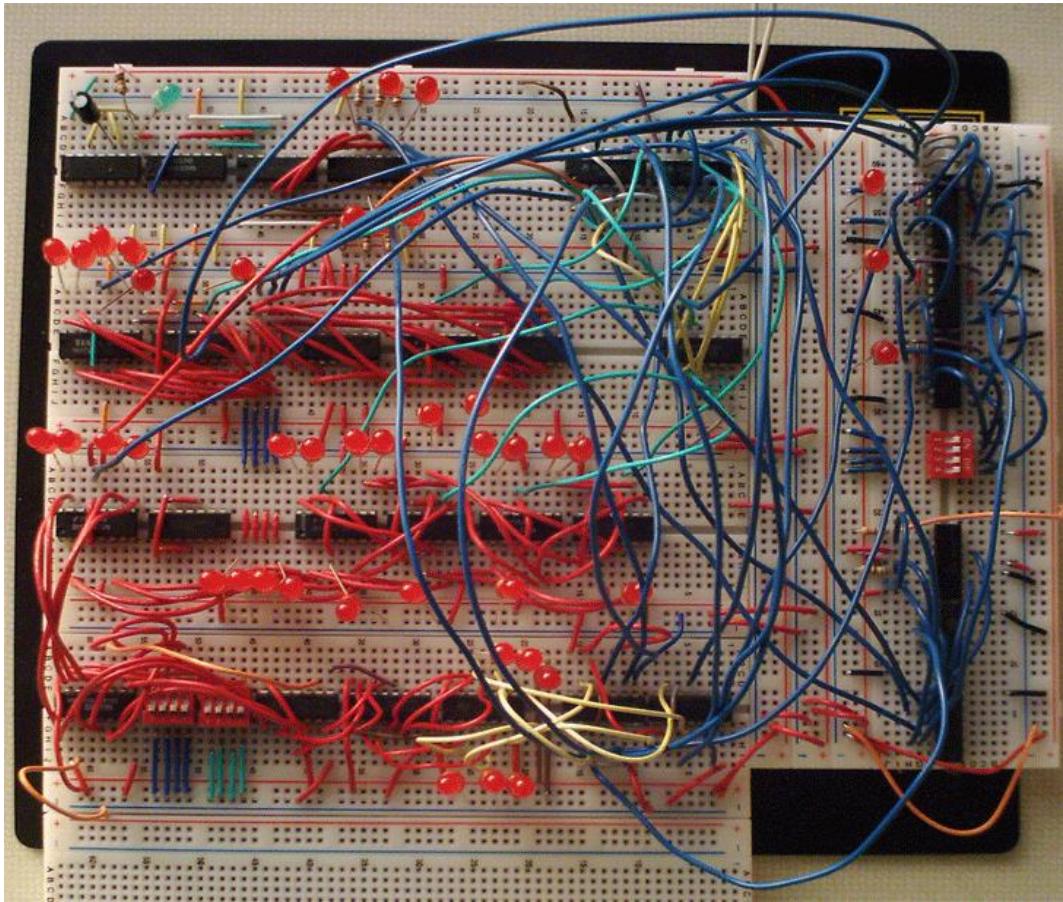


Design Example Using the 74000 Series

$$f(a,b,c) = ab + \bar{b}c$$



Design Example Using the 74000 Series



An Implementation of a 4-bit two register computer, including 6 CPU assembly instructions:
READ (read input), INCB (increment register B), MOVAB (move contents of register A to B),
MOVBA (move contents of register B to A), RETI (return from interrupt), JMP (jump).

Reference: http://en.wikipedia.org/wiki/7400_series

Programmable Logic Technologies

- **Basic Idea:** Logic functions can be realized in sum-of-product form.

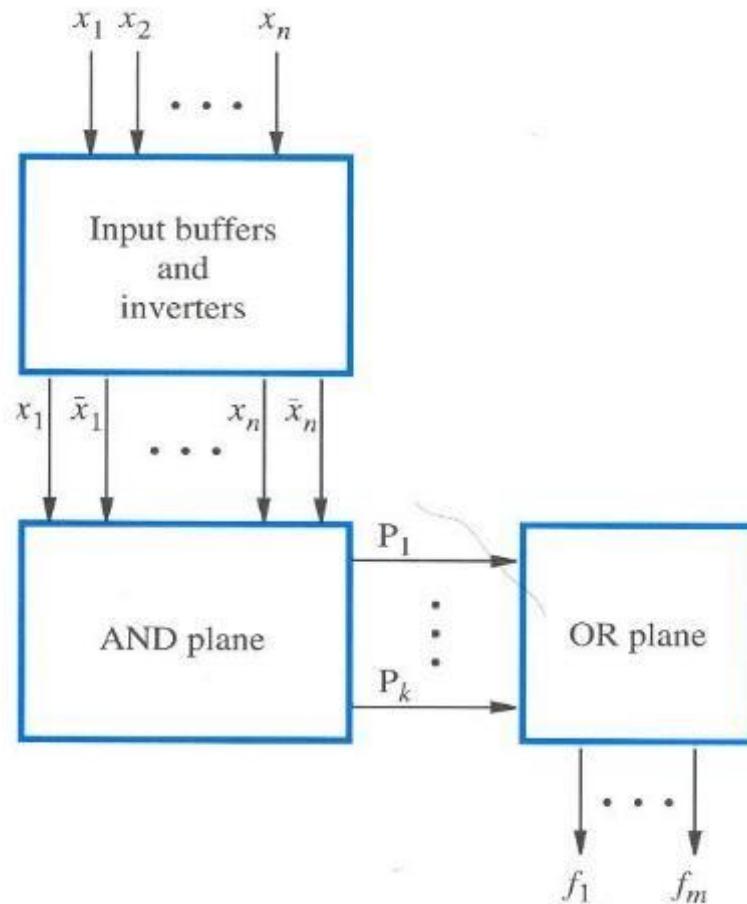
$$f(x, y, z) = \bar{x}y + x\bar{y}z$$

Technologies:

- Simple PLD (SPLD)
- Complex PLD (CPLD)

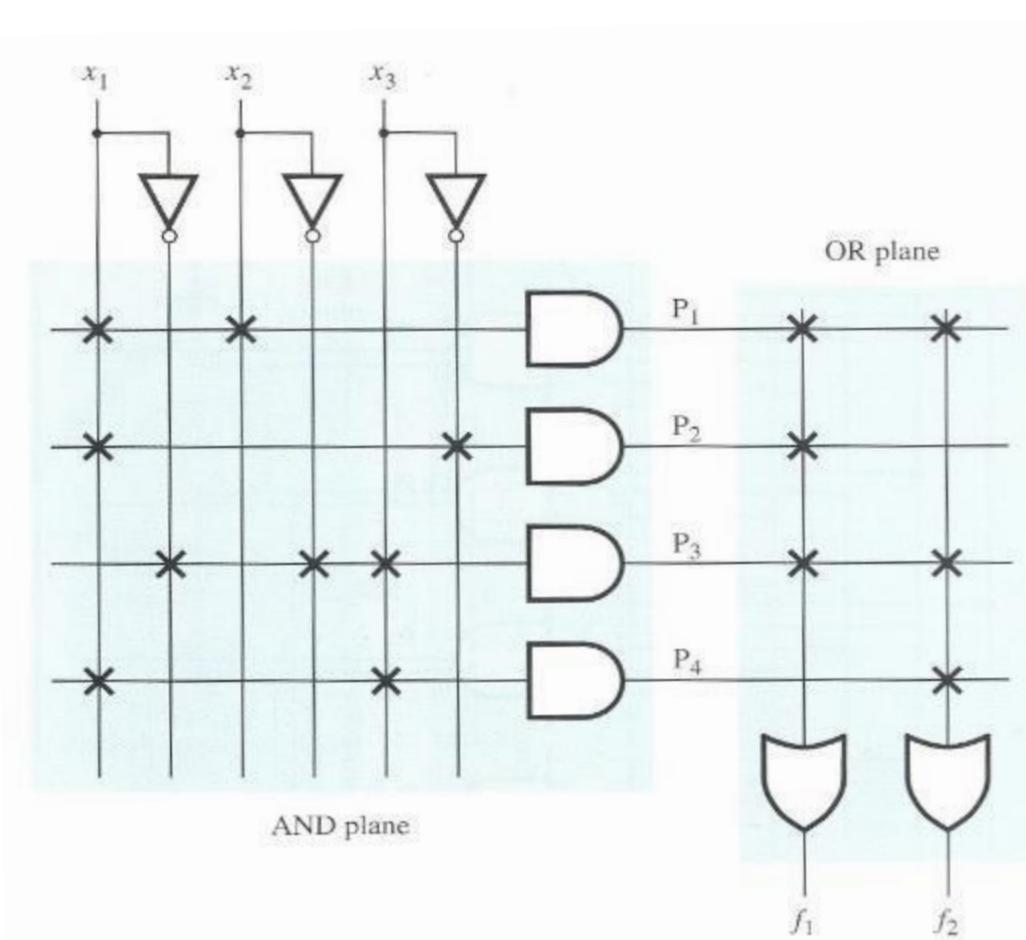
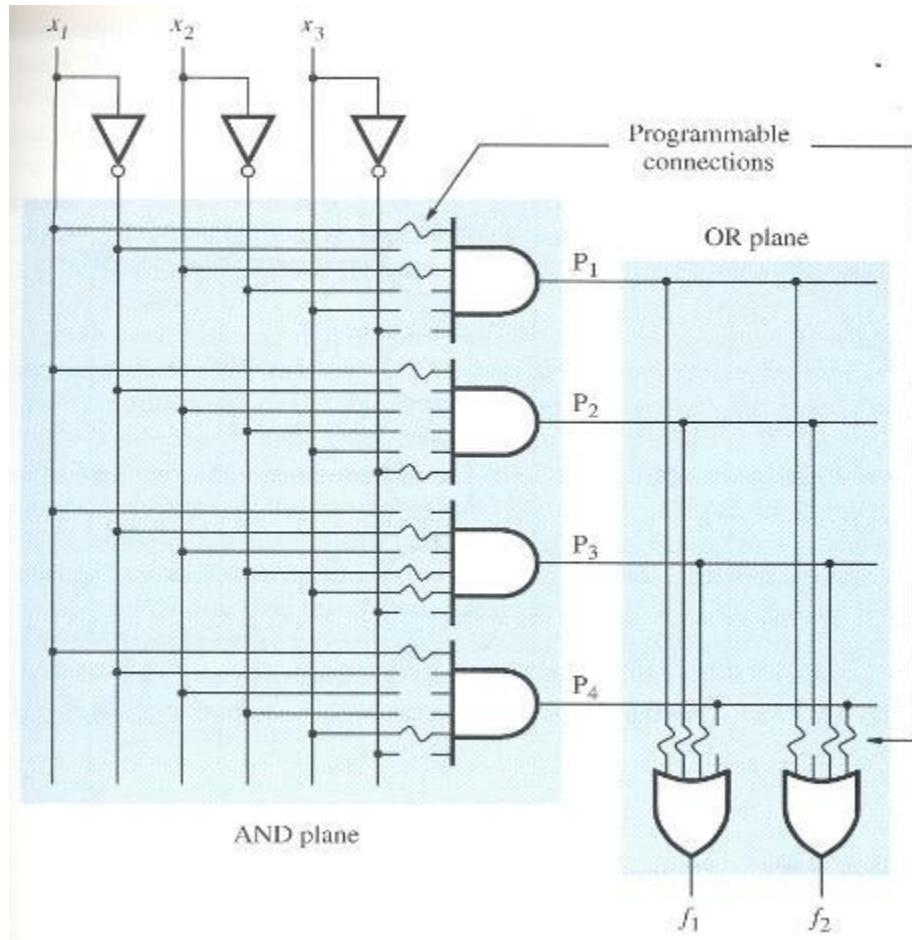
PLA (Programmable Logic Array) Technology

The basic concept: An arbitrary sum of product generator



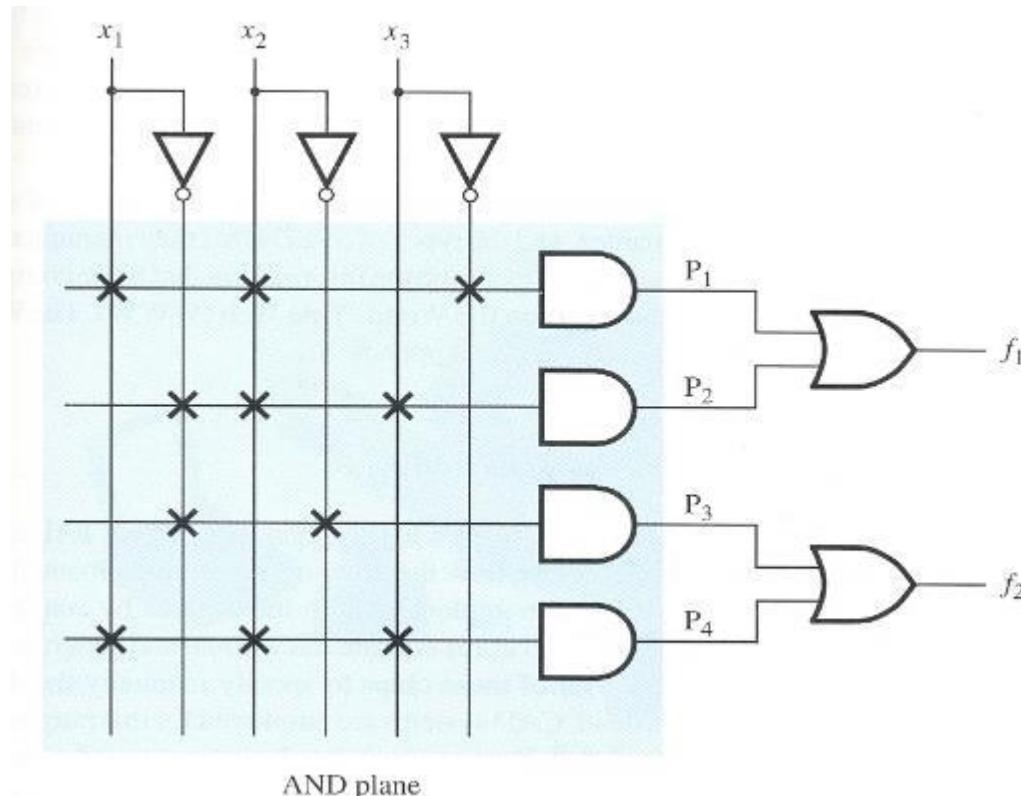
PLA (Programmable Logic Array) Technology

Example

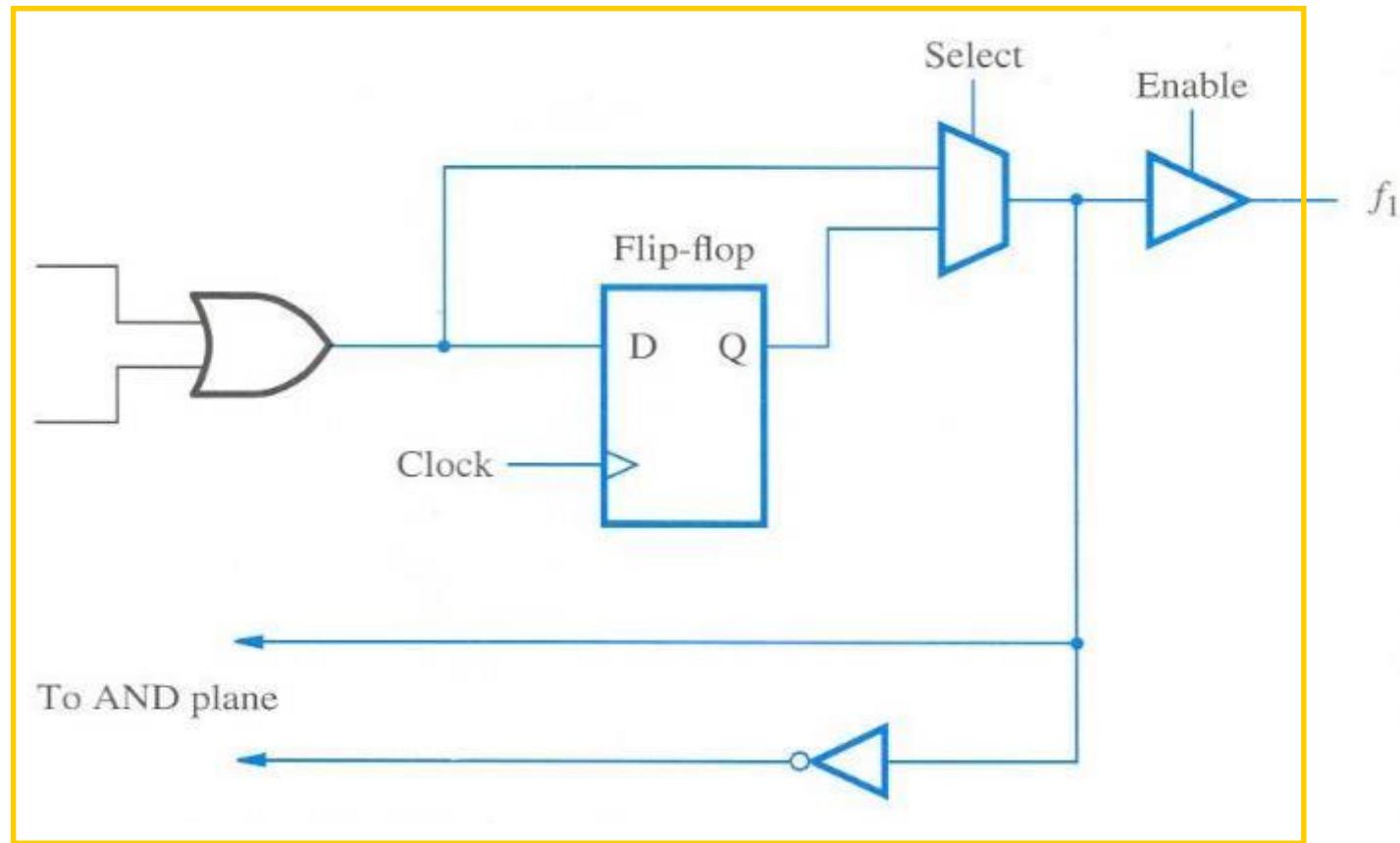


PAL (Programmable Array Logic) Technology

- PLA has both AND and OR programmable gates; but PAL has only programmable AND gates and the OR gates are fixed



PAL Extra Circuitry

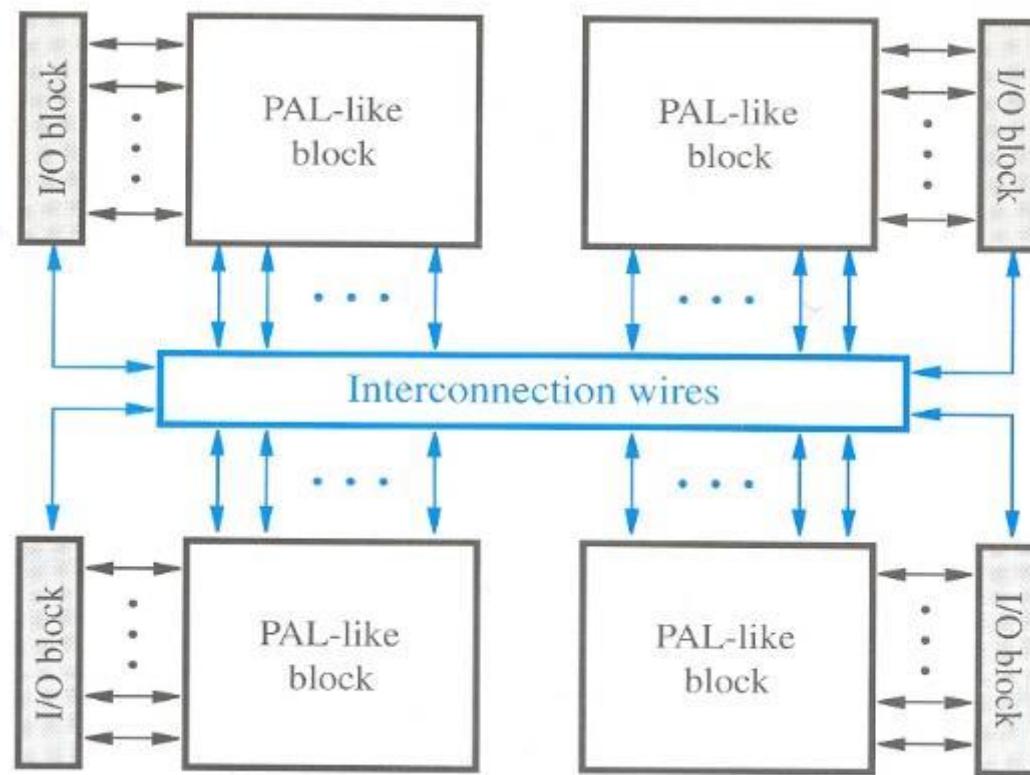


Macrocell

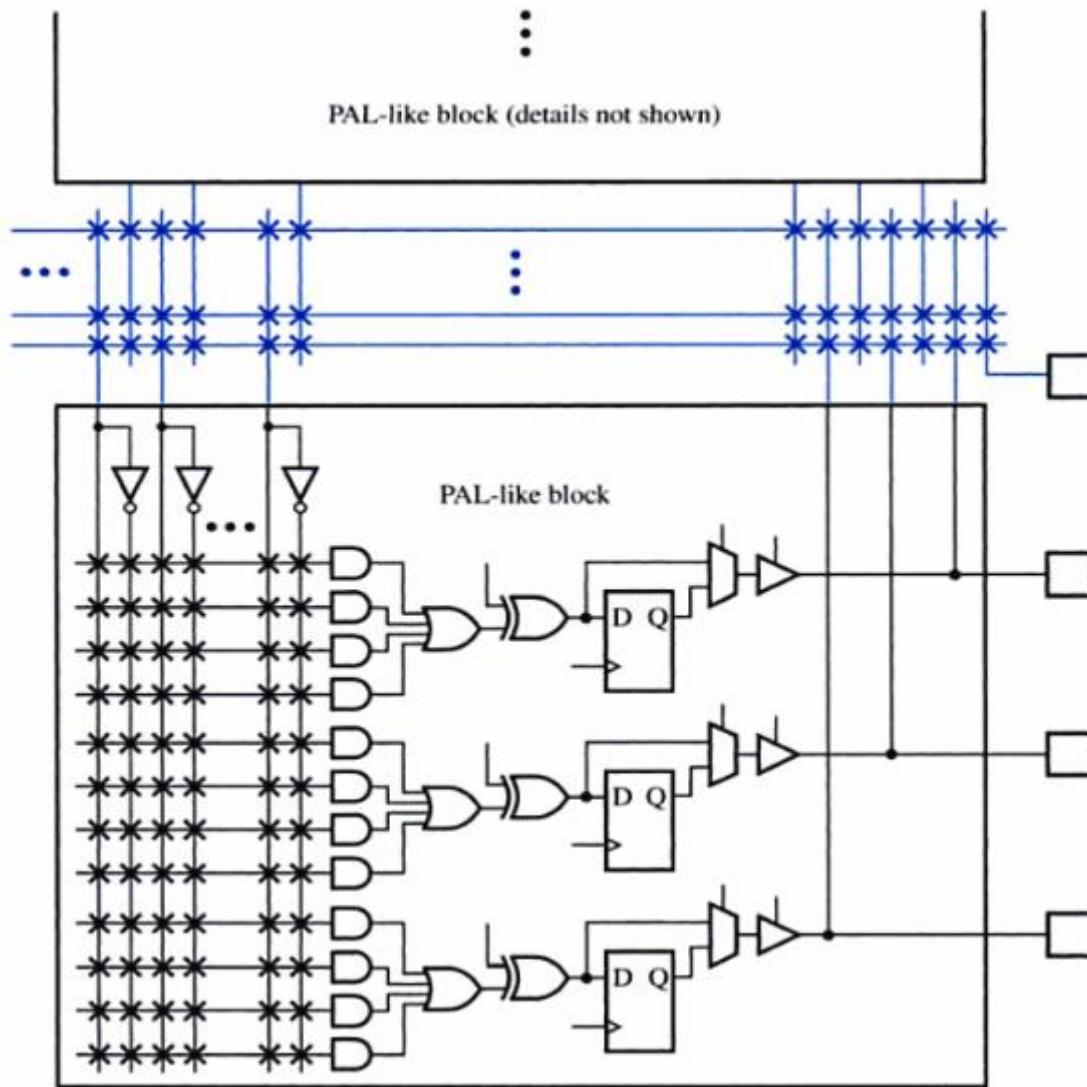
*Input – Output selection

CPLD (Complex Programmable Logic Device)

CPLDs can be considered as a set of PAL-like blocks with a set of reconfigurable interconnection network



CPLD (Complex Programmable Logic Device)



Equivalent Number of Gates

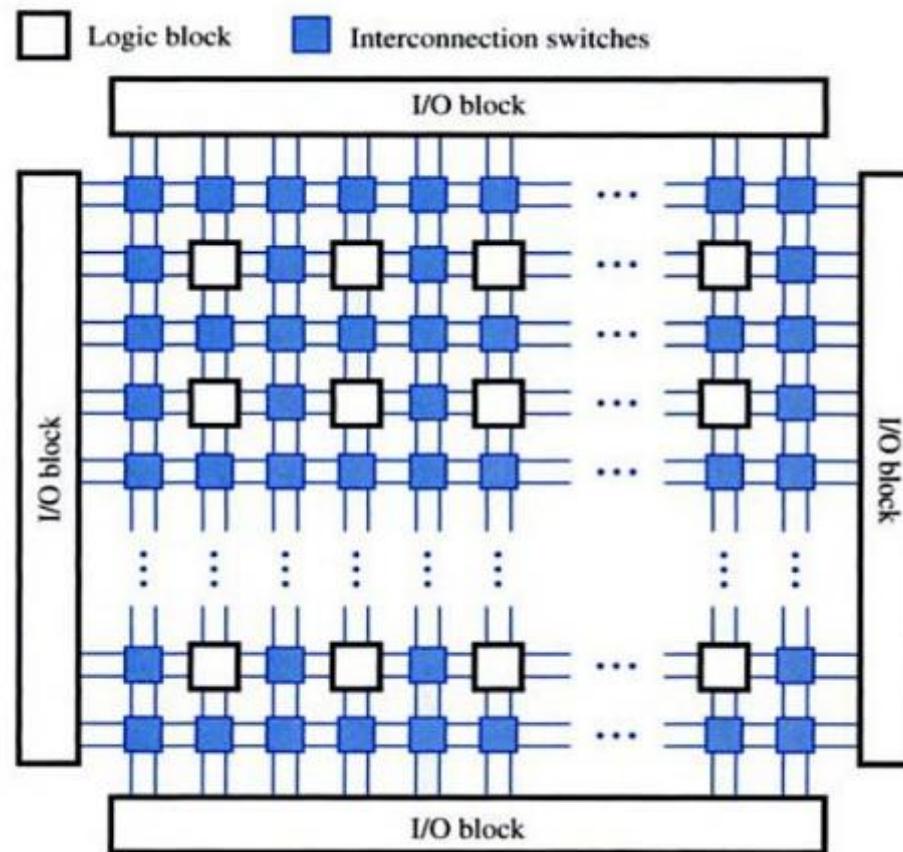
- We need a measure to compare the computation power of different PLDs:
- **Equivalent gates:** total number of two input NAND gates
- **Example:** If 1 Macrocell \approx 20 NAND gates, a 1000 Macrocell CPLD is roughly equivalent to 20,000 NAND gates

CPLD (Complex Programmable Logic Device)

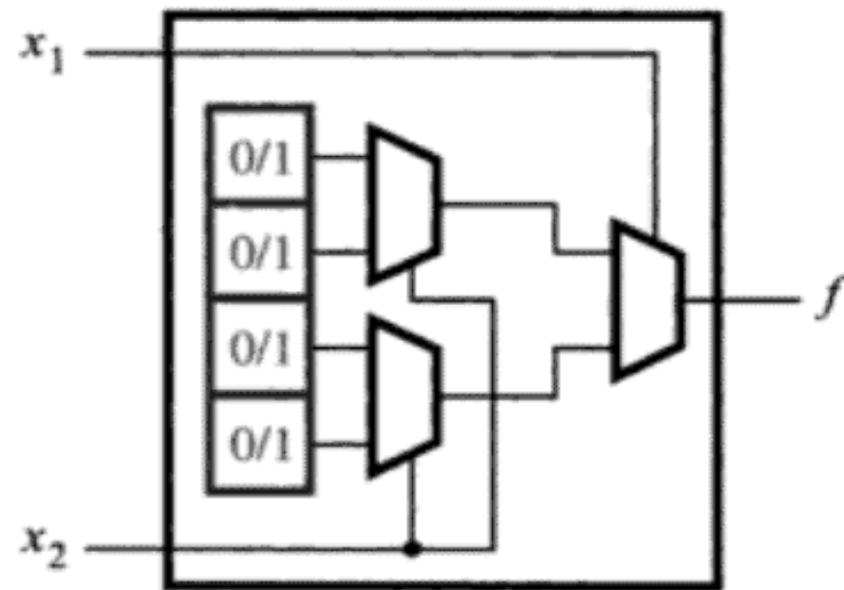


FPGA (Field-Programmable Gate Array)

- FPGAs are extensions of the idea of PROMs for logic circuit realization



FPGA Configurable Logic Blocks (CLB)



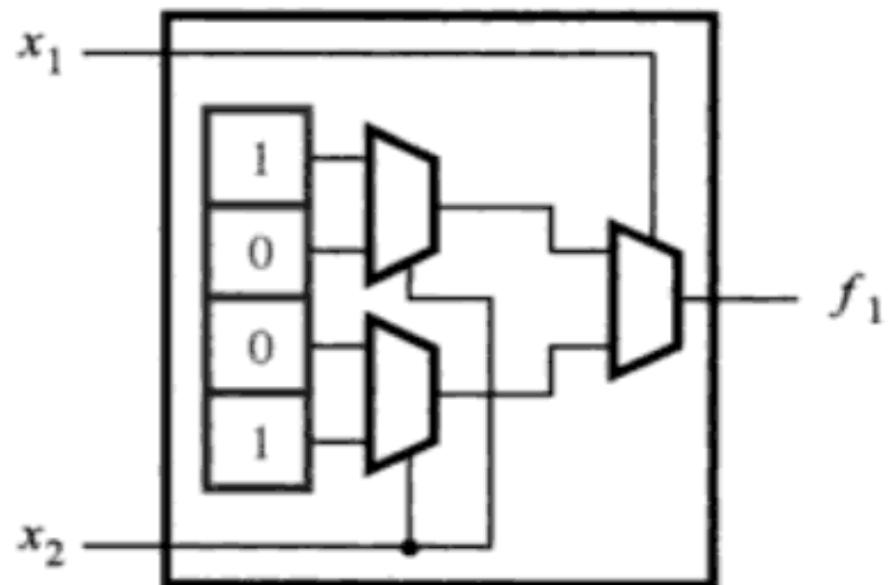
Two-input Look Up Table (LUT)

FPGA Configurable Logic Blocks (CLB)

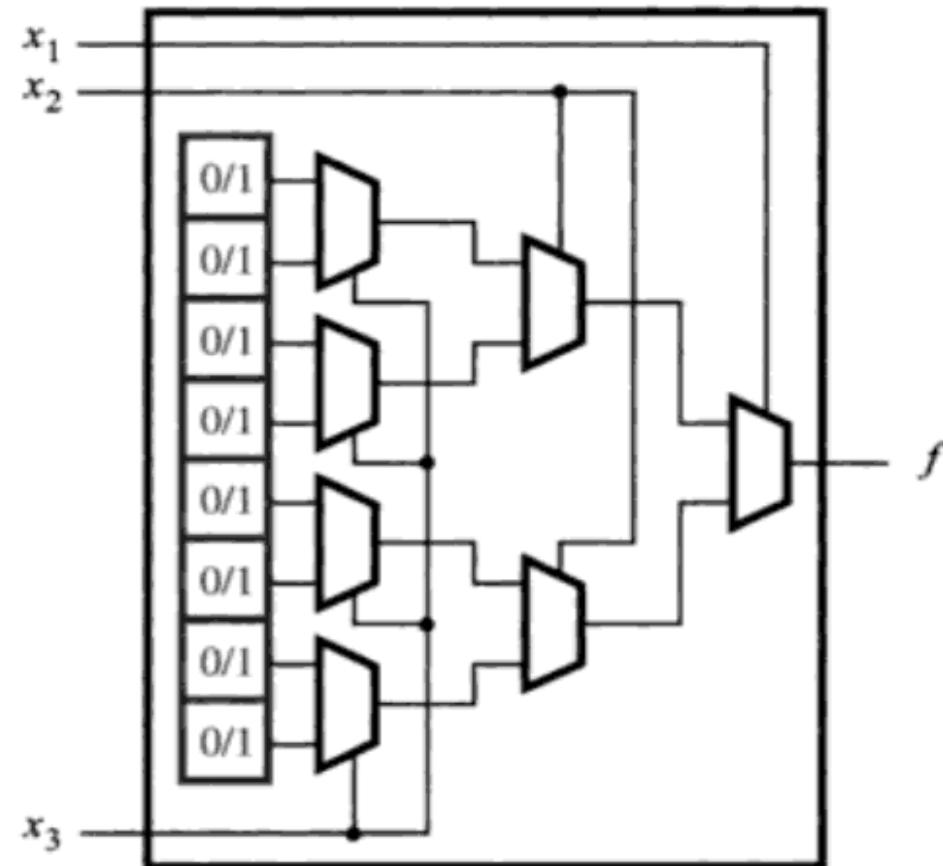
Example:

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

$$f_1 = \bar{x}_1 \bar{x}_2 + x_1 x_2$$

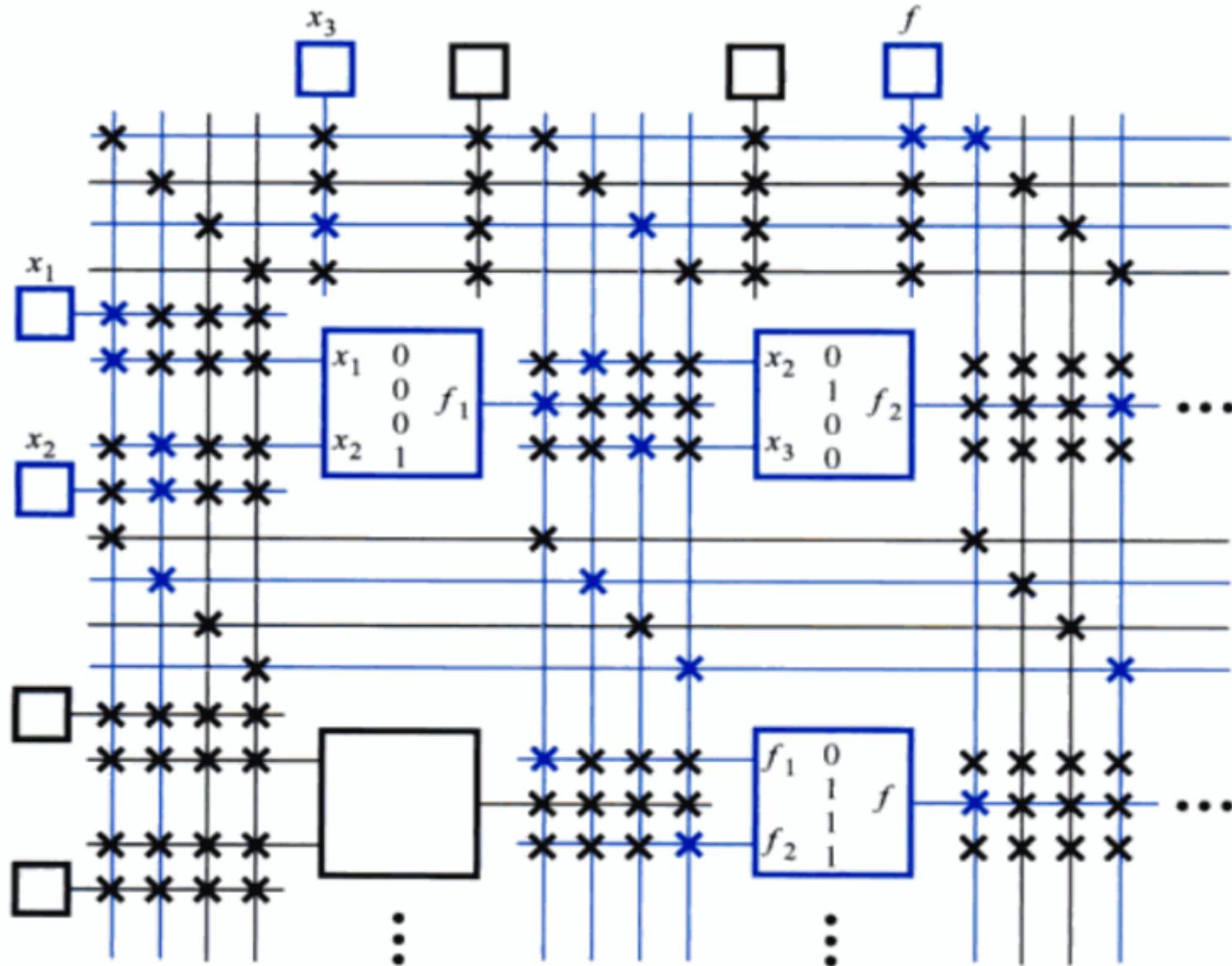


FPGA Configurable Logic Blocks (CLB)

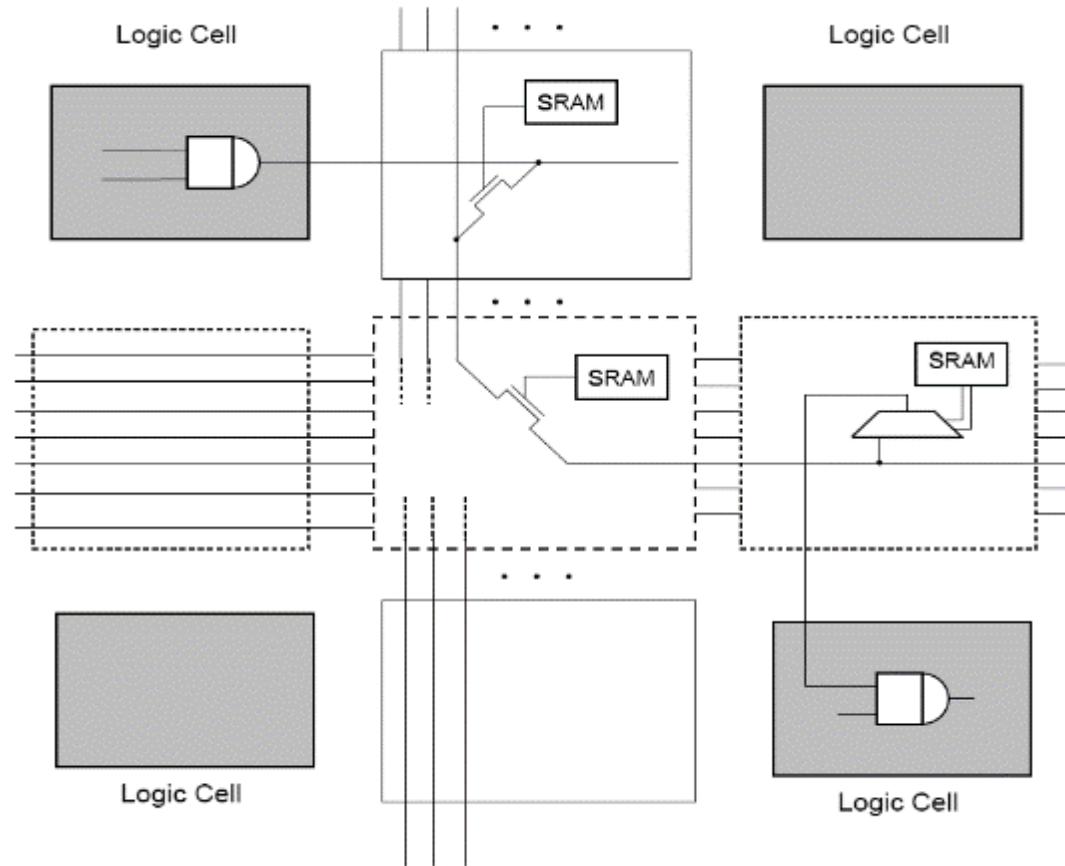


Three-input Look Up Table (LUT)

Programmed FPGA Scheme



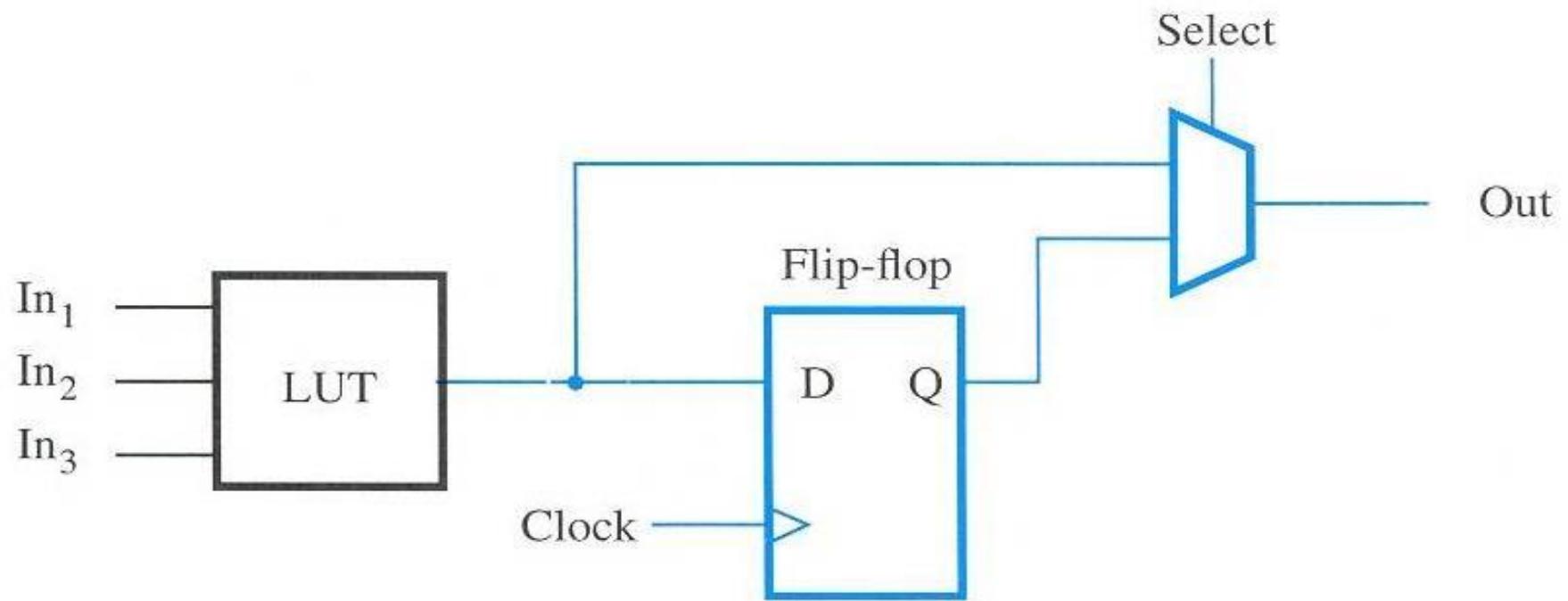
Programmable Switches



- Other switching technologies: Flash-based, Anti-fuse, etc.

FPGA Logic Block Extra Circuitry

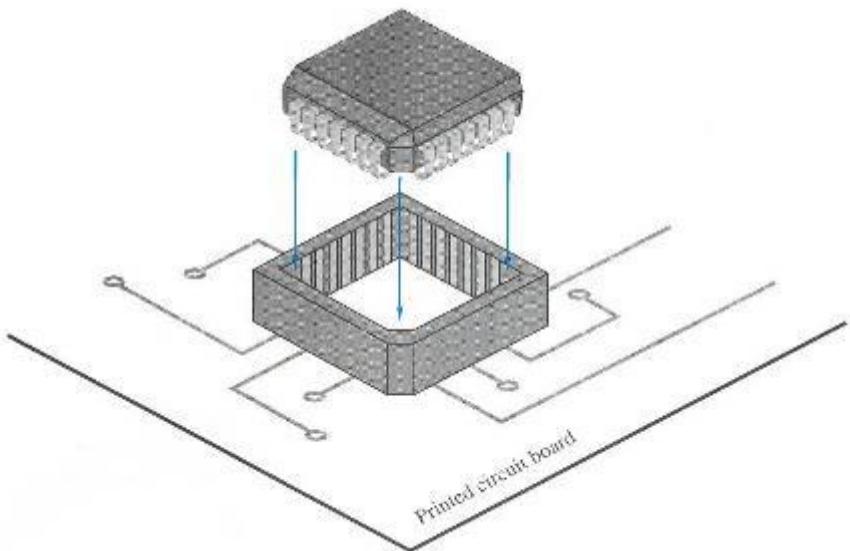
FPGA logic blocks require extra circuitry for sequential logic, routing, I/O interface, etc.



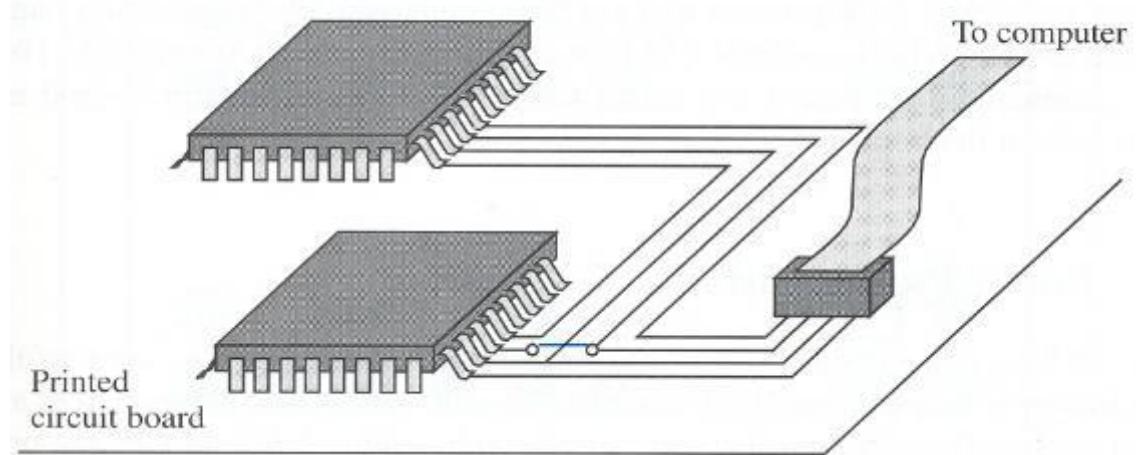
CPLD vs. FPGA

- CPLD keeps its contents without power, also known as *non-volatile*
- FPGA storage cells are *volatile* (lose their contents when power is switched off)

PLD Programming



(a) Off-board programming



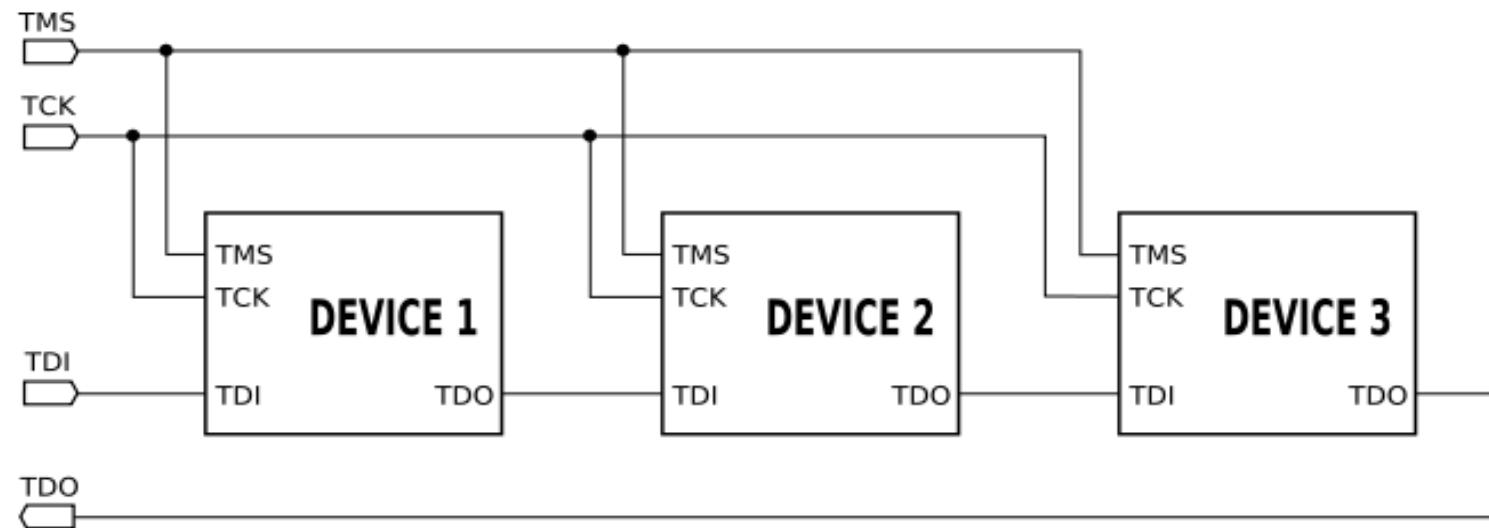
(b) On-board programming using JTAG*

* JTAG: Joint Test Action Group

JTAG

JTAG is a serial interface technology. The connector pins are:

- TDI: Test Data In
- TDO: Test Data Out
- TCK: Test Clock
- TMS: Test Mode Select
- TRST: Test Reset (optional)

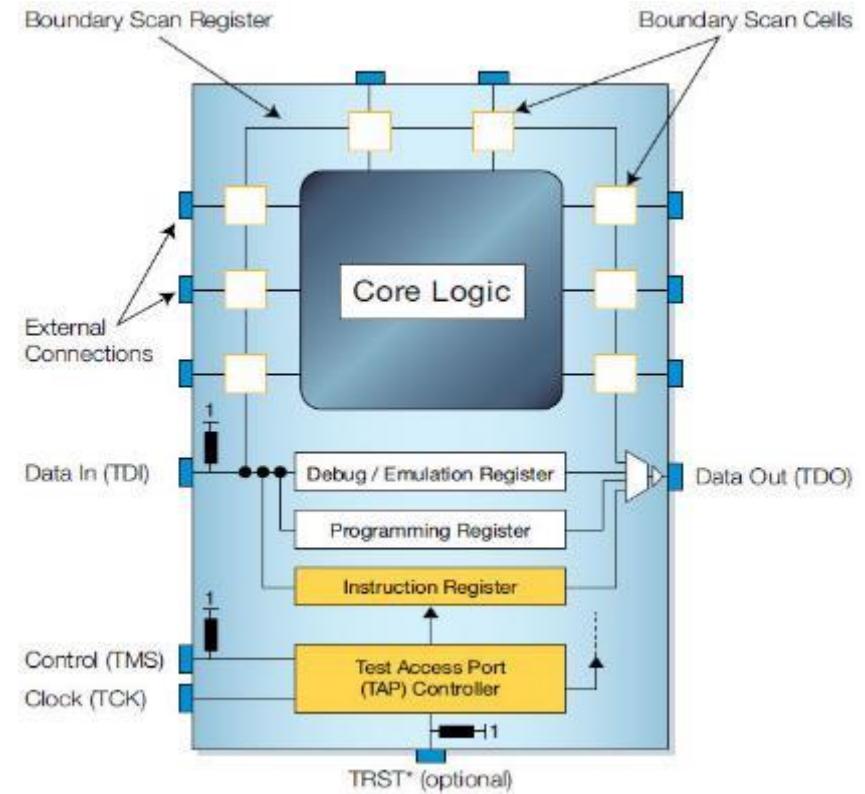
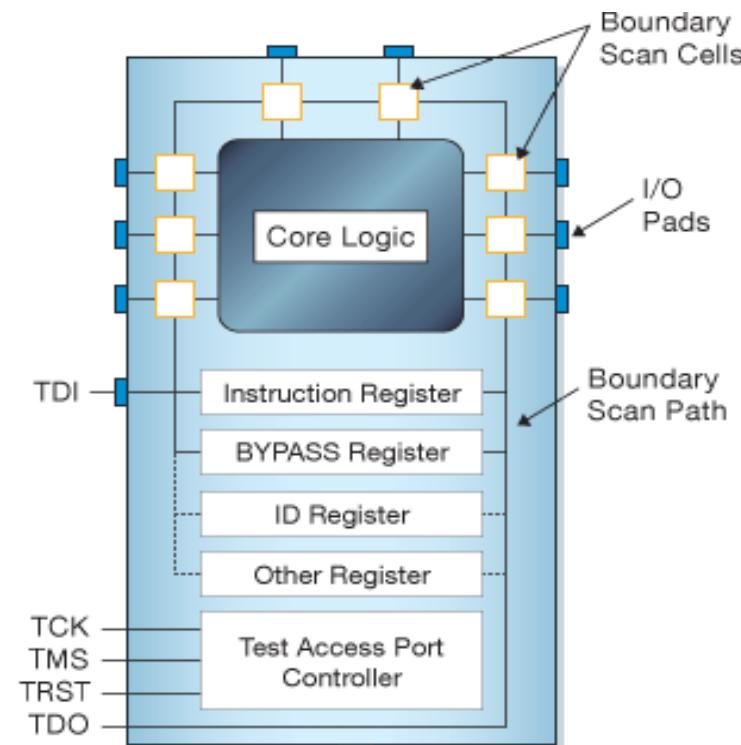


JTAG Daisy Chain

Other JTAG Applications

Examples:

- **Boundary Scan:** the ability to set and read the values on pins without direct physical access
- **Xilinx ChipScope Technology:** for in-system run-time debugging

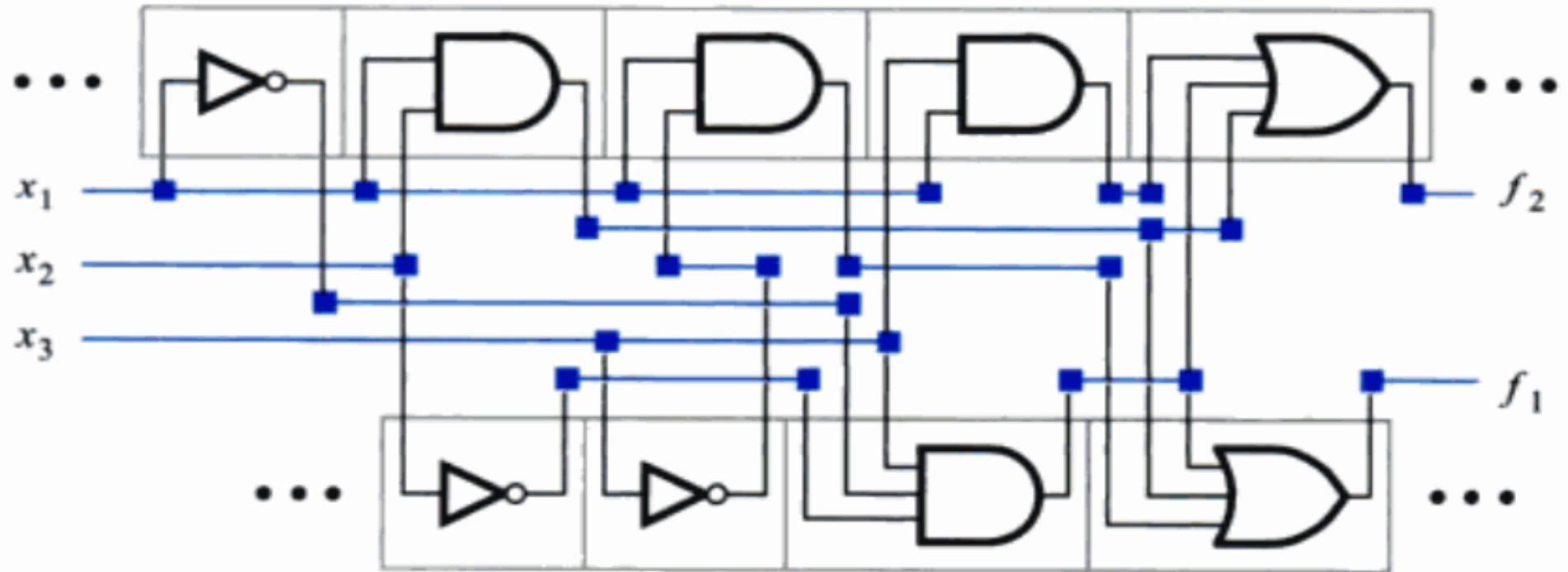


Ref: <https://www.xjtag.com/about-jtag/jtag-a-technical-overview>

ASIC (Application Specific Integrated Circuit)

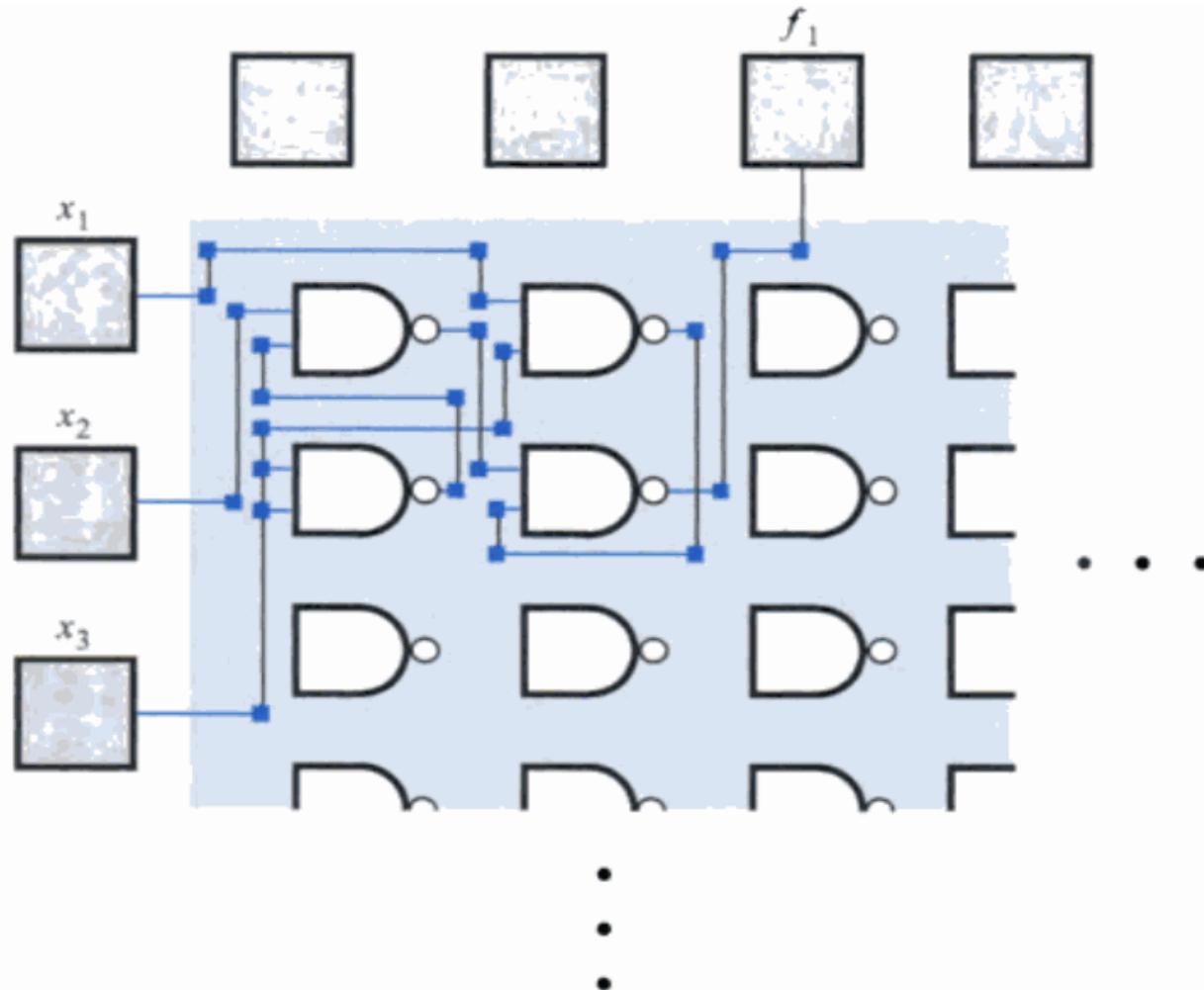
- In CPLD and FPGA programmable switches consume much space and reduce speed
- Alternatively, the *chip layout* can be totally customized; but is very expensive
- Compromise: The design may be simplified by using *standard-cell* or *gate-array* technologies

ASIC Standard-Cell Technology



Standard logic blocks are provided by manufacturers as libraries that may be connected

ASIC Gate-Array Technology



Only some parts of the chip are prefabricated

PLD Packages

- Plastic Leaded Chip Carrier (PLCC)



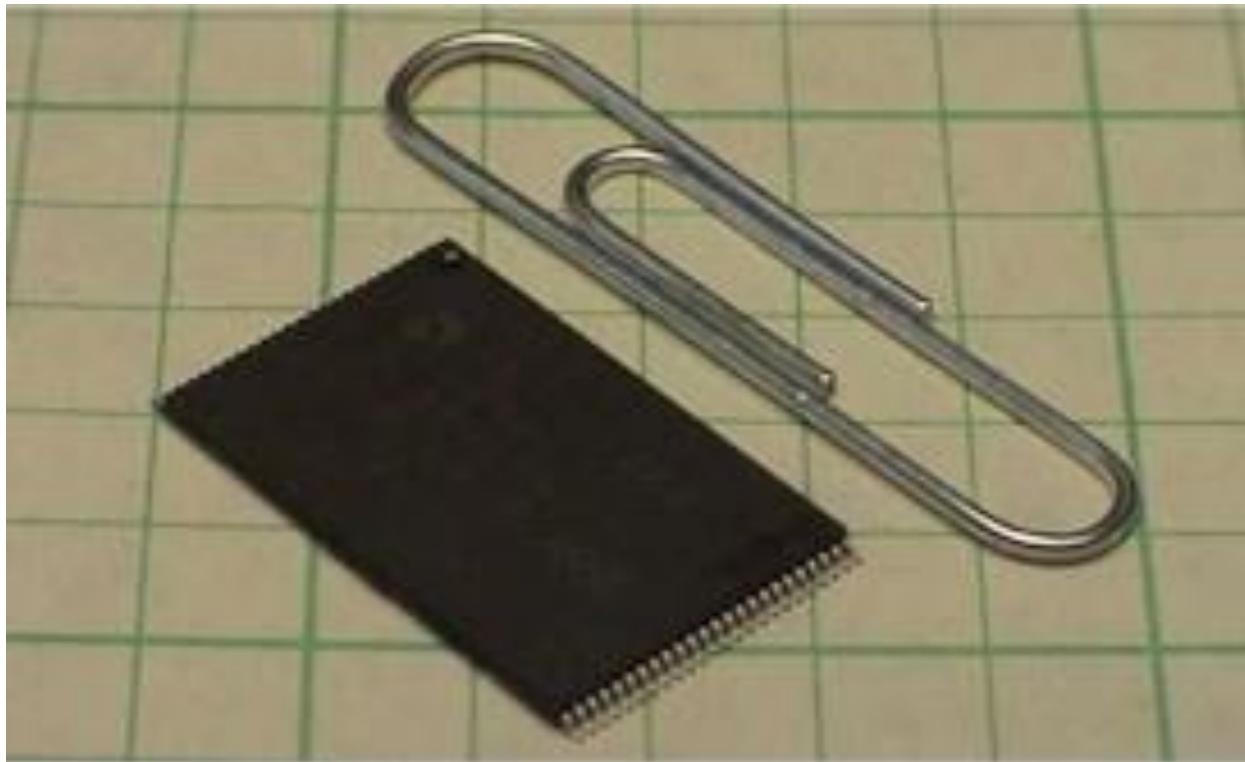
PLD Packages

- Small Outline Integrated Circuit (SOIC)
- Plastic Small Outline Package (PSOP)



PLD Packages

- Thin Small Outline Package



PLD Packages

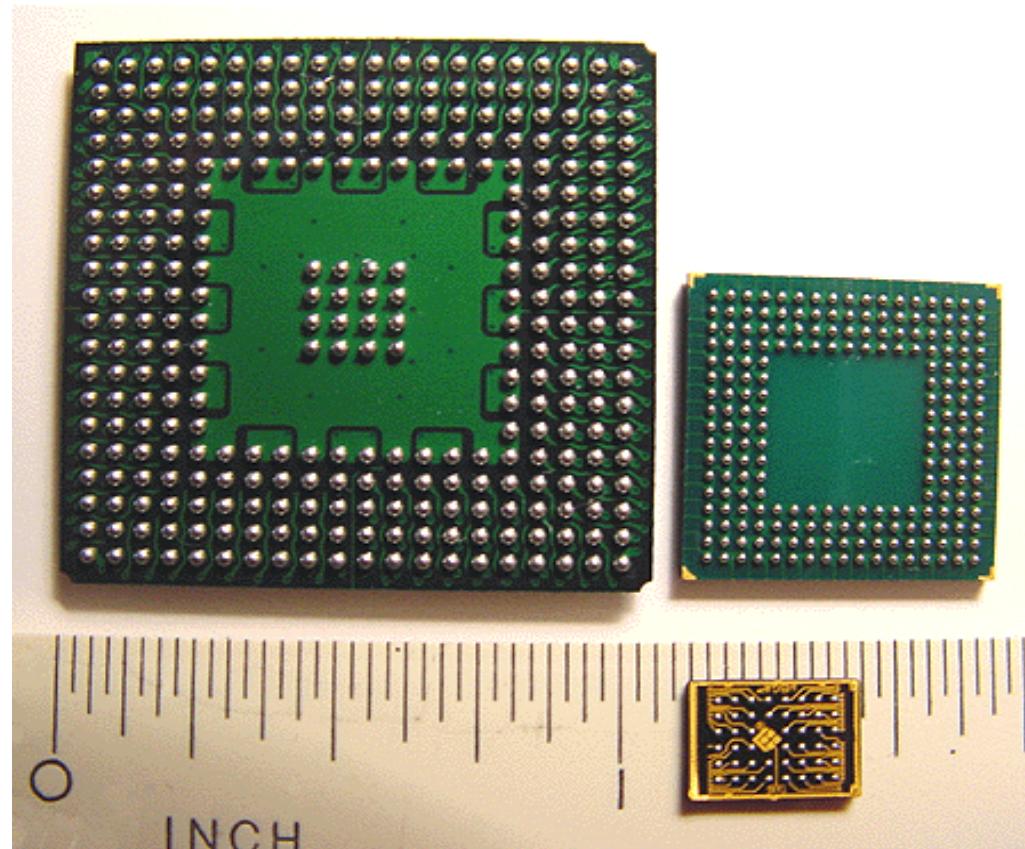
- Pin Grid Array (PGA)



PLD Packages

- Ball Grid Array (BGA)

From Computer Desktop Encyclopedia
© 2001 The Computer Language Co. Inc.



PLD Leading Companies

- Xilinx
- Altera
- Actel
- Lattice
- QuickLogic

Xilinx®

<http://www.xilinx.com/>

[Language](#) | [Documentation](#) | [Downloads](#) | [Contact Us](#)


XILINX®

[Sign in to access account](#)

[Advanced Search](#)

[Technology Solutions](#) [Products & Services](#) [Market Solutions](#) [Support](#) [Online Store](#) [About Xilinx](#)

Celebrating 25 Years of Innovation...



FPGA Inventor Ross Freeman Honored by
2009 National Inventors Hall of Fame



NATIONAL INVENTORS HALL OF FAME®, INC.
A Subsidiary of the National Inventors Hall of Fame Foundation, Inc.

[Learn More](#)

DO-254/ ED-80 Verification
Meet Industry Guidelines with Xilinx FPGAs

[View Now](#)

New eLab Video Series
See Industry Experts Discuss FPGA Topics

[View Now](#)

Technology Solutions	Products & Services	Market Solutions	Support
DSP	Silicon Devices	Aerospace & Defense	Download Center
Embedded Processing	Design Tools	Automotive	Support by Device
Connectivity	Intellectual Property	Broadcast	Documentation
Memory	Boards & Kits	Consumer	Forums
Signal Integrity	Design & Support Services	Data Processing and Storage	University Program
Power	Third Party Alliances	Industrial / Scientific / Medical	
Configuration	Training	Wired Communications	
Security		Wireless Communications	

 <p>FREE ISE™ WebPACK™ Software</p> <p>Download</p>	 <p>Buy. Evaluate. Develop.</p> <p>Search</p>	 <p>Investor Relations</p> <p>Access</p>
--	---	---

[Jobs](#) | [Events](#) | [Webcasts](#) | [News](#) | [Investors](#) | [Feedback](#) | [Legal](#) | [Privacy](#) | [Trademarks](#) | [Sitemap](#)

© 1994-2008 Xilinx, Inc. All Rights Reserved.

Altera®

<http://www.altera.com/>

The screenshot shows the official website for Altera Corporation. At the top, there's a navigation bar with links to Products, End Markets, Technology, Training, Support, About Altera, and Buy Online. The main banner features a colorful rainbow light effect and the text "Full Spectrum" followed by a subtitle: "A portfolio of transceiver FPGA and ASIC solutions as broad as your imagination." Below the banner are two buttons: "Get technical details" and "View the webcast". A section titled "New Arria II GX & Stratix IV GT FPGAs" is prominently displayed. The page is divided into three main sections: "Devices & Design Tools", "Learning Resources", and "Downloads & Licensing". Under "Devices & Design Tools", there are categories for FPGAs (Stratix IV, Arria II, Cyclone III, MAX II), CPLDs, and ASICs (HardCopy Series). "Learning Resources" includes Design Resources (Quartus II Design Software, Intellectual Property (IP), Development Kits, Reference Designs) and Technology Solutions (DSP, Embedded Processing, Transceivers, External Memory). "Downloads & Licensing" likely refers to software and licensing information. On the right side, there's a sidebar with links to the Download Center and Literature, a language selection dropdown, and sections for "What's New" (listing Altera's Stratix IV GT and Arria II GX FPGAs, Quartus II Software Version 9.0, and HardCopy IV ASICs as Product of the Year by EN-Genius Network) and "Upcoming Events" (a link to a webcast titled "Automate System Generation and IP Integration Using SOPC Builder"). At the bottom, there's a footer with links to various Altera services and a copyright notice: "Copyright © 1995-2009 Altera Corporation. All Rights Reserved."

- Altera was acquired by Intel® in 2015

Actel®

<http://www.actel.com/>

I Product Updates | Customer Portal | RSS

Google™

Home Company Products & Services Documentation Downloads Support

Now, more than ever, POWER MATTERS

REPLAY [» View Webcast](#) [» Download White Paper](#) [» Learn More](#)

ACTEL NEWS

- Financial Report** 02/03/09
Actel Announces Fourth Quarter 2008 Financial Results and Corporate Restructuring. [More »](#)
- Financial News** 01/27/09
Actel Corporation Announces Q4 FY2008 Conference Call. [More »](#)
- Product Award** 01/13/09
EDN Editors Include Actel's RTAX-DSP in Hot 100 Electronics Products of 2008. [More »](#)

.....

Q4 2008 Earnings Call Archive

PRODUCTS	TOOLS & SERVICES	SOLUTIONS	EVENTS & RESOURCES
<ul style="list-style-type: none"> > IGLOO Lowest Power FPGAs > ProASIC3 Low-Power FPGAs > Fusion Mixed-Signal FPGAs > RadTolerant FPGAs > Processors > Packaging 	<ul style="list-style-type: none"> > Design Software > Programming & Debug > Dev. Kits and Boards > Intellectual Property > Online Prototyping System > Design Services 	<ul style="list-style-type: none"> > Portable > Low Power > System Management & TCA > Medical > Automotive > Military & Aerospace 	<ul style="list-style-type: none"> > Webcasts & Training > Trade Shows > eZone Newsletter > Knowledge Base > Searches > Sales Offices

New White Paper
Power-Aware FPGA Design
[Read Now »](#)

New Product Catalog
List of products, services, and package offerings
[Read Now »](#)

SmartPower Webcast
Analyze and reduce power consumption
[View Now »](#)

Copyright © 1995 - 2009 Actel Corporation, Mountain View, CA 94043-4855 USA.
[Site Map](#) | [Terms and Conditions of Use](#) | [Privacy Policy](#)

Lattice®

<http://www.latticesemi.com/>

The screenshot shows the homepage of the Lattice Semiconductor Corporation website. At the top, there is a navigation bar with links for Home, Products, Solutions, Support, Documents, Downloads, Sales, Store, and About Us. A search bar is located at the top right, along with flags for different countries. A banner on the left highlights the "LatticeECP2M - The First Low-Cost FPGA with 3Gbps SERDES". Another banner on the right promotes "FREE Webcast Wireless Solutions with Lattice FPGAs" and encourages users to "Register Today!". Below these banners, there are sections for "Key Solutions/Products", "Events", "News/Articles", and "Quicklinks". A testimonial from Dr. Philipp Tomxich is displayed in a box. At the bottom, there are links for Legal, Privacy Policy, Press, Careers, Investor Relations, and Contact Us.

Lattice Semiconductor Corporation

New Account | Sign In | Search |

Home | Products | Solutions | Support | Documents | Downloads | Sales | Store | About Us

LatticeECP2M
The First Low-Cost FPGA with 3Gbps SERDES

3.125 Gbps SERDES • Up to 95K LUTs • DSP Blocks • PLLs/DLLs

Learn More

Key Solutions/Products

- **Wireless** - Ready to Use Wireless Solutions
- **PCI Express** - Low Cost Development Kit
- **Ethernet** - End-to-end System Solutions
- **LatticeECP2M** - Low Cost FPGA with SerDes
- **LatticeXP2** - True 90nm Flash Based FPGA
- **MachXO** - Crossover PLD: Best of CPLD and FPGA
- **ispMACH 4000ZE** - Ultra Low Power, Low Cost CPLD
- **Design Software** - Download the Latest Version

Events > More

- Synplify/Pro Design Seminar - Online
- Verilog Training Classes - Online

Webcasts > More

- Non-Volatile FPGAs for Automotive Systems
- Lattice® and Aldec®: Quick Timing Closure
- FPGA Power Calculation Techniques

News/Articles > More

- Michael G. Potter Appointed Corporate Vice President and CFO
- Fourth Quarter and Year End Financial Results
- 15 Million MachXO PLDs Shipped
- Q4 2008 Conference Call Announced

RSS

Quicklinks

- Software Licensing
- Data Sheets
- Forums
- IP Cores
- Engineers' Blog
- What's New

Newsletter Sign-up

Subscribe to email newsletters, updates and more

Your Email Address

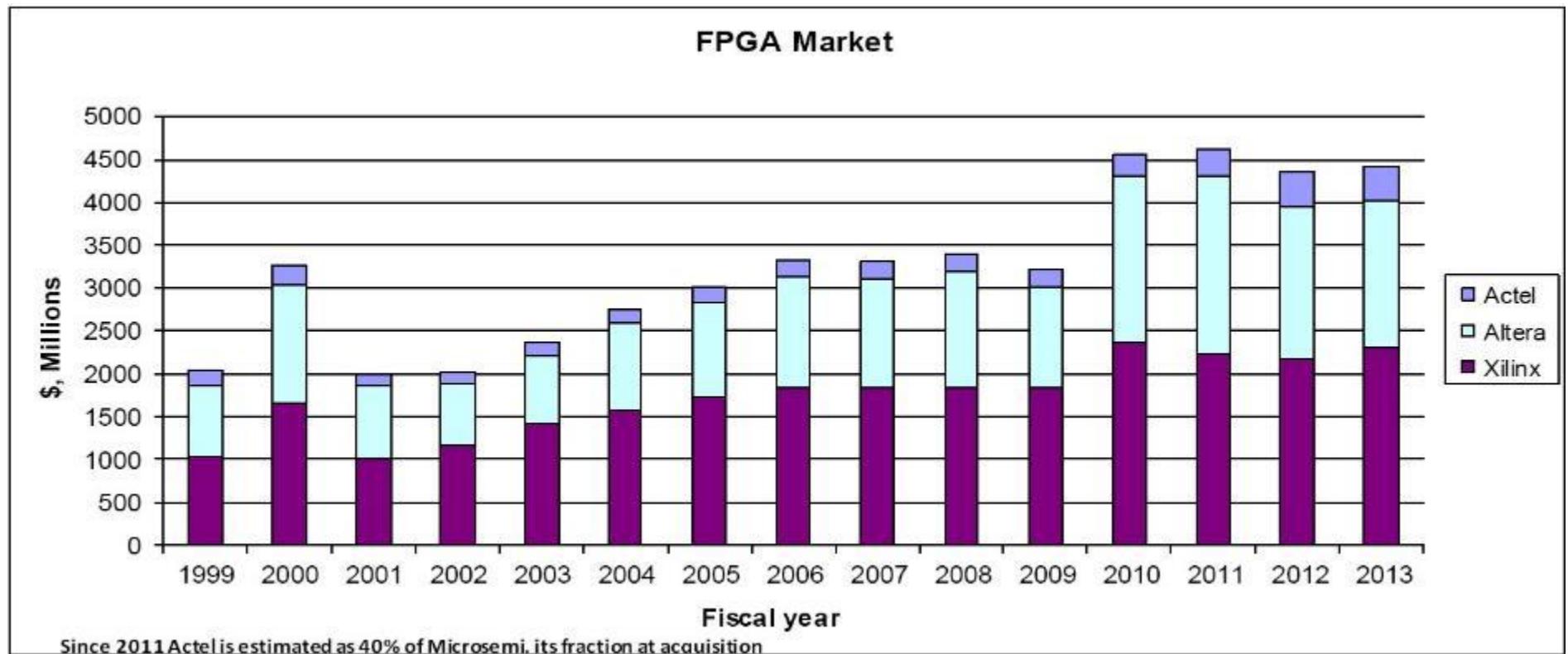
Legal | Privacy Policy | Press | Careers | Investor Relations | Contact Us | © Lattice Semiconductor Corporation 2009

QuickLogic®

<http://www.quicklogic.com/>

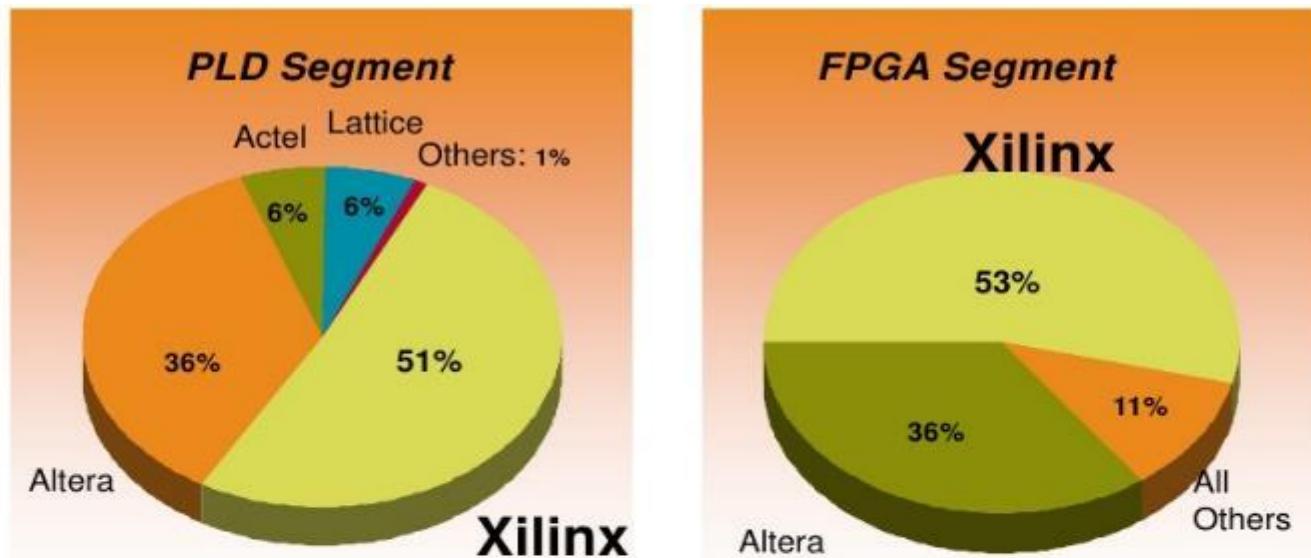
The screenshot shows the QuickLogic website homepage. At the top, there's a navigation bar with links for Solutions, Platforms, Technologies, Support, and Corporate. A search bar is also present. The main banner features the SPIDA logo and the text "Smart Programmable Integrated Data Aggregator" followed by "...for a superior user experience". To the right of the banner is a "Click for details" button with a hand cursor icon. Below the banner, there are several sections: "Platforms" (ArcticLink®, ArcticLink II, PolarPro®, PolarPro II), "Technologies" (VEE™, SPIDATM, ViaLink®), "Support" (Support Center, Mature Products, Tools, Packages, Documents), "Corporate" (Corporate Overview, Press Room, Investors, Jobs, Contact Us), and "CSSP What are CSSPs? CSSPs?". The "CSSP" section includes a sub-section "CSSP Solutions by Market" with links to Smartphone, Enterprise PDA, Personal Media Player, Portable Navigation Device, Wireless Datacard, and UMPC/PCJMD. Another "CSSP Solutions by Application" section includes links to Storage, Videofimaging, Wireless Network, Wired Network, and Security & Custom Options. On the right side, there's a "Press Release" section with links to a press release from Tuesday, February 10, 2009, and another from Monday, February 09, 2009. There's also a "Press Release Archive" and an "Events" section featuring Mobile World Congress, Needham Growth Conference Webcast, and The AEA Classic Financial Conference Webcast. At the bottom, there are links for Home, Stores, Sales Offices, Contact Us, Investors, Terms of Use, Privacy Policy, and a copyright notice for QuickLogic Corporation.

The FPGA Market (2013)



Reference: <http://www.eetimes.com/>

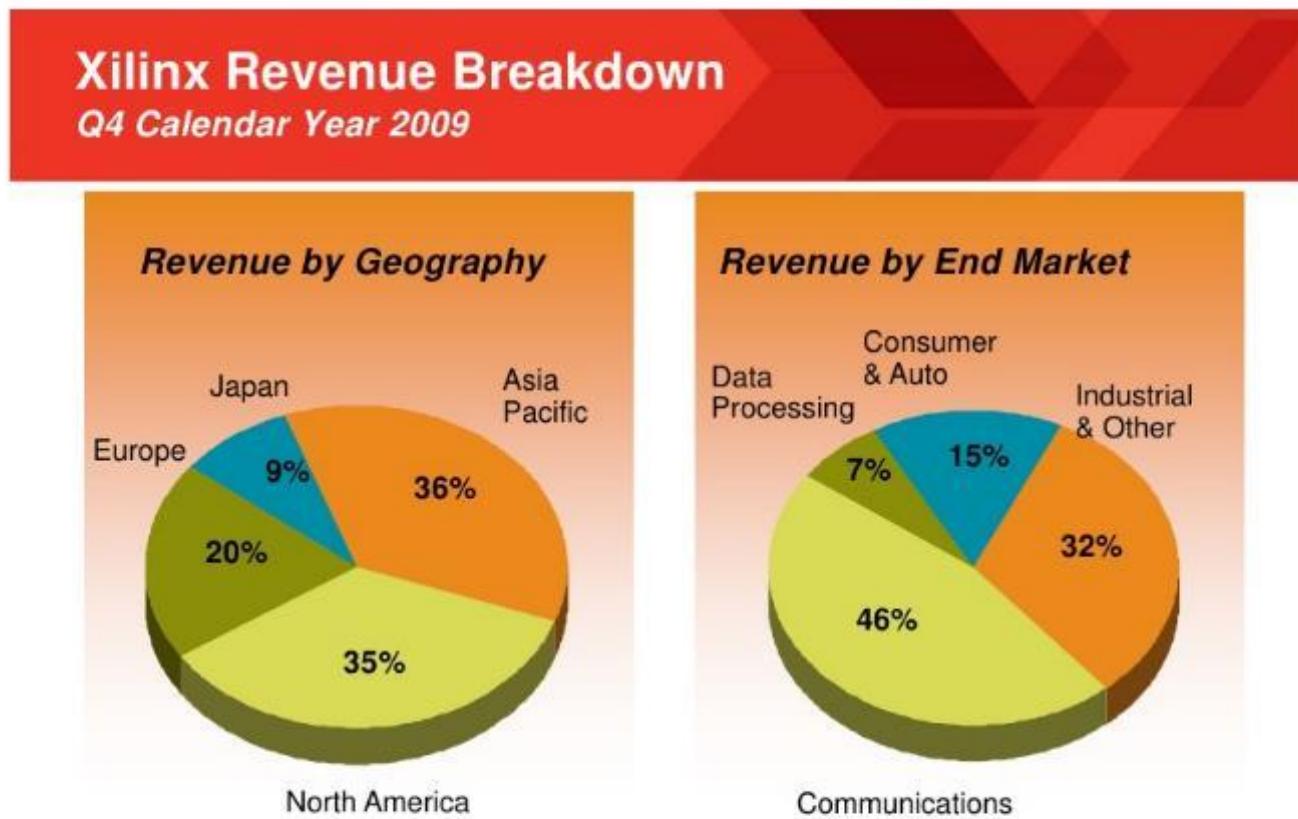
The Programmable Market Share (2009)



Xilinx revenues are greater than all other pure-play PLD companies combined.

Source: Company reports
Latest information available; computed on a 4-quarter rolling basis

Xilinx Revenue Breakdown (2009)



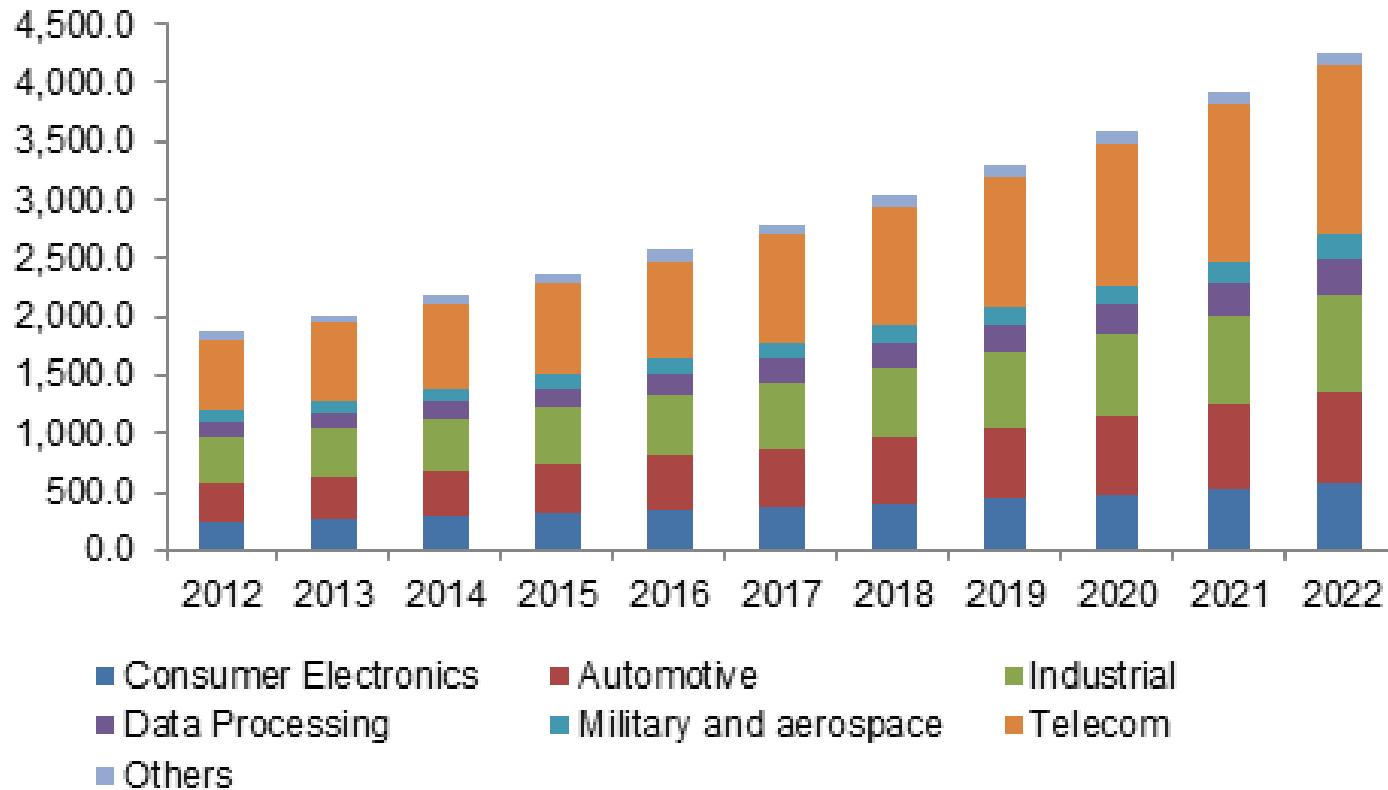
Source: Xilinx, Inc.

Page 7

 XILINX.

