



# CIC Training Manual

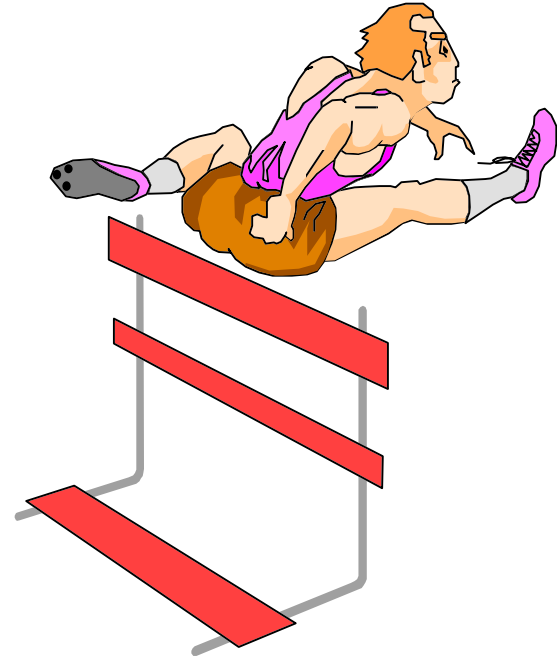
## FPGA Synthesis Training Course



July 1998

# Course Outline

- ◆ Introduction to PLD
- ◆ Altera Device Families
- ◆ Xilinx Device Families
- ◆ HDL Design Flow & Tools



## Why this course?

### For ASIC Users

**Front-End** Cell-Based IC Design Kit (2 days)  
Verilog / VHDL (2 days)  
Logic Synthesis (2 days)

**Back-End** Cell-Based IC Physical Design (3 days)

### For pure FPGA Users

Verilog / VHDL (2 days)  
FPGA Synthesis (1 day)

Altera(WS) / Xilinx(WS) (1.5 days)

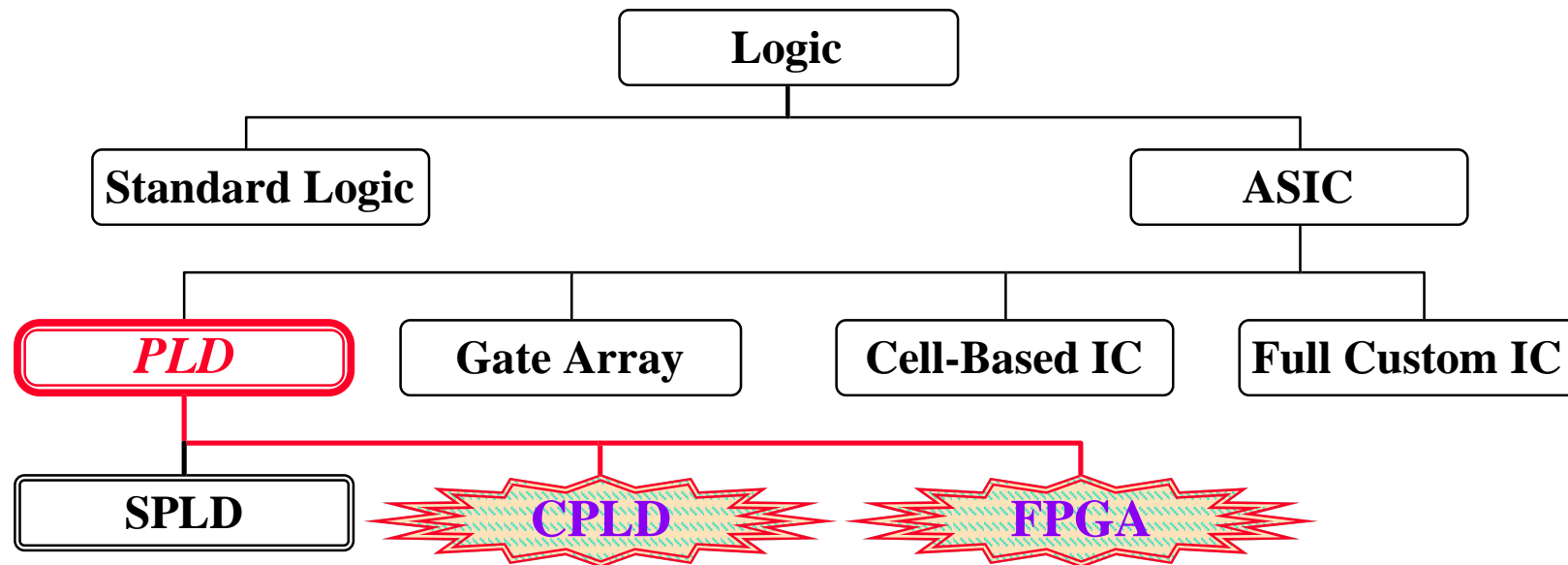
## ◆ HDL Coding Hints

- Generic Coding Techniques & Considerations
- Altera-Specific Issues
- Xilinx-Specific Issues

## ◆ Summary & Getting Help

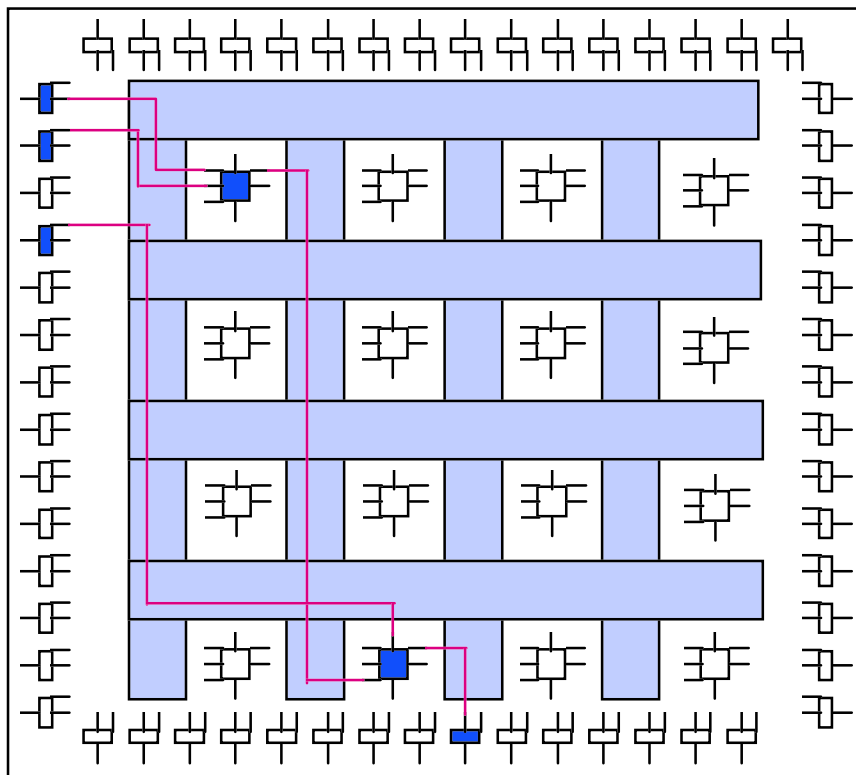


# Introduction to PLD



- PLD** : Programmable Logic Device
- SPLD** : Small/Simple Programmable Logic Device
- CPLD** : Complex Programmable Logic Device
- FPGA** : Field Programmable Gate Array

# Main Features