Reference: www.xilinx.com

Asia Pacific FPGA Market Size by Application, 2012-2022 (USD Million)



Reference: <https://www.gminsights.com/industry-analysis/field-programmable-gate-array-fpga-market-size>

References:

- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2003, Chapter 3
- B. Zeidman, *Designing with FPGAs & CPLDs*, CMP Books, 2002, Chapter 1

FPGA INTERNAL ARCHITECTURE

FPGA Internal Architecture

Current FPGA devices consist of:

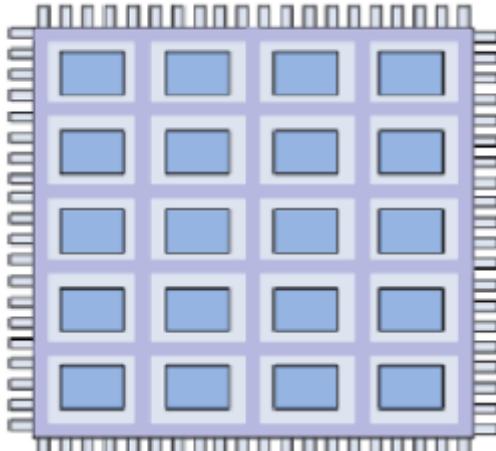
- Configurable logic
- Interconnect network
- Device-dependent peripherals and IP cores

Typical FPGA Architectures

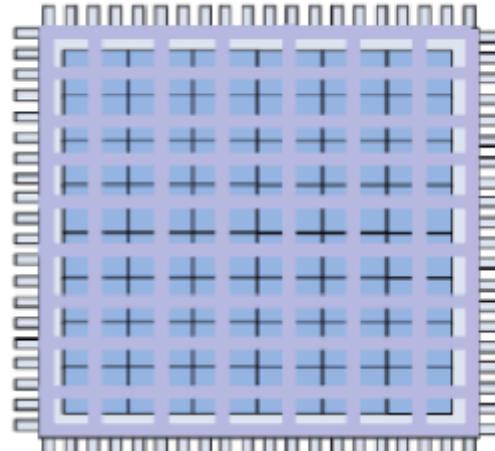
(an academic classification)

- Fine Grained (*homogeneous*)
 - Medium Grained
 - Coarse Grained (*heterogeneous*)
-
- ✓ From top to bottom the logic blocks become more complex and advanced.
 - ✓ **Node-Based Reconfigurable Architectures:** Imagine a network of computers and programmable devices, which can be reconfigured on-demand
 - ✓ Current FPGA architectures are considered **medium grain** in this classification

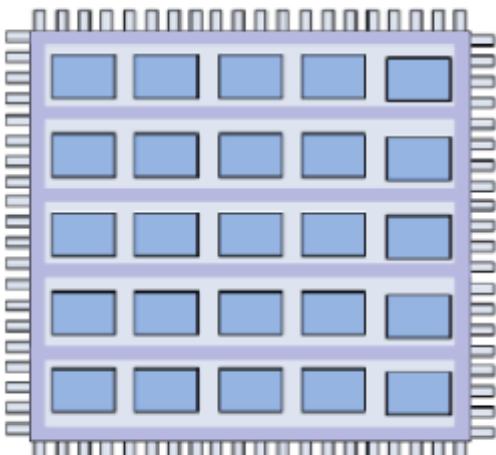
Basic FPGA Architectures



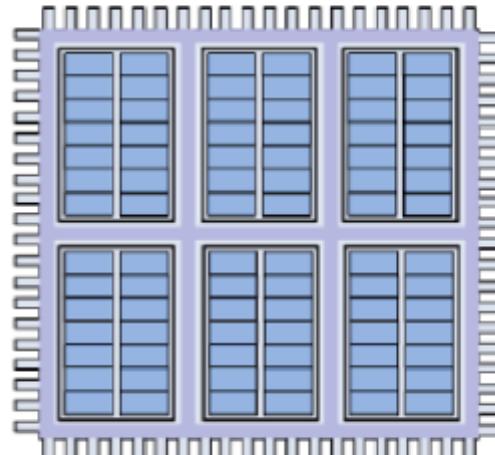
Symmetrical array



Sea of gates

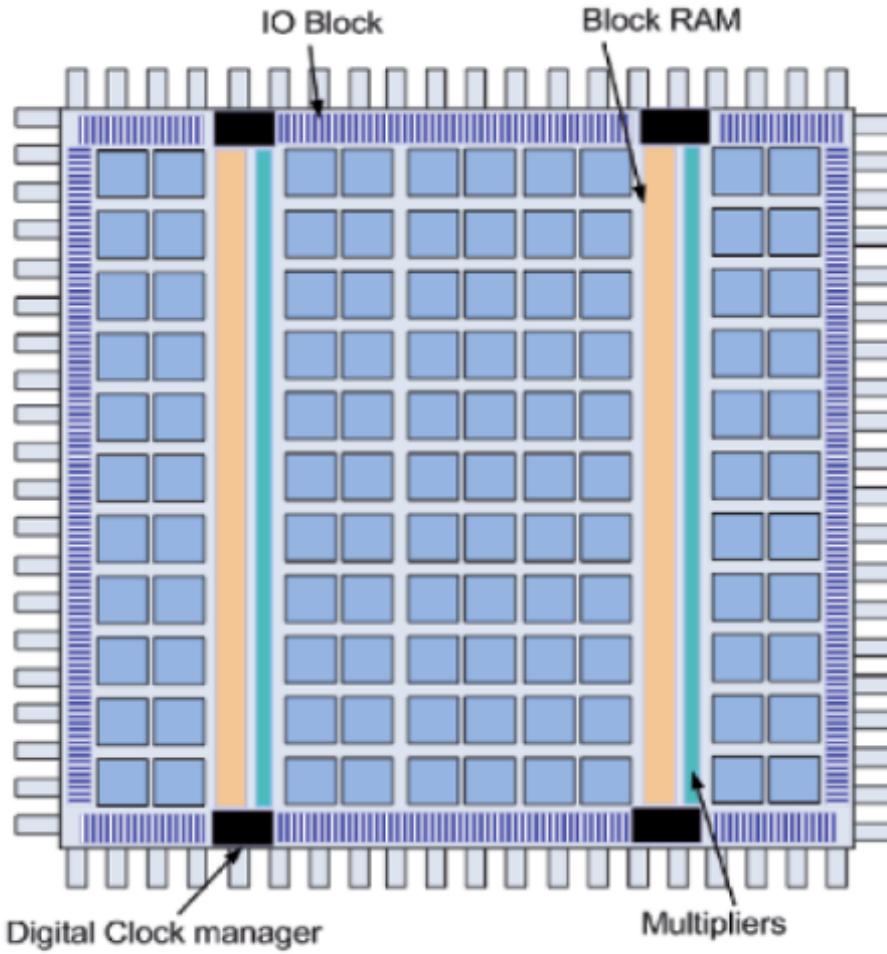


Row-based



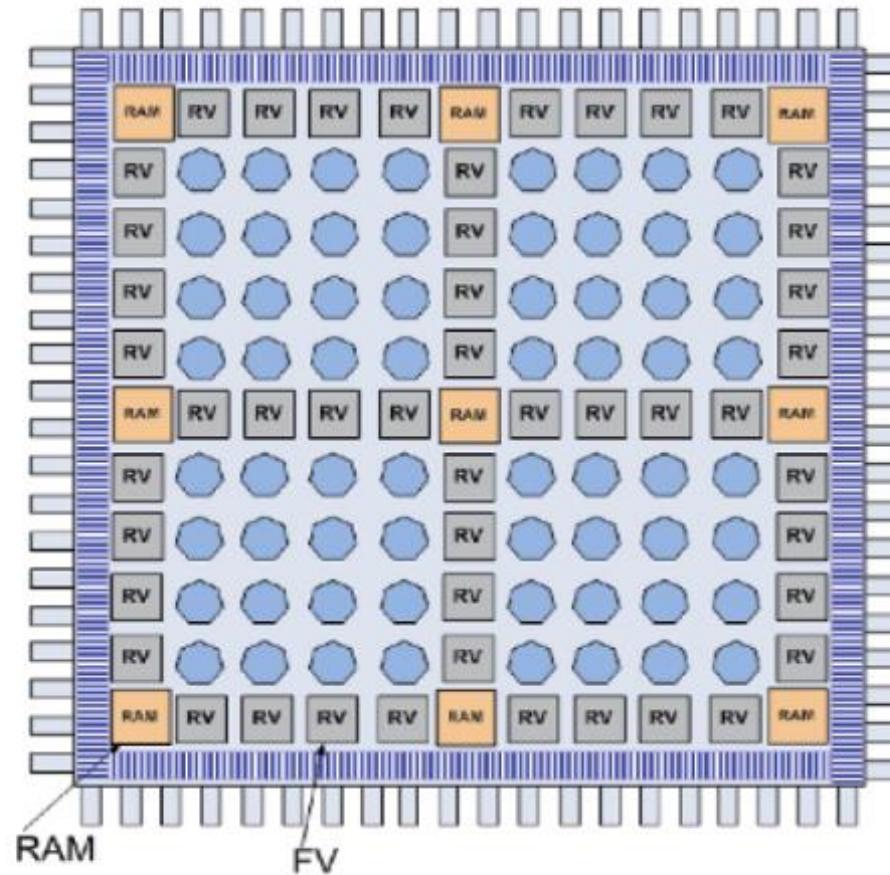
Hierarchical

Actual FPGA Architectures



(a) The Xilinx Virtex II

Actual FPGA Architectures



(b) Atmel's symmetrical array arrangement

Actual FPGA Architectures

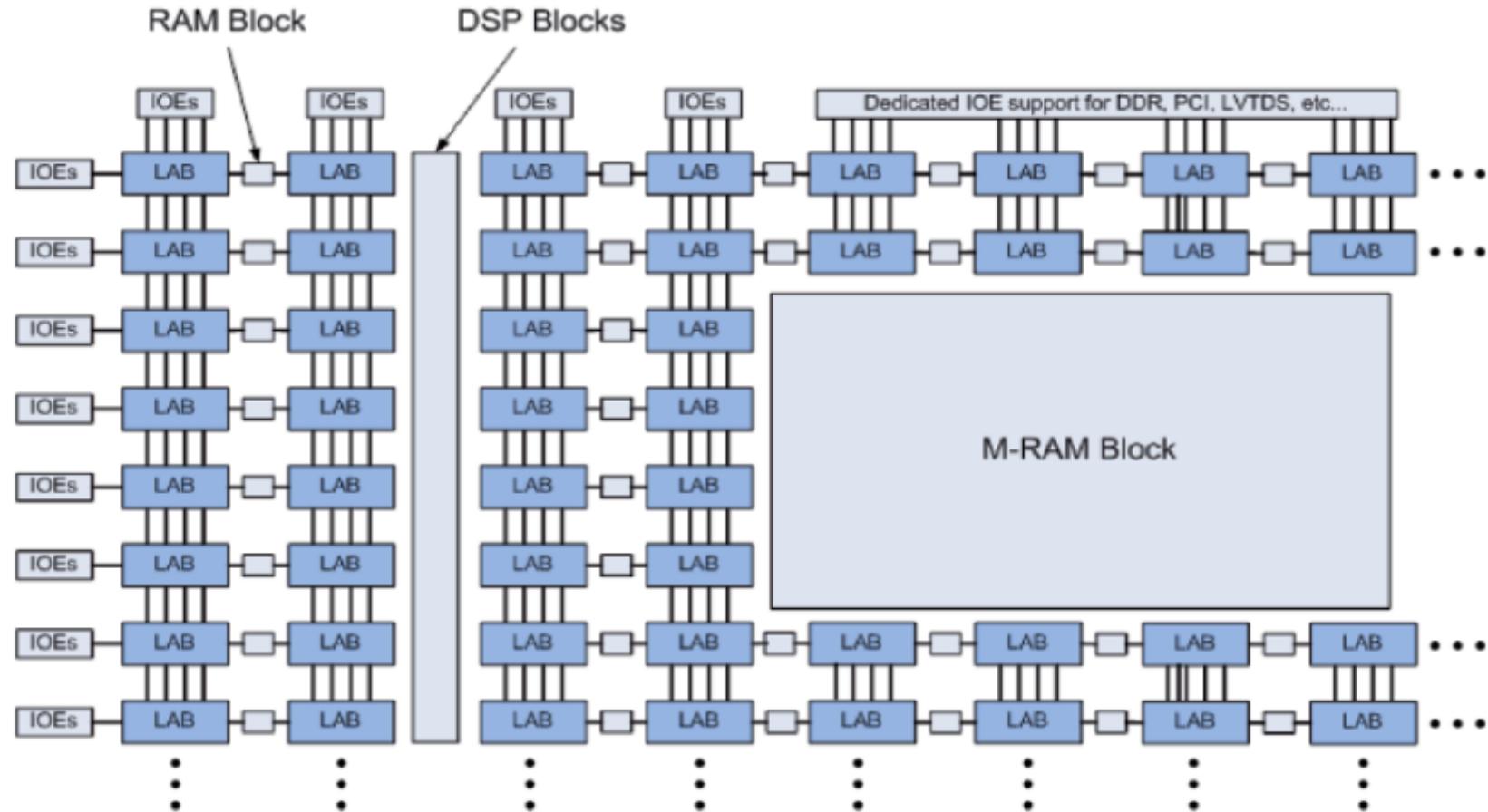


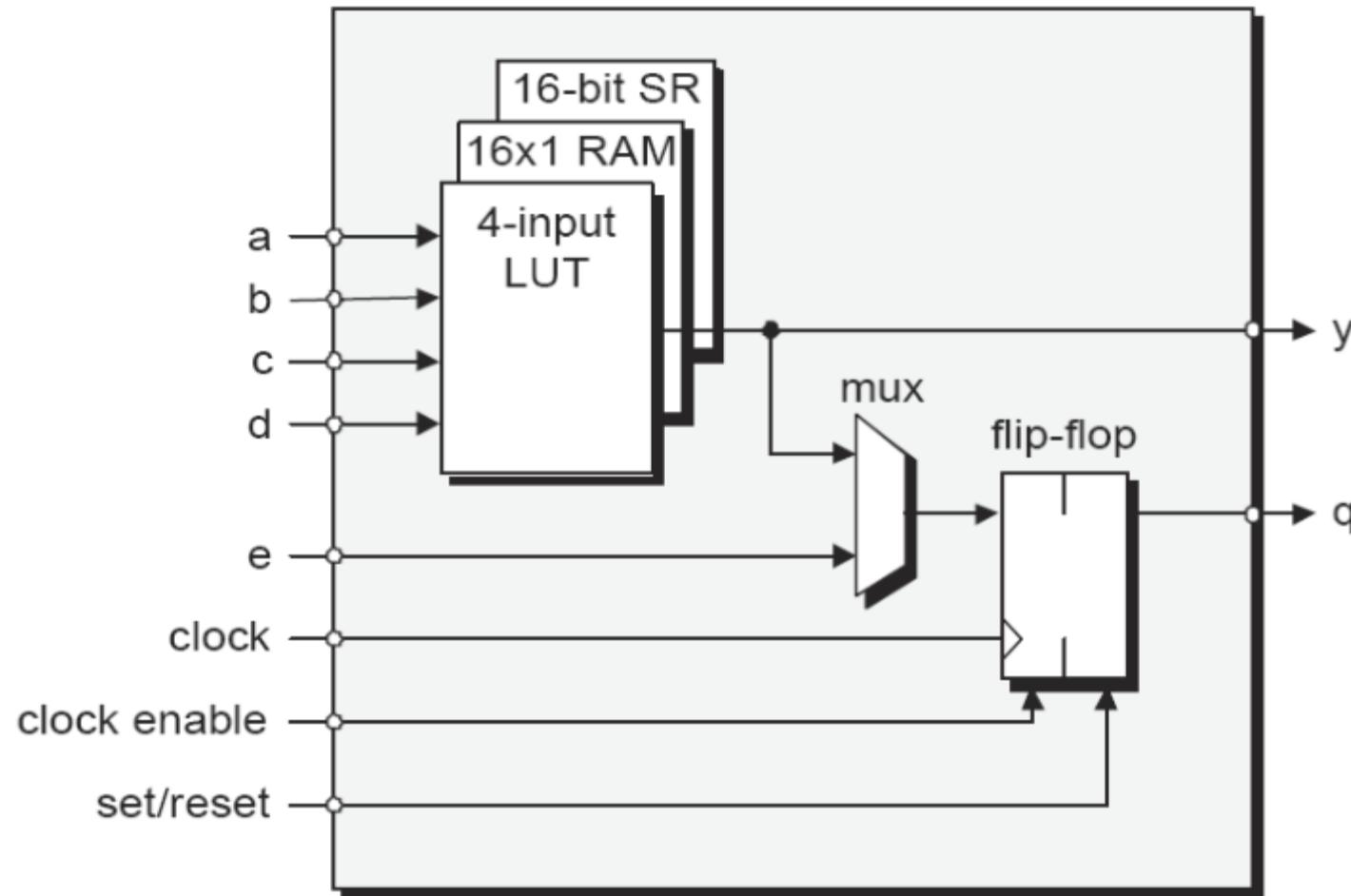
Figure 2.35. Hierarchical arrangement on the Altera Stratix II FPGA

Multipurpose Logic Blocks

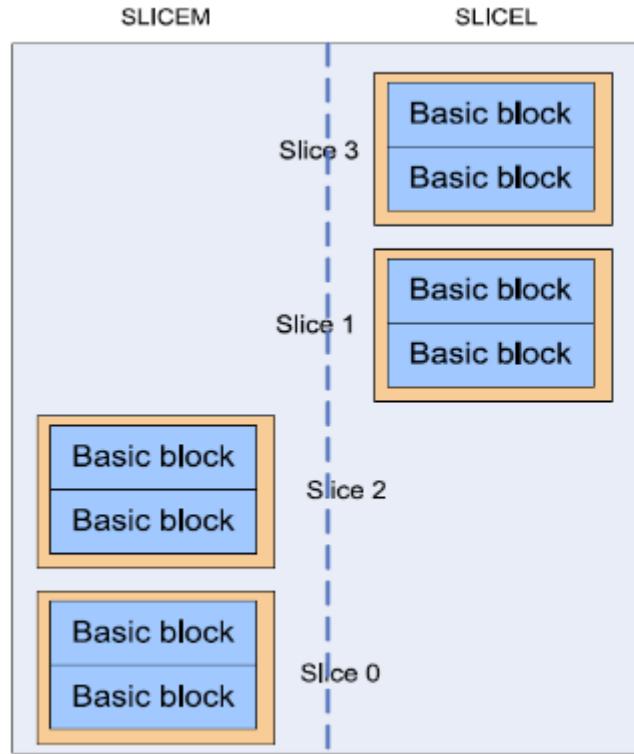
Logic blocks are commonly multi-purpose:

- Shift Registers
- Memory (RAM)
- Look-up-tables (LUT)

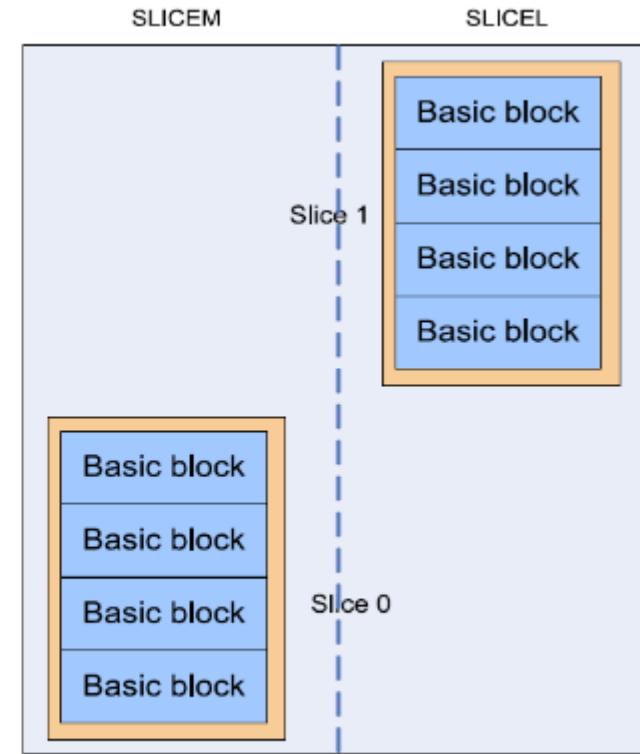
Configurable Logic Blocks



Hierarchical FPGA Architecture



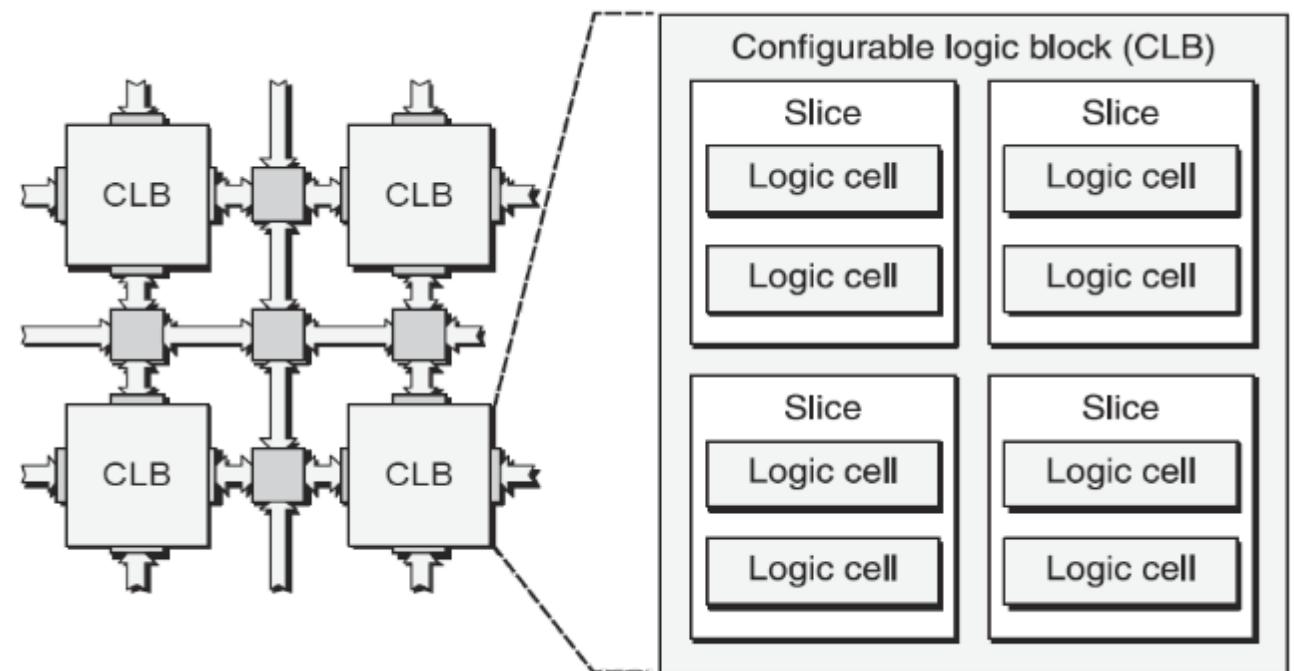
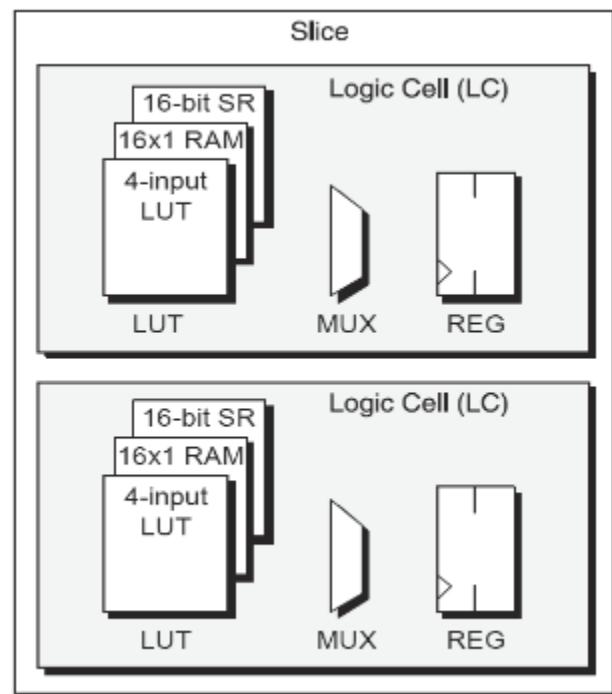
CLB in the Spartan, Virtex II, II Pro and Virtex 4



CLB in the Virtex 5

The left part slices of a CLB (SLICEM) can be configured either as combinatorial logic, or can be used as 16-bit SRAM or as shift register while right-hand slices. The SLICEL can only be configured as combinatorial logic.

Slicing

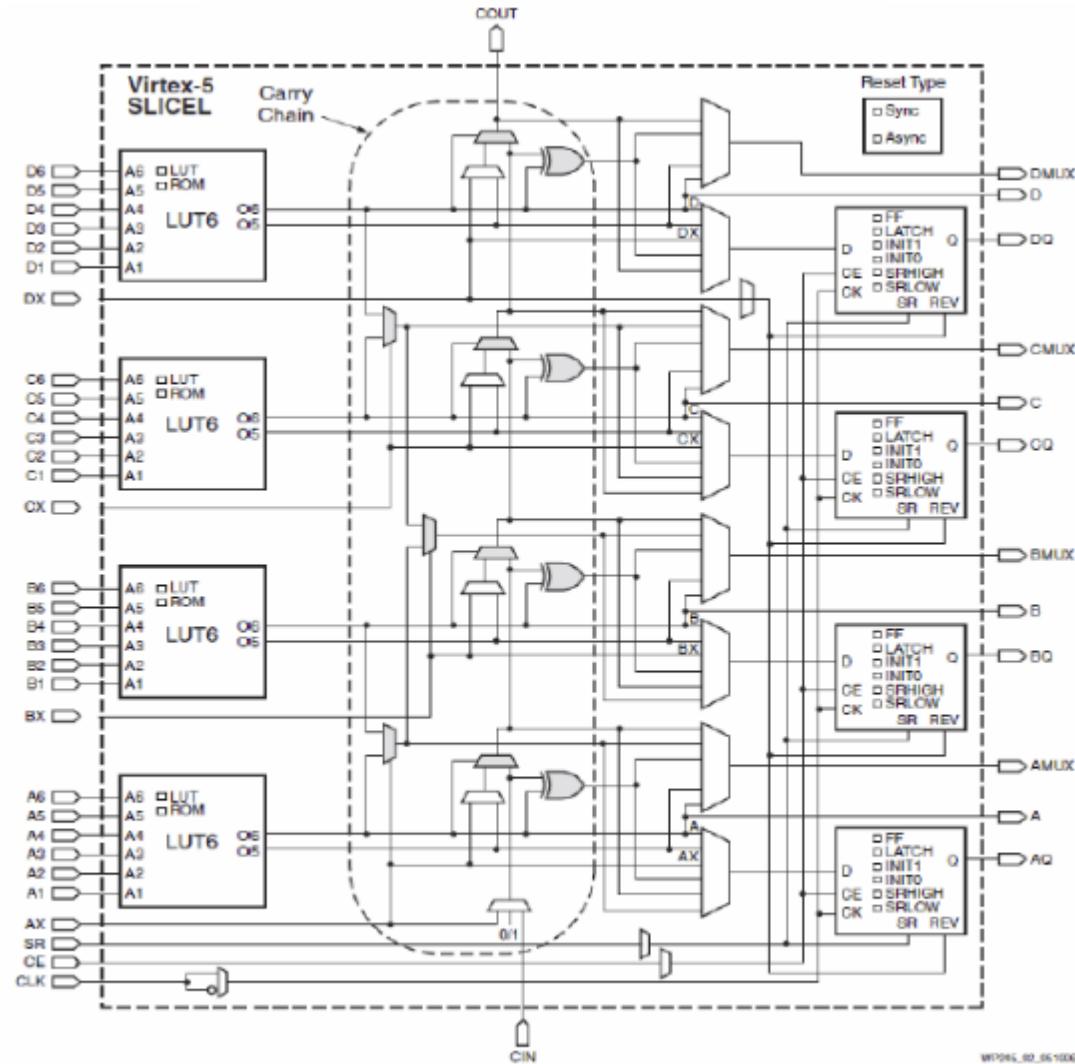


Hierarchical FPGA Architecture

- Current FPGA internal architectures have a sort of hierarchical design, both, in their CLB and interconnection networks:
 - Xilinx Terminology: Logic Cells, Slices, Configurable Logic Blocks
 - Altera Terminology: Logic Element, Logic Array Block

Why?

Xilinx Logic Blocks



Фиг. 6. Structure of the slice element within Virtex-5.

Xilinx vs. Altera Logic Cells

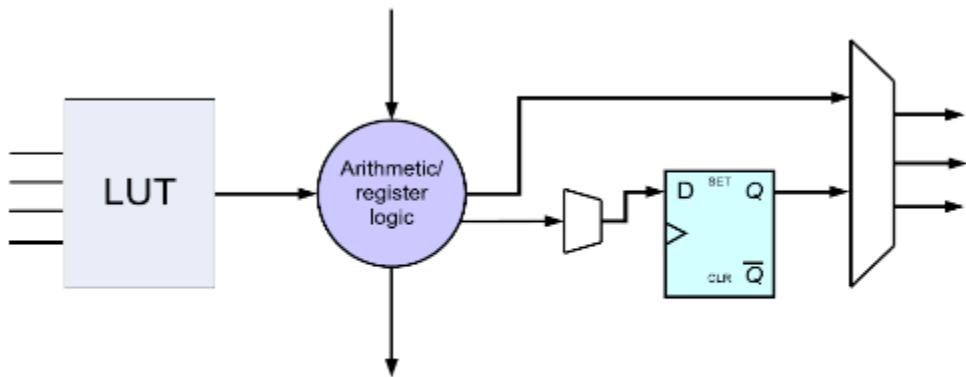


Figure 2.26. Logic Element in the Cyclone II

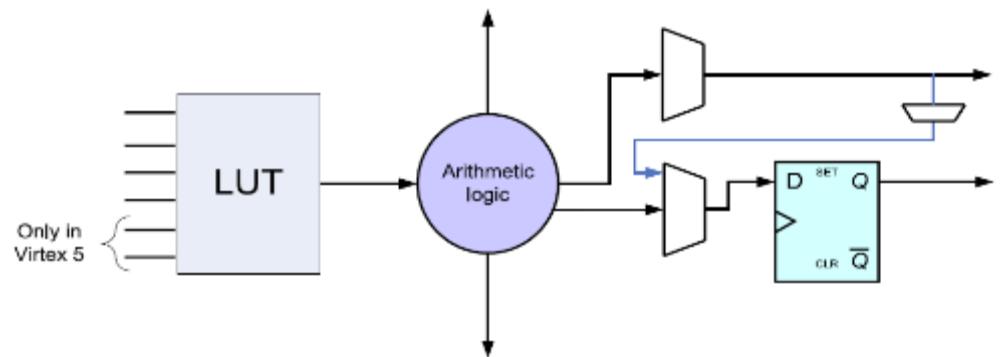


Figure 2.24. Basic block of the Xilinx FPGAs

Altera Logic Blocks

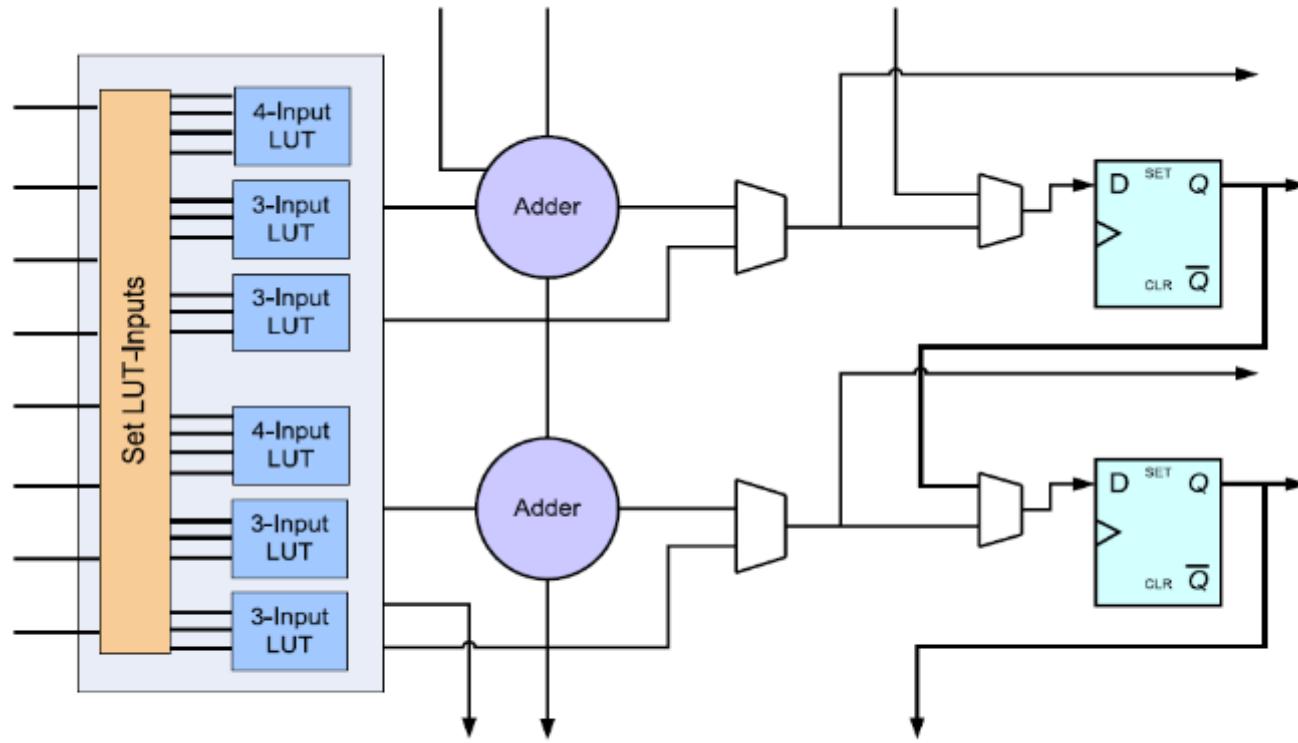


Figure 2.27. Stratix II Adaptive Logic Module

- **Question:** How do companies decide about their FPGA internal architecture? Is it a only matter of technology or taste?

Interconnect Networks

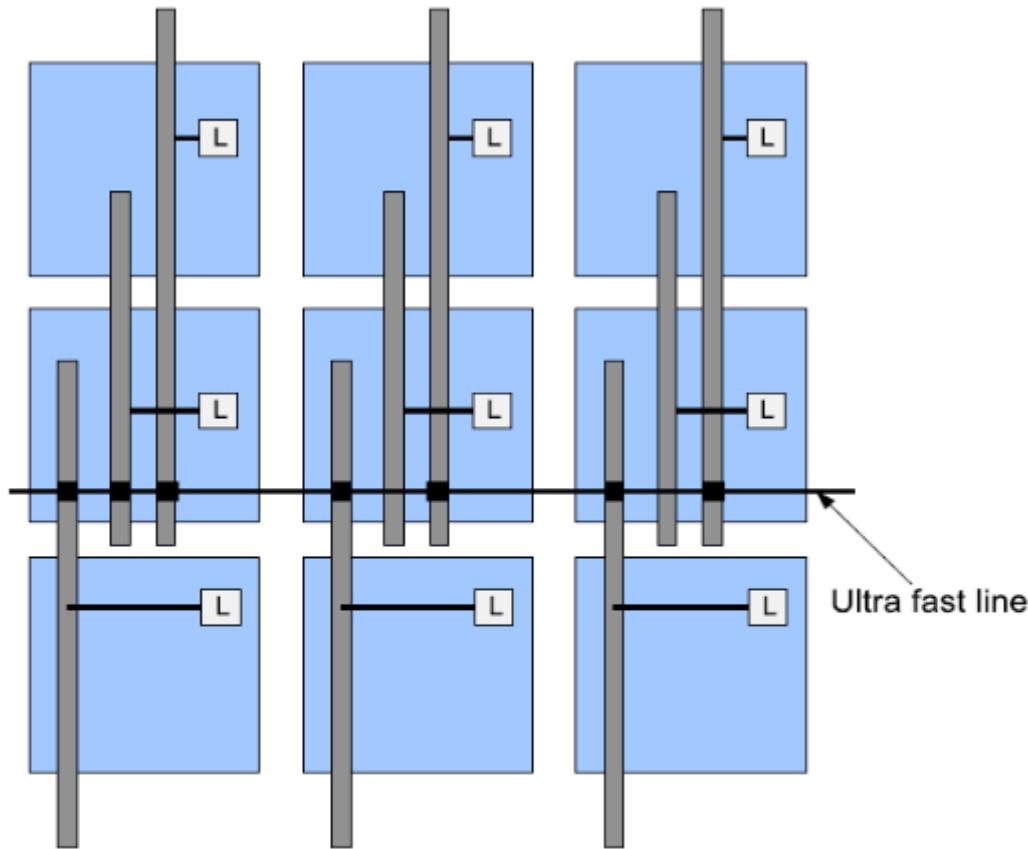
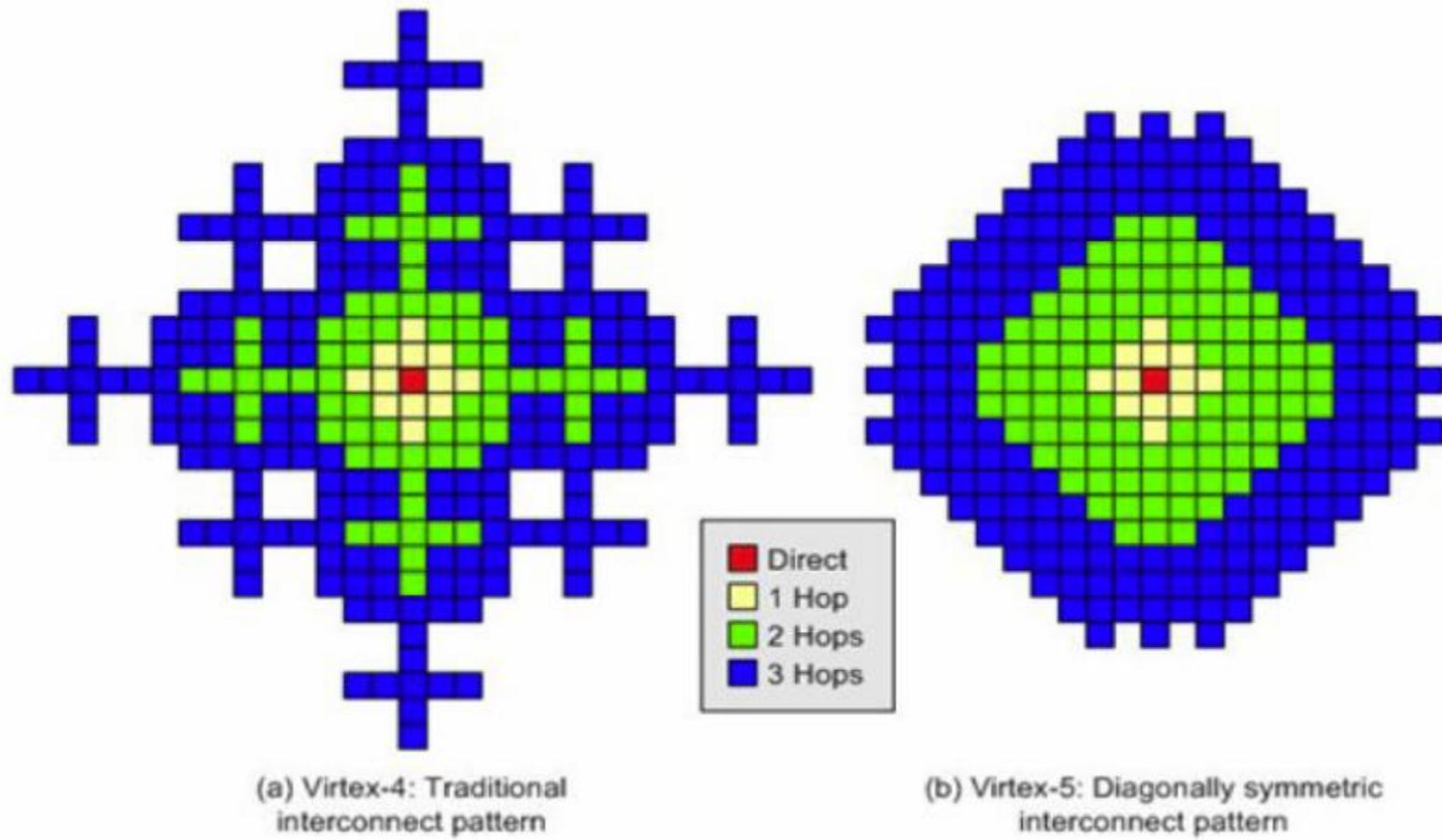


Figure 2.34. Actel ProASIC local routing resources

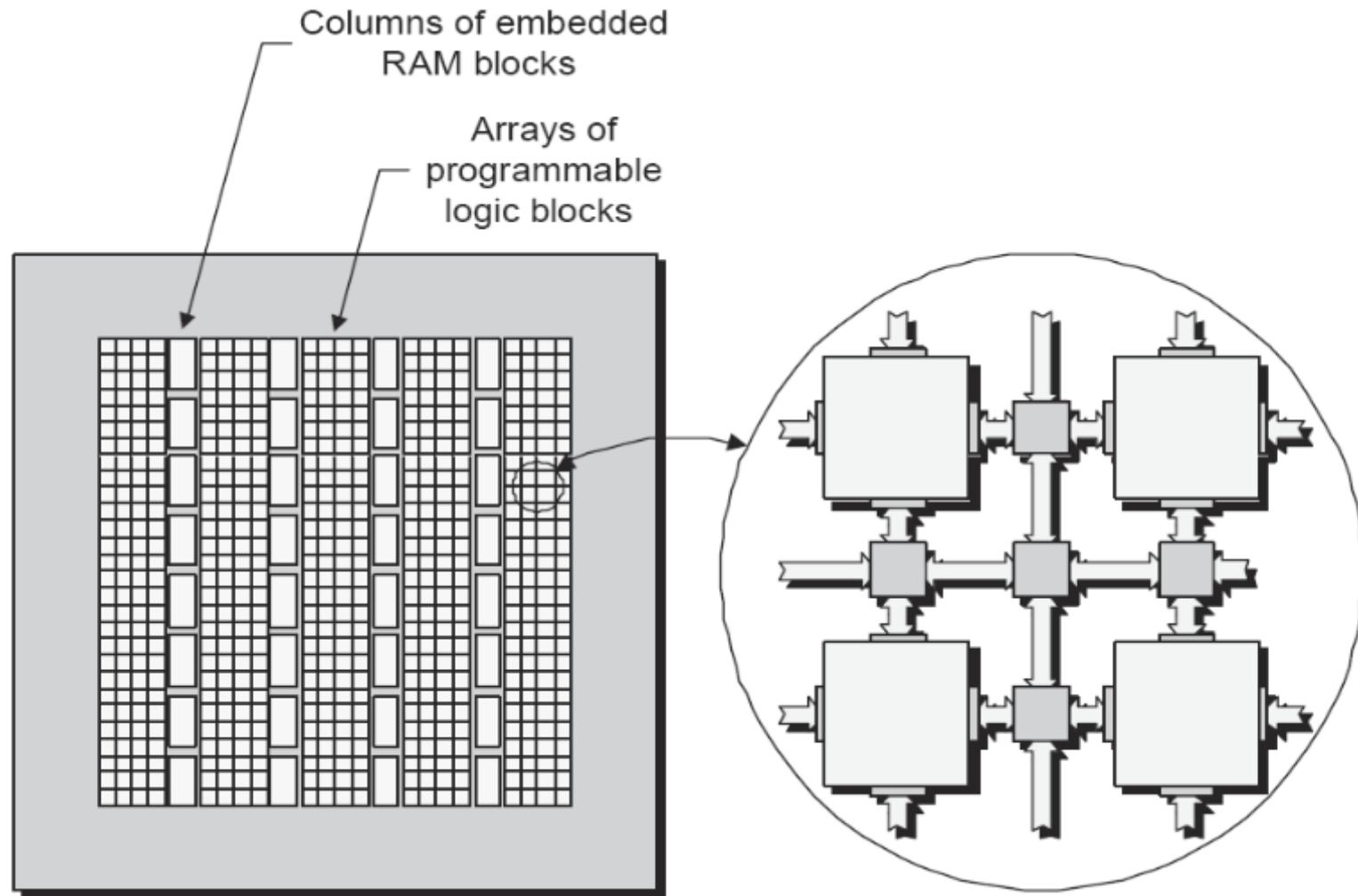
Interconnect Networks



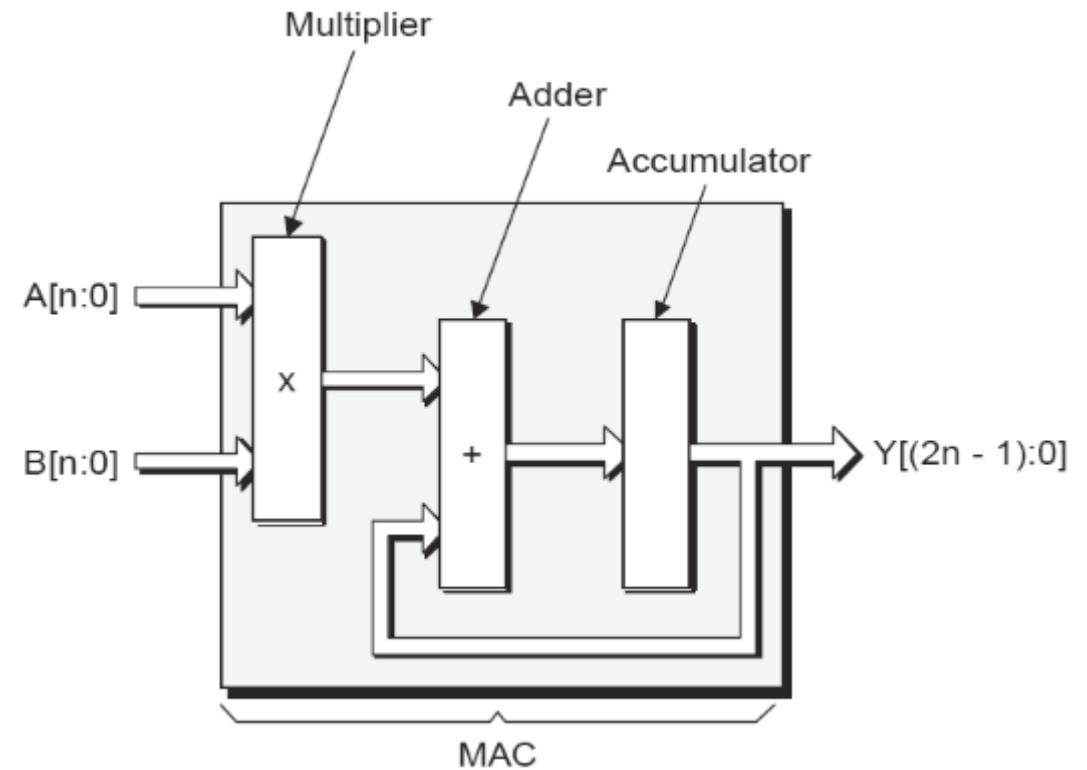
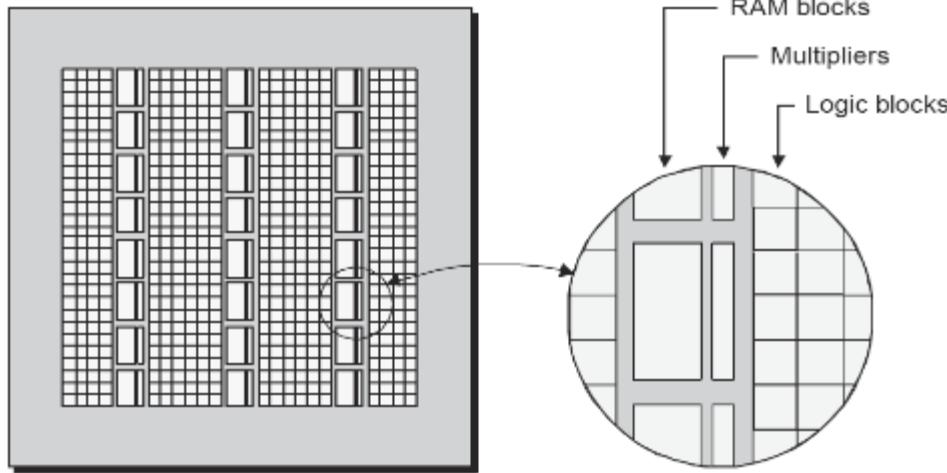
Other Peripherals within Contemporary FPGA

- Block Memories
- Digital Clock Managers
- Dedicated Adders & Multipliers
- Variety of I/O interfaces
- Embedded Processors

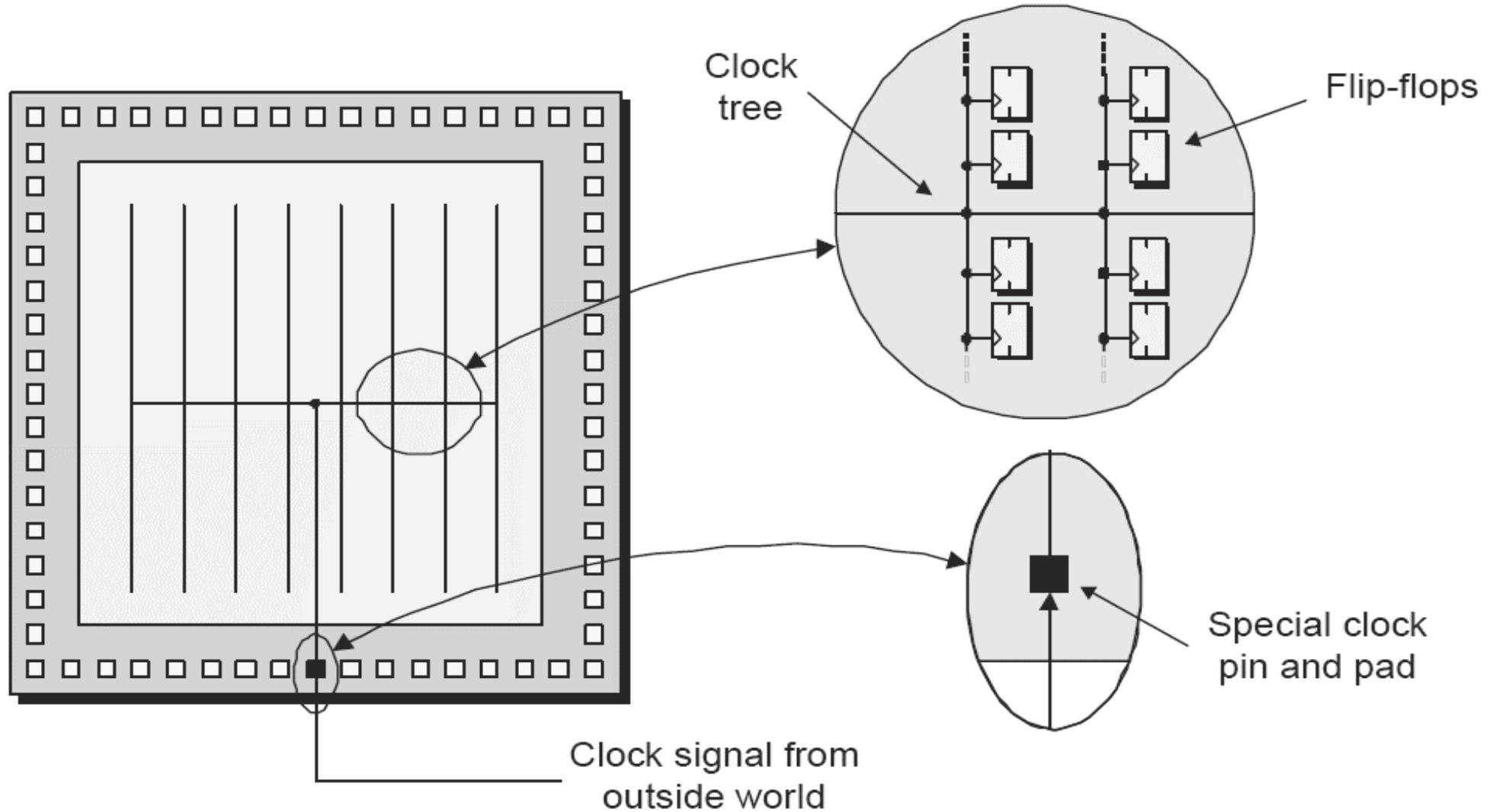
Embedded Block Memories



Embedded Multiplier, Adder, MAC



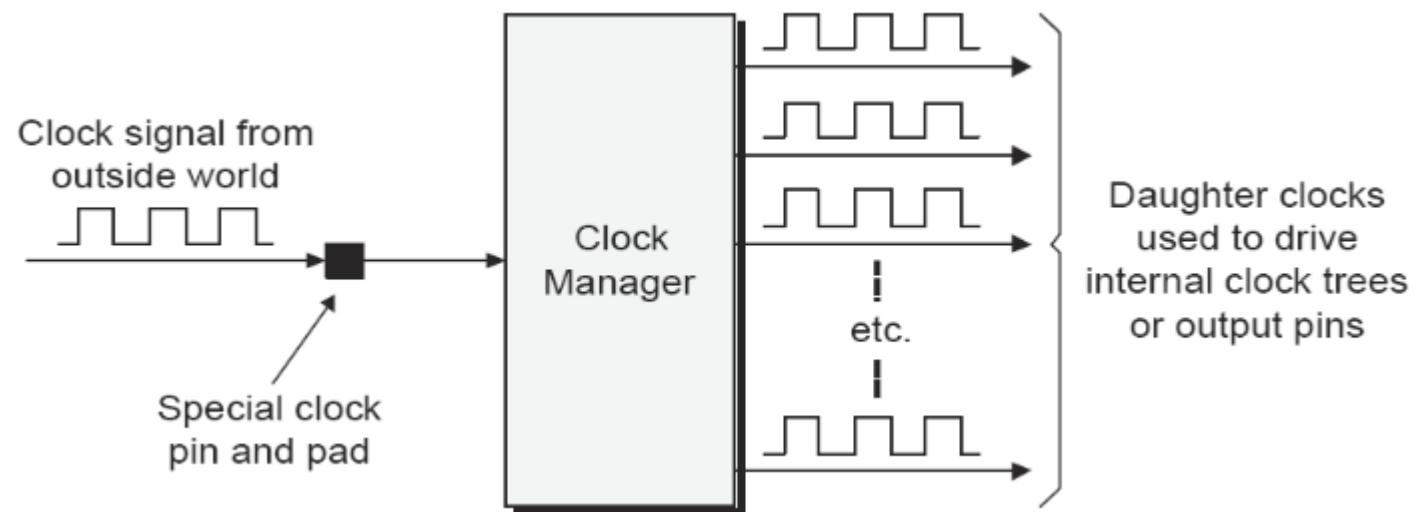
Clock Trees



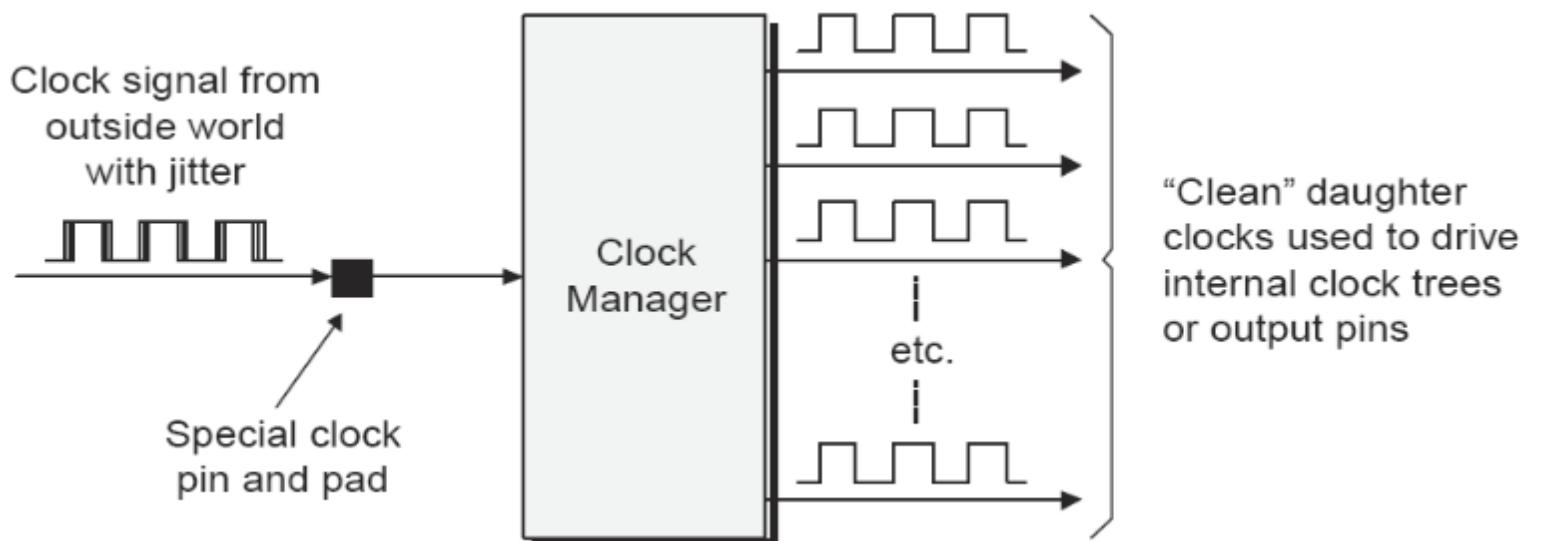
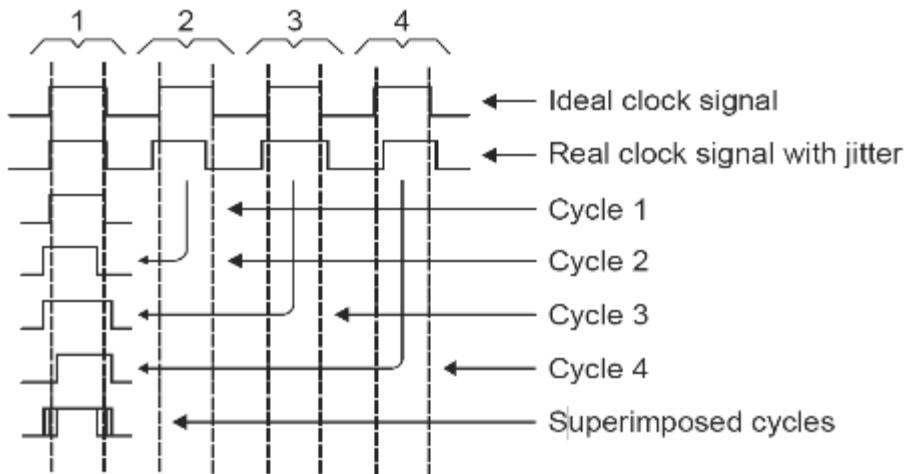
Clock Management

Usage:

1. Jitter removal
2. Frequency synthesis
3. Phase shifting
4. Clock de-skewing



1. Jitter Removal



2. Frequency Synthesis

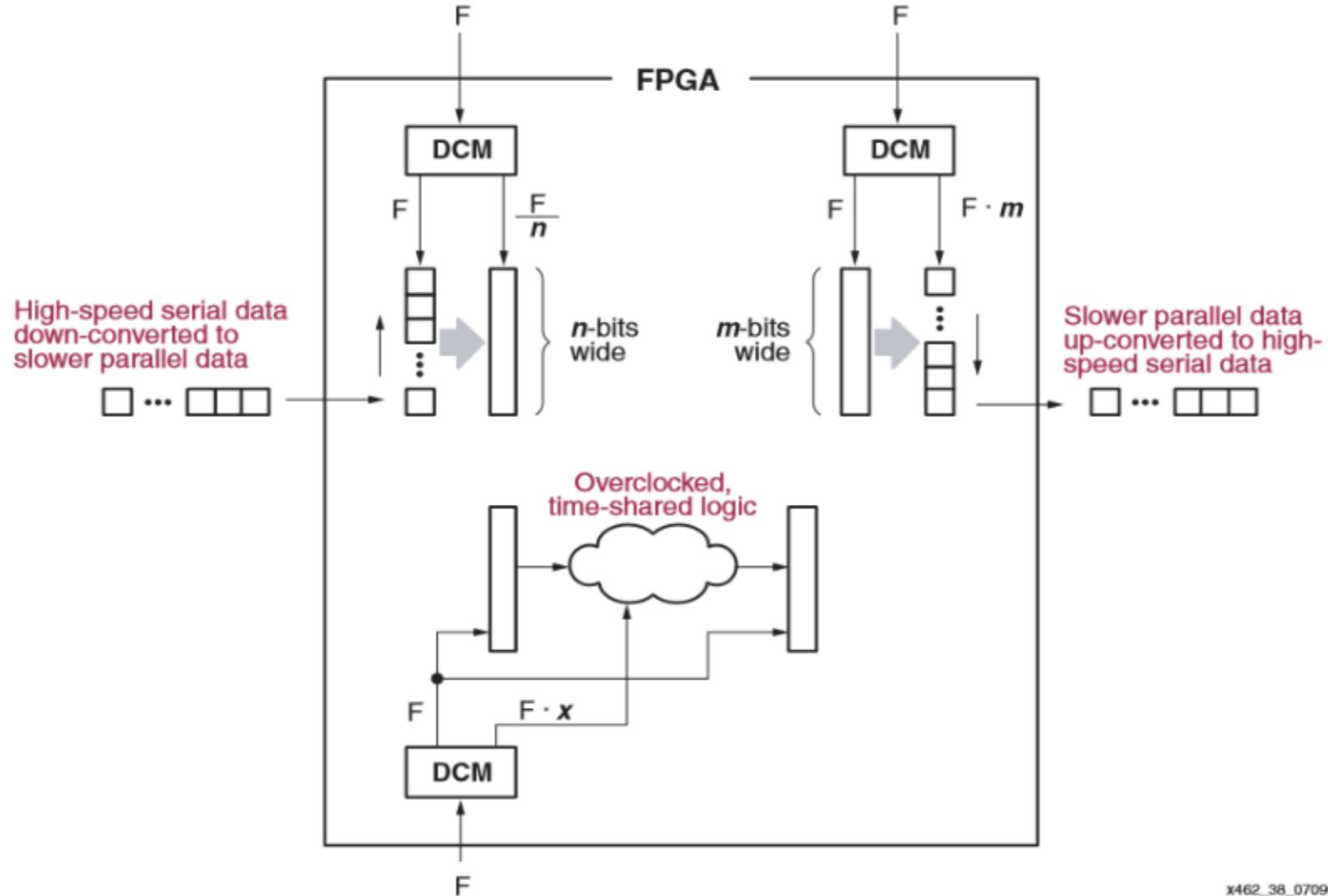
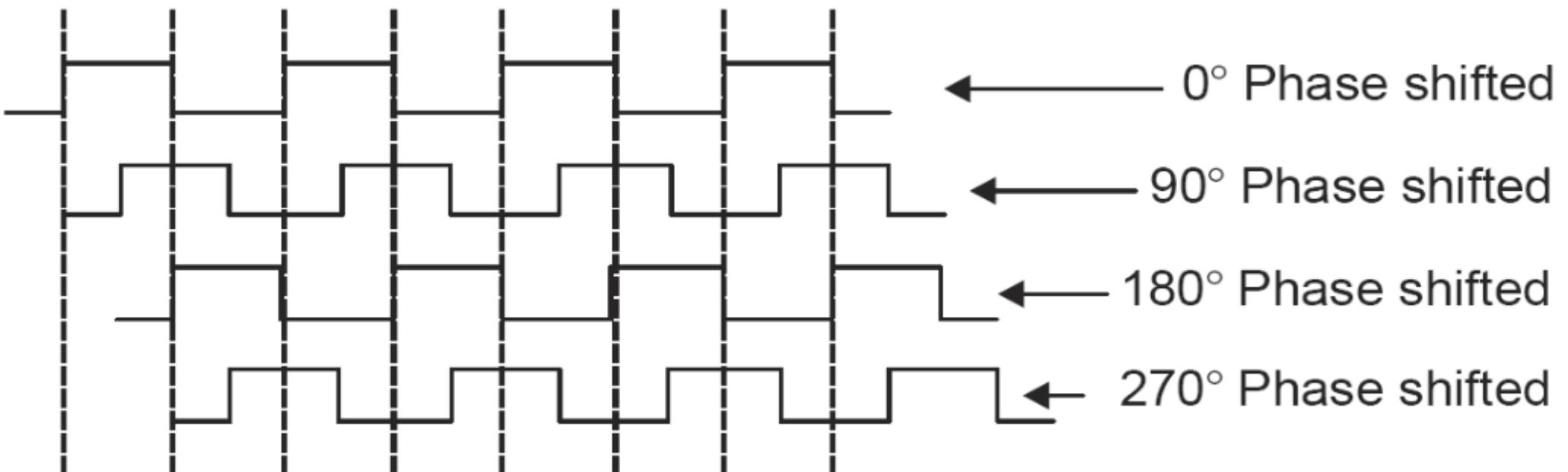


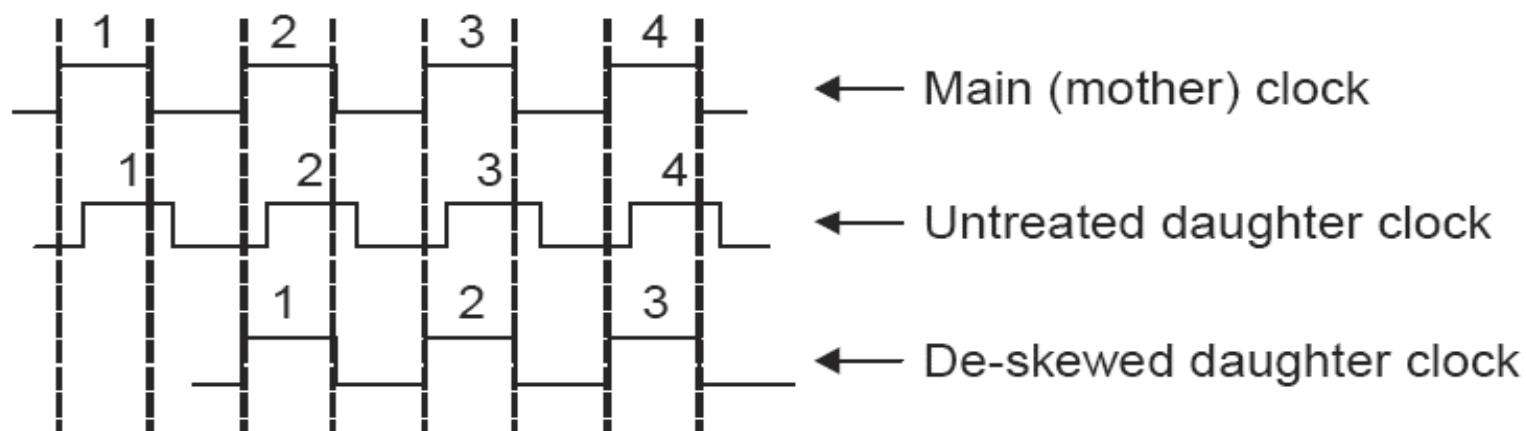
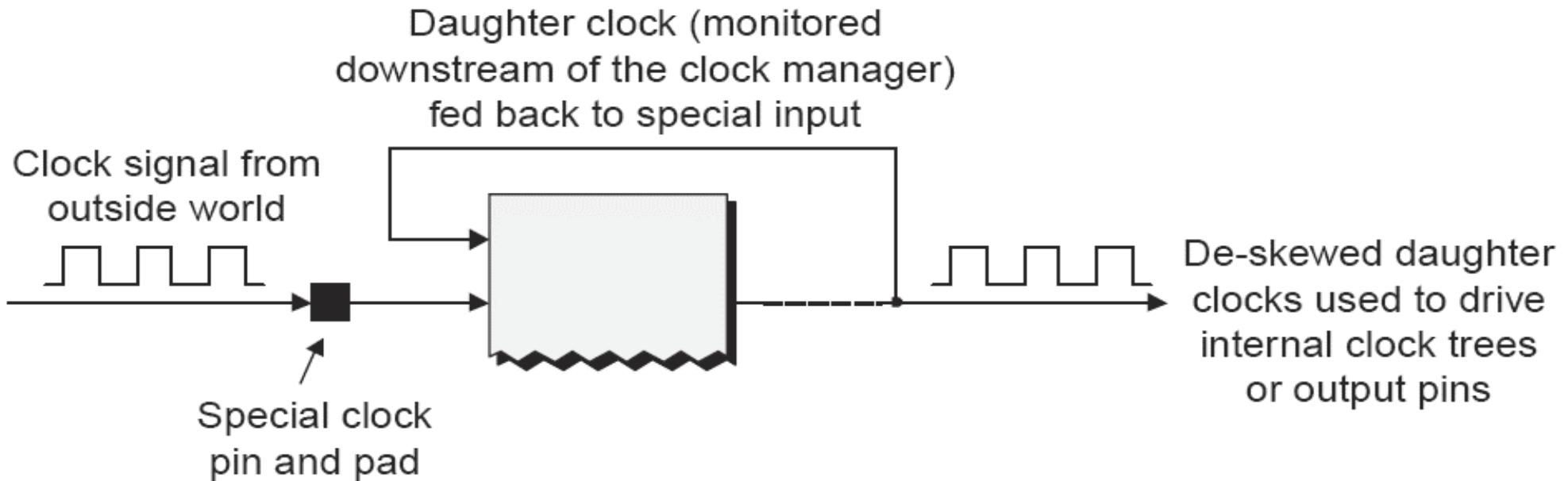
Figure 38: Common Applications of Frequency Synthesis

x462_38_070903

3. Phase Shifting



4. Clock De-skewing



Xilinx Digital Clock Manager (DCM)

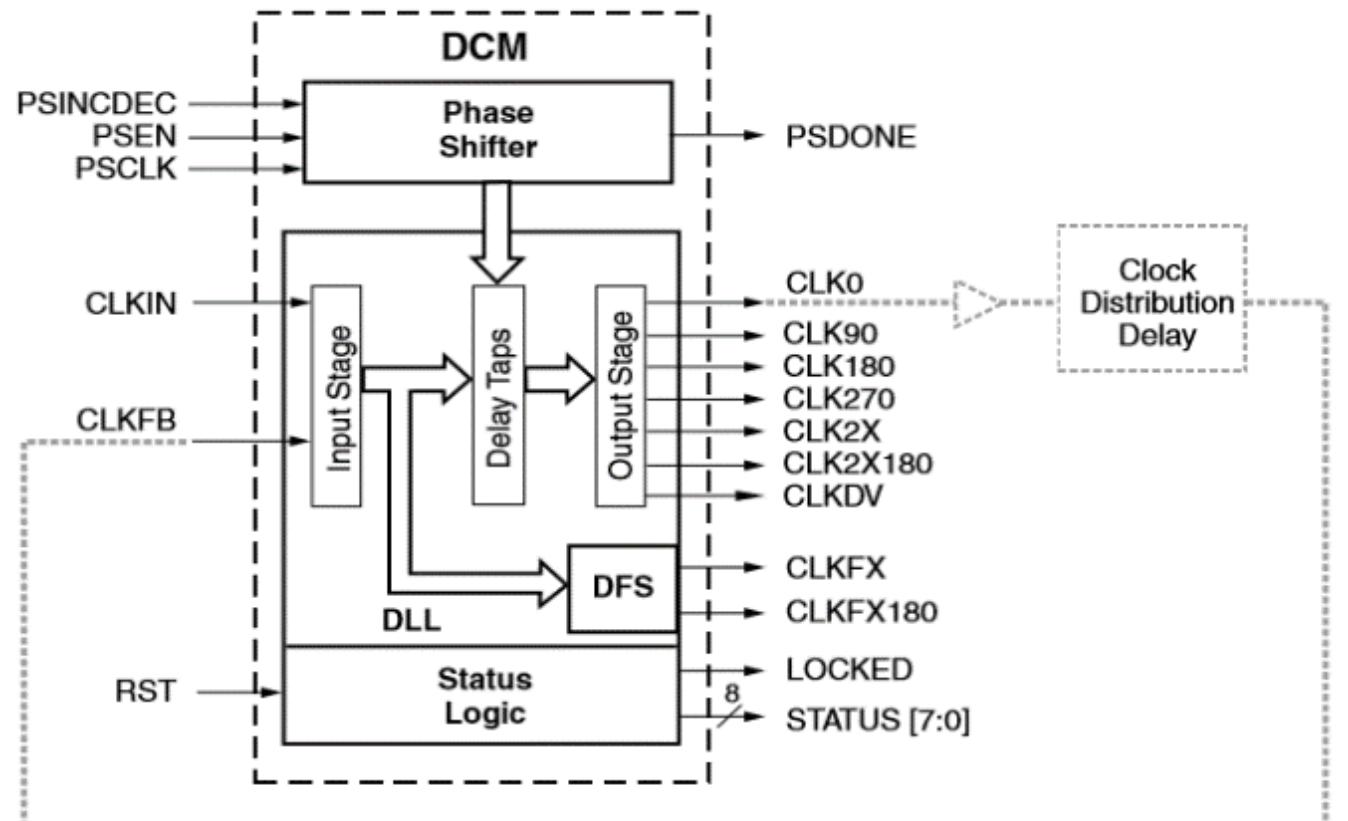
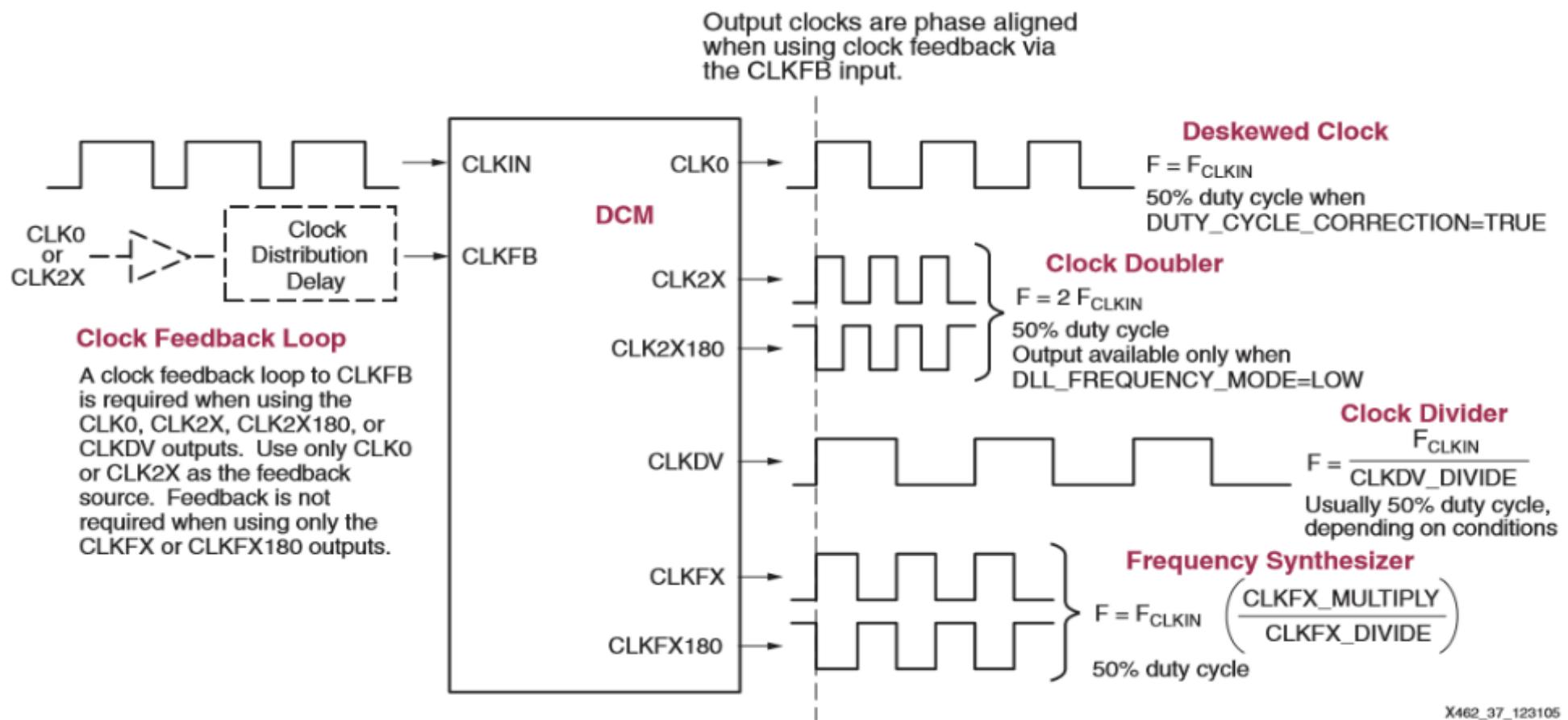


Figure 3: DCM Functional Block Diagram

Reference: http://www.xilinx.com/support/documentation/application_notes/xapp462.pdf

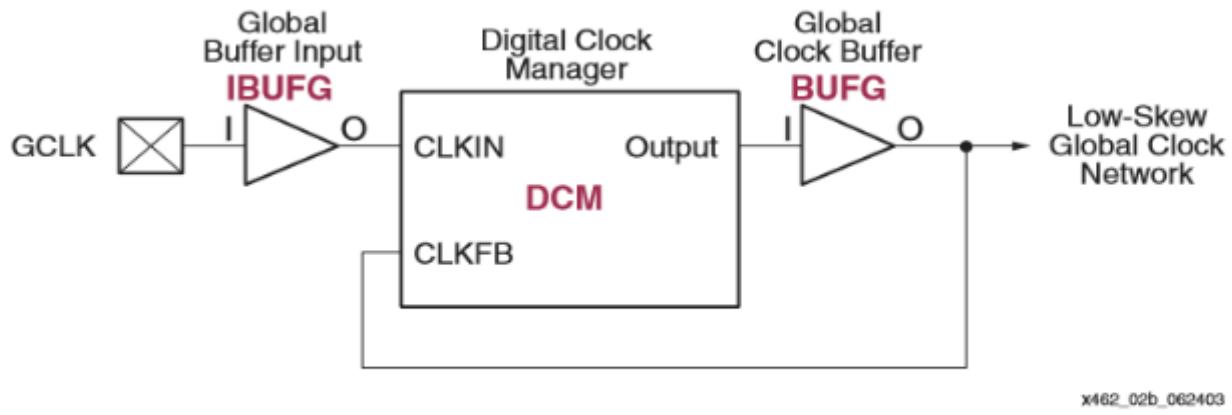
Xilinx DCM Clock Synthesis Options



Xilinx DCM Functional Overview

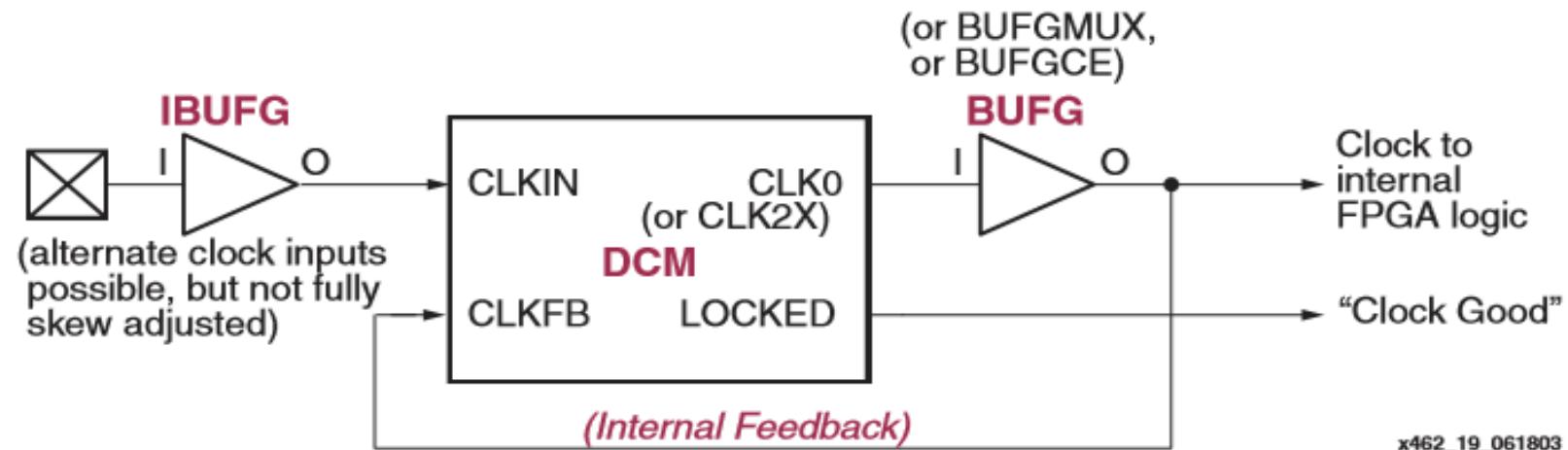


a. Global Buffer Inputs and Clock Buffers Drive a Low-Skew Global Network in the FPGA



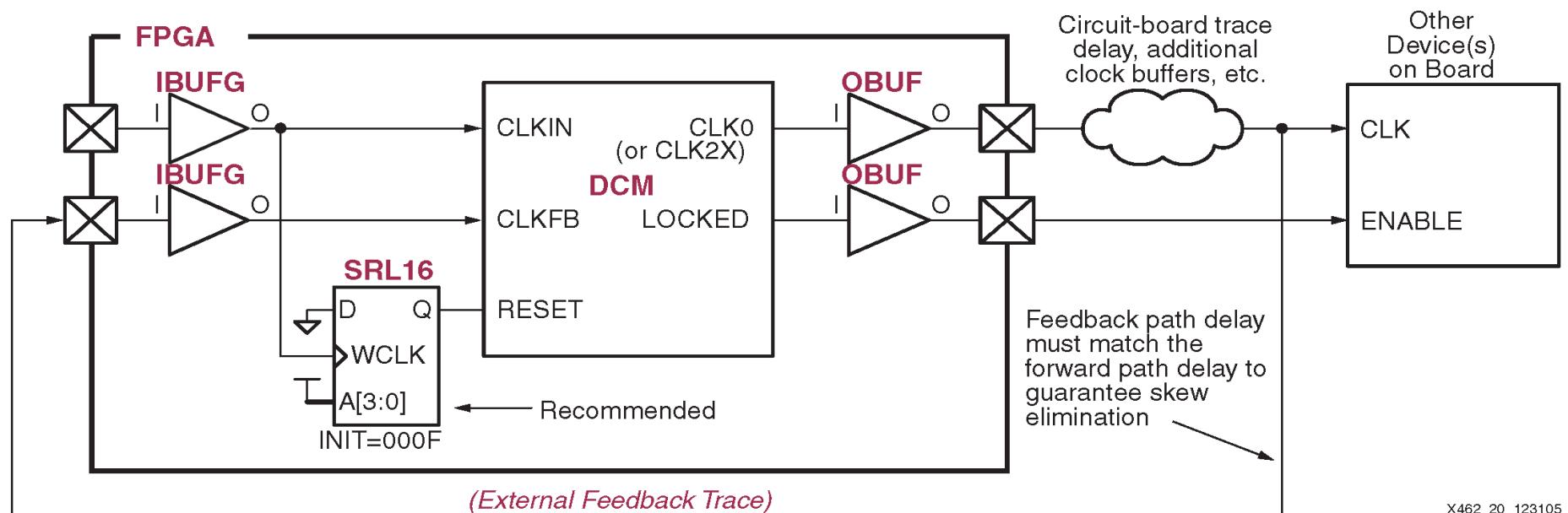
b. A Digital Clock Manager (DCM) Inserts Directly into the Global Clock Path

Internal Clock De-skewing



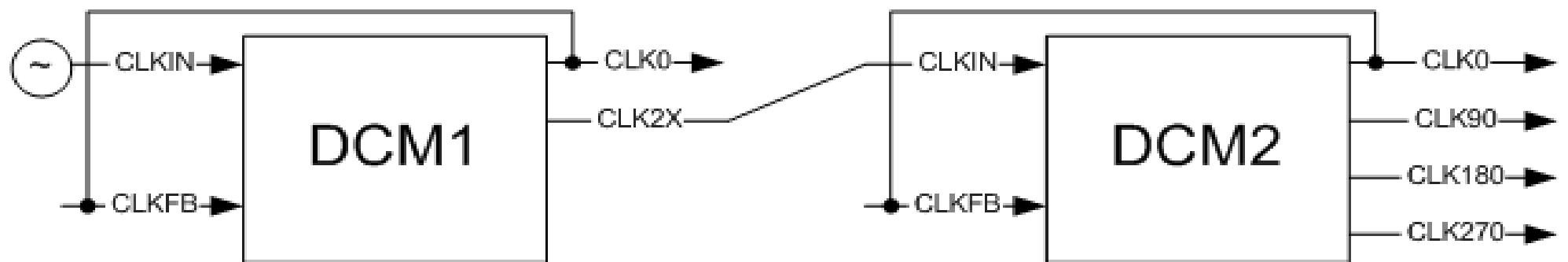
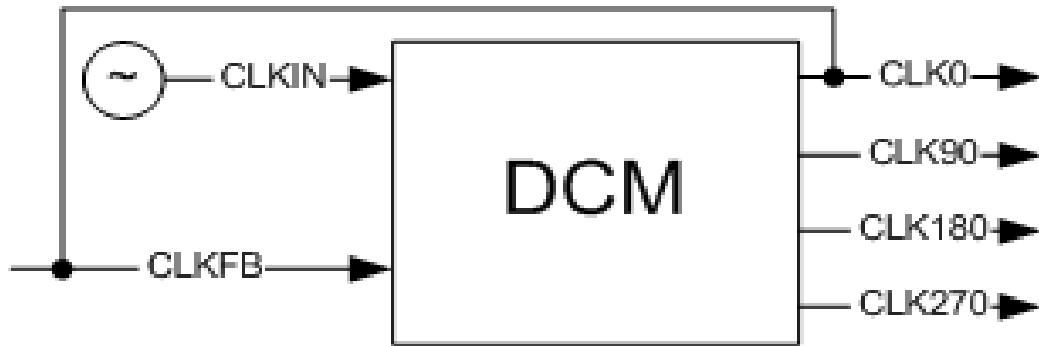
x462_19_061803

External Clock De-skewing



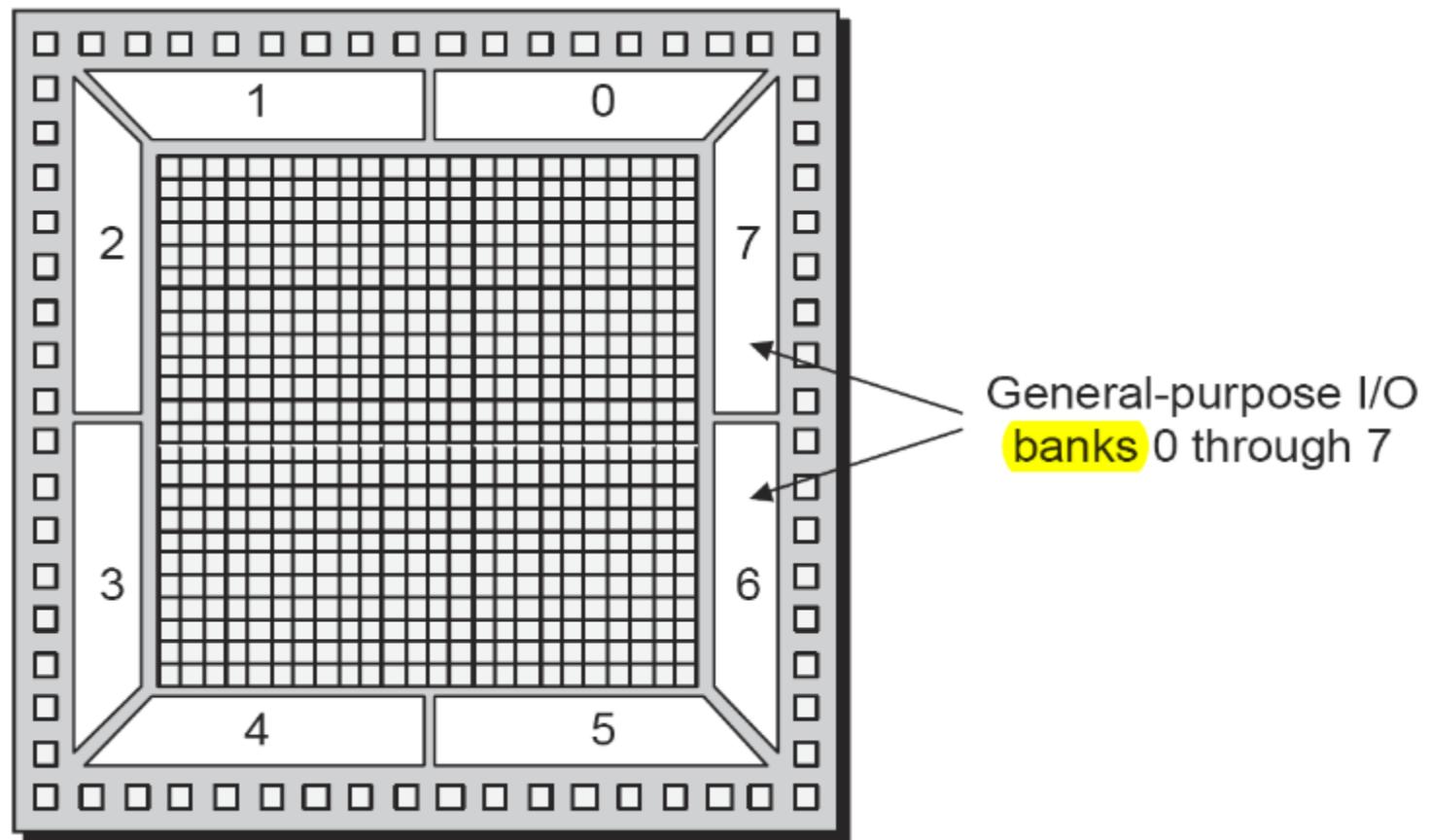
Reference: http://www.xilinx.com/support/documentation/application_notes/xapp462.pdf

DCM Cascading



General Purpose I/O

- The concept of I/O Banks

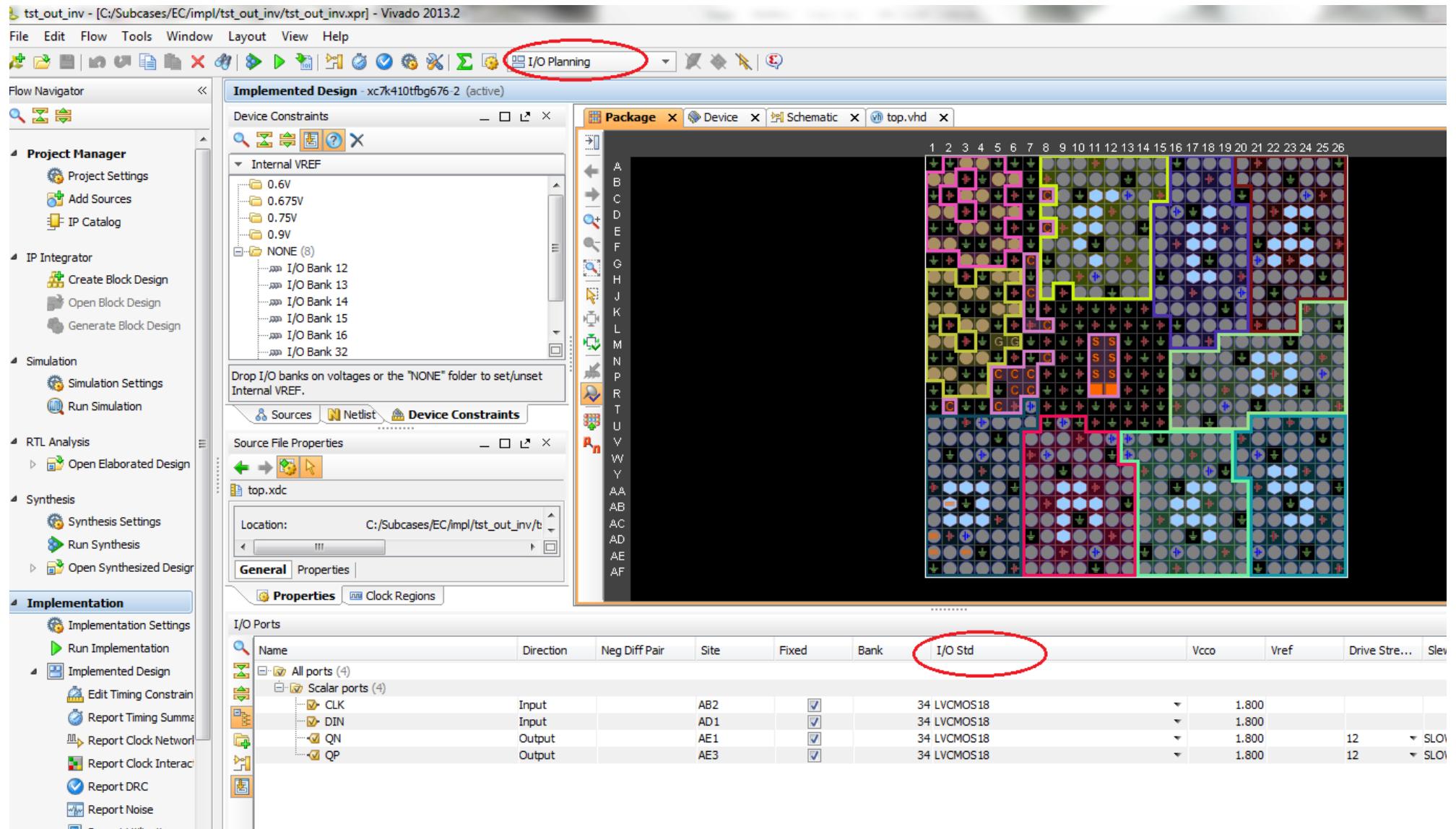


Xilinx Series 7 I/O Banks

The 7 series Xilinx FPGAs offer:

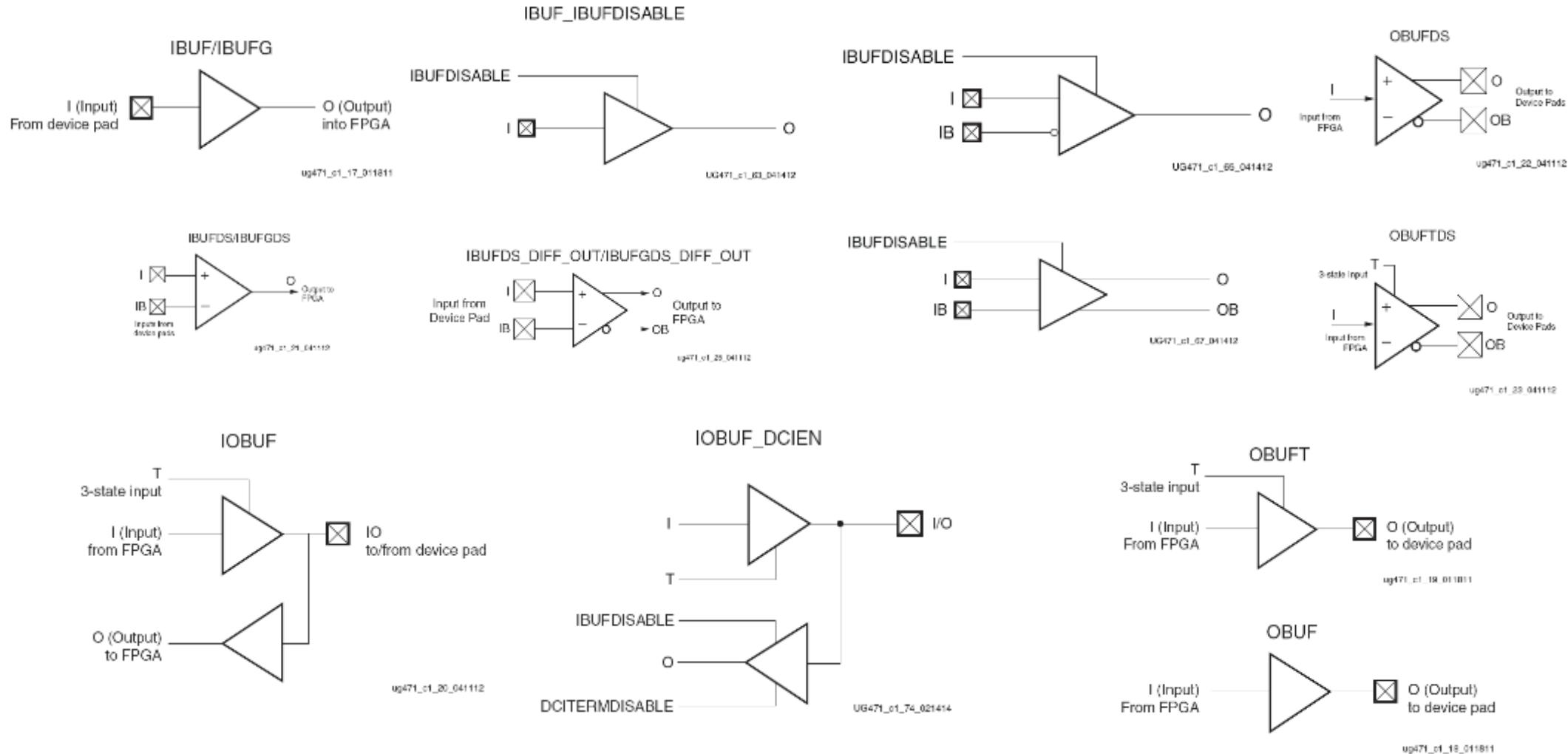
- **High-performance (HP) I/O Banks:** designed to meet the performance requirements of high-speed memory and other chip-to-chip interfaces with voltages up to 1.8V
- **High-range (HR) I/O Banks:** designed to support a wider range of I/O standards with voltages up to 3.3V
- **Different I/O voltage standards:** 3.3V, 2.5V, 1.8V, 1.5V, 1.35V, 1.2V, which includes LVTTLL, LVCMOS, etc. standards
- **Digitally-controlled impedance (DCI) and DCI cascading:** in the HP mode

Xilinx FPGA I/O Planning and I/O Banks

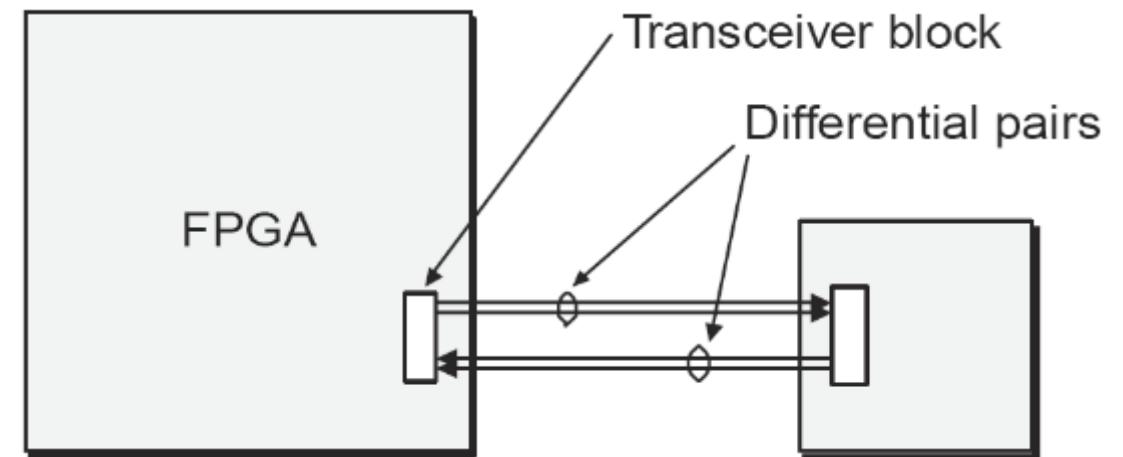
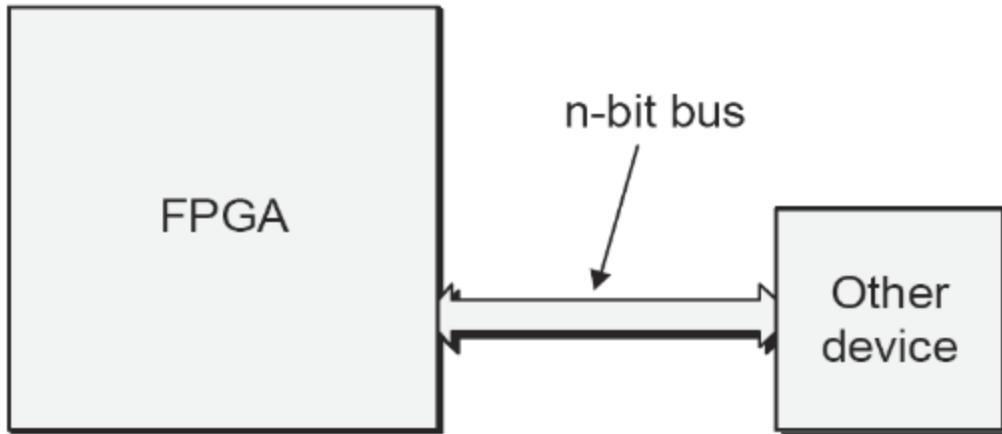


Xilinx Series 7 I/O Buffers

- Various I/O buffers are supported on standard FPGA devices:



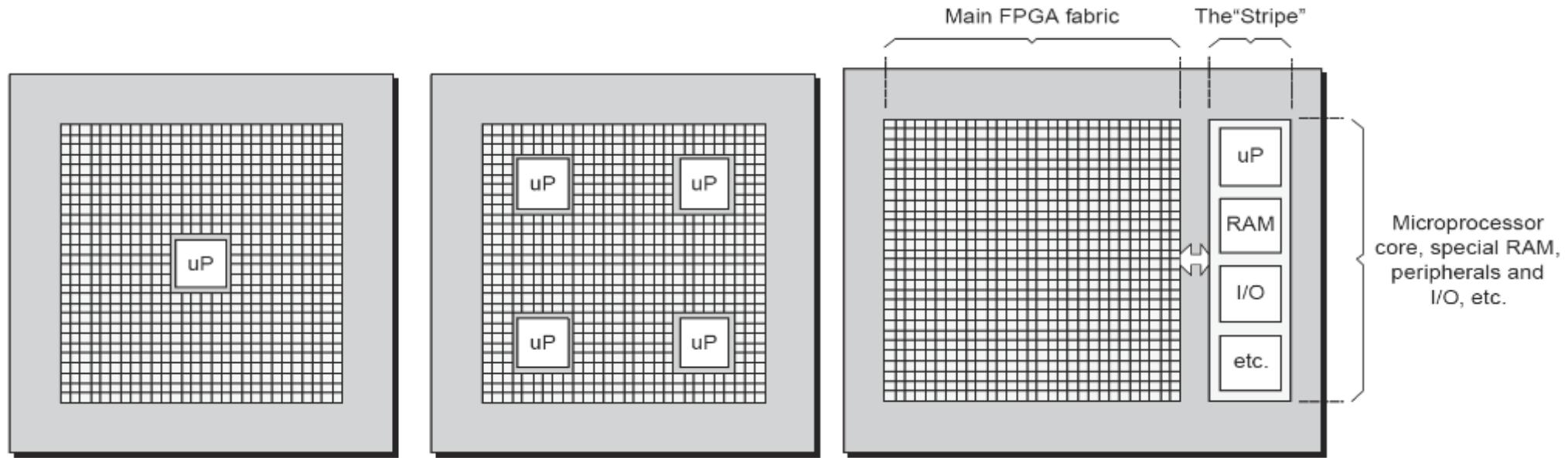
Giga-Bit Transceiver Bus



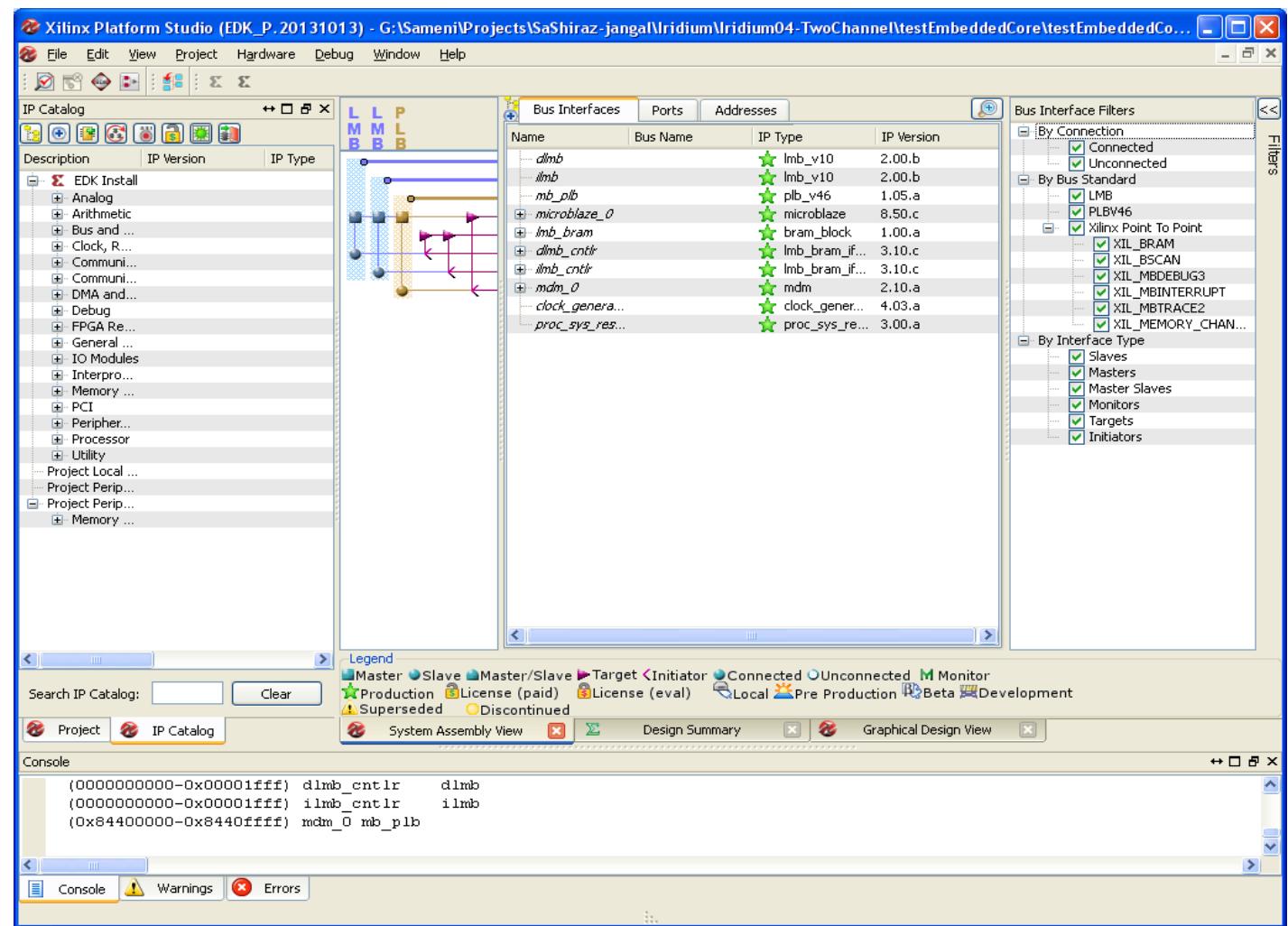
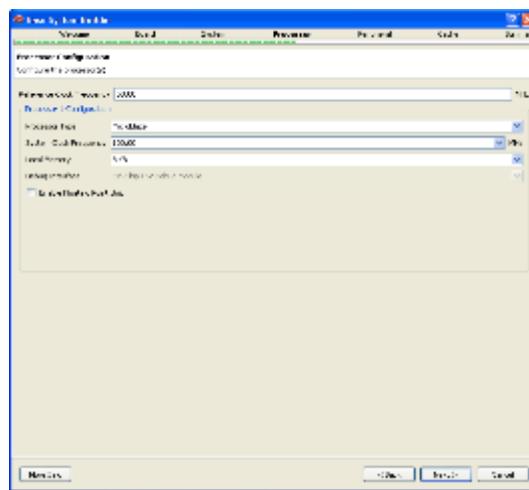
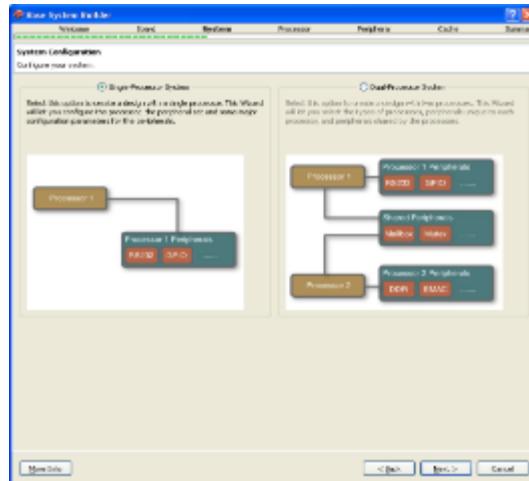
IP Cores

- Hard IP
 - In the form of pre-implemented blocks such as microprocessor cores, gigabit interfaces, multipliers, adders, MAC functions, etc.
Example: Xilinx PowerPC
- Soft IP
 - Source-level library of high-level functions that can be integrated in a custom design.
- Firm IP
 - Libraries which have already been optimally mapped, placed, and routed into a group of programmable logic blocks (and possibly combined with some hard IP blocks like multipliers, etc.) and may be integrated into a custom design.
Example: Xilinx MicroBlaze

Hard Embedded Processors



Soft Embedded Processors: Microblaze

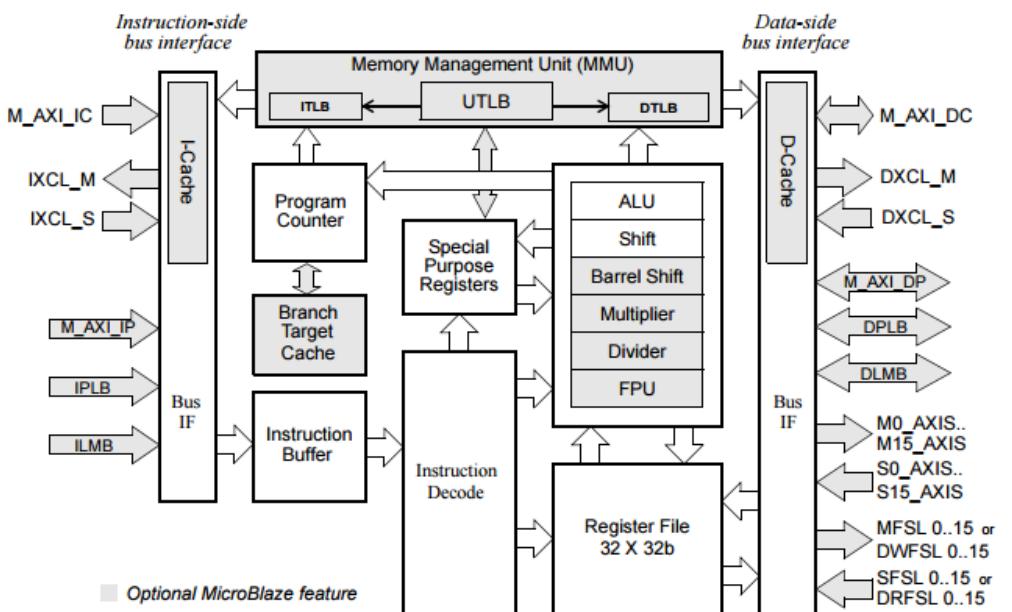


Xilinx Microblaze Core Block Diagram

MicroBlaze™ is Xilinx 32-bit RISC Harvard architecture soft processor core with a rich instruction set optimized for embedded applications.

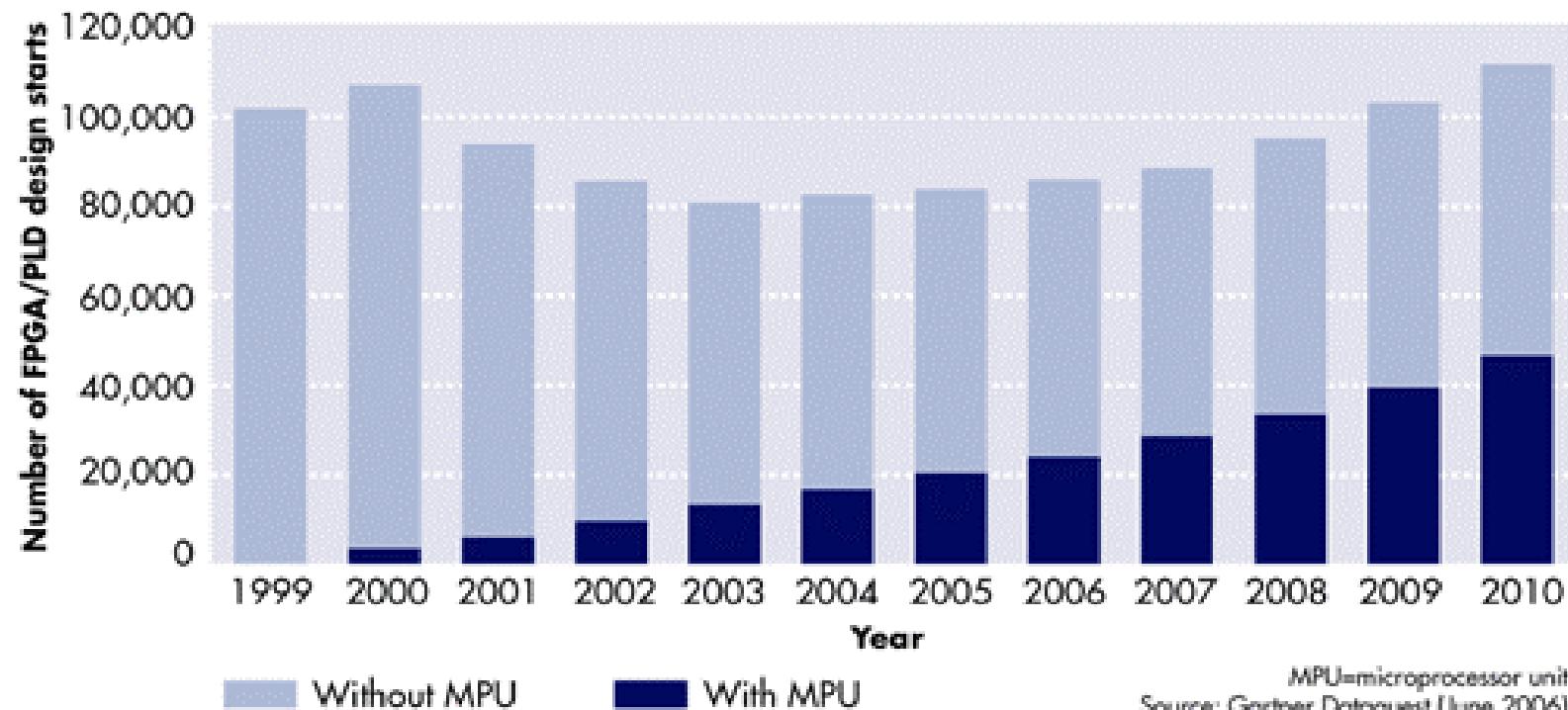
Key Features & Benefits:

- Over 70 user configurable options
- 3-stage pipeline for optimal footprint, 5-stage pipeline for maximum performance
- Supports either PLB or AXI interface
- Big-endian or Little-endian support
- Optional Memory Management Unit (MMU)
- Optional Floating Point Unit (FPU)
- Instruction and Data-side Cache



FPGA with Embedded Microprocessors

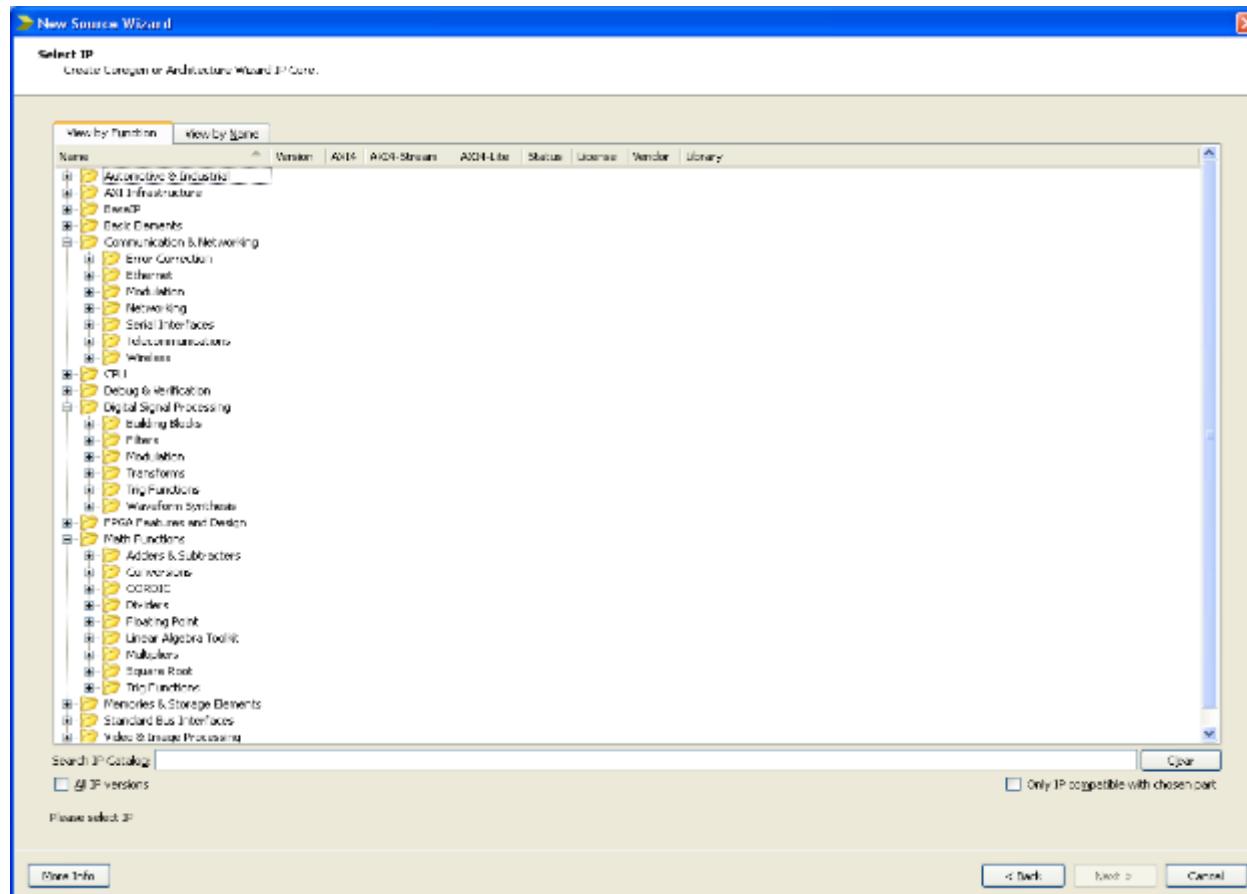
Growth of FPGA-based processing.



- Question: Why?

Xilinx Intellectual Property (IP) Cores

- Dozens of soft IP cores are provided by Xilinx and other vendors, which can be integrated into a custom design



How to Choose an FPGA?

1. Study the problem of interest.
2. Start by a preliminary system design to find a rough estimate of the resources and system clock that you might need. This might need some simulations, writing some parts of the HDL code or putting together predefined libraries or IP cores.
3. Choose/design an appropriate (overestimated) FPGA board for your application.
4. Proceed with the detailed design and implementation

Xilinx 7 Series FPGAs Overview

Max. Capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic Cells	102K	215K	478K	1,955K
Block RAM ⁽¹⁾	4.2 Mb	13 Mb	34 Mb	68 Mb
DSP Slices	160	740	1,920	3,600
DSP Performance ⁽²⁾	176 GMAC/s	929 GMAC/s	2,845 GMAC/s	5,335 GMAC/s
Transceivers	–	16	32	96
Transceiver Speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial Bandwidth	–	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	–	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	800 Mb/s	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	400	500	500	1,200
I/O Voltage	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V
Package Options	Low-Cost, Wire-Bond	Low-Cost, Wire-Bond, Lidless Flip-Chip	Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip

Notes:

1. Additional memory available in the form of distributed RAM.
2. Peak DSP performance numbers are based on symmetrical filter implementation.

Xilinx Virtex-7 Feature Summary

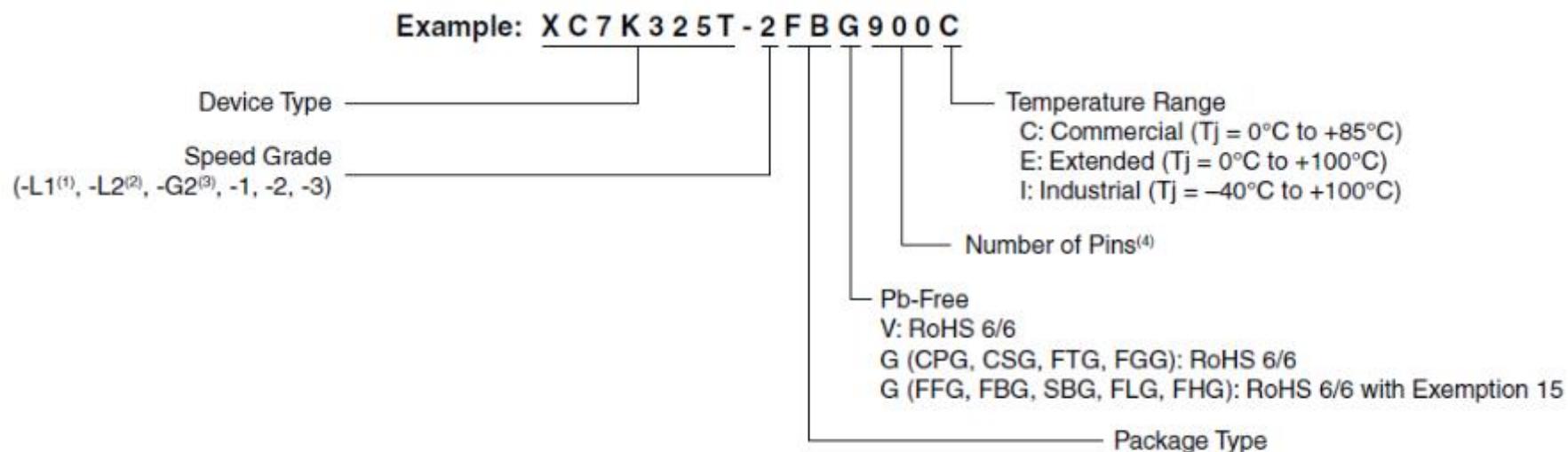
Device ⁽¹⁾	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices ⁽³⁾	Block RAM Blocks ⁽⁴⁾			CMTs ⁽⁵⁾	PCIe ⁽⁶⁾	GTX	GTH	GTZ	XADC Blocks	Total I/O Banks ⁽⁷⁾	Max User I/O ⁽⁸⁾	SLRs ⁽⁹⁾
		Slices ⁽²⁾	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)									
XC7V585T	582,720	91,050	6,938	1,260	1,590	795	28,620	18	3	36	0	0	1	17	850	N/A
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	20	2	0	80	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	900	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
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	6	300	3

Notes:

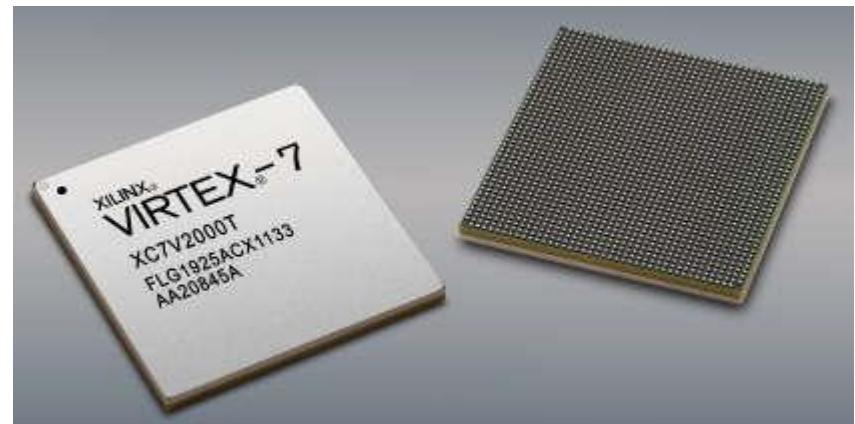
- EasyPath™-7 FPGAs are also available to provide a fast, simple, and risk-free solution for cost reducing Virtex-7 T and Virtex-7 XT FPGA designs.
- Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
- Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
- Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
- Each CMT contains one MMCM and one PLL.
- Virtex-7 T FPGA Interface Blocks for PCI Express support up to x8 Gen 2. Virtex-7 XT and Virtex-7 HT Interface Blocks for PCI Express support up to x8 Gen 3, with the exception of the XC7VX485T device, which supports x8 Gen 2.
- Does not include configuration Bank 0.
- This number does not include GTX, GTH, or GTZ transceivers.
- Super logic regions (SLRs) are the constituent parts of FPGAs that use SSI technology. Virtex-7 HT devices use SSI technology to connect SLRs with 28.05 Gb/s transceivers.

FPGA Package Numbers

FPGA ordering information (visibly marked on the IC package)



- 1) -L1 is the ordering code for the lower power, -1L speed grade.
- 2) -L2 is the ordering code for the lower power, -2L speed grade.
- 3) -G2 is the ordering code for the -2 speed grade devices with higher performance transceivers.
- 4) Some package names do not exactly match the number of pins present on that package.
See UG475: 7 Series FPGAs Packaging and Pinout User Guide for package details.



FPGA DESIGN FLOWS

PLD-Based Design Flows

1. Design Idea
2. Architectural Design
3. Design Entry
4. Behavioral Simulation (Top-Down and/or Bottom-Up)
5. Register Transfer Level (RTL) Simulation/Implementation
6. Synthesis
7. Technology Mapping
8. Placement & Routing
9. FPGA/CPLD Configuration using Bitstreams
10. Final In-System Testing
11. Fully customized IC or ASIC Fabrication
12. Gate Level & Timing Simulation/Implementation
13. Switch Level & Device Simulation/Implementation
14. Final Circuit Testing

For ASIC
Design Only

FPGA vs. Microprocessor Implementation Flow

Microprocessor	FPGA
Architectural design	Architectural design
Choice of language (C, JAVA, etc.)	Choice of language (Verilog, VHDL, etc.)
Editing programs	Editing programs
Compiling programs (.DLL, .OBJ)	Compiling programs
	Synthesizing programs (.EDIF)
Linking programs (.EXE)	Placing and routing programs (.VO, .SDF, .TTF)
Loading programs to ROM	FPGA configuration with bit files
Debugging programs	Debugging FPGA programs
Documenting programs/design	Documenting programs/design
Delivering programs	Delivering programs

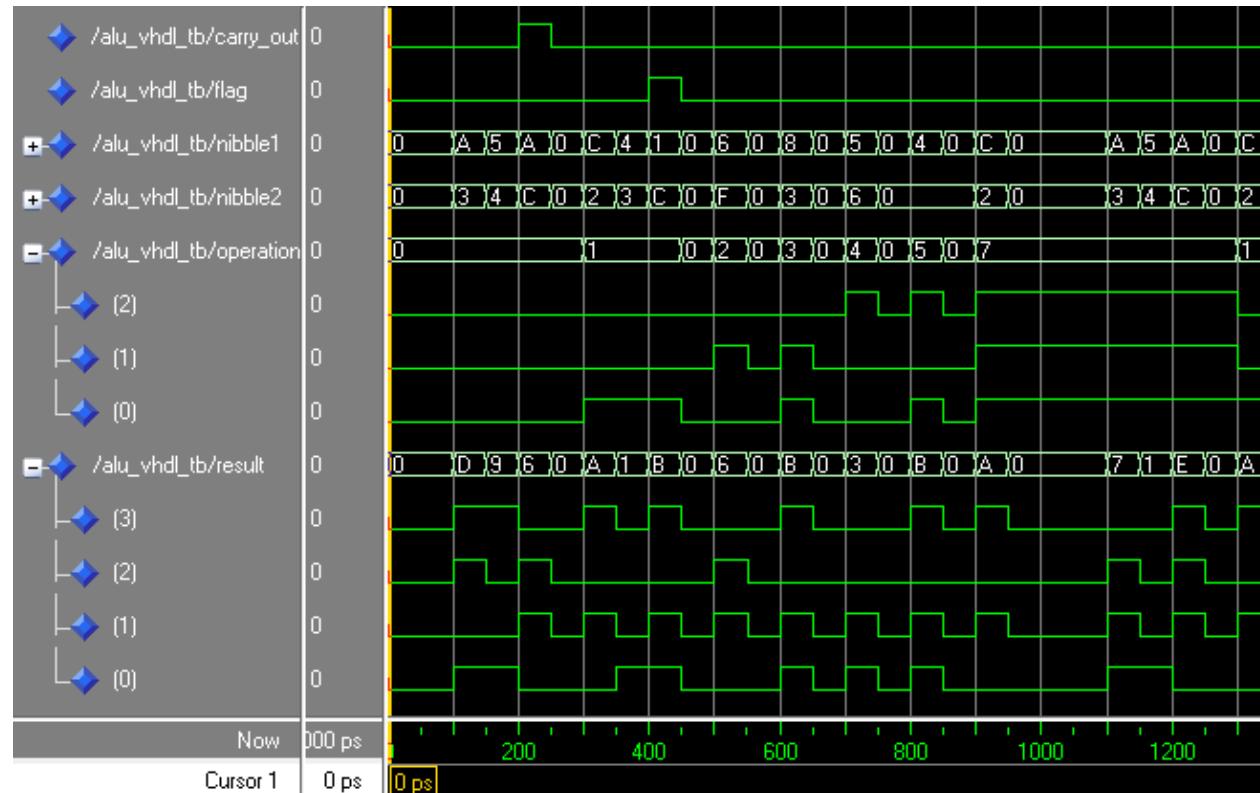
1. Design Entry

Utilities for design entries:

- Schematic Editors
 - e.g., Altium's FPGA-ready Design Components and FPGA Generic
- Hardware Description Languages (HDL)
 - e.g., Verilog, VHDL, etc.
- Finite State Machine (FSM) Editors
 - e.g., ActiveHDL® FSM editor
- System Level Tools, known as HLS
 - e.g., Matlab Simulink and Xilinx System Generator

2. Functional Simulation

- Behavioral Simulation; not necessarily implementable on hardware
- Structural Simulation; can simulate bitwise accurate models of the final hardware



3. Logic Synthesis

HDL → Boolean Equations → Technology Mapping

- The output of the synthesis stage is a Netlist including all the hardware modules and their interconnections
- Various Netlist Standards exist
 - Electronic Design Interchange Format (EDIF)
 - Xilinx Netlist Format (XNF)
 - ...

Necessity of standard tools: Consider ‘N’ vendors with distinct standards; N^2 translators are required to interchange formats in between

Summary of Xilinx FPGA Design Flow

- 1. Synthesis:** converts HDL (VHDL/Verilog) code into a gate-level netlist, represented in the terms of the UNISIM component library (a Xilinx library containing basic primitives).
- 2. Translate:** merges the incoming netlists and constraints into a Xilinx® design file.
- 3. Map:** fits the design into the available resources on the target device, and optionally, places the design.
- 4. Place and Route:** places and routes the design to the timing constraints.
- 5. Generate Programming File:** creates a bitstream file that can be downloaded to the device.

PART II

Electronic Design Automation

HARDWARE DESCRIPTION LANGUAGES

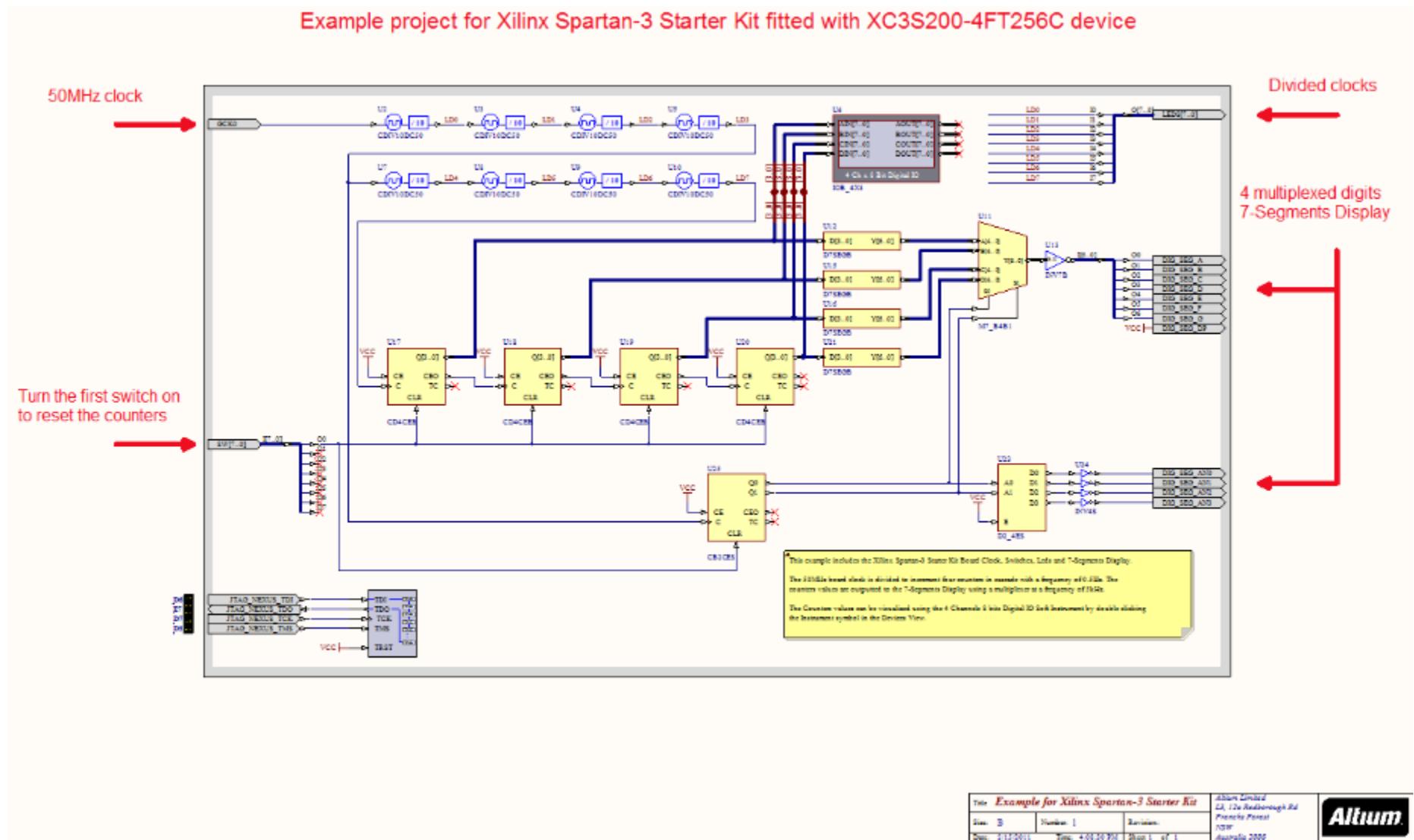
Hardware Description

How can we describe a hardware?

1. **Schematic design tools:** Visual schematic editors. e.g., Altium®, Protel®, OrCAD®, Xilinx PlanAhead®, etc.
2. **Hardware description languages:** Verilog, VHDL, etc.
3. Set of libraries and classes in software languages
4. Any other?

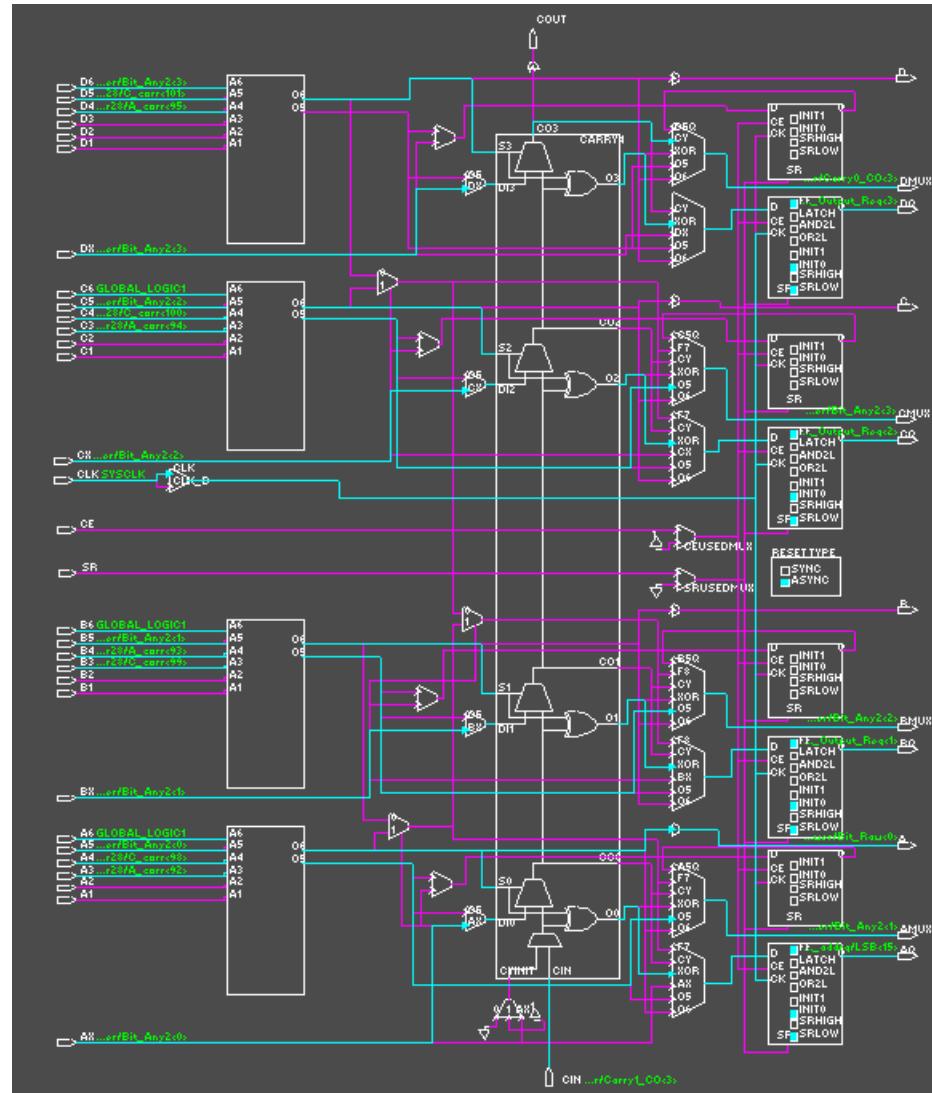
Hardware Description Examples

- Schematic editors: Altium® general FPGA design library



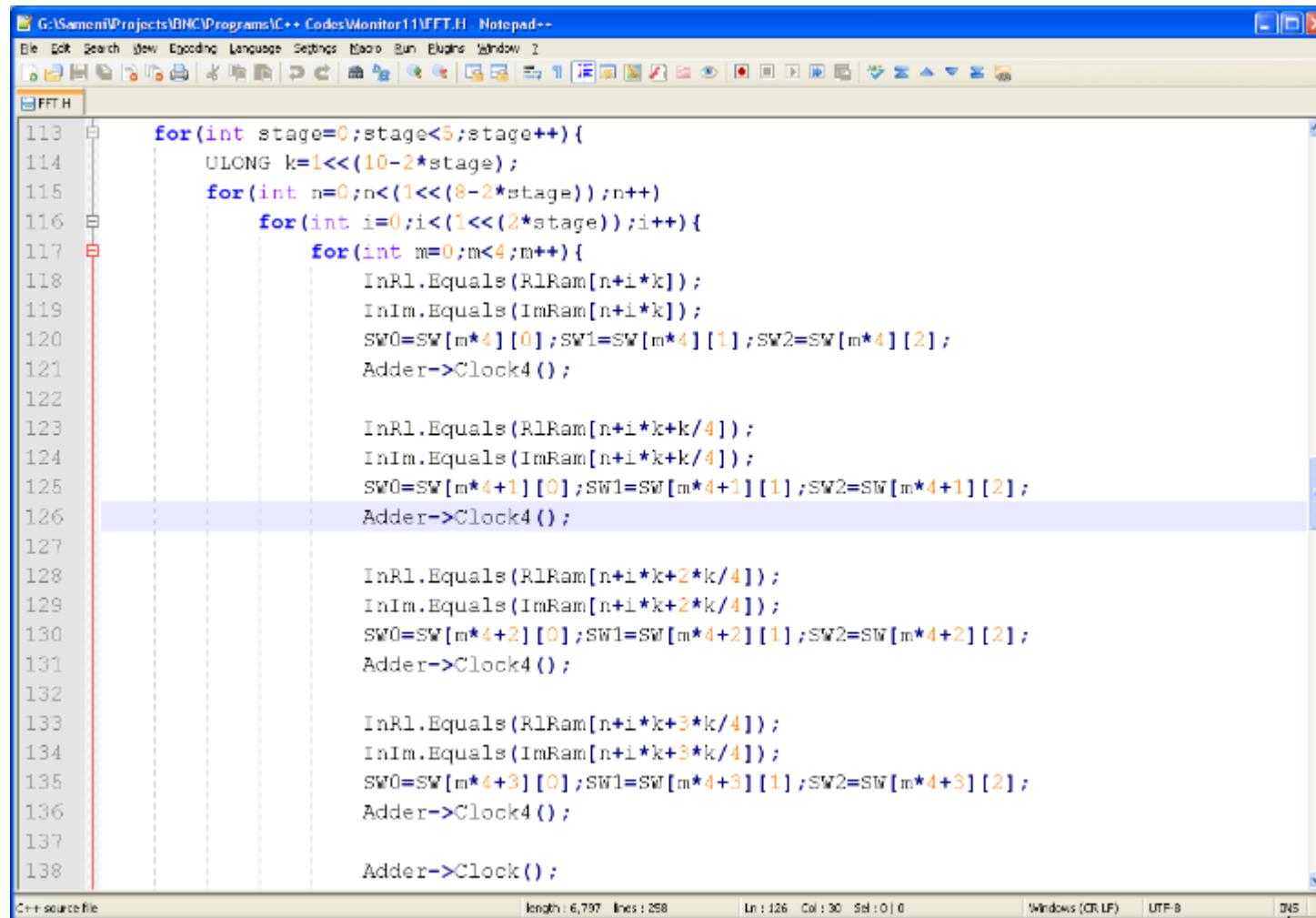
Hardware Description Examples

- Schematic editors: Xilinx Schematic Tools



Hardware Description Examples

- A C++ library to simulate hardware functionality



The screenshot shows a Notepad++ window displaying a C++ source file named FFT.H. The code implements a butterfly computation for an FFT algorithm. It uses nested loops to iterate over stages, points, and bits. The code includes memory access to RL and IM RAM, and calculations involving SW arrays and an adder. The code is annotated with line numbers from 113 to 138.

```
FFT.H
113     for(int stage=0;stage<5;stage++){
114         ULONG k=1<<(10-2*stage);
115         for(int n=0;n<(1<<(8-2*stage));n++){
116             for(int i=0;i<(1<<(2*stage));i++){
117                 for(int m=0;m<4;m++){
118                     InRl.Equals(RlRam[n+i*k]);
119                     InIm.Equals(ImRam[n+i*k]);
120                     SW0=SW[m*4][0];SW1=SW[m*4][1];SW2=SW[m*4][2];
121                     Adder->Clock4();
122
123                     InRl.Equals(RlRam[n+i*k+k/4]);
124                     InIm.Equals(ImRam[n+i*k+k/4]);
125                     SW0=SW[m*4+1][0];SW1=SW[m*4+1][1];SW2=SW[m*4+1][2];
126                     Adder->Clock4();
127
128                     InRl.Equals(RlRam[n+i*k+2*k/4]);
129                     InIm.Equals(ImRam[n+i*k+2*k/4]);
130                     SW0=SW[m*4+2][0];SW1=SW[m*4+2][1];SW2=SW[m*4+2][2];
131                     Adder->Clock4();
132
133                     InRl.Equals(RlRam[n+i*k+3*k/4]);
134                     InIm.Equals(ImRam[n+i*k+3*k/4]);
135                     SW0=SW[m*4+3][0];SW1=SW[m*4+3][1];SW2=SW[m*4+3][2];
136                     Adder->Clock4();
137
138                     Adder->Clock();
```

Hardware Description Examples

- Hardware Description Languages

```

module example /*AUTOARG*/
// Outputs
lower_out, o,
// Inputs
lower_inb, lower_ina, i
);
input i;
output o;
/*AUTOPUT*/
// Beginning of automatic inputs
input lower_ina; // To inst of inst.v
input lower_inb; // To inst of inst.v
// End of automatics
/*AUTOPUT*/
// Beginning of automatic output
output lower_out; // From inst of inst.v
// End of automatics
/*AUTOREG*/
// Beginning of automatic regs
reg          o;
// End of automatics
inst inst /*AUTINST*/
    // Outputs
    .lower_out (lower_out),
    // Inputs
    .lower_inb (lower_inb),
    .lower_ina (lower_ina));
always @ /*AUTONSENSE*/(i) begin
    o = i;
end
endmodule

```

Verilog

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity signed_adder is
6 port
7 (
8     aclr : in  std_logic;
9     clk  : in  std_logic;
10    a   : in  std_logic_vector;
11    b   : in  std_logic_vector;
12    q   : out std_logic_vector
13 );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17 signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20 assert(a'length >= b'length)
21 report "Port A must be the longer vector if different sizes!"
22 severity FAILURE;
23 q <= std_logic_vector(q_s);
24
25 adding_proc:
26 process (aclr, clk)
27 begin
28     if (aclr = '1') then
29         q_s <= (others => '0');
30     elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32     end if; -- clk'd
33 end process;
34
35 end signed adder arch;

```

VHDL

```

// definition of a model using LSF primitive modules
SC_MODULE( my_lsf_model )
{
    sca_lsf::sca_in      in;           // LSF input port
    sca_lsf::sca_out     out;          // LSF output port
    sca_lsf::sca_signal  sig;          // LSF signal

    // constructor with parameter
    my_lsf_model( sc_module_name, double fc=1.0e3 )
    {
        // instantiate pre-defined primitives here
        subl = new sca_lsf::sca_sub( "subl" );
        subl->x1( in );
        subl->x2( sig );
        subl->y( out );

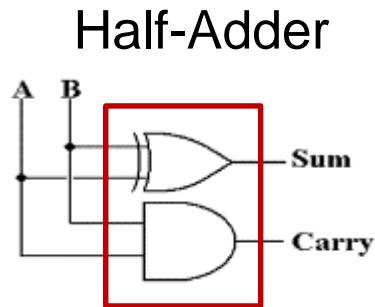
        dot1 = new sca_lsf::sca_dot( "dot1", 1.0/(2.0*M_
PI*fc) );
        dot1->x( out );
        dot1->y( sig );
    }
};


```

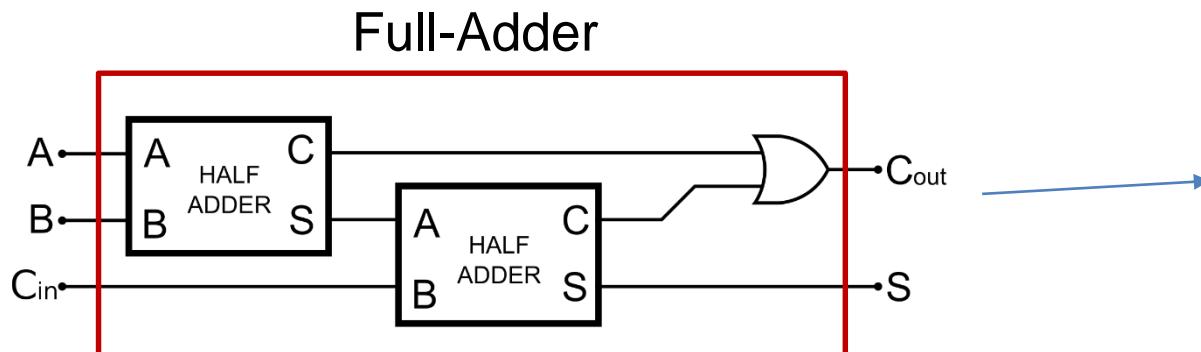
SystemC

From Schematic Editors to Hardware Description Languages

- Hardware description languages are textual means of describing a hardware
- Text is better than pictures and Karnaugh maps; as it's more simple to handle and analyze for language parsers and synthesis tools



```
// HalfAdder
module HalfAdder(A, B, S, C);
    input A, B;
    output S, C;
    xor sum(S, A, B);
    and Carry(C, A, B);
endmodule
```



```
// FullAdder
module FullAdder(A, B, Cin, S, Cout);
    input A, B, Cin;
    output S, Cout;
    wire S1, C1, C2;
    HalfAdder HA1(A, B, S1, C1);
    HalfAdder HA2(S1, Cin, S, Cout);
    or Carry(Cout, C1, C2);
endmodule
```

Hardware Description Languages (HDL)

Examples of HDL languages

- VHDL
- Verilog
- SystemC
- SystemVerilog
- JHDL
- Handel-C
- Impulse C
- ...

Hardware Description

What should a HDL look like and what features should it have?

1. Cover different **levels of abstraction**: transistor level, gate level, **register transfer level (RTL)**, system level
2. Applicable for different architectures: **CPLD**, **FPGA**, **ASIC**, etc.
3. Provide a unique description for all synthesizable hardware
4. Ability of accurate simulation before implementation. The language should be able to simulate other functionalities required for hardware description and simulation: **generating synthetic waveforms**, **reading/writing test vectors from/to files**, **setting time bases**, etc.
5. Convertible into conventional data structures such as **trees** and **graphs** for algorithmic simplifications and optimizations
6. Existence of tools (tool chains) for translating the “hardware description” into “hardware”

Current HDL Languages

- Common HDL languages support different levels of abstraction plus additional features used for **simulation**, **modeling**, and **documentation** of hardware (not necessarily synthesizable on hardware)
- The languages can be used for hardware: design, simulation, modeling, test, documentation
- **Note:** HDL languages *do not* generate executable codes; they describe **hardware**, which are later translated into hardware by electronic design automation (EDA) tools

Verilog HDL

We use Verilog HDL in this course, because

- It has all the required features of a complete HDL
- It has a rather simple syntax
- It is not as verbose as VHDL
- It is highly popular in industry (for RTL design)

Our major references:

- S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Ed., 2003
- WF Lee, *Verilog Coding for Logic Synthesis*, John-Wiley, 2003
- *Xilinx XST user's manual*, 2009

Verilog HDL History

- Verilog was created by **Prabhu Goel, Phil Moorby, Chi-Lai Huang and Douglas Warmke** in 1983-1984, as a **hardware modeling language**.
- Verilog was originally owned by **Automated Integrated Design Systems** (later renamed as **Gateway Design Automation**).
- Gateway Design Automation was purchased by **Cadence Design Systems** in 1989.
- In 1990, Cadence put the language into the public domain, with the intention that it should become a standard, non-proprietary language.
- Versions: **Verilog-95, Verilog 2001, Verilog 2005, SystemVerilog** (a superset of Verilog 2005).
- Latest versions of Xilinx ISE® support Verilog 2005.
- Xilinx Vivado® supports Verilog 2005 and SystemVerilog.

Verilog HDL Syntax

Let's start with a list of the most common digital hardware elements that we know:

- Gates: AND, OR, NOT, XOR,...
- Electronic features/elements: Wires, buffers, tristate buffers, impedance levels,...
- Multiplexers, encoders and decoders
- Finite state machines (FSM)
- Memories: RAM, ROM, dual-port vs. single port memories
- Shift registers, Barrel shifters, etc.
- Initialization and resetting mechanisms
- Combinational logic: a combination of logical components
- Sequential logic: registers (flip-flops)
- Arithmetic units: half and full adders, multipliers, counters, timers, etc.
- Logic chips: ICs with predefined timing and digital function
- Logic circuit peripherals: I/O interface, clock management
- User-defined constraints and port mapping

These elements are from different levels of abstraction; but any HDL should be able to “describe” them.

Verilog HDL Syntax

- Verilog is a **free-form language** (the positioning of characters on the programming page is insignificant)
- Combinations of numbers (`0, 1, 2, ...`), letters (`a, b, ..., z, A, B, C, ..., Z`), underscore (`_`) and Dollar-sign (`$`) can be used in variables.
- Variable names are case-sensitive
- Variable names may not start with `$` (Verilog system commands start with `$`)
- Underscores can be used between numbers as separators for better readability
- Single-line comments: `// All text is considered as comment hereafter`
- Single- or multi-line comment blocks: `/* bla bla bla */`
- Synthesis tools occasionally use comment blocks to define **synthesis attributes** (user-defined properties of a block of code) in specific formats:

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

```
LUT4 U1 (.O(O), .I0(I0), .I1(I1), .I2(I2), .I3(I3));
// synthesis attribute INIT of U1 is "8000"
```

Nets and Registers

- The `wire` keyword is used to define nets (wires) and results of **combinational logic** (using an `assign` command).
- The `reg` keyword is used to define registers and results of **sequential logic** (in an `always` block). The exception is a combinational logic defined by an `always` block.
- Wires and registers can be defined and assigned in vector form.

```
    wire a;
    wire b, c;
    wire [15:0]address;

    reg x;
    reg [7:0] bus; // little-endian form
    reg [0:5] data; // big-endian form
    reg [3:0] port1, port2;
```

Logic Values

- Verilog supports four logic values

Logic Value	Description/Usage
0	zero, low or false
1	one, high or true
z or Z	high-impedance, tristates, floating (dangling)
x or X	unknown, uninitialized, collision

- Sized vs. unsized values

```

wire [3:0] u = 4'b1010;          // sized binary
wire [7:0] x = 8'o2130;          // sized octal
wire [11:0] z = 12'd35;          // sized decimal
wire [19:0] y = 20'h12fa7;        // sized hexadecimal
wire [31:0] w = 'd35;            // unsized; NOT RECOMMENDED
wire [31:0] v = 75;              // unsized decimal (by default); NOT RECOMMENDED

wire [7:0] a = 8'b10110010;
wire [5:0] b = 6'b11_01_00; // with digit separator
wire [4:0] c = 5'b1xx0z;      // x and z values

```

Assignments

- The `assign` keyword is used to connect wires and to define single-line combinational logic.

```
wire A, B;  
reg C;  
  
wire Y = A | B; // implicit assignment  
  
wire X;  
assign X = A ^ (B & C); // explicit assignment  
  
wire [2:0] Z;  
assign Z = {A, B, C}; // Concatenation
```

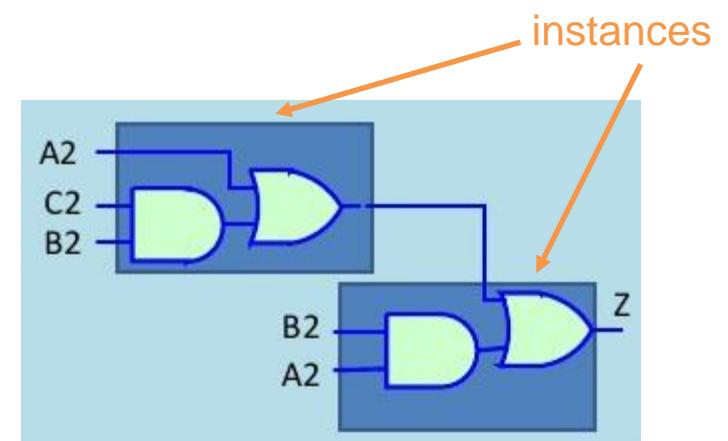
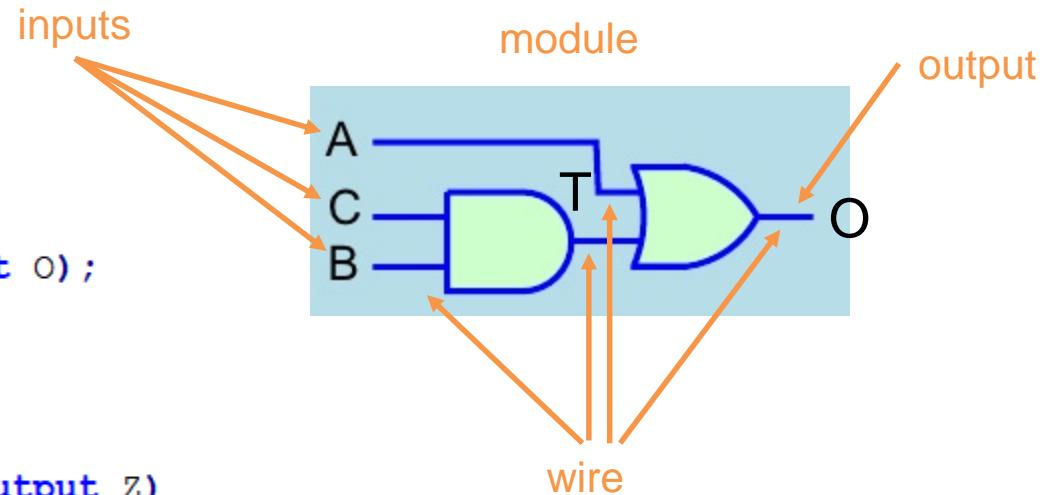
Module Definition

```

1 // Sample module definition
2 module gate(input A, input B, input C, output O);
3     wire T;
4     assign T = C & B;
5     assign O = A | T;
6 endmodule
7
8 // Alternative form:
9 module gate(input A, input B, input C, output O);
10    assign O = A | (C & B);
11 endmodule
12
13 // Module instantiation
14 module gates(input A2, input B2, input C2, output Z)
15     wire temp;
16
17     gate g1(.A(A2), .B(B2), .C(C2), .O(temp));
18     gate g2(.A(temp), .B(B2), .C(A2), .O(Z));
19
20 endmodule
  
```

module name

Instance name

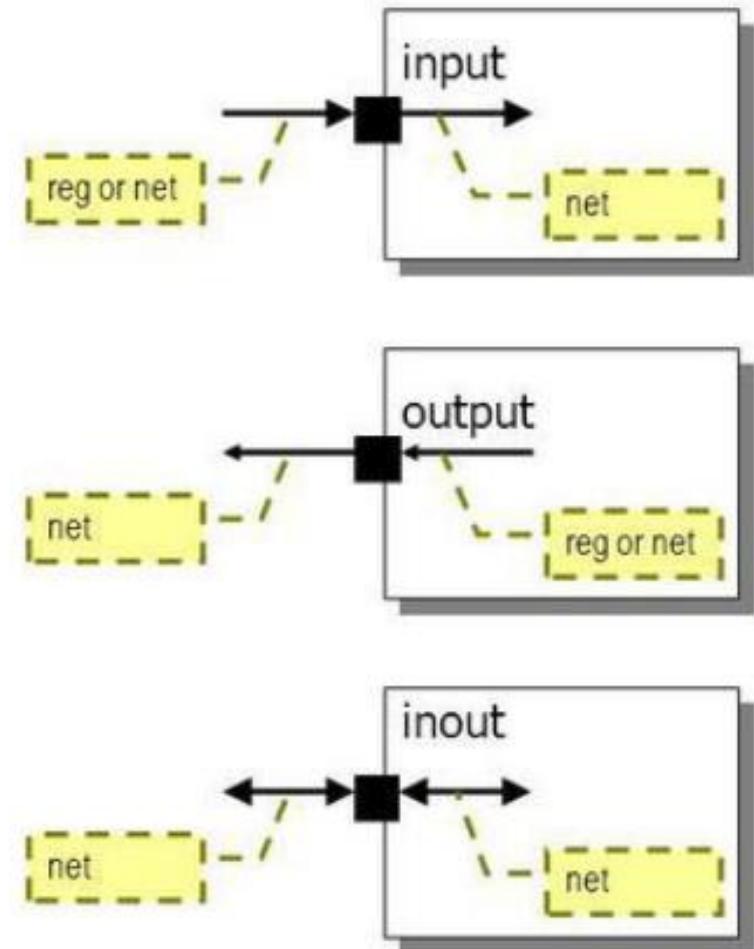


Note: A module may not be defined inside another module; but it can be instantiated.

Module Port Types

Three types of ports are available in Verilog:

1. `input`: for giving input to a module
2. `output`: for getting outputs from a module
3. `inout`: bidirectional ports which can send/receive data depending on a control line. Inouts ports should be realized using tristate buffers with appropriate control.

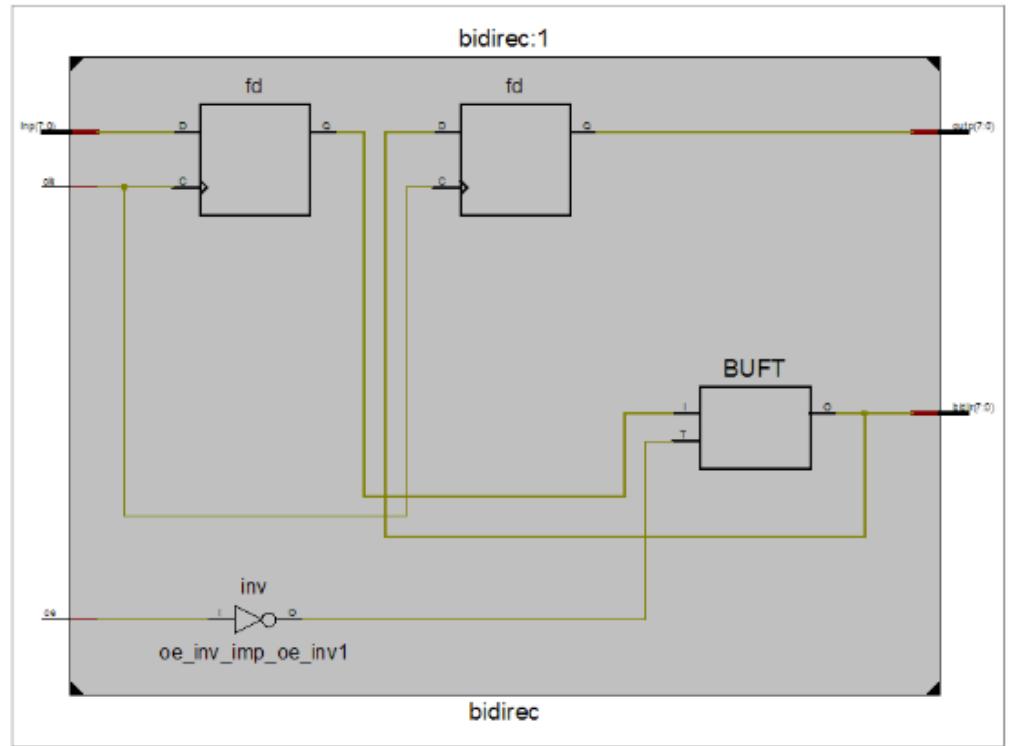


Input, Output, and Inout Port Usage in Verilog

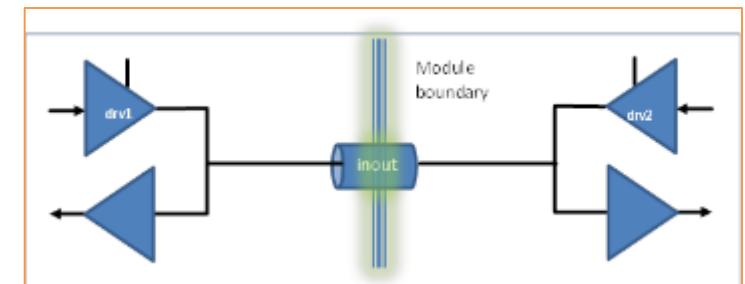
```

1 module bidirec (oe, clk, inp, outp, bidir);
2   // Port Declaration
3   input  oe;
4   input  clk;
5   input  [7:0] inp;
6   output [7:0] outp;
7   inout  [7:0] bidir;
8
9   reg   [7:0] a;          realizes a tri-state buffer
10  reg   [7:0] b;
11
12  assign bidir = oe ? a : 8'bZ ;
13  assign outp  = b;
14
15 // Always Construct
16 always @ (posedge clk) begin
17   b <= bidir;
18   a <= inp;
19 end
20 endmodule

```



Inout mechanism



Module Instance Port Mapping

- Two types of instance port mapping are supported in Verilog:

```

1 module logical(input A, input B, input C, output O);
2     assign O = A ^ (C & B);
3 endmodule
4
5 module gates(input I1, input I2, input I3, output Out)
6     wire temp;
7
8     // Instance port mapping by order:
9     logical L1 (I1, I2, I3, temp);           unconnected (dangling) port
10
11    // Instance port mapping by name:
12    logical L2 (.A(temp), .C(I1), .B(), .O(Out));
13
14 endmodule

```

Note: Port order is not important when using “by name” mapping

Note: All module ports (`input`, `output`, `inout`) are wires

Module Port Declaration

- Two forms of port declaration are possible:

```
1 // Port list and types declared separately:  
2 module adder(x, y, c_in, sum, c_out);  
3     input [3:0] x, y;  
4     input c_in;  
5     output [3:0] sum;  
6     output c_out;  
7     //...  
8     reg c_out; <-- Note the difference  
9     //...  
10 endmodule  
11  
12 // Port list and types declared at once (like in ANSI C):  
13 module adder(  
14     input [3:0] x,  
15     input [3:0] y,  
16     input c_in,  
17     output [3:0] sum,  
18     output reg c_out  
19 );  
20 /*  
21     module body  
22 */  
23 endmodule
```

Comments

Built-in and Device-Dependent Primitive Elements

- Verilog has several built-in primitive elements (switches, gates, etc.), which can be instantiated as modules: `and, nand, not, nor, or, xor, xnor, buf, bufif0, bufif1, rtranif1, nmos, pmos, rpmos, tran, rtran, pullup, pulldown, cmos, rnmos, tranif1, tranif0, notif0, notif1, rtranif0, rcmos`

Example:

```

strength levels           delay parameters
and (strong1, weak0) #(1,2) gate1(out, in1, in2); // with parameters
and a(out, in1, in2, in3);                                // with default parameters

```

- There are also device- and technology-dependent primitives:

```

LDCE LDCE_inst (
    .Q  (Q),      // Data output
    .CLR (CLR),   // Asynchronous clear/reset input
    .D   (D),      // Data input
    .G   (G),      // Gate input
    .GE  (GE) );  // Gate enable input

```

Always Blocks

- An `always` block is used to define, both, combinational and sequential logic blocks.
- Registers may only be assigned inside an `always` block (although they may represent combinational logic).
- Variables assigned in an `always` block should all be defined as `reg`

```

1 // Sequential logic
2 reg x;
3 always @ (posedge clock) begin
4     x = a & b;
5 end
6
7 // Combinational logic
8 reg y;
9 always @ (a or b) begin
10    y = a & b;
11 end

```

Flip-flop inferred → `reg x;`

No flip-flops inferred! → `reg y;`

→ **sensitivity list**: `@ (posedge clock)` vs `@ (a or b)`

→ **wire y;** vs **reg y;**

→ **assign y = a & b;** vs **y = a & b;**

→ **equivalent**: `reg y;` vs `wire y;`

→ **equivalent**: `assign y = a & b;` vs `y = a & b;`

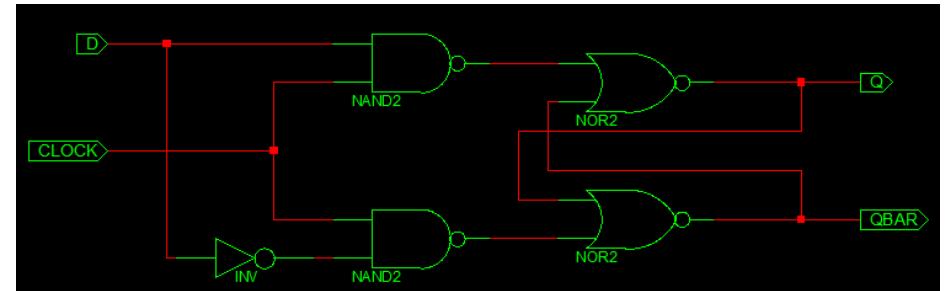
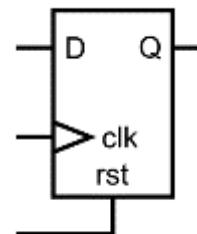
D-Type Flip-Flops (A Review)

- D-type flip-flops are the basic elements used for sequential logic design.

```

1 // D-Type Flip-Flop
2 module D_FF (Q, D, CLOCK);
3   output Q;
4   input D, CLOCK;
5   reg Q;
6   always @(posedge CLOCK)
7     Q = D;
8 endmodule

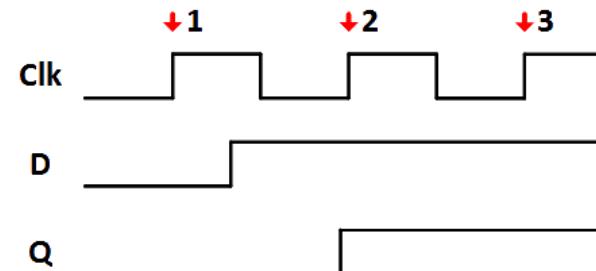
```



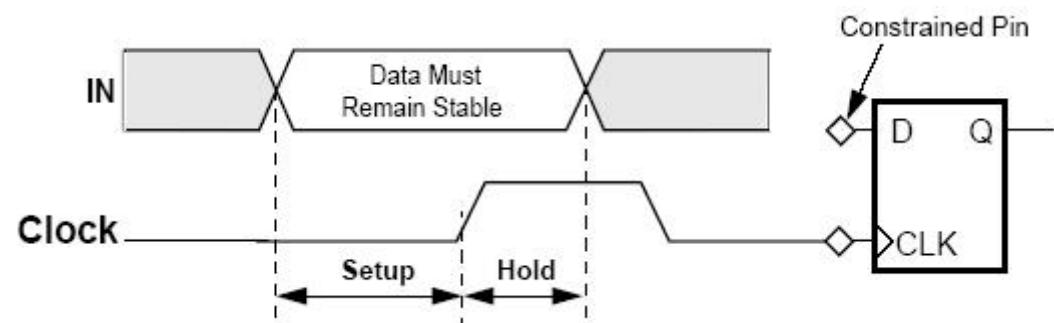
```

1 library ieee;
2 use ieee. std_logic_1164.all;
3 use ieee. std_logic_arith.all;
4 use ieee. std_logic_unsigned.all;
5
6 entity D_FF is
7 PORT(  D,CLOCK: in std_logic;
8        Q: out std_logic);
9 end D_FF;
10
11 architecture behavioral of D_FF is begin
12   process(CLOCK) begin
13     if(CLOCK='1' and CLOCK'EVENT) then
14       Q <= D;
15     end if;
16   end process;
17 end behavioral;

```



Q	D	Q(T+1)
0	0	0
0	1	1
1	0	0
1	1	1

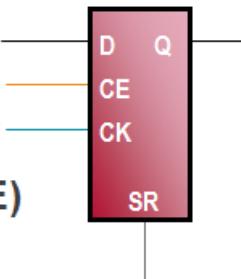


Xilinx D-Type Flip-Flops

- According to Xilinx 7 Series Manual:

Slice Flip-Flop Capabilities

- All flip-flops are D type
- All flip-flops have a single clock input (CLK)
 - Clock can be inverted at the slice boundary
- All flip-flops have an active high chip enable (CE)
- All flip-flops have an active high SR input
 - SR can be synchronous or asynchronous, as determined by the configuration bit stream
 - Sets the flip-flop value to a pre-determined state, as determined by the configuration bit stream
- All flip-flops are initialized during configuration



Always Blocks

- The following two pieces of code are identical (five flip-flops are inferred in total):

```

1 reg data;
2 reg data_log
3 reg [3:0] bus;
4 reg [3:0] bus_log;
5
6 always @ (posedge clock) begin
7   data_log <= data;
8   bus_log <= bus;
9 end

```



```

1 always @ (posedge clock) data_log <= data;
2 always @ (posedge clock) bus_log[0] <= bus[0];
3 always @ (posedge clock) bus_log[1] <= bus[1];
4 always @ (posedge clock) bus_log[2] <= bus[2];
5 always @ (posedge clock) bus_log[3] <= bus[3];

```

- We see that the always block has abbreviated the explicit declaration of five flip-flops
- Note:** All always procedures with the same sensitivity list are concurrent. They describe parallel flip-flops, which share a common clock.
- Note:** The sequence of writing wire assignments, always blocks and their internal assignments are irrelevant; *timing is managed by data-flow and state controllers, not by code line execution orders*
- Question:** What issues can raise when code line sequences become irrelevant?

Always Block Issues (1)

Question 1: What happens if a single variable is simultaneously assigned in multiple always blocks?

The diagram illustrates a Verilog code snippet with two overlapping always blocks. The top block is defined by the keyword `always`, followed by a sensitivity list `@(posedge clk1)`, and an assignment statement `data = A;`. The bottom block is also defined by `always`, followed by `@(posedge clk2)`, and `data = B;`. A large red 'X' is drawn across both blocks, indicating that this code is invalid according to Verilog syntax rules.

Answer: The Verilog syntax does not allow this (a register may only be assigned in a single always block). Problem solved!...

Question: What if we really need to change a register value by two different clock?

Answer: We need to find another way for this later (using indirect assignments).

Always Block Issues (2)

Question 2: What happens if data dependency exists between two register assignments?

```
always @ (posedge clock) begin  
    temp = datain;  
    dataout = temp;  
end
```

vs.

```
always @ (posedge clock) begin  
    dataout = temp;  
    temp = datain;  
end
```

Answer: Race condition; we need to find a solution.

Solution: Verilog has two different assignment operators: **Blocking** and **Non-blocking**

```
always @ (posedge clock) begin // Blocking assignments
```

```
    temp = datain;  
    dataout = temp;  
end
```

```
always @ (posedge clock) begin // Non-blocking assignments
```

```
    temp <= datain;  
    dataout <= temp;  
end
```

Blocking vs. Non-Blocking Assignments

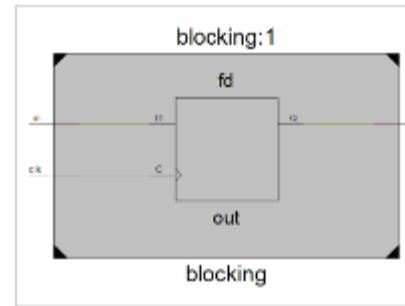
Syntactic difference:

Blocking assignment: Evaluation and assignment are somehow immediate (blocks all other assignments and evaluations that use the same variable)

```

1 module blocking(in, clk, out);
2   input in, clk;
3   output out;
4   reg q1, q2, out;
5   always @ (posedge clk) begin
6     q1 = in;
7     q2 = q1;
8     out = q2;
9   end
10 endmodule

```

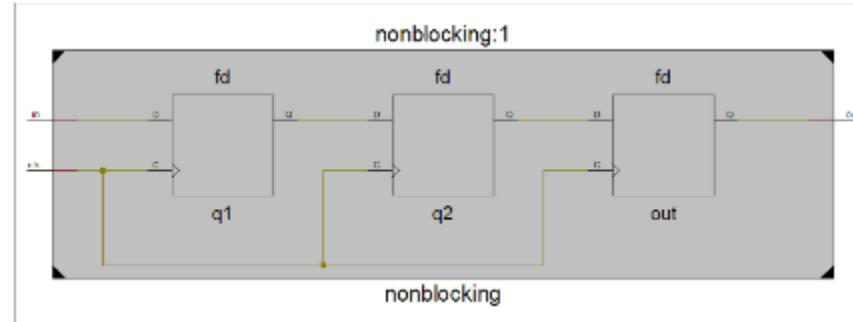


Non-blocking assignment: All assignments that use the variable are deferred until all right-hand sides have been evaluated (end of simulation time-step)

```

1 module nonblocking(in, clk, out);
2   input in, clk;
3   output out;
4   reg q1, q2, out;
5   always @ (posedge clk) begin
6     q1 <= in;
7     q2 <= q1;
8     out <= q2;
9   end
10 endmodule

```



Guideline: Blocking assignments are only used for combinational logic description. Use non-blocking assignments for sequential register assignment.

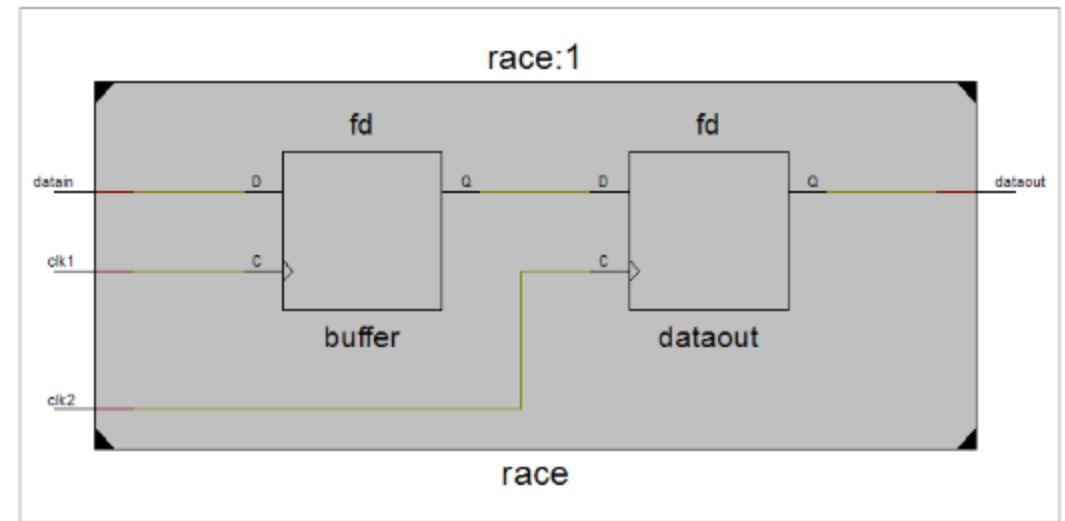
Always Block Issues (3)

Question: What happens if two always blocks (with different sensitivity lists) have data dependency between their register assignments?

```

1 module race(
2     input clk1,
3     input clk2,
4     input datain,
5     output reg dataout
6 );
7 reg buffer;
8 always @ (posedge clk1)
9     buffer <= datain;
10 always @ (posedge clk2)
11     dataout <= buffer;
12 endmodule

```



Answer: Race condition; no syntactic solutions exist for this issue. Should be avoided/resolved by proper design.

Example: Passing data between different **clock domains**.

Signal Drive Strength

- In logic circuit design, nets can have different strength levels ranging from supply/ground (strongest) to high-impedance (weakest).
- Verilog supports various strength levels to model the driving strength phenomenon:

Strength level	Description	Keywords	Degree
Supply drive	Power supply connections	<code>supply0, supply1</code>	7 (strongest)
Strong drive	Default gate and assign output strength	<code>strong0, strong1</code>	6
Pull drive	Gate and assign output strength	<code>pull0, pull1</code>	5
Large capacitor	Size of <code>trireg</code> net capacitor	<code>large</code>	4
Weak	Gate and assign output strength	<code>weak0, weak1</code>	3
Medium capacitor	Size of <code>trireg</code> net capacitor	<code>medium</code>	2
Small capacitor	Size of <code>trireg</code> net capacitor	<code>small0, small1</code>	1
High impedance	High Impedance	<code>highz0, highz1</code>	0 (weakest)

Signal Strength Collisions

	supply1	strong1	pull1	weak1	highz1
supply0	x	0	0	0	0
strong0	1	x	0	0	0
pull0	1	1	x	0	0
weak0	1	1	1	x	0
highz0	1	1	1	1	z

Signal Drive Strength Examples

Logic strength levels

```
1 // Example 1: Instance of and gate with strong1 strength and weak0 strength specified.  
2     and (strong1, weak0) b(o, i1, i2);  
3  
4 // Example 2: The charge strength declaration for trireg net.  
5     trireg (medium) t;  
6  
7 // Example 3: buffers  
8     buf (strong1, weak0) g1 (y, a);  
9     buf (pull1, supply0) g2 (y, b);  
10    /* If a = 0 and b = 0 then y will be 0 with supply strength  
11       If a = 0 and b = 1 then y will be 1 with pull strength  
12       If a = 1 and b = 0 then y will be 0 with supply strength  
13       If a = 1 and b = 1 then y will be 1 with strong strength */  
14  
15 // Example 4:  
16     buf (strong1, weak0) g1 (y, a);  
17     buf (strong1, weak0) g1 (y, b);  
18     /* If a = 0 and b = 0 then y will be 0 with weak strength.  
19       If a = 1 and b = 1 then y will be 1 with strong strength. */  
20  
21 // Example 5: If a = 1 and b = 0 then y will be x with strong strength.  
22     buf (strong1, weak0) g1 (y, a);  
23     buf (weak1, strong0) g1 (y, b);  
24  
25 //Example 6:  
26     bufif0 (strong1, weak0) g1 (y, i1, ctrl);  
27     bufif0 (strong1, weak0) g2 (y, i2, ctrl);  
28  
29 // Example 7:  
30     and (strong1, weak0) u1 (Q, A, B);  
31     trireg (small) c1;  
32     assign (weak1, strong0) Q = A + B;
```

Parameterized Module Design

- Verilog supports parametric module definitions
- **Example 1:** A parametric-length multiplexer

```
1 // 2-to-1 multiplexer, W-bit data
2 module mux2 #(parameter W=1)// data width, default 1 bit
3     (input [W-1:0] a,b, input sel, output [W-1:0] z);
4     assign z = sel ? b : a;
5     assign zbar = ~z;
6 endmodule
7
8 // 4-to-1 multiplexer, W-bit data
9 module mux4 #(parameter W=1) // data width, default 1 bit
10    (input [W-1:0] d0,d1,d2,d3, input [1:0] sel, output [W-1:0] z);
11    wire [W-1:0] z1,z2;
12    mux2 #(.W(W)) m1(.sel(sel[0]), .a(d0), .b(d1), .z(z1));
13    mux2 #(.W(W)) m2(.sel(sel[0]), .a(d2), .b(d3), .z(z2));
14    mux2 #(.W(W)) m3(.sel(sel[1]), .a(z1), .b(z2), .z(z));
15 endmodule
16
17 // The top-module looks like this:
18 module topmodule (d0, d1, d2, d3, sel, z);
19     parameter WIDTH = 5; // here's where all sub-module bit-widths are specified
20     input [WIDTH-1 : 0] d0, d1, d2, d3;
21     input [1 : 0] sel;
22     output [WIDTH-1 : 0] z;
23     mux4 #(.W(WIDTH)) mux4instance (.d0(d0), .d1(d1), .d2(d2), .d3(d3), .sel(sel), .z(z));
24 endmodule
```

Parameterized Module Design

Example 2: A parametric full-adder

```

1  /*
2   Full Adder Module with Parameter
3   Written by referencedesigner.com
4 */
5 module fulladder (in1, in2, cin, sum, cout);
6   parameter N = 4;
7   input wire [N-1:0] in1 , in2 ;
8   input wire cin;
9   output wire [N-1:0] sum;
10  output wire cout ;
11
12  wire [N:0] tempsum ;
13
14  assign tempsum = {1'b0, in1} + {1'b0, in2} + cin ;
15  assign sum = tempsum[N-1:0] ;
16  assign cout= tempsum[N] ;
17 endmodule

```

```

1  /*
2   Full Adder Module parameter instantiation
3 */
4 `timescale 1ns / 100ps
5 module fulladdertb;
6
7 reg [3:0] input1;
8 reg [3:0] input2;
9 reg carryin;
10
11 wire [3:0] out;
12 wire carryout;
13
14 fulladder #(4) uut (
15   .in1(input1),
16   .in2(input2),
17   .cin(carryin),
18   .sum(out),
19   .cout(carryout)
20 );

```

For-Loops in Verilog

- For-loops in their software-like usage are not synthesizable in Verilog.
- **Question:** Why?
- In synthesizable Verilog codes, for-loops are merely used for writing shorter **scripts** that generate codes.
- We will learn alternative code generation methods in later sections.

```

1 module for_loop_synthesis (i_Clock);
2   input i_Clock;
3   integer ii=0;
4   reg [3:0] r_Shift_With_For = 4'h1;
5   reg [3:0] r_Shift-Regular = 4'h1;
6
7   // Performs a shift left using a for loop
8   always @ (posedge i_Clock)
9     begin
10    for(ii=0; ii<3; ii=ii+1)
11      r_Shift_With_For[ii+1] <= r_Shift_With_For[ii];
12  end
13
14  // Performs a shift left using regular statements
15  always @ (posedge i_Clock)
16    begin
17      r_Shift-Regular[1] <= r_Shift-Regular[0];
18      r_Shift-Regular[2] <= r_Shift-Regular[1];
19      r_Shift-Regular[3] <= r_Shift-Regular[2];
20    end
21 endmodule
22
23
24 module for_loop_synthesis_tb (); // Test-bench
25   reg r_Clock = 1'b0;
26   // Instantiate the Unit Under Test (UUT)
27   for_loop_synthesis UUT (.i_Clock(r_Clock));
28   always
29     #10 r_Clock = !r_Clock;
30 endmodule

```

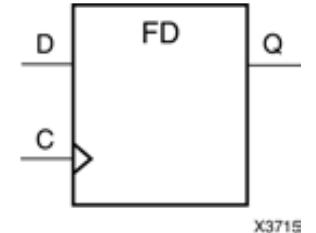
For-Loops in VHDL

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity Example_For_Loop is
6     port (i_Clock : std_logic);
7 end Example_For_Loop;
8
9 architecture behave of Example_For_Loop is
10    signal r_Shift_With_For : std_logic_vector(3 downto 0) := X"1";
11    signal r_Shift-Regular : std_logic_vector(3 downto 0) := X"1";
12 begin
13     -- Creates a Left Shift using a For Loop
14     p_Shift_With_For : process (i_Clock)
15     begin
16         if rising_edge(i_Clock) then
17             for ii in 0 to 2 loop
18                 r_Shift_With_For(ii+1) <= r_Shift_With_For(ii);
19             end loop; -- ii
20         end if;
21     end process;
22
23     -- Performs a shift left using regular assignments
24     p_Shift_Without_For : process (i_Clock)
25     begin
26         if rising_edge(i_Clock) then
27             r_Shift-Regular(1) <= r_Shift-Regular(0);
28             r_Shift-Regular(2) <= r_Shift-Regular(1);
29             r_Shift-Regular(3) <= r_Shift-Regular(2);
30         end if;
31     end process;
32 end behave;
```

Hardware Description Language (HDL) Standard Coding Techniques

- In the sequel we study standard design entries and coding styles, which guarantee synthesizable codes for low-level implementation using EDA tools.
- **Major Reference:** Xilinx XST User Guide, UG627 (v 11.3) September 16, 2009.
URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf

Flip-Flop with Positive-Edge Clock



Verilog

```

1 //          // Flip-Flop with Positive-Edge Clock
2 //
3 module v_registers_1 (C, D, Q);
4   input C, D;
5   output Q;
6   reg Q;
7   always @(posedge C)
8   begin
9     Q <= D;
10    end
11 endmodule

```

VHDL

```

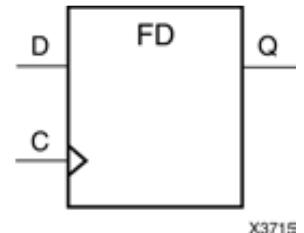
1 --          -- Flip-Flop with Positive-Edge Clock
2 --
3 library ieee;
4 use ieee.std_logic_1164.all;
5 entity registers_1 is
6   port(C, D : in std_logic;
7         Q : out std_logic);
8 end registers_1;
9 architecture archi of registers_1 is
10 begin
11   process (C)
12   begin
13     if (C'event and C='1') then
14       Q <= D;
15     end if;
16   end process;
17 end archi;

```

if (rising_edge(C)) then



Flip-Flop with Positive Edge Clock with INITSTATE of the Flop Set



Verilog

```

1 module test(d, c, q);
2   input d;
3   input c;
4   output q;
5   reg qtemp = 'b1 ;
6   always @ (posedge c)
7     begin
8       qtemp = d;
9     end
10    assign q = qtemp;
11 endmodule

```

Drawback: Only provides power-on initialization, does not have run-time resetting property.

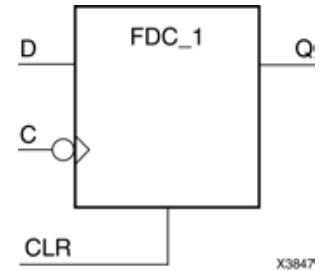
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity registers_1 is
4   port(c, d : in std_logic;
5        q : out std_logic);
6 end registers_1;
7 architecture archi of registers_1 is
8 signal qtemp : std_logic := '1';
9 begin
10 process (c)
11 begin
12   if (c'event and c='1') then
13     qtemp <= d;
14   end if;
15   q <= qtemp;
16 end process;
17 end archi;

```

Flip-Flop with Negative-Edge Clock and Asynchronous Reset



Verilog

```

1 module v_registers_2 (C, D, CLR, Q);
2   input C, D, CLR;
3   output Q;
4   reg Q;
5   always @ (negedge C or posedge CLR)
6   begin
7     if (CLR)
8       Q <= 1'b0;
9     else
10      Q <= D;
11   end
12 endmodule

```

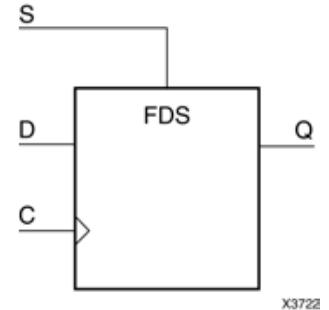
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity registers_2 is
4   port(C, D, CLR : in std_logic;
5        Q : out std_logic);
6 end registers_2;
7 architecture archi of registers_2 is
8   begin
9     process (C, CLR)
10    begin
11      if (CLR = '1') then
12        Q <= '0';
13      elsif (C'event and C='0') then
14        Q <= D;
15      end if;
16    end process;
17 end archi;

```

Flip-Flop with Positive-Edge Clock and Synchronous Set



Verilog

```

1 module v_registers_3 (C, D, S, Q);
2   input C, D, S;
3   output Q;
4   reg Q;
5   always @ (posedge C)
6   begin
7     if (S)
8       Q <= 1'b1;
9     else
10      Q <= D;
11   end
12 endmodule

```

Note: Verilog and VHDL have **if** and **else** in their syntax (as in software languages); but with totally different interpretations: “if-elses are not executed; they are means of hardware description.”

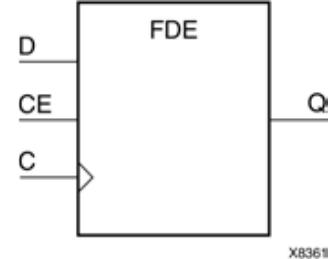
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity registers_3 is
4   port(C, D, S : in std_logic;
5        Q : out std_logic);
6 end registers_3;
7 architecture archi of registers_3 is
8 begin
9   process (C)
10  begin
11    if (C'event and C='1') then
12      if (S='1') then
13        Q <= '1';
14      else
15        Q <= D;
16      end if;
17    end if;
18  end process;
19 end archi;

```

Flip-Flop with Positive-Edge Clock and Clock Enable



Verilog

```

1 module v_registers_4 (C, D, CE, Q);
2   input C, D, CE;
3   output Q;
4   reg Q;
5   always @(posedge C)
6   begin
7     if (CE)
8       Q <= D;
9   end
10 endmodule

```

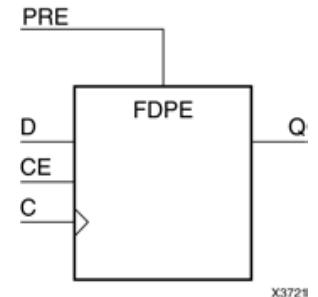
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity registers_4 is
4   port(C, D, CE : in std_logic;
5        Q : out std_logic);
6 end registers_4;
7 architecture archi of registers_4 is
8 begin
9   process (C)
10  begin
11    if (C'event and C='1') then
12      if (CE='1') then
13        Q <= D;
14      end if;
15    end if;
16  end process;
17 end archi;

```

4-Bit Register with Positive-Edge Clock, Asynchronous Set, and Clock Enable



Verilog

```

1 module v_registers_5 (C, D, CE, PRE, Q);
2   input C, CE, PRE;
3   input [3:0] D;
4   output [3:0] Q;
5   reg [3:0] Q;
6   always @ (posedge C or posedge PRE)
7     begin
8       if (PRE)
9         Q <= 4'b1111;
10      else
11        if (CE)
12          Q <= D;
13    end
14 endmodule

```

VHDL

```

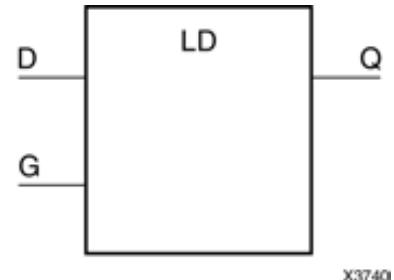
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity registers_5 is
4   port(C, CE, PRE : in std_logic;
5         D : in std_logic_vector (3 downto 0);
6         Q : out std_logic_vector (3 downto 0));
7 end registers_5;
8 architecture archi of registers_5 is
9   begin
10    process (C, PRE)
11      begin
12        if (PRE='1') then
13          Q <= "1111";
14        elsif (C'event and C='1') then
15          if (CE='1') then
16            Q <= D;
17          end if;
18        end if;
19      end process;
20  end archi;

```

Note: Notice the way that Verilog and VHDL define and assign “vectors of registers” in an abbreviated way. That’s why circuit schematics don’t resemble the HDL code.

Question: How many flip-flops are inferred by this piece of code?

Latch with Positive Gate



Verilog

```

1 module v_latches_1 (G, D, Q);
2   input G, D;
3   output Q;
4   reg Q;
5   always @(G or D)
6   begin
7     if (G)
8       Q = D;
9   end
10 endmodule

```

VHDL

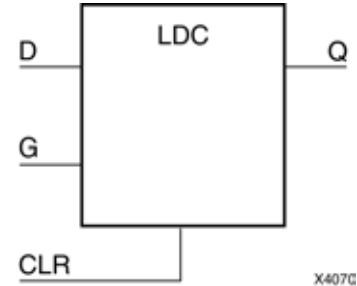
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity latches_1 is
4   port(G, D : in std_logic;
5        Q : out std_logic);
6 end latches_1;
7 architecture archi of latches_1 is
8 begin
9   process (G, D)
10  begin
11    if (G='1') then
12      Q <= D;
13    end if;
14  end process;
15 end archi;

```

Note: No flip-flops inferred.

Latch with Positive Gate and Asynchronous Reset



Verilog

```

1 module v_latches_2 (G, D, CLR, Q);
2   input G, D, CLR;
3   output Q;
4   reg Q;
5   always @(G or D or CLR)
6   begin
7     if (CLR)
8       Q = 1'b0;
9     else if (G)
10      Q = D;
11   end
12 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity latches_2 is
4   port(G, D, CLR : in std_logic;
5        Q : out std_logic);
6 end latches_2;
7 architecture archi of latches_2 is
8 begin
9   process (CLR, D, G)
10  begin
11    if (CLR='1') then
12      Q <= '0';
13    elsif (G='1') then
14      Q <= D;
15    end if;
16  end process;
17 end archi;

```

Important Note: The coding style defines the inferred hardware, not the variable names!

4-Bit Latch with Inverted Gate and Asynchronous Set

Verilog

```

1 module v_latches_3 (G, D, PRE, Q);
2   input G, PRE;
3   input [3:0] D;
4   output [3:0] Q;
5   reg [3:0] Q;
6   always @(G or D or PRE)
7   begin
8     if (PRE)
9       Q = 4'b1111;
10    else if (~G)
11      Q = D;
12  end
13 endmodule

```

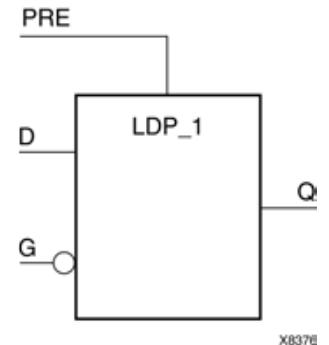
always @(*)
alternative form

VHDL

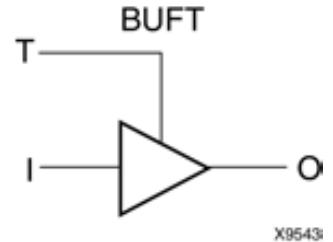
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity latches_3 is
4   port(D : in std_logic_vector(3 downto 0);
5        G, PRE : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end latches_3;
8 architecture archi of latches_3 is
9 begin
10   process (PRE, G, D)
11   begin
12     if (PRE='1') then
13       Q <= "1111";
14     elsif (G='0') then
15       Q <= D;
16     end if;
17   end process;
18 end archi;

```



Tristate Description Using Combinatorial Process and Always



Verilog

```

1 // Always Form
2 module v_three_st_1 (T, I, O);
3   input T, I;
4   output O;
5   reg O;
6   always @(T or I)
7   begin
8     if (~T)
9       O = I;
10    else
11      O = 1'bZ;
12  end
13 endmodule
14
15 // Assign Form
16 module v_three_st_2 (T, I, O);
17   input T, I;
18   output O;
19   assign O = (~T) ? I: 1'bZ;
20 endmodule

```

alternative form: explicit definition of a tristate buffer

```
bufif0 U1(data_bus, in, data_enable_low);
```

VHDL

```

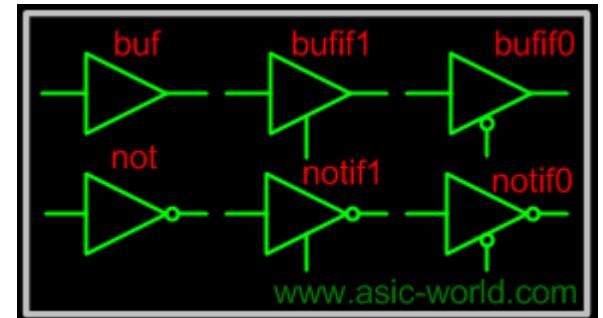
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity three_st_1 is
4   port(T : in std_logic;
5        I : in std_logic;
6        O : out std_logic);
7 end three_st_1;
8 -- Process Form
9 architecture archi of three_st_1 is
10 begin
11   process (I, T)
12   begin
13     if (T='0') then
14       O <= I;
15     else
16       O <= 'Z';
17     end if;
18   end process;
19 end archi;
20
21 -- Concurrent Form
22 architecture archi of three_st_1 is
23 begin
24   O <= I when (T='0') else 'Z';
25 end archi;

```

Common Buffers

- Buffers may also be used as built-in primitives.

Gate	Description
not	Output inverter
buf	Output buffer.
bufif0	Tri-state buffer, Active low enable.
bufif1	Tri-state buffer, Active high enable.
notif0	Tristate inverter, Low enable.
notif1	Tristate inverter, High enable.



Example:

```
bufif0 (weak1, pull0) #(4,5,3) (data_out, data_in, ctrl);
```

Unsigned Up-Counter with Asynchronous Reset

Verilog

```

1 module v_counters_1 (C, CLR, Q);
2   input C, CLR;
3   output [3:0] Q;
4   reg [3:0] tmp;
5   always @ (posedge C or posedge CLR)
6   begin
7     if (CLR)
8       tmp <= 4'b0000;
9     else
10       tmp <= tmp + 1'b1;
11   end
12   assign Q = tmp;
13 endmodule

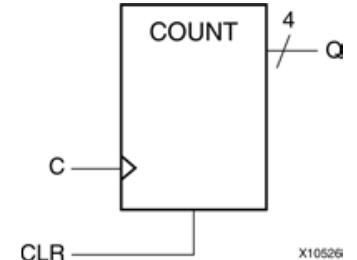
```

VHDL

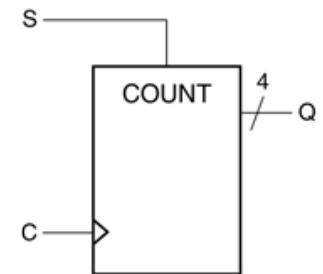
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_1 is
5   port(C, CLR : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end counters_1;
8 architecture archi of counters_1 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C, CLR)
12   begin
13     if (CLR='1') then
14       tmp <= "0000";
15     elsif (C'event and C='1') then
16       tmp <= tmp + 1;
17     end if;
18   end process;
19   Q <= tmp;
20 end archi;

```



Unsigned Down-Counter with Synchronous Set



X10527

Verilog

```

1 module v_counters_2 (C, S, Q);
2   input C, S;
3   output [3:0] Q;
4   reg [3:0] tmp;
5   always @ (posedge C)
6   begin
7     if (S)
8       tmp <= 4'b1111;
9     else
10       tmp <= tmp - 1'b1;
11   end
12   assign Q = tmp;
13 endmodule

```

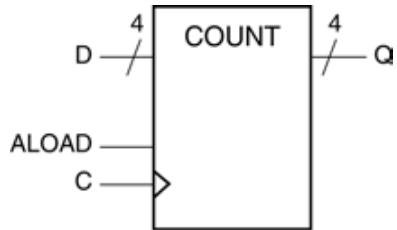
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_2 is
5   port(C, S : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end counters_2;
8 architecture archi of counters_2 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C)
12   begin
13     if (C'event and C='1') then
14       if (S='1') then
15         tmp <= "1111";
16       else
17         tmp <= tmp - 1;
18       end if;
19     end if;
20   end process;
21   Q <= tmp;
22 end archi;

```

Unsigned Up-Counter with Asynchronous Load from Primary Input



Verilog

```

1 module v_counters_3 (C, ALOAD, D, Q);
2   input C, ALOAD;
3   input [3:0] D;
4   output [3:0] Q;
5   reg [3:0] tmp;
6   always @ (posedge C or posedge ALOAD)
7   begin
8     if (ALOAD)
9       tmp <= D;
10    else
11      tmp <= tmp + 1'b1;
12  end
13  assign Q = tmp;
14 endmodule

```

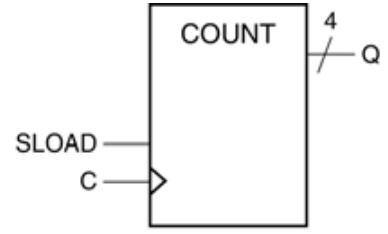
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_3 is
5   port(C, ALOAD : in std_logic;
6         D : in std_logic_vector(3 downto 0);
7         Q : out std_logic_vector(3 downto 0));
8 end counters_3;
9 architecture archi of counters_3 is
10 signal tmp: std_logic_vector(3 downto 0);
11 begin
12   process (C, ALOAD, D)
13   begin
14     if (ALOAD='1') then
15       tmp <= D;
16     elsif (C'event and C='1') then
17       tmp <= tmp + 1;
18     end if;
19   end process;
20   Q <= tmp;
21 end archi;

```

Unsigned Up-Counter with Synchronous Load with Constant



Verilog

```

1 module v_counters_4 (C, SLOAD, Q);
2   input C, SLOAD;
3   output [3:0] Q;
4   reg [3:0] tmp;
5   always @(posedge C)
6   begin
7     if (SLOAD)
8       tmp <= 4'b1010;
9     else
10      tmp <= tmp + 1'b1;
11   end
12   assign Q = tmp;
13 endmodule

```

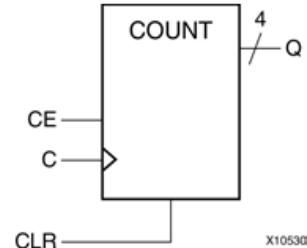
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_4 is
5   port(C, SLOAD : in std_logic;
6         Q : out std_logic_vector(3 downto 0));
7 end counters_4;
8 architecture archi of counters_4 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C)
12   begin
13     if (C'event and C='1') then
14       if (SLOAD='1') then
15         tmp <= "1010";
16       else
17         tmp <= tmp + 1;
18       end if;
19     end if;
20   end process;
21   Q <= tmp;
22 end archi;

```

Unsigned Up-Counter with Asynchronous Reset



Verilog

```

1 module v_counters_5 (C, CLR, CE, Q);
2   input C, CLR, CE;
3   output [3:0] Q;
4   reg [3:0] tmp;
5   always @ (posedge C or posedge CLR)
6   begin
7     if (CLR)
8       tmp <= 4'b0000;
9     else if (CE)
10      tmp <= tmp + 1'b1;
11   end
12   assign Q = tmp;
13 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_5 is
5   port(C, CLR, CE : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end counters_5;
8 architecture archi of counters_5 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C, CLR)
12   begin
13     if (CLR='1') then
14       tmp <= "0000";
15     elsif (C'event and C='1') then
16       if (CE='1') then
17         tmp <= tmp + 1;
18       end if;
19     end if;
20   end process;
21   Q <= tmp;
22 end archi;

```

Unsigned Up/Down-Counter with Asynchronous Reset

Verilog

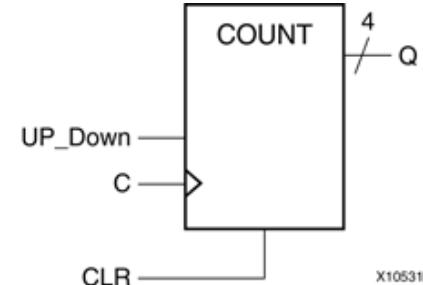
```

1 module v_counters_6 (C, CLR, UP_DOWN, Q);
2   input C, CLR, UP_DOWN;
3   output [3:0] Q;
4   reg [3:0] tmp;
5   always @(posedge C or posedge CLR)
6   begin
7     if (CLR)
8       tmp <= 4'b0000;
9     else if (UP_DOWN)
10      tmp <= tmp + 1'bl;
11    else
12      tmp <= tmp - 1'bl;
13  end
14  assign Q = tmp;
15 endmodule

```

Sample applications:

- FIFO valid data counter
- Chirp signal generator



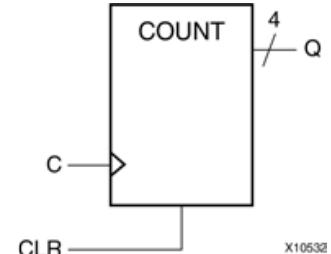
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity counters_6 is
5   port(C, CLR, UP_DOWN : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end counters_6;
8 architecture archi of counters_6 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C, CLR)
12   begin
13     if (CLR='1') then
14       tmp <= "0000";
15     elsif (C'event and C='1') then
16       if (UP_DOWN='1') then
17         tmp <= tmp + 1;
18       else
19         tmp <= tmp - 1;
20       end if;
21     end if;
22   end process;
23   Q <= tmp;
24 end archi;

```

Signed Up-Counter with Asynchronous Reset



Verilog

```

1 module v_counters_7 (C, CLR, Q);
2   input C, CLR;
3   output signed [3:0] Q;
4   reg signed [3:0] tmp;
5   always @ (posedge C or posedge CLR)
6   begin
7     if (CLR)
8       tmp <= 4'b0000;
9     else
10       tmp <= tmp + 1'b1;
11   end
12   assign Q = tmp;
13 endmodule

```

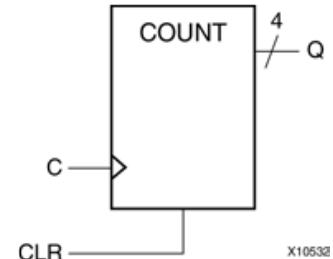
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_signed.all;
4 entity counters_7 is
5   port(C, CLR : in std_logic;
6        Q : out std_logic_vector(3 downto 0));
7 end counters_7;
8 architecture archi of counters_7 is
9 signal tmp: std_logic_vector(3 downto 0);
10 begin
11   process (C, CLR)
12   begin
13     if (CLR='1') then
14       tmp <= "0000";
15     elsif (C'event and C='1') then
16       tmp <= tmp + 1;
17     end if;
18   end process;
19   Q <= tmp;
20 end archi;

```

Signed Up-Counter with Asynchronous Reset and Modulo Maximum



Verilog

```

1 module v_counters_8 (C, CLR, Q);
2   parameter
3     MAX_SQRT = 4,
4     MAX = (MAX_SQRT*MAX_SQRT);
5   input C, CLR;
6   output [MAX_SQRT-1:0] Q;
7   reg [MAX_SQRT-1:0] cnt;
8   always @ (posedge C or posedge CLR)
9   begin
10     if (CLR)
11       cnt <= 0;
12     else
13       cnt <= (cnt + 1) %MAX;
14   end
15   assign Q = cnt;
16 endmodule

```

Note: Not very practical, since MAX should be a power of two

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 entity counters_8 is
5   generic (MAX : integer := 16);
6   port(C, CLR : in std_logic;
7         Q : out integer range 0 to MAX-1);
8 end counters_8;
9 architecture archi of counters_8 is
10 signal cnt : integer range 0 to MAX-1;
11 begin
12   process (C, CLR)
13   begin
14     if (CLR='1') then
15       cnt <= 0;
16     elsif (rising_edge(C)) then
17       cnt <= (cnt + 1) mod MAX ;
18     end if;
19   end process;
20   Q <= cnt;
21 end archi;

```

Unsigned Up Accumulator with Asynchronous Reset

Verilog

```

1 module v_accumulators_1 (C, CLR, D, Q);
2   input C, CLR;
3   input [3:0] D;
4   output [3:0] Q;
5   reg [3:0] tmp;
6   always @ (posedge C or posedge CLR)
7   begin
8     if (CLR)
9       tmp = 4'b0000;
10    else
11      tmp = tmp + D;
12  end
13  assign Q = tmp;
14 endmodule

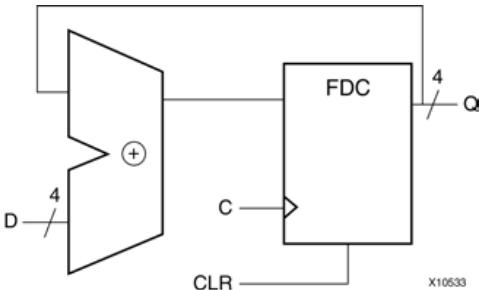
```

VHDL

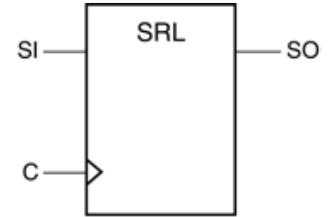
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity accumulators_1 is
5   port(C, CLR : in std_logic;
6         D : in std_logic_vector(3 downto 0);
7         Q : out std_logic_vector(3 downto 0));
8 end accumulators_1;
9 architecture archi of accumulators_1 is
10 signal tmp: std_logic_vector(3 downto 0);
11 begin
12   process (C, CLR)
13   begin
14     if (CLR='1') then
15       tmp <= "0000";
16     elsif (C'event and C='1') then
17       tmp <= tmp + D;
18     end if;
19   end process;
20   Q <= tmp;
21 end archi;

```



Shift-Left Register with Positive-Edge Clock, Serial In and Serial Out



X10534

Verilog

```

1 module v_shift_registers_1 (C, SI, SO);
2   input C,SI;
3   output SO;
4   reg [7:0] tmp;
5   always @(posedge C)
6   begin
7     tmp = {tmp[6:0], SI};
8   end
9   assign SO = tmp[7];
10 endmodule

```

Note: If the shift register has a synchronous parallel load, or multiple set or reset signals, no SRL16 is implemented.

Guideline: For better area efficiency using built-in SRL, avoid using sets/resets, whenever not needed.

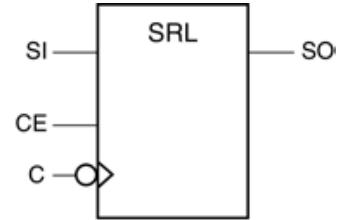
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_1 is
4   port(C, SI : in std_logic;
5         SO : out std_logic);
6 end shift_registers_1;
7 architecture archi of shift_registers_1 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10 process (C)
11 begin
12   if (C'event and C='1') then
13     for i in 0 to 6 loop
14       tmp(i+1) <= tmp(i);
15     end loop;
16     tmp(0) <= SI;
17   end if;
18 end process;
19 SO <= tmp(7);
20 end archi;

```

Shift-Left Register with Negative-Edge Clock, Clock Enable, Serial In and Serial Out



X10535

Verilog

```

1 module v_shift_registers_2 (C, CE, SI, SO);
2   input C,SI, CE;
3   output SO;
4   reg [7:0] tmp;
5   always @(negedge C)
6   begin
7     if (CE)
8       begin
9         tmp = {tmp[6:0], SI};
10      end
11    end
12    assign SO = tmp[7];
13 endmodule

```

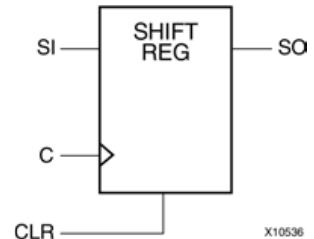
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_2 is
4   port(C, SI, CE : in std_logic;
5           SO : out std_logic);
6 end shift_registers_2;
7 architecture archi of shift_registers_2 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10  process (C)
11  begin
12    if (C'event and C='0') then
13      if (CE='1') then
14        for i in 0 to 6 loop
15          tmp(i+1) <= tmp(i);
16        end loop;
17        tmp(0) <= SI;
18      end if;
19    end if;
20  end process;
21  SO <= tmp(7);
22 end archi;

```

Shift-Left Register With Positive-Edge Clock, Asynchronous Reset, Serial In and Serial Out



Verilog

```

1 module v_shift_registers_3 (C, CLR, SI, SO);
2   input C,SI,CLR;
3   output SO;
4   reg [7:0] tmp;
5   always @ (posedge C or posedge CLR)
6   begin
7     if (CLR)
8       tmp <= 8'b00000000;
9     else
10      tmp <= {tmp[6:0], SI};
11   end
12   assign SO = tmp[7];
13 endmodule

```

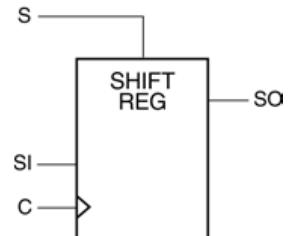
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_3 is
4   port(C, SI, CLR : in std_logic;
5         SO : out std_logic);
6 end shift_registers_3;
7 architecture archi of shift_registers_3 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10   process (C, CLR)
11   begin
12     if (CLR='1') then
13       tmp <= (others => '0');
14     elsif (C'event and C='1') then
15       tmp <= tmp(6 downto 0) & SI;
16     end if;
17   end process;
18   SO <= tmp(7);
19 end archi;

```

Shift-Left Register with Positive-Edge Clock, Synchronous Set, Serial In and Serial Out



Verilog

```

1 module v_shift_registers_4 (C, S, SI, SO);
2   input C,SI,S;
3   output SO;
4   reg [7:0] tmp;
5   always @(posedge C)
6   begin
7     if (S)
8       tmp <= 8'b11111111;
9     else
10      tmp <= {tmp[6:0], SI};
11   end
12   assign SO = tmp[7];
13 endmodule

```

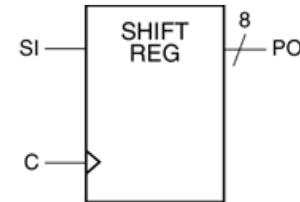
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_4 is
4   port(C, SI, S : in std_logic;
5         SO : out std_logic);
6 end shift_registers_4;
7 architecture archi of shift_registers_4 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10   process (C, S)
11   begin
12     if (C'event and C='1') then
13       if (S='1') then
14         tmp <= (others => '1');
15       else
16         tmp <= tmp(6 downto 0) & SI;
17       end if;
18     end if;
19   end process;
20   SO <= tmp(7);
21 end archi;

```

Shift-Left Register with Positive-Edge Clock, Serial In and Parallel Out



X10538

Verilog

```

1 module v_shift_registers_5 (C, SI, PO);
2   input C, SI;
3   output [7:0] PO;
4   reg [7:0] tmp;
5   always @ (posedge C)
6     tmp <= {tmp[6:0], SI};
7   assign PO = tmp;
8 endmodule

```

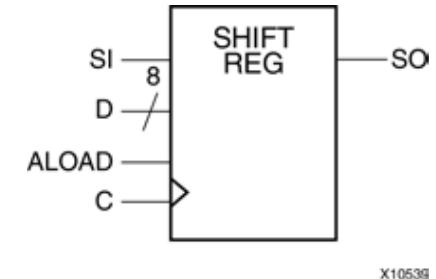
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_5 is
4   port(C, SI : in std_logic;
5        PO : out std_logic_vector(7 downto 0));
6 end shift_registers_5;
7 architecture archi of shift_registers_5 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10   process (C)
11     begin
12       if (C'event and C='1') then
13         tmp <= tmp(6 downto 0)& SI;
14       end if;
15     end process;
16   PO <= tmp;
17 end archi;

```

Shift-Left Register With Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out



Verilog

```

1 module v_shift_registers_6 (C, ALOAD, SI, D, SO);
2   input C,SI,ALOAD;
3   input [7:0] D;
4   output SO;
5   reg [7:0] tmp;
6   always @ (posedge C or posedge ALOAD)
7   begin
8     if (ALOAD)
9       tmp <= D;
10    else
11      tmp <= {tmp[6:0], SI};
12  end
13  assign SO = tmp[7];
14 endmodule

```

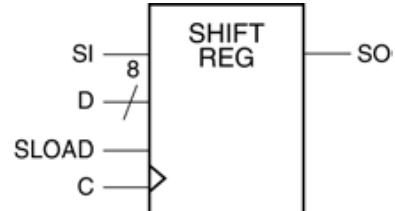
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_6 is
4   port(C, SI, ALOAD : in std_logic;
5         D : in std_logic_vector(7 downto 0);
6         SO : out std_logic);
7 end shift_registers_6;
8 architecture archi of shift_registers_6 is
9 signal tmp: std_logic_vector(7 downto 0);
10 begin
11   process (C, ALOAD, D)
12   begin
13     if (ALOAD='1') then
14       tmp <= D;
15     elsif (C'event and C='1') then
16       tmp <= tmp(6 downto 0) & SI;
17     end if;
18   end process;
19   SO <= tmp(7);
20 end archi;

```

Shift-Left Register With Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out



X10540

Verilog

```

1 module v_shift_registers_7 (C, SLOAD, SI, D, SO);
2   input C,SI,SLOAD;
3   input [7:0] D;
4   output SO;
5   reg [7:0] tmp;
6   always @(posedge C)
7   begin
8     if (SLOAD)
9       tmp <= D;
10    else
11      tmp <= {tmp[6:0], SI};
12  end
13  assign SO = tmp[7];
14 endmodule

```

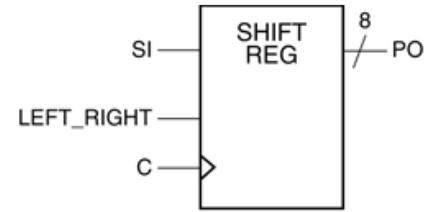
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_7 is
4   port(C, SI, SLOAD : in std_logic;
5         D : in std_logic_vector(7 downto 0);
6         SO : out std_logic);
7 end shift_registers_7;
8 architecture archi of shift_registers_7 is
9 signal tmp: std_logic_vector(7 downto 0);
10 begin
11   process (C)
12   begin
13     if (C'event and C='1') then
14       if (SLOAD='1') then
15         tmp <= D;
16       else
17         tmp <= tmp(6 downto 0) & SI;
18       end if;
19     end if;
20   end process;
21   SO <= tmp(7);
22 end archi;

```

Shift-Left/Shift-Right Register With Positive-Edge Clock, Serial In and Parallel Out



Verilog

```

1 module v_shift_registers_8 (C, SI, LEFT_RIGHT, PO);
2   input C, SI, LEFT_RIGHT;
3   output [7:0] PO;
4   reg [7:0] tmp;
5   always @(posedge C)
6   begin
7     if (LEFT_RIGHT==1'b0)
8       tmp <= {tmp[6:0], SI};
9     else
10      tmp <= {SI, tmp[7:1]};
11   end
12   assign PO = tmp;
13 endmodule

```

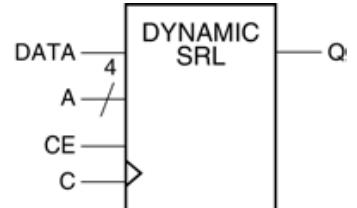
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity shift_registers_8 is
4   port(C, SI, LEFT_RIGHT : in std_logic;
5        PO : out std_logic_vector(7 downto 0));
6 end shift_registers_8;
7 architecture archi of shift_registers_8 is
8 signal tmp: std_logic_vector(7 downto 0);
9 begin
10  process (C)
11  begin
12    if (C'event and C='1') then
13      if (LEFT_RIGHT='0') then
14        tmp <= tmp(6 downto 0) & SI;
15      else
16        tmp <= SI & tmp(7 downto 1);
17      end if;
18    end if;
19  end process;
20  PO <= tmp;
21 end archi;

```

Dynamic Shift Register With Positive-Edge Clock, Serial In and Serial Out



Verilog

```

1 module v_dynamic_shift_registers_1 (Q,CE,CLK,D,A);
2   input CLK, D, CE;
3   input [3:0] A;
4   output Q;
5   reg [15:0] data;
6   assign Q = data[A];
7   always @ (posedge CLK)
8   begin
9     if (CE == 1'b1)
10       data <= {data[14:0], D};
11   end
12 endmodule

```

VHDL

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4 entity dynamic_shift_registers_1 is
5   port(CLK : in std_logic;
6        DATA : in std_logic;
7        CE : in std_logic;
8        A : in std_logic_vector(3 downto 0);
9        Q : out std_logic);
10 end dynamic_shift_registers_1;
11 architecture rtl of dynamic_shift_registers_1 is
12   constant DEPTH_WIDTH : integer := 16;
13   type SRL_ARRAY is array (0 to DEPTH_WIDTH-1) of std_logic;
14   -- The type SRL_ARRAY can be array
15   -- (0 to DEPTH_WIDTH-1) of
16   -- std_logic_vector(BUS_WIDTH downto 0)
17   -- or array (DEPTH_WIDTH-1 downto 0) of
18   -- std_logic_vector(BUS_WIDTH downto 0)
19   -- (the subtype is forward (see below))
20   signal SRL_SIG : SRL_ARRAY;
21 begin
22   PROC_SRL16 : process (CLK)
23   begin
24     if (CLK'event and CLK = '1') then
25       if (CE = '1') then
26         SRL_SIG <= DATA & SRL_SIG(0 to DEPTH_WIDTH-2);
27       end if;
28     end if;
29   end process;
30   Q <= SRL_SIG(conv_integer(A));
31 end rtl;

```

Further reading on Shift-Register applications:

https://www.xilinx.com/support/documentation/white_papers/wp271.pdf

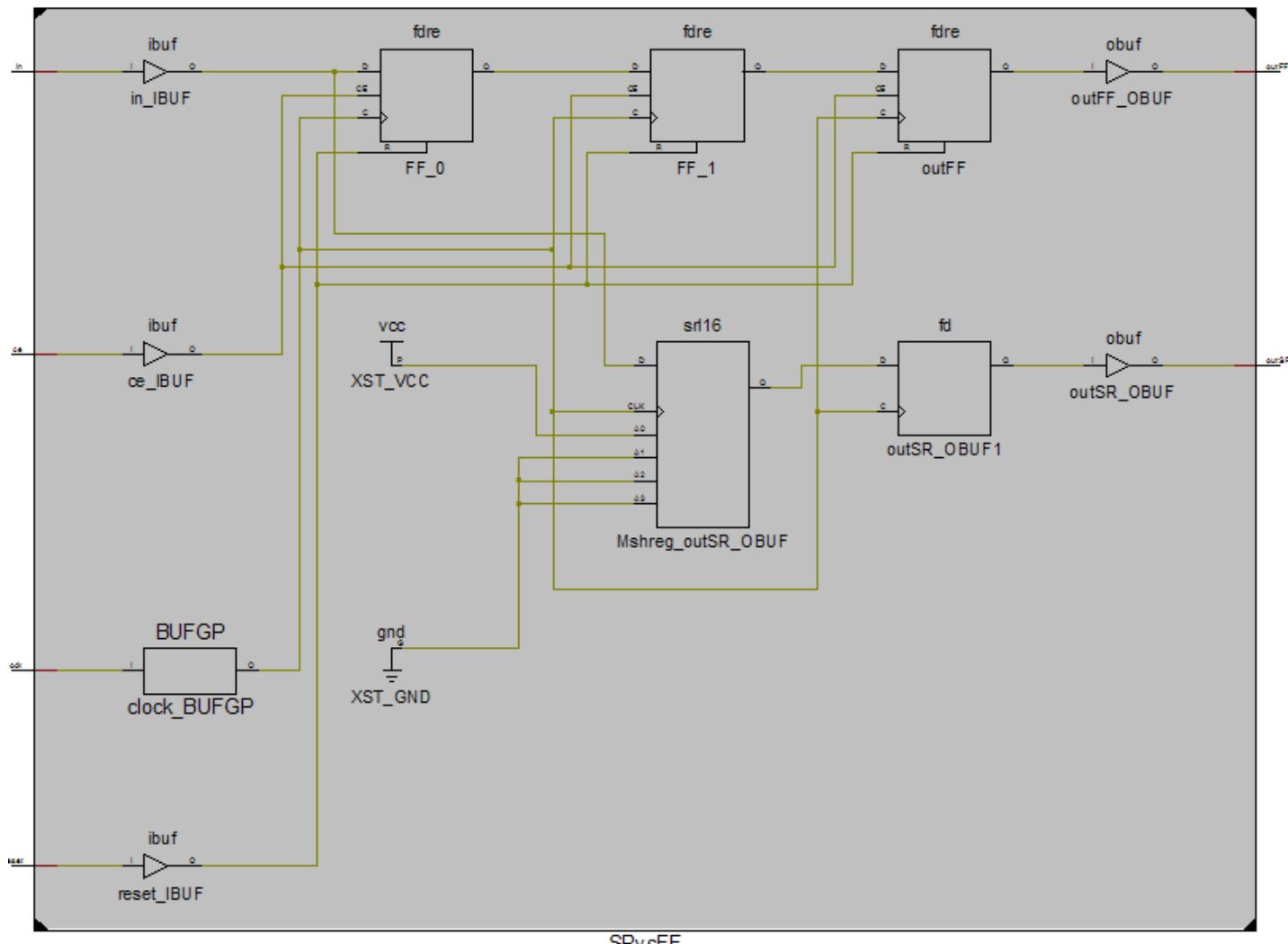
Shift Registers vs. Flip-Flops

```

1 module SRvsFF(
2     input clock,
3     input reset,
4     input ce,
5     input in,
6     output reg outSR,
7     output reg outFF
8 );
9
10    reg FF[0:1];
11
12    always @ (posedge clock) begin
13        if(reset)begin
14            outFF <= 0;
15            FF[0] <= 0;
16            FF[1] <= 0;
17        end
18        else if(ce)begin
19            outFF <= FF[1];
20            FF[1] <= FF[0];
21            FF[0] <= in;
22        end
23    end
24
25    reg SR[0:1];
26
27    always @ (posedge clock) begin
28        outSR <= SR[1];
29        SR[1] <= SR[0];
30        SR[0] <= in;
31    end
32
33 endmodule

```

Technology Schematic on Xilinx Spartan 3



Shift Register Applications

Shift Registers have various applications including:

- Pipeline Compensation
- Pseudo Random Number (Noise) Generation
- Serial Frame Synchronization (in telecommunications)
- Running Average using an Adder Tree
- Running Average Using an Accumulator
- Pulse Generation and Clock Division
- Multi-stage Dividers
- Forcing the Hot State
- Pattern Generation
- FIR Filter
- FIFO
- A Complete RS-232 Receiver

Multiplexers in Verilog

- If-Then-Else or Case can be used for multiplexers (MUXs) description.
- If one describes a MUX using a Case statement, and does not specify all values of the selector, the result may be latches instead of a multiplexer. When writing MUXs, one can use don't care to describe selector values.
- XST decides whether to infer the MUXs during the Macro Inference step. If the MUX has several inputs that are the same, XST can decide not to infer it. One can use the MUX_EXTRACT constraint to force XST to infer the MUX.
- Verilog Case statements can be: full or not full; parallel or not parallel
- A Verilog Case statement is:
 - Full: if all possible branches are specified
 - Parallel: if it does not contain branches that can be executed simultaneously

Multiplexers in Verilog

Multiplexers Full and Parallel

```

1 module full (sel, i1, i2, i3, i4, o1);
2   input [1:0] sel;
3   input [1:0] i1, i2, i3, i4;
4   output [1:0] o1;
5   reg [1:0] o1;
6   always @ (sel or i1 or i2 or i3 or i4)
7   begin
8     case (sel)
9       2'b00: o1 = i1;
10      2'b01: o1 = i2;
11      2'b10: o1 = i3;
12      2'b11: o1 = i4;
13    endcase
14  end
15 endmodule

```

Multiplexers Not Full But Parallel

```

1 module notfull (sel, i1, i2, i3, o1);
2   input [1:0] sel;
3   input [1:0] i1, i2, i3;
4   output [1:0] o1;
5   reg [1:0] o1;
6   always @ (sel or i1 or i2 or i3)
7   begin
8     case (sel)
9       2'b00: o1 = i1;
10      2'b01: o1 = i2;
11      2'b10: o1 = i3;
12    endcase
13  end
14 endmodule

```

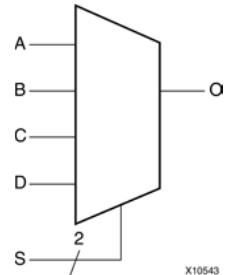
Multiplexers in Verilog

Multiplexers Neither Full Nor Parallel

```
1 module notfull_notparallel (sel1, sel2, i1, i2, o1);
2   input [1:0] sel1, sel2;
3   input [1:0] i1, i2;
4   output [1:0] o1;
5   reg [1:0] o1;
6   always @(sel1 or sel2)
7   begin
8     case (2'b00)
9       sel1: o1 = i1;
10      sel2: o1 = i2;
11    endcase
12  end
13 endmodule
```

Note: XST automatically determines the characteristics of the Case statements and generates logic using multiplexers, priority encoders, or latches that best implement the exact behavior of the Case statement.

MUX using IF Statements



Verilog

```

1 module v_multiplexers_1 (a, b, c, d, s, o);
2   input a,b,c,d;
3   input [1:0] s;
4   output o;
5   reg o;
6   always @ (a or b or c or d or s)
7   begin
8     if (s == 2'b00) o = a;
9     else if (s == 2'b01) o = b;
10    else if (s == 2'b10) o = c;
11    else o = d;
12  end
13 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity multiplexers_1 is
4   port (a, b, c, d : in std_logic;
5         s : in std_logic_vector (1 downto 0);
6         o : out std_logic);
7 end multiplexers_1;
8 architecture archi of multiplexers_1 is
9 begin
10  process (a, b, c, d, s)
11  begin
12    if (s = "00") then o <= a;
13    elsif (s = "01") then o <= b;
14    elsif (s = "10") then o <= c;
15    else o <= d;
16    end if;
17  end process;
18 end archi;

```

MUX using Case Statements

Verilog

```

1 module v_multiplexers_2 (a, b, c, d, s, o);
2   input a,b,c,d;
3   input [1:0] s;
4   output o;
5   reg o;
6   always @ (a or b or c or d or s)
7   begin
8     case (s)
9       2'b00 : o = a;
10      2'b01 : o = b;
11      2'b10 : o = c;
12      default : o = d;
13    endcase
14  end
15 endmodule

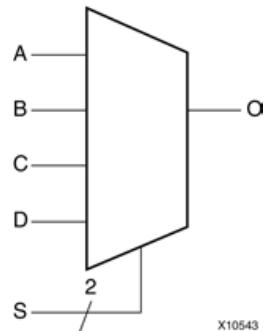
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity multiplexers_2 is
4   port (a, b, c, d : in std_logic;
5         s : in std_logic_vector (1 downto 0);
6         o : out std_logic);
7 end multiplexers_2;
8 architecture archi of multiplexers_2 is
9 begin
10  process (a, b, c, d, s)
11  begin
12    case s is
13      when "00" => o <= a;
14      when "01" => o <= b;
15      when "10" => o <= c;
16      when others => o <= d;
17    end case;
18  end process;
19 end archi;

```



X10543

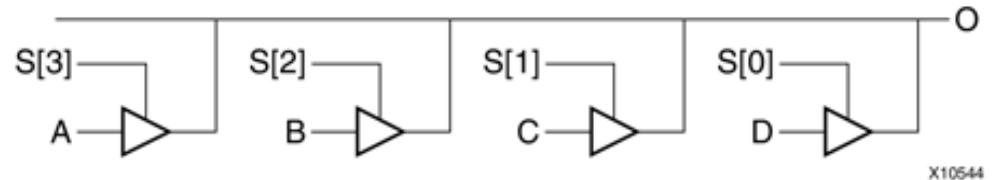
MUX using Tristate Buffers

Verilog

```

1 module v_multiplexers_3 (a, b, c, d, s, o);
2   input a,b,c,d;
3   input [3:0] s;
4   output o;
5   assign o = s[3] ? a :1'bz;
6   assign o = s[2] ? b :1'bz;
7   assign o = s[1] ? c :1'bz;
8   assign o = s[0] ? d :1'bz;
9 endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity multiplexers_3 is
4   port (a, b, c, d : in std_logic;
5         s : in std_logic_vector (3 downto 0);
6         o : out std_logic);
7 end multiplexers_3;
8 architecture archi of multiplexers_3 is
9 begin
10   o <= a when (s(0)='0') else 'Z';
11   o <= b when (s(1)='0') else 'Z';
12   o <= c when (s(2)='0') else 'Z';
13   o <= d when (s(3)='0') else 'Z';
14 end archi;

```

Missing Else Statement Leading to a Latch Inference

Verilog

```

1 module v_multiplexers_4 (a, b, c, s, o);
2   input a,b,c;
3   input [1:0] s;
4   output o;
5   reg o;
6   always @(a or b or c or s)
7   begin
8     if (s == 2'b00) o = a;
9     else if (s == 2'b01) o = b;
10    else if (s == 2'b10) o = c;
11  end
12 endmodule

```

VHDL

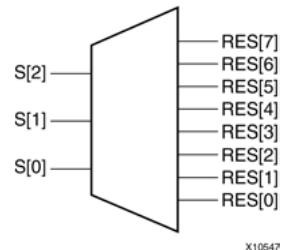
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity multiplexers_4 is
4   port (a, b, c: in std_logic;
5         s : in std_logic_vector (1 downto 0);
6         o : out std_logic);
7 end multiplexers_4;
8 architecture archi of multiplexers_4 is
9 begin
10  process (a, b, c, s)
11  begin
12    if (s = "00") then o <= a;
13    elsif (s = "01") then o <= b;
14    elsif (s = "10") then o <= c;
15    end if;
16  end process;
17 end archi;

```

Caution! Unless you actually intended to describe such a latch, add the missing **else** statement. Leaving out an **else** statement may also result in errors during simulation.

One-Hot Decoders



Verilog

```

1 module v_decoders_1 (sel, res);
2   input [2:0] sel;
3   output [7:0] res;
4   reg [7:0] res;
5   always @(sel or res)
6   begin
7     case (sel)
8       3'b000 : res = 8'b00000001;
9       3'b001 : res = 8'b00000010;
10      3'b010 : res = 8'b00000100;
11      3'b011 : res = 8'b00001000;
12      3'b100 : res = 8'b00010000;
13      3'b101 : res = 8'b00100000;
14      3'b110 : res = 8'b01000000;
15      default : res = 8'b10000000;
16    endcase
17  end
18 endmodule

```

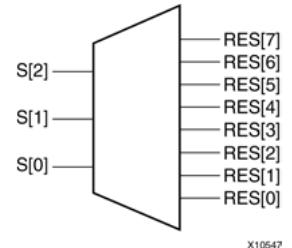
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity decoders_1 is
4   port (sel: in std_logic_vector (2 downto 0);
5         res: out std_logic_vector (7 downto 0));
6 end decoders_1;
7 architecture archi of decoders_1 is
8 begin
9   res <= "00000001" when sel = "000" else
10    "00000010" when sel = "001" else
11    "00000100" when sel = "010" else
12    "00001000" when sel = "011" else
13    "00010000" when sel = "100" else
14    "00100000" when sel = "101" else
15    "01000000" when sel = "110" else
16    "10000000";
17 end archi;

```

One-Cold Decoders



Verilog

```

1 module v_decoders_2 (sel, res);
2   input [2:0] sel;
3   output [7:0] res;
4   reg [7:0] res;
5   always @(sel)
6   begin
7     case (sel)
8       3'b000 : res = 8'b11111110;
9       3'b001 : res = 8'b11111101;
10      3'b010 : res = 8'b11111011;
11      3'b011 : res = 8'b11110111;
12      3'b100 : res = 8'b11101111;
13      3'b101 : res = 8'b11011111;
14      3'b110 : res = 8'b10111111;
15      default : res = 8'b01111111;
16    endcase
17  end
18 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity decoders_2 is
4   port (sel: in std_logic_vector (2 downto 0);
5         res: out std_logic_vector (7 downto 0));
6 end decoders_2;
7 architecture archi of decoders_2 is
8 begin
9   res <= "11111110" when sel = "000" else
10    "11111101" when sel = "001" else
11    "11111011" when sel = "010" else
12    "11110111" when sel = "011" else
13    "11101111" when sel = "100" else
14    "11011111" when sel = "101" else
15    "10111111" when sel = "110" else
16    "01111111";
17 end archi;

```

No Decoder Inference (Unused Decoder Output)

Verilog

```

1 module v_decoders_3 (sel, res);
2   input [2:0] sel;
3   output [7:0] res;
4   reg [7:0] res;
5   always @(sel)
6   begin
7     case (sel)
8       3'b000 : res = 8'b00000001;
9       // unused decoder output
10      3'b001 : res = 8'bxxxxxxxxx;
11      3'b010 : res = 8'b000000100;
12      3'b011 : res = 8'b000001000;
13      3'b100 : res = 8'b000100000;
14      3'b101 : res = 8'b001000000;
15      3'b110 : res = 8'b010000000;
16      default : res = 8'b100000000;
17     endcase
18   end
19 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity decoders_3 is
4   port (sel: in std_logic_vector (2 downto 0);
5         res: out std_logic_vector (7 downto 0));
6 end decoders_3;
7 architecture archi of decoders_3 is
8 begin
9   res <= "00000001" when sel = "000" else
10    -- unused decoder output
11    "XXXXXXXX" when sel = "001" else
12    "00000100" when sel = "010" else
13    "00001000" when sel = "011" else
14    "00010000" when sel = "100" else
15    "00100000" when sel = "101" else
16    "01000000" when sel = "110" else
17    "10000000";
18 end archi;

```

No Decoder Inference (Some Selector Values Unused)

Verilog

```

1 module v_decoders_4 (sel, res);
2   input [2:0] sel;
3   output [7:0] res;
4   reg [7:0] res;
5   always @(sel or res)
6   begin
7     case (sel)
8       3'b000 : res = 8'b00000001;
9       3'b001 : res = 8'b00000010;
10      3'b010 : res = 8'b00000100;
11      3'b011 : res = 8'b00001000;
12      3'b100 : res = 8'b00010000;
13      3'b101 : res = 8'b00100000;
14      // 110 and 111 selector values are unused
15      default : res = 8'bxxxxxxxxx;
16    endcase
17  end
18 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity decoders_4 is
4   port (sel: in std_logic_vector (2 downto 0);
5         res: out std_logic_vector (7 downto 0));
6 end decoders_4;
7 architecture archi of decoders_4 is
8 begin
9   res <= "00000001" when sel = "000" else
10    "00000010" when sel = "001" else
11    "00000100" when sel = "010" else
12    "00001000" when sel = "011" else
13    "00010000" when sel = "100" else
14    "00100000" when sel = "101" else
15    -- 110 and 111 selector values are unused
16    "XXXXXXXX";
17 end archi;

```

Priority Encoders

Verilog

```

1 (* priority_extract="force" *)
2 module v_priority_encoder_1 (sel, code);
3     input [7:0] sel;
4     output [2:0] code;
5     reg [2:0] code;
6     always @(sel)
7     begin
8         if (sel[0]) code = 3'b000;
9         else if (sel[1]) code = 3'b001;
10        else if (sel[2]) code = 3'b010;
11        else if (sel[3]) code = 3'b011;
12        else if (sel[4]) code = 3'b100;
13        else if (sel[5]) code = 3'b101;
14        else if (sel[6]) code = 3'b110;
15        else if (sel[7]) code = 3'b111;
16        else code = 3'bxxx;
17    end
18 endmodule

```

VHDL

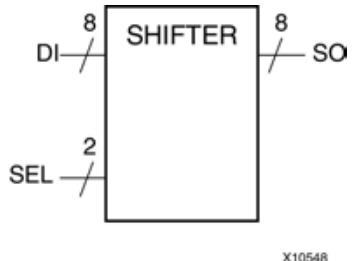
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity priority_encoder_1 is
4     port ( sel : in std_logic_vector (7 downto 0);
5            code : out std_logic_vector (2 downto 0));
6     attribute priority_extract: string;
7     attribute priority_extract of priority_encoder_1: entity is "force";
8 end priority_encoder_1;
9 architecture archi of priority_encoder_1 is
10 begin
11     code <= "000" when sel(0) = '1' else
12     "001" when sel(1) = '1' else
13     "010" when sel(2) = '1' else
14     "011" when sel(3) = '1' else
15     "100" when sel(4) = '1' else
16     "101" when sel(5) = '1' else
17     "110" when sel(6) = '1' else
18     "111" when sel(7) = '1' else
19     "---";
20 end archi;

```

Sample application: Prioritized interrupt mechanism design

Logical Shifter One



Verilog

```

1 module v_logical_shifters_1 (DI, SEL, SO);
2   input [7:0] DI;
3   input [1:0] SEL;
4   output [7:0] SO;
5   reg [7:0] SO;
6   always @(DI or SEL)
7   begin
8     case (SEL)
9       2'b00 : SO = DI;
10      2'b01 : SO = DI << 1;
11      2'b10 : SO = DI << 2;
12      default : SO = DI << 3;
13    endcase
14  end
15 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity logical_shifters_1 is
5   port(DI : in unsigned(7 downto 0);
6        SEL : in unsigned(1 downto 0);
7        SO : out unsigned(7 downto 0));
8 end logical_shifters_1;
9 architecture archi of logical_shifters_1 is
10 begin
11   with SEL select
12     SO <= DI when "00",
13     DI sll 1 when "01",
14     DI sll 2 when "10",
15     DI sll 3 when others;
16 end archi;

```

Logical Shifter Two (no logic shifters inferred)

Verilog

```

1  /*
2  XST does not infer a logical shifter for
3  this example, as not all of the selector
4  values are presented.
5  */
6  module v_logical_shifters_2 (DI, SEL, SO);
7      input [7:0] DI;
8      input [1:0] SEL;
9      output [7:0] SO;
10     reg [7:0] SO;
11     always @(DI or SEL)
12     begin
13         case (SEL)
14             2'b00 : SO = DI;
15             2'b01 : SO = DI << 1;
16             default : SO = DI << 2;
17         endcase
18     end
19 endmodule

```

VHDL

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  entity logical_shifters_2 is
5      port(DI : in unsigned(7 downto 0);
6            SEL : in unsigned(1 downto 0);
7            SO : out unsigned(7 downto 0));
8  end logical_shifters_2;
9  architecture archi of logical_shifters_2 is
10 begin
11     with SEL select
12         SO <= DI when "00",
13         DI sll 1 when "01",
14         DI sll 2 when others;
15 end archi;

```

Logical Shifter Three (no logic shifters inferred)

Verilog

```

1  /*
2  XST does not infer a logical shifter for
3  this example, as the value is not
4  incremented by 1 for each consequent
5  binary value of the selector.
6  */
7 module v_logical_shifters_3 (DI, SEL, SO);
8   input [7:0] DI;
9   input [1:0] SEL;
10  output [7:0] SO;
11  reg[7:0] SO;
12  always @ (DI or SEL)
13 begin
14   case (SEL)
15     2'b00 : SO = DI;
16     2'b01 : SO = DI << 1;
17     2'b10 : SO = DI << 3;
18     default : SO = DI << 2;
19   endcase
20 end
21 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity logical_shifters_3 is
5   port(DI : in unsigned(7 downto 0);
6        SEL : in unsigned(1 downto 0);
7        SO : out unsigned(7 downto 0));
8 end logical_shifters_3;
9 architecture archi of logical_shifters_3 is
10 begin
11   with SEL select
12     SO <= DI when "00",
13     DI sll 1 when "01",
14     DI sll 3 when "10",
15     DI sll 2 when others;
16 end archi;

```

Unsigned Adder

Verilog

```

1 module v_adders_1(A, B, SUM);
2   input [7:0] A;
3   input [7:0] B;
4   output [7:0] SUM;
5   assign SUM = A + B;
6 endmodule

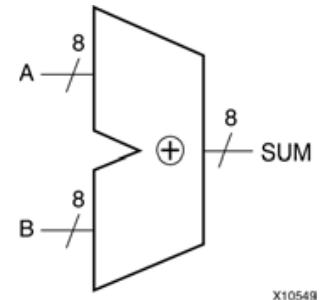
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity adders_1 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        SUM : out std_logic_vector(7 downto 0));
7 end adders_1;
8 architecture archi of adders_1 is
9 begin
10   SUM <= A + B;
11 end archi;

```



Unsigned Adder with Carry

Verilog

```

1 module v_adders_2(A, B, CI, SUM);
2   input [7:0] A;
3   input [7:0] B;
4   input CI;
5   output [7:0] SUM;
6   assign SUM = A + B + CI;
7 endmodule

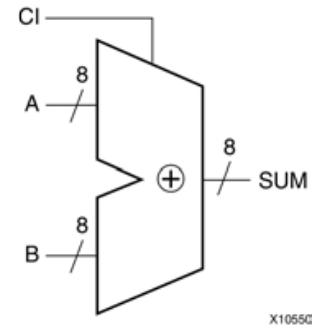
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity adders_2 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        CI : in std_logic;
7        SUM : out std_logic_vector(7 downto 0));
8 end adders_2;
9 architecture archi of adders_2 is
10 begin
11   SUM <= A + B + CI;
12 end archi;

```



Unsigned Adder with Carry Out

Verilog

```

1 module v_adders_3(A, B, SUM, CO);
2   input [7:0] A;
3   input [7:0] B;
4   output [7:0] SUM;
5   output CO;
6   wire [8:0] tmp;
7   assign tmp = A + B;
8   assign SUM = tmp [7:0];
9   assign CO = tmp [8];
10 endmodule

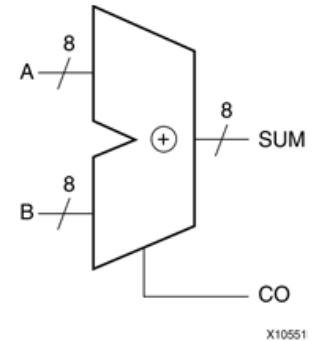
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5 entity adders_3 is
6   port(A,B : in std_logic_vector(7 downto 0);
7        SUM : out std_logic_vector(7 downto 0);
8        CO : out std_logic);
9 end adders_3;
10 architecture archi of adders_3 is
11 signal tmp: std_logic_vector(8 downto 0);
12 begin
13   tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)),9);
14   SUM <= tmp(7 downto 0);
15   CO <= tmp(8);
16 end archi;

```



Unsigned Adder with Carry in and Carry Out

Verilog

```

1 module v_adders_4(A, B, CI, SUM, CO);
2   input CI;
3   input [7:0] A;
4   input [7:0] B;
5   output [7:0] SUM;
6   output CO;
7   wire [8:0] tmp;
8   assign tmp = A + B + CI;
9   assign SUM = tmp [7:0];
10  assign CO = tmp [8];
11 endmodule

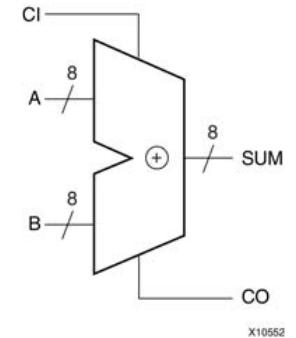
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5 entity adders_4 is
6   port(A,B : in std_logic_vector(7 downto 0);
7        CI : in std_logic;
8        SUM : out std_logic_vector(7 downto 0);
9        CO : out std_logic);
10 end adders_4;
11 architecture archi of adders_4 is
12 signal tmp: std_logic_vector(8 downto 0);
13 begin
14   tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B) + conv_integer(CI)),8);
15   SUM <= tmp(7 downto 0);
16   CO <= tmp(8);
17 end archi;

```



Signed Adder

Verilog

```

1 module vadders_5 (A,B,SUM);
2   input signed [7:0] A;
3   input signed [7:0] B;
4   output signed [7:0] SUM;
5   wire signed [7:0] SUM;
6   assign SUM = A + B;
7 endmodule

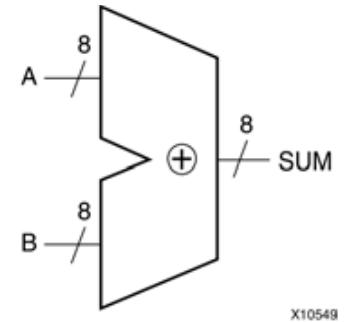
```

VHDL

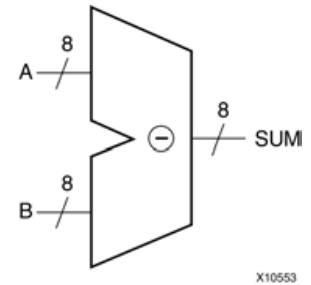
```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_signed.all;
4 entity adders_5 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        SUM : out std_logic_vector(7 downto 0));
7 end adders_5;
8 architecture archi of adders_5 is
9 begin
10   SUM <= A + B;
11 end archi;

```



Unsigned Subtractor



Verilog

```

1 module vadders_6(A, B, RES);
2   input [7:0] A;
3   input [7:0] B;
4   output [7:0] RES;
5   assign RES = A - B;
6 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity adders_6 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        RES : out std_logic_vector(7 downto 0));
7 end adders_6;
8 architecture archi of adders_6 is
9 begin
10   RES <= A - B;
11 end archi;

```

Unsigned Subtractor with Borrow

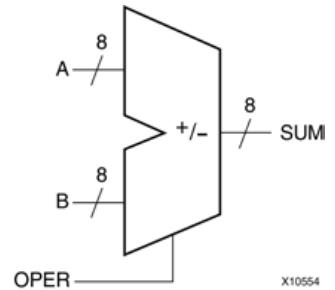
Verilog

```
1 module v_adders_8(A, B, BI, RES);
2   input [7:0] A;
3   input [7:0] B;
4   input BI;
5   output [7:0] RES;
6   assign RES = A - B - BI;
7 endmodule
```

VHDL

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 entity adders_8 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        BI : in std_logic;
7        RES : out std_logic_vector(7 downto 0));
8 end adders_8;
9 architecture archi of adders_8 is
10 begin
11   RES <= A - B - BI;
12 end archi;
```

Unsigned Adder/Subtractor



Verilog

```

1 module v_adders_7(A, B, OPER, RES);
2   input OPER;
3   input [7:0] A;
4   input [7:0] B;
5   output [7:0] RES;
6   reg [7:0] RES;
7   always @ (A or B or OPER)
8   begin
9     if (OPER==1'b0) RES = A + B;
10    else RES = A - B;
11  end
12 endmodule

```

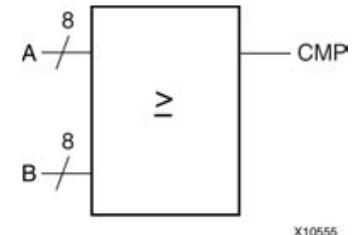
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity adders_7 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        OPER: in std_logic;
7        RES : out std_logic_vector(7 downto 0));
8 end adders_7;
9 architecture archi of adders_7 is
10 begin
11   RES <= A + B when OPER='0'
12   else A - B;
13 end archi;

```

Unsigned Greater or Equal Comparator



Verilog

```

1 module v_comparator_1 (A, B, CMP);
2   input [7:0] A;
3   input [7:0] B;
4   output CMP;
5   assign CMP = (A >= B) ? 1'b1 : 1'b0;
6 endmodule

```

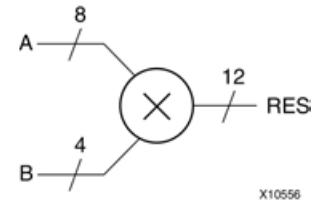
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity comparator_1 is
5   port(A,B : in std_logic_vector(7 downto 0);
6        CMP : out std_logic);
7 end comparator_1;
8 architecture archi of comparator_1 is
9 begin
10   CMP <= '1' when A >= B else '0';
11 end archi;

```

Unsigned Multiplier



Verilog

```

1 module v_multipliers_1(A, B, RES);
2   input [7:0] A;
3   input [3:0] B;
4   output [11:0] RES;
5   assign RES = A * B;
6 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity multipliers_1 is
5   port(A : in std_logic_vector(7 downto 0);
6        B : in std_logic_vector(3 downto 0);
7        RES : out std_logic_vector(11 downto 0));
8 end multipliers_1;
9 architecture beh of multipliers_1 is
10 begin
11   RES <= A * B;
12 end beh;

```

Sequential Complex Multipliers in Verilog

```

1 module v_multipliers_8(CLK,A,B,Oper_Load,Oper_AddSub, RES);
2   parameter A_WIDTH = 18;
3   parameter B_WIDTH = 18;
4   parameter RES_WIDTH = 48;
5   parameter P_WIDTH = A_WIDTH+B_WIDTH;
6   input CLK;
7   input signed [A_WIDTH-1:0] A, B;
8   input Oper_Load, Oper_AddSub;
9   // Oper_Load Oper_AddSub Operation
10  // 0 0 R= +A*B
11  // 0 1 R= -A*B
12  // 1 0 R=R+A*B
13  // 1 1 R=R-A*B
14   output [RES_WIDTH-1:0] RES;
15   reg oper_load0 = 0;
16   reg oper_addsub0 = 0;
17   reg signed [P_WIDTH-1:0] p1 = 0;
18   reg oper_load1 = 0;
19   reg oper_addsub1 = 0;
20   reg signed [RES_WIDTH-1:0] res0 = 0;
21   reg signed [RES_WIDTH-1:0] acc;
22
23   always @ (posedge CLK)
24     begin
25       oper_load0 <= Oper_Load;
26       oper_addsub0 <= Oper_AddSub;
27       p1 <= A*B;
28       oper_load1 <= oper_load0;
29       oper_addsub1 <= oper_addsub0;
30       if (oper_load1==1'b1)
31         acc = res0;
32       else
33         acc = 0;
34       if (oper_addsub1==1'b1)
35         res0 <= acc-p1;
36       else
37         res0 <= acc+p1;
38     end
39   assign RES = res0;
40 endmodule

```

Note:

Considering that $(a_r + ja_i)(b_r + jb_i) = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)$:

- The first two cycles compute:

$$\text{Res_real} = A_{\text{real}} * B_{\text{real}} - A_{\text{imag}} * B_{\text{imag}}$$
- The second two cycles compute:

$$\text{Res_imag} = A_{\text{real}} * B_{\text{imag}} + A_{\text{imag}} * B_{\text{real}}$$

Sequential Complex Multipliers in VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity multipliers_8 is
5   generic(A_WIDTH: positive:=18;
6           B_WIDTH: positive:=18;
7           RES_WIDTH: positive:=48);
8   port( CLK: in std_logic;
9         A: in signed(A_WIDTH-1 downto 0);
10        B: in signed(B_WIDTH-1 downto 0);
11        Oper_Load: in std_logic;
12        Oper_AddSub: in std_logic;
13        -- Oper_Load Oper_AddSub Operation
14        -- 0 0 R= +A*B
15        -- 0 1 R= -A*B
16        -- 1 0 R=R+A*B
17        -- 1 1 R=R-A*B
18        RES: out signed(RES_WIDTH-1 downto 0)
19      );
20 end multipliers_8;

```

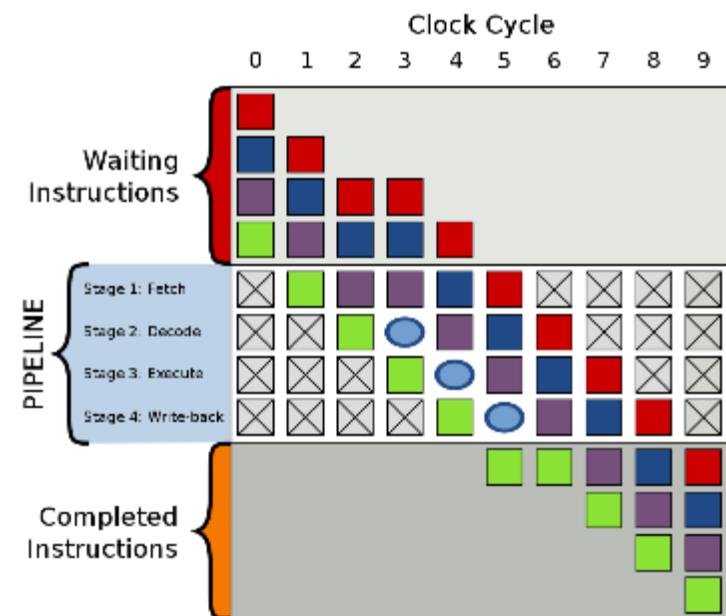
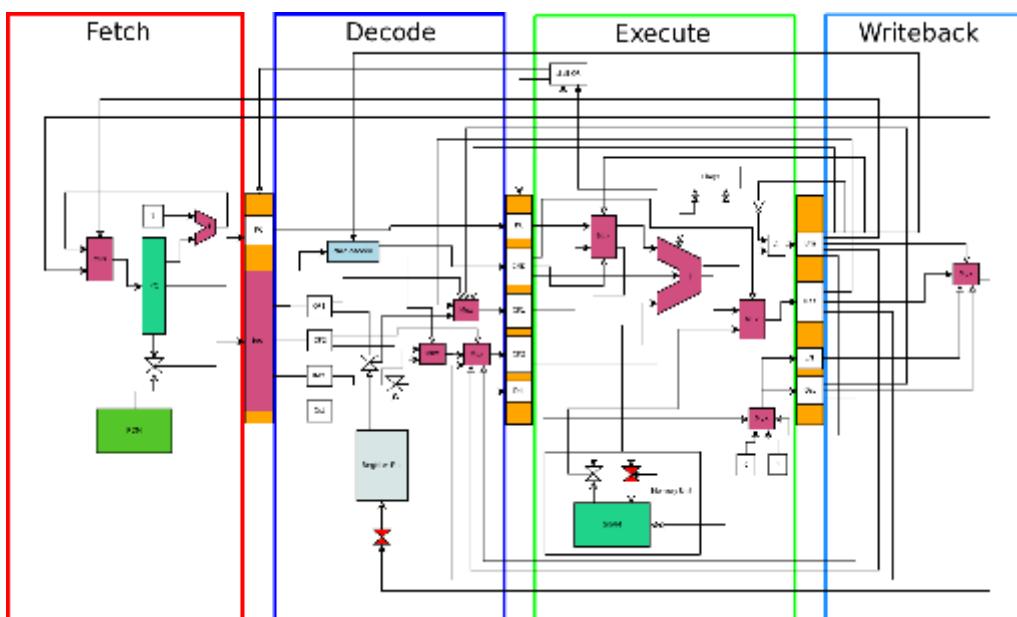
```

21 architecture beh of multipliers_8 is
22   constant P_WIDTH: integer:=A_WIDTH+B_WIDTH;
23   signal oper_load0: std_logic:='0';
24   signal oper_addsub0: std_logic:='0';
25   signal p1: signed(P_WIDTH-1 downto 0):=(others=>'0');
26   signal oper_load1: std_logic:='0';
27   signal oper_addsub1: std_logic:='0';
28   signal res0: signed(RES_WIDTH-1 downto 0);
29 begin
30   process (clk)
31     variable acc: signed(RES_WIDTH-1 downto 0);
32   begin
33     if rising_edge(clk) then
34       oper_load0 <= Oper_Load;
35       oper_addsub0 <= Oper_AddSub;
36       p1 <= A*B;
37       oper_load1 <= oper_load0;
38       oper_addsub1 <= oper_addsub0;
39       if (oper_load1='1') then
40         acc := res0;
41       else
42         acc := (others=>'0');
43       end if;
44       if (oper_addsub1='1') then
45         res0 <= acc-p1;
46       else
47         res0 <= acc+p1;
48       end if;
49     end if;
50   end process;
51   RES <= res0;
52 end architecture;

```

Pipelining

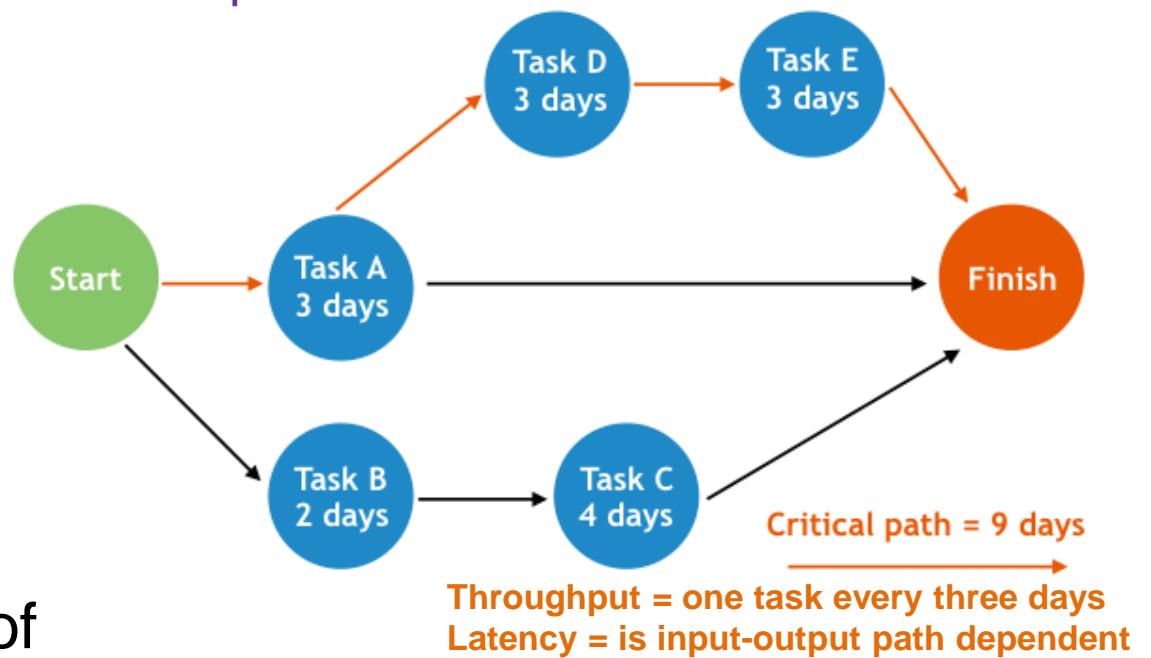
- Pipelining is a general technique for improving design timing and hardware utilization efficiency by using parallel units that simultaneously process the output of preceding stages of the pipeline.
- Implementing combinational logic using pipelines can significantly reduce the critical path delay.



A Few Definitions

- **(Input-Output) Latency:** the amount of time it takes to travel through the pipe.
- **Critical Path:** Longest combinational path between the output of one flip-flop to the input of another flip-flop (sharing a common clock)
- **Throughput:** The maximum rate of data flowing in or out of a data-path (the inverse of the critical path)

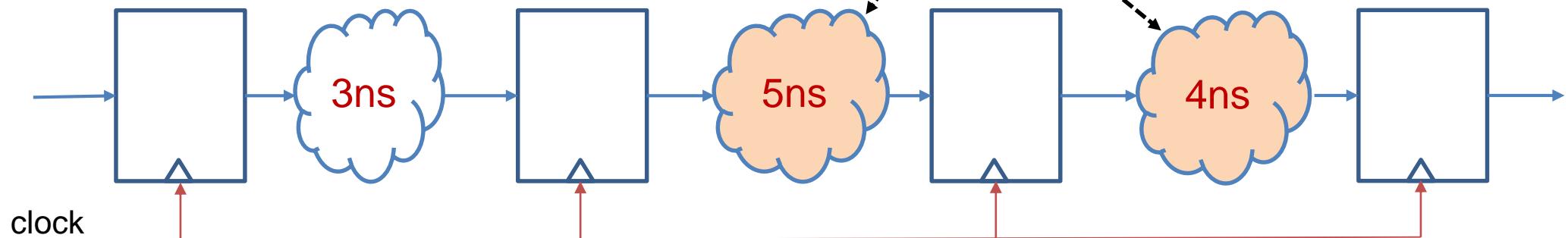
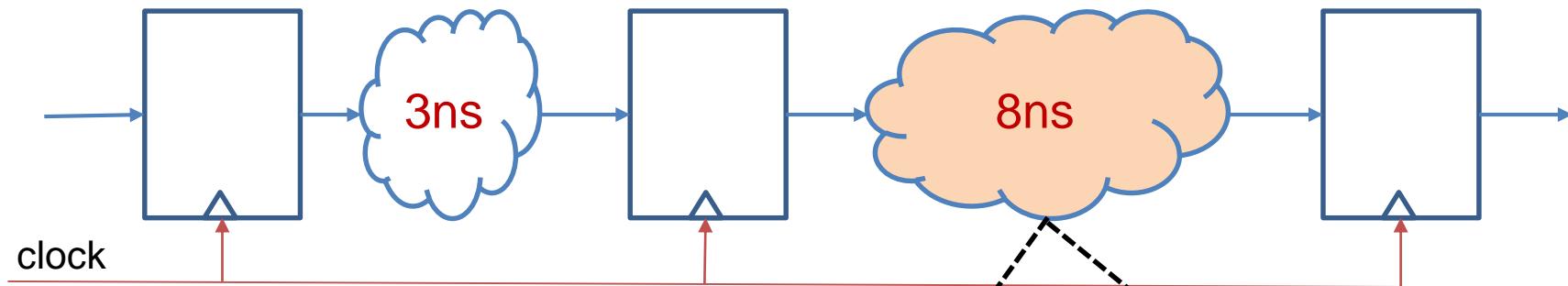
Example:



Pipelining Critical Paths

- Pipelining can shorten the critical path and improve the throughput (possibly) at a cost of an increased latency between the input-output

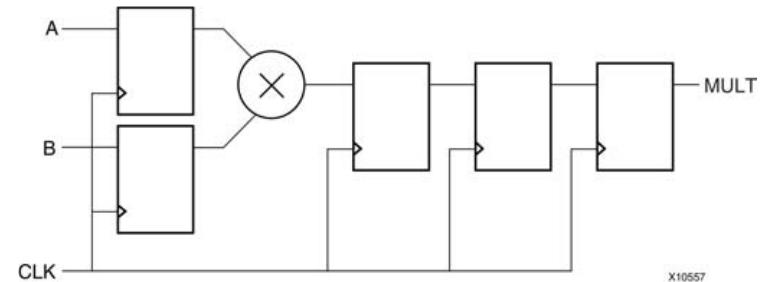
Critical path = 8ns, Max Throughput = 125MHz, I/O Latency = 3 clocks (24ns @ $f_{clock}=125MHz$)



New critical path = 5ns, Max Throughput = 200MHz, I/O Latency = 4 clocks (20ns @ $f_{clock}=200MHz$)

(we will discuss much more about pipelining in digital systems design up to end of the course)

Pipelined Multiplier (Outside, Single)



Verilog

```

1 (*mult_style="pipe_lut")
2 module v_multipliers_2(clk, A, B, MULT);
3     input clk;
4     input [17:0] A;
5     input [17:0] B;
6     output [35:0] MULT;
7     reg [35:0] MULT;
8     reg [17:0] a_in, b_in;
9     wire [35:0] mult_res;
10    reg [35:0] pipe_1, pipe_2, pipe_3;
11    assign mult_res = a_in * b_in;
12    always @(posedge clk)
13    begin
14        a_in <= A; b_in <= B;
15        pipe_1 <= mult_res;
16        pipe_2 <= pipe_1;
17        pipe_3 <= pipe_2;
18        MULT <= pipe_3;
19    end
20 endmodule

```

Note: This code is automatically replaced by a four-stage pipeline multiplier, only if the intermediate pipeline registers (`pipe_1`, `pipe_2` and `pipe_3`) are not used elsewhere in the code. **Question:** Why?

VHDL

```

2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity multipliers_2 is
5     generic(A_port_size : integer := 18;
6             B_port_size : integer := 18);
7     port(clk : in std_logic;
8           A : in unsigned (A_port_size-1 downto 0);
9           B : in unsigned (B_port_size-1 downto 0);
10          MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));
11          attribute mult_style: string;
12          attribute mult_style of multipliers_2: entity is "pipe_lut";
13 end multipliers_2;
14 architecture beh of multipliers_2 is
15     signal a_in, b_in : unsigned (A_port_size-1 downto 0);
16     signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
17     signal pipe_1,
18     pipe_2,
19     pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);
20 begin
21     mult_res <= a_in * b_in;
22     process (clk)
23     begin
24         if (clk'event and clk='1') then
25             a_in <= A; b_in <= B;
26             pipe_1 <= mult_res;
27             pipe_2 <= pipe_1;
28             pipe_3 <= pipe_2;
29             MULT <= pipe_3;
30         end if;
31     end process;
32 end beh;

```

Pipelined Multiplier (Inside, Single)

Verilog

```

1 (*mult_style="pipe_lut"*)
2 module v_multipliers_3(clk, A, B, MULT);
3     input clk;
4     input [17:0] A;
5     input [17:0] B;
6     output [35:0] MULT;
7     reg [35:0] MULT;
8     reg [17:0] a_in, b_in;
9     reg [35:0] mult_res;
10    reg [35:0] pipe_2, pipe_3;
11    always @(posedge clk)
12    begin
13        a_in <= A; b_in <= B;
14        mult_res <= a_in * b_in;
15        pipe_2 <= mult_res;
16        pipe_3 <= pipe_2;
17        MULT <= pipe_3;
18    end
19 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity multipliers_3 is
5     generic(A_port_size: integer := 18;
6             B_port_size: integer := 18);
7     port(clk : in std_logic;
8           A : in unsigned ((A_port_size-1 downto 0));
9           B : in unsigned ((B_port_size-1 downto 0));
10          MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));
11          attribute mult_style: string;
12          attribute mult_style of multipliers_3: entity is "pipe_lut";
13 end multipliers_3;
14 architecture beh of multipliers_3 is
15     signal a_in, b_in : unsigned ((A_port_size-1 downto 0));
16     signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
17     signal pipe_2,
18     pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);
19 begin
20     process (clk)
21     begin
22         if (clk'event and clk='1') then
23             a_in <= A; b_in <= B;
24             mult_res <= a_in * b_in;
25             pipe_2 <= mult_res;
26             pipe_3 <= pipe_2;
27             MULT <= pipe_3;
28         end if;
29     end process;
30 end beh;

```

Pipelined Multiplier (Outside, Shift) in Verilog

```
1 (*mult_style="pipe_lut")
2 module v_multipliers_4(clk, A, B, MULT);
3     input clk;
4     input [17:0] A;
5     input [17:0] B;
6     output [35:0] MULT;
7     reg [35:0] MULT;
8     reg [17:0] a_in, b_in;
9     wire [35:0] mult_res;
10    reg [35:0] pipe_regs [2:0];
11    integer i;
12    assign mult_res = a_in * b_in;
13    always @ (posedge clk)
14    begin
15        a_in <= A; b_in <= B;
16        pipe_regs[2] <= mult_res;
17        for (i=0; i<=1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];
18        MULT <= pipe_regs[0];
19    end
20 endmodule
```

Pipelined Multiplier (Outside, Shift) in VHDL

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity multipliers_4 is
5     generic(A_port_size: integer := 18;
6             B_port_size: integer := 18);
7     port(clk : in std_logic;
8           A : in unsigned (A_port_size-1 downto 0);
9           B : in unsigned (B_port_size-1 downto 0);
10          MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));
11 attribute mult_style: string;
12 attribute mult_style of multipliers_4: entity is "pipe_lut";
13 end multipliers_4;
14 architecture beh of multipliers_4 is
15 signal a_in, b_in : unsigned (A_port_size-1 downto 0);
16 signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
17 type pipe_reg_type is array (2 downto 0) of unsigned ((A_port_size+B_port_size-1) downto 0);
18 signal pipe_regs : pipe_reg_type;
19 begin
20     mult_res <= a_in * b_in;
21     process (clk)
22     begin
23         if (clk'event and clk='1') then
24             a_in <= A; b_in <= B;
25             pipe_regs <= mult_res & pipe_regs(2 downto 1);
26             MULT <= pipe_regs(0);
27         end if;
28     end process;
29 end beh;
```

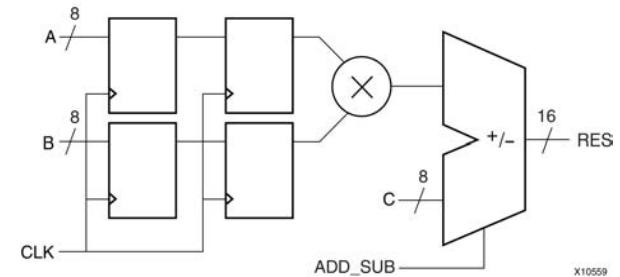
Multiplier Adder With 2 Register Levels on Multiplier Inputs

Verilog

```

1 module v_multipliers_5 (clk, A, B, C, RES);
2   input clk;
3   input [7:0] A;
4   input [7:0] B;
5   input [7:0] C;
6   output [15:0] RES;
7   reg [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
8   wire [15:0] multaddsub;
9   always @(posedge clk)
10 begin
11   A_reg1 <= A; A_reg2 <= A_reg1;
12   B_reg1 <= B; B_reg2 <= B_reg1;
13 end
14 assign multaddsub = A_reg2 * B_reg2 + C;
15 assign RES = multaddsub;
16 endmodule

```



VHDL

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 entity multipliers_5 is
5   generic (p_width: integer:=8);
6   port (clk : in std_logic;
7         A, B, C : in std_logic_vector(p_width-1 downto 0);
8         RES : out std_logic_vector(p_width*2-1 downto 0));
9 end multipliers_5;
10 architecture beh of multipliers_5 is
11   signal A_reg1, A_reg2,
12   B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
13   signal multaddsub : std_logic_vector(p_width*2-1 downto 0);
14 begin
15   multaddsub <= A_reg2 * B_reg2 + C;
16   process (clk)
17     begin
18       if (clk'event and clk='1') then
19         A_reg1 <= A; A_reg2 <= A_reg1;
20         B_reg1 <= B; B_reg2 <= B_reg1;
21       end if;
22     end process;
23   RES <= multaddsub;
24 end beh;

```

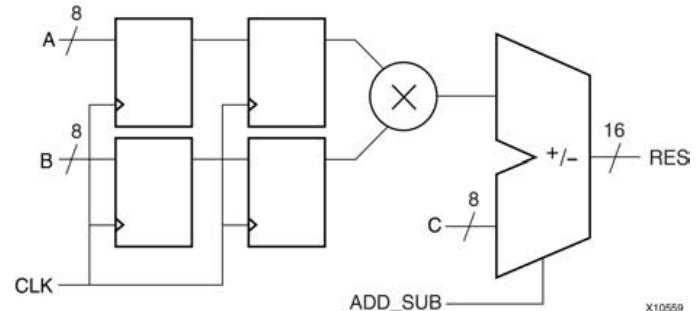
Multiplier Adder/Subtractor With 2 Register Levels on Multiplier Inputs

Verilog

```

1 module v_multipliers_6 (clk, add_sub, A, B, C, RES);
2   input clk,add_sub;
3   input [7:0] A;
4   input [7:0] B;
5   input [7:0] C;
6   output [15:0] RES;
7   reg [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
8   wire [15:0] mult, multaddsub;
9   always @(posedge clk)
10 begin
11   A_reg1 <= A; A_reg2 <= A_reg1;
12   B_reg1 <= B; B_reg2 <= B_reg1;
13 end
14 assign mult = A_reg2 * B_reg2;
15 assign multaddsub = add_sub ? C + mult : C - mult;
16 assign RES = multaddsub;
17 endmodule

```



VHDL

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 entity multipliers_6 is
5   generic (p_width: integer:=8);
6   port (clk,add_sub: in std_logic;
7         A, B, C: in std_logic_vector(p_width-1 downto 0);
8         RES: out std_logic_vector(p_width*2-1 downto 0));
9 end multipliers_6;
10 architecture beh of multipliers_6 is
11   signal A_reg1, A_reg2,
12   B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
13   signal mult, multaddsub : std_logic_vector(p_width*2-1 downto 0);
14 begin
15   mult <= A_reg2 * B_reg2;
16   multaddsub <= C + mult when add_sub = '1' else C - mult;
17   process (clk)
18   begin
19     if (clk'event and clk='1') then
20       A_reg1 <= A; A_reg2 <= A_reg1;
21       B_reg1 <= B; B_reg2 <= B_reg1;
22     end if;
23   end process;
24   RES <= multaddsub;
25 end beh;

```

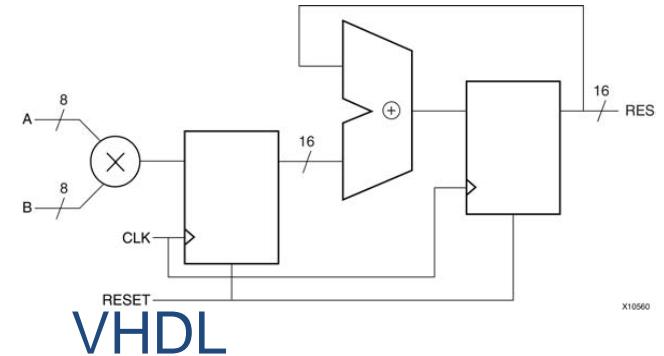
Multiplier Up Accumulate with Register after Multiplication

Verilog

```

1 module v_multipliers_7a (clk, reset, A, B, RES);
2   input clk, reset;
3   input [7:0] A;
4   input [7:0] B;
5   output [15:0] RES;
6   reg [15:0] mult, accum;
7   always @ (posedge clk)
8   begin
9     if (reset)
10       mult <= 16'b0000000000000000;
11     else
12       mult <= A * B;
13   end
14   always @ (posedge clk)
15   begin
16     if (reset)
17       accum <= 16'b0000000000000000;
18     else
19       accum <= accum + mult;
20   end
21   assign RES = accum;
22 endmodule

```



```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 entity multipliers_7a is
5   generic (p_width: integer:=8);
6   port (clk, reset: in std_logic;
7         A, B: in std_logic_vector(p_width-1 downto 0);
8         RES: out std_logic_vector(p_width*2-1 downto 0));
9 end multipliers_7a;
10 architecture beh of multipliers_7a is
11 signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
12 begin
13   process (clk)
14   begin
15     if (clk'event and clk='1') then
16       if (reset = '1') then
17         accum <= (others => '0');
18         mult <= (others => '0');
19       else
20         accum <= accum + mult;
21         mult <= A * B;
22       end if;
23     end if;
24   end process;
25   RES <= accum;
26 end beh;

```

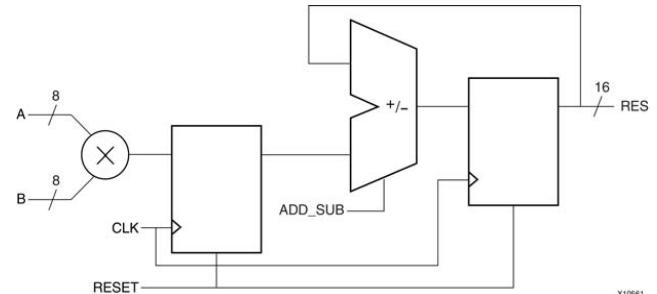
Multiplier Up/Down Accumulate with Register after Multiplication

Verilog

```

1 module v_multipliers_7b (clk, reset, add_sub, A, B, RES);
2   input clk, reset, add_sub;
3   input [7:0] A;
4   input [7:0] B;
5   output [15:0] RES;
6   reg [15:0] mult, accum;
7   always @(posedge clk)
8   begin
9     if (reset)
10       mult <= 16'b0000000000000000;
11     else
12       mult <= A * B;
13   end
14   always @(posedge clk)
15   begin
16     if (reset)
17       accum <= 16'b0000000000000000;
18     else
19       if (add_sub)
20         accum <= accum + mult;
21       else
22         accum <= accum - mult;
23   end
24   assign RES = accum;
25 endmodule

```



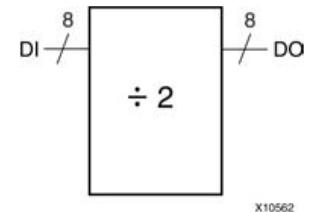
VHDL

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 entity multipliers_7b is
5   generic (p_width: integer:=8);
6   port (clk, reset, add_sub: in std_logic;
7         A, B: in std_logic_vector(p_width-1 downto 0);
8         RES: out std_logic_vector(p_width*2-1 downto 0));
9 end multipliers_7b;
10 architecture beh of multipliers_7b is
11   signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
12 begin
13   process (clk)
14   begin
15     if (clk'event and clk='1') then
16       if (reset = '1') then
17         accum <= (others => '0');
18       mult <= (others => '0');
19     else
20       if (add_sub = '1') then
21         accum <= accum + mult;
22       else
23         accum <= accum - mult;
24       end if;
25       mult <= A * B;
26     end if;
27   end process;
28   RES <= accum;
29 end beh;
30

```

Division by Constant Powers of 2 Dividers



Verilog

```

1 module v_divider_1 (DI, DO);
2   input [7:0] DI;
3   output [7:0] DO;
4   assign DO = DI / 2;
5 endmodule

```

Notes:

- Dividers are supported only when the divisor is a constant and is a power of 2. In that case, the operator is implemented as a shifter. Otherwise, XST issues an error message.
- IP cores or custom code can be used for other divisors.

Question: Why aren't dividers built-in primitives like adders and multipliers?

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity divider_1 is
5   port(DI : in unsigned(7 downto 0);
6        DO : out unsigned(7 downto 0));
7 end divider_1;
8 architecture archi of divider_1 is
9 begin
10   DO <= DI / 2;
11 end archi;

```

Resource Sharing (Hardware Reuse)

- The goal of **resource sharing** (also known as **hardware reuse** or **folding**) is to minimize the number of operators and the subsequent logic in the synthesized design. This optimization is based on the principle that two similar arithmetic resources may be implemented as one single arithmetic operator if they are never used at the same time.
- Resource sharing is commonly handled by synthesis tools automatically, unless if prevented by user constraints and synthesis attributes.
- If the optimization goal is speed, disabling resource sharing may give better results.

(we will discuss much more about resource sharing in digital systems design up to end of the course)

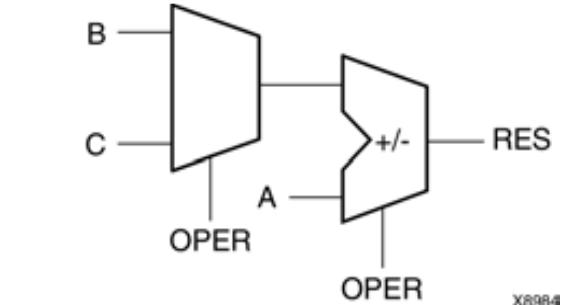
Resource Sharing Example

Verilog

```

1 module v_resource_sharing_1 (A, B, C, OPER, RES);
2   input [7:0] A, B, C;
3   input OPER;
4   output [7:0] RES;
5   wire [7:0] RES;
6   assign RES = !OPER ? A + B : A - C;
7 endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity resource_sharing_1 is
5   port(A,B,C : in std_logic_vector(7 downto 0);
6        OPER : in std_logic;
7        RES : out std_logic_vector(7 downto 0));
8 end resource_sharing_1;
9 architecture archi of resource_sharing_1 is
10 begin
11   RES <= A + B when OPER='0' else A - C;
12 end archi;

```

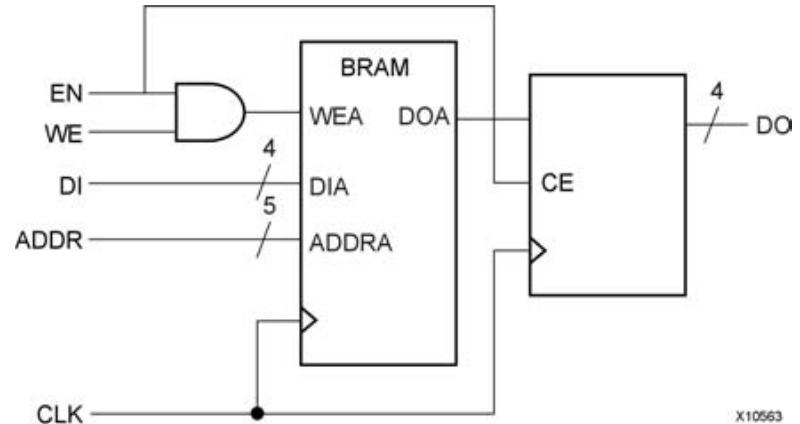
Single-Port RAM in Read-First Mode

Verilog

```

1 module v_rams_01 (clk, en, we, addr, di, dout);
2   input clk;
3   input we;
4   input en;
5   input [5:0] addr;
6   input [15:0] di;
7   output [15:0] dout;
8   reg [15:0] RAM [63:0];
9   reg [15:0] dout;
10  always @(posedge clk)
11 begin
12   if (en)
13     begin
14       if (we)
15         RAM[addr]<=di;
16         dout <= RAM[addr];
17     end
18   end
19 endmodule

```



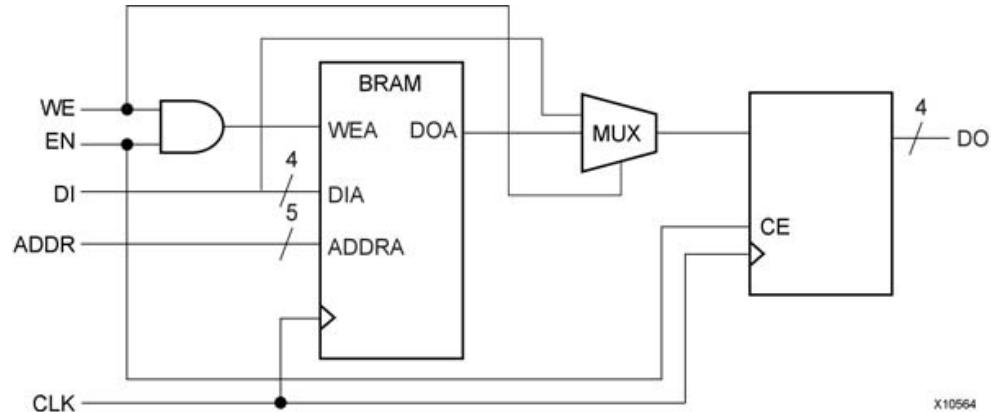
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_01 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         en : in std_logic;
8         addr : in std_logic_vector(5 downto 0);
9         di : in std_logic_vector(15 downto 0);
10        do : out std_logic_vector(15 downto 0));
11 end rams_01;
12 architecture syn of rams_01 is
13   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
14   signal RAM: ram_type;
15 begin
16   process (clk)
17   begin
18     if clk'event and clk = '1' then
19       if en = '1' then
20         if we = '1' then
21           RAM(conv_integer(addr)) <= di;
22         end if;
23         do <= RAM(conv_integer(addr));
24       end if;
25     end if;
26   end process;
27 end syn;

```

Single-Port RAM in Write-First Mode in Verilog



Template 1

```

1 module v_rams_02a (clk, we, en, addr, di, dout);
2   input clk;
3   input we;
4   input en;
5   input [5:0] addr;
6   input [15:0] di;
7   output [15:0] dout;
8   reg [15:0] RAM [63:0];
9   reg [15:0] dout;
10  always @(posedge clk)
11    begin
12      if (en)
13        begin
14          if (we)
15            begin
16              RAM[addr] <= di;
17              dout <= di;
18            end
19          else
20              dout <= RAM[addr];
21        end
22    end
23 endmodule

```

Template 2

```

1 module v_rams_02b (clk, we, en, addr, di, dout);
2   input clk;
3   input we;
4   input en;
5   input [5:0] addr;
6   input [15:0] di;
7   output [15:0] dout;
8   reg [15:0] RAM [63:0];
9   reg [5:0] read_addr;
10  always @(posedge clk)
11    begin
12      if (en)
13        begin
14          if (we)
15              RAM[addr] <= di;
16              read_addr <= addr;
17            end
18        end
19      assign dout = RAM[read_addr];
20    endmodule

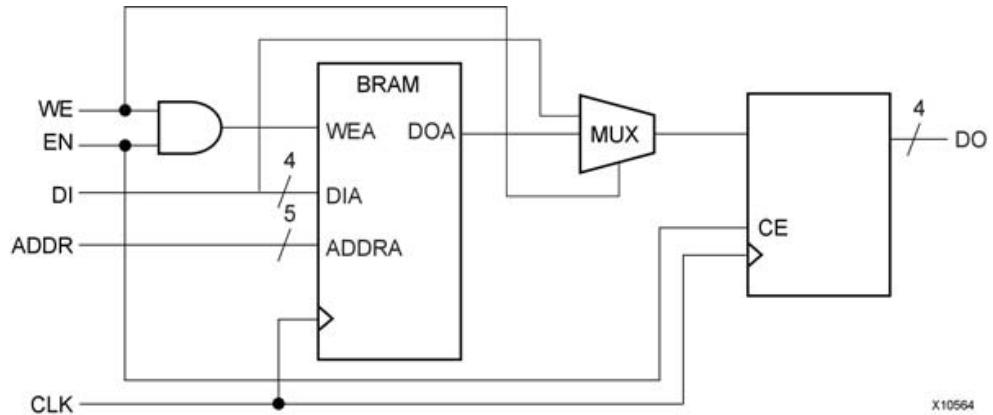
```

Single-Port RAM in Write-First Mode in VHDL

Template 1

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_02a is
5     port (clk : in std_logic;
6             we : in std_logic;
7             en : in std_logic;
8             addr : in std_logic_vector(5 downto 0);
9             di : in std_logic_vector(15 downto 0);
10            do : out std_logic_vector(15 downto 0));
11 end rams_02a;
12 architecture syn of rams_02a is
13     type ram_type is array (63 downto 0)
14         of std_logic_vector (15 downto 0);
15     signal RAM : ram_type;
16 begin
17     process (clk)
18     begin
19         if clk'event and clk = '1' then
20             if en = '1' then
21                 if we = '1' then
22                     RAM(conv_integer(addr)) <= di;
23                     do <= di;
24                 else
25                     do <= RAM( conv_integer(addr));
26                 end if;
27             end if;
28         end process;
29     end syn;
30 
```



Template 2

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_02b is
5     port (clk : in std_logic;
6             we : in std_logic;
7             en : in std_logic;
8             addr : in std_logic_vector(5 downto 0);
9             di : in std_logic_vector(15 downto 0);
10            do : out std_logic_vector(15 downto 0));
11 end rams_02b;
12 architecture syn of rams_02b is
13     type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
14     signal RAM : ram_type;
15     signal read_addr: std_logic_vector(5 downto 0);
16 begin
17     process (clk)
18     begin
19         if clk'event and clk = '1' then
20             if en = '1' then
21                 if we = '1' then
22                     ram(conv_integer(addr)) <= di;
23                 end if;
24                 read_addr <= addr;
25             end if;
26         end if;
27     end process;
28     do <= ram(conv_integer(read_addr));
29 end syn;

```

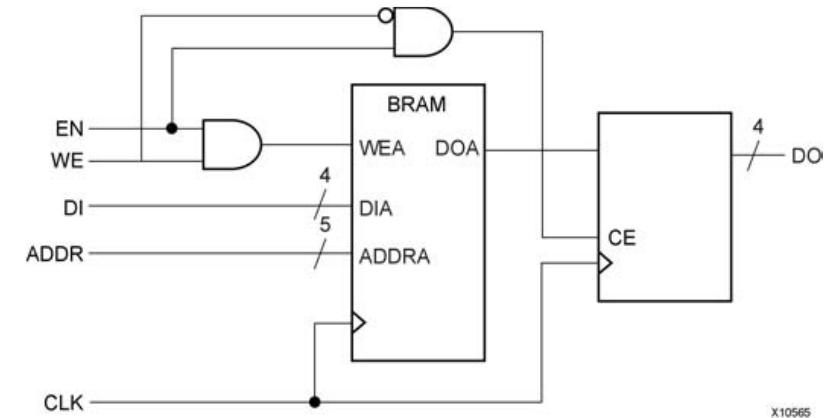
Single-Port RAM in No-Change Mode

Verilog

```

1 module v_rams_03 (clk, we, en, addr, di, dout);
2   input clk;
3   input we;
4   input en;
5   input [5:0] addr;
6   input [15:0] di;
7   output [15:0] dout;
8   reg [15:0] RAM [63:0];
9   reg [15:0] dout;
10  always @(posedge clk)
11    begin
12      if (en)
13        begin
14          if (we)
15            RAM[addr] <= di;
16          else
17            <= RAM[addr];
18        end
19    end
20 endmodule

```



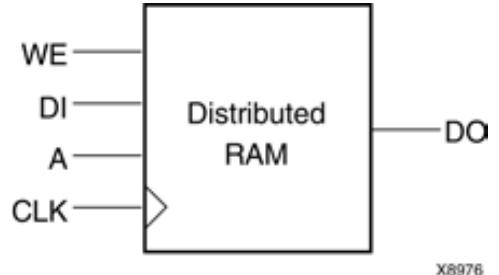
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_03 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         en : in std_logic;
8         addr : in std_logic_vector(5 downto 0);
9         di : in std_logic_vector(15 downto 0);
10        do : out std_logic_vector(15 downto 0));
11 end rams_03;
12 architecture syn of rams_03 is
13   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
14   signal RAM : ram_type;
15 begin
16   process (clk)
17   begin
18     if clk'event and clk = '1' then
19       if en = '1' then
20         if we = '1' then
21           RAM(conv_integer(addr)) <= di;
22         else
23           do <= RAM(conv_integer(addr));
24         end if;
25       end if;
26     end if;
27   end process;
28 end syn;

```

Single-Port RAM with Asynchronous Read



Verilog

```

1 module v_rams_04 (clk, we, a, di, dout);
2   input clk;
3   input we;
4   input [5:0] a;
5   input [15:0] di;
6   output [15:0] dout;
7   reg [15:0] ram [63:0];
8   always @ (posedge clk) begin
9     if (we)
10       ram[a] <= di;
11   end
12   assign dout = ram[a];
13 endmodule

```

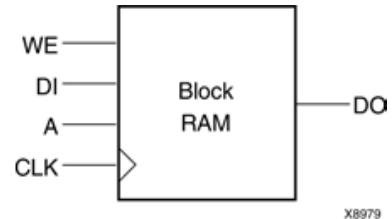
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_04 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         a : in std_logic_vector(5 downto 0);
8         di : in std_logic_vector(15 downto 0);
9         do : out std_logic_vector(15 downto 0));
10 end rams_04;
11 architecture syn of rams_04 is
12   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
13   signal RAM : ram_type;
14 begin
15   process (clk)
16   begin
17     if (clk'event and clk = '1') then
18       if (we = '1') then
19         RAM(conv_integer(a)) <= di;
20       end if;
21     end if;
22   end process;
23   do <= RAM(conv_integer(a));
24 end syn;

```

Single-Port RAM with Synchronous Read (Read Through)



Verilog

```

1 module v_rams_07 (clk, we, a, di, do);
2   input clk;
3   input we;
4   input [5:0] a;
5   input [15:0] di;
6   output [15:0] do;
7   reg [15:0] ram [63:0];
8   reg [5:0] read_a;
9   always @(posedge clk) begin
10     if (we)
11       ram[a] <= di;
12     read_a <= a;
13   end
14   assign do = ram[read_a];
15 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_07 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         a : in std_logic_vector(5 downto 0);
8         di : in std_logic_vector(15 downto 0);
9         do : out std_logic_vector(15 downto 0));
10 end rams_07;
11 architecture syn of rams_07 is
12   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
13   signal RAM : ram_type;
14   signal read_a : std_logic_vector(5 downto 0);
15 begin
16   process (clk)
17   begin
18     if (clk'event and clk = '1') then
19       if (we = '1') then
20         RAM(conv_integer(a)) <= di;
21       end if;
22       read_a <= a;
23     end if;
24   end process;
25   do <= RAM(conv_integer(read_a));
26 end syn;

```

Single-Port RAM with Enable

Verilog

```

1 module v_rams_08 (clk, en, we, a, di, dout);
2   input clk;
3   input en;
4   input we;
5   input [5:0] a;
6   input [15:0] di;
7   output [15:0] dout;
8   reg [15:0] ram [63:0];
9   reg [5:0] read_a;
10  always @(posedge clk) begin
11    if (en)
12      begin
13        if (we)
14          ram[a] <= di;
15        read_a <= a;
16      end
17    end
18    assign dout = ram[read_a];
19 endmodule

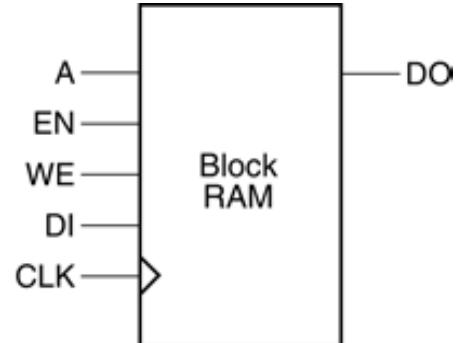
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_08 is
5   port (clk : in std_logic;
6         en : in std_logic;
7         we : in std_logic;
8         a : in std_logic_vector(5 downto 0);
9         di : in std_logic_vector(15 downto 0);
10        do : out std_logic_vector(15 downto 0));
11 end rams_08;
12 architecture syn of rams_08 is
13   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
14   signal RAM : ram_type;
15   signal read_a : std_logic_vector(5 downto 0);
16 begin
17   process (clk)
18     begin
19       if (clk'event and clk = '1') then
20         if (en = '1') then
21           if (we = '1') then
22             RAM(conv_integer(a)) <= di;
23           end if;
24           read_a <= a;
25         end if;
26       end if;
27     end process;
28     do <= RAM(conv_integer(read_a));
29 end syn;

```



Dual-Port RAM with Asynchronous Read

Verilog

```

1 module v_rams_09 (clk, we, a, dpra, di, spo, dpo);
2   input clk;
3   input we;
4   input [5:0] a;
5   input [5:0] dpra;
6   input [15:0] di;
7   output [15:0] spo;
8   output [15:0] dpo;
9   reg [15:0] ram [63:0];
10  always @ (posedge clk) begin
11    if (we)
12      ram[a] <= di;
13  end
14  assign spo = ram[a];
15  assign dpo = ram[dpra];
16 endmodule

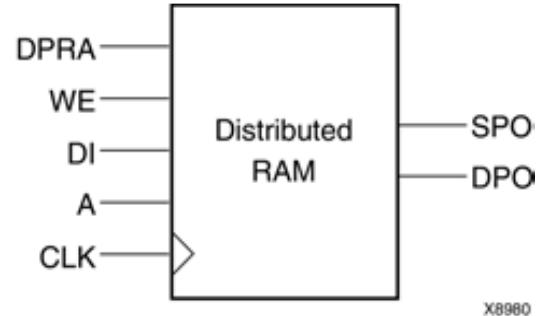
```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_09 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         a : in std_logic_vector(5 downto 0);
8         dpra : in std_logic_vector(5 downto 0);
9         di : in std_logic_vector(15 downto 0);
10        spo : out std_logic_vector(15 downto 0);
11        dpo : out std_logic_vector(15 downto 0));
12 end rams_09;
13 architecture syn of rams_09 is
14   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
15   signal RAM : ram_type;
16 begin
17   process (clk)
18   begin
19     if (clk'event and clk = '1') then
20       if (we = '1') then
21         RAM(conv_integer(a)) <= di;
22       end if;
23     end if;
24   end process;
25   spo <= RAM(conv_integer(a));
26   dpo <= RAM(conv_integer(dpra));
27 end syn;

```



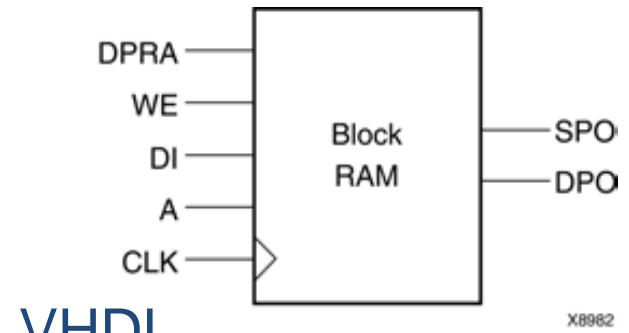
Dual-Port RAM with Synchronous Read (Read Through)

Verilog

```

1 module v_rams_11 (clk, we, a, dpra, di, spo, dpo);
2   input clk;
3   input we;
4   input [5:0] a;
5   input [5:0] dpra;
6   input [15:0] di;
7   output [15:0] spo;
8   output [15:0] dpo;
9   reg [15:0] ram [63:0];
10  reg [5:0] read_a;
11  reg [5:0] read_dpra;
12  always @(posedge clk) begin
13    if (we)
14      ram[a] <= di;
15    read_a <= a;
16    read_dpra <= dpra;
17  end
18  assign spo = ram[read_a];
19  assign dpo = ram[read_dpra];
20 endmodule

```



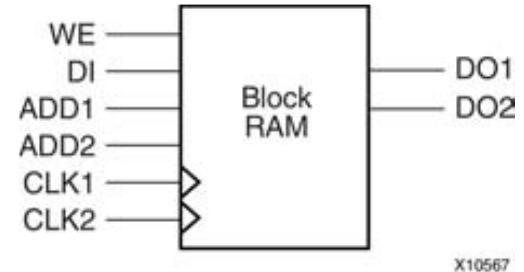
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_11 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         a : in std_logic_vector(5 downto 0);
8         dpra : in std_logic_vector(5 downto 0);
9         di : in std_logic_vector(15 downto 0);
10        spo : out std_logic_vector(15 downto 0);
11        dpo : out std_logic_vector(15 downto 0));
12 end rams_11;
13 architecture syn of rams_11 is
14   type ram_type is array (63 downto 0)
15     of std_logic_vector (15 downto 0);
16   signal RAM : ram_type;
17   signal read_a : std_logic_vector(5 downto 0);
18   signal read_dpra : std_logic_vector(5 downto 0);
19 begin
20   process (clk)
21     begin
22       if (clk'event and clk = '1') then
23         if (we = '1') then
24           RAM(conv_integer(a)) <= di;
25         end if;
26         read_a <= a;
27         read_dpra <= dpra;
28       end if;
29     end process;
30   spo <= RAM(conv_integer(read_a));
31   dpo <= RAM(conv_integer(read_dpra));
32 end syn;

```

Dual-Port RAM with Synchronous Read (Read Through) and Two Clocks



Verilog

```

1 module v_rams_12 (clk1, clk2, we, add1, add2, di, do1, do2);
2   input clk1;
3   input clk2;
4   input we;
5   input [5:0] add1;
6   input [5:0] add2;
7   input [15:0] di;
8   output [15:0] do1;
9   output [15:0] do2;
10  reg [15:0] ram [63:0];
11  reg [5:0] read_add1;
12  reg [5:0] read_add2;
13  always @(posedge clk1) begin
14    if (we)
15      ram[add1] <= di;
16    read_add1 <= add1;
17  end
18  assign do1 = ram[read_add1];
19  always @(posedge clk2) begin
20    read_add2 <= add2;
21  end
22  assign do2 = ram[read_add2];
23 endmodule

```

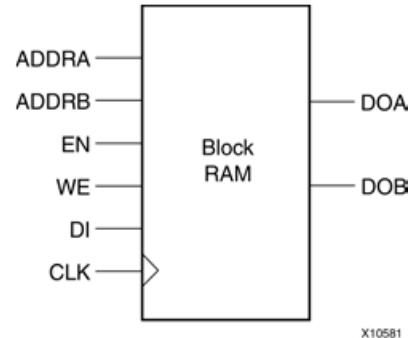
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_12 is
5   port (clk1 : in std_logic;
6         clk2 : in std_logic;
7         we : in std_logic;
8         add1 : in std_logic_vector(5 downto 0);
9         add2 : in std_logic_vector(5 downto 0);
10        di : in std_logic_vector(15 downto 0);
11        do1 : out std_logic_vector(15 downto 0);
12        do2 : out std_logic_vector(15 downto 0));
13 end rams_12;
14 architecture syn of rams_12 is
15   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
16   signal RAM : ram_type;
17   signal read_add1 : std_logic_vector(5 downto 0);
18   signal read_add2 : std_logic_vector(5 downto 0);
19 begin
20   process (clk1)
21   begin
22     if (clk1'event and clk1 = '1') then
23       if (we = '1') then
24         RAM(conv_integer(add1)) <= di;
25       end if;
26       read_add1 <= add1;
27     end if;
28   end process;
29   do1 <= RAM(conv_integer(read_add1));
30   process (clk2)
31   begin
32     if (clk2'event and clk2 = '1') then
33       read_add2 <= add2;
34     end if;
35   end process;
36   do2 <= RAM(conv_integer(read_add2));
37 end syn;

```

Dual-Port RAM with One Enable Controlling Both Ports



Verilog

```

1 module v_rams_13 (clk, en, we, addra, addrb, di, doa, dob);
2   input clk;
3   input en;
4   input we;
5   input [5:0] addra;
6   input [5:0] addrb;
7   input [15:0] di;
8   output [15:0] doa;
9   output [15:0] dob;
10  reg [15:0] ram [63:0];
11  reg [5:0] read_addr;
12  reg [5:0] read_addrb;
13  always @(posedge clk) begin
14    if (en)
15      begin
16        if (we)
17          ram[addra] <= di;
18          read_addr <= addra;
19          read_addrb <= addrb;
20      end
21  end
22  assign doa = ram[read_addr];
23  assign dob = ram[read_addrb];
24 endmodule

```

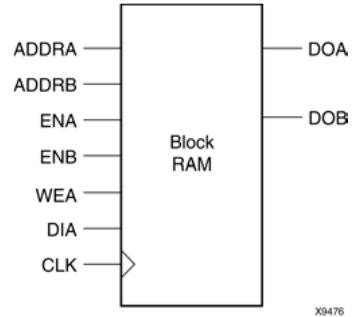
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_13 is
5   port (clk : in std_logic;
6         en : in std_logic;
7         we : in std_logic;
8         addra : in std_logic_vector(5 downto 0);
9         addrb : in std_logic_vector(5 downto 0);
10        di : in std_logic_vector(15 downto 0);
11        doa : out std_logic_vector(15 downto 0);
12        dob : out std_logic_vector(15 downto 0));
13 end rams_13;
14 architecture syn of rams_13 is
15 type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
16 signal RAM : ram_type;
17 signal read_addr : std_logic_vector(5 downto 0);
18 signal read_addrb : std_logic_vector(5 downto 0);
19 begin
20 process (clk)
21 begin
22   if (clk'event and clk = '1') then
23     if (en = '1') then
24       if (we = '1') then
25         RAM(conv_integer(addr)) <= di;
26       end if;
27       read_addr <= addra;
28       read_addrb <= addrb;
29     end if;
30   end if;
31 end process;
32 doa <= RAM(conv_integer(read_addr));
33 dob <= RAM(conv_integer(read_addrb));
34 end syn;

```

Dual Port RAM with Enable on Each Port



Verilog

```

1 module v_rams_14 (clk,ena,enb,wea,addr_a,addr_b,dia,doa,dob);
2   input clk;
3   input ena;
4   input enb;
5   input wea;
6   input [5:0] addr_a;
7   input [5:0] addr_b;
8   input [15:0] dia;
9   output [15:0] doa;
10  output [15:0] dob;
11  reg [15:0] ram [63:0];
12  reg [5:0] read_addr_a;
13  reg [5:0] read_addr_b;
14  always @(posedge clk) begin
15    if (ena)
16      begin
17        if (wea)
18          ram[addr_a] <= dia;
19        read_addr_a <= addr_a;
20      end
21    if (enb)
22      read_addr_b <= addr_b;
23  end
24  assign doa = ram[read_addr_a];
25  assign dob = ram[read_addr_b];
26 endmodule

```

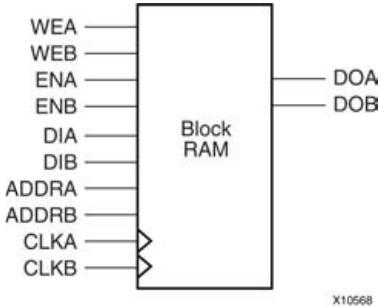
VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_14 is
5   port (clk : in std_logic;
6         ena : in std_logic;
7         enb : in std_logic;
8         wea : in std_logic;
9         addr_a : in std_logic_vector(5 downto 0);
10        addr_b : in std_logic_vector(5 downto 0);
11        dia : in std_logic_vector(15 downto 0);
12        doa : out std_logic_vector(15 downto 0);
13        dob : out std_logic_vector(15 downto 0));
14 end rams_14;
15 architecture syn of rams_14 is
16   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
17   signal RAM : ram_type;
18   signal read_addr_a : std_logic_vector(5 downto 0);
19   signal read_addr_b : std_logic_vector(5 downto 0);
20 begin
21   process (clk)
22   begin
23     if (clk'event and clk = '1') then
24       if (ena = '1') then
25         if (wea = '1') then
26           RAM(conv_integer(addr_a)) <= dia;
27         end if;
28         read_addr_a <= addr_a;
29       end if;
30       if (enb = '1') then
31         read_addr_b <= addr_b;
32       end if;
33     end if;
34   end process;
35   doa <= RAM(conv_integer(read_addr_a));
36   dob <= RAM(conv_integer(read_addr_b));
37 end syn;

```

Dual-Port Block RAM with Two Write Ports



Verilog

```

1 module v_rams_16 (clka,clkb,ena,enb,wea,web,
2                   addra,addrb,dia,dib,doa,dob);
3   input clka,clkb,ena,enb,wea,web;
4   input [5:0] addra,addrb;
5   input [15:0] dia,dib;
6   output [15:0] doa,dob;
7   reg [15:0] ram [63:0];
8   reg [15:0] doa,dob;
9   always @(posedge clka) begin
10     if (ena)
11       begin
12         if (wea)
13           ram[addra] <= dia;
14           doa <= ram[addra];
15       end
16   end
17   always @(posedge clkb) begin
18     if (enb)
19       begin
20         if (web)
21           ram[addrb] <= dib;
22           dob <= ram[addrb];
23       end
24   end
25 endmodule

```

VHDL

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4 entity rams_16 is
5   port(clka : in std_logic;
6        clkb : in std_logic;
7        ena : in std_logic;
8        enb : in std_logic;
9        wea : in std_logic;
10       web : in std_logic;
11       addra : in std_logic_vector(5 downto 0);
12      addrb : in std_logic_vector(5 downto 0);
13      dia : in std_logic_vector(15 downto 0);
14      dib : in std_logic_vector(15 downto 0);
15      doa : out std_logic_vector(15 downto 0);
16      dob : out std_logic_vector(15 downto 0));
17 end rams_16;
18 architecture syn of rams_16 is
19   type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
20   shared variable RAM : ram_type;
21 begin
22   process (CLKA)
23   begin
24     if CLKA'event and CLKA = '1' then
25       if ENA = '1' then
26         if WEA = '1' then
27           RAM(conv_integer(ADDRA)) := DIA;
28         end if;
29         DOA <= RAM(conv_integer(ADDRA));
30       end if;
31     end if;
32   end process;
33   process (CLKB)
34   begin
35     if CLKB'event and CLKB = '1' then
36       if ENB = '1' then
37         if WEB = '1' then
38           RAM(conv_integer(ADDRB)) := DIB;
39         end if;
40         DOB <= RAM(conv_integer(ADDRB));
41       end if;
42     end if;
43   end process;
44 end syn;

```

Multiple Write Statements

Verilog

```

1 reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
2 always @(posedge clk)
3 begin
4     if (we[1])
5         RAM[addr][2*WIDTH-1:WIDTH] <= di[2*WIDTH-1:WIDTH];
6     end if;
7     if (we[0])
8         RAM[addr][WIDTH-1:0] <= di[WIDTH-1:0];
9     end if;
10    dout <= RAM[addr];
11 end

```

VHDL

```

1 type ram_type is array (SIZE-1 downto 0)
2 of std_logic_vector (2*WIDTH-1 downto 0);
3 signal RAM : ram_type;
4 ...
5 process(clk)
6 begin
7     if posedge(clk) then
8         if we(1) = '1' then
9             RAM(conv_integer(addr))(2*WIDTH-1 downto WIDTH) <= di(2*WIDTH-1 downto WIDTH);
10        end if;
11        if we(0) = '1' then
12            RAM(conv_integer(addr))(WIDTH-1 downto 0) <= di(WIDTH-1 downto 0);
13        end if;
14        do <= RAM(conv_integer(addr));
15    end if;
16 end process;

```

Read-First Mode: Single-Port BRAM with Byte-wide Write Enable (2 Bytes)

Verilog

```

1 module v_rams_24 (clk, we, addr, di, dout);
2   parameter SIZE = 512;
3   parameter ADDR_WIDTH = 9;
4   parameter DI_WIDTH = 8;
5   input clk;
6   input [1:0] we;
7   input [ADDR_WIDTH-1:0] addr;
8   input [2*DI_WIDTH-1:0] di;
9   output [2*DI_WIDTH-1:0] dout;
10  reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
11  reg [2*DI_WIDTH-1:0] dout;
12  reg [DI_WIDTH-1:0] di0, di1;
13  always @(we or di)
14  begin
15    if (we[1])
16      di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
17    else
18      di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
19    if (we[0])
20      di0 = di[DI_WIDTH-1:0];
21    else
22      di0 = RAM[addr][DI_WIDTH-1:0];
23  end
24  always @ (posedge clk)
25  begin
26    RAM[addr]<={di1,di0};
27    dout <= RAM[addr];
28  end
29 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_24 is
5   generic (SIZE : integer := 512;
6           ADDR_WIDTH : integer := 9;
7           DI_WIDTH : integer := 8);
8   port (clk : in std_logic;
9         we : in std_logic_vector(1 downto 0);
10        addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
11        di : in std_logic_vector(2*DI_WIDTH-1 downto 0);
12        do : out std_logic_vector(2*DI_WIDTH-1 downto 0));
13 end rams_24;
14 architecture syn of rams_24 is
15   type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
16   signal RAM : ram_type;
17   signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
18 begin
19   process (we, di)
20   begin
21     if we(1) = '1' then
22       di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
23     else
24       di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
25     end if;
26     if we(0) = '1' then
27       di0 <= di(DI_WIDTH-1 downto 0);
28     else
29       di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
30     end if;
31   end process;
32   process (clk)
33   begin
34     if (clk'event and clk = '1') then
35       RAM(conv_integer(addr)) <= di1 & di0;
36       do <= RAM(conv_integer(addr));
37     end if;
38   end process;
39 end syn;

```

Write-First Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes)

Verilog

```

1 module v_rams_25 (clk, we, addr, di, dout);
2   parameter SIZE = 512;
3   parameter ADDR_WIDTH = 9;
4   parameter DI_WIDTH = 8;
5   input clk;
6   input [1:0] we;
7   input [ADDR_WIDTH-1:0] addr;
8   input [2*DI_WIDTH-1:0] di;
9   output [2*DI_WIDTH-1:0] dout;
10  reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
11  reg [2*DI_WIDTH-1:0] dout;
12  reg [DI_WIDTH-1:0] di0, dil;
13  reg [DI_WIDTH-1:0] do0, dol;
14  always @(we or di)
15  begin
16    if (we[1])
17      begin
18        dil = di[2*DI_WIDTH-1:1*DI_WIDTH];
19        dol = di[2*DI_WIDTH-1:1*DI_WIDTH];
20      end
21    else
22      begin
23        dil = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
24        dol = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
25      end
26    if (we[0])
27      begin
28      di0 <= di[DI_WIDTH-1:0];
29      do0 <= di[DI_WIDTH-1:0];
30    end
31    else
32      begin
33      di0 <= RAM[addr][DI_WIDTH-1:0];
34      do0 <= RAM[addr][DI_WIDTH-1:0];
35    end
36  end
37  always @ (posedge clk)
38  begin
39    RAM[addr] <= {dil,di0};
40    dout <= {dol,do0};
41  end
42 endmodule

```

VHDL

```

4  entity rams_25 is
5    generic (SIZE : integer := 512;
6             ADDR_WIDTH : integer := 9;
7             DI_WIDTH : integer := 8);
8    port (clk : in std_logic;
9           we : in std_logic_vector(1 downto 0);
10          addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
11          di : in std_logic_vector(2*DI_WIDTH-1 downto 0);
12          do : out std_logic_vector(2*DI_WIDTH-1 downto 0));
13  end rams_25;
14  architecture syn of rams_25 is
15    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
16    signal RAM : ram_type;
17    signal di0, dil : std_logic_vector (DI_WIDTH-1 downto 0);
18    signal do0, dol : std_logic_vector (DI_WIDTH-1 downto 0);
19  begin
20    process (we, di)
21    begin
22      if we(1) = '1' then
23        dil <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
24        dol <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
25      else
26        dil <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
27        dol <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
28      end if;
29      if we(0) = '1' then
30        di0 <= di(DI_WIDTH-1 downto 0);
31        do0 <= di(DI_WIDTH-1 downto 0);
32      else
33        di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
34        do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
35      end if;
36    end process;
37    process (clk)
38    begin
39      if (clk'event and clk = '1') then
40        RAM(conv_integer(addr)) <= dil & di0;
41        do <= dol & do0;
42      end if;
43    end process;
44  end syn;

```

No-Change Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes)

Verilog

```

1 module v_rams_26 (clk, we, addr, di, dout);
2   parameter SIZE = 512;
3   parameter ADDR_WIDTH = 9;
4   parameter DI_WIDTH = 8;
5   input clk;
6   input [1:0] we;
7   input [ADDR_WIDTH-1:0] addr;
8   input [2*DI_WIDTH-1:0] di;
9   output [2*DI_WIDTH-1:0] dout;
10  reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
11  reg [2*DI_WIDTH-1:0] dout;
12  reg [DI_WIDTH-1:0] di0, di1;
13  reg [DI_WIDTH-1:0] do0, do1;
14  always @(we or di)
15  begin
16    if (we[1])
17      di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
18    else
19      begin
20        di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
21        do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
22      end
23    if (we[0])
24      di0 <= di[DI_WIDTH-1:0];
25    else
26      begin
27        di0 <= RAM[addr][DI_WIDTH-1:0];
28        do0 <= RAM[addr][DI_WIDTH-1:0];
29      end
30  end
31  always @ (posedge clk)
32  begin
33    RAM[addr]<={di1,di0};
34    dout <= {do1,do0};
35  end
36 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_26 is
5   generic (SIZE : integer := 512;
6           ADDR_WIDTH : integer := 9;
7           DI_WIDTH : integer := 8);
8   port (clk : in std_logic;
9         we : in std_logic_vector(1 downto 0);
10        addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
11        di : in std_logic_vector(2*DI_WIDTH-1 downto 0);
12        do : out std_logic_vector(2*DI_WIDTH-1 downto 0));
13 end rams_26;
14 architecture syn of rams_26 is
15   type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
16   signal RAM : ram_type;
17   signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
18   signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
19 begin
20   process (we, di)
21   begin
22     if we(1) = '1' then
23       di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
24     else
25       di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
26       do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
27     end if;
28     if we(0) = '1' then
29       di0 <= di(DI_WIDTH-1 downto 0);
30     else
31       di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
32       do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
33     end if;
34   end process;
35   process (clk)
36   begin
37     if (clk'event and clk = '1') then
38       RAM(conv_integer(addr)) <= di1 & di0;
39       do <= do1 & do0;
40     end if;
41   end process;
42 end syn;

```

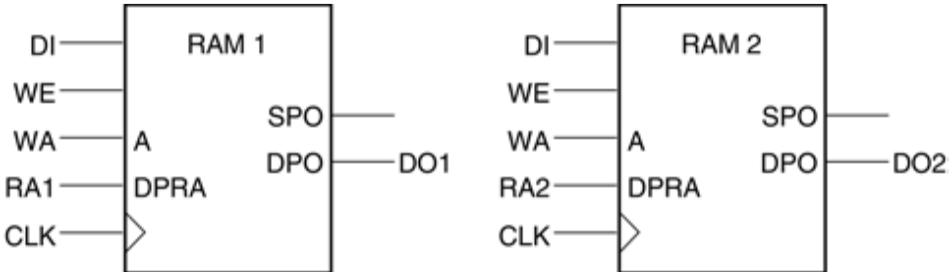
Multiple-Port RAM Descriptions

Verilog

```

1 module v_rams_17 (clk, we, wa, ral, ra2, di, dol, do2);
2   input clk;
3   input we;
4   input [5:0] wa;
5   input [5:0] ral;
6   input [5:0] ra2;
7   input [15:0] di;
8   output [15:0] dol;
9   output [15:0] do2;
10  reg [15:0] ram [63:0];
11  always @(posedge clk)
12 begin
13   if (we)
14     ram[wa] <= di;
15 end
16 assign dol = ram[ral];
17 assign do2 = ram[ra2];
18 endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_17 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         wa : in std_logic_vector(5 downto 0);
8         ral : in std_logic_vector(5 downto 0);
9         ra2 : in std_logic_vector(5 downto 0);
10        di : in std_logic_vector(15 downto 0);
11        dol : out std_logic_vector(15 downto 0);
12        do2 : out std_logic_vector(15 downto 0));
13 end rams_17;
14 architecture syn of rams_17 is
15   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
16   signal RAM : ram_type;
17 begin
18   process (clk)
19   begin
20     if (clk'event and clk = '1') then
21       if (we = '1') then
22         RAM(conv_integer(wa)) <= di;
23       end if;
24     end if;
25   end process;
26   dol <= RAM(conv_integer(ral));
27   do2 <= RAM(conv_integer(ra2));
28 end syn;

```

Block RAM with Reset Pin

Verilog

```

1 module v_rams_18 (clk, en, we, rst, addr, di, dout);
2   input clk;
3   input en;
4   input we;
5   input rst;
6   input [5:0] addr;
7   input [15:0] di;
8   output [15:0] dout;
9   reg [15:0] ram [63:0];
10  reg [15:0] dout;
11  always @(posedge clk)
12 begin
13   if (en) // optional enable
14     begin
15       if (we) // write enable
16         ram[addr] <= di;
17       if (rst) // optional reset
18         dout <= 16'h0000;
19       else
20         dout <= ram[addr];
21     end
22   end
23 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_18 is
5   port (clk : in std_logic;
6         en : in std_logic;
7         we : in std_logic;
8         rst : in std_logic;
9         addr : in std_logic_vector(5 downto 0);
10        di : in std_logic_vector(15 downto 0);
11        do : out std_logic_vector(15 downto 0));
12 end rams_18;
13 architecture syn of rams_18 is
14   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
15   signal ram : ram_type;
16 begin
17   process (clk)
18   begin
19     if clk'event and clk = '1' then
20       if en = '1' then -- optional enable
21         if we = '1' then -- write enable
22           ram(conv_integer(addr)) <= di;
23         end if;
24         if rst = '1' then -- optional reset
25           do <= (others => '0');
26         else
27           do <= ram(conv_integer(addr)) ;
28         end if;
29       end if;
30     end if;
31   end process;
32 end syn;

```

Block RAM with Optional Output Registers

Verilog

```

1 module v_rams_19 (clk1, clk2, we, en1, en2,
2   addr1, addr2, di, res1, res2);
3   input clk1;
4   input clk2;
5   input we, en1, en2;
6   input [5:0] addr1;
7   input [5:0] addr2;
8   input [15:0] di;
9   output [15:0] res1;
10  output [15:0] res2;
11  reg [15:0] res1;
12  reg [15:0] res2;
13  reg [15:0] RAM [63:0];
14  reg [15:0] do1;
15  reg [15:0] do2;
16  always @(posedge clk1)
17 begin
18   if (we == 1'b1)
19     RAM[addr1] <= di;
20   do1 <= RAM[addr1];
21 end
22 always @(posedge clk2)
23 begin
24   do2 <= RAM[addr2];
25 end
26 always @(posedge clk1)
27 begin
28   if (en1 == 1'b1)
29     res1 <= do1;
30 end
31 always @(posedge clk2)
32 begin
33   if (en2 == 1'b1)
34     res2 <= do2;
35 end
36 endmodule

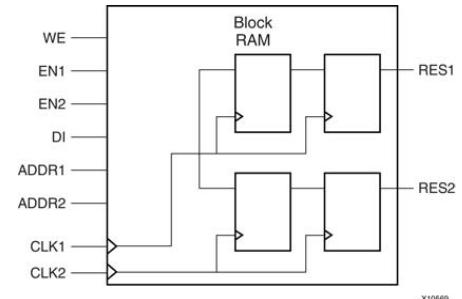
```

VHDL

```

1 library IEEE;
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5 entity rams_19 is
6   port (clk1, clk2 : in std_logic;
7         we, en1, en2 : in std_logic;
8         addr1 : in std_logic_vector(5 downto 0);
9         addr2 : in std_logic_vector(5 downto 0);
10        di : in std_logic_vector(15 downto 0);
11        res1 : out std_logic_vector(15 downto 0);
12        res2 : out std_logic_vector(15 downto 0));
13 end rams_19;
14 architecture beh of rams_19 is
15   type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
16   signal ram : ram_type;
17   signal do1 : std_logic_vector(15 downto 0);
18   signal do2 : std_logic_vector(15 downto 0);
19 begin
20   process (clk1)
21   begin
22     if rising_edge(clk1) then
23       if we = '1' then
24         ram(conv_integer(addr1)) <= di;
25       end if;
26       do1 <= ram(conv_integer(addr1));
27     end if;
28   end process;
29   process (clk2)
30   begin
31     if rising_edge(clk2) then
32       do2 <= ram(conv_integer(addr2));
33     end if;
34   end process;
35   process (clk1)
36   begin
37     if rising_edge(clk1) then
38       if en1 = '1' then
39         res1 <= do1;
40       end if;
41     end if;
42   end process;
43   process (clk2)
44   begin
45     if rising_edge(clk2) then
46       if en2 = '1' then
47         res2 <= do2;
48       end if;
49     end if;
50   end process;
51 end beh;

```



Initializing RAM Directly in HDL Code

Verilog

```

1 // Binary
2 reg [15:0] ram [63:0];
3 initial begin
4     ram[63] = 16'b0111100100000101;
5     ram[62] = 16'b0000010110111101;
6     ram[61] = 16'b1100001101010000;
7     //...
8     ram[0] = 16'b0000100101110011;
9 end
10
11 // Hexadecimal
12 reg [19:0] ram [63:0];
13 initial begin
14     ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
15     ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
16     //...
17     ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
18 end
19 //...
20 always @(posedge clk)
21 begin
22     if (we)
23         ram[addr] <= di;
24         dout <= ram[addr];
25 end

```

VHDL

```

1 ...
2 type ram_type is array (0 to 63) of std_logic_vector(19 downto 0);
3 signal RAM : ram_type :=
4 (
5     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
6     X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
7     X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
8     X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
9     X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
10    X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
11    X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
12    X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
13    X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
14    X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
15    X"0030D", X"02341", X"08201", X"0400D");
16 ...
17 process (clk)
18 begin
19     if rising_edge(clk) then
20         if we = '1' then
21             RAM(conv_integer(a)) <= di;
22         end if;
23         ra <= a;
24     end if;
25 end process;
26 ...
27 do <= RAM(conv_integer(ra));

```

Initializing RAM Directly in HDL Code

Dual Port Block RAM Initialization in Verilog

```
1  module v_rams_20b (clk1, clk2, we, addr1, addr2, di, do1, do2);
2      input clk1, clk2;
3      input we;
4      input [7:0] addr1, addr2;
5      input [15:0] di;
6      output [15:0] do1, do2;
7      reg [15:0] ram [255:0];
8      reg [15:0] do1, do2;
9      integer index;
10     initial begin
11         for (index = 0 ; index <= 99 ; index = index + 1) begin
12             ram[index] = 16'h8282;
13         end
14         for (index= 100 ; index <= 255 ; index = index + 1) begin
15             ram[index] = 16'hB8B8;
16         end
17     end
18     always @ (posedge clk1)
19     begin
20         if (we)
21             ram[addr1] <= di;
22         do1 <= ram[addr1];
23     end
24     always @ (posedge clk2)
25     begin
26         do2 <= ram[addr2];
27     end
28 endmodule
```

Initializing RAM from an External File in Verilog

Verilog

```

1 module v_rams_20c (clk, we, addr, din, dout);
2   input clk;
3   input we;
4   input [5:0] addr;
5   input [31:0] din;
6   output [31:0] dout;
7   reg [31:0] ram [0:63];
8   reg [31:0] dout;
9   initial
10 begin
11   $readmem("rams_20c.data",ram, 0, 63);
12 end
13 always @ (posedge clk)
14 begin
15   if (we)
16     ram[addr] <= din;
17     dout <= ram[addr];
18 end
19 endmodule

```

Hexadecimal/Binary in text format

The image shows two Notepad++ windows side-by-side. The left window, titled 'rams_20c.data', contains a list of 17 hexadecimal values representing memory contents. The right window, titled 'rams_8x32.data', contains a list of 8 binary values representing memory contents.

1	8e62a8e6
2	df759df7
3	9d0099d0
4	41abf41a
5	b7651b76
6	af47caf4
7	2dd972dd
8	99ad399a
9	d72d7d72
10	610a1610
11	aaf87aaaf
12	a0446a04
13	06704067
14	24114241
15	b64c4b64
16	91ec791e
17	cab85cab
18	3dc163dc
19	80756007

1	00001111000011110000111100001111
2	01001010001000001100000010000100
3	000000000011111000000000001000001
4	11111101010000011100010000100100
5	00001111000011110000111100001111
6	01001010001000001100000010000100
7	000000000011111000000000001000001
8	11111101010000011100010000100100

Initializing RAM from an External File in VHDL

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use std.textio.all;
5 entity rams_20c is
6     port(clk : in std_logic;
7          we : in std_logic;
8          addr : in std_logic_vector(5 downto 0);
9          din : in std_logic_vector(31 downto 0);
10         dout : out std_logic_vector(31 downto 0));
11 end rams_20c;
12 architecture syn of rams_20c is
13     type RamType is array(0 to 63) of bit_vector(31 downto 0);
14     impure function InitRamFromFile (RamFileName : in string) return RamType is
15         FILE RamFile : text is in RamFileName;
16         variable RamFileLine : line;
17         variable RAM : RamType;
18     begin
19         for I in RamType'range loop
20             readline (RamFile, RamFileLine);
21             read (RamFileLine, RAM(I));
22         end loop;
23         return RAM;
24     end function;
25     signal RAM : RamType := InitRamFromFile("rams_20c.data");
26 begin
27     process (clk)
28     begin
29         if clk'event and clk = '1' then
30             if we = '1' then
31                 RAM(conv_integer(addr)) <= to_bitvector(din);
32             end if;
33             dout <= to_stdlogicvector(RAM(conv_integer(addr)));
34         end if;
35     end process;
36 end syn;
```

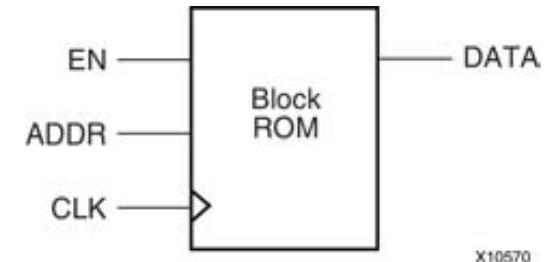
ROM with Registered Output, Example 1

Verilog

```

1 module v_rams_21a (clk, en, addr, data);
2   input clk;
3   input en;
4   input [5:0] addr;
5   output reg [19:0] data;
6   always @ (posedge clk) begin
7     if (en)
8       case(addr)
9         6'b000000: data <= 20'h0200A; 6'b100000: data <= 20'h02222;
10        6'b000001: data <= 20'h00300; 6'b100001: data <= 20'h04001;
11        6'b000010: data <= 20'h08101; 6'b100010: data <= 20'h00342;
12        6'b000011: data <= 20'h04000; 6'b100011: data <= 20'h0232B;
13        6'b000100: data <= 20'h08601; 6'b100100: data <= 20'h00900;
14        6'b000101: data <= 20'h0233A; 6'b100101: data <= 20'h0302;
15        6'b000110: data <= 20'h00300; 6'b100110: data <= 20'h00102;
16        6'b000111: data <= 20'h08602; 6'b100111: data <= 20'h04002;
17        6'b001000: data <= 20'h02310; 6'b101000: data <= 20'h00900;
18        6'b001001: data <= 20'h0203B; 6'b101001: data <= 20'h08201;
19        6'b001010: data <= 20'h08300; 6'b101010: data <= 20'h02023;
20        6'b001011: data <= 20'h04002; 6'b101011: data <= 20'h00303;
21        6'b001100: data <= 20'h08201; 6'b101100: data <= 20'h02433;
22        6'b001101: data <= 20'h00500; 6'b101101: data <= 20'h00301;
23        6'b001110: data <= 20'h04001; 6'b101110: data <= 20'h04004;
24        6'b01111: data <= 20'h02500; 6'b101111: data <= 20'h00301;
25        6'b010000: data <= 20'h00340; 6'b110000: data <= 20'h00102;
26        6'b010001: data <= 20'h00241; 6'b110001: data <= 20'h02137;
27        6'b010010: data <= 20'h04002; 6'b110010: data <= 20'h02036;
28        6'b010011: data <= 20'h08300; 6'b110011: data <= 20'h00301;
29        6'b010100: data <= 20'h08201; 6'b110100: data <= 20'h00102;
30        6'b010101: data <= 20'h00500; 6'b110101: data <= 20'h02237;
31        6'b010110: data <= 20'h08101; 6'b110110: data <= 20'h04004;
32        6'b010111: data <= 20'h00602; 6'b110111: data <= 20'h00304;
33        6'b011000: data <= 20'h04003; 6'b111000: data <= 20'h04040;
34        6'b011001: data <= 20'h0241E; 6'b111001: data <= 20'h02500;
35        6'b011010: data <= 20'h00301; 6'b111010: data <= 20'h02500;
36        6'b011011: data <= 20'h00102; 6'b111011: data <= 20'h02500;
37        6'b011100: data <= 20'h02122; 6'b111100: data <= 20'h0030D;
38        6'b011101: data <= 20'h02021; 6'b111101: data <= 20'h02341;
39        6'b011110: data <= 20'h00301; 6'b111110: data <= 20'h08201;
40        6'b011111: data <= 20'h00102; 6'b111111: data <= 20'h0400D;
41      endcase
42    end
43  endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_21a is
5   port (clk : in std_logic;
6         en : in std_logic;
7         addr : in std_logic_vector(5 downto 0);
8         data : out std_logic_vector(19 downto 0));
9 end rams_21a;
10 architecture syn of rams_21a is
11   type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
12   signal ROM : rom_type:= (
13     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
14     X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
15     X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
16     X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
17     X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
18     X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
19     X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
20     X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
21     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
22     X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
23     X"0030D", X"02341", X"08201", X"0400D");
24 begin
25   process (clk)
26     begin
27       if (clk'event and clk = '1') then
28         if (en = '1') then
29           data <= ROM(conv_integer(addr));
30         end if;
31       end if;
32     end process;
33   end syn;

```

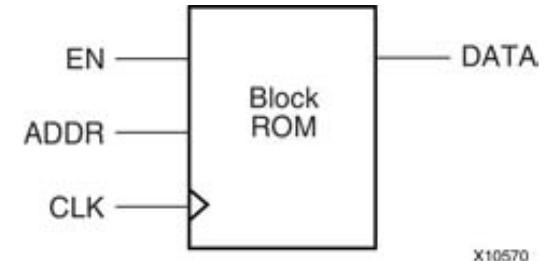
ROM with Registered Output, Example 2

Verilog

```

1 module v_rams_2lb (clk, en, addr, data);
2   input clk;
3   input en;
4   input [5:0] addr;
5   output reg [19:0] data;
6   reg [19:0] rdata;
7   always @(addr) begin
8     case(addr)
9       6'b000000: rdata <= 20'h0200A; 6'b100000: rdata <= 20'h02222;
10      6'b000001: rdata <= 20'h00300; 6'b100001: rdata <= 20'h04001;
11      6'b000010: rdata <= 20'h08101; 6'b100010: rdata <= 20'h00342;
12      6'b000011: rdata <= 20'h04000; 6'b100011: rdata <= 20'h0232B;
13      6'b000100: rdata <= 20'h08601; 6'b100100: rdata <= 20'h00900;
14      6'b000101: rdata <= 20'h0233A; 6'b100101: rdata <= 20'h00302;
15      6'b000110: rdata <= 20'h00300; 6'b100110: rdata <= 20'h00102;
16      6'b000111: rdata <= 20'h08602; 6'b100111: rdata <= 20'h04002;
17      6'b001000: rdata <= 20'h02310; 6'b101000: rdata <= 20'h00900;
18      6'b001001: rdata <= 20'h0203B; 6'b101001: rdata <= 20'h08201;
19      6'b001010: rdata <= 20'h08300; 6'b101010: rdata <= 20'h02023;
20      6'b001011: rdata <= 20'h04002; 6'b101011: rdata <= 20'h00303;
21      6'b001100: rdata <= 20'h08201; 6'b101100: rdata <= 20'h02433;
22      6'b001101: rdata <= 20'h00500; 6'b101101: rdata <= 20'h00301;
23      6'b001110: rdata <= 20'h04001; 6'b101110: rdata <= 20'h04004;
24      6'b001111: rdata <= 20'h02500; 6'b101111: rdata <= 20'h00301;
25      6'b010000: rdata <= 20'h00340; 6'b110000: rdata <= 20'h00102;
26      6'b010001: rdata <= 20'h00241; 6'b110001: rdata <= 20'h02137;
27      6'b010010: rdata <= 20'h04002; 6'b110010: rdata <= 20'h02036;
28      6'b010011: rdata <= 20'h08300; 6'b110011: rdata <= 20'h00301;
29      6'b010100: rdata <= 20'h08201; 6'b110100: rdata <= 20'h00102;
30      6'b010101: rdata <= 20'h00500; 6'b110101: rdata <= 20'h02237;
31      6'b010110: rdata <= 20'h08101; 6'b110110: rdata <= 20'h04004;
32      6'b010111: rdata <= 20'h00602; 6'b110111: rdata <= 20'h00304;
33      6'b011000: rdata <= 20'h04003; 6'b111000: rdata <= 20'h04040;
34      6'b011001: rdata <= 20'h0241E; 6'b111001: rdata <= 20'h02500;
35      6'b011010: rdata <= 20'h00301; 6'b111010: rdata <= 20'h02500;
36      6'b011011: rdata <= 20'h00102; 6'b111011: rdata <= 20'h02500;
37      6'b011100: rdata <= 20'h02122; 6'b111100: rdata <= 20'h0030D;
38      6'b011101: rdata <= 20'h02021; 6'b111101: rdata <= 20'h02341;
39      6'b011110: rdata <= 20'h00301; 6'b111110: rdata <= 20'h08201;
40      6'b011111: rdata <= 20'h00102; 6'b111111: rdata <= 20'h0400D;
41    endcase
42  end
43  always @(posedge clk) begin
44    if (en)
45      data <= rdata;
46  end
47 endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_2lb is
5   port (clk : in std_logic;
6         en : in std_logic;
7         addr : in std_logic_vector(5 downto 0);
8         data : out std_logic_vector(19 downto 0));
9 end rams_2lb;
10 architecture syn of rams_2lb is
11   type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
12   signal ROM : rom_type:= (
13     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
14     X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
15     X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
16     X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
17     X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
18     X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
19     X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
20     X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
21     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
22     X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
23     X"0030D", X"02341", X"08201", X"0400D");
24   signal rdata : std_logic_vector(19 downto 0);
25 begin
26   rdata <= ROM(conv_integer(addr));
27   process (clk)
28   begin
29     if (clk'event and clk = '1') then
30       if (en = '1') then
31         data <= rdata;
32       end if;
33     end if;
34   end process;
35 end syn;

```

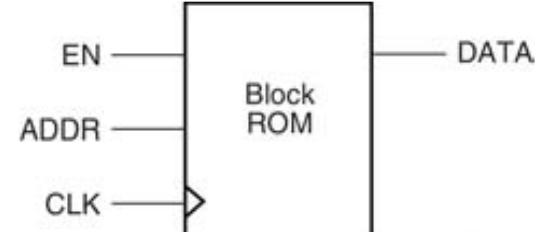
ROM with Registered Address

Verilog

```

1 module v_rams_2lc (clk, en, addr, data);
2   input clk;
3   input en;
4   input [5:0] addr;
5   output reg [19:0] data;
6   reg [5:0] raddr;
7   always @(posedge clk) begin
8     if (en)
9       raddr <= addr;
10    end
11  always @(raddr) begin
12    case(raddr)
13      6'b000000: data <= 20'h0200A; 6'b100000: data <= 20'h02222;
14      6'b000001: data <= 20'h00300; 6'b100001: data <= 20'h04001;
15      6'b000010: data <= 20'h08101; 6'b100010: data <= 20'h00342;
16      6'b000011: data <= 20'h04000; 6'b100011: data <= 20'h0232B;
17      6'b000100: data <= 20'h08601; 6'b100100: data <= 20'h00900;
18      6'b000101: data <= 20'h0233A; 6'b100101: data <= 20'h00302;
19      6'b000110: data <= 20'h00300; 6'b100110: data <= 20'h00102;
20      6'b000111: data <= 20'h08602; 6'b100111: data <= 20'h04002;
21      6'b001000: data <= 20'h02310; 6'b101000: data <= 20'h00900;
22      6'b001001: data <= 20'h0203B; 6'b101001: data <= 20'h08201;
23      6'b001010: data <= 20'h08300; 6'b101010: data <= 20'h02023;
24      6'b001011: data <= 20'h04002; 6'b101011: data <= 20'h00303;
25      6'b001100: data <= 20'h08201; 6'b101100: data <= 20'h02433;
26      6'b001101: data <= 20'h00500; 6'b101101: data <= 20'h00301;
27      6'b001110: data <= 20'h04001; 6'b101110: data <= 20'h04004;
28      6'b001111: data <= 20'h02500; 6'b101111: data <= 20'h00301;
29      6'b010000: data <= 20'h00340; 6'b110000: data <= 20'h00102;
30      6'b010001: data <= 20'h00241; 6'b110001: data <= 20'h02137;
31      6'b010010: data <= 20'h04002; 6'b110010: data <= 20'h02036;
32      6'b010011: data <= 20'h08300; 6'b110011: data <= 20'h00301;
33      6'b010100: data <= 20'h08201; 6'b110100: data <= 20'h00102;
34      6'b010101: data <= 20'h00500; 6'b110101: data <= 20'h02237;
35      6'b010110: data <= 20'h08101; 6'b110110: data <= 20'h04004;
36      6'b010111: data <= 20'h00602; 6'b110111: data <= 20'h00304;
37      6'b011000: data <= 20'h04003; 6'b111000: data <= 20'h04040;
38      6'b011001: data <= 20'h0241E; 6'b111001: data <= 20'h02500;
39      6'b011010: data <= 20'h00301; 6'b111010: data <= 20'h02500;
40      6'b011011: data <= 20'h00102; 6'b111011: data <= 20'h02500;
41      6'b011100: data <= 20'h02122; 6'b111100: data <= 20'h0030D;
42      6'b011101: data <= 20'h02021; 6'b111101: data <= 20'h02341;
43      6'b011110: data <= 20'h00301; 6'b111110: data <= 20'h08201;
44      6'b011111: data <= 20'h00102; 6'b111111: data <= 20'h0400D;
45    endcase
46  end
47 endmodule

```



VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_2lc is
5   port (clk : in std_logic;
6         en : in std_logic;
7         addr : in std_logic_vector(5 downto 0);
8         data : out std_logic_vector(19 downto 0));
9 end rams_2lc;
10 architecture syn of rams_2lc is
11   type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
12   signal ROM : rom_type:= (
13     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
14     X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
15     X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
16     X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
17     X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
18     X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
19     X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
20     X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
21     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
22     X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
23     X"0030D", X"02341", X"08201", X"0400D");
24   signal raddr : std_logic_vector(5 downto 0);
25 begin
26   process (clk)
27   begin
28     if (clk'event and clk = '1') then
29       if (en = '1') then
30         raddr <= addr;
31       end if;
32     end if;
33   end process;
34   data <= ROM(conv_integer(raddr));
35 end syn;

```

X10571

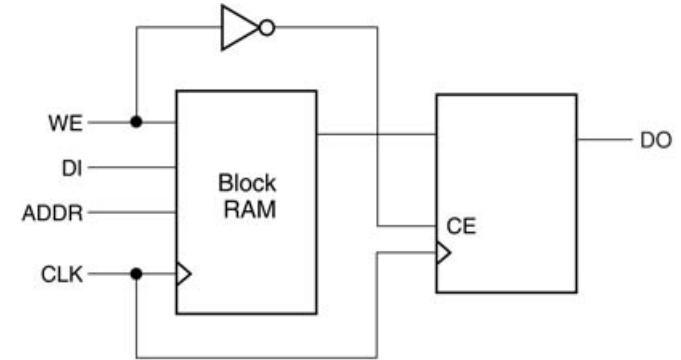
Pipelined Distributed RAM

Verilog

```

1 module v_rams_22 (clk, we, addr, di, dout);
2   input clk;
3   input we;
4   input [8:0] addr;
5   input [3:0] di;
6   output [3:0] dout;
7   (*ram_style="pipe_distributed"*)
8   reg [3:0] RAM [511:0];
9   reg [3:0] dout;
10  reg [3:0] pipe_reg;
11  always @(posedge clk) begin
12    if (we)
13      RAM[addr] <= di;
14    else
15      pipe_reg <= RAM[addr];
16    dout <= pipe_reg;
17  end
18 endmodule

```



VHDL

```

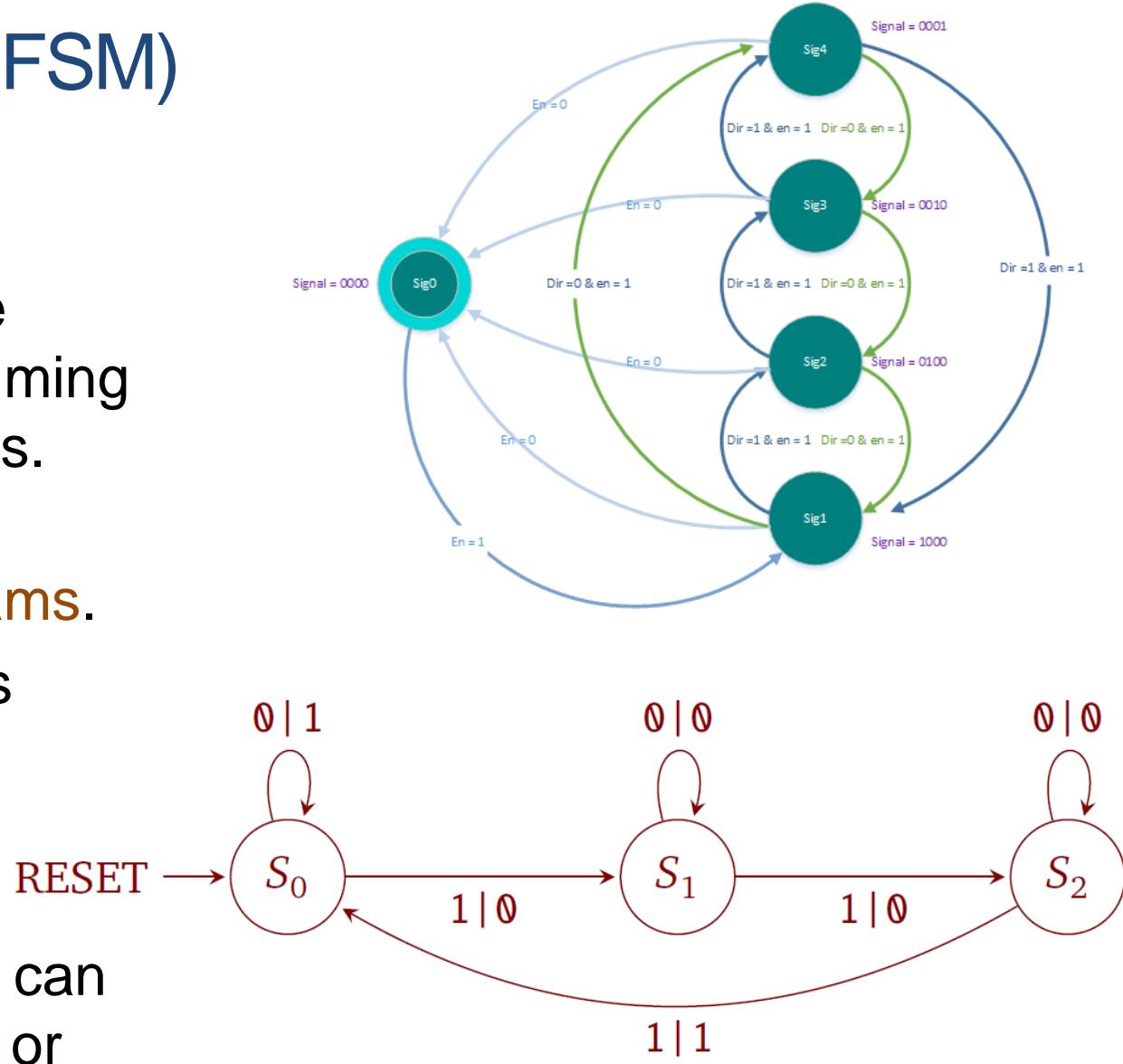
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 entity rams_22 is
5   port (clk : in std_logic;
6         we : in std_logic;
7         addr : in std_logic_vector(8 downto 0);
8         di : in std_logic_vector(3 downto 0);
9         do : out std_logic_vector(3 downto 0));
10 end rams_22;
11 architecture syn of rams_22 is
12   type ram_type is array (511 downto 0) of std_logic_vector (3 downto 0);
13   signal RAM : ram_type;
14   signal pipe_reg: std_logic_vector(3 downto 0);
15   attribute ram_style: string;
16 attribute ram_style of RAM: signal is "pipe_distributed";
17 begin
18   process (clk)
19   begin
20     if clk'event and clk = '1' then
21       if we = '1' then
22         RAM(conv_integer(addr)) <= di;
23       else
24         pipe_reg <= RAM( conv_integer(addr));
25       end if;
26       do <= pipe_reg;
27     end if;
28   end process;
29 end syn;

```

X10572

Finite State Machines (FSM)

- Finite state machines (automata) are used as the backbone controllers and timing managers of digital systems.
- FSMs can be graphically illustrated by **bubble diagrams**.
- Flawless design of FSMs is critical for a proper system function.
- FSMs with dead-ends or erroneous state-transitions can result in **hardware hanging** or malfunctions.



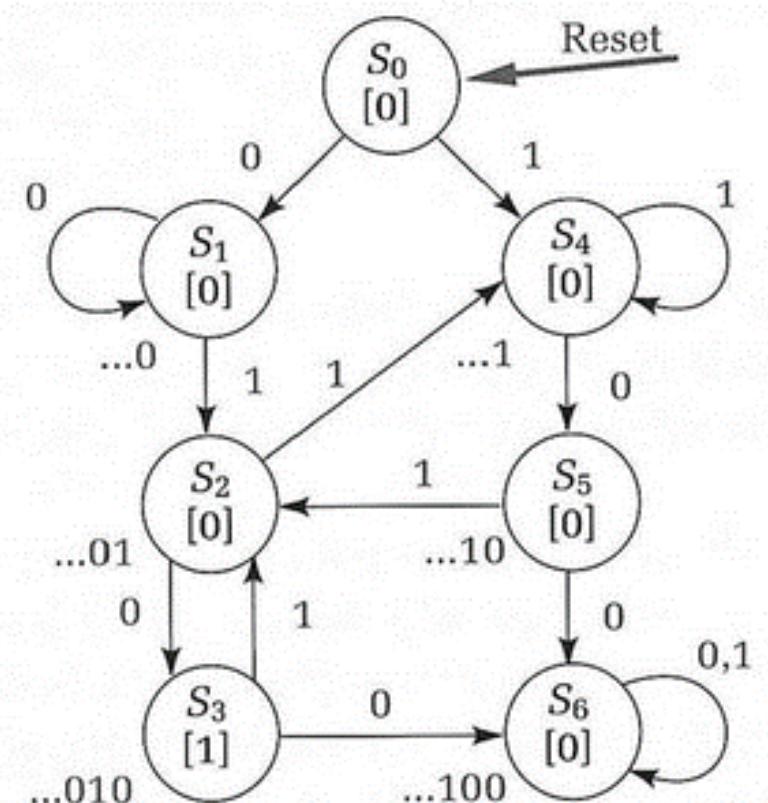
Finite State Machines Components

An FSM consists of:

- Inputs
- Outputs
- States and state sequences
- State transition rules
- Initial conditions (states)
- Resetting mechanism

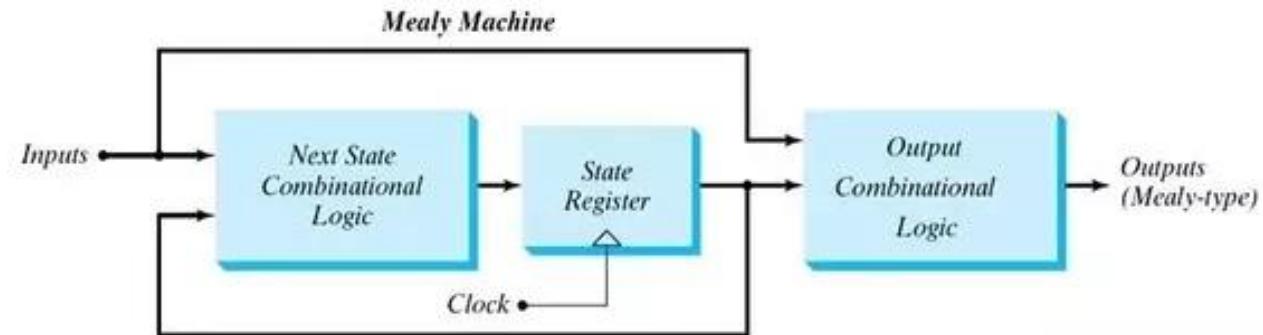
FSM Implementation issues:

- State encoding technique
- Robust FSM design and state recovery mechanism

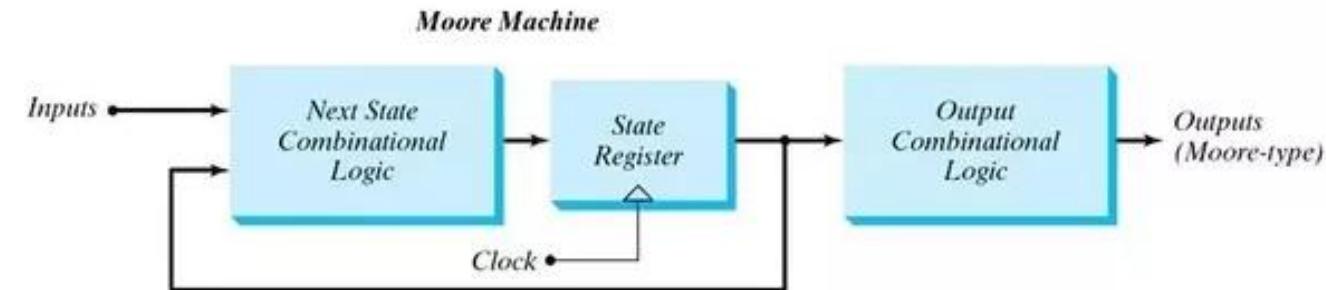


Finite State Machine (FSM) Representation Incorporating Mealy and Moore Machines

$$\text{Mealy Machine: } \begin{cases} s_{k+1} = f(s_k, x_k) \\ y_k = g(s_k, x_k) \end{cases}$$



$$\text{Moore Machine: } \begin{cases} s_{k+1} = f(s_k, x_k) \\ y_k = g(s_k) \end{cases}$$



Research Topic: According to the above representation, Mealy and Moore machines can be studied from a state-space perspective. The rich literature of state-space analysis from **Control Theory** can be used to study the properties of logic circuits.

FSM Encoding Techniques

HDL synthesis tools support various FSM encoding techniques including:

- One-Hot
- Gray
- Compact
- Johnson
- Sequential
- Speed1
- User Defined
- Auto Encoding

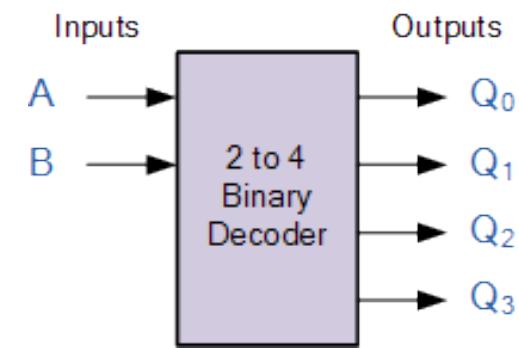
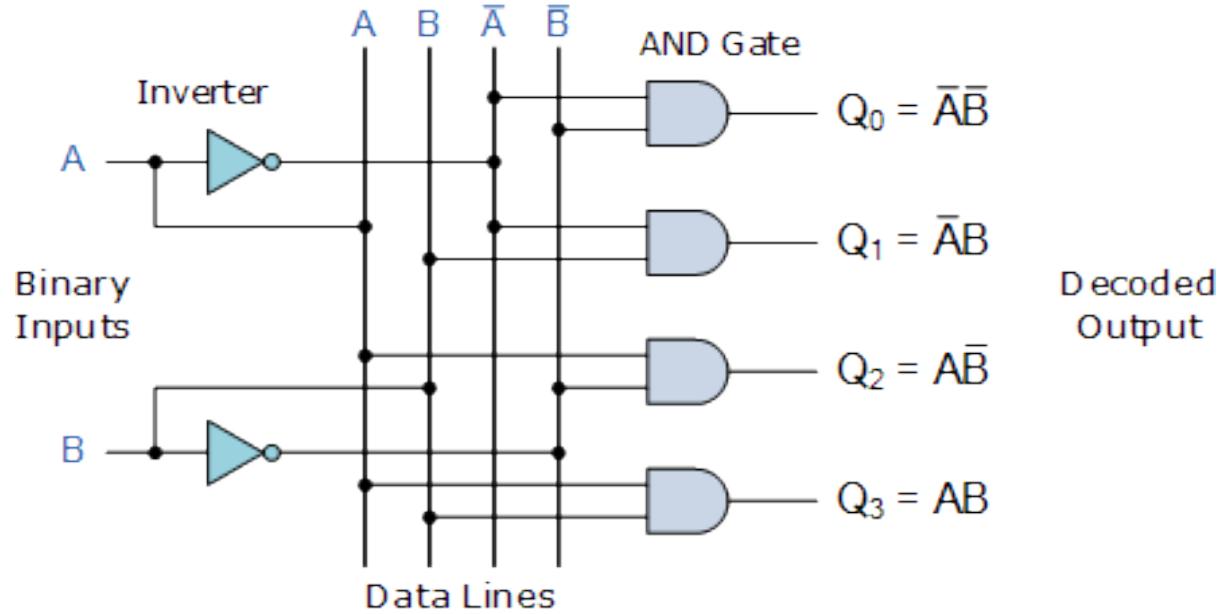
#	Binary	Gray	Johnson	One-hot
0	000	000	0000	00000001
1	001	001	0001	00000010
2	010	011	0011	00000100
3	011	010	0111	00001000
4	100	110	1111	00010000
5	101	111	1110	00100000
6	110	101	1100	01000000
7	111	100	1000	10000000

Same number of bits as binary

Two adjacent codes only differ by one bit

Sample FSM Encoding Logic

The One-Hot Encoder



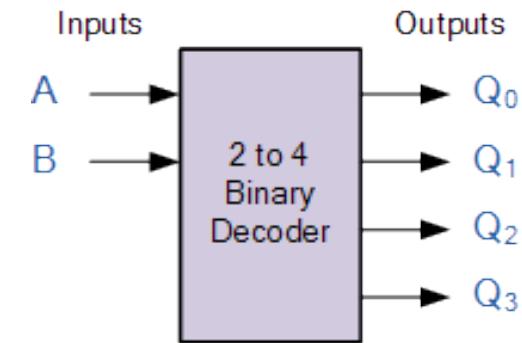
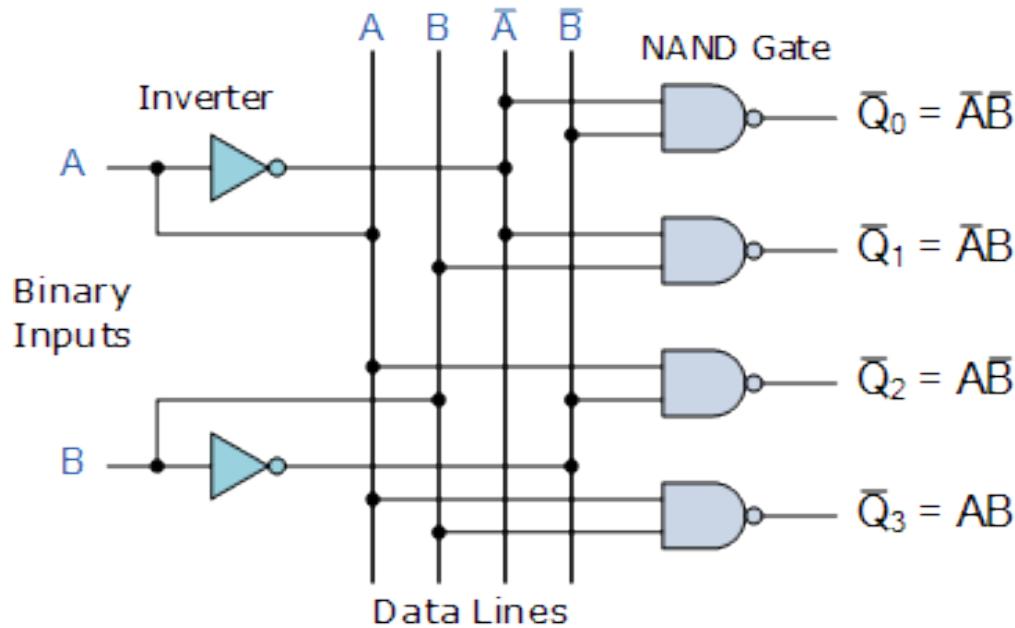
Decoded Output

Truth Table

A	B	Q_0	Q_1	Q_2	Q_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Sample FSM Encoding Logic

The One-Cold Encoder



		A	B	Q_0	Q_1	Q_2	Q_3
Truth	Table	0	0	1	0	0	0
		0	1	0	1	0	0
		1	0	0	1	1	0
		1	1	0	0	1	1

FSM Implementation Issues

- State encoding methods defer in speed, area and robustness to state transition errors.
- On FPGA, FSMs are commonly implemented using BRAM or LUT
- Synthesis tools such as XST can add logic to an FSM implementation that will let the state machine recover from an invalid state. During run-time, if an FSM enters an invalid state, this extra logic will take it back to a known state, called a **recovery state** (the ‘reset state’ by default). This is called a **Safe FSM Implementation**

Finite State Machine Coding Example: A Single Process

Verilog

```

1 module v_fsm_1 (clk, reset, xl, outp);
2   input clk, reset, xl;
3   output outp;
4   reg outp;
5   reg [1:0] state;
6   parameter s1 = 2'b00; parameter s2 = 2'b01;
7   parameter s3 = 2'b10; parameter s4 = 2'b11;
8   initial state = 2'b00;
9   always@(posedge clk or posedge reset)
10 begin
11   if (reset) begin
12     state <= s1; outp <= 1'b1;
13   end
14   else begin
15     case (state)
16       s1: begin
17         if (xl==1'b1) begin
18           state <= s2; outp <= 1'b1;
19         end
20         else begin
21           state <= s3; outp <= 1'b0;
22         end
23       end
24       s2: begin
25         state <= s4; outp <= 1'b1;
26       end
27       s3: begin
28         state <= s4; outp <= 1'b0;
29       end
30       s4: begin
31         state <= s1; outp <= 1'b0;
32       end
33     endcase
34   end
35 end
36 endmodule

```

VHDL

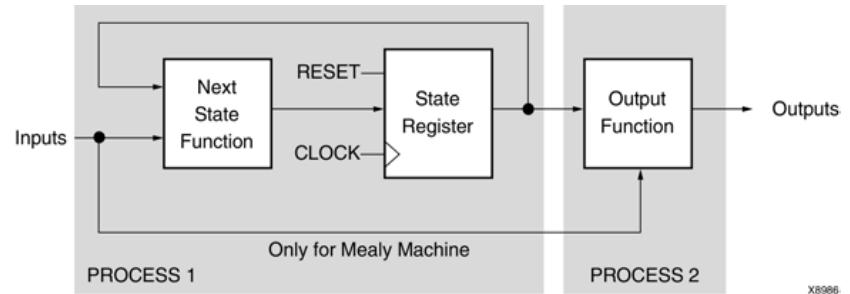
```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 entity fsm_1 is
4   port ( clk, reset, xl : IN std_logic;
5         outp : OUT std_logic);
6 end entity;
7 architecture beh1 of fsm_1 is
8   type state_type is (s1,s2,s3,s4);
9   signal state: state_type ;
10 begin
11   process (clk,reset)
12   begin
13     if (reset ='1') then
14       state <=s1;
15       outp<='1';
16     elsif (clk='1' and clk'event) then
17       case state is
18         when s1 => if xl='1' then
19           state <= s2;
20           outp <= '1';
21         else
22           state <= s3;
23           outp <= '0';
24         end if;
25         when s2 => state <= s4; outp <= '0';
26         when s3 => state <= s4; outp <= '0';
27         when s4 => state <= s1; outp <= '1';
28       end case;
29     end if;
30   end process;
31 end beh1;

```

Finite State Machine Coding

Example: Two Processes



Verilog

```

1 module v_fsm_2 (clk, reset, x1, outp);
2   input clk, reset, x1;
3   output outp;
4   reg outp;
5   reg [1:0] state;
6   parameter s1 = 2'b00; parameter s2 = 2'b01;
7   parameter s3 = 2'b10; parameter s4 = 2'b11;
8   initial state = 2'b00;
9   always @(posedge clk or posedge reset)
10 begin
11   if (reset)
12     state <= s1;
13   else
14     begin
15       case (state)
16         s1: if (x1==1'b1)
17           state <= s2;
18         else
19           state <= s3;
20         s2: state <= s4;
21         s3: state <= s4;
22         s4: state <= s1;
23       endcase
24     end
25   end
26   always @ (state)
27   begin
28     case (state)
29       s1: outp = 1'b1;
30       s2: outp = 1'b1;
31       s3: outp = 1'b0;
32       s4: outp = 1'b0;
33     endcase
34   end
35 endmodule

```

VHDL

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 entity fsm_2 is
4   port ( clk, reset, x1 : IN std_logic;
5         outp : OUT std_logic);
6 end entity;
7 architecture beh1 of fsm_2 is
8   type state_type is (s1,s2,s3,s4);
9   signal state: state_type ;
10 begin
11   process1: process (clk,reset)
12   begin
13     if (reset = '1') then state <=s1;
14     elsif (clk='1' and clk'Event) then
15       case state is
16         when s1 => if x1='1' then
17           state <= s2;
18         else
19           state <= s3;
20         end if;
21         when s2 => state <= s4;
22         when s3 => state <= s4;
23         when s4 => state <= s1;
24       end case;
25     end if;
26   end process process1;
27   process2 : process (state)
28   begin
29     case state is
30       when s1 => outp <= '1';
31       when s2 => outp <= '1';
32       when s3 => outp <= '0';
33       when s4 => outp <= '0';
34     end case;
35   end process process2;
36 end beh1;

```

Finite State Machine Coding

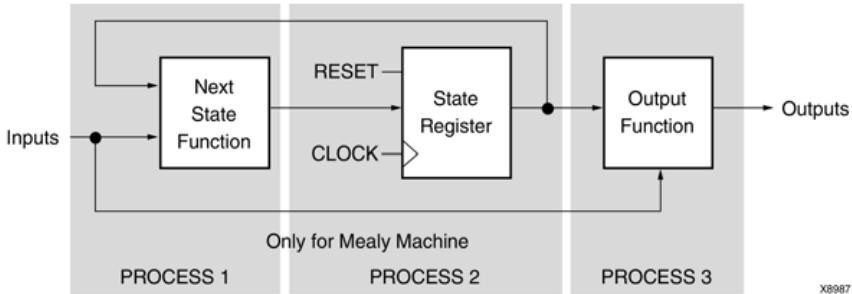
Example: Three Processes

Verilog

```

1 module v_fsm_3 (clk, reset, x1, outp);
2   input clk, reset, x1;
3   output outp;
4   reg outp;
5   reg [1:0] state;
6   reg [1:0] next_state;
7   parameter s1 = 2'b00; parameter s2 = 2'b01;
8   parameter s3 = 2'b10; parameter s4 = 2'b11;
9   initial state = 2'b00;
10  always @(posedge clk or posedge reset)
11 begin
12    if (reset) state <= s1;
13    else state <= next_state;
14 end
15 always @(state or x1)
16 begin
17   case (state)
18     s1: if (x1==1'b1)
19       next_state = s2;
20     else
21       next_state = s3;
22     s2: next_state = s4;
23     s3: next_state = s4;
24     s4: next_state = s1;
25   endcase
26 end
27 always @ (state)
28 begin
29   case (state)
30     s1: outp = 1'b1;
31     s2: outp = 1'b1;
32     s3: outp = 1'b0;
33     s4: outp = 1'b0;
34   endcase
35 end
36 endmodule

```



VHDL

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 entity fsm_3 is
4   port ( clk, reset, x1 : IN std_logic;
5         outp : OUT std_logic );
6 end entity;
7 architecture beh1 of fsm_3 is
8   type state_type is (s1,s2,s3,s4);
9   signal state, next_state: state_type ;
10 begin
11   process1: process (clk,reset)
12   begin
13     if (reset='1') then
14       state <=s1;
15     elsif (clk='1' and clk'Event) then
16       state <= next_state;
17     end if;
18   end process process1;
19   process2 : process (state, x1)
20   begin
21     case state is
22       when s1 => if x1='1' then
23         next_state <= s2;
24       else
25         next_state <= s3;
26       end if;
27       when s2 => next_state <= s4;
28       when s3 => next_state <= s4;
29       when s4 => next_state <= s1;
30     end case;
31   end process process2;
32   process3 : process (state)
33   begin
34     case state is
35       when s1 => outp <= '1';
36       when s2 => outp <= '1';
37       when s3 => outp <= '0';
38       when s4 => outp <= '0';
39     end case;
40   end process process3;
41 end beh1;

```

Black Boxes

- A design may contain Electronic Data Interchange Format (EDIF) or NGC files generated by synthesis tools, schematic text editors, or any other design entry mechanism, which can be treated as black-boxes during synthesis
- These modules must be instantiated in the code in order to be connected to the rest of the design; but the netlist is propagated to the final top-level netlist without being processed by the synthesis tool.
- Synthesis tools such as XST enables one to attach specific constraints to these Black Box instances.
- One may also have a design block for which an RTL model exists; but the designer's own implementation of this block is in the form of an EDIF netlist and the RTL model is valid for simulation purposes only.

Black-Box Coding Techniques

Verilog

```

1 module v_my_block (in1, in2, dout);
2   input in1, in2;
3   output dout;
4 endmodule
5
6 module v_black_box_1 (DI_1, DI_2, DOUT);
7   input DI_1, DI_2;
8   output DOUT;
9   v_my_block inst (
10     .in1(DI_1),
11     .in2(DI_2),
12     .dout(DOUT)
13   );
14 endmodule

```

VHDL

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity black_box_1 is
4   port(DI_1, DI_2 : in std_logic;
5        DOUT : out std_logic);
6 end black_box_1;
7 architecture archi of black_box_1 is
8   component my_block
9     port (I1 : in std_logic;
10           I2 : in std_logic;
11           O : out std_logic);
12   end component;
13   begin
14     inst: my_block port map (I1=>DI_1,I2=>DI_2,O=>DOUT);
15   end archi;

```

Note: The concept of black-boxes is similar to the notion of precompiled **static libraries** in software languages, which are bypassed by the **compiler** and are linked to the rest of the code by the **linker**.

Question: Name a hardware analog for **dynamic libraries** in software languages

Summary

- Synthesizable HDL coding styles were reviewed in this section.
- These guidelines are for practice and not memorization. In practice, as far as a designer is aware of describing a hardware and thinks logically and concurrently, one does not need to think of the realized hardware

ADVANCED FPGA CODING TECHNIQUES

Toggling a Flag with Multiple Clocks

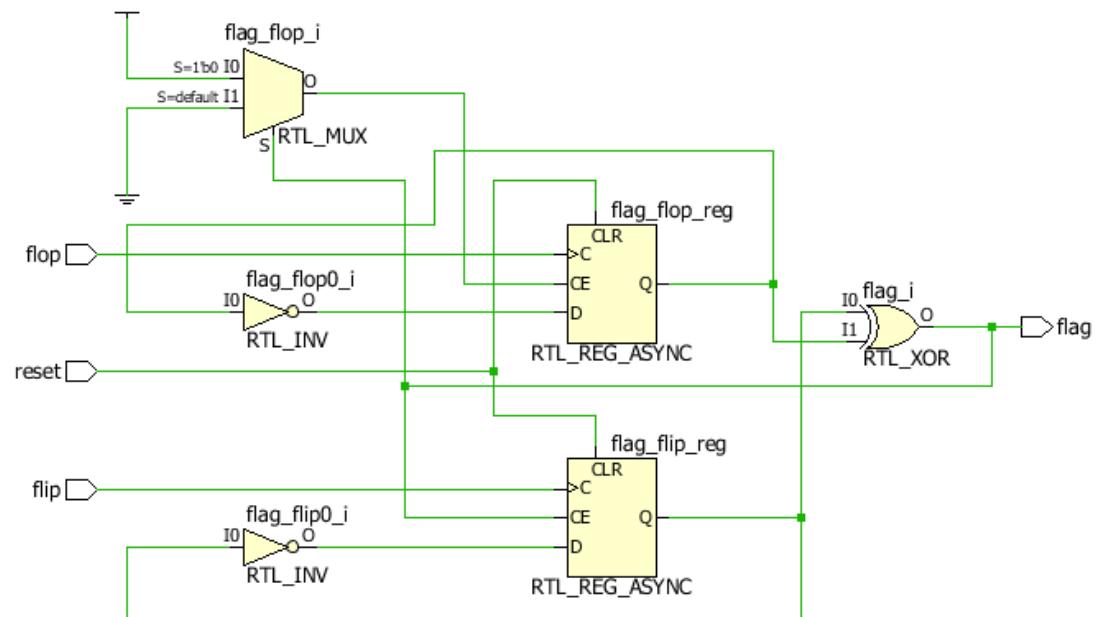
Standard D-type Flip-Flops do not support more than a single clonk. But in practice, there are cases where we need to change a flag using two independent clocks.

Example: handshaking mechanisms

```

1 module handshake (
2     input flip,
3     input flop,
4     input reset,
5     output flag
6 );
7
8     // initialize by flag = 0
9     reg flag_flip = 1'b0;
10    reg flag_flop = 1'b0;
11
12    assign flag = flag_flip ^ flag_flop;
13
14    always @ (posedge flip or posedge reset) begin
15        if(reset)
16            flag_flip <= 0;
17        else if(~flag)
18            flag_flip <= ~flag_flip;
19    end
20
21    always @ (posedge flop or posedge reset) begin
22        if(reset)
23            flag_flop <= 0;
24        else if(flag)
25            flag_flop <= ~flag_flop;
26    end
27 endmodule

```



Clock Speed Reduction

Apart from DCMs, various methods exist for clock speed reduction, including:

```

1 reg clockhalf = 1'b0;
2 always @ (posedge clock)
3     clockhalf <= ~clockhalf;
4
5 always @ (posedge clockhalf) begin
6     //...
7 end

```

Gated-Clock; not recommended nor supported
on most FPGA devices

```

1 reg flag = 1'b0;
2 always @ (posedge clock)
3     flag <= ~flag;
4
5 always @ (posedge clock)
6     if(flag) begin
7         //...
8     end

```

Standard method for clock halving using
FF clock enable

```

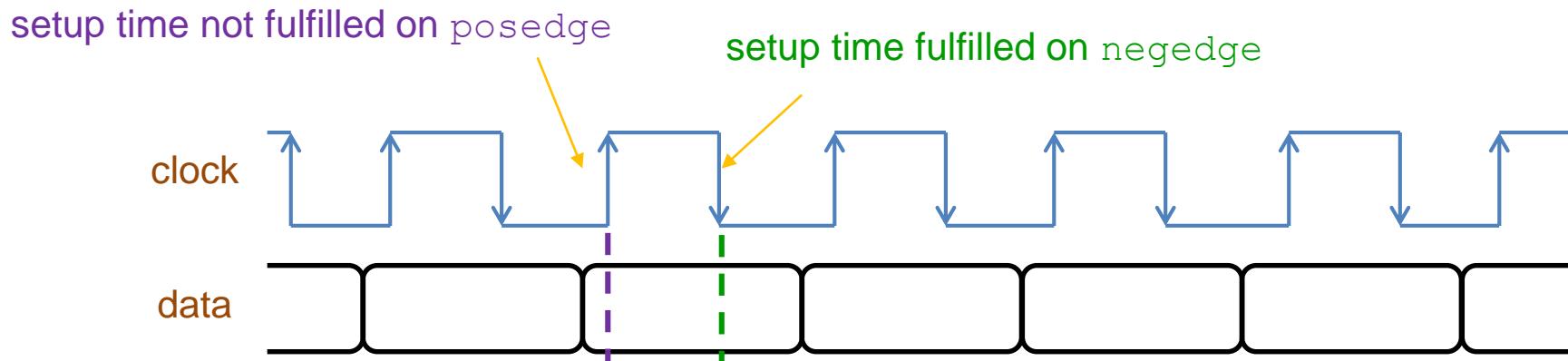
1 parameter DIVIDE = 3'd6;
2 parameter DIVIDE_MINUS_ONE = MAX - 1'd1;
3 reg ce = 1'b0;
4 reg [2:0] counter = 3'd0;
5 always @ (posedge clock)
6     if(counter == DIVIDE_MINUS_ONE) begin
7         counter <= 3'd0;
8         ce <= 1'b1;
9     end
10    else begin
11        counter <= counter + 3'd1;
12        ce <= 1'b0;
13    end
14
15 always @ (posedge clock)
16     if(ce) begin
17         //...
18     end

```

Standard method for clock division using FF clock enable

Mixed Clock-Edge Design

- It is possible to use both **positive** and **negative** clock edges in a single design; but **it should be avoided as much as possible**
- Using mixed clock-edges **does not double the clock rate**; but it rather reduces the time for combination logic result settlements
- Utilization of mixed clock-edges should be confined to phase compensation between two signals when setup or hold-times are not fulfilled using a single edge (commonly at FPGA I/O)
- **Example:**



Standard Resetting Mechanisms

- Although both **synchronous** and **asynchronous** reset mechanisms are supported in FPGA designs, it is highly recommended to use a unified resetting mechanism throughout the entire design.
- Synchronous resets** with sufficient flip-flop synchronizer stages are preferred over **asynchronous resets** (due to lower probability of **metastability**)
- Even if the original reset command is asynchronous (e.g. using a push-button or software command), it is good practice to make an internal synchronous reset flag

```

1  always @ (posedge clock or posedge reset)
2    if (reset)begin // Asynchronous Reset
3      //...
4    end
5    else begin
6      //...
7    end

```

Supported asynchronous reset mechanism

```

1  always @ (posedge clock)
2    if (reset)begin // Synchronous Reset
3      //...
4    end
5    else begin
6      //...
7    end

```

Preferred synchronous reset mechanism

```

1  always @ (posedge clock)begin // synchronizer Flip-Flops
2    temp <= asynchreset;
3    synchreset <= temp;
4  end
5
6  always @ (posedge clock)
7    if (synchreset)begin // Synchronous Reset
8      //...
9    end
10   else begin
11     //...
12   end

```

Generating synchronous from asynchronous reset flag

Increasing Fan-out by HW Replication

- The maximum fan-out of a logic circuit output is the maximum number of gate inputs it can drive without loading effects disturb its function (switching speed and voltage level)
- In contemporary FPGAs, Flip-Flop fan-outs are very high (several hundreds) and only the most frequently used signals (such as CLOCK, RESET, CE, ...) may face fan-out issues
- The fan-out of a logic circuit may be increased by user constraints or hardware replication in HDL

synthesis attributes to avoid register optimization and merging of ce1 and ce2

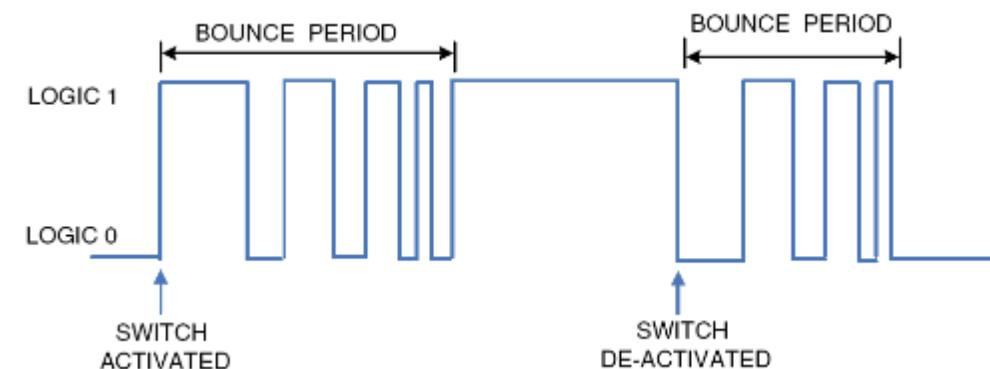
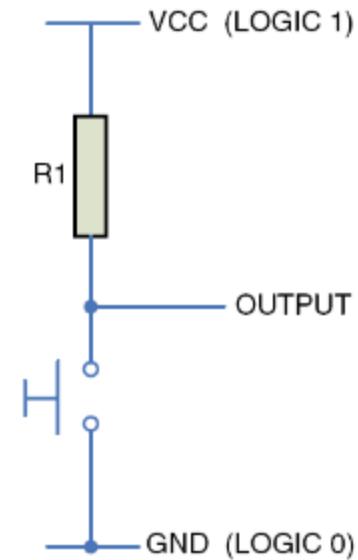
```

1 parameter DIVIDE = 3'd6;
2 parameter DIVIDE_MINUS_ONE = MAX - 1'd1;
3 (* dont_touch = "true" *) reg ce1 = 1'b0;
4 (* dont_touch = "true" *) reg ce2 = 1'b0;
5 reg [2:0] counter = 3'd0;
6 always @ (posedge clock)
7   if(counter == DIVIDE_MINUS_ONE) begin
8     counter <= 3'd0;
9     ce1 <= 1'b1;
10    ce2 <= 1'b1;
11  end
12  else begin
13    counter <= counter + 3'd1;
14    ce1 <= 1'b0;
15    ce2 <= 1'b0;
16  end
17
18 always @ (posedge clock) if(ce1) begin
19   // First instance of usage
20 end
21
22 always @ (posedge clock) if(ce2) begin
23   // Second instance of usage
24 end

```

Debouncing

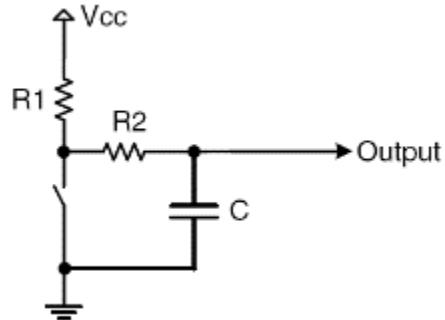
- In digital designs, **bouncing** (between 0 and 1) occurs during manual switch transitions
- The objective of **debouncing** is to avoid the mis-detection or multiple counting of events during switch transitions
- Debouncing can be implemented both in **hardware (analog)** and **software (digital)**



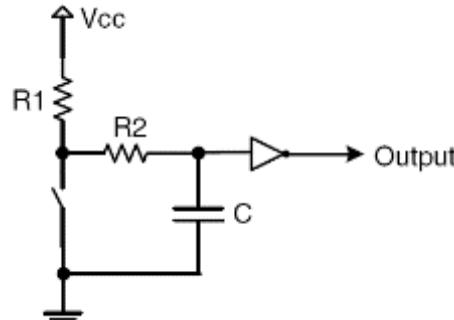
Reference: Arora, M. (2011). *The art of hardware architecture: Design methods and techniques for digital circuits*. Springer Science & Business Media, Chapter 8

Hardware Debouncing Techniques

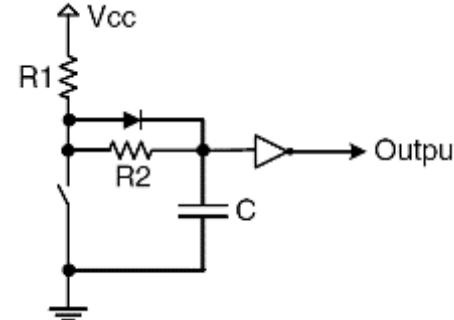
- Various hardware debouncing mechanisms:



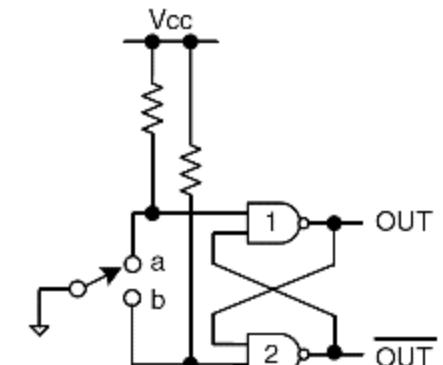
RC debouncer



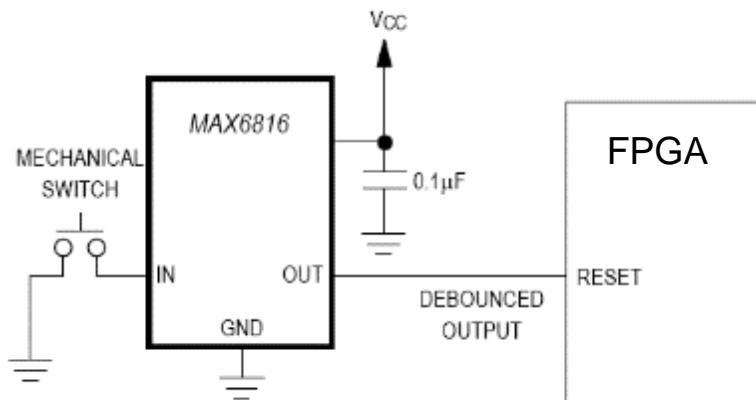
RC debouncer with digital logic



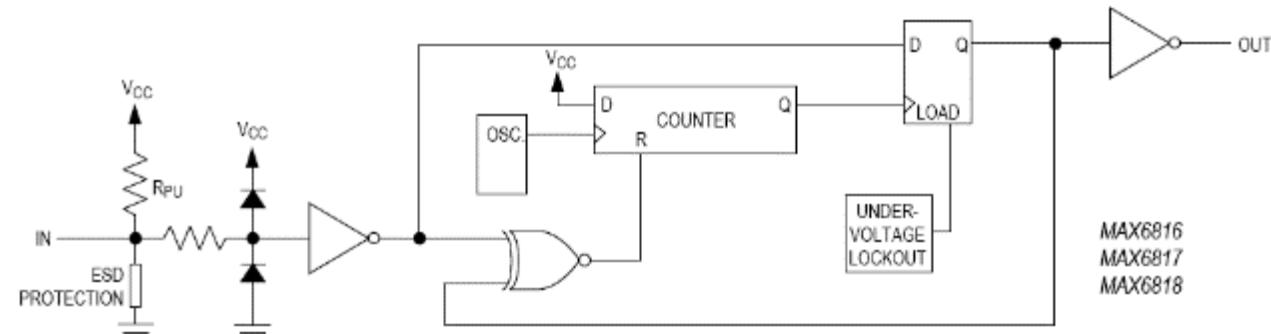
Robust RC debouncer
with digital logic



SR debouncer



IC debouncer



MAX6816's internal mechanism

Software Debouncing Techniques

- Software debouncing mechanisms:

```

1 ; Interrupt Service Routine Pseudo-Code
2 DR: PUSH PSW ; SAVE PROGRAM STATUS WORD
3 LOOP: CALL DELAY ; WAIT A FIXED TIME PERIOD
4 IN SWITCH ; READ SWITCH
5 CMP ACTIVE ; IS IT STILL ACTIVATED?
6 JT LOOP ; IF TRUE, JUMP BACK
7 CALL DELAY ;
8 POP PSW ; RESTORE PROGRAM STATUS
9 EI ; RE-ENABLE INTERRUPTS
10 RETI ; RETURN BACK TO MAIN PROGRAM

```

ISR assembly language debouncer pseudo-code

```

1 // Variable used to count:
2 unsigned char counter;
3 // Variable used as the minimum duration of a valid pulse:
4 unsigned char T_valid;
5 void main(){
6     P1 = 255;           // Initialize port 1 as input port
7     T_valid = 100;    /* Arbitrary number from 0 to 255 where
8                      |          | the pulse is validated */
9     while(1){ // infinite loop
10        if (counter < 255){ // prevent the counter to roll
11            // back to 0
12            counter++;
13        }
14        if (P1_0 == 1){
15            counter = 0; // reset the counter back to 0
16        }
17        if (counter > T_valid){
18            // Code to be executed when a valid
19            // pulse is detected.
20        }
21    }
22 }
23 }

```

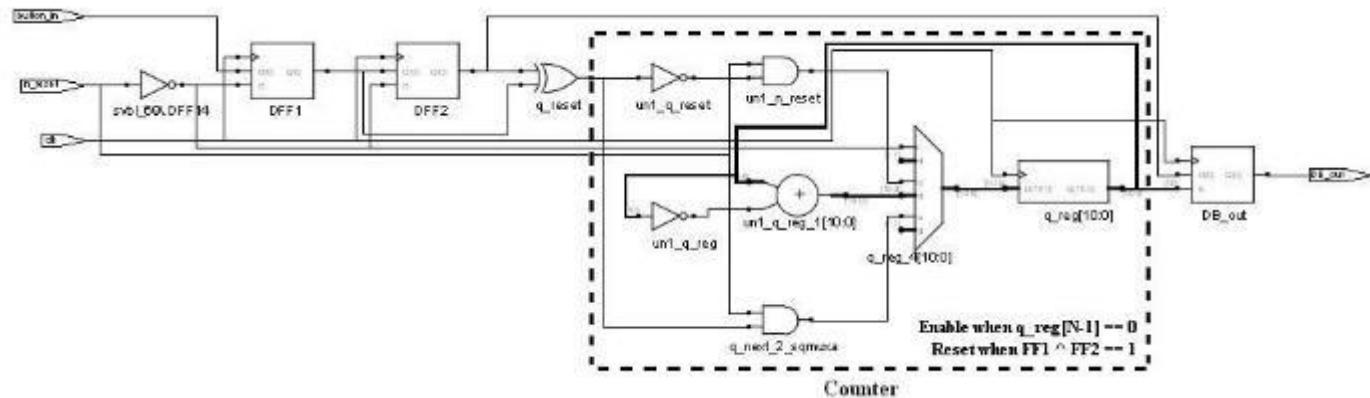
C language debouncer pseudo-code

HDL Debouncing Techniques

```

1 module DeBounce(input clk, n_reset, button_in, output reg DB_out);
2     // internal constant
3     parameter N = 11;
4     reg [N-1 : 0] q_reg; // timing regs
5     reg [N-1 : 0] q_next;
6     reg DFF1, DFF2; // input flip-flops
7     wire q_add; // control flags
8     wire q_reset;
9     assign q_reset = (DFF1 ^ DFF2); // xor input FFs to look for level change to reset counter
10    assign q_add = ~q_reg[N-1]; // add to counter when q_reg msb is equal to 0
11    // combo counter to manage q_next
12    always @ ( q_reset, q_add, q_reg)
13        case( {q_reset , q_add})
14            2'b00 : q_next <= q_reg;
15            2'b01 : q_next <= q_reg + 1;
16            default : q_next <= { N {1'b0} }
17        endcase
18    // Flip flop inputs and q_reg update
19    always @ ( posedge clk )
20        if(n_reset == 1'b0) begin
21            DFF1 <= 1'b0;
22            DFF2 <= 1'b0;
23            q_reg <= { N {1'b0} };
24        end
25        else begin
26            DFF1 <= button_in;
27            DFF2 <= DFF1;
28            q_reg <= q_next;
29        end
30    // counter control
31    always @ ( posedge clk )
32        if(q_reg[N-1] == 1'b1)
33            DB_out <= DFF2;
34        else
35            DB_out <= DB_out;
36    endmodule

```



Ref: <https://eewiki.net/pages/viewpage.action?pageId=13599139>

PART III

Advanced Topics in Digital Design and Implementation

NUMBER REPRESENTATION

Number Representation in PLD Systems

- While number representation is fully standardized and rather automatically handled in multipurpose CPUs and GPUs (and is rarely a concern for the designer), it is an essential and time-taking part of most FPGA-based designs.
- In this section, we study:
 - The most common number representation standards
 - Fixed-point representation issues
 - Statistical analysis of truncation and rounding errors during data acquisition (using analog-to-digital converters) and calculations

An Overview of Binary Number Representation

- For many reasons radix-2 has remained the dominant number representation in digital hardware design:
 - In early technologies: the difficulty of generating high-speed switching logic circuits with more than two distinct and distinguishable levels of voltages.
 - In current technologies: besides the simplicity of radix-2, the huge body of literature, algorithms, codes, hardware (transistors, gates, etc.), and engineering experience and conventions, which already exist for radix-2 calculations makes it too expensive to migrate to higher radices.

Binary Number Representation

Number representation can be studied from various aspects, including:

- Numbers of Interest:
 - Integers
 - Reals
- Sign Representation:
 - Unsigned
 - Signed
- Fractional Number Representation:
 - Fixed-point
 - Floating-point

Accuracy of Finite Length Binary Number Representations

Question: How accurate is it to represent numbers (integer or fractional) in radix-2 using finite number of bits?

Basis Representation Theorem: For a given base b , any integer $x \in \mathbb{Z}$ can be uniquely represented as follows:

$$x = a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b^1 + a_0$$

where $a_j \in \{0, 1, \dots, b-1\}$ and $a_k \neq 0$.

Dyadic Rationals Theorem: The dyadic rational set \mathbb{P} (numbers which can be represented as an integer divided by a power of 2), is dense in the set of real numbers \mathbb{R} . This means that for any $x \in \mathbb{R}$, there exists a $y \in \mathbb{P}$ that is “as close as you like” to x .

Conclusion: Real numbers can be approximated in radix-2 with finite number of bits, up to a desired level of precision.

Signed Binary Number Representation Standards

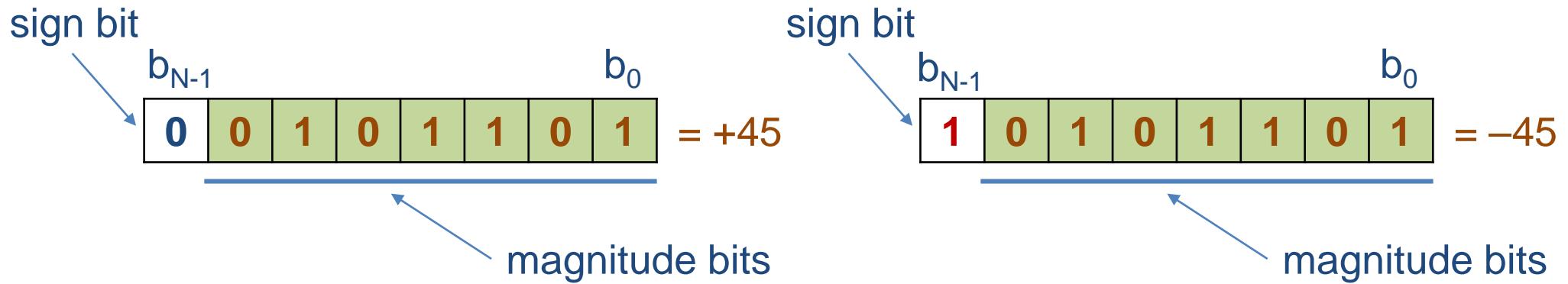
The most popular signed binary number representation standards are:

- Sign-Magnitude
- One's-Complement
- Two's-Complement
- Straight Offset Binary (SOB)
- Binary Coded Decimal (BCD)
- Canonical Signed Digit (CSD)

Sign-Magnitude Representation

The MSB is reserved for sign representation (0 for + and 1 for -). The remaining bits are used to represent the absolute magnitude. With N bits, it can code from $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$.

Decimal equivalent: $X_{10} = (-1)^{b_{N-1}}[b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \dots + b_12 + b_0]$



Advantage: Simple to generate and convert

Disadvantage: There are two zeros (+0 and -0); difficult to handle during arithmetic operations

One's Complement

The MSB denotes the sign (0 for + and 1 for -). With N bits, it can code from $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$. Each bit corresponds to a coefficient of a power of two in its decimal equivalent.

$$\text{Decimal equivalent: } X_{10} = -b_{N-1}(2^{N-1} - 1) + b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \dots + b_12 + b_0$$

all bits one

sign bit	b_{N-1}	b_0								$= +45$	sign bit	b_{N-1}	b_0								$= -45$
	0	0	1	0	1	1	0	1				1	1	0	1	0	0	1	0		

Advantage: Simple to generate and convert

Disadvantage: There are two zeros (+0 and -0); difficult to handle during arithmetic operations

Two's Complement

The MSB denotes the sign (0 for + and 1 for -). With N bits, it can code from -2^{N-1} to $(2^{N-1} - 1)$. Each bit corresponds to a coefficient of a power of two in its decimal equivalent.

$$\text{Decimal equivalent: } X_{10} = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \dots + b_12 + b_0$$

doesn't fit into N bits



Advantage: No repeated zeros; can code -2^{N-1} ; no sign control needed during arithmetic operations, and several other advantages (is the most popular signed number representation format)

Disadvantage: Slightly more difficult to read the decimal equivalent from the binary form (for human).

One's Complement vs. Two's Complement

- 2's complement is the most common binary representation used in computation machines.
- A major property of 2's complement is that the binary values are increased by one-by-one from the most negative to the most positive without a break (by discarding any carry values beyond the word length).
- The default implementation of arithmetic operations in Verilog (since Verilog 2001) is in this format.

1's Complement Representation

Value	Binary
-128	1111 1111 1111 1111
-127	1111 1111 1111 1110
-126	1111 1111 1111 1101
...
-1	1111 1111 1111 1110
0	0000 0000 0000 0000
+1	0000 0000 0001 0000
+2	0000 0010 0000 0000
...
-125	1000 0010 0000 0000
-124	1000 0001 0000 0000
-123	1000 0000 0000 0000
...
-2	1111 1111 1111 1110
-1	1111 1111 1111 1101
0	0000 0000 0000 0000
+1	0000 0001 0000 0000
+2	0000 0010 0000 0000
...
-126	1000 0010 0000 0000
-127	1000 0001 0000 0000
-128	1000 0000 0000 0000

2's Complement Representation

Value	Binary
-128	1000 0000 0000 0000
-127	1111 1111 1111 1111
-126	1111 1111 1111 1110
...
-1	1111 1111 1111 1110
0	0000 0000 0000 0000
+1	0000 0001 0000 0000
+2	0000 0010 0000 0000
...
-125	0111 1111 1111 1110
-124	0111 1111 1111 1101
-123	0111 1111 1111 1100
...
-2	1111 1111 1111 1110
-1	1111 1111 1111 1101
0	0000 0000 0000 0000
+1	0000 0001 0000 0000
+2	0000 0010 0000 0000
...
-126	0111 1111 1111 1110
-127	0111 1111 1111 1101
-128	0111 1111 1111 1111

Finding One and Two's Complements

1's Complement: Flip all the bits (0 to 1, and 1 to 0)

2's Complement:

- **Method 1:** Calculate the 1's complement, plus one
- **Method 2:** Subtract the number from 2^N (this is where the name 2's complement comes from)
- **Method 3:** Starting from the LSB, preserve all the bits as they are, up to (and including) the right most 1. Flip all the remaining bits up to the MSB

Note: The 2's complement of -2^{N-1} can not be represented in N bits. Therefore, during calculations, it's 2's complement overflows and becomes equal to itself (just like the 2's complement of zero)! This phenomenon can be mathematically explained by the [orbit-stabilizer theorem](#).

Properties of Two's Complement

- When fitting an N bit 2's complement number into M bits ($M > N$), the number should be **sign extended**, i.e., the left most $M-N$ bits should be filled with the MSB (sign bit) of the original number:

```
wire signed [7:0] w;
wire signed [11:0] x;
assign x = {{4{w[7]}}, w};
```

- In arithmetic right-shifts, the number should be filled by the sign bit from the left:

```
wire signed [7:0] w;

wire signed [7:0] xr = w >>> 3; // arithmetic shift-right (filled by signs from MSB)
wire signed [7:0] yr = w >> 3; // logic shift-right (filled by zeros from MSB)

wire signed [7:0] xl = w <<< 3; // arithmetic shift-left (filled by zeros from LSB)
wire signed [7:0] yl = w << 3; // logic shift-left (filled by zeros from LSB)
```

Properties of Two's Complement (continued)

3. No additional circuits are required for handling the signs during addition or subtraction (except for overflow checking). In fact, 2's complement numbers can be treated as unsigned numbers during such arithmetic operations.
4. Overflow check: If two numbers with the same sign are added, overflow occurs **if and only if** the result has an opposite sign.

Example:

$$\begin{array}{r} 0111 \quad (\text{carry}) \\ 0111 \quad (7) \\ + 0011 \quad (3) \\ \hline 1010 \quad (-6) \quad \text{invalid!} \end{array}$$

Properties of Two's Complement (continued)

5. **Two's Complement Intermediate Overflow Property:** “In successive calculation using 2's complement arithmetic (**allowing overflows instead of saturation**), if it is guaranteed that the final result will fit in the assigned registers, then intermediate overflows are harmless and will not affect the final answer.

Example (IIR Filter): $y_n = a.y_{n-1} + x_n$

Refs:

- Khan, S. A. (2011). *Digital design of signal processing systems: a practical approach*. John Wiley & Sons., Section 3.5.7
- Smith, J. O. (2007). *Introduction to digital filters: with audio applications* (Vol. 2). Julius Smith., P. 201

Note: Very interesting property; but I haven't seen a rigorous statement or proof for it, yet. Please let me know, if you find a good reference.

Straight Offset Binary (SOB)

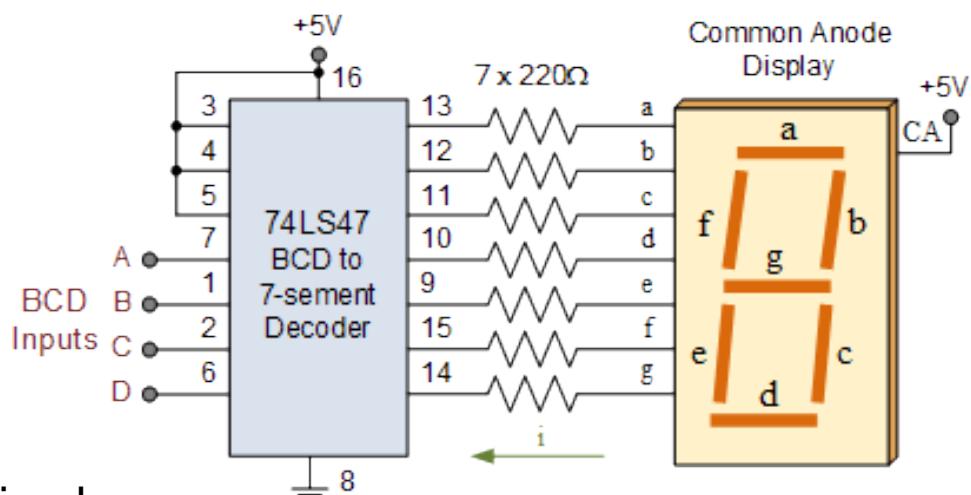
- Offset Binary is a binary code in which the code represents analog values between positive and negative Full-Scale
- Using N bits, starts assigns all-zeros to -2^{N-1} and increments one-by-one up to $2^{N-1} - 1$.
- **Conversion to 2's complement:** Flip the MSB to convert from SOB to 2's complement and vice versa.
- **Application:** SOB is most common in Flash Analog-to-Digital Converters (ADC) and Digital-to-Analog Converters (DAC) that use ladder comparators.

SOB	Decimal	2's Complement
1111	7	0111
1110	6	0110
1101	5	0101
1100	4	0100
1011	3	0011
1010	2	0010
1001	1	0001
1000	0	0000
0111	-1	1111
0110	-2	1110
0101	-3	1101
0100	-4	1100
0011	-5	1011
0010	-6	1010
0001	-7	1001
0000	-8	1000

Binary Coded Decimal (BCD)

- A class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits (usually four or eight).
- Special bit patterns are used for a sign or for other indications (e.g., error or overflow)
- **Applications:** whenever human interaction is needed; such as LCDs, 7-segments, etc.

Decimal digit	BCD Code			
	8 4 2 1	4 2 2 1	5 4 2 1	3 0 4 2 1
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 0 1
2	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0 0
3	0 0 1 1	0 0 1 1	0 0 1 1	0 0 1 1 1
4	0 1 0 0	1 0 0 0	1 0 0 0	0 1 0 0 0
5	0 1 0 1	0 0 1 1	0 0 1 1	1 0 0 1 1
6	0 1 1 0	1 1 0 0	1 1 0 0	1 0 1 0 0
7	0 1 1 1	1 1 0 1	1 1 0 1	1 0 1 0 1
8	1 0 0 0	1 1 1 0	1 1 1 0	1 0 1 1 1
9	1 0 0 1	1 1 1 1	1 1 1 1	1 1 0 0 0



Canonical Signed Digit (CSD)

- CCD is a three-symbol coding system in terms of powers of two.
- It uses a sequence of (+, 0, -) to code numbers. For example, the integer 23 can be expanded as follows:

$$23 = + 2^5 - 2^3 - 2^0$$

In CCD, 23 is coded as (+0–00–), i.e.,

- Positive powers of two are denoted by +
- Negative powers of two are denoted by –
- Missing powers of two are denoted by 0

- CCD is popular in some digital signal processors (DSP)

Note: CCD is a non-unique number representation

Note: Statistically, the probability of a digit being zero in CCD can be shown to be close to 66% (vs. 50% in 2's complement encoding). This property leads to more efficient hardware implementations of add/subtract networks and multiplication by constants.

Further Reading: Khan, S. A. (2011). *Digital design of signal processing systems: a practical approach*. John Wiley & Sons., Chapter 6

Fractional Number Representation

The most common binary representations of fractional numbers are:

- **Floating-Point:** Uses an exponential representation of a number; it is used in most CPUs and some DSP. In FPGA, **floating point units (FPUs)** are provided by some vendors as hard or soft IP
- **Fixed-Point:** Uses positive and negative powers of two expansion of a number with a fixed radix point; it is commonly used in fixed-point DSP and microcontrollers
- **Mixed-Precision:** Uses positive and negative powers of two expansion of a number with a different radix point (at each point of the computing system); it is commonly used in FPGA design

Floating-Point Number Representation

- The basic idea of floating point (FP) representation is to approximate a real number in terms of a fixed number of significant digits (**significands** or **mantissa**) scaled by an **exponent** of a fixed **base** (e.g., 2, 10, 16, etc.).
- For example:

$$1.2345 = 12345 \times 10^{-4}$$

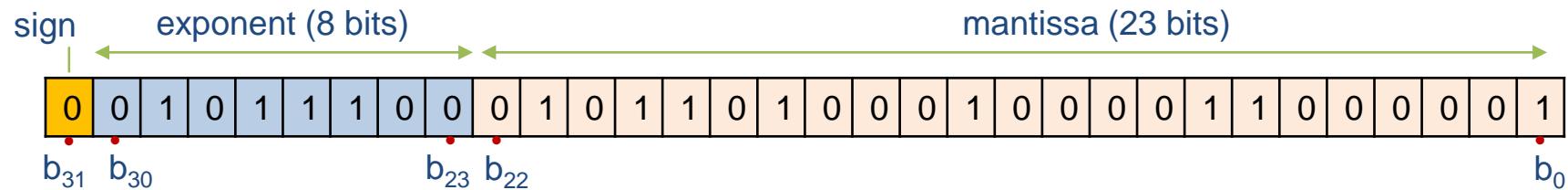


exponent
significand base

- Apparently, not all real numbers can be represented in this format (using finite number of digits). However, FP provides an approximation with a fixed relative error throughout the real line (i.e., small errors for small numbers and larger errors for large numbers).

IEEE 754 Single-Precision Binary Floating-Point Format

- According to IEEE 754 floating-point standard:



- The decimal equivalent is:

$$X_{10} = (-1)^S \times 2^{e-B} \times (1 + \sum_{i=1}^M b_{M-i} 2^{-i})$$

The exponent is selected such that the left-most bit of the mantissa is always 1 (which isn't stored in the binary form), making the representation *unique*.

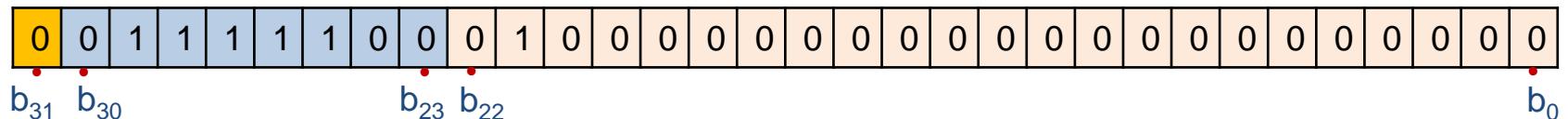
where:

- Total number of bits is 32 in single precision ([binary32](#)) and 64 in double precision ([binary64](#))
- S is the sign bit (b_{31} in single precision and b_{63} in double precision)
- e is the exponent (8 bits in single precision and 11 bits in double precision)
- B is a constant bias (equal to 127 in single precision and 1023 in double precision)
- M is the fractional length (23 bits in single precision and 52 bits in double precision)

Single-Precision Binary Floating-Point Examples

Example 1: binary floating point to decimal

$$0x3E200000 = (0011\ 1110\ 0010\ 0000\ 0000\ 0000\ 0000)_2$$



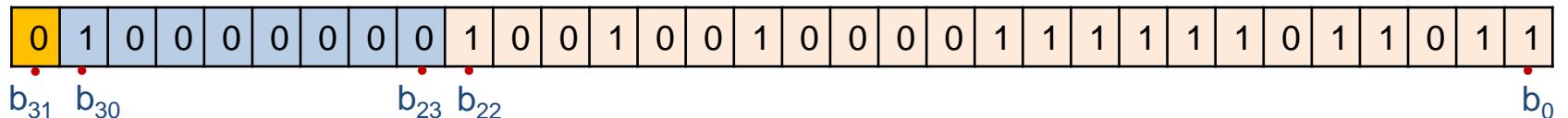
The decimal equivalent is $(-1)^0 \times 2^{124-127} \times (1 + 0.25) = 0.15625$

Example 2: decimal to hex/binary floating point

Scale the number in the form of $\pm 2^a m$, where $1 \leq |m| < 2$ and $a \in \mathbb{Z}$, to find the exponent and mantissa

$$\pi (3.1415926535897932384626433832795...) \approx \underline{3.1415927410125732421875}$$

the most accurate 32-bit single-precision approximation for π



$$\text{which is } 0x40490FDB = (0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1011)_2$$

Floating-Point Arithmetic

Addition/Subtraction:

1. Make the smallest exponent equal to the biggest (by right-shifting the mantissa)
2. Add/subtract the mantissas (note that the smaller ones may vanish to 0 during the right-shifts)

Multiplication/Division:

1. Add/subtract the exponents
2. Multiply/Divide the mantissas
3. Scale and round the results

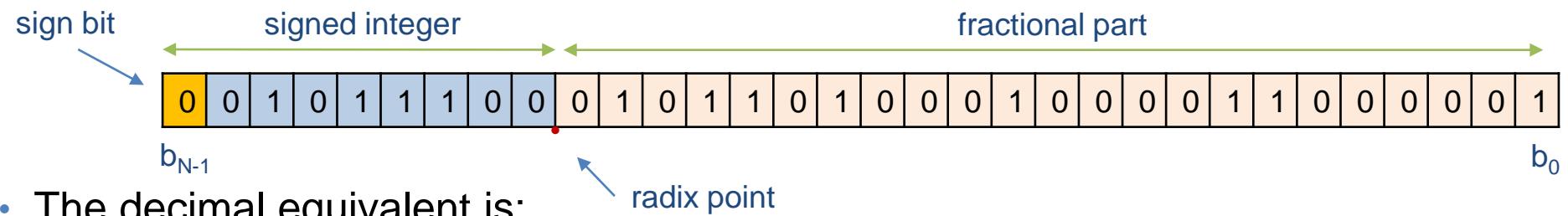
Special Values:

Floating-point representation has reserved codes for special values including: 0^+ , 0^- , $+\infty$, $-\infty$, and Not-a-Number (NaN) such as $0/0$, $+\infty/-\infty$, $0 \times \infty$

Note: Due to the (implicit) leading 1 in front of the mantissa, zero needs to be defined as a special value (when all the bits of the exponent and mantissa are zero), which is different from epsilon ($\pm 2^{-127}$)

Fixed-Point Number Representation

- Fixed-point is basically the 2's complement representation with a fixed power-of-two scaling factor for changing the radix point to enable fractional number representations:



- The decimal equivalent is:

$$X_{10} = 2^{-M} \times (-b_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i)$$

signed two's complement

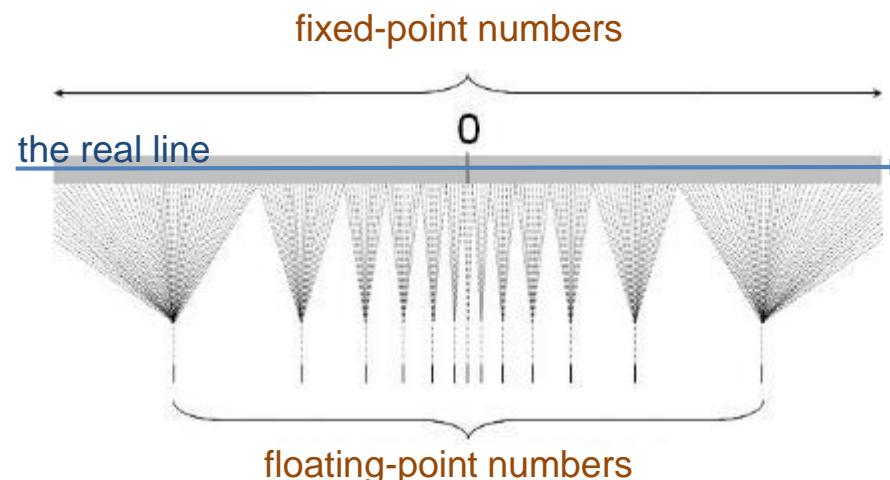
where:

- N is the total number of bits
- M is the fractional point

Note: In fixed-point systems the radix point location is assumed to be fixed throughout the entire system. That's where the name comes from.

Floating-Point vs. Fixed-Point

1. For the same number of bits, they can (almost) code the same number of real numbers.
2. Fixed-point uses all possible codes for number representation, while floating point reserves a few codes for **special values**. Floating-point has a larger dynamic range (the ratio of the largest to smallest number that are represented)
3. In fixed-point, the range of its MIN and MAX over the real line is quantized to equally spaced numbers (therefore the approximation error is uniform from MIN to MAX); in floating-point, the spacing of numbers is non-uniform (groups of numbers with a fixed **intra-gaps** but different **inter-gaps**)
4. Fixed-point hardware architectures are simpler than floating-point architectures; floating-point architectures have additional circuitry for handing special values.



Inspired from: Izquierdo, Luis R. and Polhill, J. Gary (2006). 'Is Your Model Susceptible to Floating-Point Errors?'. Journal of Artificial Societies and Social Simulation 9(4)4
[<http://jasss.soc.surrey.ac.uk/9/4/4.html>](http://jasss.soc.surrey.ac.uk/9/4/4.html)

The $Q_{m,n}$ Fixed-Point Convention

- In order to denote the total number of bits and the bits assigned to the integer and fractional parts of a fixed-point number, various conventions exist. For example,
 - Texas Instruments' Q_N format (or $Q_{1,N}$) assumes 1 bit (the sign bit) as the integer part and N bits for the fractional part.
 - Matlab's fixed-point toolbox takes the total number of bits and the fractional length to form an **fi-object**.

Throughout this course, we use the $Q_{m,n}$ convention, where:

- m is the number of bits assigned to the **integer** part
- n is the number of bits assigned to the **fractional** part
- $N = m + n$ is the **total number of bits** (including the sign)
- The numbers are **signed**, therefore the MSB represents the sign

Fixed-Point Arithmetic

Addition/Subtraction:

1. Align the radix points
2. Zero pad the LSB of numbers with shorter fractional lengths
3. Sign extend the MSB of numbers with shorter integer lengths
4. Apply addition/subtraction

Multiplication/Division:

1. Apply multiplication/division as if they were integer valued (regardless of the radix point)
2. Find the appropriate radix point by adding/subtracting the radix points

Note: Bit-growth occurs during fixed-points arithmetic, which is handled by either:

1. increasing the number of bits,
2. truncation/rounding from the LSB or MSB (is discussed in details later), or
3. a combination of both 1 and 2

Bit-Growth in Fixed-Point Arithmetic

In order to guarantee that no overflow occurs during arithmetic operations, the number of output bits should be longer than the arithmetic operands:

$$1. \quad Q_{m_1.n_1} \pm Q_{m_2.n_2} = Q_{m.n}$$

where $m = \max(m_1, m_2) + 1$ and $n = \max(n_1, n_2)$

$$2. \quad Q_{m_1.n_1} \times Q_{m_2.n_2} = Q_{m.n}$$

where $m = m_1 + m_2$ and $n = n_1 + n_2$

Note: During multiplication, $N = N_1 + N_2 - 1$ is generally enough. The only exception (requiring $N = N_1 + N_2$) is for signed numbers when the two most negative numbers (-2^{N_1-1} and -2^{N_2-1}) are multiplied together, resulting in $+2^{(N_1+N_2-2)}$, which overflows in $N = N_1 + N_2 - 1$ bits and requires $N = N_1 + N_2$. This single bit can be saved by either:

1. Making sure that the two operands are never equal to the most negative numbers (this is possible when one of the operands is a known constant)
2. Approximating $2^{(N_1+N_2-2)}$ with $2^{(N_1+N_2-2)} - 1$! Yes, this approximation is OK in many systems.

Controlling Bit-Growth in Fixed-Point Systems

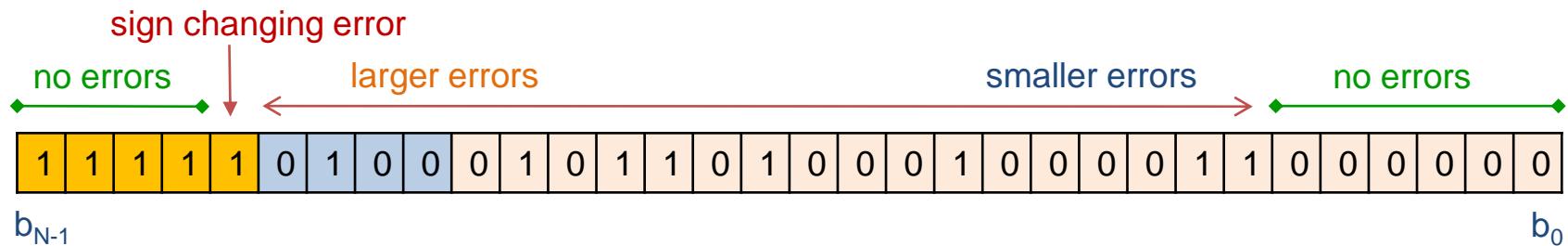
It is impractical (and unnecessary) to increase the number of bits after successive arithmetic operations. Bit growth can be controlled by discarding either from the LSB or MSB of the arithmetic result.

- **When to discard from the MSB?**

- Only possible when the full-length is not utilized or the arithmetic operation (mathematically) guarantees that no bit growths occur → **results in no errors**
- If the full-length is utilized → **causes large sign/amplitude errors**

- **When to discard from the LSB?**

- The right most LSB zeros can be discarded without any errors
- Truncating/rounding non-zero LSB results in **relatively small errors**, depending on the number's magnitude
- A **stochastic framework** is required to analyze the **average truncation/rounding error effect**.



Truncation/Rounding Error Analysis

- The truncation procedure can be modeled by an operator $Q(\cdot)$:

$$y_n = Q(x_n) = x_n + e_n$$

x : input sample (signal)

y : truncated/rounded result

e : truncation/rounding error

n : sample index

- The impact of truncation error depends on both the original sample (signal) and the truncated values' amplitudes.
- In **continuous data streams**, the most common approach for studying the truncation error impact is to measure the ratio of the average data power to the average noise power, known as the **signal-to-noise ratio (SNR)**:

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{E\{x_n^2\}}{E\{e_n^2\}} \right)$$

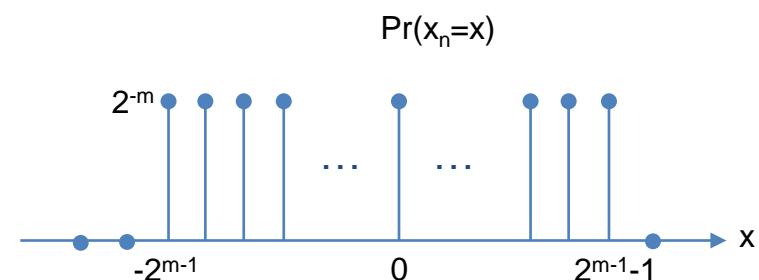
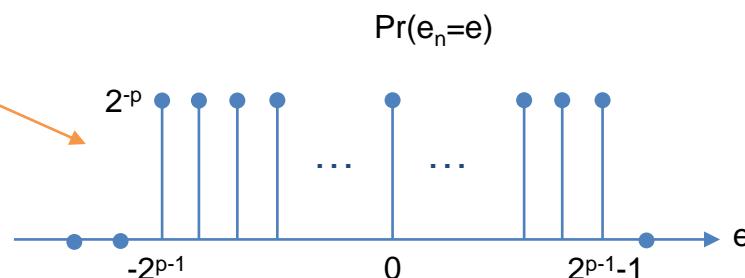
where $E\{\cdot\}$ denotes averaging (or stochastic **expectation**) over all ensembles.

Note: The calculation of the SNR requires prior assumptions regarding the input stream and the truncation error distribution.

Truncation/Rounding Error SNR Calculation

Suppose that we have an m bit signed integer sequence x_n , for which we want to **round** the p LSB bits (to zero) and obtain y_n . Assuming a **uniform distribution** for x_n , the **probability density functions (pdf)** of x_n and the error sequence e_n are:

rounding instead
of truncation (+
and - error
values)



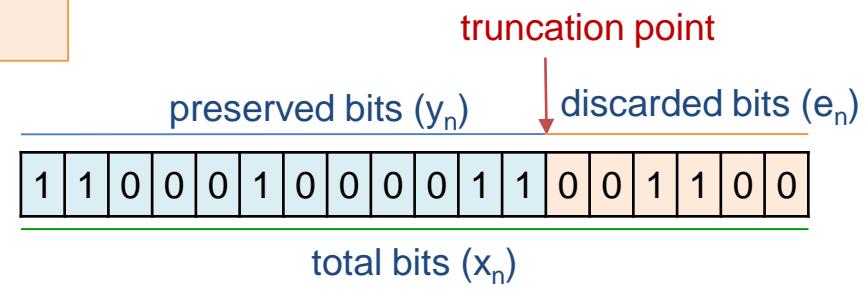
Error mean:

$$\bar{e} = E\{e_n\} = \sum_{i=-2^{p-1}}^{2^{p-1}-1} i \cdot \frac{1}{2^p} = -\frac{1}{2}$$

slightly biases
towards negative
numbers

Error variance:

$$\sigma_e^2 = E\{(e_n - \bar{e})^2\} = \sum_{i=-2^{p-1}}^{2^{p-1}-1} (i + \frac{1}{2})^2 \cdot \frac{1}{2^p} = \frac{2^{2p} - 1}{12}$$



Truncation/Rounding Error SNR Calculation

(continued)

Similar results hold for the mean and variance of x_n . Therefore the SNR is:

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{\sigma_x^2}{\sigma_e^2} \right) = 10 \log_{10} \left(\frac{2^{2m} - 1}{2^{2p} - 1} \right)$$

which for large p can be approximated as:

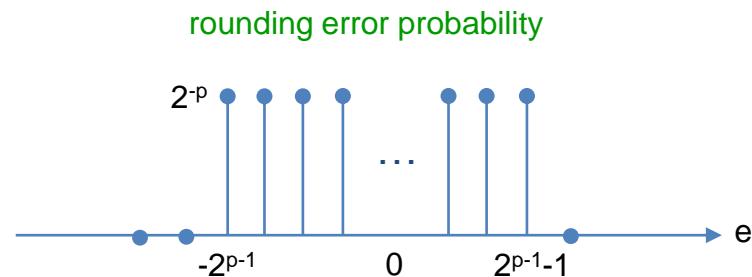
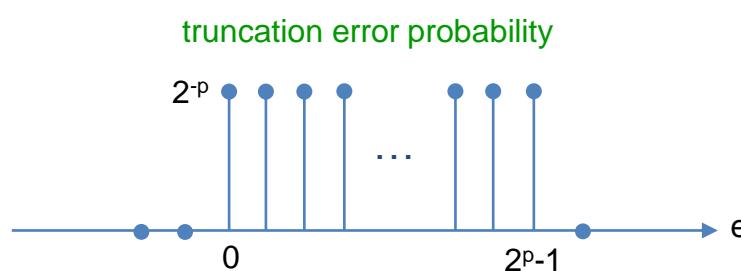
$$\text{SNR}_{\text{dB}} \approx 10 \log_{10} \left(\frac{2^{2m}}{2^{2p}} \right) = 20(m - p) \log_{10} 2 \approx 6.02(m - p)$$

Note: This is the 6dB per-bit rule of thumb: *truncating each bit reduces the SNR for about 6dB*. We will find a similar rule later for ADC performance with different signal and noise distributions.

Exercise: Derive the above equations (mean and variance of error) analytically. Do the results change if the number is in the $Q_{m,n}$ format?

Truncation vs. Rounding

- While truncation simply discards the unnecessary bits, rounding approximates with the closest number.
- Rounding is commonly preferred over truncation, as it is **less-biased** (the very small bias is due to the representation of -2^{p-1} in 2's complement).



Example: `round(3.7) = 4; truncate(3.7) = 3;`

- Truncation versus rounding in Verilog:

```

1 reg [11:0] a;
2 wire [9:0] a_round = a[11:2] + a[1];
3 wire [9:0] a_trunc = a[11:2];

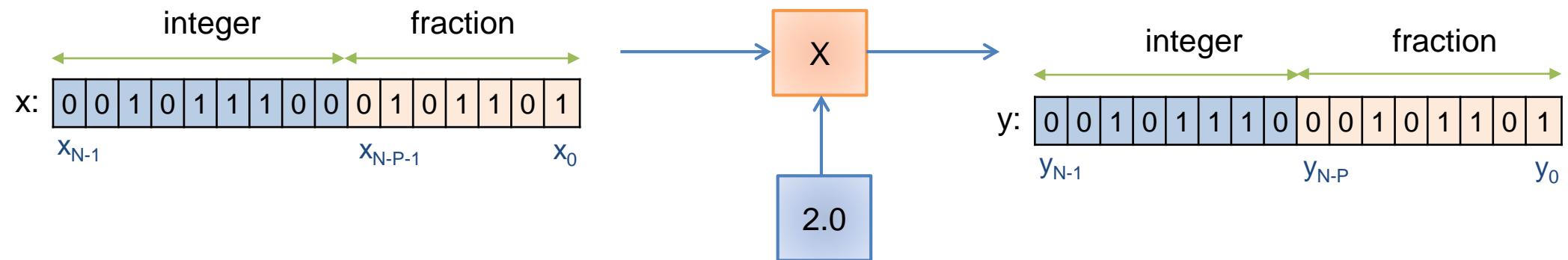
```

Radix-10 equivalent trick:

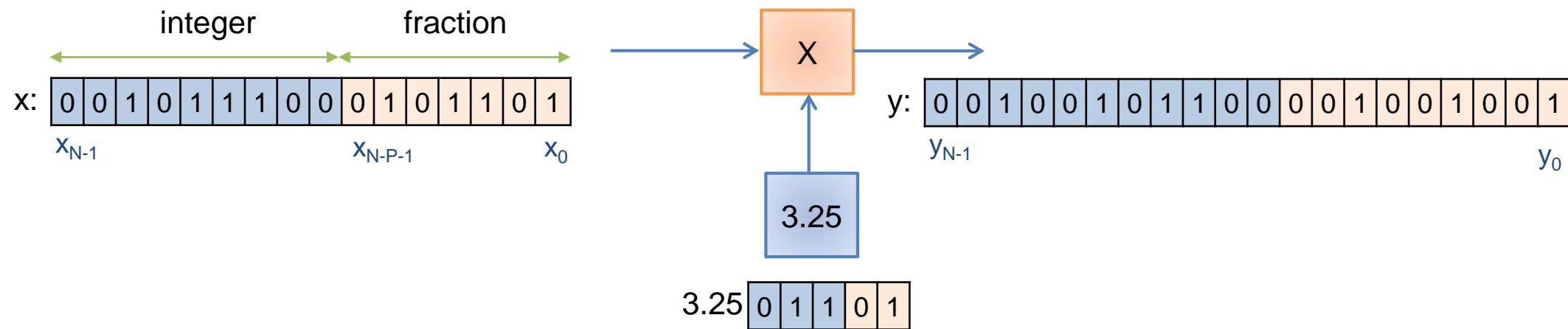
$[3.7 + 0.5] = 4;$
 $[3.2 + 0.5] = 3;$

Mixed-Precision Multiplication Examples

Example 1: Multiplication by constant powers of two: no multiplication is required; only the radix point convention changes; no error increase

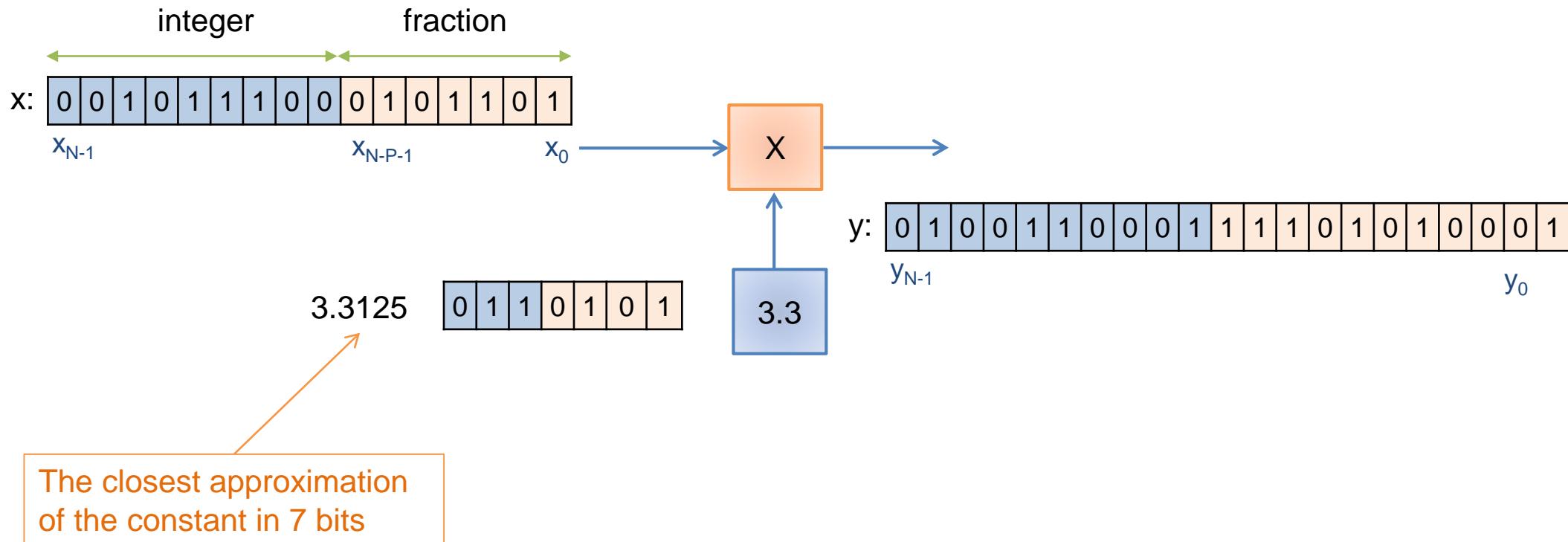


Example 2: Multiplication by constant non powers of two: multiplication is required; the radix point and register length may change; error might be added due to output truncation



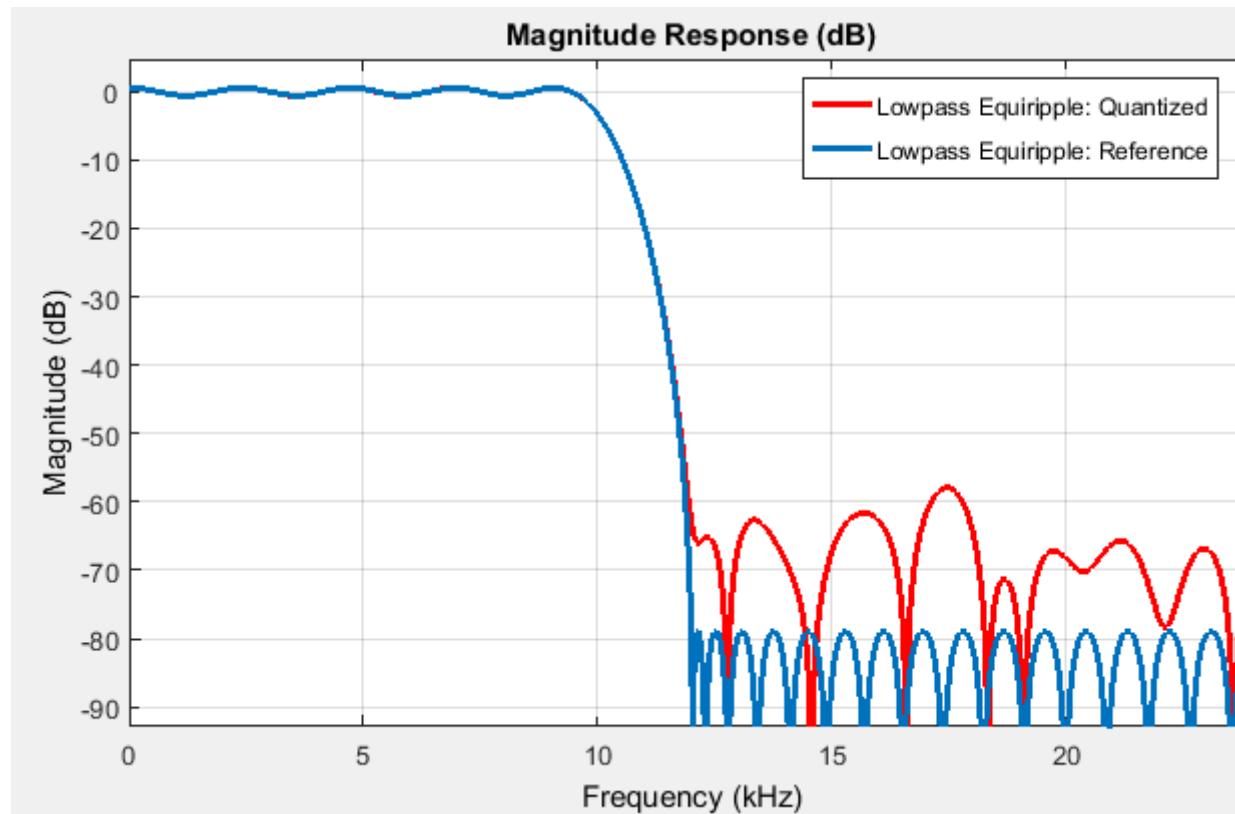
Mixed-Precision Multiplication Examples

Example 3: Multiplication by fractional non powers of two that can not be represented by sum of powers of two: Unavoidable representation error, even before multiplication



Mixed-Precision Multiplication Examples

Rounding/truncating the coefficients in data/signal processing systems can change the nominal performance of the system. For example, in filter design:

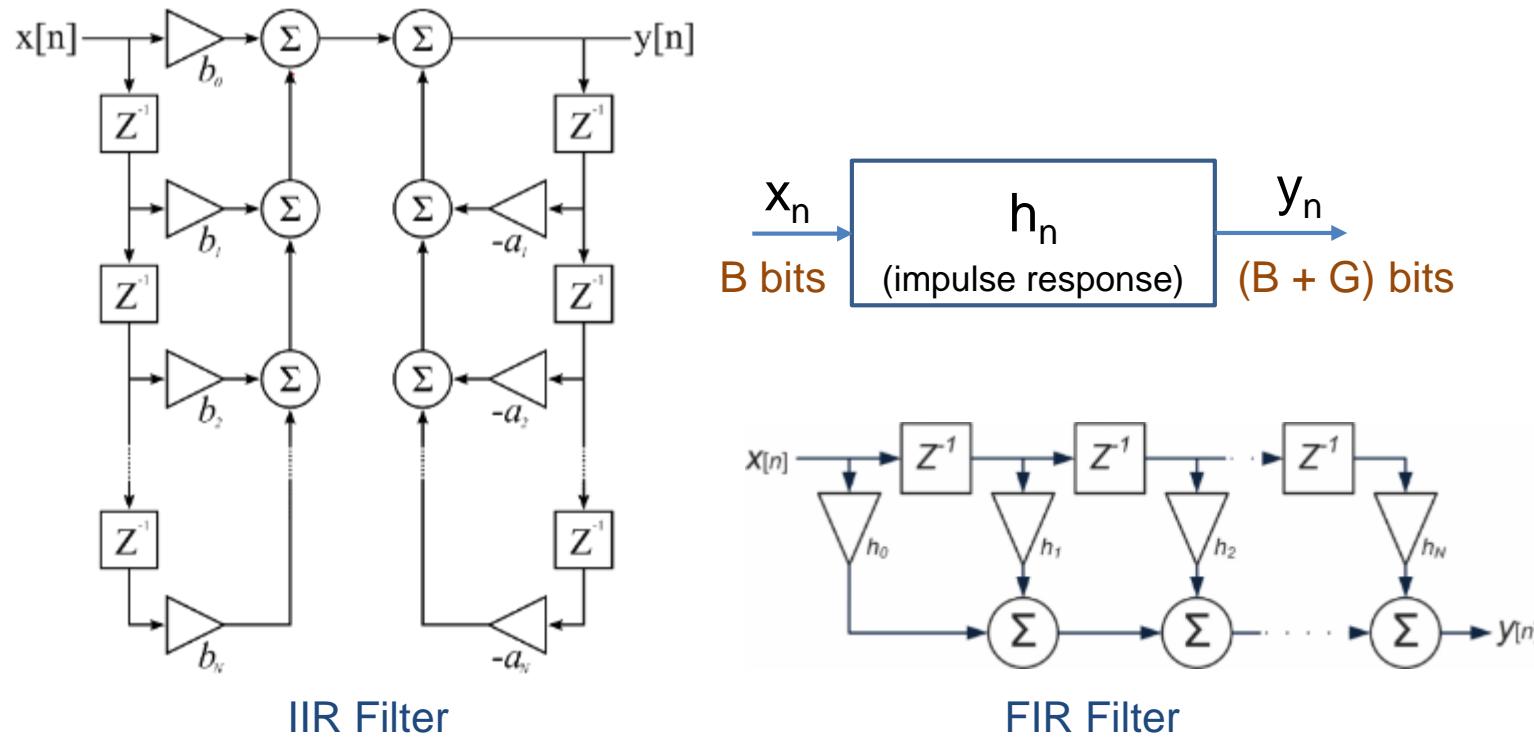


Sample lowpass filter designed in Matlab FDATool in double precision floating-point (blue) and after quantization with 12-bit fixed-point (red)

Mixed-Precision in Digital Filters

Example 4: Discrete-time convolution $y_n = x_n * h_n = \sum_m h_m x_{n-m}$: The maximum bit growth in the output is equal to the length of the filter coefficients L1–Norm:

$$G = \left\lceil \log_2 \left(\sum_m |h_m| \right) \right\rceil$$



Note: From Signals & Systems Theory we know that for a stable causal filter $\sum_m |h_m| = B < \infty$. Therefore “the output of a stable filter with a bounded input can always be stored in a register of finite length without overflow”

Further Notes on Fixed-Point and Mixed-Precision

- Note 1:** The radix point does not necessarily need to be within the range of the register length. **Example:** An 8-bit register can be used to represent fixed-point numbers with a decimal point below the LSB or above the MSB. For example, the following are legitimate fixed-point numbers, even though the register length is only 8 bits:

implied but not stored with the number

0	1	1	0	1	0	1	0
---	---	---	---	---	---	---	---

$$x = 2^{+15} \cdot (2^1 + 2^3 + 2^5 + 2^6)$$

1	0	1	0	1	0	1	1
---	---	---	---	---	---	---	---

$$x = 2^{-12} \cdot (2^0 + 2^1 + 2^3 + 2^5 + 2^6 - 2^7)$$

- Note 2:** In practice, fixed or floating-point numbers can have an arbitrary and implicit scaling factor, which is known to the designer; but is not coded or stored with the number. These scaling factors are only incorporated when numbers are mapped to their corresponding physical values (voltage, temperature, current, etc.) for user visualization or analysis. **Example:** Uniform analog-to-digital convertors map their input voltage to the output code with a constant scaling factor, which is known by the designer; but does not affect internal FPGA calculations.

Coefficient Scaling and Rounding

- In order to store real numbers in finite-length registers (fixed or floating-point), the numbers should be multiplied by appropriate **scaling factors** and rounded/truncated to fit in the registers.

Examples:

$$y_{\text{fixed}} = \text{round}(2^{16} \times y_{\text{real}})$$
$$y_{\text{fixed}} = \text{round}(3.14 \times y_{\text{real}})$$

- When scaling a set of coefficients (time-series, filter coefficients, etc.) to fit in N bits, the optimal performance (with minimum quantization error) is obtained when the maximum/minimum scaled values are equal to the maximum/minimum possible numbers (-2^{N-1} and $2^{N-1}-1$).
- Therefore, the optimal scaling factor is not necessarily a power of two (e.g., see Matlab FDAtool's quantization and scaling options)

Bit-Growth in Digital Filter Implementation*

(optional)

In digital filter implementation, the **L1–Norm bit growth** $G = \lceil \log_2(\sum_m |h_m|) \rceil$ is the worst-case (most pessimistic), which does not make any assumptions on the input signal. This formula can be relaxed (approximated) in some cases.

1. **Instantaneously narrow-band signals:** For signals having a dominant frequency peak at each time instant:

$$x_n = A \cos(\omega_0 n + \theta) \rightarrow y_n \approx |H(e^{j\omega_0})| A \cos(\omega_0 n + \varphi_{\omega_0})$$

$$\text{Bit Growth } G_0 = \left\lceil \log_2 \left(\max_{-\pi \leq \omega < \pi} |H(e^{j\omega})| \right) \right\rceil$$

2. **Random input signals:** Using **Parseval's theorem**, the output variance of a filter with a random input is related to its input variance as follows:

$$\sigma_y^2 = \sigma_x^2 \sum_m |h_m|^2$$

Therefore, with the following bit-growth, the probability of overflow at a filter's output is (almost) equal to the probability of input overflow:

$$\text{Bit Growth } G_1 = \left\lceil \log_2 \left(\sqrt{\sum_m |h_m|^2} \right) \right\rceil$$

Bit-Growth in Digital Filter Implementation*

(optional)

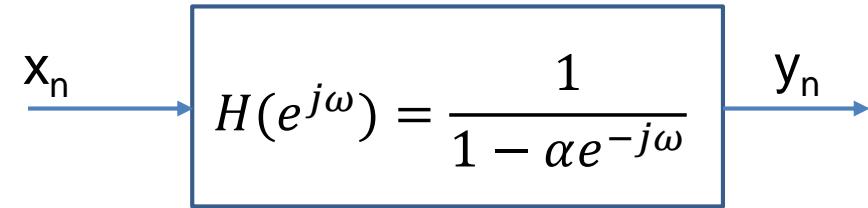
Example: A first-order lowpass IIR filter: $y_n = \alpha y_{n-1} + x_n$ ($0 < \alpha < 1$)

The impulse response is $h_n = \alpha^n u[n]$

Therefore $\sum_m |h_m| = \frac{1}{1-\alpha}$ and $\sum_m |h_m|^2 = \frac{1}{1-\alpha^2}$

Bit-growth analysis for $\alpha = 0.9$:

- L1-Norm: $G = \lceil \log_2(\sum_m |h_m|) \rceil = \left\lceil \log_2 \left(\frac{1}{1-\alpha} \right) \right\rceil = \lceil 3.3219 \rceil = 4$
- Narrow-band assumption: $G_0 = \left\lceil \log_2 \left(\frac{1}{|1-\alpha|} \right) \right\rceil = \lceil 3.3219 \rceil = 4$
- Parseval's theorem for stochastic inputs: $G_1 = \left\lceil \log_2 \left(\frac{1}{\sqrt{1-\alpha^2}} \right) \right\rceil = \lceil 1.198 \rceil = 2$

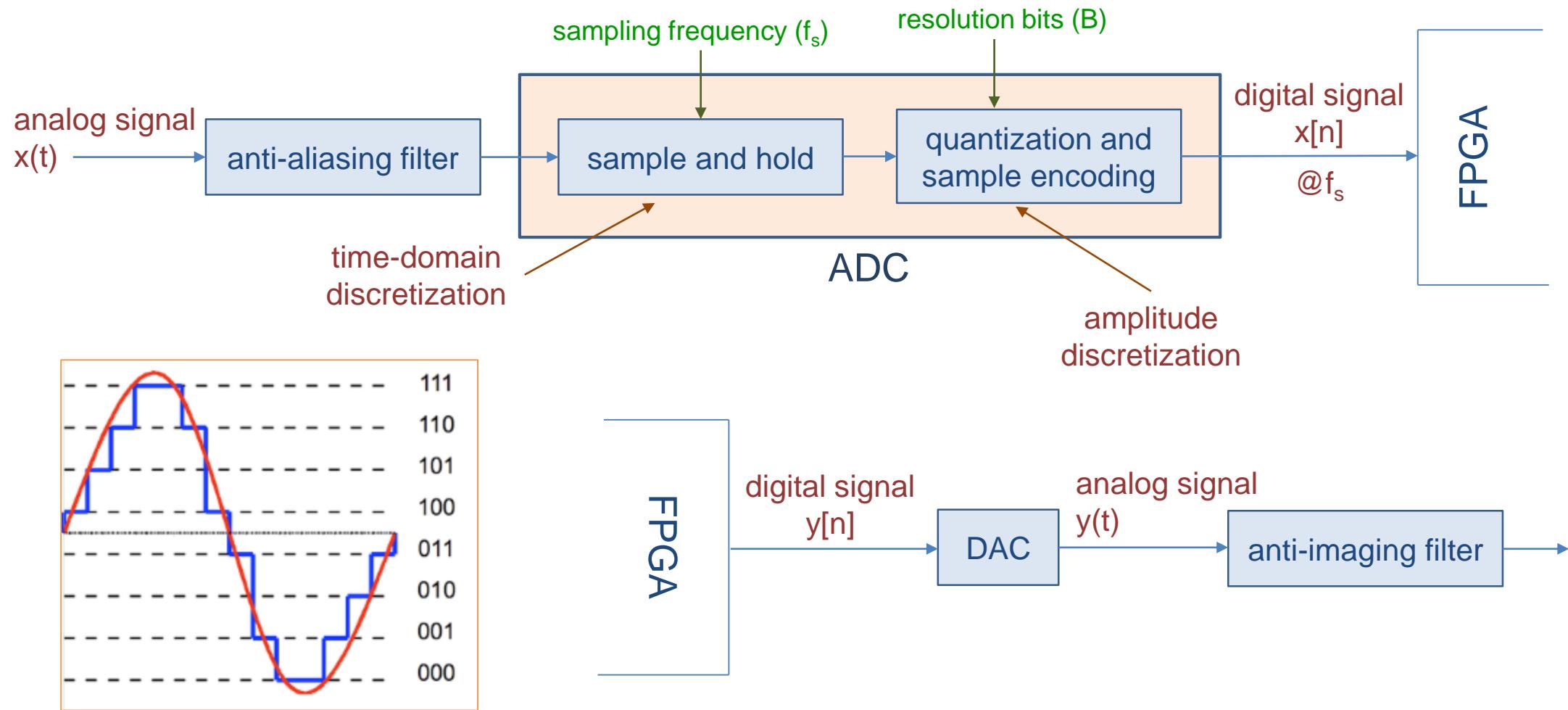


Result: In this example the **L1-norm** and **narrow-band** assumption, both demand 4 additional bits at the output y_n ; but according to the output **variance** criterion if we are fine with occasional overflows, adding only 2 bits is statistically OK.

ANALOG TO DIGITAL CONVERTORS AND DIGITAL TO ANALOG CONVERTORS

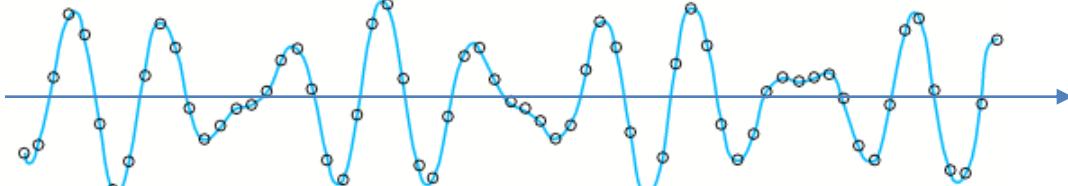
Analog to Digital Convertor (ADC) vs. Digital to Analog Convertor (DAC)

ADC and DAC are integral parts of most FPGA-based signal processing systems

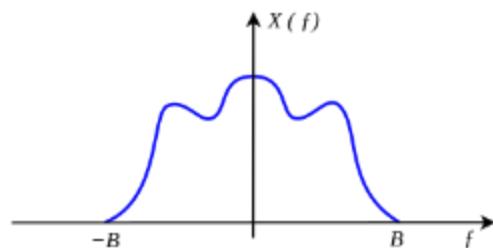


The Nyquist Rate

- The **Nyquist sampling theorem** defines the minimum number of samples acquired from a band-limited analog signal per unit time, in order to guarantee the reconstruction of the original signal from these samples. It requires: $f_s \geq 2B$



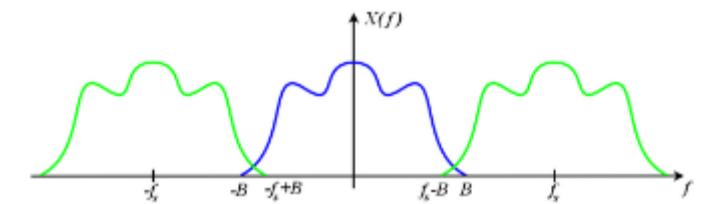
time-domain signal and its samples



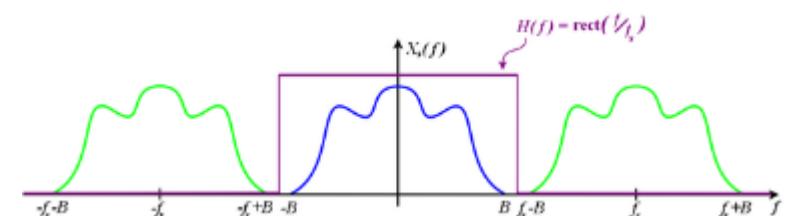
band-limited signal in the frequency domain

Ref: https://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem

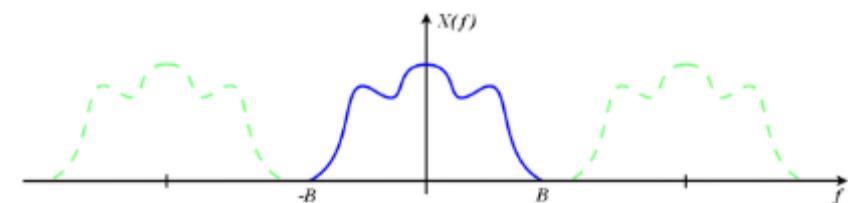
Further Reading: Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2nd Ed.)*. Prentice-Hall, Inc., 1996



After impulse train sampling with $f_s < 2B$; Nyquist rate violated



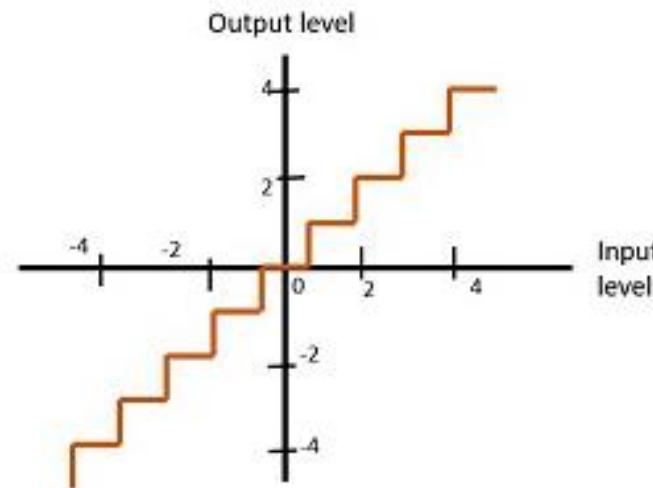
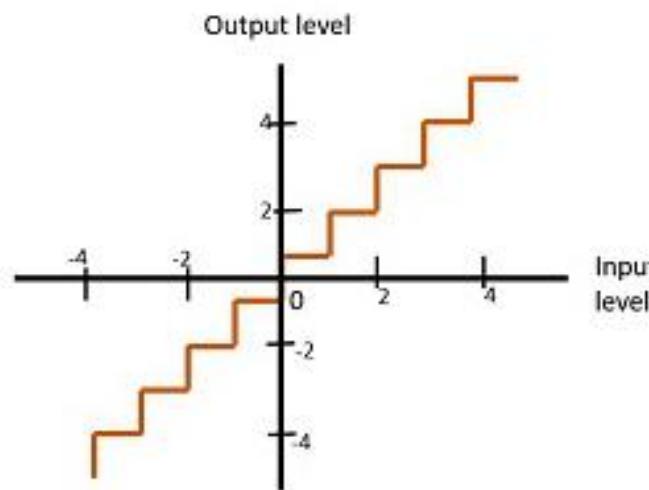
After impulse train sampling with $f_s > 2B$; Nyquist rate fulfilled



reconstructed signal

ADC Encoding Curve

- The mapping between the input voltage of an ADC and the output code can be described by an **encoding curve**.
- In a binary encoding ADC with B bits, the input voltage range $[V_{\min}, V_{\max}]$ is divided into 2^B segments and any input voltage within this range is approximated with one of the nearest voltages and represented by a code.
- The ADC encoding curve may be **uniform** or **non-uniform**.
- For example, the following are two uniform encoding curves, based on **rounding** (left) and **truncation** (right)



Question: How to quantify the performance of an ADC?

ADC Quantization Error Analysis

- The effect of ADC quantization error can be analyzed with a method similar to SNR calculation due to rounding/truncation. The quantization procedure can be modeled by a quantization operator $Q(\cdot)$:

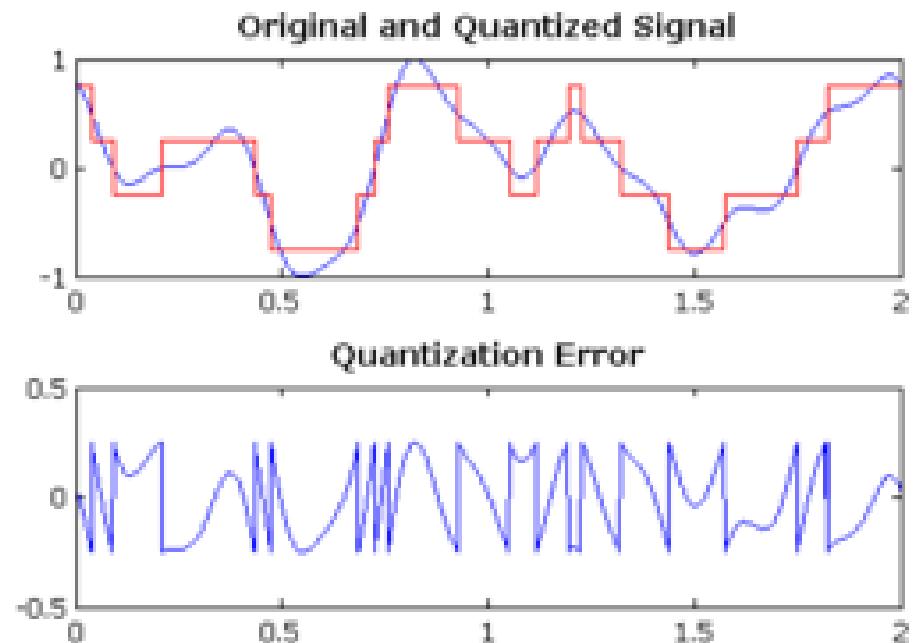
$$y_n = Q(x_n) = x_n + e_n$$

x_n : ADC input sample (after zero-order hold), y_n : quantized result, e_n : quantization error

- We again use the **signal-to-noise ratio (SNR)** as the performance measure:

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{E\{x_n^2\}}{E\{e_n^2\}} \right)$$

- This analysis requires some assumptions regarding the input signal and the quantization error **probability density functions**



ADC Quantization Error Analysis (continued)

Quantization model: $y_n = Q(x_n) = x_n + e_n$

Assumptions:

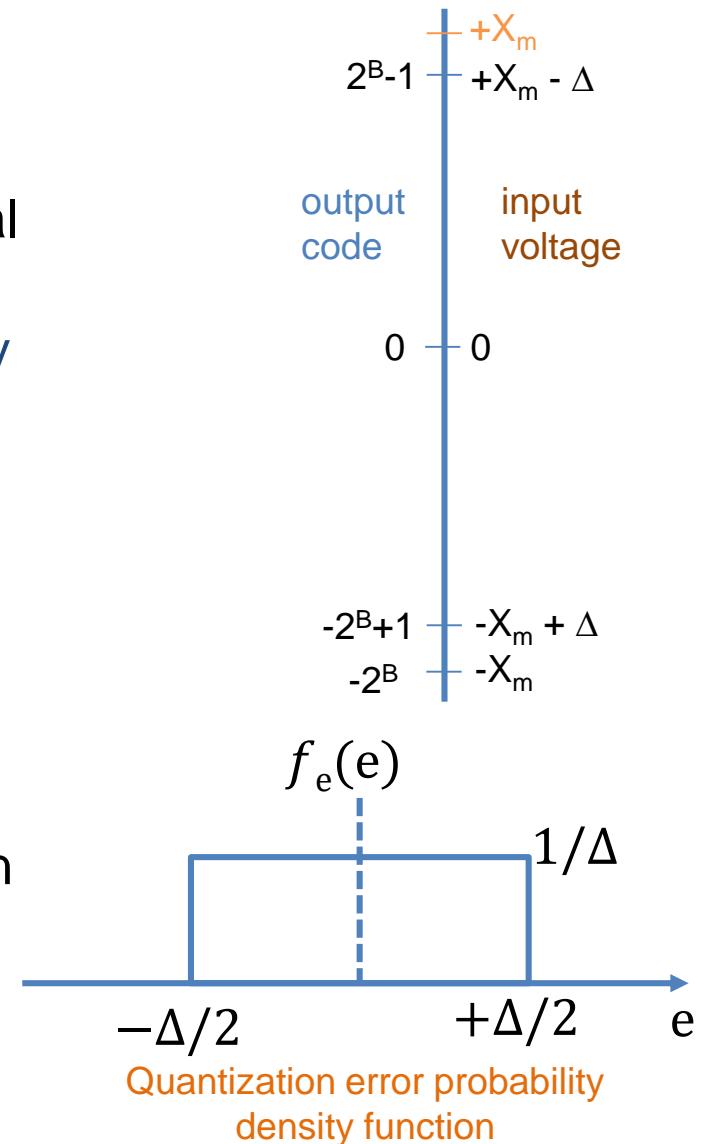
1. The signal x_n is a signed real value in $[-X_m, X_m]$
2. The quantizer is B bit and it divides $[-X_m, X_m]$ into 2^B equal segments of length $\Delta = 2X_m/2^B$
3. The signal x_n and the quantization error e_n are statistically independent (we will study the counter assumption later)
4. The quantization error samples e_n are independent identically distributed (*iid*) with a uniform distribution between $-\Delta/2$ and $\Delta/2$

Therefore: $\bar{e} = E\{e\} = \int_{-\infty}^{+\infty} e f_e(e) de = 0$

$$\sigma_e^2 = E\{(e - \bar{e})^2\} = \int_{-\infty}^{+\infty} (e - \bar{e})^2 f_e(e) de = \frac{\Delta^2}{12}$$

We have calculated the denominator of the SNR equation. In the sequel we consider three cases for the input signal:

Sinusoidal (deterministic) signal, **Gaussian distributed** stochastic signal, **Uniformly distributed** stochastic signal



ADC Quantization SNR with Sinusoidal Input

- Sinusoidal input signals are the standard measurement method for calculating ADC SNR.
- Assuming $x_n = X_m \cos(\omega n)$, we have $E\{x_n\} = 0$ and $E\{x_n^2\} = X_m^2/2$.

Therefore:

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{E\{x_n^2\}}{E\{e_n^2\}} \right) = 10 \log_{10} \left(\frac{\frac{X_m^2}{2}}{\frac{\Delta^2}{12}} \right) = 10 \log_{10} \left(\frac{\frac{X_m^2}{2}}{\frac{4X_m^2}{12 \times 2^{2B}}} \right)$$

or

$$\text{SNR}_{\text{dB}} \approx 6.02B + 1.76 \text{dB}$$

Note: This is the well-known 6dB per-bit rule, which should be memorized as a rule of thumb by any hardware engineer!

ADC Quantization SNR with Uniformly Distributed Input

- We next assume that the input signal is a stochastic random variable, uniformly distributed between $-X_m$ and X_m : $x_n \sim U(-X_m, X_m)$
- Therefore we have $E\{x_n\} = 0$ and $E\{x_n^2\} = X_m^2/3$.
Therefore:

$$\text{SNR}_{\text{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{3}}{\frac{\Delta^2}{12}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{3}}{\frac{4X_m^2}{12 \times 2^{2B}}}\right)$$

or

$$\text{SNR}_{\text{dB}} \approx 6.02B$$

Note: The 1.76dB is no longer there, but we still see the 6dB per-bit property.

ADC Quantization SNR with Gaussian Distributed Input

- We finally assume that the input signal is a stochastic random variable, with a Gaussian distribution $x_n \sim N(0, \sigma_x^2)$.
- The Gaussian distribution has infinite tails and overflow at the ADC input is unavoidable. However, the probability of overflow is reduced by controlling the input variance σ_x^2 relative to the ADC reference voltages $-X_m$ and X_m .
- Let's assume $X_m = k\sigma_x$. According to the Gaussian curve, for $k = 1, 2, 3$, and 4 , the probability of ADC input overflow is 31.73% , 4.55% , 0.26% , and 0.01% , respectively.
- Assuming $k = 4$, we have $E\{x_n\} = 0$ and $E\{x_n^2\} = X_m^2/16$. Therefore:

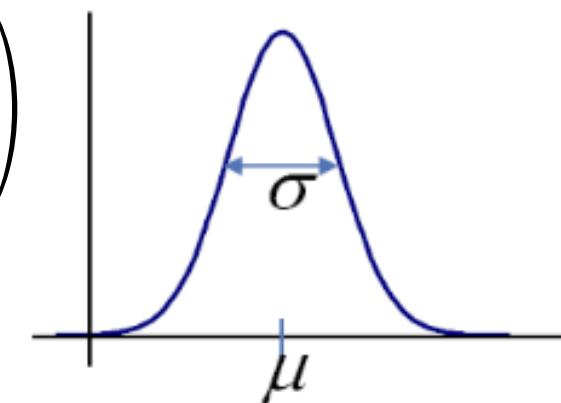
$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{E\{x_n^2\}}{E\{e_n^2\}} \right) = 10 \log_{10} \left(\frac{\frac{X_m^2}{16}}{\frac{\Delta^2}{12}} \right) = 10 \log_{10} \left(\frac{\frac{X_m^2}{16}}{\frac{4X_m^2}{12 \times 2^{2B}}} \right)$$

or

$$\text{SNR}_{\text{dB}} \approx 6.02B - 7.27 \text{dB}$$

Note: We still see the 6dB per-bit property.

Note: ADC ICs commonly have an out-of-range (OTR) pin for reporting input overflow per-sample



Non-ideal ADC

- Practical ADC circuitry are never ideal and do not reach their nominal performance ($\text{SNR} = 6.02B + 1.76\text{dB}$).
- The standard approach to measure the true performance of an ADC is by giving it a sinusoidal input signal with an amplitude of 1dB below full-scale (to avoid overflow) and measuring the real SNR and the **effective number of bits (ENOB)**:

True SNR measured by giving a full dynamic-range sinusoidal to the ADC and measuring the SNR of an acquired block of data

$$\text{ENOB} = \frac{\text{SNR}_{\text{dB}} - 1.76\text{dB}}{6.02}$$

The effective number of bits; a real-value, always smaller than the nominal number of ADC bits ($\text{ENOB} < B$)

ENOB Examples

- AD9246 14-Bit, 80 MSPS/105 MSPS/125 MSPS, 1.8 V Analog-to-Digital Converter:

Parameter ¹	Temp	AD9246BCPZ-80			AD9246BCPZ-105			AD9246BCPZ-125			Unit
		Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	
SIGNAL-TO-NOISE-RATIO (SNR)	$f_{IN} = 2.4 \text{ MHz}$ $f_{IN} = 70 \text{ MHz}$ $f_{IN} = 100 \text{ MHz}$ $f_{IN} = 170 \text{ MHz}$	25°C	71.9		71.9			71.9			dBc
		25°C	71.9		71.9			71.7			dBc
		Full	70.8		69.5			69.5			dBc
		25°C	71.6		71.6			71.6			dBc
		25°C	70.9		70.9			70.8			dBc
SIGNAL-TO-NOISE AND DISTORTION (SINAD)	$f_{IN} = 2.4 \text{ MHz}$ $f_{IN} = 70 \text{ MHz}$ $f_{IN} = 100 \text{ MHz}$ $f_{IN} = 170 \text{ MHz}$	25°C	71.1		71.1			71.1			dBc
		25°C	71.5		70.8			70.6			dBc
		Full	70.4		68.5			68.5			dBc
		25°C	70.6		70.6			70.6			dBc
		25°C	69.9		69.9			69.8			dBc
EFFECTIVE NUMBER OF BITS (ENOB)	$f_{IN} = 2.4 \text{ MHz}$ $f_{IN} = 70 \text{ MHz}$ $f_{IN} = 100 \text{ MHz}$ $f_{IN} = 170 \text{ MHz}$	25°C	11.7		11.7			11.7			Bits
		25°C	11.6		11.6			11.6			Bits
		25°C	11.6		11.6			11.6			Bits
		25°C	11.5		11.5			11.5			Bits

Non-uniform ADC Encoding Curves

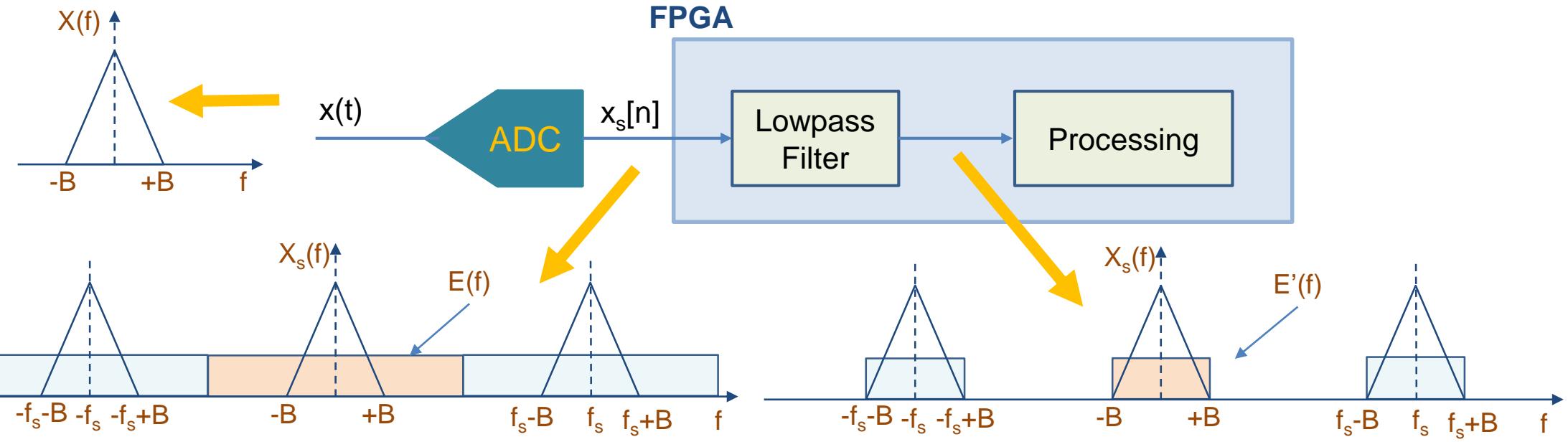
- Intuitively, in ADC with non-uniformly distributed inputs, many bits are “wasted” for **low-probability** samples (there are profound theoretical proofs behind this intuition). One could use the bits more efficiently by:
 1. **Using non-uniform ADC encoding curves:** Divide $-X_m$ and X_m into unequal segments (assign smaller segments to higher probabilities and larger segments to lower probability values). **Example:** A-law and μ -law companding algorithms used in old 8-bit PCM digital communication systems for better use of the dynamic range
 2. **Making the input sequence distribution uniform:** A useful theorem from random variables:

If a **random variable** (RV) x with a probability density function (pdf) $f_x(x)$ and cumulative distribution function (CDF) $F_x(x)$ passes a nonlinear memoryless system with a characteristics $u = F_x(x)$, the output u is **uniformly distributed**. Also, if a uniformly distributed RV u is given to $y = F_x^{-1}(u)$, the output has a distribution $f_x(\cdot)$.

Note: This property can be used to make arbitrary RVs from uniform distributions and vice versa in FPGA.

ADC SNR Improvement by Over-Sampling

- Looking back at the quantization model $y_n = Q(x_n) = x_n + e_n$, the quantization error samples e_n were assumed to be **independent identically distributed (iid)**. Therefore, the quantization noise has a white spectrum and its total power $E\{e_n^2\}$ is equally distributed over the entire **Nyquist-band** $[0, f_s]$.
- If the signal is over-sampled beyond the Nyquist rate, the ADC SNR can be improved by lowpass filtering the ADC outputs (in the digital domain).
- In this case, we have: $\text{SNR}_{\text{dB}} \approx 6.02B + 1.76\text{dB} + 10\log_{10}(\text{OSR})$, where OSR is the **over-sampling ration** ($f_s/2B$)



ADC SNR Improvement by Over-Sampling

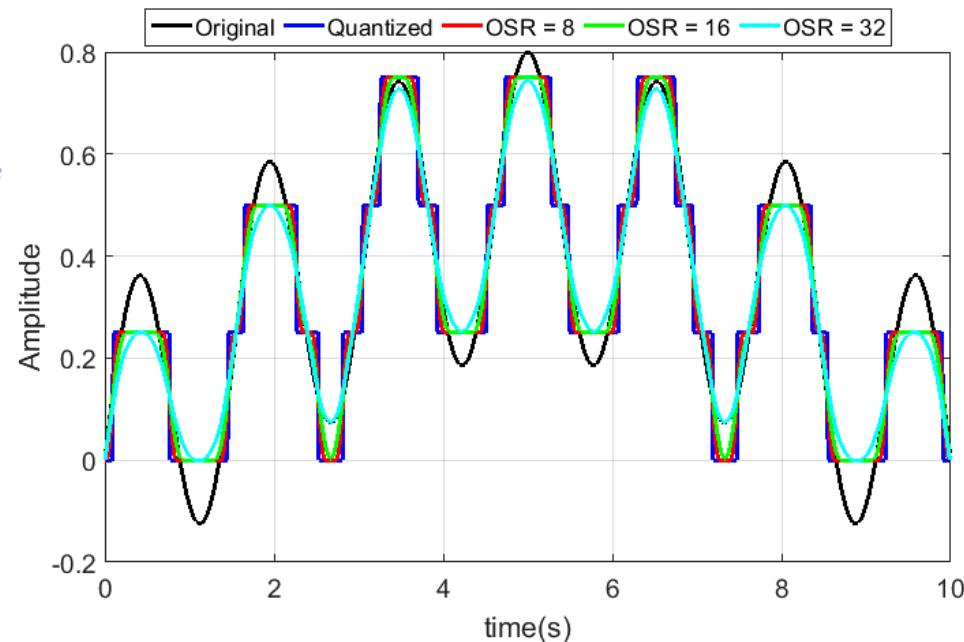
(continued)

- **Over-sampling rule of thumb:** “Each factor of two above the Nyquist rate, is equivalent to 3dB of SNR improvement (after low-pass filtering)”. Therefore, SNR improvement by OSR is expensive!
 - **Question:** OSR = 4 improves the SNR for 6dB, equivalent to 1 bit of higher resolution. Does this mean that we can have a **mono-bit ADC** that is equivalent to a 12-bit ADC?!
- Answer:** Yes (to some extent)!

```

7 % Signal simulation
8 N = 1000;
9 n = 0 : N-1;
10 fs = 100; % analog signal simulation
11 t = n/fs;
12 f1 = 0.05;      f2 = 0.65;
13 x = .5*sin(2*pi*f1*t) + 0.3*sin(2*pi*f2*t) + 1e-4*randn(size(t));
14 %(The Nyquist rate is twice the maximum frequency)
15
16 % Amplitude quantization
17 B = 3; % number of quantization bits
18 y = double(fi(x, 1, B));
19
20 % Signal recovery using over-sampling ratio (OSR)
21 OSR1 = 8;
22 OSR2 = 16;
23 OSR3 = 32;
24 y1_smoothed = filtfilt(ones(1, OSR1), OSR1, y);
25 y2_smoothed = filtfilt(ones(1, OSR2), OSR2, y);
26 y3_smoothed = filtfilt(ones(1, OSR3), OSR3, y);

```

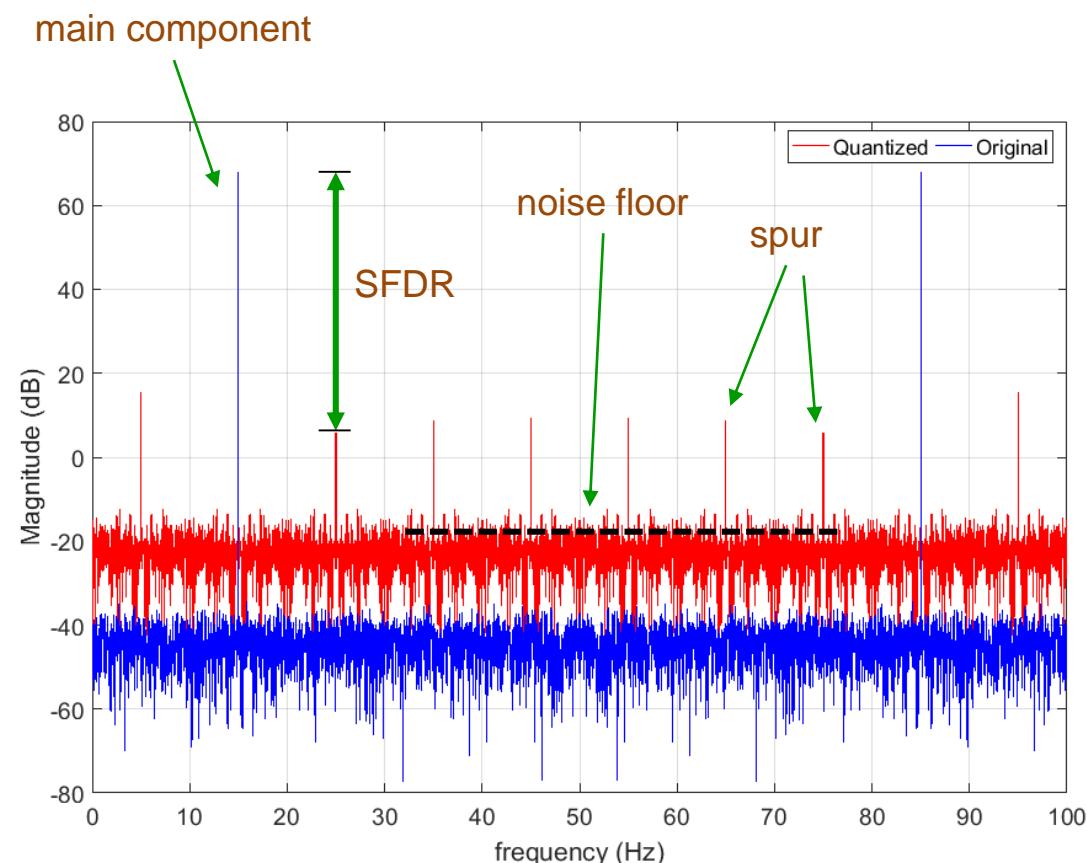


Spurious-Free Dynamic Range (SFDR)

- Looking back at the quantization model $y_n = Q(x_n) = x_n + e_n$, the quantization error e_n was assumed to be independent from x_n . However, this assumption is violated in low number of bits.

Spurs are notable components and spikes of noise within a signal's spectrum and above the noise floor, which do not correspond to the original signal; but are somehow correlated with it (they move in the spectrum as the sampling frequency changes or as the signal components move).

SFDR is the gap (in dB) between the original frequency component and the strongest spur



Spurious-Free Dynamic Range Improvement

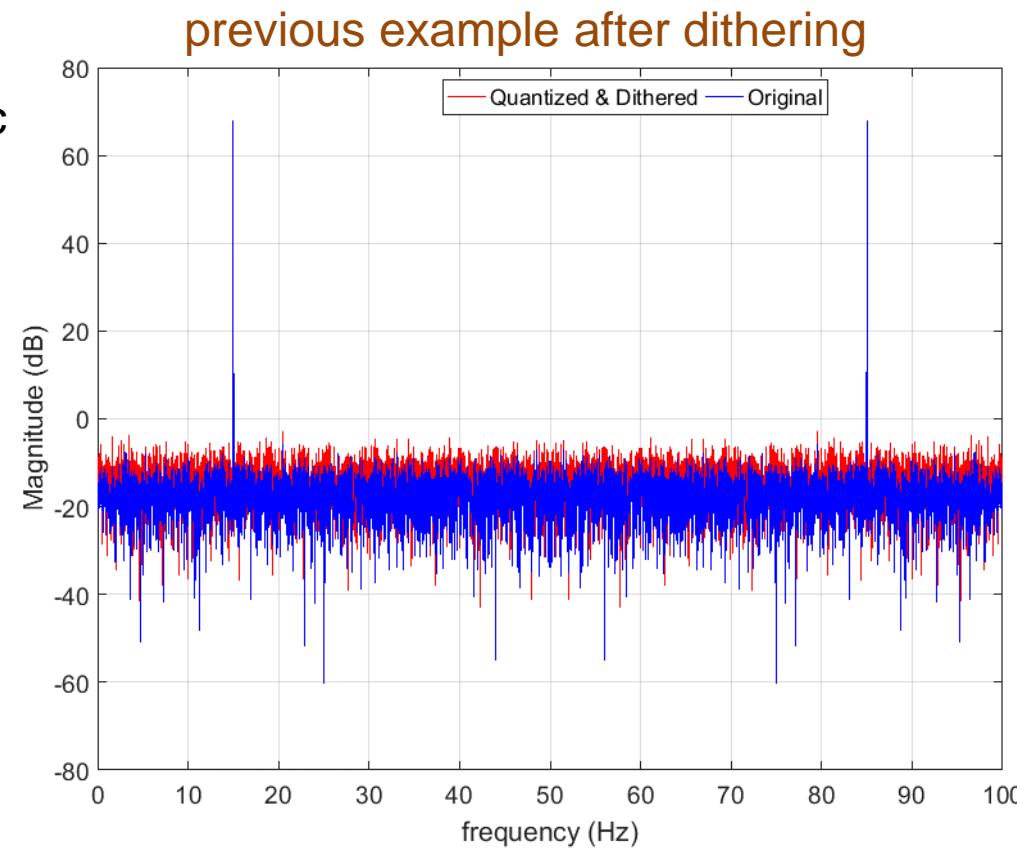
Spurs are very important in practice, as they are commonly mistaken with the original signal components.

Note: Spurs can also occur during FPGA arithmetic truncation/rounding

How to improve the SFDR?

1. Increase the number of ADC (quantization) bits
2. Break the correlation between the signal and quantization (rounding/truncation) errors by adding dithers prior to quantization (rounding/truncation), e.g., by using high-thermal noise resistors in ADC inputs

Dither is a noise (at the level of the signal's LSB) intentionally added to the signal before quantization to de-correlate the signal and quantization noise



Note: Dithering improves the SFDR at a cost of decreasing the SNR (increasing the noise floor)

Note: Dithers can be generated in FPGA using linear-feedback shift registers (LFSR)

Further Reading on ADC and DAC* (Optional)

- ADC internal technologies: ladder, flash, delta-sigma modulation
- Integral nonlinearity (INL)
- Clock jitter
- DAC technologies
- Contemporary FPGAs with built-in ADCs
- Quadrature ADC sampling techniques (for high speed)
- Mono-bit technologies
- ADC/DAC tradeoffs

Further reading: refer to the references on ADC/DAC in the course's references folder

WORD LENGTH SELECTION IN FPGA-BASED ARITHMETIC

Background

- Real-world applications require the representation of real-valued data in floating-point or fixed-point formats
- Real numbers can be approximated in these formats using **the necessary number of bits** and by proper **scaling**

Question 1: How many bits should be used for internal calculations?

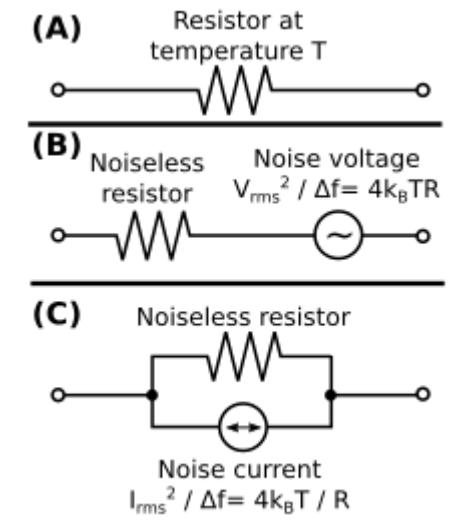
Answer: Considering that coefficient quantization and rounding/truncation introduce additional errors to the input data, the internal register lengths are selected to meet **the minimum required SNR** (selected by the designer)

Question 2: How to choose the minimum required SNR?

Answer: It is application-dependent

Word-Length Selection in FPGA Designs

- The most common sources of noise in analog and digital electronics systems are
 - Thermal noise of electronic devices and elements
 - Quantization errors in digital systems, due to number representation in finite-length registers and rounding/truncation
- In mixed analog digital designs (containing analog elements, ADC, DAC, FPGA, processors, etc.) the conventional standard is to keep the fixed-point computational errors at the same level or below the input analog noise level



Thermal noise model
of a resistor

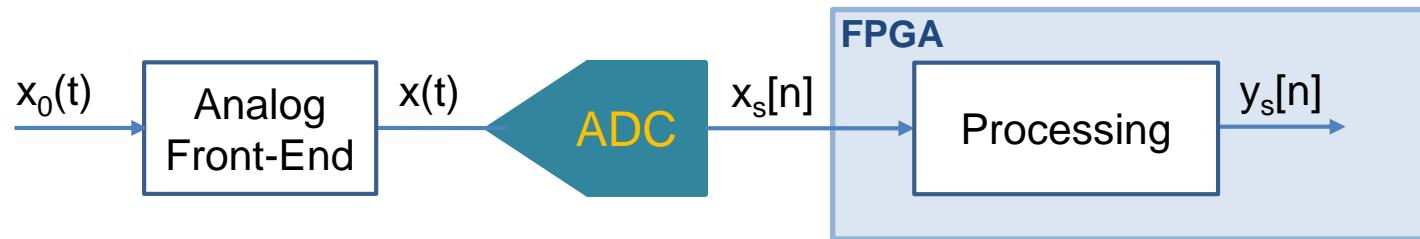
Input Word-Length Selection Procedure

How to determine the input noise level and internal register lengths?

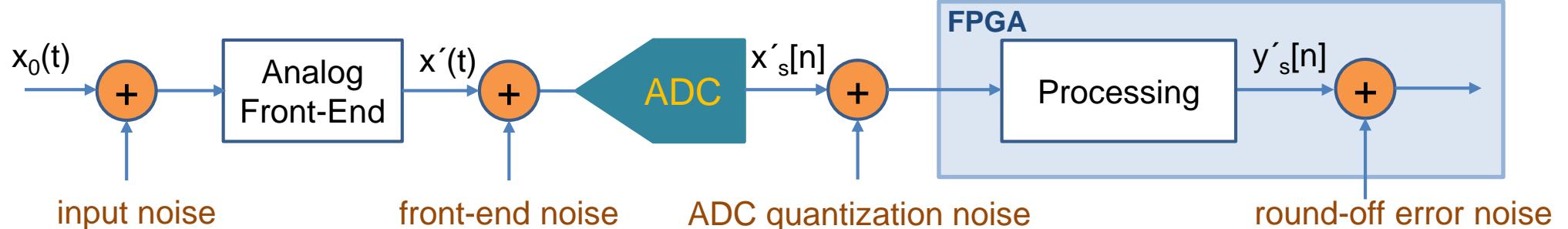
1. Thermal noise (noise figure) calculation of all analog elements, up to the digital units (beyond the scope of this course)
2. Calculating the ENOB of the ADC
3. Selecting the processing register lengths such that the internal FPGA quantization errors are below (or at the same level as) the above items

Note: For pure digital processing or when the input noise level is unknown for the digital designer, the noise level can be assumed to be half the input register LSB

Ideal System:



Real System:



Input Word-Length Selection Procedure (continued)

- Note: As far as the FPGA designer is concerned, the input noise and the analog front-end noise can usually be lumped in the ADC quantization noise (as factors that reduce the input ENOB)
- For example, with a 16-bit ADC, the 3 LSB may fluctuate due to the different noise factors (input noise, device thermal noise, ADC quantization error)

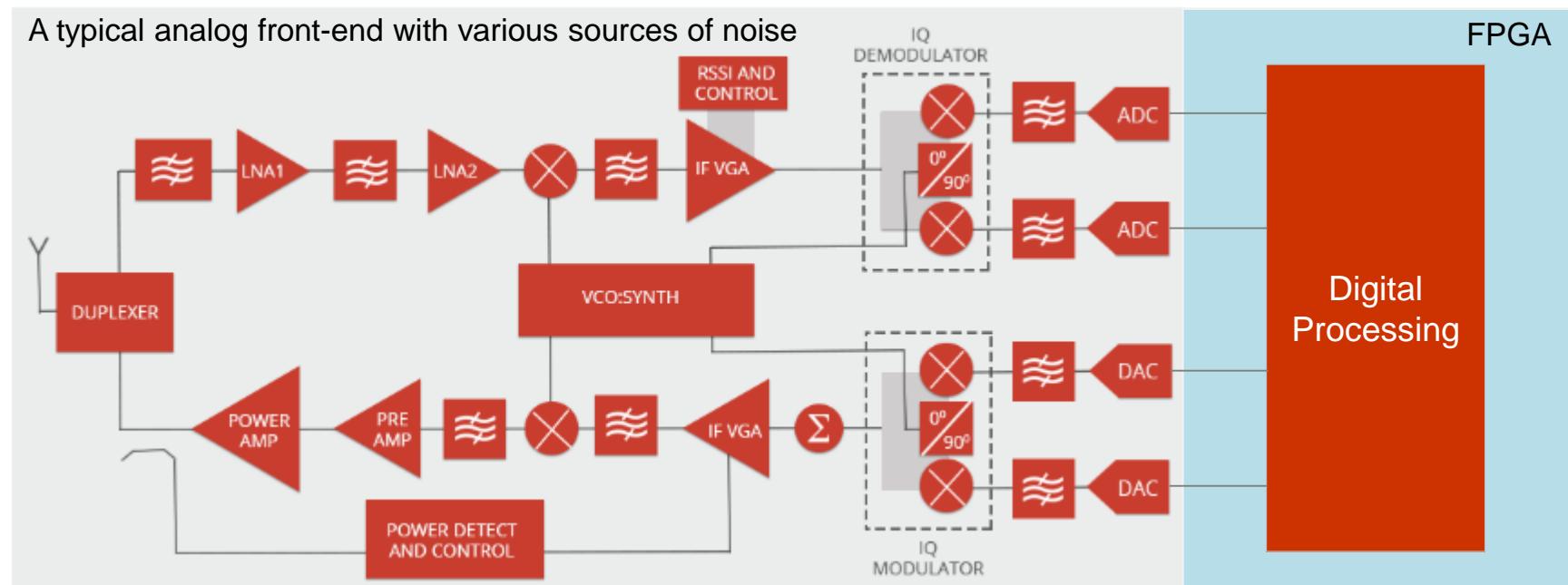
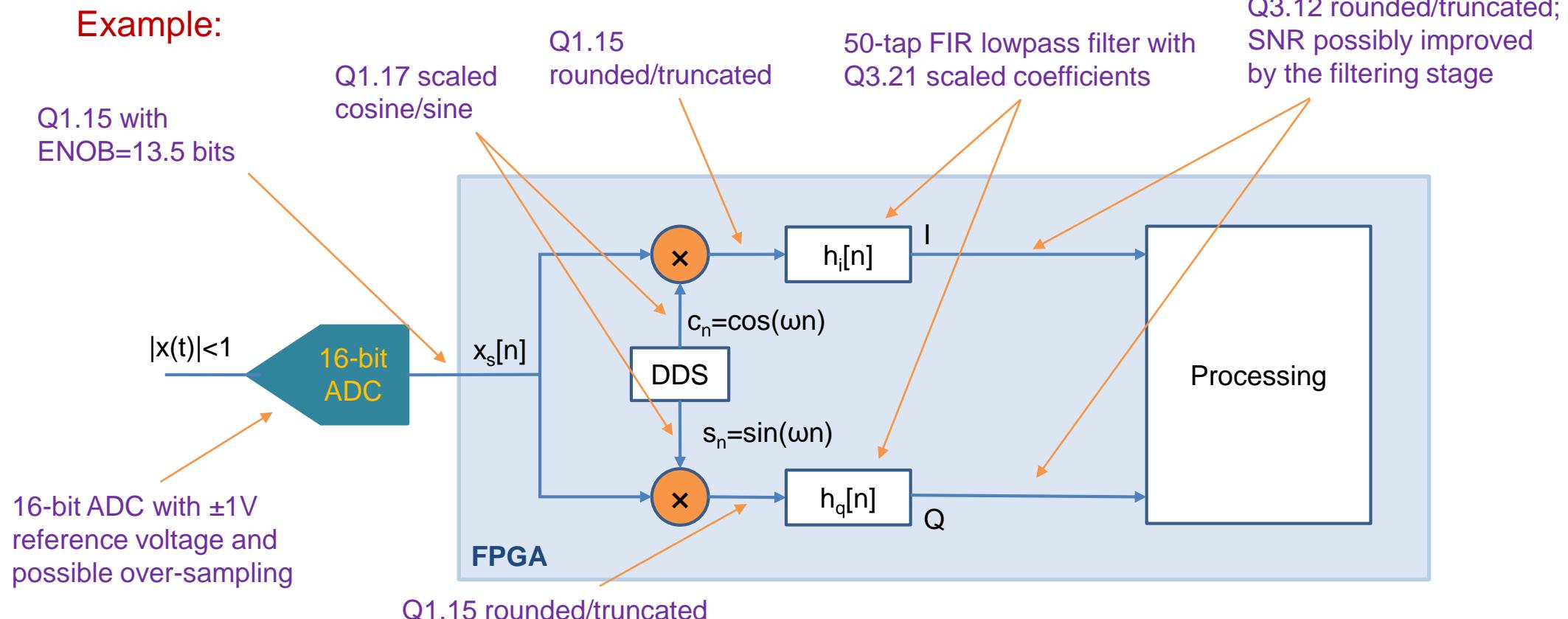


Image adapted from: <http://www.azcom.it/index.php/services/rf-design/analog-front-end-afe/>

Intermediate Word-Length Selection in FPGA Designs

Intermediate calculation word-length selection follows similar rules: “try to preserve the signal-to-noise ratio during calculations, as much as possible”

Example:



Note 1: The internal register lengths are selected according to the input noise level and ENOB, not the ADC number of bits

Note 2: The SNR can be increased due to the processing gain. For example, remember the SNR improvement due to over-sampling noted in the previous section

ARBITRARY WAVEFORM GENERATION

Waveform Generation

The calculation/generation of arbitrary functions/waveforms of the form $y = f(x)$ is required in many computational and signal processing applications. We study several methods for this purpose:

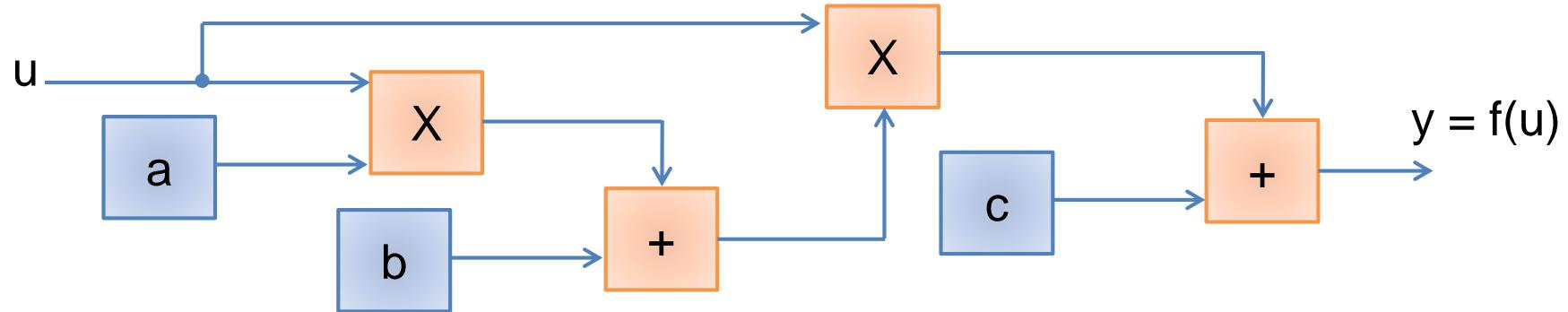
- **Arbitrary functions:**
 - Direct Implementations (functional calculation)
 - Lookup-Tables & Interpolated Lookup-Tables
- **Special functions:**
 - CORDIC machines
- **Periodic functions:**
 - NCO and Periodic Waveform Generators
 - Recursive Oscillators
- **Random signal:**
 - LFSR

Direct Function Implementation

Depending on the function form, $y = f(u)$ can be implemented using its direct mathematical form or **truncated Taylor expansion**:

Example 1: $y = f(u) = a \cdot u^2 + b \cdot u + c = u \cdot (a \cdot u + b) + c$

Requires two multipliers and two adders



Example 2: $y = f(u) \approx f(a) + f'(a) \cdot (u - a)$

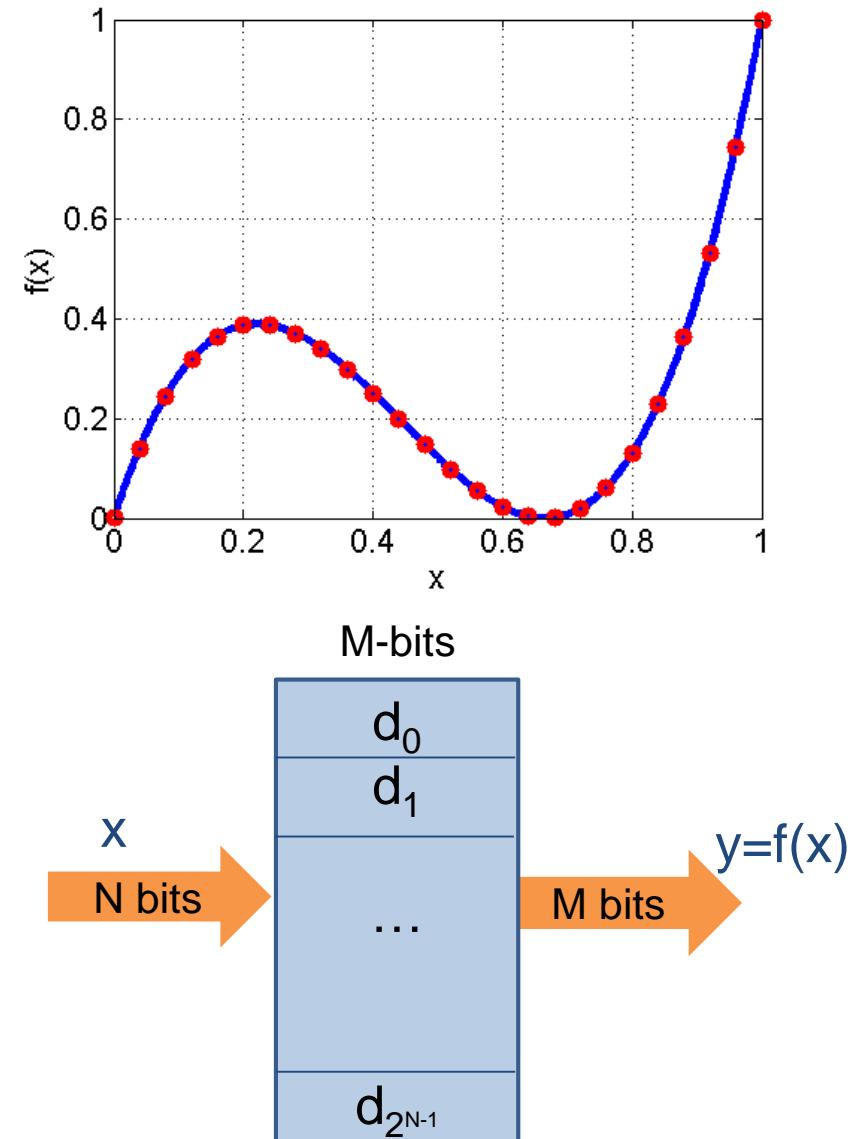
Requires a multiplier and two adders for a first-order approximation

Note 1: The implementation of the direct form of a function on FPGA is simplified when the expansion coefficients are constants or powers of 2.

Note 2: The approximated Taylor expansion is only accurate for **smooth functions**

Functional Implementation by Lookup Tables (LUT)

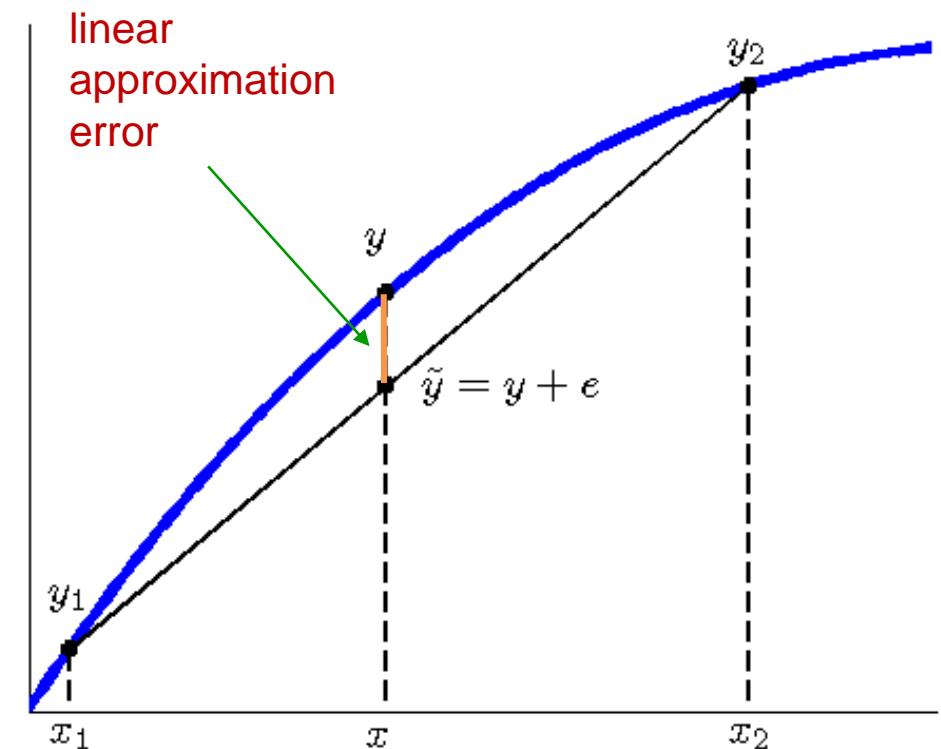
- In order to implement $y = f(x)$ over a finite domain, one may pre-calculate and store the values of y over the entire domain of x in a memory. The values of x can next be used as the address bus of the memory during runtime.
- LUT-based implementation of functions is applicable for arbitrary functions (not necessarily smooth); but requires a lot of memory when x has many bits.
- The accuracy of this method depends on the function form, and the number of bits assigned to x (N) and y (M)



Functional Implementation by Interpolated LUT

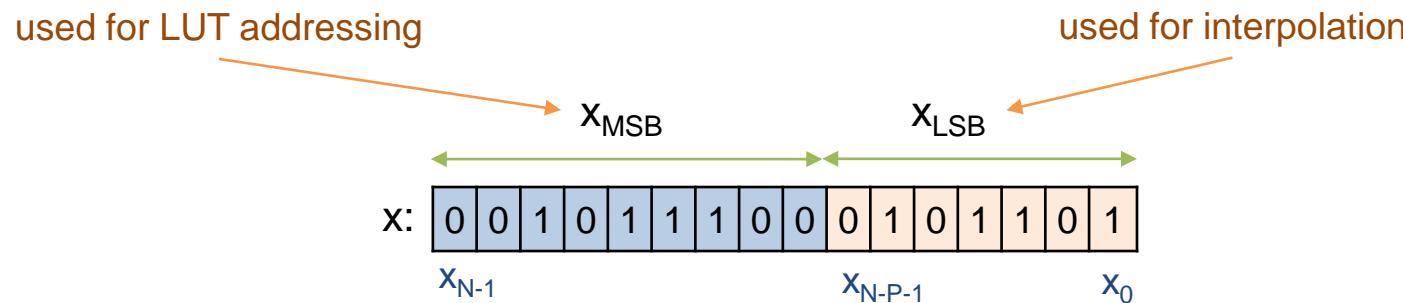
- For smooth functions, LUT-based methods can be made more memory-efficient, if they are combined with interpolation (linear, quadratic, spline, etc.)
- For example, in linear interpolation, we interpolate between successive values of the LUT with appropriate weights:

$$y \approx \frac{(x - x_1)y_2 + (x_2 - x)y_1}{(x_2 - x_1)} = y_1 + \frac{(x - x_1)}{(x_2 - x_1)} (y_2 - y_1)$$



Interpolated LUT Implementation

- Linear interpolated LUTs can be implemented very efficiently using a single or dual-port LUT and minor computations.
- Idea:** Suppose that x has N bits, which means that an LUT of length 2^N is required for its complete implementation. However, if one uses the P MBS bits of x ($P < N$) for addressing a 2^P points LUT, the $N-P$ LSB bits of x could be used for linear interpolating between two successive samples of the P -point LUT.



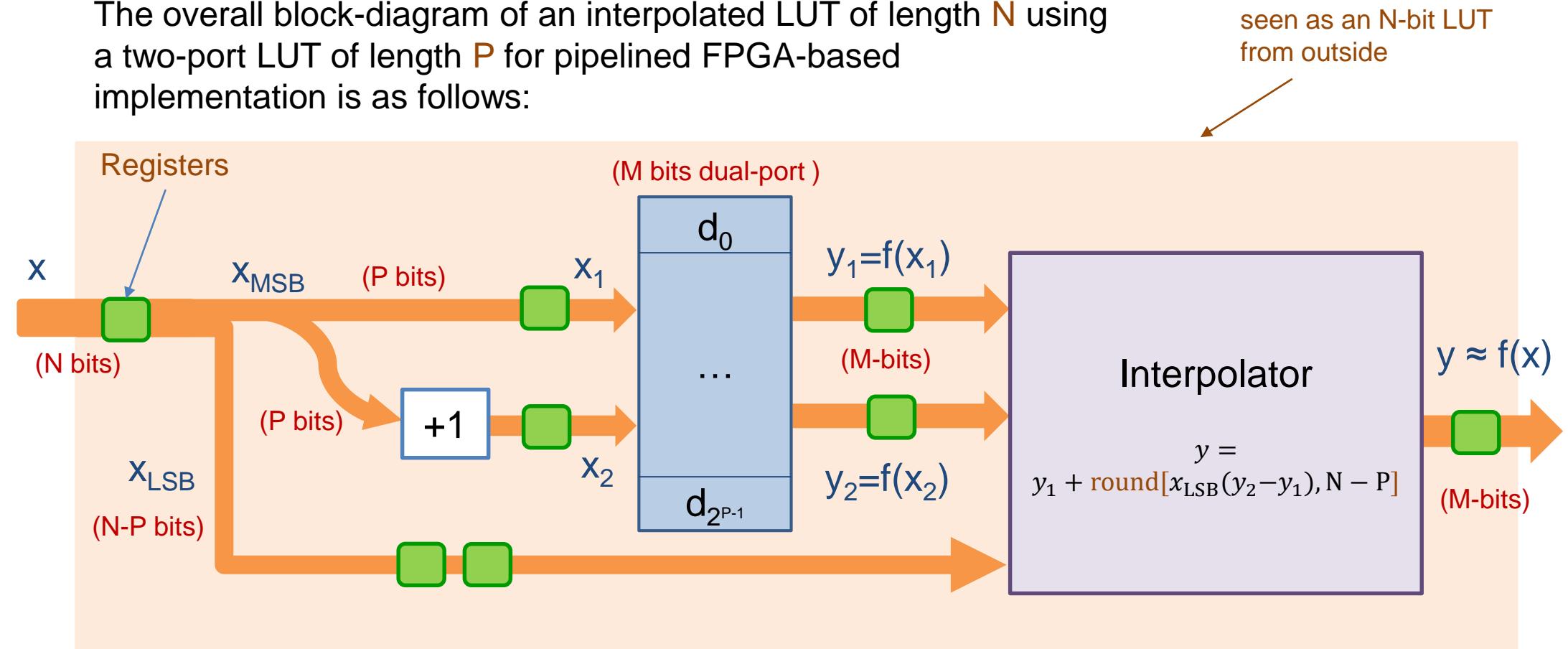
- Therefore, we can write $y_1 = f(x_{\text{MSB}})$, $y_2 = f(x_{\text{MSB}} + 1)$ and calculate the first-order interpolation as follows:

$$y = y_1 + \frac{(x - x_1)}{(x_2 - x_1)} (y_2 - y_1) = y_1 + \frac{x_{\text{LSB}} \times (y_2 - y_1)}{2^{N-P}}$$

This division requires only a shift (rounding)

Interpolated LUT Implementation Diagram

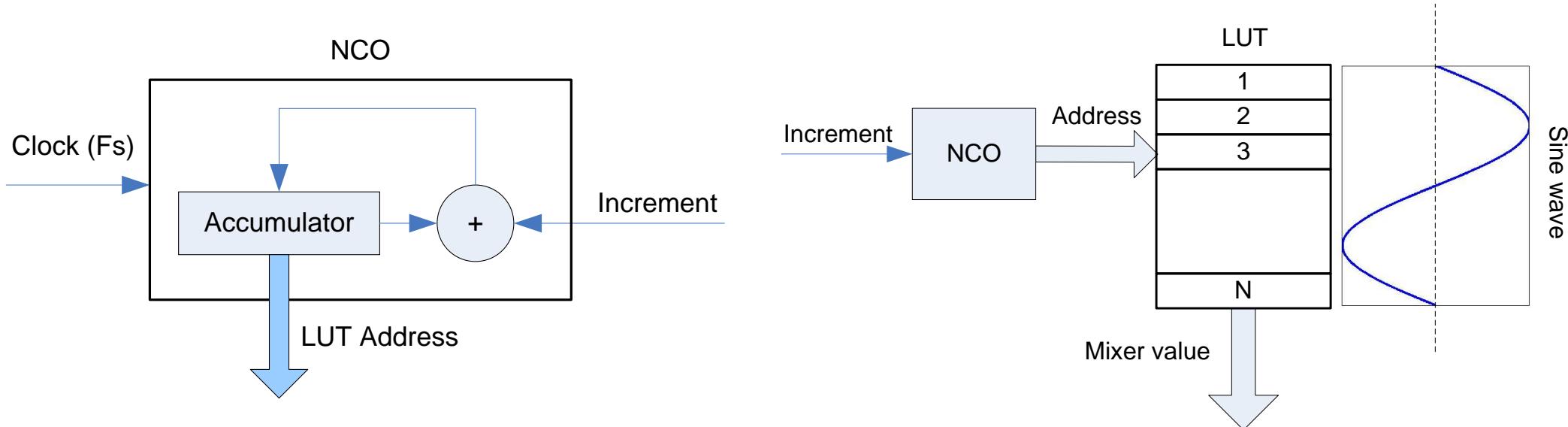
The overall block-diagram of an interpolated LUT of length N using a two-port LUT of length P for pipelined FPGA-based implementation is as follows:



Note: Similar ideas can be implemented using quadratic and spline interpolations. See the following reference for further ideas and general LUT-based methods: Behrooz, P. (2000). Computer arithmetic: Algorithms and hardware designs. *Oxford University Press*, Chapter 24

Periodic Signal Generators

An efficient method for generating periodic signals is to combine an LUT with a numerically controlled oscillator (NCO)



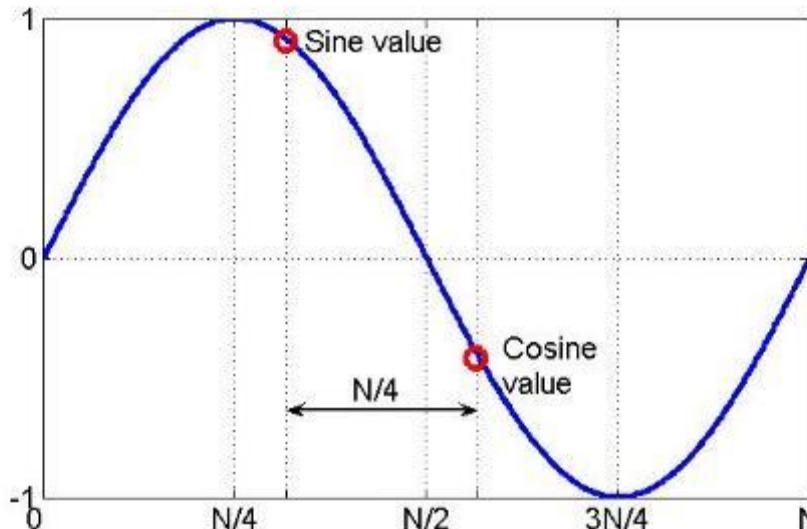
Example: In order to generate a sinusoidal signal with frequency f_0 in a sampling rate f_s , using an LUT of length N , the NCO increment can be found as follows:

$$\text{inc} = \frac{Nf_0}{f_s}$$

Note: As a sinusoidal signal, inc should be smaller than $N/2$ to fulfill the Nyquist sampling rate.

Notes on Periodic Signal Generators

1. Sine and Cosines can be produced using a single two-port LUT with $\frac{1}{4}$ of initial address offset between the two ports.



2. Sine/cosine generation is precise (with no phase errors), if the desired frequency (f_0), sampling frequency (f_s), LUT length (N) and LUT address increment (inc) satisfy:

$$\frac{f_0}{f_s} = \frac{\text{inc}}{N}$$

Sine Wave Generator Examples

- **Example 1:** We want to generate a sine wave with frequency $f_0=10.7\text{MHz}$ at a sampling rate of $f_s=38.4\text{MHz}$. Noting that $10.7\text{MHz}/38.4\text{MHz} = 107/384$, we can have a 384-point LUT with inc=107.
- **Example 2:** We want to generate a sine wave with frequency $f_0=10.7\text{MHz}$ at a sampling rate of $f_s=42.8\text{MHz}$. Noting that $10.7\text{MHz}/42.8\text{MHz} = 1/4$, we can have a 4-point LUT, which is basically a 4-state selector that circulates between 0,+1,0, and -1 (no LUT needed).
- **Example 3:** We want to make a **direct digital synthesizer (DDS)** for generating sine waves at a sampling frequency of $f_s=100\text{MHz}$. The DDS should be able to synthesize frequency from DC to 50MHz (Nyquist rate), with frequency steps of $\Delta f=100\text{kHz}$. A LUT of length $N=1000$ is required.

CORDIC Machines

- The direct implementation of arbitrary functions requires considerable **logic resources** and LUT-based methods require considerable **memory**.
- Classes of mathematical functions can be generated with a combination of small-size LUTs and set of **shifts and adds/subtracts**.
- The **Coordinate Rotation Digital Computer** (CORDIC) is one such method
- The CORDIC machine was invented in 1956 by Jack E. Volder to be used in B58 bomber's navigation system for accurate real-time digital calculations

Volder's CORDIC Algorithm

Volder's original algorithm is a set of recursive multiplier-free equations:

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n} \end{cases}$$

where

- $\arctan 2^{-n}$ are pre-calculated and stored in a LUT
- $d_n = \text{sign}(z_n)$ (+1 if $z_n \geq 0$ and -1 if $z_n < 0$)

If $|z_n| < \theta_{max} = \sum_{n=0}^{\infty} \arctan 2^{-n} = 1.7432866 \dots$, it can be shown that:

$$\lim_{n \rightarrow \infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K \times \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix}$$

where $K = \prod_{n=0}^{\infty} \sqrt{1 + 2^{-2n}} = 1.6467603 \dots$

CORDIC Machine Principles

- The non-restoring decomposition of an arbitrary angle:

$$\theta = \sum_{k=0}^{\infty} d_k w_k, d_k = \pm 1, w_k = \tan^{-1}(2^{-k})$$

The nonrestoring algorithm:

The following algorithm converges to θ :

$$t_0 = 0$$

$$t_{n+1} = t_n + d_n w_n$$

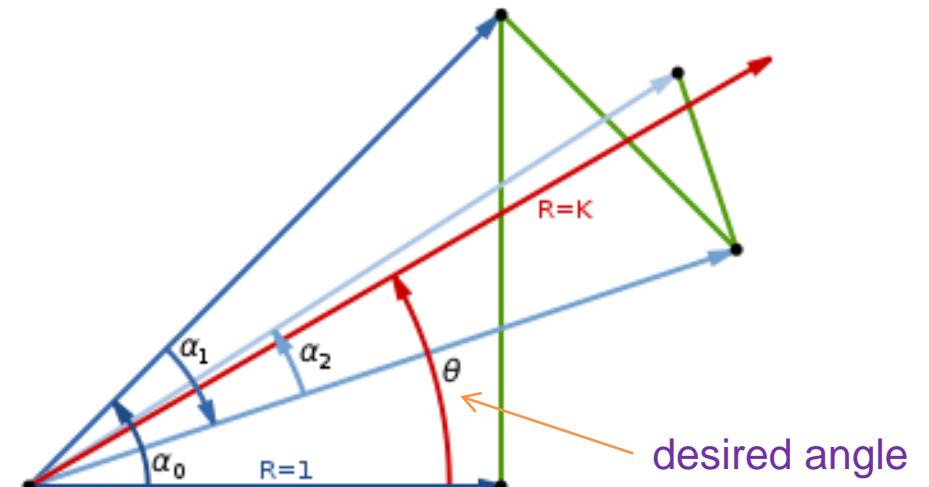
$$d_n = \begin{cases} 1 & \text{if } t_n \leq \theta \\ -1 & \text{otherwise} \end{cases}$$

Or in the reverse direction:

$$t_0 = \theta$$

$$t_{n+1} = t_n - d_n w_n$$

$$d_n = \begin{cases} 1 & \text{if } t_n \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



The CORDIC Algorithm in Circular Rotation Mode

- According to the restoring algorithm, for an arbitrary angle θ , successive rotations can be used to rotate from zero to θ (or from θ to 0):

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} \cos(d_n w_n) & -\sin(d_n w_n) \\ \sin(d_n w_n) & \cos(d_n w_n) \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

or

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \cos(w_n) \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

- The term $\cos(w_n) = 1/\sqrt{1 + 2^{-2n}}$ is the only required multiplication, which can be omitted, as it **does not alter the rotation angles** and only changes the **vector magnitudes**.
- Alternatively, depending on the number of iterations P , $A = 1 / \prod_{n=0}^P \sqrt{1 + 2^{-2n}}$ can be compensated as a constant multiplier.

Alternative Forms of the CORDIC Algorithm

- Alternative modes of the CORDIC algorithm include:

Type	m	w_k	$d_n = \text{sign}z_n$ (Rotation Mode)	$d_n = -\text{sign}y_n$ (Vectoring Mode)
circular	1	$\arctan 2^{-k}$	$x_n \rightarrow K(x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \rightarrow K(y_0 \cos z_0 + x_0 \sin z_0)$ $z_n \rightarrow 0$	$x_n \rightarrow K\sqrt{x_0^2 + y_0^2}$ $y_n \rightarrow 0$ $z_n \rightarrow z_0 + \arctan \frac{y_0}{x_0}$
linear	0	2^{-k}	$x_n \rightarrow x_0$ $y_n \rightarrow y_0 + x_0 z_0$ $z_n \rightarrow 0$	$x_n \rightarrow x_0$ $y_n \rightarrow 0$ $z_n \rightarrow z_0 + \frac{y_0}{x_0}$
hyperbolic	-1	$\tanh^{-1} 2^{-k}$	$x_n \rightarrow K'(x_1 \cosh z_1 + y_1 \sinh z_1)$ $y_n \rightarrow K'(y_1 \cosh z_1 + x_1 \sinh z_1)$ $z_n \rightarrow 0$	$x_n \rightarrow K'\sqrt{x_1^2 - y_1^2}$ $y_n \rightarrow 0$ $z_n \rightarrow z_1 + \tanh^{-1} \frac{y_1}{x_1}$

$$\begin{cases} x_{n+1} = x_n - m d_n y_n 2^{-\sigma(n)} \\ y_{n+1} = y_n + d_n x_n 2^{-\sigma(n)} \\ z_{n+1} = z_n - d_n w_{\sigma(n)}, \end{cases}$$

Circular ($m = 1$)	$\sigma(n) = n$
Linear ($m = 0$)	$\sigma(n) = n$
Hyperbolic ($m = -1$)	$\sigma(n) = n - k$ where k is the largest integer such that $3^{k+1} + 2k - 1 \leq 2n$

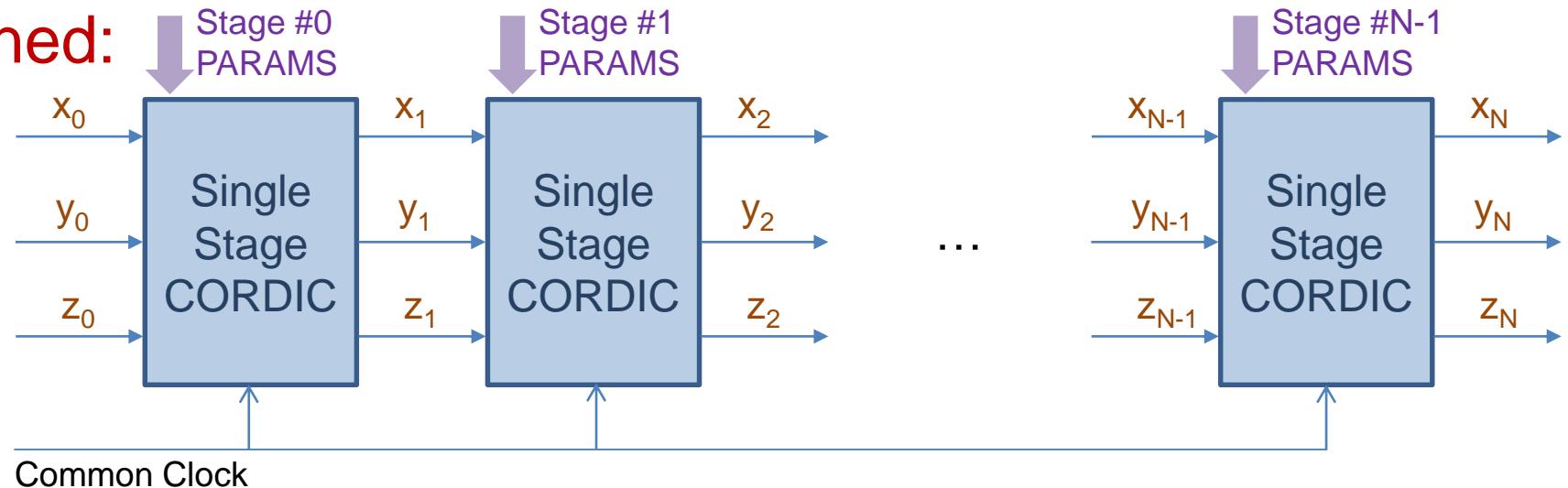
Note: The implementation of CORDIC on FPGA requires attention in word length selection and number representation

CORDIC Implementation on FPGA

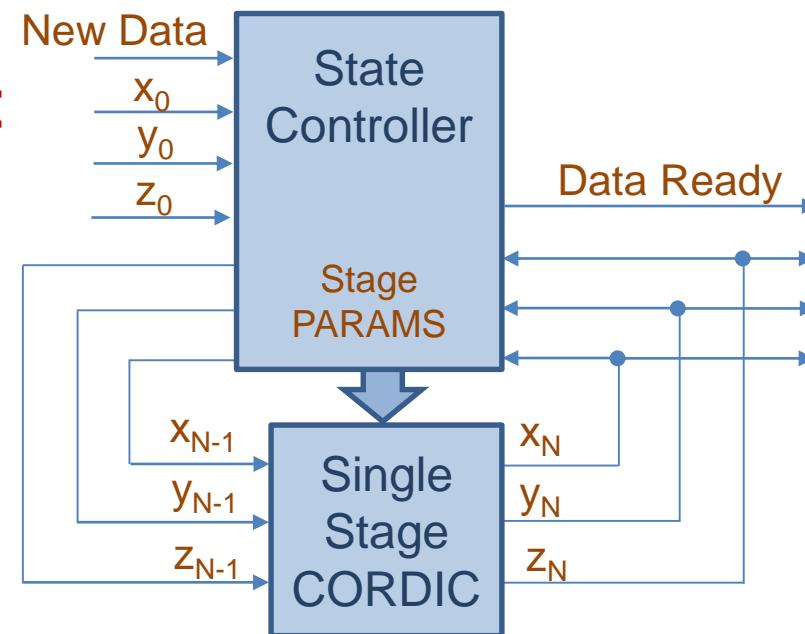
- Before implementation, the CORDIC parameters need to be set:
 1. Choose the CORDIC mode
 2. Set the input and output lengths and $Q_{m.n}$ data format
 3. Find the required number of CORDIC iterations **by simulation**, such that the calculation error is smaller than the LSB of the selected word lengths
 4. Implement the CORDIC machine using **pipelining or resource sharing** (or a combination of both)

CORDIC Implementation on FPGA (continued)

- Pipelined:

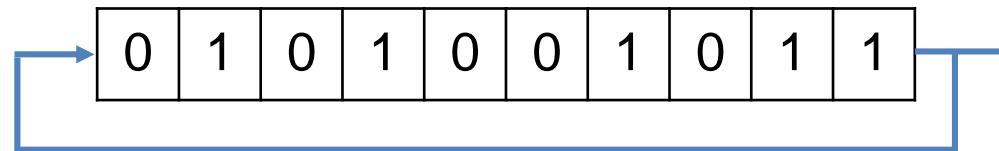


- Resource Shared:



Periodic Sequence Generation using Feedback Shift Registers

- Consider a chain of **N** registers with a common clock and arbitrary initial values (known as the **seed**) connected in feedback:

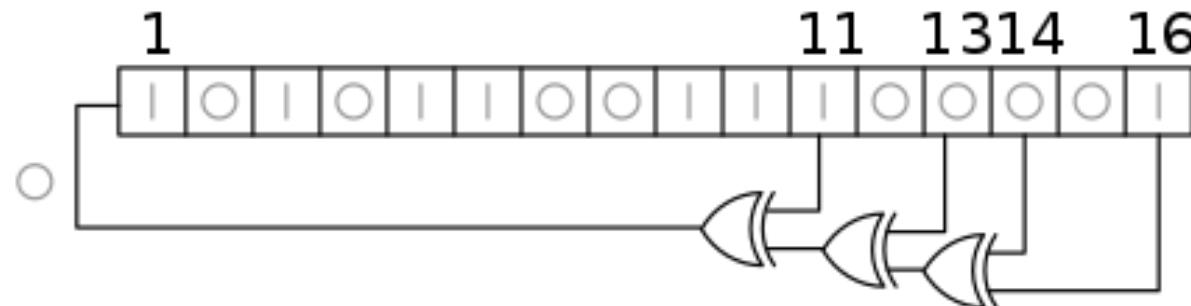


- The generated sequence is apparently periodic with (maximum) period N samples (N/f_s seconds)
- In FPGA, this feedback mechanism can be used to generate special periodic sequences at a very low cost (using **shift registers**)
- Next, suppose that the feedback bit is a Boolean function of the intermediate bits:

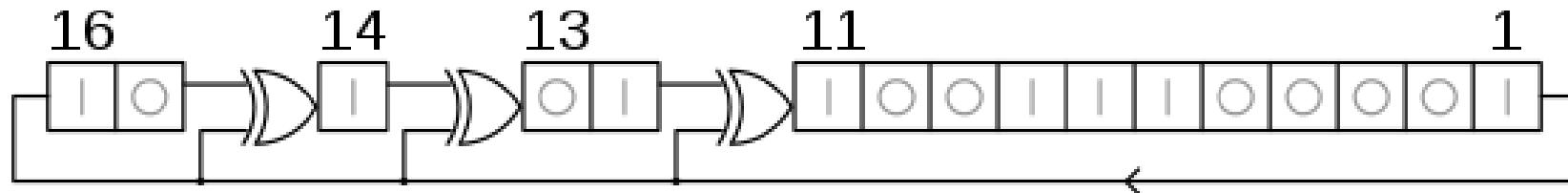
$$\begin{cases} x_0[n] = f(x_0[n-1], x_1[n-1], \dots, x_{N-1}[n-1]) \\ x_1[n] = x_0[n-1] \\ \dots \\ x_{N-1}[n] = x_{N-2}[n-1] \end{cases}$$

Periodic Sequence Generation using Feedback Shift Registers (continued)

Examples:



A 16-bit Fibonacci LFSR



A 16-bit Galois LFSR

Pseudo Random Number Generation using LFSR

- Linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function (e.g. XOR, XNOR, etc.) of its previous state
- The initial value of the LFSR is called the seed
- LFSRs are deterministic FSM, as the output stream is completely determined by its initial state and the linear function
- Since the register has a finite number of states, LFSR has a periodic cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that are pseudo-random (have a very long period).
- An N-bit LFSR is called maximum-length, if it cycles over all 2^N possible states except 0 (from which it would not exit from)

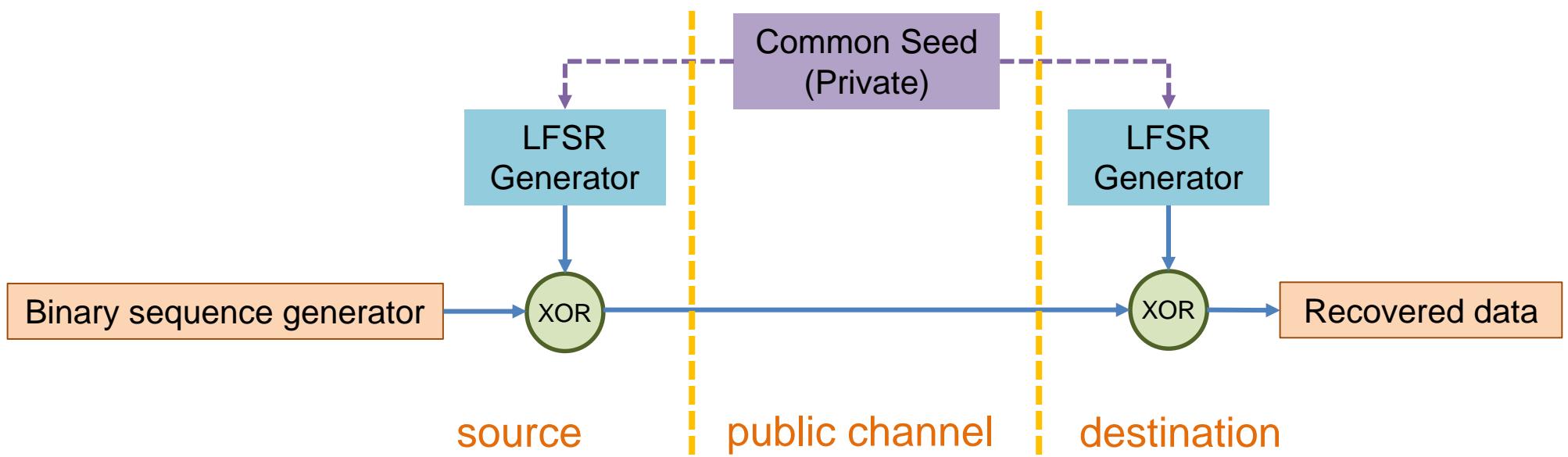
Pseudo Random Number Generation using LFSR (continued)

- In a maximum-length LFSR The length of LFSR can be selected such that even at the highest available flip-flop clocking speeds, the periodicity is not observed in centuries!
- **Example:** A maximum-length LFSR of length 64 clocked at 1GHz, takes $(2^{64}-1)/1\text{GHz} \approx 585$ years to repeat itself!
- Moreover, with an appropriate choice of the LFSR length and the feedback function (also known as the **LFSR polynomial**), the generated sequence **resembles a fully stochastic sequence**, which passes all the statistical tests of stochastic white noise.
- In this case, the periodic sequence may only be repeated by having the initial seed.
- LFSR have profound mathematical bases with numerous applications in coding, security, numeric computation, etc.

Ref: See the following for a nice introduction on the mathematics behind LFSR (Galois Fields):
<http://inst.eecs.berkeley.edu/~cs150/sp03/handouts/15/LectureA/lec27-6up>

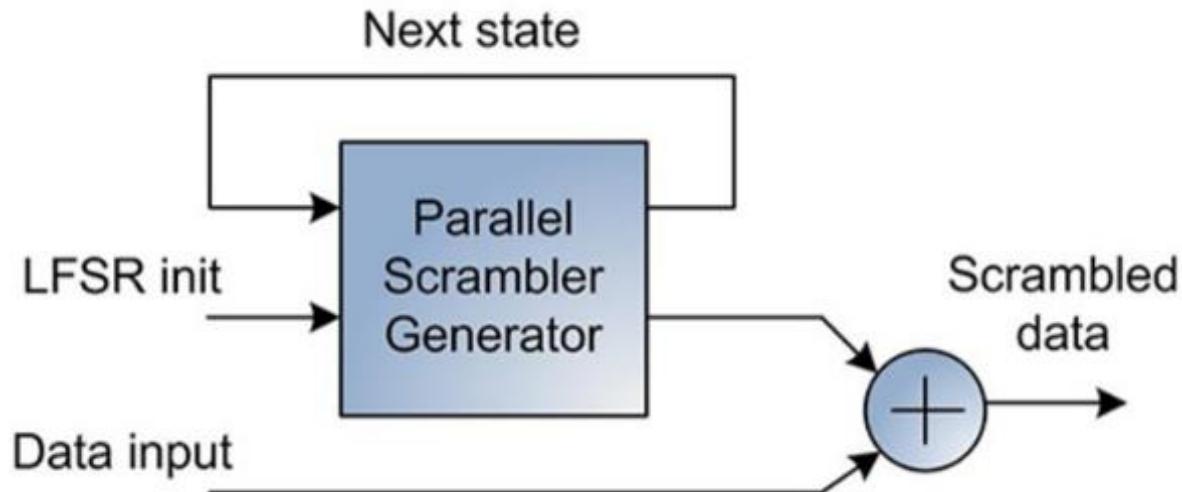
Other Applications of LFSR

1. **Counters:** LFSR can be used as extremely efficient counters (only requiring shift-registers and a few XOR), when the counting order is not important. For example for **FSM encoding** and **micro-codes**
2. **Cyclic Redundancy Check (CRC):** LFSR can be used to generate CRC for error detection and correction
3. **Data Encryption/Decryption:** LFSRs can be used for encryption of data transmitted over public channels



Other Applications of LFSR (continued)

4. **Scramblers:** Scramblers are used in many communication and storage protocols to randomize the transmitted data in order to remove long sequences of logic zeros and ones.

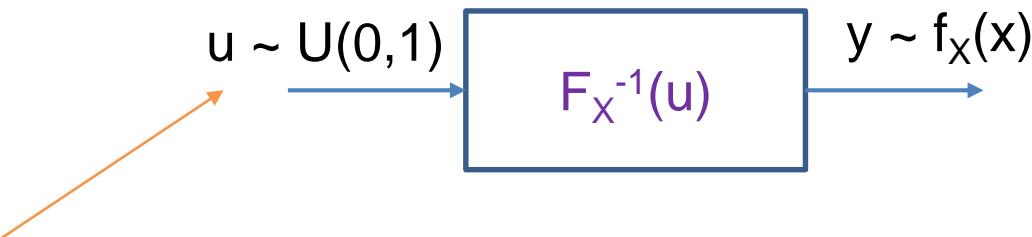


Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

Pseudo Random Numbers with Arbitrary Distributions

As noted before:

If a random variable (RV) x with a probability density function (pdf) $f_x(x)$ and cumulative distribution function (CDF) $F_x(x)$ passes a nonlinear memoryless system with a characteristics $u = F_x(x)$, the output u is uniformly distributed. Also, if a uniformly distributed RV u is given to $y = F_x^{-1}(u)$, the output has a distribution $f_x(\cdot)$.



Random or pseudo-random uniformly distributed variable

Pseudo Random Signals with Arbitrary Spectral Color*(optional)

Alternative methods for generating signal/noise with arbitrary spectra include:

- Frequency modulation using fast frequency sweeps (e.g. using a Chirp signal)
- Bandpass filtering pseudo-random white noise
- Superposition of synthetic signals and noise

PIPELINING & DESIGN TIMING IMPROVEMENT TECHNIQUES

Background

- The notion of pipelining was introduced before, as a means of improving the design timing, to achieve the design constraints (clock speed)
- Different techniques for pipelining and timing improvement in FPGA systems are presented in this section, including:
 - Retiming
 - Re-pipelining
 - Cut-set retiming
 - C-slow retiming
 - Pipelining in feedback systems

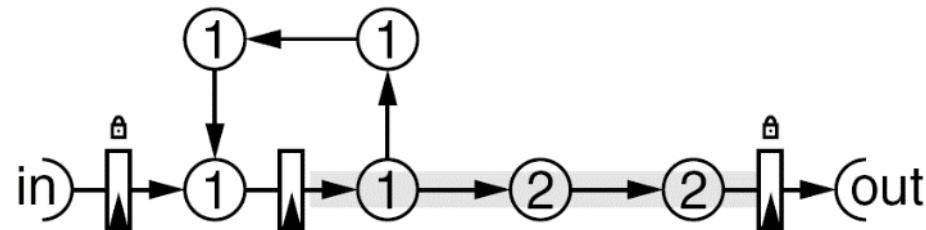
References:

- Hauck, Scott, and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Vol. 1. Elsevier, 2010, Chapter 18
- Khan, Shoab Ahmed. *Digital design of signal processing systems: a practical approach*. John Wiley & Sons, 2011, Chapter 7

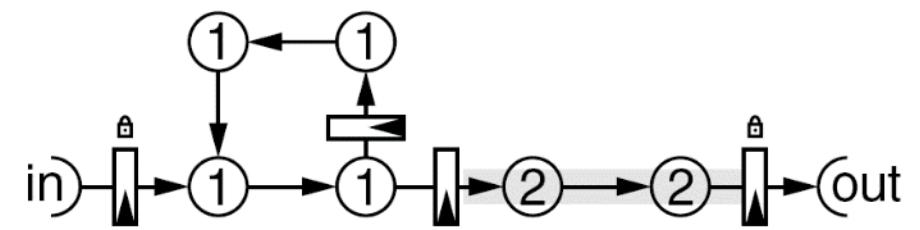
Retiming

- Retiming consists of reducing the critical path (increasing the clock speed) by moving the pipeline registers to an “**optimal position**”.

Example: In the following, each circle denotes combination logic, with the number representing the combinational latency



Before retiming



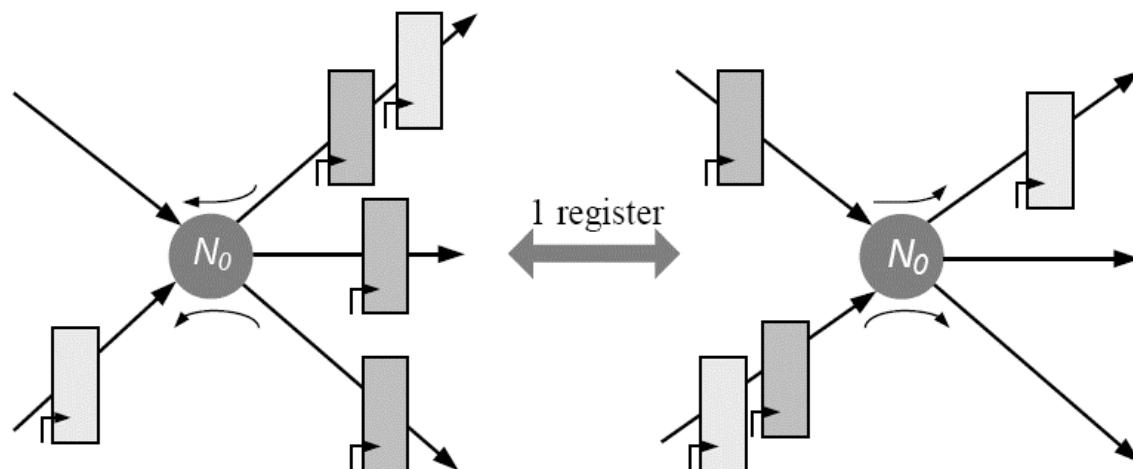
After retiming

- The objective of retiming is to automate this procedure in a systematic manner with concise algorithms, which **1)** guarantee that the circuit’s I/O transfer function is not changed and **2)** can be implemented in CAD tools (for instance during the synthesis or technology mapping stages)
- **Limitation:** Retiming **cannot** improve the design clock speed beyond the optimal register placement

Retiming (continued)

- For systematic retiming, a digital circuit is converted to a **data flow graph (DFG)**. Next, by using graph theory based theorems, the registers are systematically moved across the computational nodes (combinational logic), without changing the **input/output transfer function** of the original DFG.

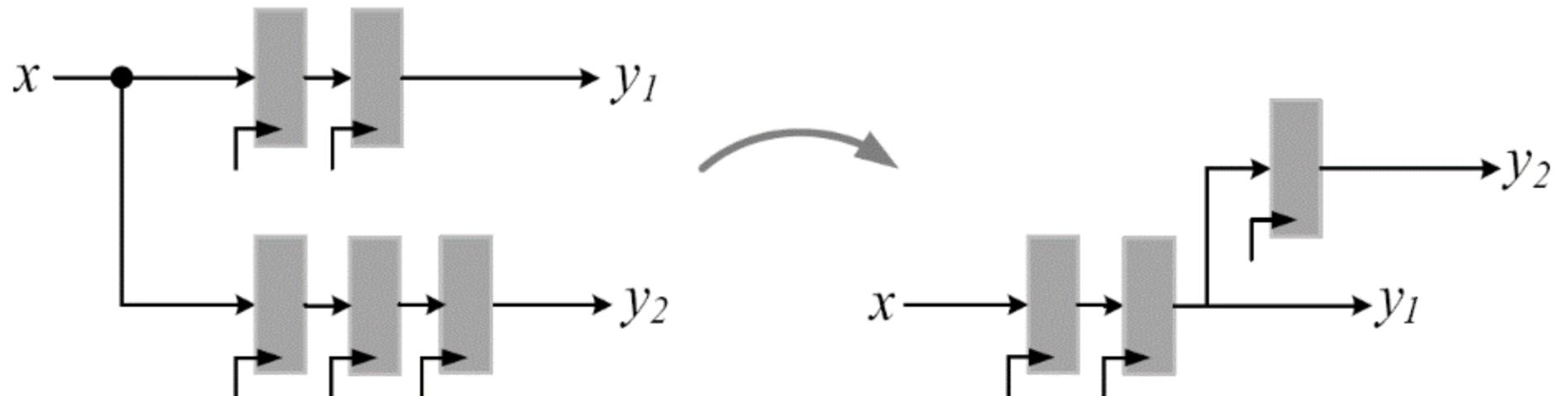
Delay Transfer Theorem: “without affecting the transfer function of the system, registers can be transferred from each incoming **edge** of a **node** of a DFG, to all outgoing edges of the same node or vice versa” [Khan, 2011, p. 304].



Retiming (continued)

- Retiming can also be used to merge excess registers to reduce the area utilization.

Example:

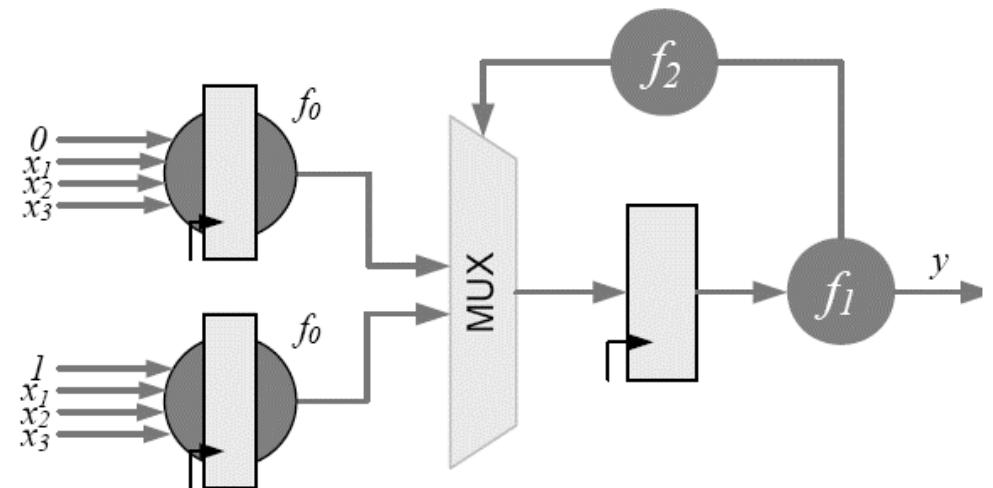
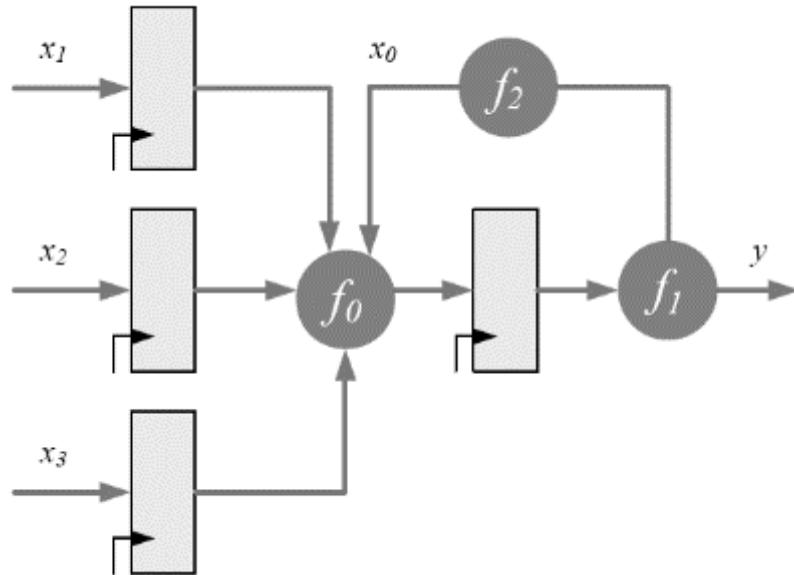


Shannon Decomposition Retiming

- The **Shannon decomposition** can be used to improve the timing of Boolean functions. Accordingly:

$$f(a_0, a_1, \dots, a_{N-1}) = \bar{a}_0 \cdot f(0, a_1, \dots, a_{N-1}) + a_0 \cdot f(1, a_1, \dots, a_{N-1})$$

Example:

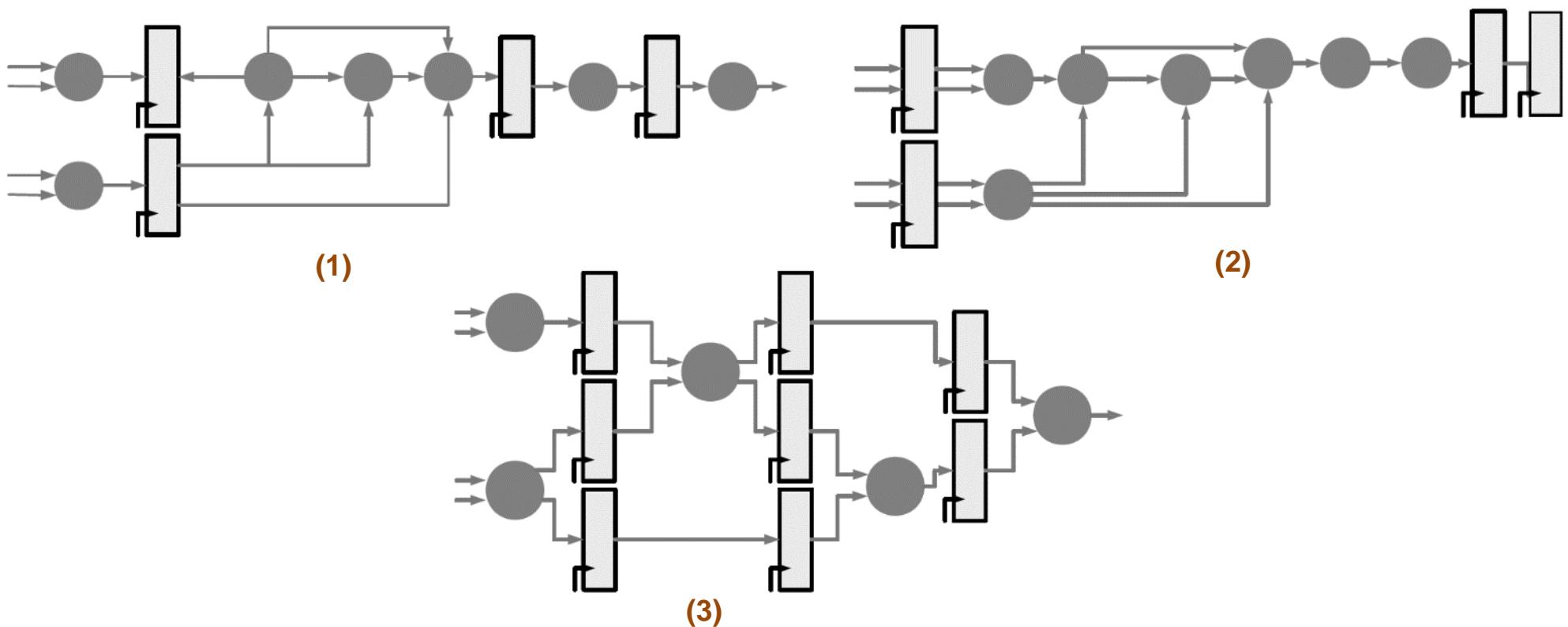


Note: The Shannon decomposition is specifically useful for FPGA-based designs, which are implemented on fixed-input LUTs

Peripheral Retiming

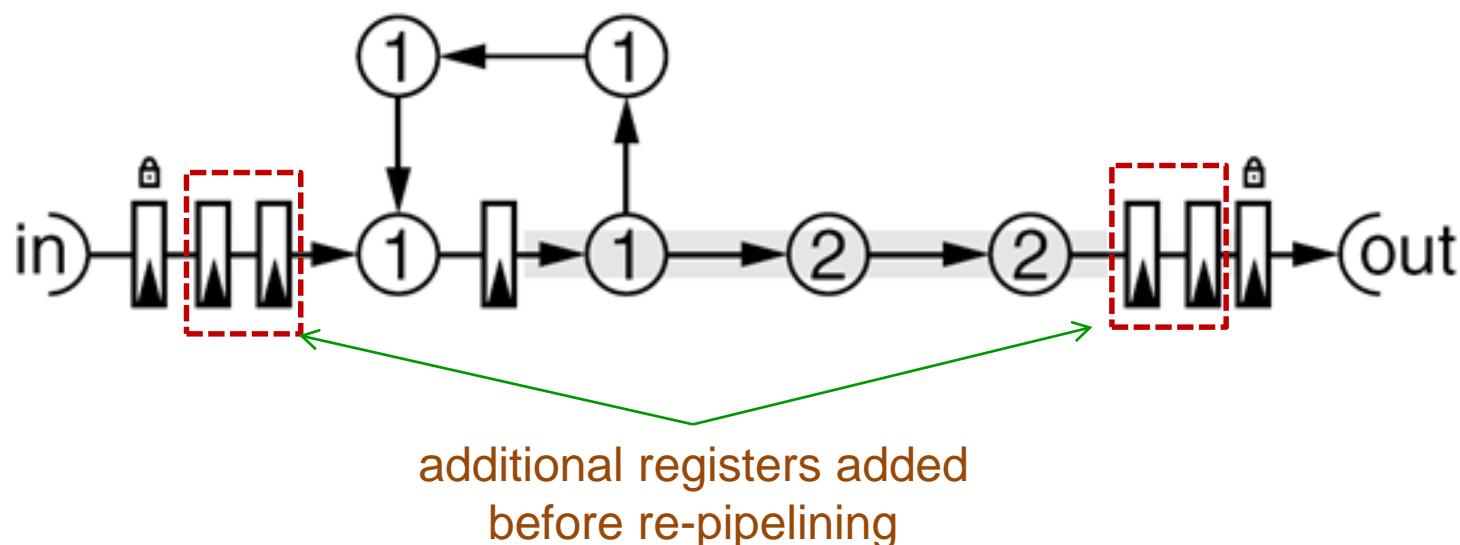
- In this technique: 1) all the internal registers are shifted to the input or output of the design; 2) the combinational logic is simplified; finally 3) the registers are pushed to their optimal position by conventional retiming.

Example:



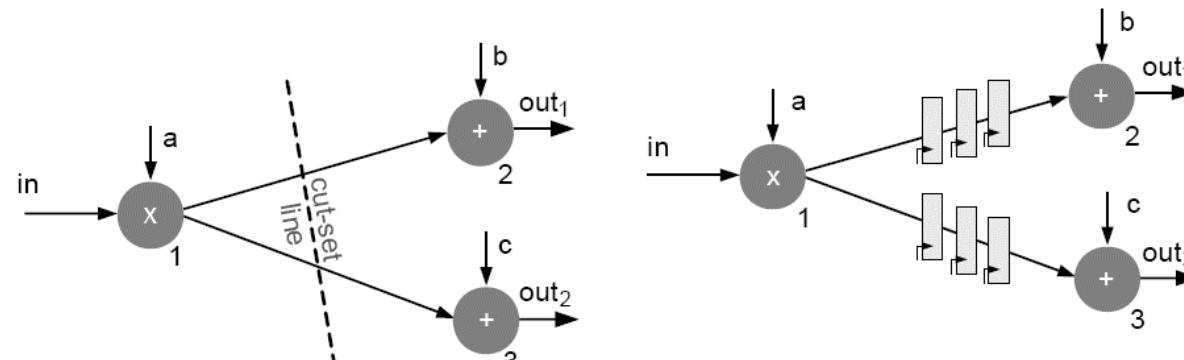
Re-pipelining

- In feed-forward designs, re-pipelining adds additional registers at the input or output and then moves these registers across the design (by retiming) to obtain the best performance.
- The cost of re-pipelining is the additional number of registers added to the pipeline which adds a constant clock latency between the input and output; but the other properties of the design are preserved.

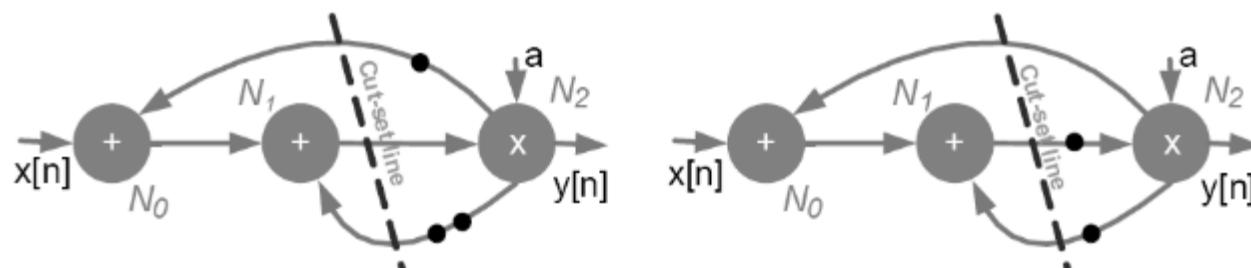


Cut-set Retiming

- More generally, cut-set retiming permits the addition of arbitrary number of registers in a forward path, or moving registers from the input to the output (or vice versa) of a **cut-set**, while preserving the I/O transfer function.
- Reminder:** In Graph theory, a **cut** is a virtual partitioning of the edges of a graph into two disjoint subsets, known as **cut-sets**.



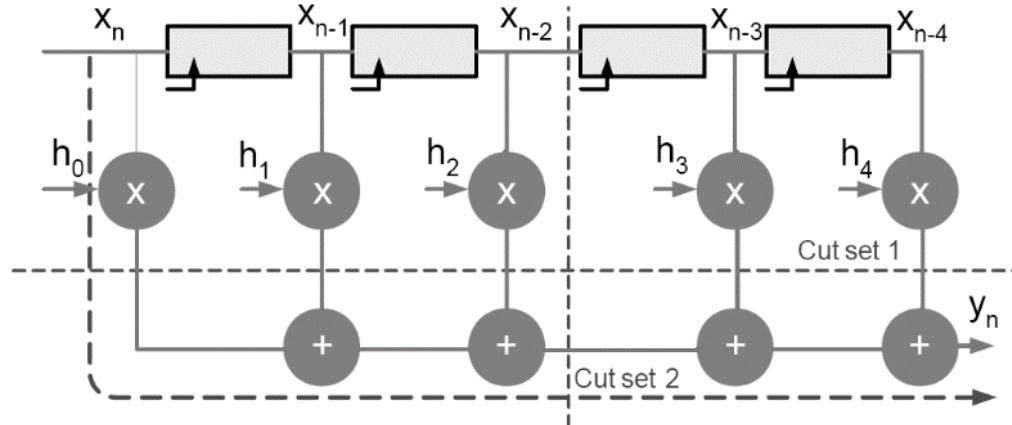
adding registers in feed-forward cut-sets



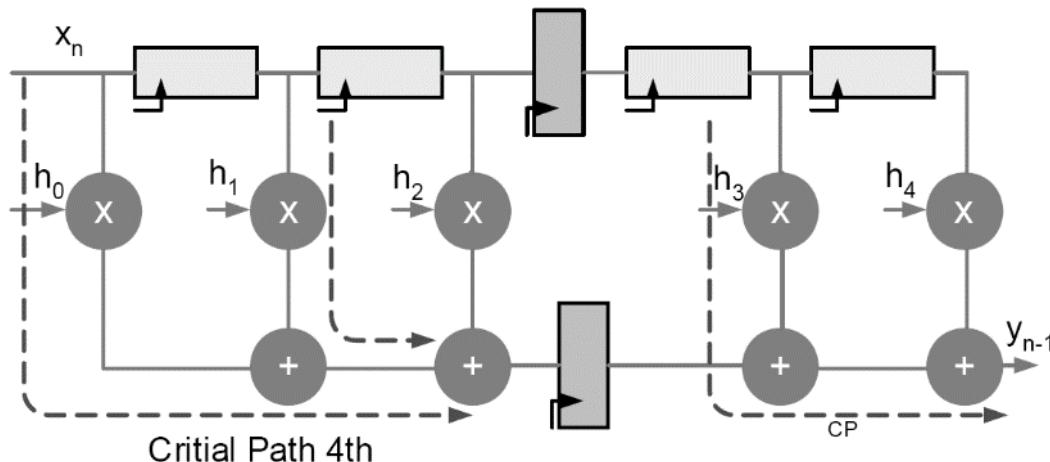
moving registers from cut-set output to cut-set inputs

Cut-set Retiming (continued)

Example 1: FIR filter retiming



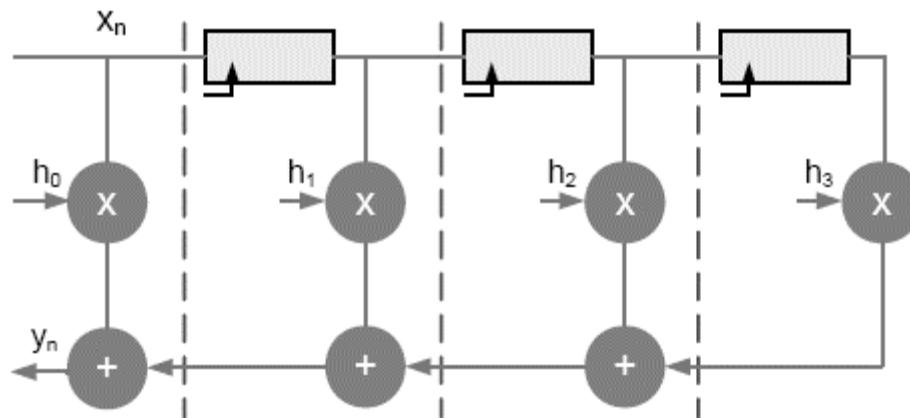
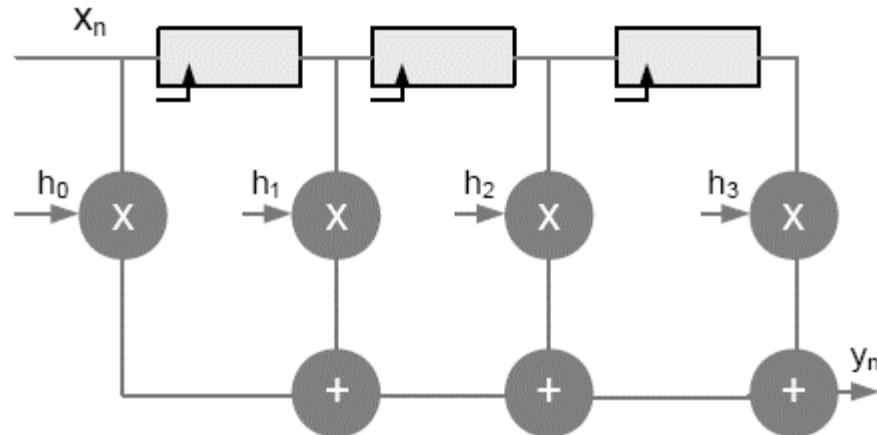
Two possible cut-sets



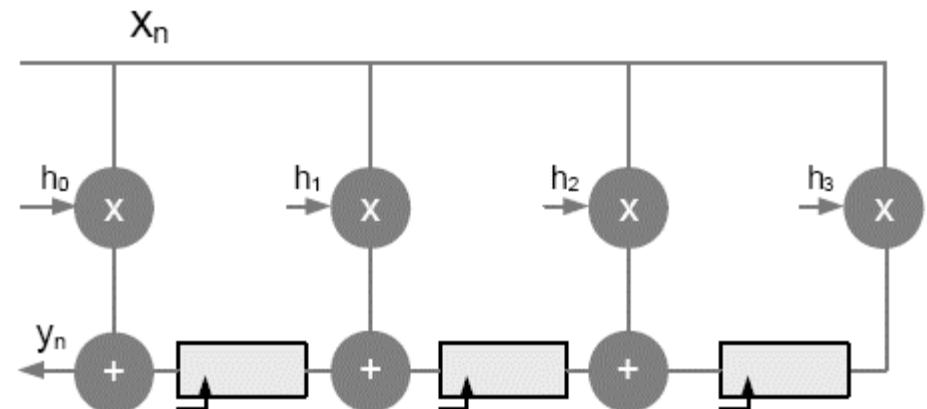
Re-pipelining across feed-forward cut-set 2

Cut-set Retiming (continued)

Example 2: FIR filter retiming, second approach: multiple cut-set retiming



Three cut-sets with feedback paths

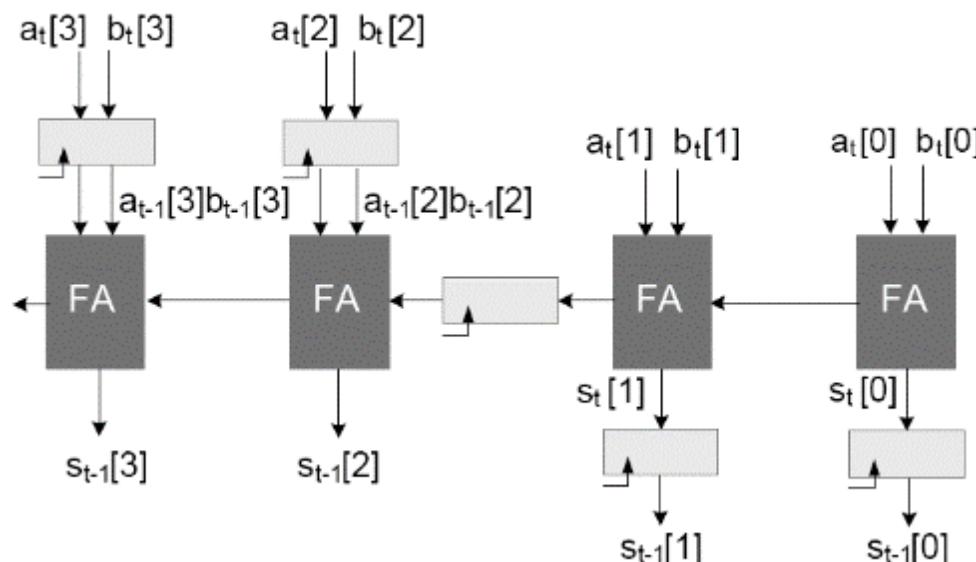
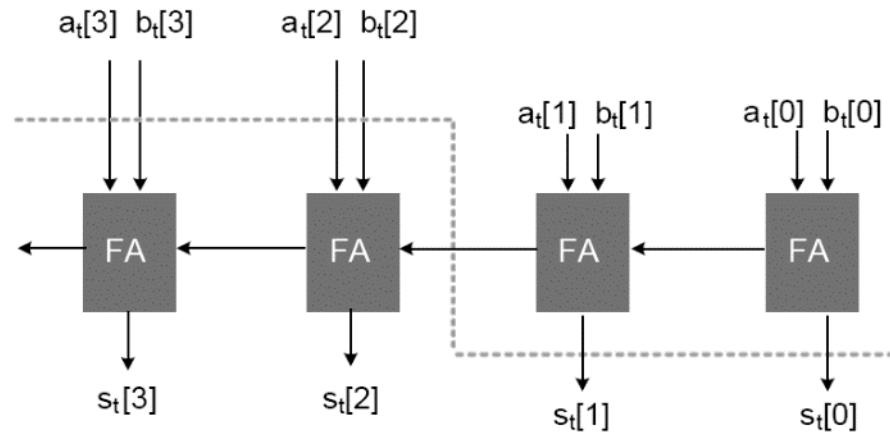


After cut-set retiming

Cut-set Retiming

(continued)

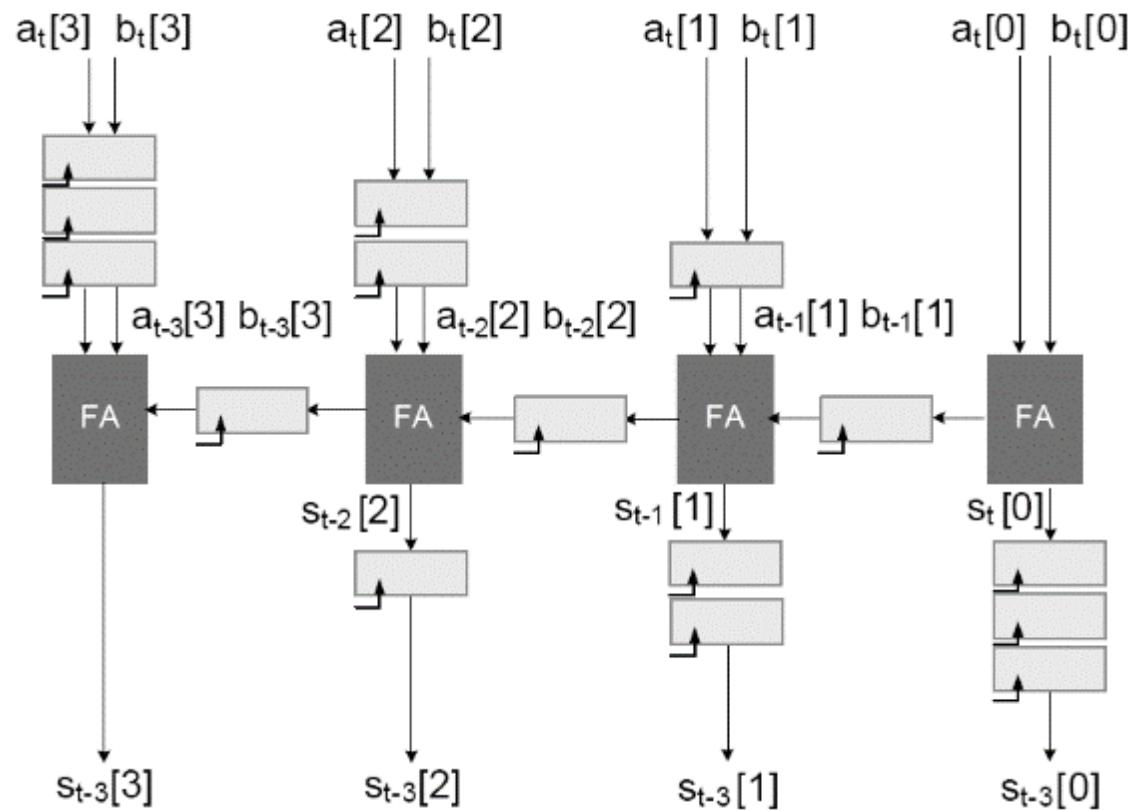
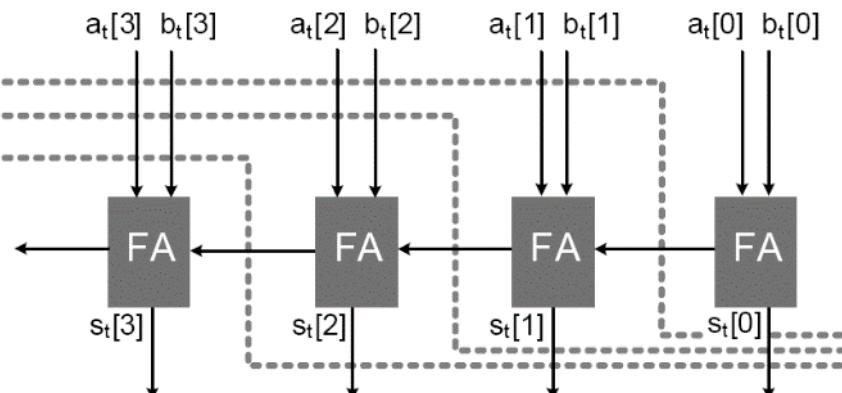
Example 3: 4-bit ripple carry adder (RCA) retiming



Cut-set Retiming

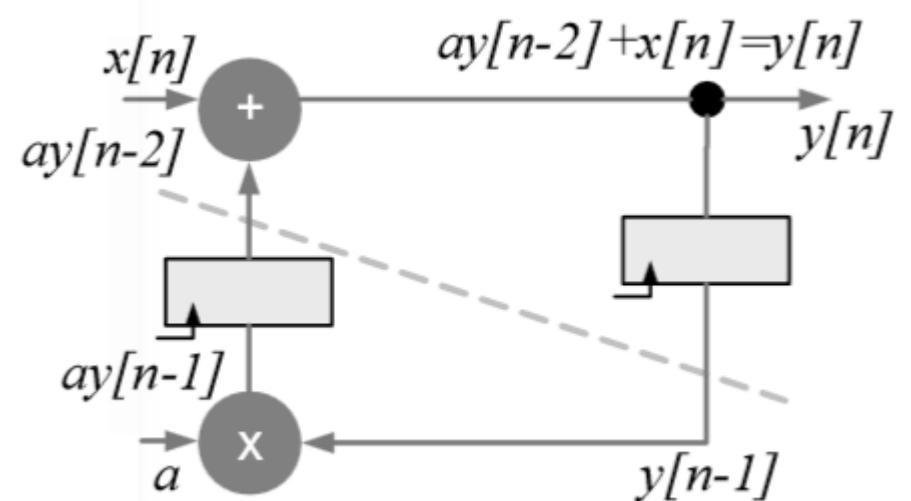
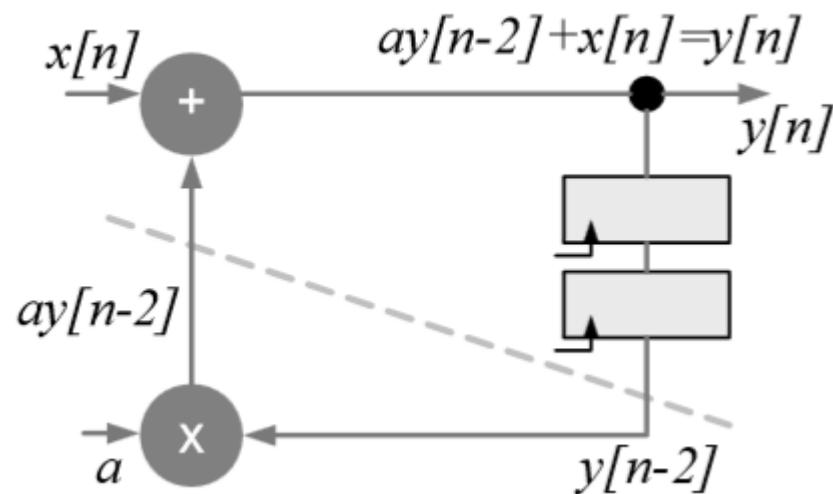
(continued)

Example 4: 4-bit ripple carry adder (RCA) retiming; second approach



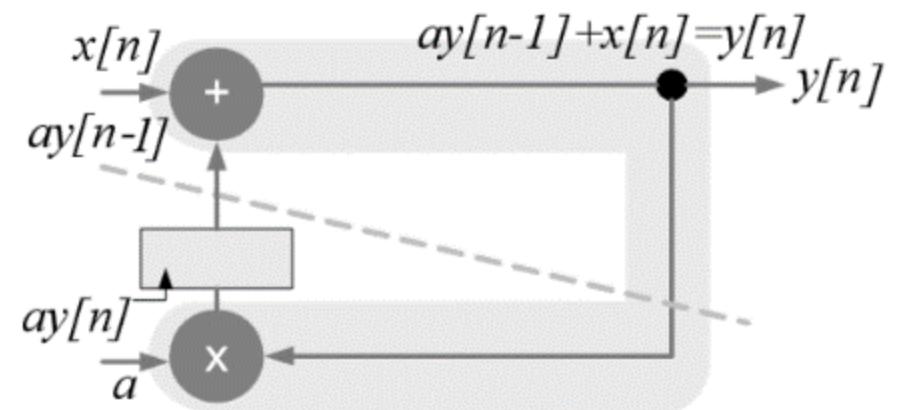
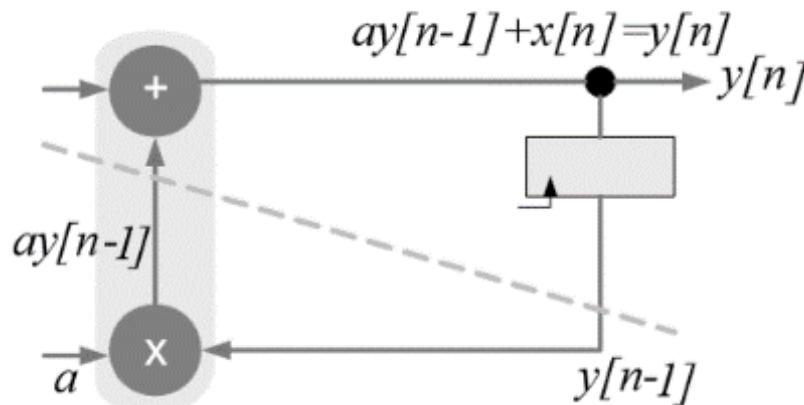
Cut-set Retiming (continued)

Example 5: Second-order IIR filter



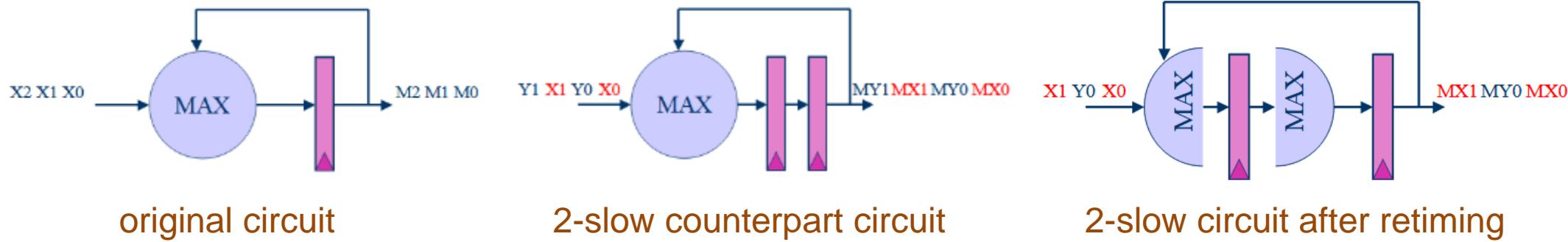
Cut-set Retiming (continued)

- Cut-set retiming does not always result in an improved timing.
- **Example:** In a first-order IIR filter, the critical path is not changed by cut-set retiming of the feedback loop.



C-Slow Retiming

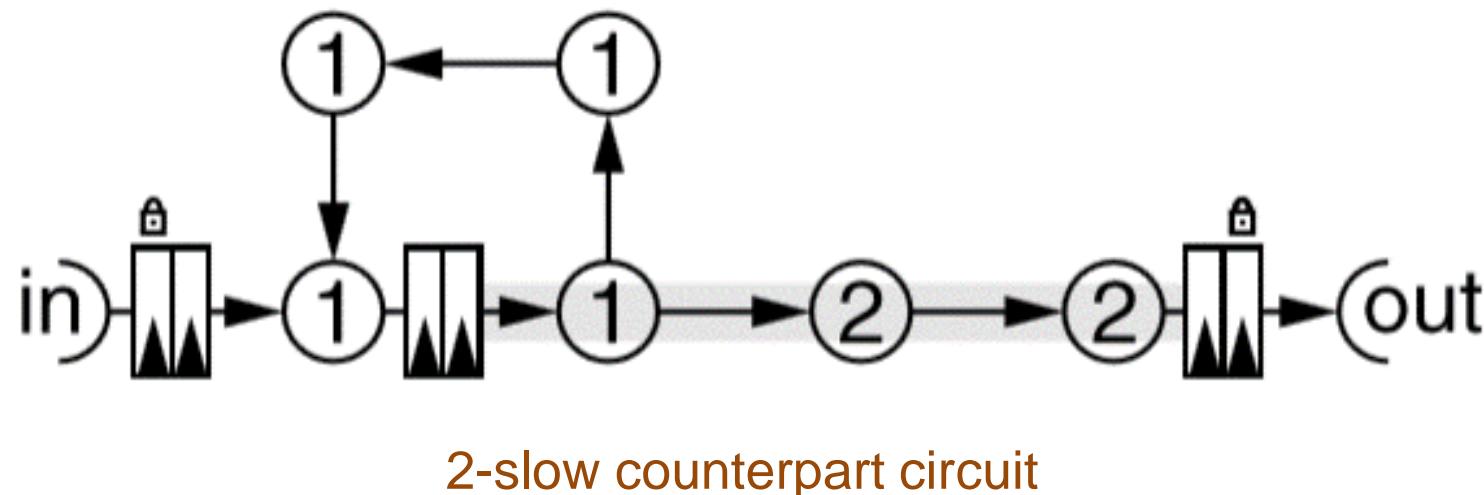
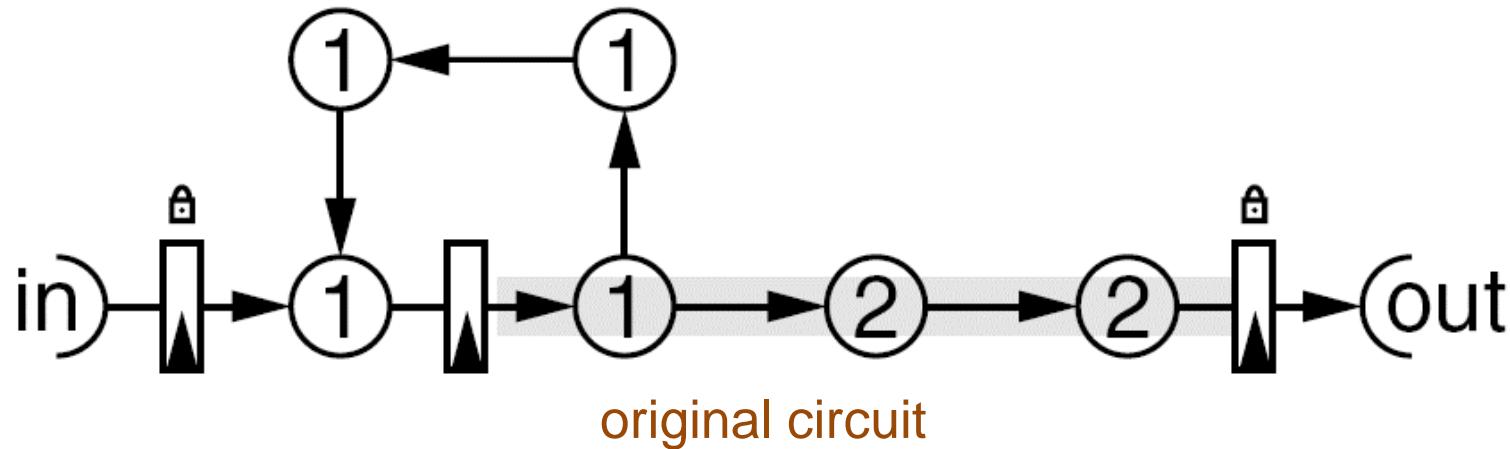
- C-slow retiming consists of replicating all the registers of a synchronous design C times, followed by moving the registers (conventional retiming), or by splitting the circuit into C distinct parallel paths which multiplex and switch between the input data and results.



Note: The design interleaves between two computations (2-slow): on the first clock cycle, it accepts the first input for the first data stream; on the second clock cycle, it accepts the first input for the second stream, and on the third it accepts the second input for the first stream. Due to the interleaved nature of the design, *the two streams of execution will never interfere* (on odd clock cycles, the first stream of execution accepts input; on even clock cycles, the second stream accepts input).

C-Slow Retiming (continued)

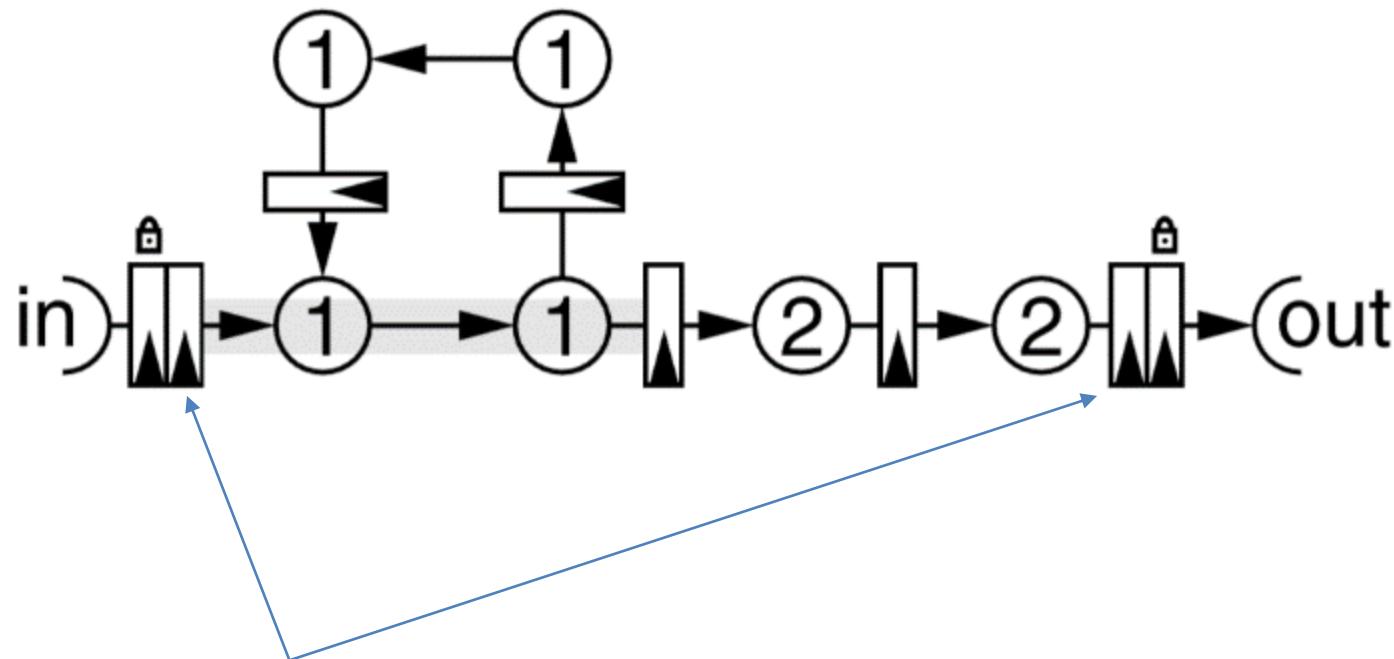
Example:



C-Slow Retiming (continued)

Example (continued):

- 2-slow retiming after moving the registers to their optimal position (the critical path is reduced from 5 to 2 time units):
- This architecture can process two parallel data paths with interleaved data



excess feed-forward registers can
be eliminated after retiming

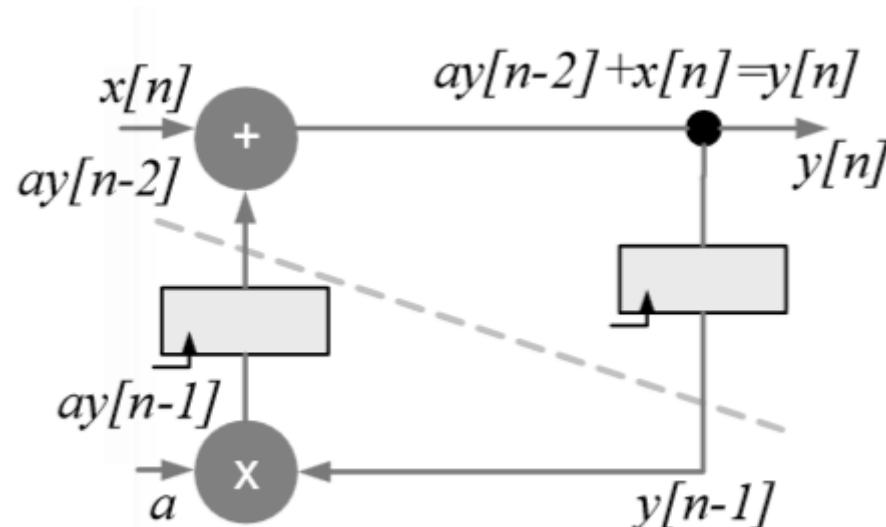
C-Slow Retiming (continued)

- **Example:** A single 2-slow retimed IIR filter architecture can be used to process the **real** and **imaginary** parts of a complex-valued digital filter by interleaving the real and imaginary parts of the input:

$$y_r[n] + j y_i[n] = h[n]^* (x_r[n] + j x_i[n]) = h[n]^* x_r[n] + j h[n]^* x_i[n]$$

$x_r[0]$	$x_i[0]$	$x_r[1]$	$x_i[1]$	$x_r[2]$	$x_i[2]$	$x_r[3]$	\dots
----------	----------	----------	----------	----------	----------	----------	---------

$y_r[0]$	$y_i[0]$	$y_r[1]$	$y_i[1]$	$y_r[2]$	$y_i[2]$	$y_r[3]$	\dots
----------	----------	----------	----------	----------	----------	----------	---------



C-Slow Retiming by Data Stream Interleaving

- The disjoint data stream property of C-slow retiming can be used to obtain **parallel hardware threads**, which interleave the input data stream between C identical circuits, each working at $1/C$ of the input clock rate and finally multiplexing the results back together. This method is referred to as **unfolding** in some textbooks.
- The idea is related to **loop unrolling** used for optimizing for-loops in multicore processors and GPUs
- The complementary method is **hardware folding (hardware reuse)**, which uses a single hardware and a **scheduler** (FSM controller) to reduce the hardware size.
- **Note:** Systematic and *ad hoc* retiming and resource sharing may additionally be used to improve the area and timing performance of the design.

C-Slow Retiming by Data Stream Interleaving (continued)

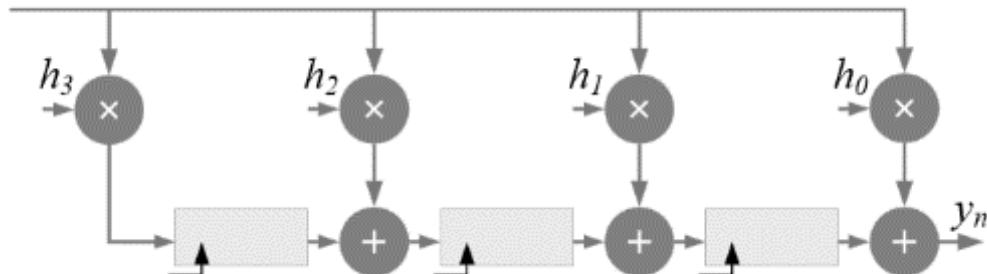
Algorithm: Any DFG can be unfolded by an unfolding factor J using the following two steps:

- S_0) To unfold the graph, each node U of the original DFG is replicated J times as U_0, \dots, U_{J-1} in the unfolded DFG.
- S_1) For two connected nodes U and V in the original DFG with w delays, draw J edges such that each edge j ($= 0, \dots, J-1$) connects node U_j to node $V_{(j+w)\%J}$ with $\text{floor}[(j+w)/J]$ delays.

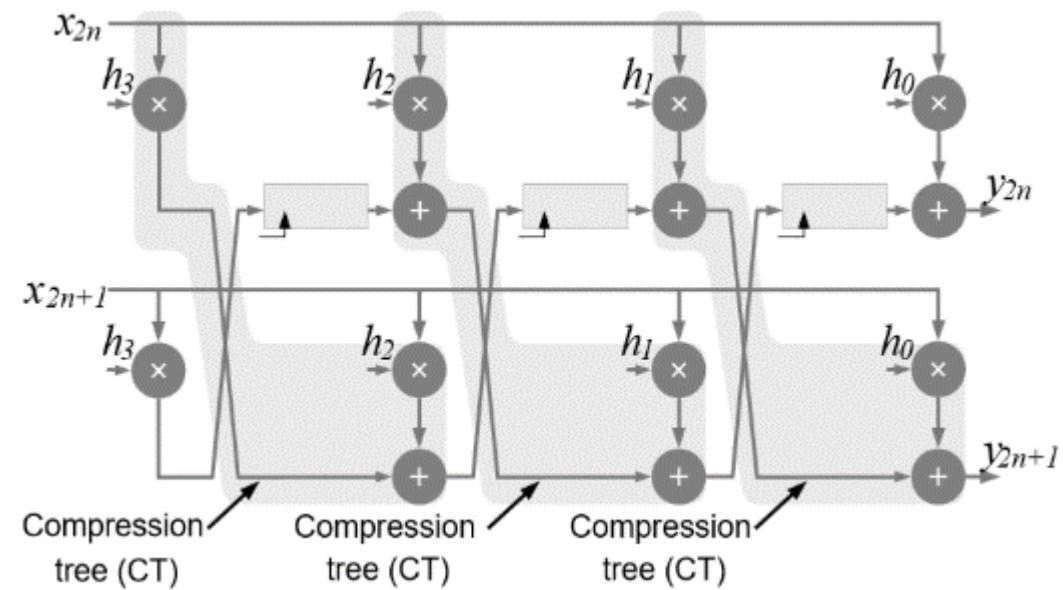
Ref: [Khan, 2011] p. 349

C-Slow Retiming by Data Stream Interleaving (continued)

Example: Feed-forward example



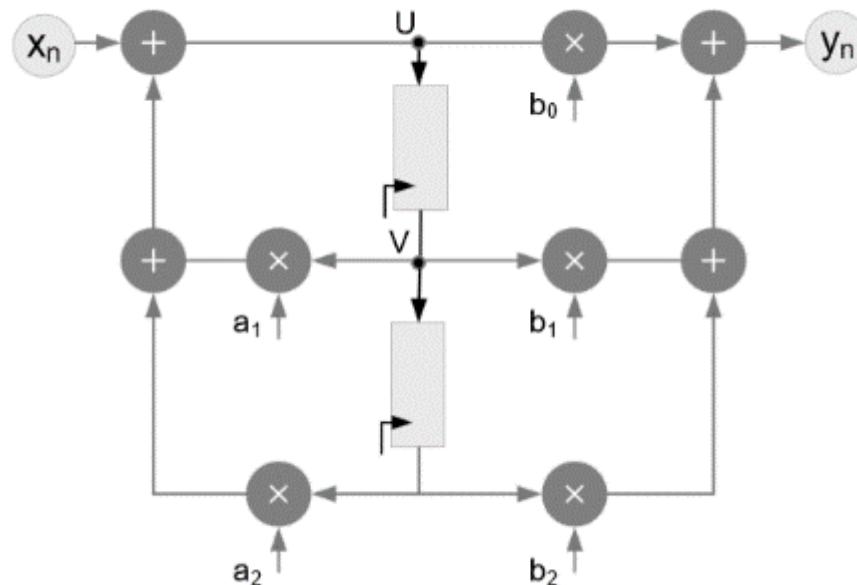
Original circuit



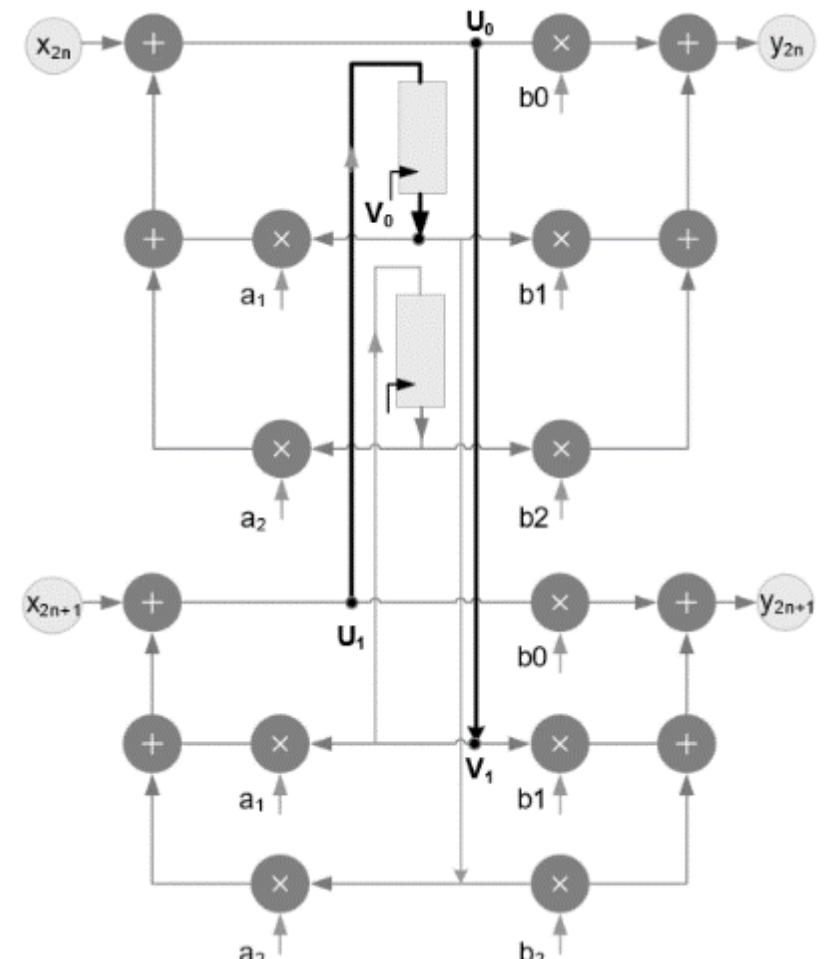
Unfolded system (2-fold)

C-Slow Retiming by Data Stream Interleaving (continued)

Example: Feedback systems



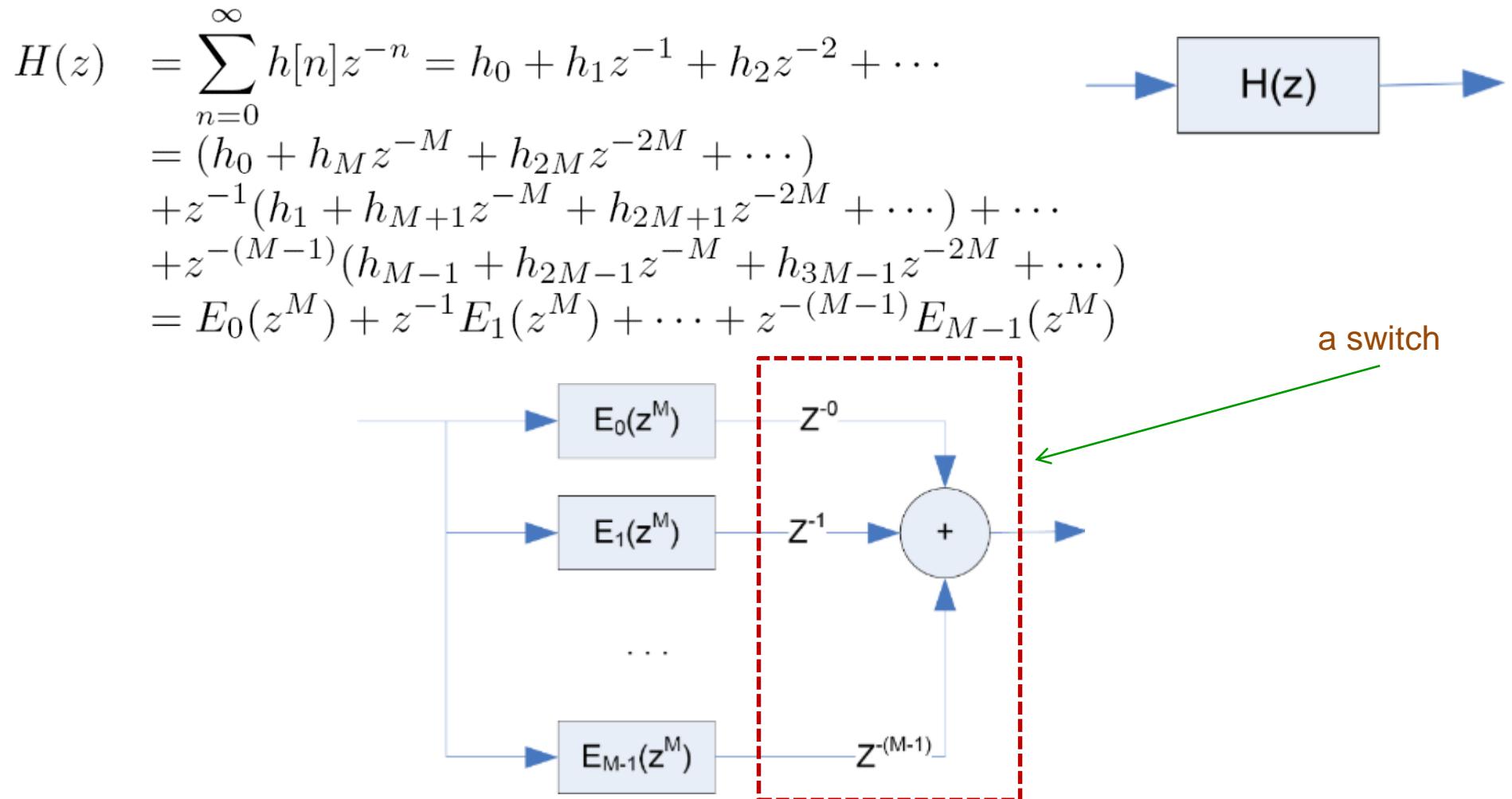
Original circuit



Unfolded system (2-fold)

C-Slow Retiming by Data Stream Interleaving Example* (optional)

- Example: Polyphase filter Implementation

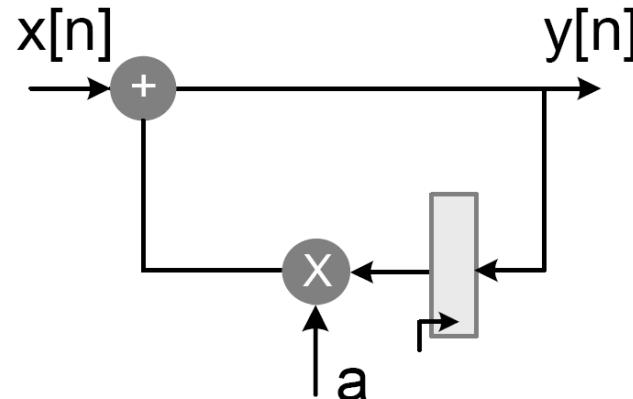


Pipelining Feedback Systems by Algorithmic Modifications

- Pipelining digital systems with feedback is a challenging issue and is not always solved using the previous methods. In this section, we study a few techniques for pipelining such systems by **algorithmic modifications**, using a simple case study.

Example: Consider a first-order recursion $y[n] = a \cdot y[n-1] + x[n]$.

- Such equations appear in many applications, e.g., infinite-impulse response (IIR) filters in signal processing
- The multiplication is problematic for pipelining, since the result of $a \cdot y[n-1]$ is needed for calculating $y[n]$ before the next clock edge arrives



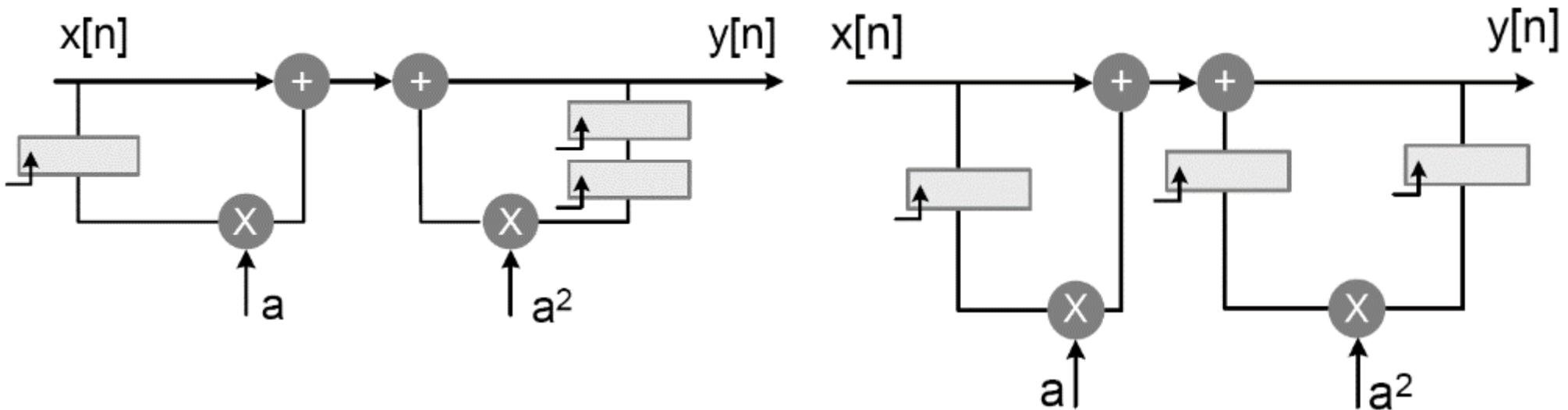
Solution?

Pipelining Feedback Systems by Algorithmic Modifications (continued)

- The first-order recursion can be rewritten as follows:

$$y[n] = a \cdot y[n-1] + x[n] = a \cdot (a \cdot y[n-2] + x[n-1]) + x[n] = a^2 \cdot y[n-2] + a \cdot x[n-1] + x[n]$$

- This modified form requires more architecture (compared to the original form); but it can be pipelined:

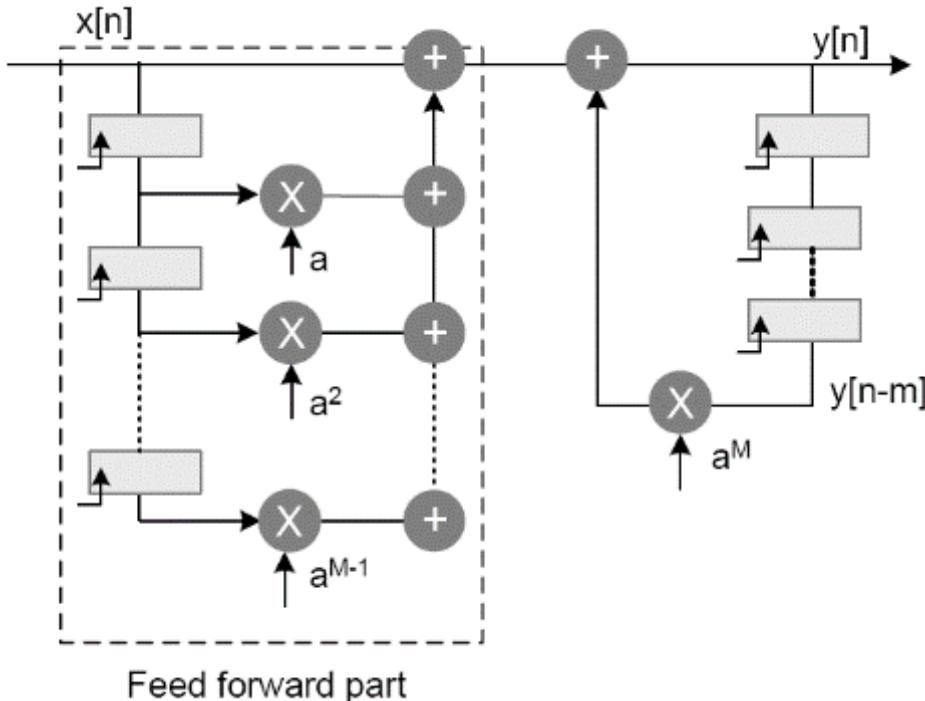


Pipelining Feedback Systems by Algorithmic Modifications (continued)

- More generally:

$$y[n] = a \cdot y[n-1] + x[n] = a^M \cdot y[n-M] + (x[n] + a \cdot x[n-1] + \dots + a^{M-1} \cdot x[n-M+1])$$

- This form can be pipelined as follows:



Note*: From the signal processing viewpoint, we are using the following property of the z-transform of the system response:

$$\begin{aligned} H(z) &= 1/(1-az^{-1}) \\ &= (1 + az^{-1} + \dots + a^{M-1}z^{-M+1})/(1 - a^Mz^{-M}) \end{aligned}$$

In other words, we are adding overlapping zeros and poles to the transfer function, in favor of pipelining

- This method is known as **look ahead transformation** in the literature.

Architectural Improvements by Algorithmic Modifications*(optional)

- Replacing a system with its algorithmically equivalent counterpart (in favor of architectural improvement) is very common in digital implementations.
- Example:** Consider a moving average filter (used for lowpass filtering) defined by the input-output recursion: $y[n] = x[n] + x[n-1] + \dots + x[n-N+1]$

Accordingly the impulse response and transfer functions of the system are:

$$h[n] = \delta[n] + \delta[n-1] + \dots + \delta[n-N+1] \quad \text{or} \quad H(z) = 1 + z^{-1} + \dots + z^{-N+1}$$

The FPGA implementation of this system requires N-input adders, which can cause huge combinational delays for large N.

A method for improving this limitation is by using **pipelined adder-trees**.

Alternatively, one may use the equivalent system: $y[n] = y[n-1] + x[n] - x[n-N]$

We have used the fact that:

$$\begin{aligned} H(z) &= (1 + z^{-1} + \dots + z^{-N+1}) \\ &= (1 - z^{-N}) / (1 - z^{-1}) \end{aligned}$$

- Cascaded Integrator Comb (CIC)** also known as **Hogenauer** filters, which are very common in FPGA-based designs due to their multiplier-free property, are based on this method.

Further Reading

- Further reading on pipelining, folding and unfolding techniques for feed-forward and feedback systems:
 1. Khan, Shoab Ahmed. **Digital design of signal processing systems: a practical approach.** John Wiley & Sons, 2011, Chapter 7.
 2. Meyer-Baese, Uwe, and U. Meyer-Baese. **Digital signal processing with field programmable gate arrays.** Vol. 2. Berlin: Springer, 2004, Chapter 4.
 3. Hauck, Scott, and Andre DeHon. **Reconfigurable computing: the theory and practice of FPGA-based computation.** Vol. 1. Elsevier, 2010, Chapter 18

METASTABILITY & MULTIPLE CLOCK DOMAINS

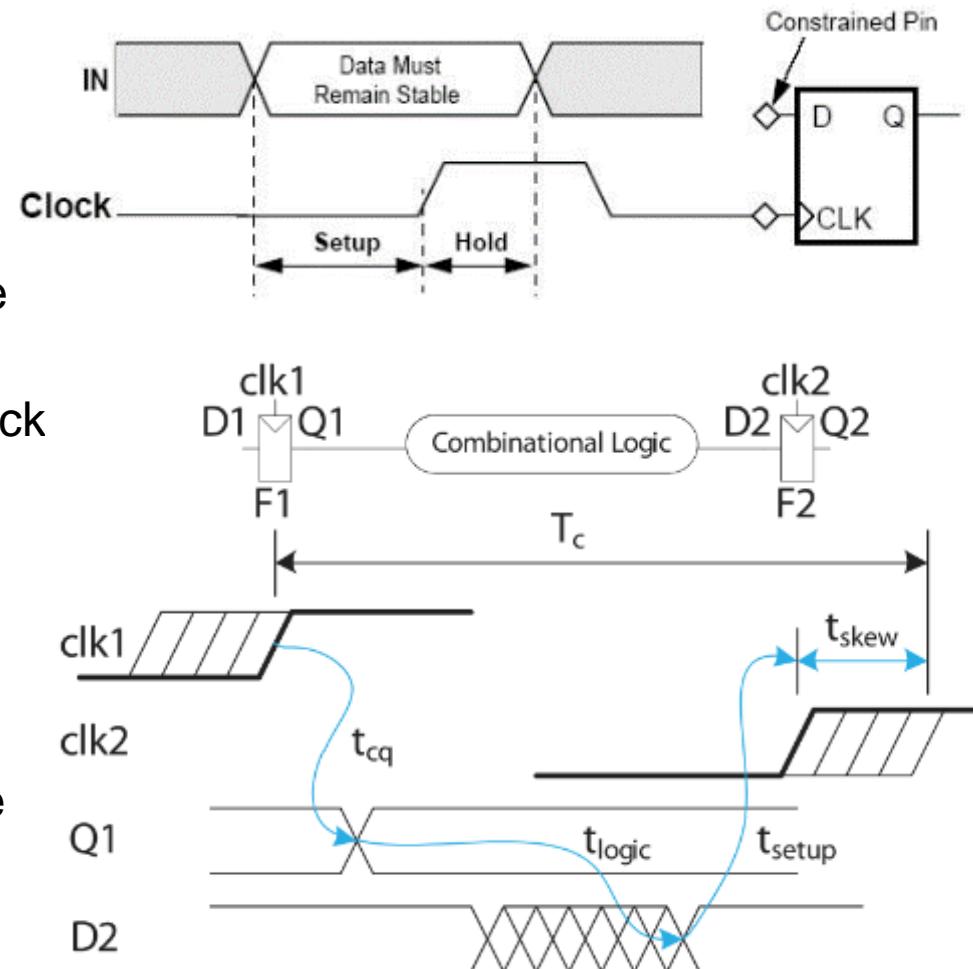
Introduction

- Up to now, we have considered flip-flops and other logic devices as fully **deterministic** elements.
- However, in reality, no two flip-flops are “**exactly**” the same. The (minor) deviations between the electronic aspects and fabrication indeterminacies of these elements result in **stochastic** behaviors.
- Although current FPGA vendors guarantee extremely robust behaviors and extremely low probabilities of device failures, the consideration of the stochastic aspects are inevitable in certain cases, including **multiple clock domain** applications, which may result in **metastability**.
- In this section, we study some of the stochastic aspects of digital elements, such as flip-flops and robust design methods that reduce the probability of metastability and failure of digital systems.

Reference: M. Arora. *The art of hardware architecture: Design methods and techniques for digital circuits*. Springer Science & Business Media, 2011.

Review of Logic Circuits Timing Parameters

- **Clock period (t_C)**: clock edge-to-edge time; inverse of **clock frequency (f_C)**
- **Clock Skew (t_{skew})**: indeterminacy of the clock edge arrival time
- **Setup Time (t_{setup})**: data should be stable before clock edge
- **Hold Time (t_{hold})**: data should be stable after clock edge
- **Propagation Delay (t_{CQ})**: clock edge to stable output
- **Combinational delay (t_{logic})**: combinational logic circuit settling time
- **Setup Slack (t_{slack})**: minimum data required time minus data arrival time:
 - **Positive**: timing met
 - **Negative**: timing violated
- We want: $t_C \geq t_{CQ} + t_{logic} + t_{logic} + t_{skew}$



Note: HIGH-to-LOW and LOW-to-HIGH times are not necessarily the same

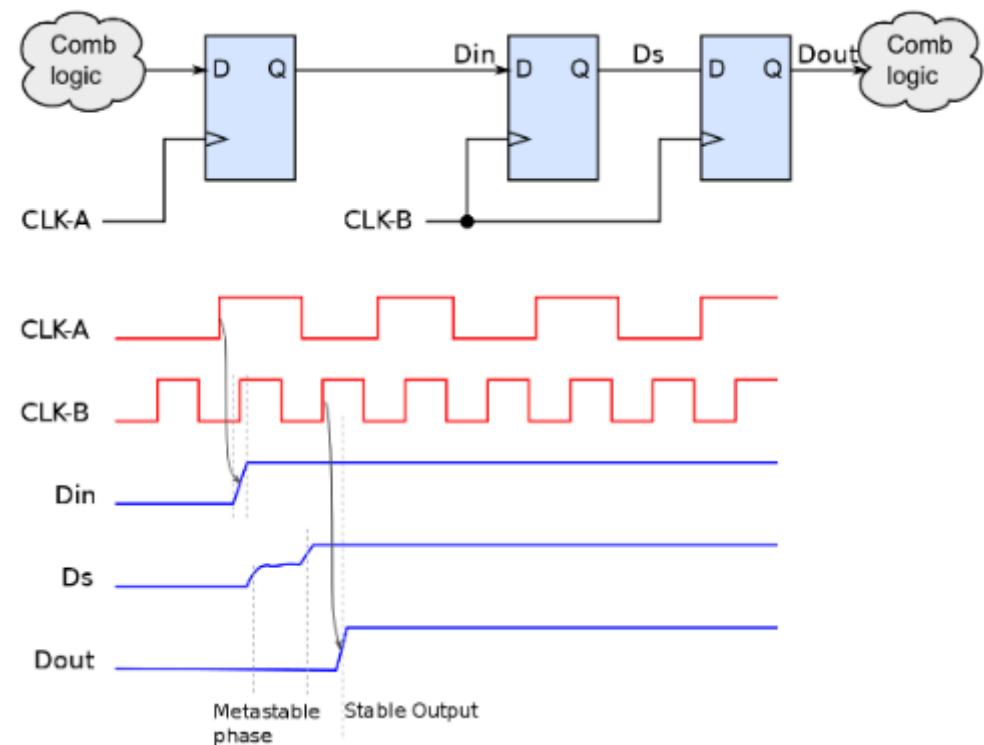
Review of Logic Circuits Timing Parameters (continued)

- **Note:** All the listed parameters are stochastic in reality (vary over time and space)
- In single clock designs, the clock frequency (f_C) is selected such that the slack requirement is met. The maximum clock reported by synthesis tools is based on such calculations
- In multiple clock designs, the timing cannot be guaranteed when crossing between **clock domains**
- **Result:** The output logic is not known (HIGH, LOW, or even a voltage in between). This is known as **metastability**

Metastability

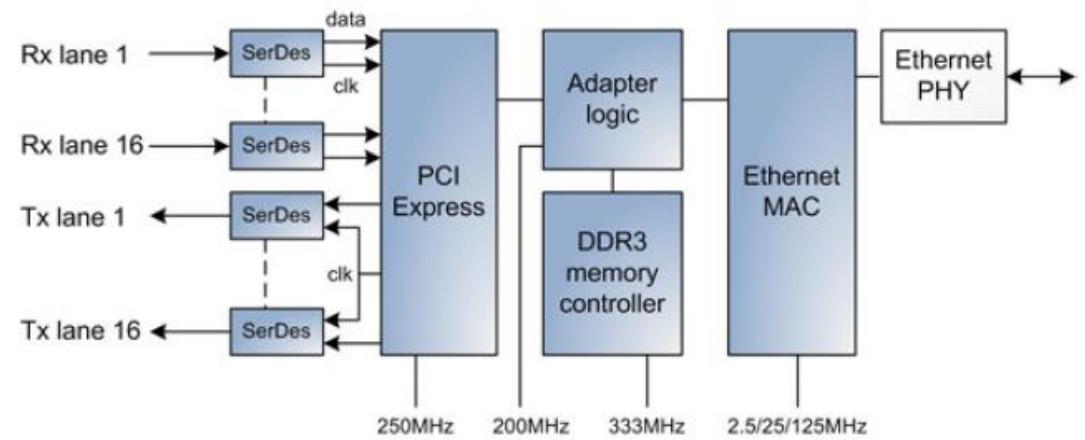
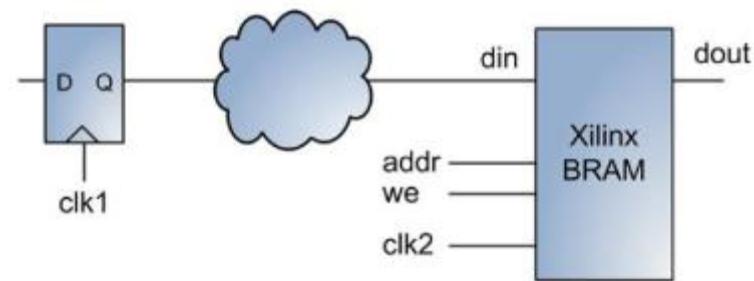
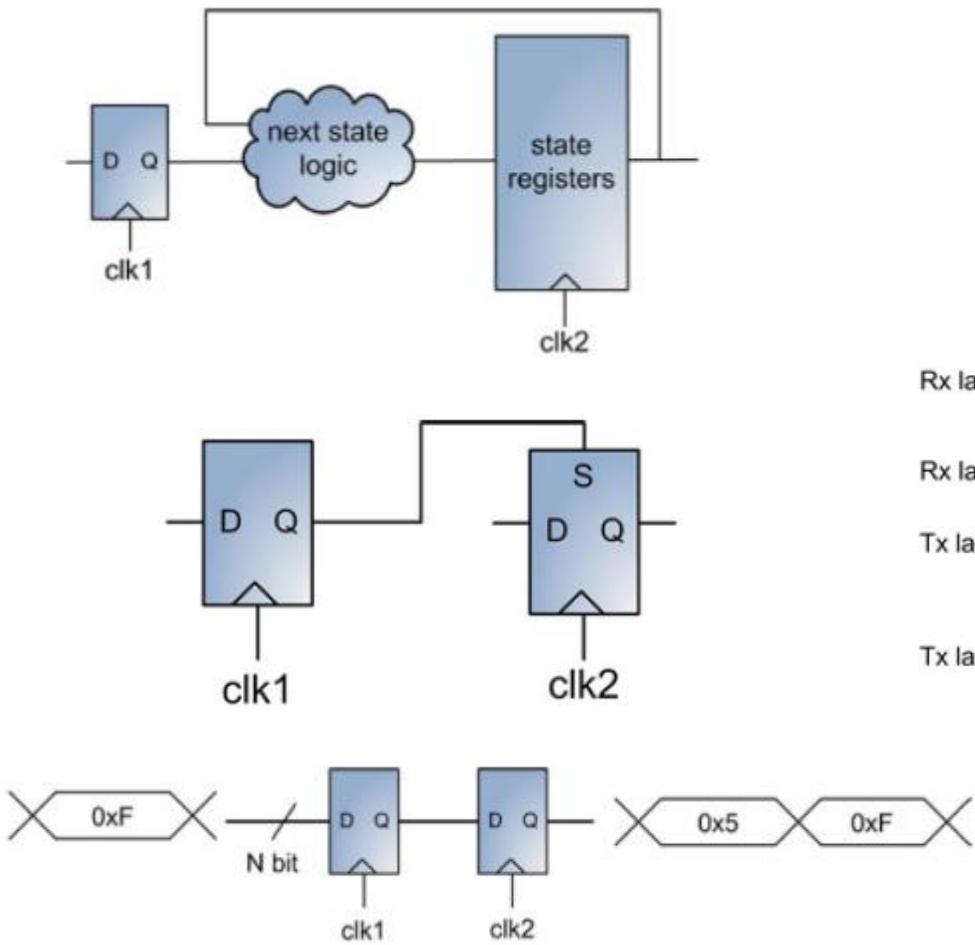
Metastability can occur when:

1. A flip-flop's slack timing is violated (high clock rate)
2. The data input to a flip-flop is asynchronous to the clock (leading to **setup** or **hold-time** violations)
3. When using multiple unsynchronized **clock domains**.



- During metastability t_{CQ} becomes longer than its nominal value.
- The additional time beyond t_{CQ} , which a metastable circuit requires to become stable is called the settling time (t_{MET})

Metastability Examples



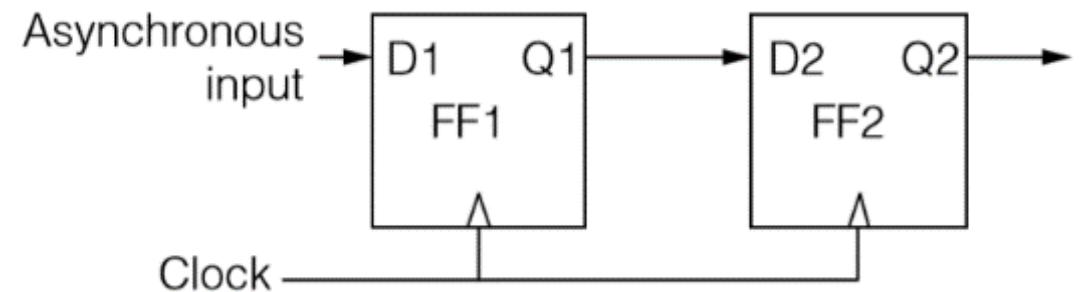
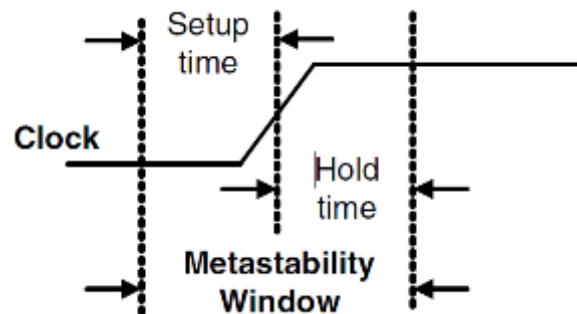
Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

Statistical Analysis of Metastability

How often does metastability occur?

Considering t_C as the FF clock period (inverse of f_C), t_D as the asynchronous data period (inverse of f_D), and w as the metastability window length:

- Considering the data transition probability to be uniform over the entire clock period and independent of the clock, the probability of data transition during a metastable window is $w/t_C = w \cdot f_C$
- Therefore, the rate of metastability is $w \cdot f_C \cdot f_D$ (times per seconds)



Statistical Analysis of Metastability (continued)

How long does it take to recover from metastability?

- It can be shown that the electronic properties of flip-flops eventually take it back a stable state (0 or 1)
- Assuming that a flip-flop becomes metastable at $t=0$, the probability of remaining in metastability after t_{MET} seconds has been shown to be (approximately) exponentially decaying over time, i.e.:

$$\Pr(\text{staying metastable} \geq t_{MET}) = e^{\frac{-t_{MET}}{\tau}}$$

where τ is a device and technology dependent parameter.

- Reference: Ginosar, Ran. "Metastability and synchronizers: A tutorial." *IEEE Design & Test of Computers* 28.5 (2011): 23-35.

Statistical Analysis of Metastability (continued)

Probability of Failure:

- If the output of a flip-flop is sampled t_{MET} seconds after the clock edge, the probability of failure (malfunction) is

$$\Pr(\text{failure}) = \Pr(\text{enter metastability AND stay metastable } t_{MET} \text{ or longer})$$

- The above two events are statistically independent. Hence:

$$\Pr(\text{failure}) = \Pr(\text{enter metastability}) \cdot \Pr(\text{stay metastable } t_{MET} \text{ or longer})$$

Mean Time Between Failures (MTBF) for Metastable Flip-Flops

The industrial standard formula for **Failure Rate** and **Mean Time Between Failures (MTBF)** of a single stage metastable flip-flop is:

$$\text{Failure Rate} = \frac{1}{\text{MTBF}} = f_D \cdot \Pr(\text{failure}) = f_D \cdot W \cdot f_C \times e^{\frac{-t_{\text{MET}}}{\tau}}$$

Metastable window probability
(how often we are in a metastable window)

The probability of remaining in
metastability for t_{MET} seconds

where:

- f_C : system clock rate (Flip-Flop clock)
- f_D : (asynchronous) input data clock rate
- W : metastability window length constant
- τ : metastability time constant
- t_{MET} : time delay for the metastability to resolve itself

Note: W and τ are constants depending on the **setup-time** and **hold-time** of the device (vendor and technology dependent)

MTBF Calculation

Example 1: Consider a 28nm ASIC high-performance CMOS with $W=20\text{ps}$ and $\tau=10\text{ps}$ (typical values for this process technology). Assuming $f_C=1\text{GHz}$ and $f_D=100\text{MHz}$, we find $\text{MTBF}=4\times10^{29}$ years for a single-stage synchronizer (the universe is estimated to be 10^{10} years old).

MTBF Calculation (continued)

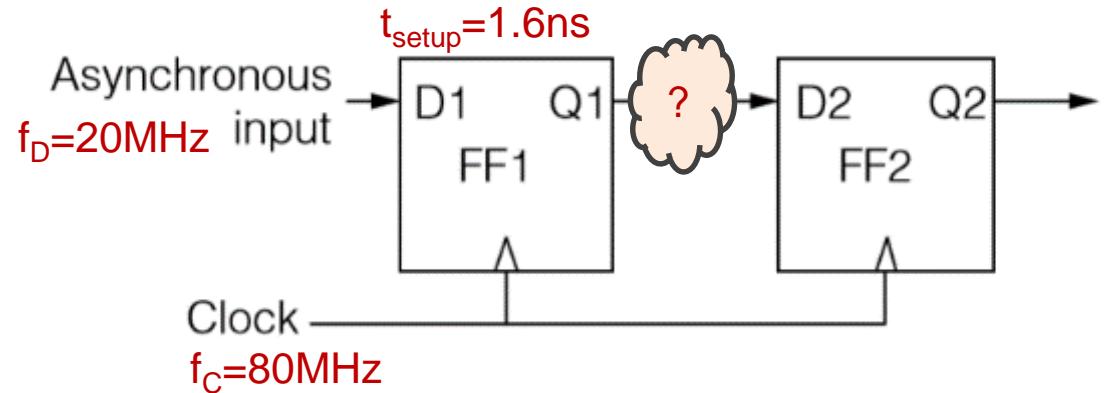
Example 2: Suppose we want to guarantee a 1year MTBF (approximately 3×10^7 s) on an Altera FLEX 10K CPLD. The MTBF constants of this family of Altera devices can be seen in the table below. In certain devices of this family $t_{\text{setup}} = 1.6\text{ns}$. For a data frequency $f_D = 20\text{MHz}$ and clock frequency $f_C = 80\text{MHz}$ we have:

$$t_{\text{MET}} = \frac{\ln(3 \times 10^7) + \ln[(80 \times 10^6)(20 \times 10^6)(1.01 \times 10^{-13})]}{1.268 \times 10^{10}} = 1.76\text{ns}$$

In this example the combination circuit shown in the figure can have the following maximum combinational delay to fulfil the required MTBF:

$$t_{\text{logic}} \leq 12.5\text{ns} - 1.76\text{ns} - 1.6\text{ns} = 9.14\text{ns}$$

Note: Due to the logarithmic form of the equation, increasing t_{MET} to 2.12ns increases the MTBF to 100 years.



Device	W	$1/\tau$
FLEX 10K	1.01×10^{-13}	1.268×10^{10}
FLEX 8000	1.01×10^{-13}	1.268×10^{10}
FLEX 6000	1.01×10^{-13}	1.268×10^{10}
MAX 9000	2.98×10^{-17}	5.023×10^9
MAX 7000	2.98×10^{-17}	5.023×10^9

MTBF of Multistage Synchronizers

For multistage synchronizers:

$$\text{MTBF} = \frac{1}{W \cdot f_C \cdot f_D} \times e^{\frac{t_{\text{MET}_1}}{\tau}} \times e^{\frac{t_{\text{MET}_2}}{\tau}} \times \dots$$

where t_{MET_1} , t_{MET_2} , etc. are the time delay for the metastability to resolve itself in each synchronizer stage.

How many synchronizer stages are required? The parameters W and τ are commonly provided by IC manufacturers; f_C and f_D are also known by-design. The designer can define a desired MTBF, calculate t_{MET} and decide about the number of required stages to fulfil the required MTBF.

Metastability Guidelines

Avoiding metastability (by design):

1. Avoiding real-time data transfer between different clock domains
2. Using a single global clock instead of multiple clock domains
3. Avoiding gated clocks and using standard clock decreasing techniques (using clock enable)

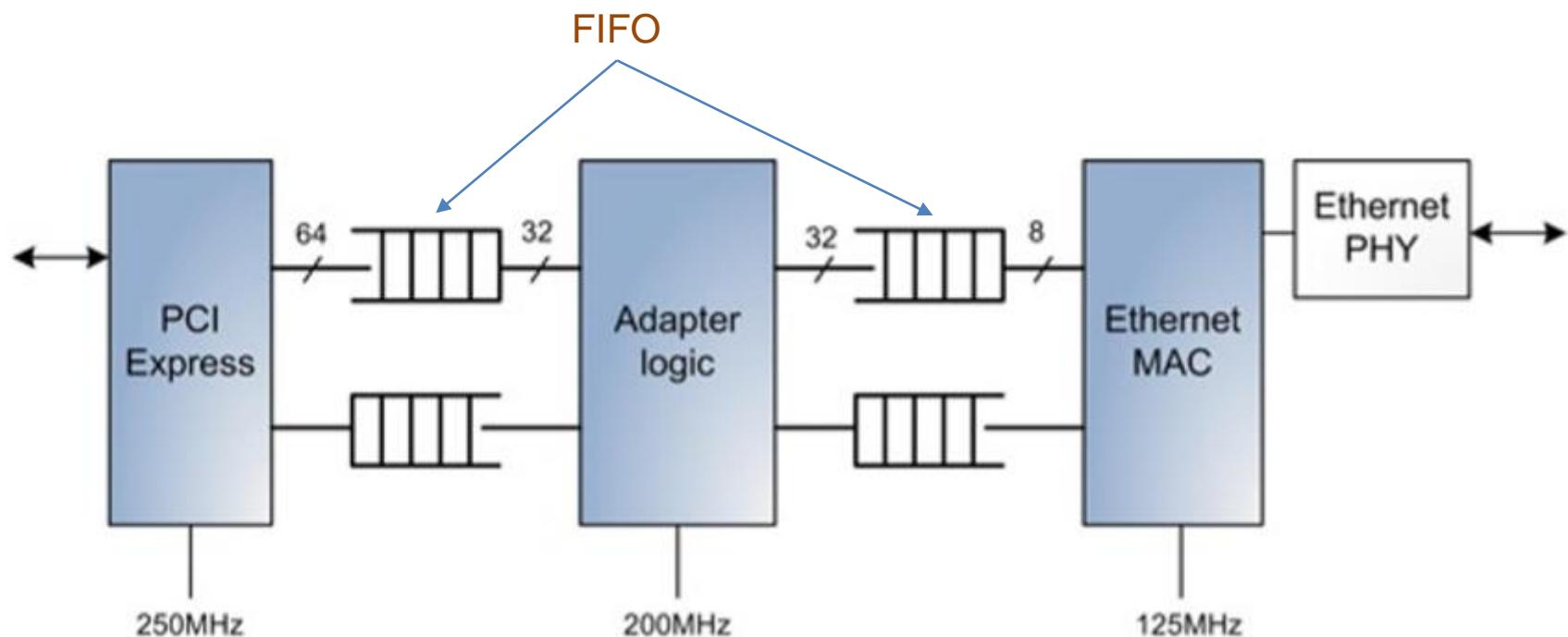
Solving metastability (by implementation):

1. Clock synchronization using DCMs
2. Using synchronizers (register chains and asynchronous FIFOs) to reduce the probability of metastability

Note: These methods only resolve metastability; but do not solve other rate mismatch issues, when transferring data between different clock domains. For example, sampling a data that changes with $f_D=80\text{MHz}$, at a clock rate of $f_C=100\text{MHz}$, results in regular repeated samples and sampling it at $f_C=60\text{MHz}$ results in regular data loss (even without metastability).

Metastability Guidelines (continued)

Example: Using FIFOs while crossing different clock domains

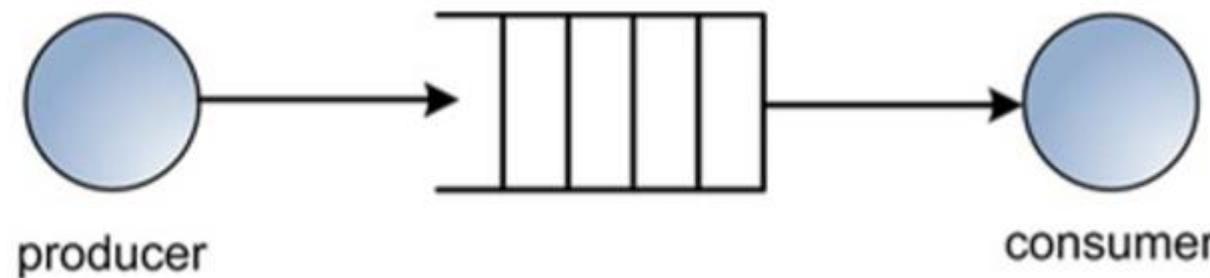


Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

FIFO Size Selection

How to select the FIFO size? The overall producer data rate should not exceed the consumer rate of processing the data.

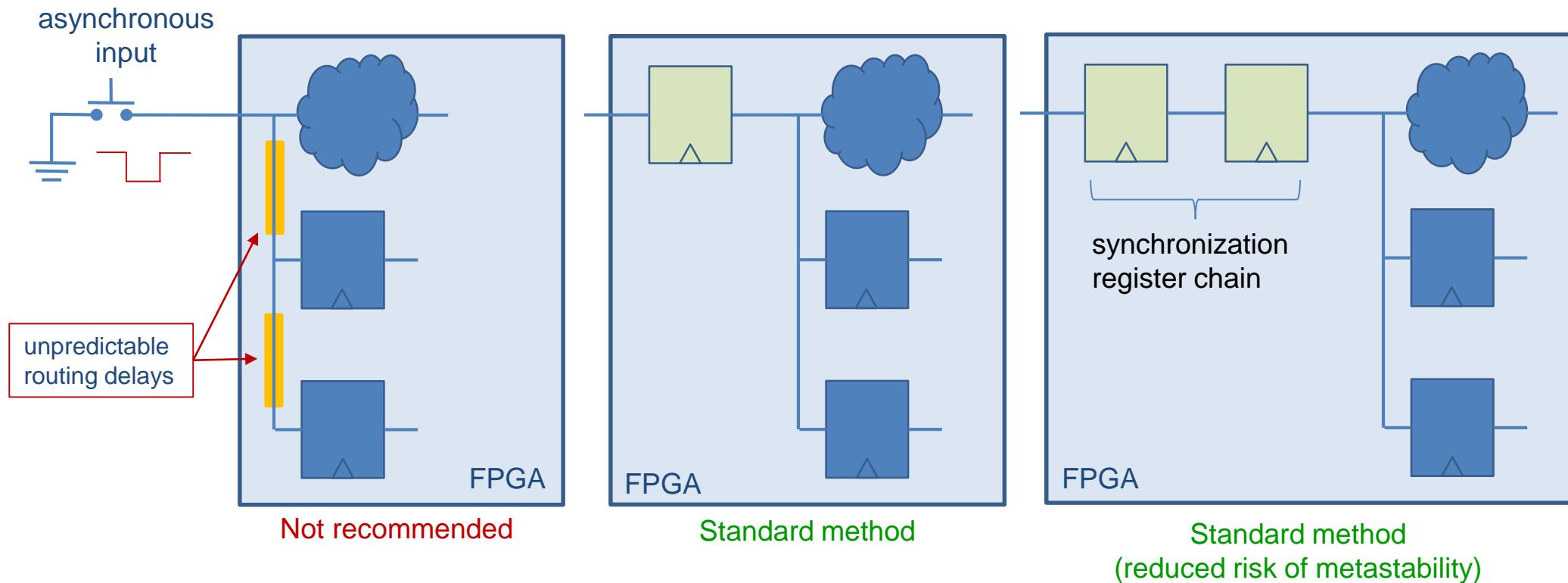
Note: A FIFO can not overcome rate differences (no matter how deep it is, it'll eventually overflow if the producer's data rate is consistently higher than consumer's). A FIFO can only overcome temporary producer-consumer rate differences by buffering the excess data.



Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

Applications: Metastability due to Top-Module Asynchronous Inputs

The standard procedure for working with top-module (asynchronous) inputs is to pass them through one or more layers of flop-flops before any internal usage.

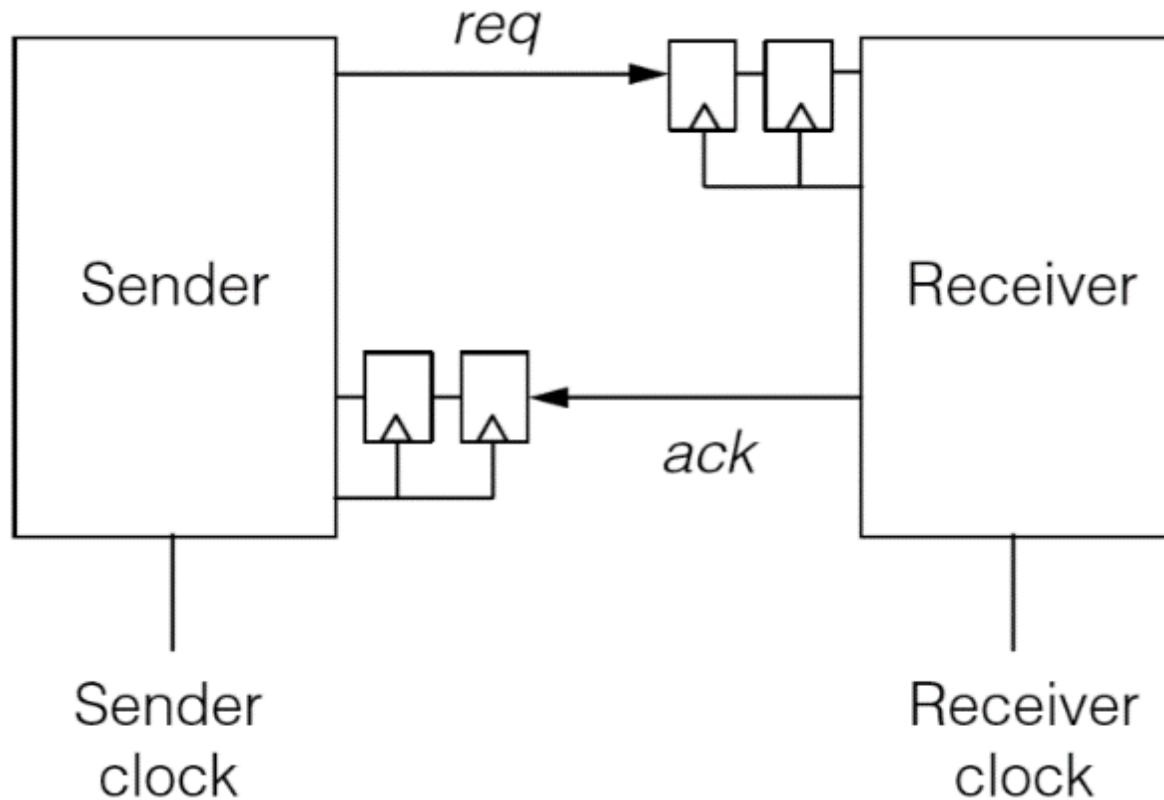


Note: The probability of metastability decreases by increasing the number of FF layers

Question: How to handle asynchronous input buses (group of asynchronous inputs)?

Answer: By placing user defined constraints over the bus routing length.

Applications: Metastability in Two-Way Control/Acknowledge Systems



- Reference: Ginosar, Ran. "Metastability and synchronizers: A tutorial." *IEEE Design & Test of Computers* 28.5 (2011): 23-35.

Flip-Flop MTBF in Xilinx FPGA

Example: Xilinx Virtex II, metastability datasheet

Table legend:

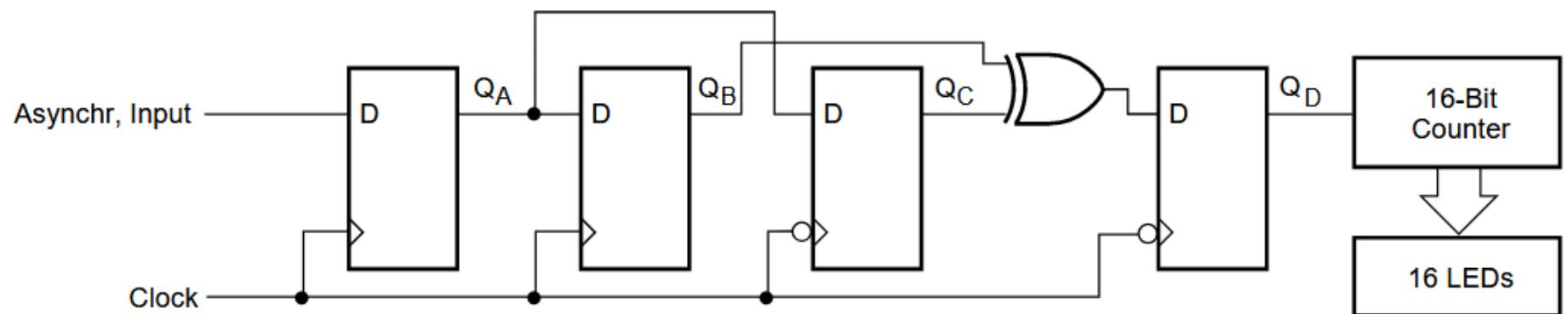
$$\text{MTBF} = \frac{e^{K2 * \tau}}{F1 * F2 * K1}$$

Device	XC2VP4			XC2VP4			XC4005E –3
V _{CC} (V)	1.5	1.35	1.65	1.5	1.35	1.65	5.0
Flip-Flop Type	CLB	CLB	CLB	IOB	IOB	IOB	CLB
Low Frequency (MHz)	300	310	390	310	300	420	109
Half Period (ps)	1667	1613	1283	1613	1667	1190	4587
MTBF1 (ms)	60,000	20,000	60,000	30,000	30,000	30,000	1,000
High Frequency (MHz)	390	420	490	420	430	500	124.4
Half Period (ps)	1282	1190	1020	1190	1163	1000	4019
MTBF2 (ms)	1.69	1.046	5.16	0.987	1.84	6.96	0.016
Half Period Difference (ps)	385	423	262	423	504	190	568
Ln (MTBF1 / MTBF2)	10.478	9.86	9.36	10.322	9.70	8.37	11.09
K2 (per ns)	27.2	23.3	35.7	24.4	19.24	44.05	19.52
1 / K2 = tau (ps)	36.8	42.9	28.0	41.0	52.0	22.7	51.2
MTBF multiply / 100 (ps)	15.2	10.3	35.6	11.5	6.85	81.8	7.04

Ref: https://china.xilinx.com/support/documentation/application_notes/xapp094.pdf

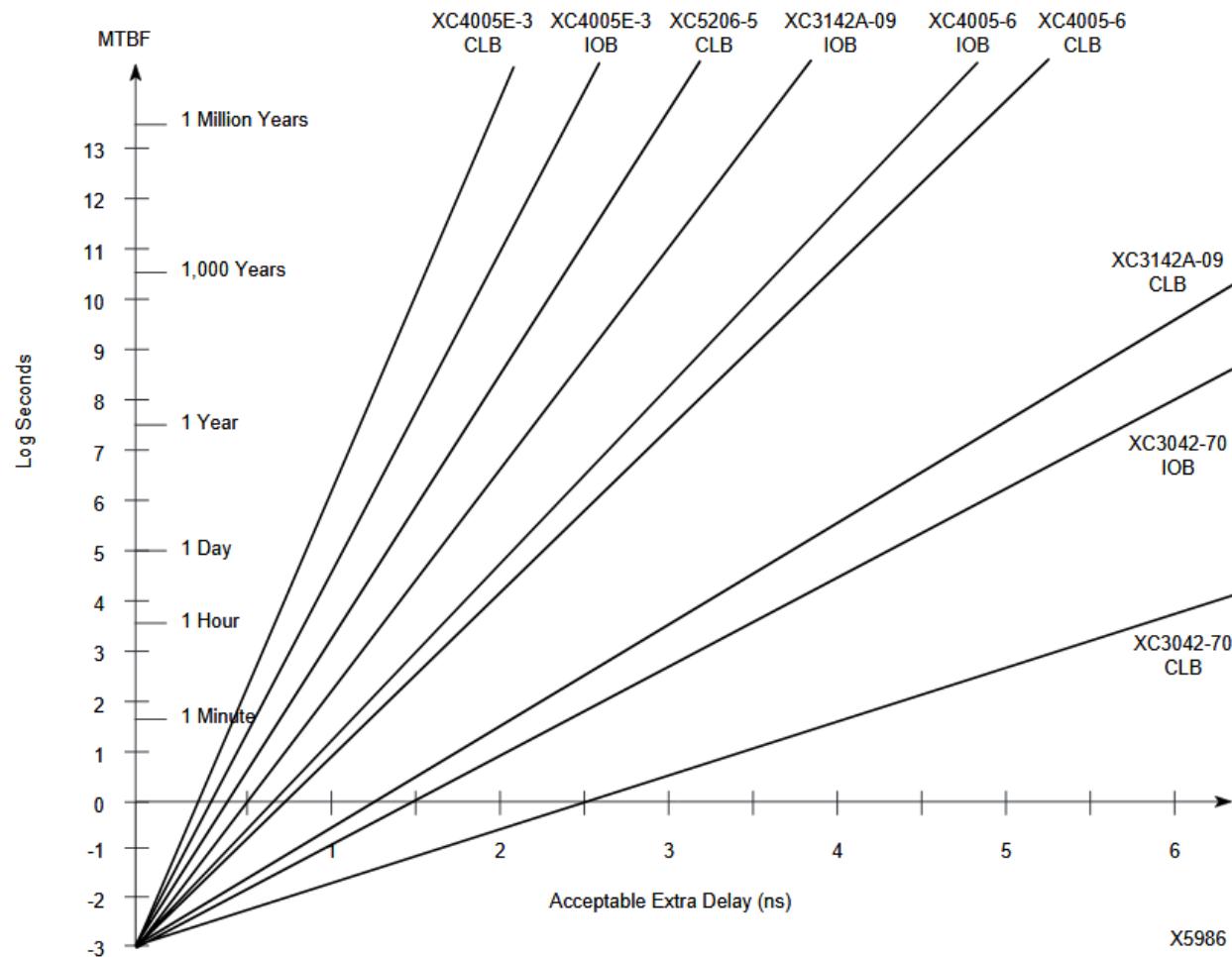
Note: Xilinx doesn't seem to list the FF MTBF of its newer devices; but it reports them in Vivado® during implementation.

Xilinx's Metastability Test Circuit



Ref: Xilinx Metastability Considerations (XAPP077.pdf January 1997, Available:
<http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf>)

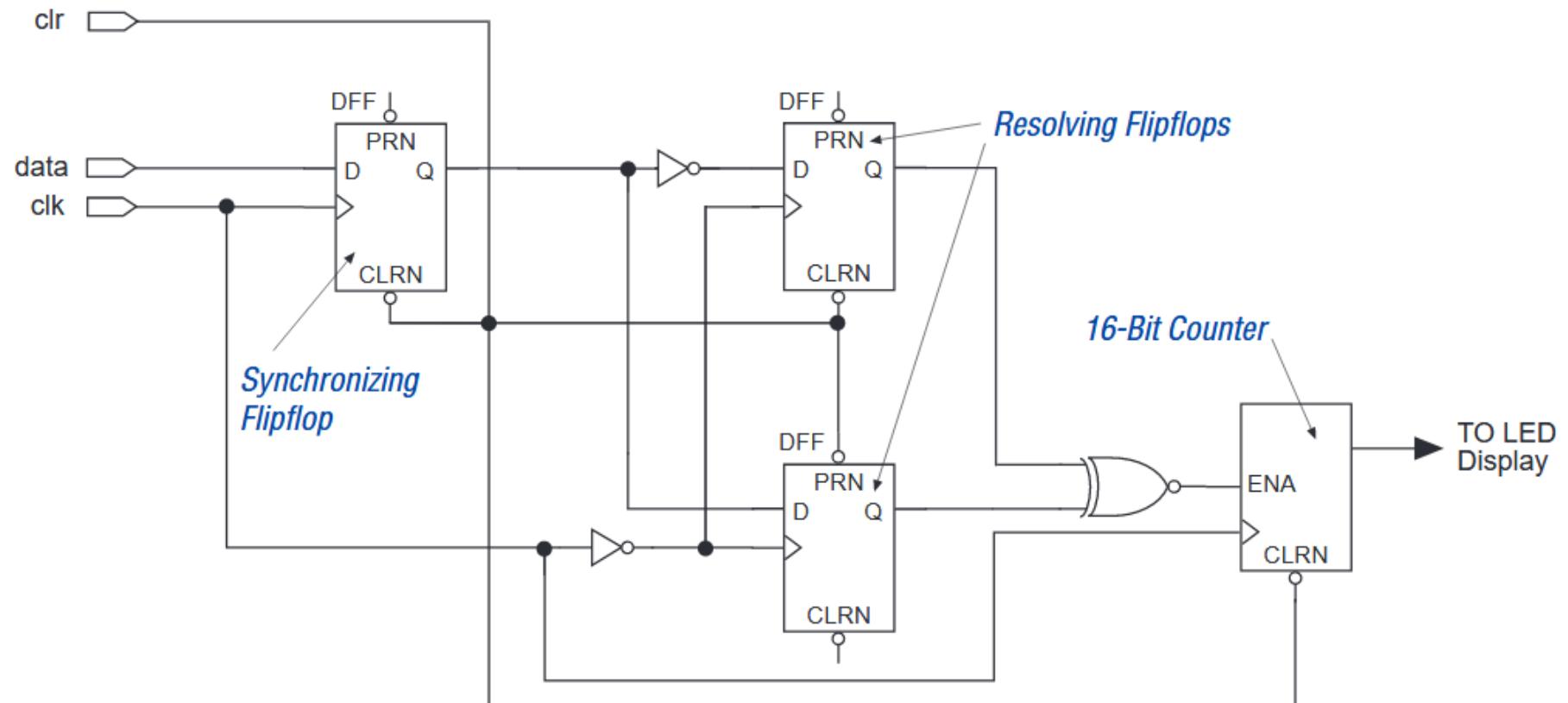
Xilinx's Metastability Test Results



Ref: Xilinx Metastability Considerations (XAPP077.pdf January 1997, Available:
<http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf>)

Altera's Metastability Test Circuit

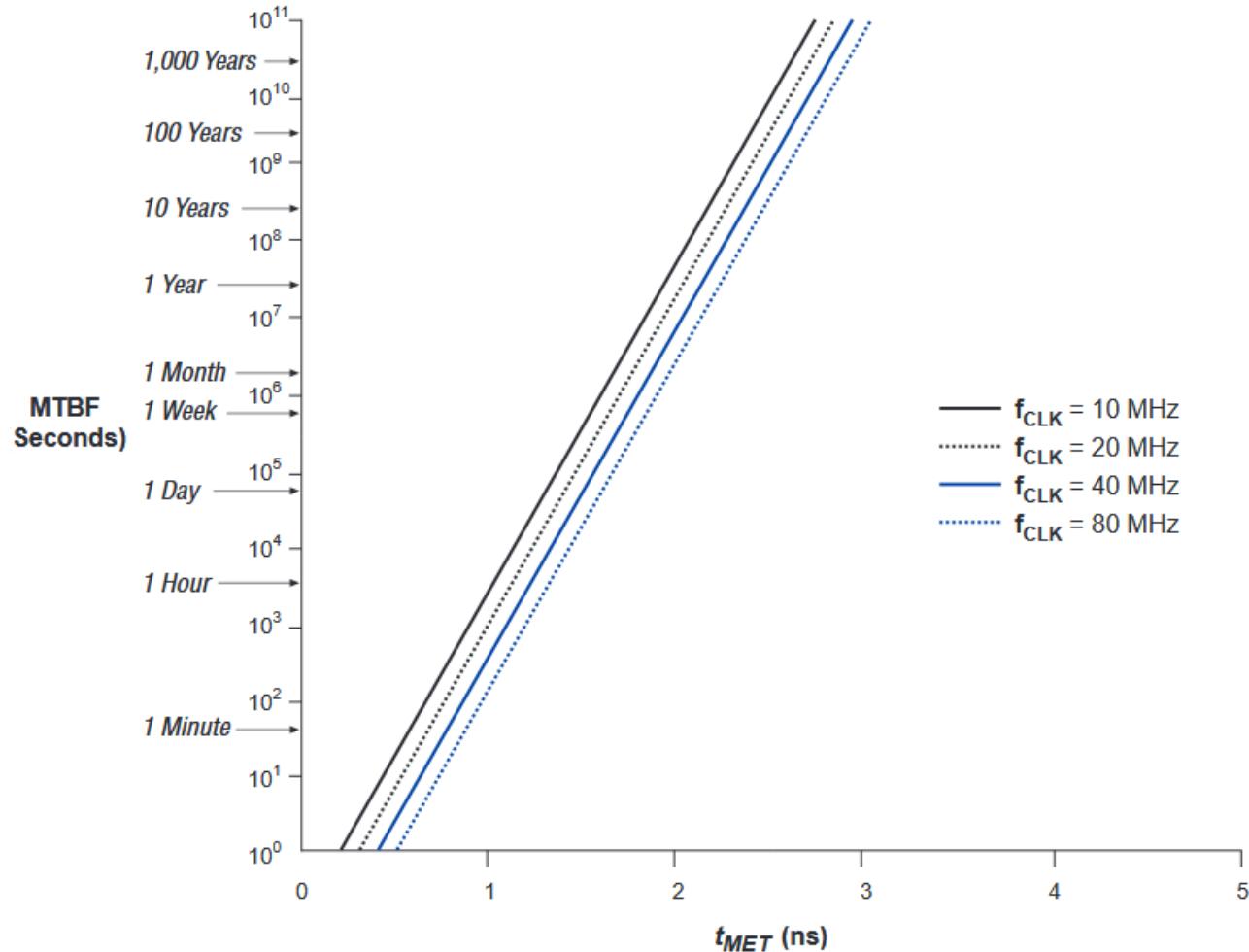
All logic is part of the device under test.



Ref: [ftp://ftp.altera.com/pub/lit_req/document/an/an042.pdf](http://ftp.altera.com/pub/lit_req/document/an/an042.pdf)

Altera's Metastability Test Results

Figure 6. FLEX 10K, FLEX 8000 & FLEX 6000 MTBF Values



Further Readings on Metastability

- Kilts, Steve. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007.
- Arora, Mohit. *The art of hardware architecture: Design methods and techniques for digital circuits*. Springer Science & Business Media, 2011.
- Ginosar, Ran. "Metastability and synchronizers: A tutorial." *IEEE Design & Test of Computers* 28.5 (2011): 23-35.
- <http://www.ti.com/jp/lit/an/scza004a/scza004a.pdf>
- <http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf>
- https://www.altera.com/en_US/pdfs/literature/wp/wp-01082-quartus-ii-metastability.pdf

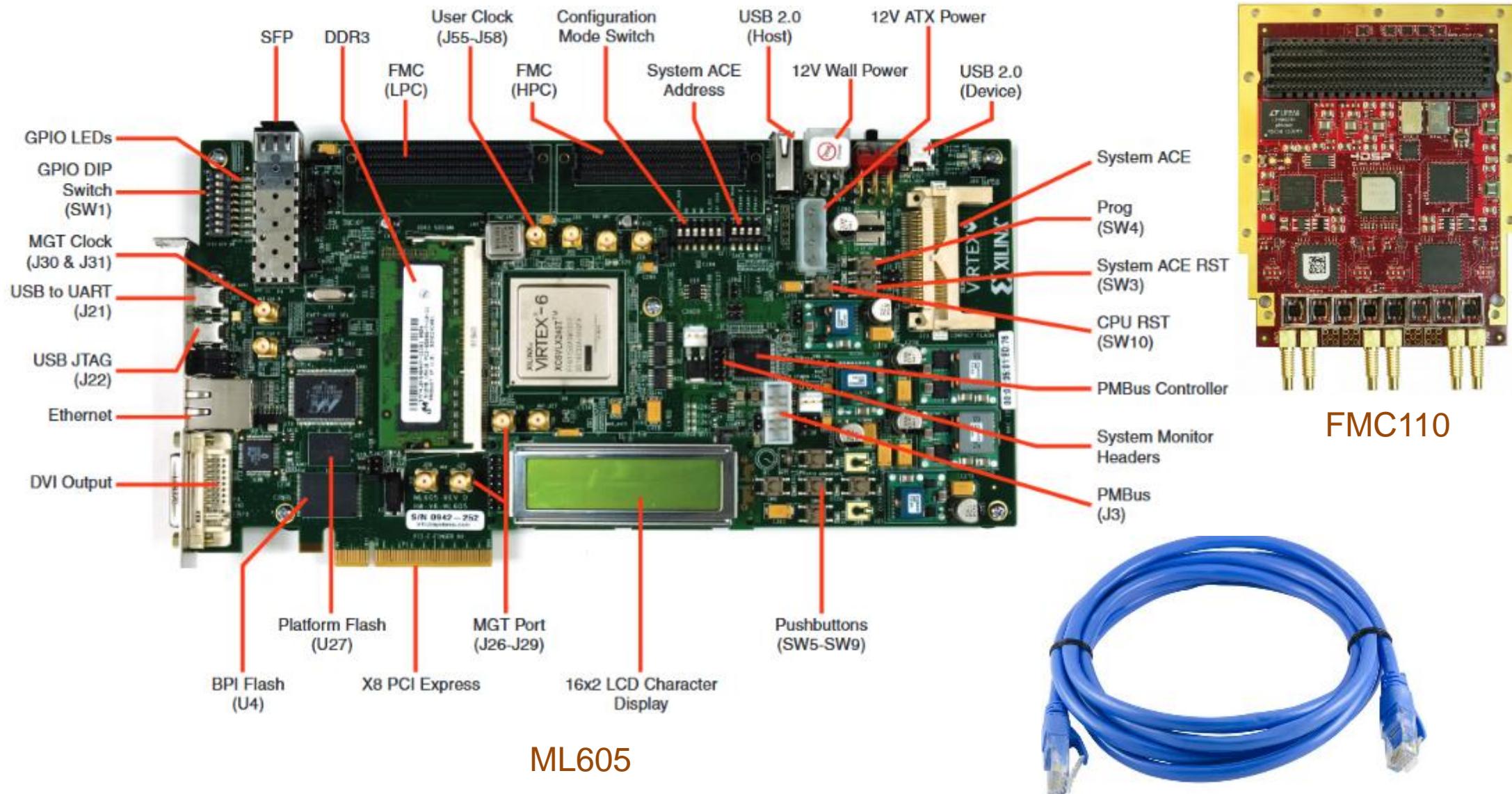
MEMORY-MAP DESIGN IN FPGA-BASED SYSTEMS

Introduction**

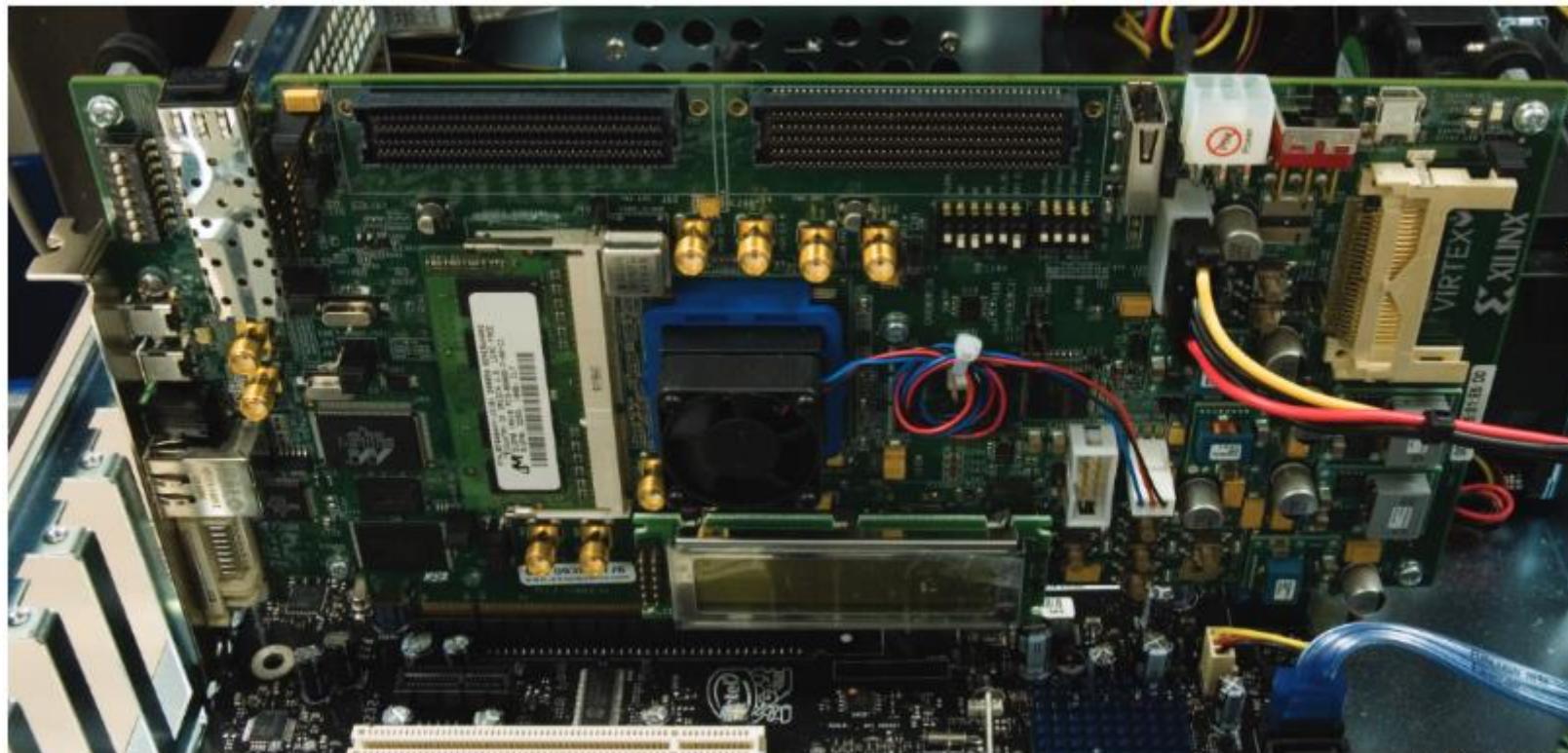
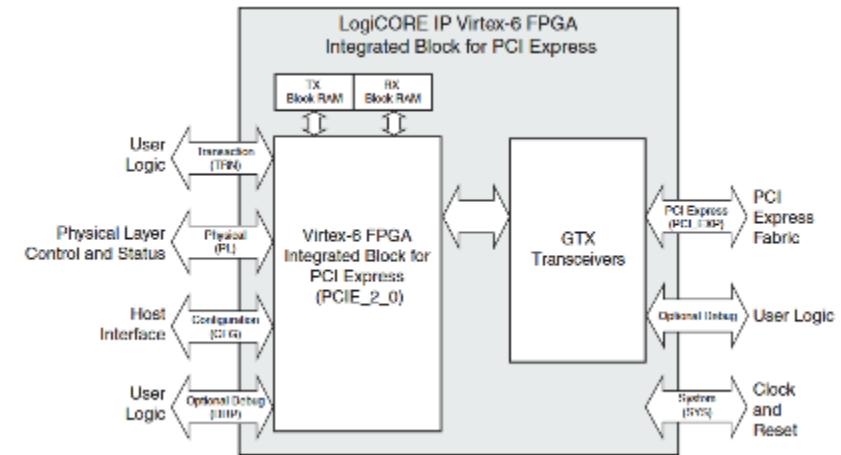
- Complex FPGA-based systems can contain multiple units (modules), each having multiple operation modes that are selected by appropriate control pins (or control bus) and give output messages in different occasions (handshakes, error codes, overflow flags, etc.)
- Each element of a design should have a unique address in the system's **memory map**, which can be accessed via proper commands
- In mixed CPU-FPGA systems, the internal memory map of the FPGA is commonly accessible by the software units
- The design of a memory map is discussed in this section by examples

**This section is presented from industrial project source codes

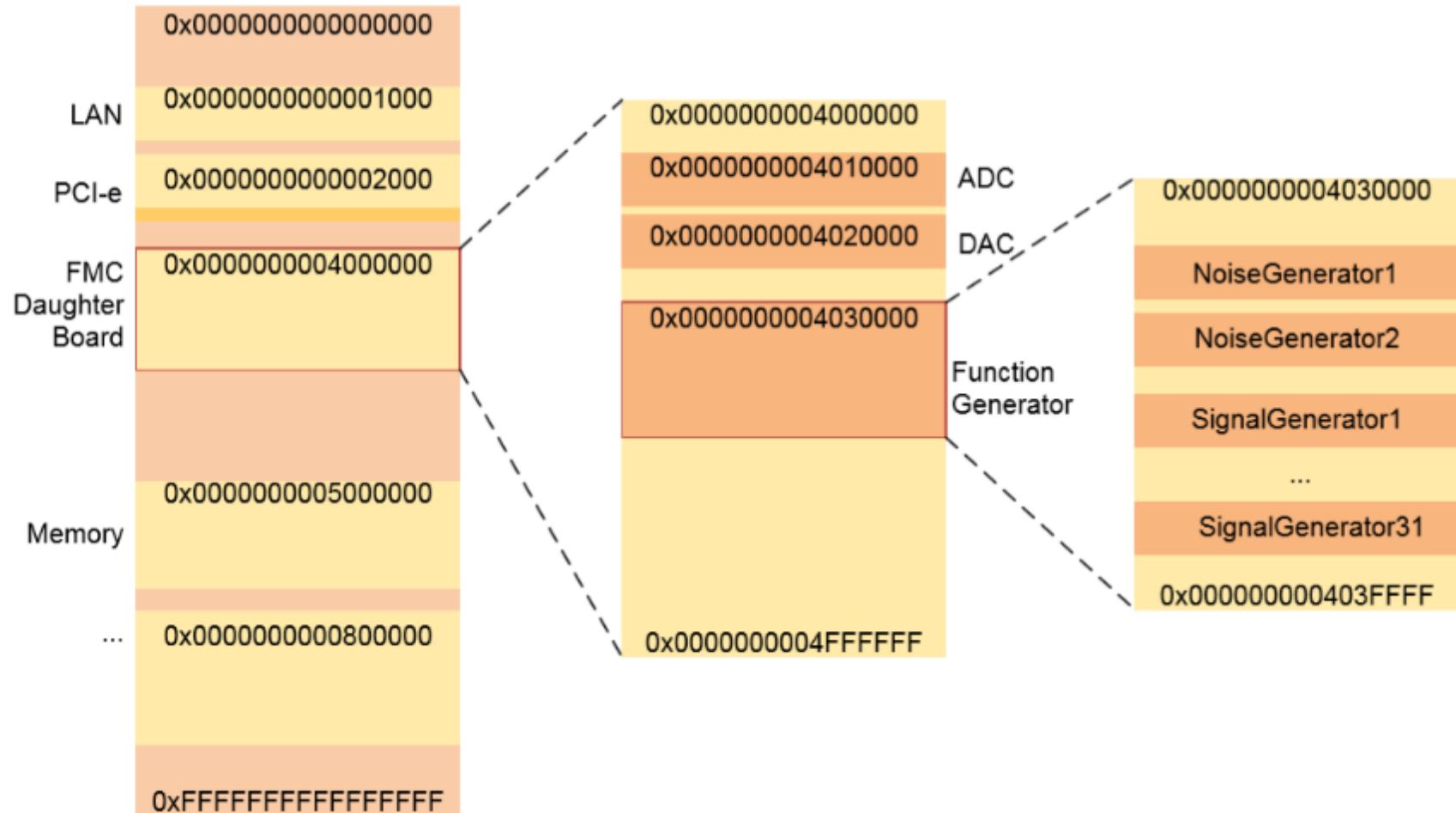
Example: Xilinx ML605 Virtex-6 Evaluation Board



Example: Xilinx ML605 Virtex-6 Evaluation Board (continued)



Memory Map



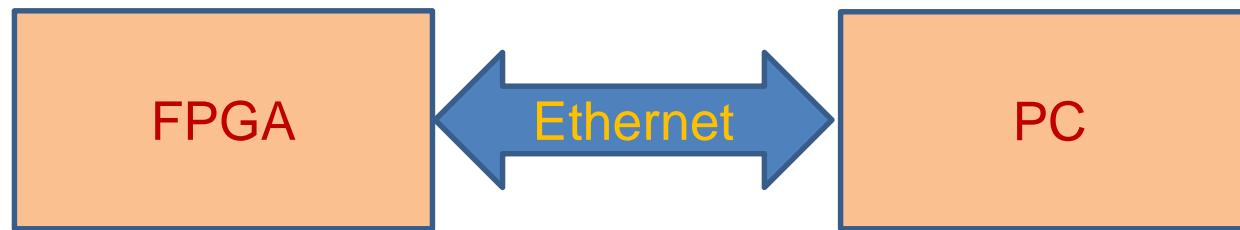
Memory map conceptual illustration

Accessing the Memory Map

- The internal memory map (MMap) of the FPGA and the protocol for accessing the MM is designed and implemented by the FPGA designer
- The MMap can be accessed through any of the I/O ports of the FPGA board. For example:
 - Ethernet
 - PCI-e
 - JTAG
 - USB
 - Etc.

Accessing the Memory Map (continued)

- **Example:** Suppose that we use the Ethernet as the access port of the ML605 FPGA board. The FPGA board can send and receive Ethernet packets, which can have an arbitrary format after decoding:



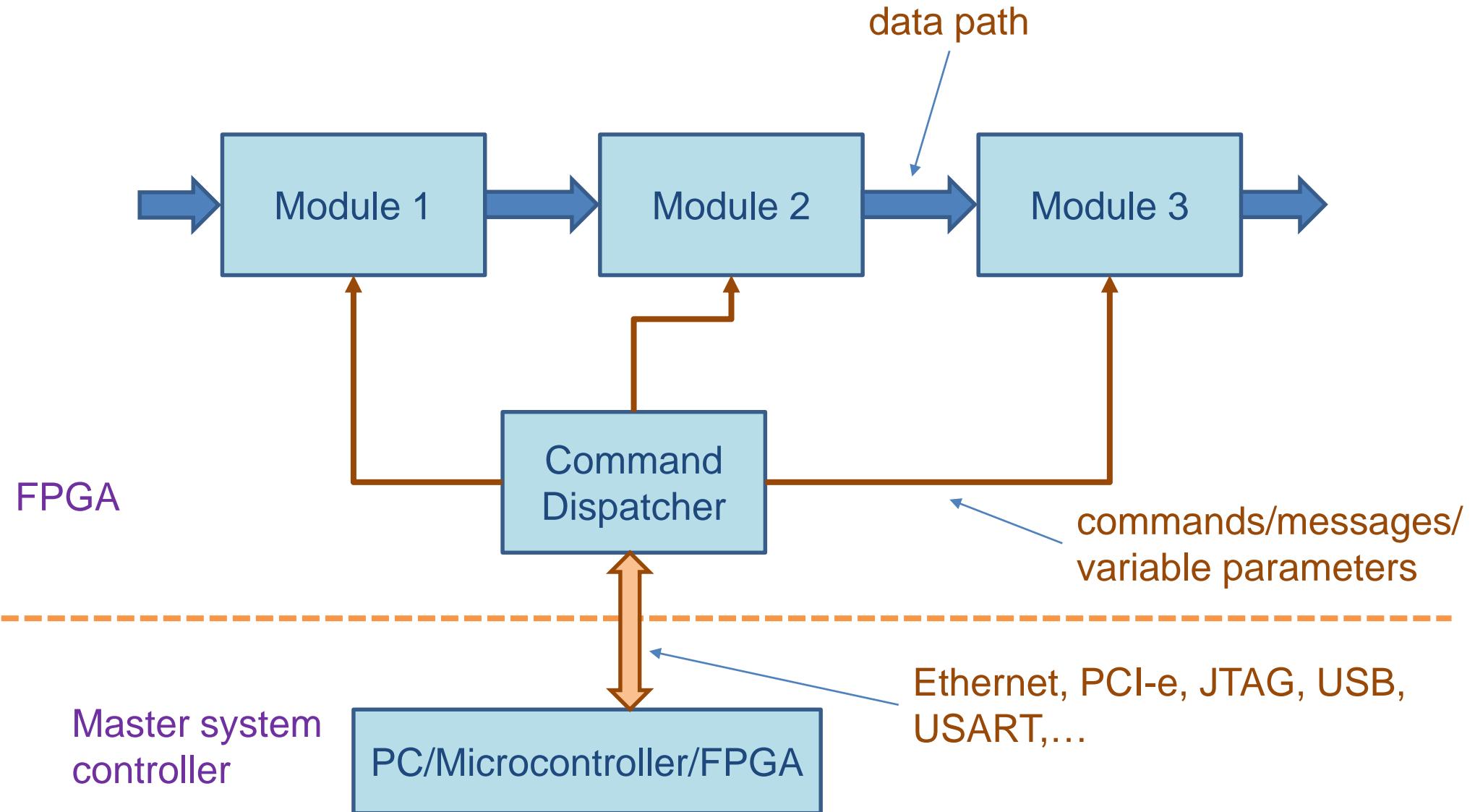
Decoded Ethernet packets (arbitrary format defined by the designer):

flags	address	data
p_{N-1}		p_0

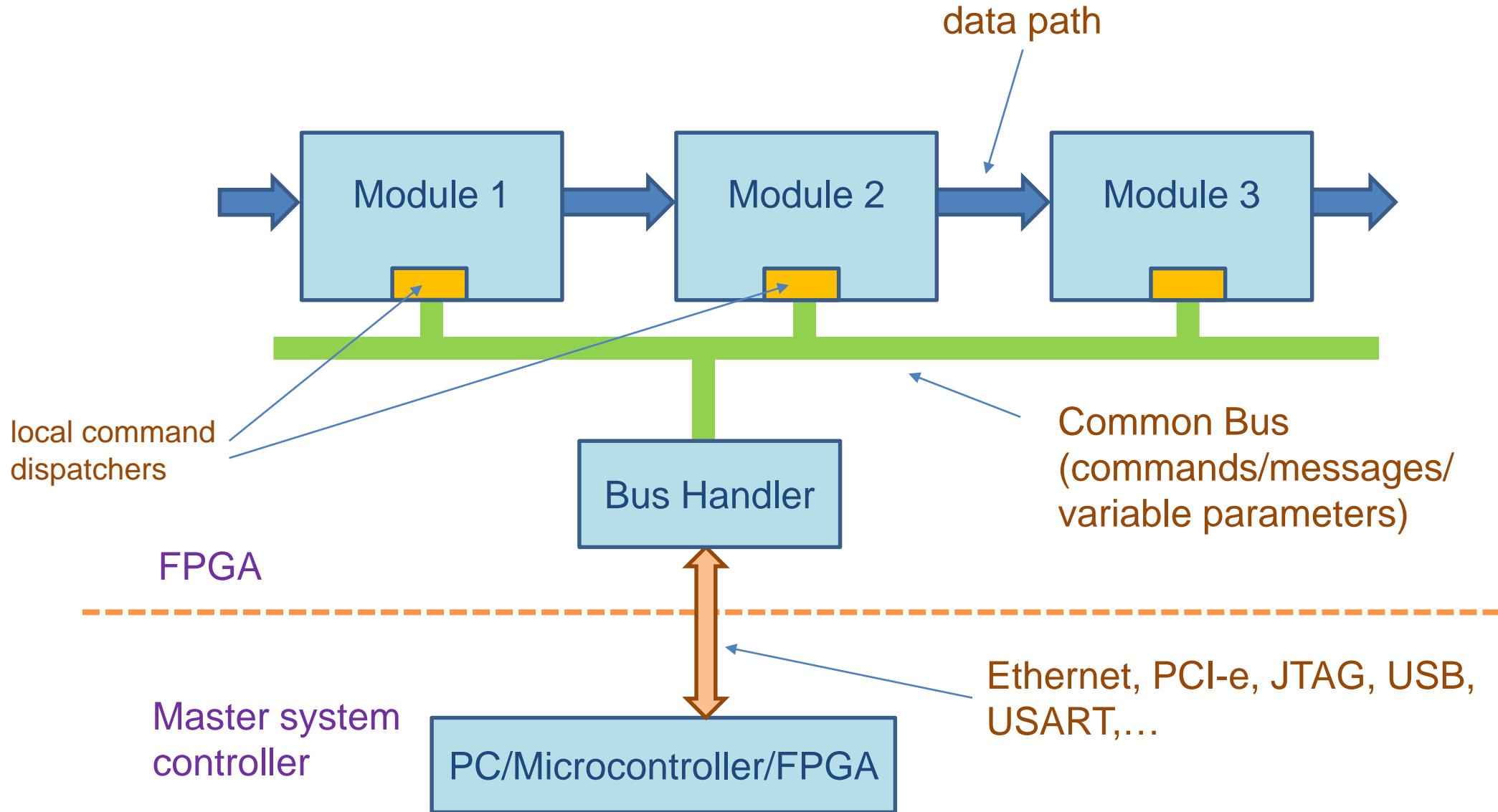
Memory Map Implementation Techniques

- **Centralized:** All modules have a set of input ports for commands and output ports for handshaking and messages. All input commands (to the FPGA) or output messages (from the FPGA) are handled by a single module (a command or message dispatcher), which has access to the command ports of all modules. The only command/message interface of the FPGA to the output world is this module.
- **Distributed:** There are no centralized command/message dispatchers. A common command bus is shared between all modules. Each module has a unique address (or address offset with respect to the top-module, for nested modules) in the memory map of the system. The commands/messages are handled locally by each module.

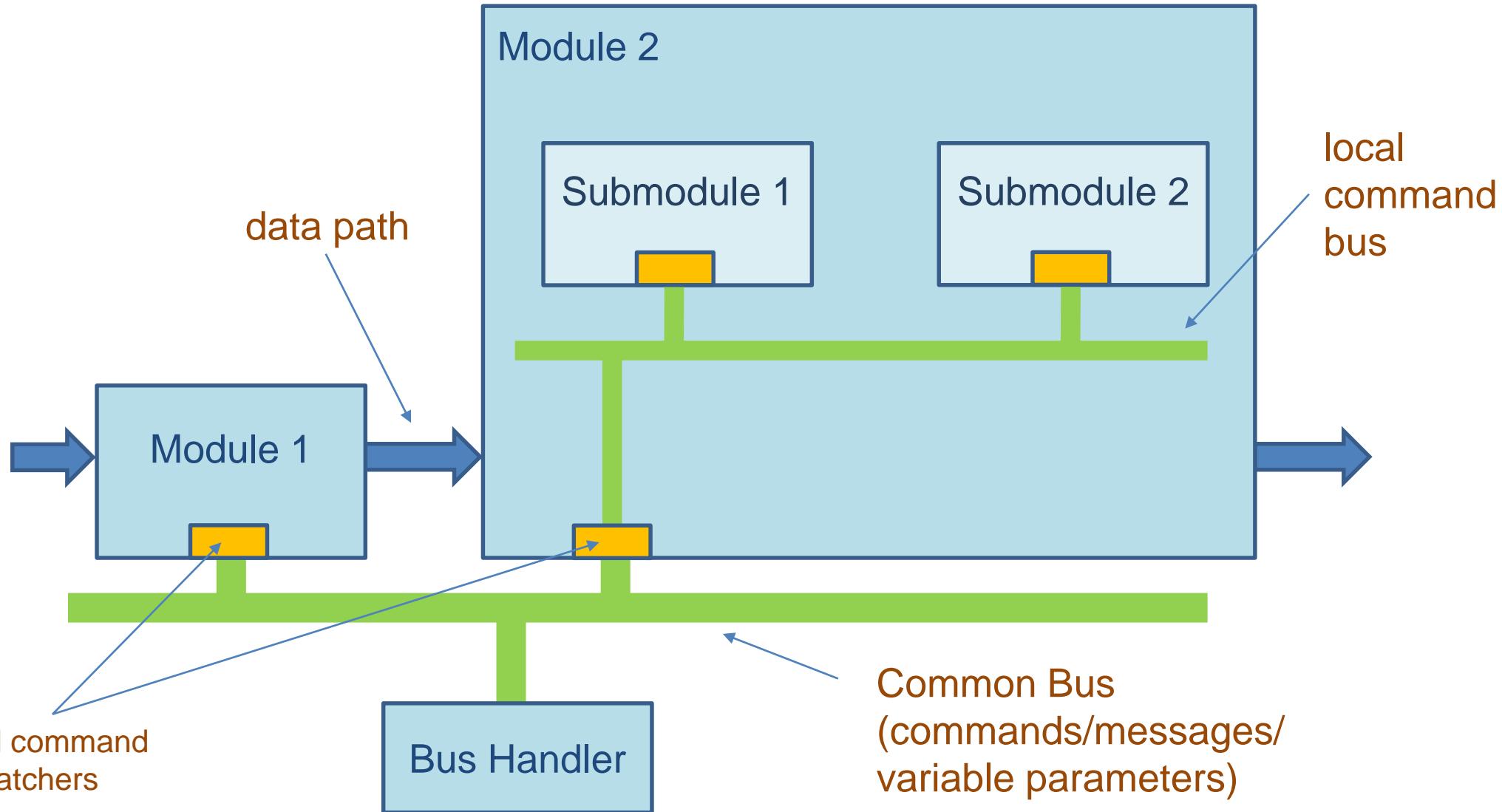
Centralized Memory Map Design



Distributed Memory Map Design



Nested Memory Maps



Centralized vs Distributed Memory Maps

	Advantages	Drawbacks
Centralized	<ul style="list-style-type: none"> • Less prone to design errors and bus write conflicts (centralized command dispatching) • Simpler for constructing the memory map (explicit memory map addresses) • No local command dispatchers ✓ Recommended for small and medium size designs 	<ul style="list-style-type: none"> • All command/message ports appear as input/output ports of modules (more complication in the top-module)
Distributed	<ul style="list-style-type: none"> • Simplified top-module • No centralized command dispatchers required • Simpler for extension (similar module instances can be added to the design in a “plug-and-play” like manner) ✓ Recommended for complicated designs with possible future extensions 	<ul style="list-style-type: none"> • More prone to design errors and bus handling by individual modules • More complicated memory map encoding/decoding • Each module requires a command dispatcher

DATA COMMUNICATION METHODS & PROTOCOLS

Introduction

- As with other aspects of FPGA designs, **data transfer** inside FPGA and between FPGA systems can be fully customized.
- In this section we review the most common techniques used for data transfer in FPGA designs
- The two classes of data transfer methods that we study are:
 - Stream Transfer
 - Packet Transfer

Continuous Stream Data Transfer

- **Stream Transfer:** used for continuous and synchronous data transfer between modules
- **Usage:** ADC, DAC, continuous data streams
- **Advantage:** no handshaking overheads; can use the maximum possible throughput between two endpoints
- **Disadvantage:** requires synchronization; even minor asynchrony between the sender and receiver clocks results in metastability, data replication or data loss
- **Note:** depending on the processing algorithm, continuous data streams can be **up-sampled** or **down-sampled** throughout processing

Packet (Block) Data Transfer

- **Packet (Block) Transfer:** used for discrete data transfer between modules
- **Usage:** data/message communication between asynchronous modules
- **Advantage:** enables data transfer between different clock domains; robust to minor sender/receiver clock frequency mismatch (depending on the block size)
- **Disadvantage:** requires handshaking, packing overhead (start/stop/CRC words), reduced bandwidth and packing/unpacking hardware overheads

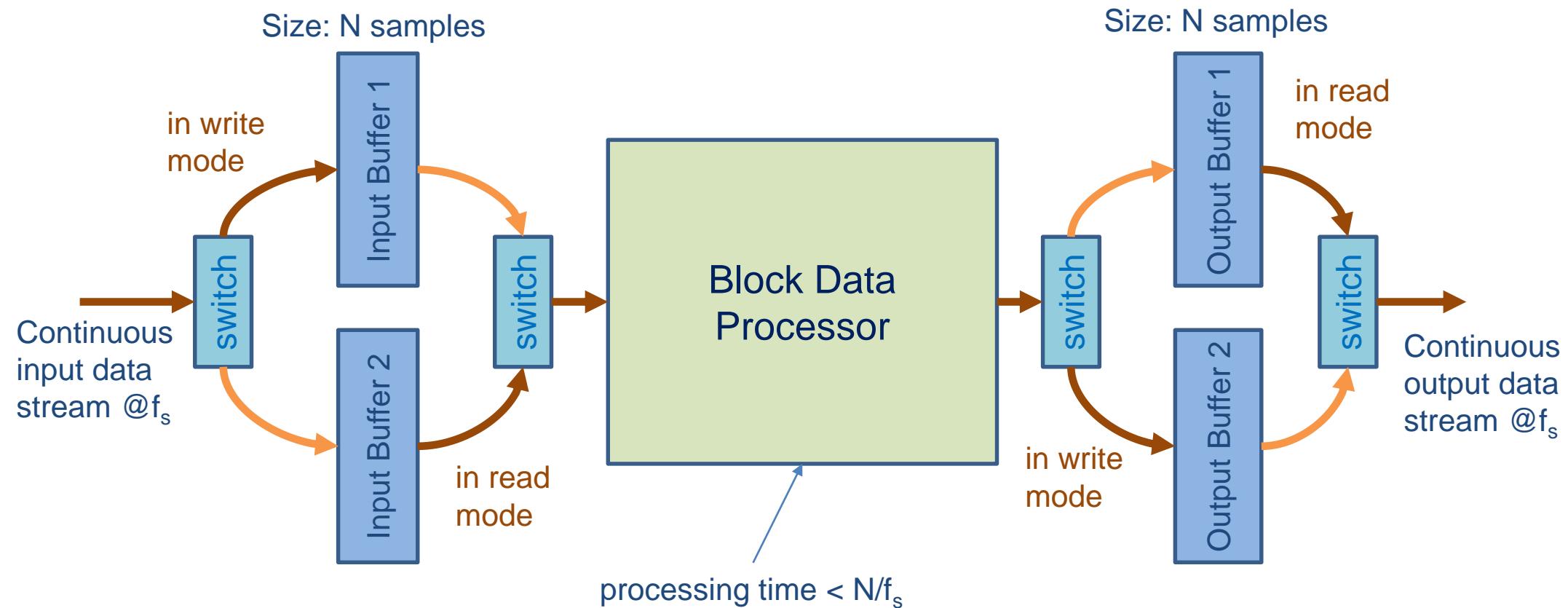
Block Processing of Streamed Data

- A common requirement in many data processing systems is the **block-wise processing of continuous data streams**. Examples include: DFT filtering, Reed-Solomon encoding, H.264 encoding, etc.
- The standard technique for implementing such algorithms is to use a dual-buffer at the interface between the continuous data stream and the block processor.
- As a rule if the block-wise algorithm processes a block of data faster than the data stream is accumulated in the input buffer (and read from the output buffer), no data loss occurs in the input (or output) and the block processing is masked from the outer world.

Block Processing of Streamed Data

(continued)

Dual-buffer implementation: When input is streamed in InBuff1, the block processor is working on previous data written in InBuff2. When the block processor is downloading its results in OutBuff2, the previous results are streamed from OutBuff1 to the output, etc.



The ARM Advanced Microcontroller Bus Architecture (AMBA)

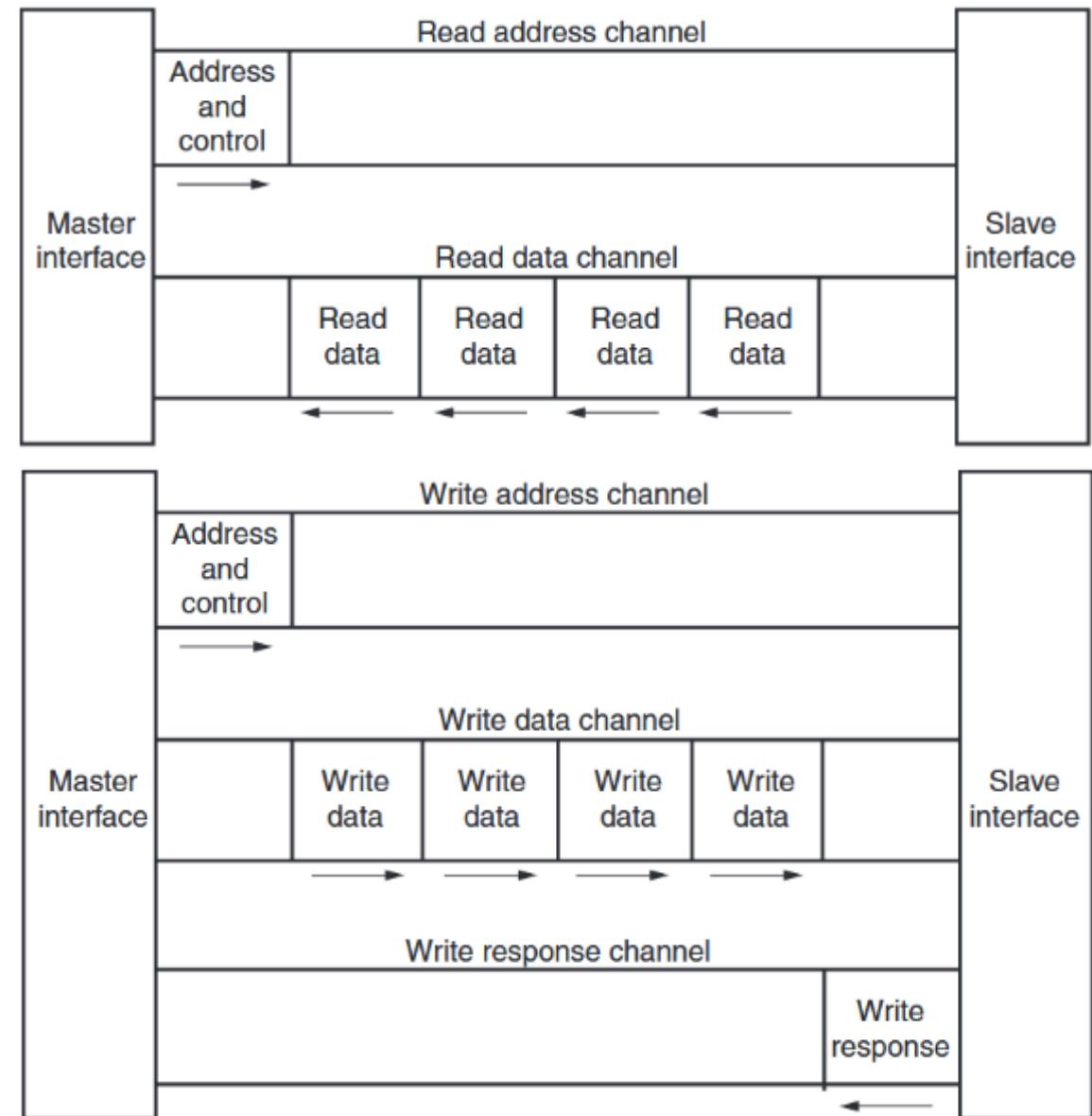
- Although on-chip data communications are rather arbitrary (especially in FPGA-based systems), standard protocols have been developed, which are currently adopted and supported by many processor, FPGA and ASIC vendors.
- The Arm AMBA is an open standard for the connection, management and communication of functional blocks in a **system-on-a-chip (SoC)**, including FPGA-based systems.
- The AMBA AXI4 and AXI-Lite protocols are currently used in many Xilinx tools and IP cores
- AMBA AXI uses READY/VALID handshaking mechanisms

AMBA AXI4 and AXI-Lite Interfaces

- AXI4 and AXI-Lite interfaces consist of five different channels:
 - Read Address Channel
 - Write Address Channel
 - Read Data Channel
 - Write Data Channel
 - Write Response Channel

References and further reading on AXI interface protocols:

- AXI Reference Guide, https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf
- AMBA® AXI™ and ACE™ Protocol Specification, <https://www.arm.com/products/system-ip/amba-specifications>
- AXI4-Streaming to StellarIP Interface, http://www.4dsp.com/pdf/AN001_KC705_FMC104_AXI_FFTcore_tutorial.pdf



SCALABLE DESIGNS AND AUTOMATIC HDL CODE GENERATION

Scalable Design and Automatic HDL Code Generation

- Verilog and VHDL have limited features for scalable and parametric designs (such as `genvar`, `generate`, etc.)
- In this section, we will learn how to write scripts in other languages (C, Java, Python, Matlab, etc.) to generate synthesizable HDL codes
- These methods can be used to generate user defined HDL libraries, Netlists and EDIF files.
- The basic idea is to open a `.v` or `.vhd` file in another language and start writing in it with Verilog or VHDL supported syntax, while using the flexibilities and features of the higher level language.

Scalable Design and Automatic HDL Code Generation (continued)

Example 1: Matlab script for generating Running DFT Verilog code

Scalable Design and Automatic HDL Code Generation (continued)

Example 1 (continued): Output Verilog file

```
1 `timescale 1ns / 1ps
2 ///////////////////////////////////////////////////////////////////
3 // Company:      Signal Processing Center (SPC), Shiraz University
4 // Engineer:     Reza Sameni
5 // Web:          www.sameni.info
6 //
7 // Create Date:  13:38:50 01-Jun-2018
8 // Design Name: Running Discrete Fourier Transform (DFT)
9 // Module Name: RunningDFT8
10 // Project Name: Real Time Spectrum Analysis
11 // Target Devices: Virtex-5
12 // Tool versions:
13 // Description: This module calculates the 8-point running Discrete Fourier
14 //                 Transform (DFT) of a stream of complex valued signal over 8
15 //                 frequency samples.
16 //                 The module has been automatically generated by RunningDFTGenerator.m.
17 //
18 // Dependencies: Xilinx Core Generator Complex Multiplier
19 //
20 // Revision:
21 // Revision 0.01 - File Created
22 // Additional Comments: www.spc.shirazu.ac.ir
23 //
24 ///////////////////////////////////////////////////////////////////
25 module RunningDFT8(clock, ce, reset, logresult, xr, xi, loggedaddress, xxrlogged, xxilogged);
26
27   input clock;
28   input ce;
29   input reset;
30   input logresult;
31   input [15:0] xr;
32   input [15:0] xi;
33   input [2:0] loggedaddress;
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 1 (continued): Output Verilog file continued

```

34 output reg [18:0]xxrlogged;
35 output reg [18:0]xxilogged;
36
37 reg [2:0]oldestpointer;
38 reg [15:0]fifo[7:0];
39 reg [15:0]fifoi[7:0];
40 reg [18:0]deltar;
41 reg [18:0]deltai;
42 reg [18:0]xxrloggedarray[7:0];
43 reg [18:0]xxiloggedarray[7:0];
44
45 // for simulation only
46 integer i;
47 initial begin
48   for(i = 0 ; i < 8 ; i=i+1)begin
49     fifo[i] = 0;
50     fifoi[i] = 0;
51   end
52   for(i = 0 ; i < 8 ; i=i+1)begin
53     xxrloggedarray[i] = 0;
54     xxiloggedarray[i] = 0;
55   end
56   oldestpointer = 0;
57   deltar = 0;
58   deltai = 0;
59 end
60 // end simulation initialization
61
62 always @(posedge clock)if(ce)begin
63   if(reset)begin
64     oldestpointer <= 0;
65     deltar <= 0;
66     deltai <= 0;

```

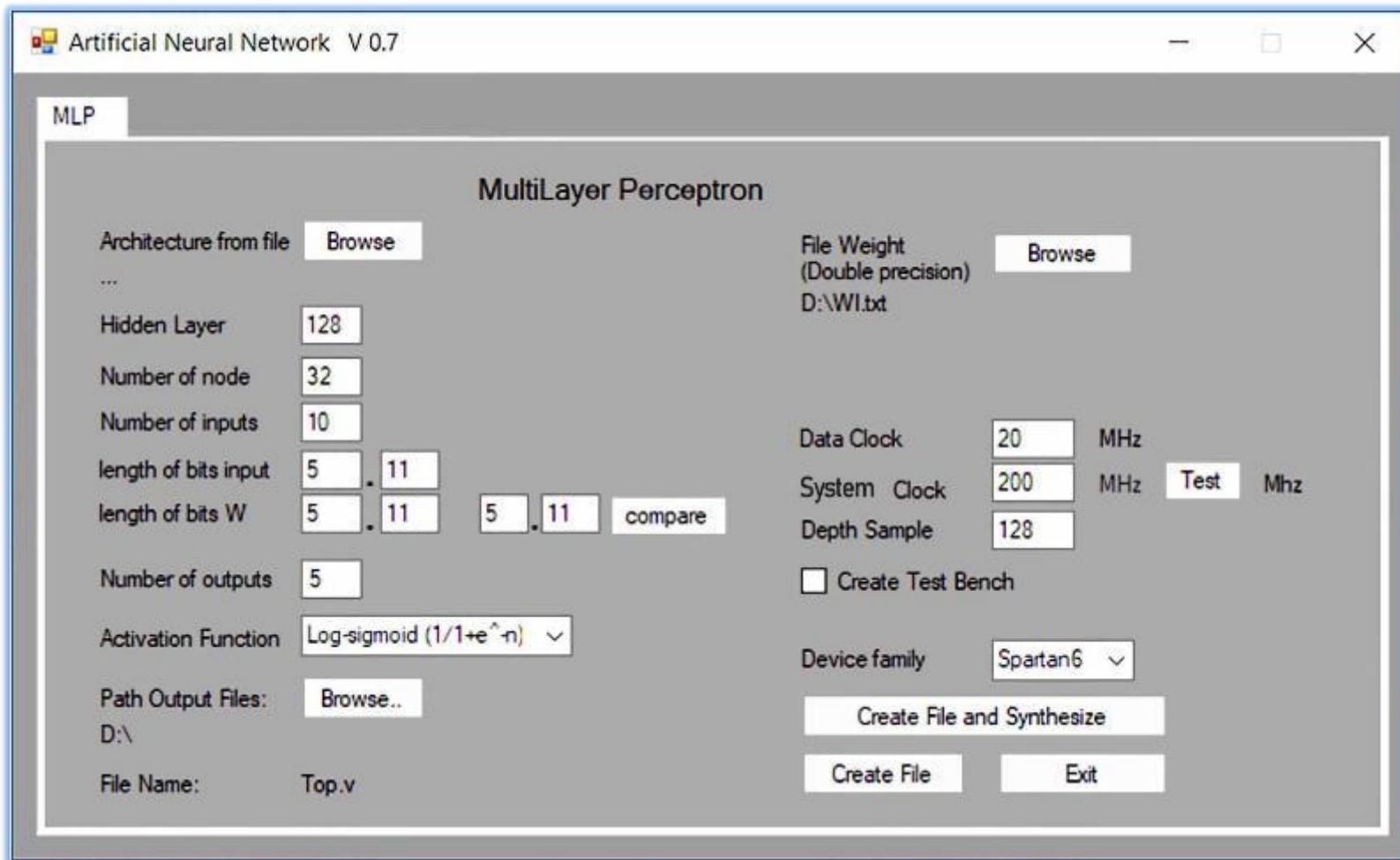
```

67   end
68   else begin
69     deltar <= xr - fifo[oldestpointer];
70     deltai <= xi - fifoi[oldestpointer];
71     fifo[oldestpointer] <= xr;
72     fifoi[oldestpointer] <= xi;
73     oldestpointer <= oldestpointer + 1;
74   end
75 end
76
77 wire [18:0]xxr0;
78 wire [18:0]xxi0;
79 MultiplyAxB multiplier0 (
80   .ar(xxr0 + deltar),
81   .ai(xxi0 + deltai),
82   .br(16'd32767),
83   .bi(16'd0),
84   .clk(clock),
85   .ce(ce),
86   .sclr(reset),
87   .pr(xxr0),
88   .pi(xxi0)) /* synthesis syn_noprune =1 syn_preserve = 1 */;
89
90 wire [18:0]xxr1;
91 wire [18:0]xxi1;
92 MultiplyAxB multiplier1 (
93   .ar(xxr1 + deltar),
94   .ai(xxi1 + deltai),
95   .br(16'd23170),
96   .bi(16'd23170),
97   .clk(clock),
98   .ce(ce),
99   .sclr(reset),

```

Scalable Design and Automatic HDL Code Generation (continued)

Example 2: Generating Multilayer Perceptron Artificial Neural Networks RTL codes in C# (By Pejman Torabi, Shiraz University)



Scalable Design and Automatic HDL Code Generation (continued)

Example 2 (continued): Generated modules

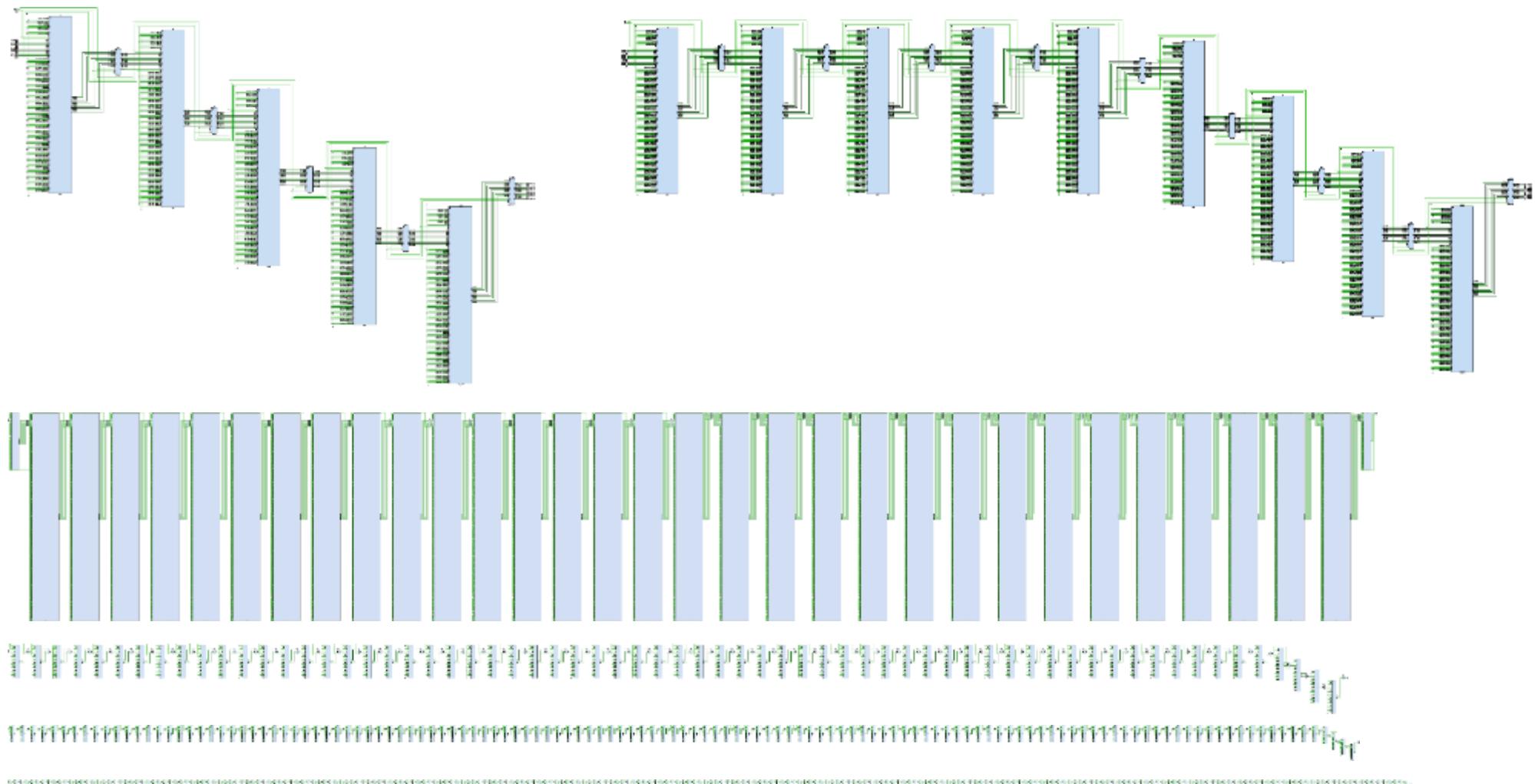
```

module TOP ( In1, In2, In3, In4, In5, In6, In7, In8,
             Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
             clk, en, res);
module Layer1 ( i1 ,i2, i3, i4, i5, i6, i7, i8,
                  w001001,w001002,w001003,w001004,w001005,w001006,w001007,w001008, B001, ...
                  Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
                  clk,en,res);
module Layer ( i1, i2, i3, i4, i5, i6, i7, i8,
                  w001001, w001002, w001003, w001004, w001005, w001006, w001007,w001008, B001, ...
                  Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
                  clk, en, res);
module ActFunc ( In_AF1, In_AF2, In_AF3, In_AF4, In_AF5, In_AF6, In_AF7, In_AF8,
                  Out_AF1, Out_AF2, Out_AF3, Out_AF4, Out_AF5, Out_AF6, Out_AF7, Out_AF8,
                  clk, en, res);
module Function_Interpolation (inputVal, outputVal, clk);
module mult (a, b, z, clk);
module Layer_End ( i1,i2,i3,i4,i5,i6,i7,i8,
                     w001001, w001002, w001003, w001004, w001005, w001006, w001007, w001008, B001,
                     ...
                     Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
                     clk, en, res);
module ActFunc_End (In_AF1, In_AF2, In_AF3, In_AF4, In_AF5, In_AF6, In_AF7, In_AF8,
                     Out_AF1, Out_AF2, Out_AF3, Out_AF4, Out_AF5, Out_AF6, Out_AF7, Out_AF8,
                     clk,en,res);

```

Scalable Design and Automatic HDL Code Generation (continued)

Example 2 (continued): RTL schematic of the generated codes



Scalable Design and Automatic HDL Code Generation (continued)

Example 3: Xilinx HEX file generation in Matlab

```
1 clear;
2 close all;
3
4 N = 340;
5 n = (0:N-1);
6 nbits = 16;
7
8 sine = sin(2*pi*n/N);
9 cosine = cos(2*pi*n/N);
10
11 figure
12 hold on
13 stem(cosine);
14 stem(sine,'r');
15 grid
16
17 fid1 = fopen('sine_init_vals.hex','w');
18 fid2 = fopen('cosine_init_vals.hex','w');
19 for i = 1:N
20     fprintf(fid1,'%s\n',hex(fi(sine(i),1,nbits,nbits-1)));
21     fprintf(fid2,'%s\n',hex(fi(cosine(i),1,nbits,nbits-1)));
22 end
23 fclose(fid1);
24 fclose(fid2);
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 3 (continued): Output HEX file

1 0000	21 2e3d	41 563c	61 7295	81 7f74
2 025e	22 3070	42 57f7	62 739e	82 7fa7
3 04bb	23 329e	43 59ab	63 749c	83 7fce
4 0718	24 34c8	44 5b58	64 7591	84 7fea
5 0974	25 36ed	45 5cfb	65 767b	85 7ffa
6 0bcf	26 390e	46 5e98	66 775b	86 7fff
7 0e2a	27 3b29	47 602c	67 7831	87 7ffa
8 1083	28 3d40	48 61b7	68 78fc	88 7fea
9 12db	29 3f51	49 633a	69 79bc	89 7fce
10 1531	30 415c	50 64b4	70 7a72	90 7fa7
11 1785	31 4362	51 6625	71 7b1d	91 7f74
12 19d7	32 4562	52 678e	72 7bbd	92 7f37
13 1c27	33 475c	53 68ed	73 7c53	93 7eee
14 1e75	34 494f	54 6a43	74 7cde	94 7e9b
15 20bf	35 4b3d	55 6b90	75 7d5d	95 7e3c
16 2307	36 4d23	56 6cd4	76 7dd2	96 7dd2
17 254c	37 4f03	57 6e0e	77 7e3c	97 7d5d
18 278e	38 50dc	58 6f3e	78 7e9b	98 7cde
19 29cc	39 52ae	59 7065	79 7eee	99 7c53
20 2c07	40 5478	60 7182	80 7f37	100 7bbd

...

Scalable Design and Automatic HDL Code Generation (continued)

Example 4: Xilinx coefficient file generation in C

```
1 #include<stdio.h>
2 #include<math.h>
3 #define PI (4*atan(1))
4 void main()
5 {
6     int N = 400;
7     int Radix = 16;
8     int Coefficient_Width = 16;
9     int MaxNum = (1<<(Coefficient_Width-1)) - 1;
10    int AllOnes = (1<<Coefficient_Width) - 1;
11    unsigned *CoefData;
12    CoefData = new unsigned[N];
13    FILE* fil;
14    fil = fopen("Sine400.coe", "w");
15    fprintf(fil,";\n; XILINX CORE Generator(tm) Look-Up-Table (.COE) File\n";
16    Generated by Reza Sameni\n;\n; Generated on: 7-May-2012 12:52:00\n;\n");
17    fprintf(fil,"memory_initialization_radix = %d;\n", Radix);
18    ///fprintf(fil,"Coefficient_Width = %d;\n", Coefficient_Width);
19    fprintf(fil,"memory_initialization_vector = ");
20    for(int i = 0 ; i < N ; i++)
21    {
22        CoefData[i] = int(MaxNum*sin(i*2.*PI/N)) & AllOnes;
23        if(i < N-1)
24            fprintf(fil,"%x,", CoefData[i]);
25        else
26            fprintf(fil,"%x;\n", CoefData[i]);
27    }
28    fclose(fil);
29 }
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 4 (continued): Output COE file

```
1 ;  
2 ; XILINX CORE Generator(tm) Look-Up-Table (.COE) File  
3 ; Generated by Reza Sameni  
4 ;  
5 ; Generated on: 7-May-2012 12:52:00  
6 ;  
7 memory_initialization_radix = 16;  
8 memory_initialization_vector =  
0,202,405,607,809,a0a,c0b,e0b,100a,1208,1405,1601,17fb,19f4,1beb,1de1,1fd4,21c6,23b5,25a2,278d,2975  
,2b5b,2d3e,2f1e,30fb,32d5,34ac,367f,384f,3a1b,3be4,3da9,3f6a,4127,42e0,4495,4645,47f1,4999,4b3b,4cd  
9,4e73,5007,5196,5196,5320,54a5,5624,579e,5912,5a81,5bea,5d4e,5eab,6002,6154,629f,63e4,6523,665b,678d,68  
b8,69dc,6afa,6c12,6d22,6e2b,6f2e,7029,711e,720b,72f1,73d0,74a7,7578,7640,7701,77bb,786d,7918,79bb,7  
a56,7ae9,7b75,7bf9,7c75,7ce9,7d56,7dba,7e17,7e6b,7eb8,7efc,7f39,7f6d,7f99,7fbe,7fda,7fee,7ffa,7fff,  
7ffa,7fee,7fda,7fbe,7f99,7f6d,7f39,7efc,7eb8,7e6b,7e17,7dba,7d56,7ce9,7c75,7bf9,7b75,7ae9,7a56,79bb  
,7918,786d,77bb,7701,7640,7578,74a7,73d0,72f1,720b,711e,7029,6f2e,6e2b,6d22,6c12,6afa,69dc,68b8,678  
d,665b,6523,63e4,629f,6154,6002,5eab,5d4e,5bea,5a81,5912,579e,5624,54a5,5320,5196,5007,4e73,4cd9,4b  
3b,4999,47f1,4645,4495,42e0,4127,3f6a,3da9,3be4,3a1b,384f,367f,34ac,32d5,30fb,2f1e,2d3e,2b5b,2975,2  
78d,25a2,23b5,21c6,1fd4,1de1,1beb,19f4,17fb,1601,1405,1208,100a,e0b,c0b,a0a,809,607,405,202,0,fdf,  
fbfb,f9f9,f7f7,f5f6,f3f5,f1f5,eff6,edf8,ebfb,e9ff,e805,e60c,e415,e21f,e02c,de3a,dc4b,da5e,d873,d68b  
,d4a5,d2c2,d0e2,cf05,cd2b,cb54,c981,c7b1,c5e5,c41c,c257,c096,bed9,bd20,bb6b,b9bb,b80f,b667,b4c5,b32  
7,b18d,aff9,ae6a,ace0,ab5b,a9dc,a862,a6ee,a57f,a416,a2b2,a155,9ffe,9eac,9d61,9c1c,9add,99a5,9873,97  
48,9624,9506,93ee,92de,91d5,90d2,8fd7,8ee2,8df5,8d0f,8c30,8b59,8a88,89c0,88ff,8845,8793,86e8,8645,8  
5aa,8517,848b,8407,838b,8317,82aa,8246,81e9,8195,8148,8104,80c7,8093,8067,8042,8026,8012,8006,8001,  
8006,8012,8026,8042,8067,8093,80c7,8104,8148,8195,81e9,8246,82aa,8317,838b,8407,848b,8517,85aa,8645  
,86e8,8793,8845,88ff,89c0,8a88,8b59,8c30,8d0f,8df5,8ee2,8fd7,90d2,91d5,92de,93ee,9506,9624,9748,987  
3,99a5,9add,9c1c,9d61,9eac,9ffe,a155,a2b2,a416,a57f,a6ee,a862,a9dc,ab5b,ace0,ae6a,aff9,b18d,b327,b4  
c5,b667,b80f,b9bb,bb6b,bed9,c096,c257,c41c,c5e5,c7b1,c981,cb54,cd2b,cf05,d0e2,d2c2,d4a5,d68b,d  
873,da5e,dc4b,de3a,e02c,e21f,e415,e60c,e9ff,ebfb,edf8,eff6,f1f5,f3f5,f5f6,f7f7,f9f9,fbfb,fdf;
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 5: Automatic listing generation for LaTeX reports. Project reports (specifically in LaTeX) can be automatically updated with the latest version of the source codes

```

1 clear;
2 close all;
3 clc;
4 d = dir('*.v');
5 fid = fopen('VerilogSourceCodeListing.tex','w');
6 for i = 1 : length(d)
7     I = strfind(d(i).name,'_');
8     lbl = d(i).name;
9     if(~isempty(I))
10        dd = d(i).name;
11        for j = 1: length(I)
12            %dd = [dd(1:(I(j)-1 + (j-1))) '\' dd( (I(j) + (j-1)):end)];
13            dd = [dd(1:(I(j)-1 - (j-1))) '\' dd( (I(j+1) - (j-1)):end)];
14        end
15        lbl = dd;
16    end
17    fprintf(fid, '\\lstinputlisting[basicstyle = \\footnotesize, language = Verilog, label = %s, caption
= %s]{SourceCodes/Verilog/%s}\n',d(i).name, d(i).name, d(i).name);
18 end
19 fclose(fid);
20
21 d = dir('ipcore_dir\\*.xco');
22 fid = fopen('XilinxCoreGeneratorCodeListing.tex','w');
23 for i = 1 : length(d)
24     I = strfind(d(i).name,'_');
25     lbl = d(i).name;
26     if(~isempty(I))
27        dd = d(i).name;
28        for j = 1: length(I)
29            %dd = [dd(1:(I(j)-1 + (j-1))) '\' dd( (I(j) + (j-1)):end)];
30            dd = [dd(1:(I(j)-1 - (j-1))) dd( (I(j) + 1 - (j-1)):end)];
31        end
32        lbl = dd;
33    end
34    fprintf(fid, '\\lstinputlisting[basicstyle = \\footnotesize, language = Verilog, label = %s, caption
= %s]{SourceCodes/Verilog/ipcore_dir/%s}\n',lbl, lbl, d(i).name);
35 end
36 fclose(fid);

```

Scalable Design and Automatic HDL Code Generation (continued)

Example 5: Output LaTeX listing

```
1 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
AnalyticSignal.v, caption = AnalyticSignal.v]{SourceCodes/Verilog/
AnalyticSignal.v}
2 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
CICFilter.v, caption = CICFilter.v]{SourceCodes/Verilog/CICFilter.v}
3 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
CICFilter2Channel.v, caption = CICFilter2Channel.v]{SourceCodes/Verilog/
CICFilter2Channel.v}
4 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
CICFilter6Channel.v, caption = CICFilter6Channel.v]{SourceCodes/Verilog/
CICFilter6Channel.v}
5 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
ClockDecimator.v, caption = ClockDecimator.v]{SourceCodes/Verilog/
ClockDecimator.v}
6 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DFTCore.v, caption = DFTCore.v]{SourceCodes/Verilog/DFTCore.v}
7 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DFTCoreTop.v, caption = DFTCoreTop.v]{SourceCodes/Verilog/DFTCoreTop.v}
8 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DFTCoreTopTB.v, caption = DFTCoreTopTB.v]{SourceCodes/Verilog/
DFTCoreTopTB.v}
9 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DIFMArbitrage.v, caption = DIFMArbitrage.v]{SourceCodes/Verilog/
DIFMArbitrage.v}
10 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DIFMCorrelator.v, caption = DIFMCorrelator.v]{SourceCodes/Verilog/
DIFMCorrelator.v}
11 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
DIFMCorrelator2.v, caption = DIFMCorrelator2.v]{SourceCodes/Verilog/
DIFMCorrelator2.v}
```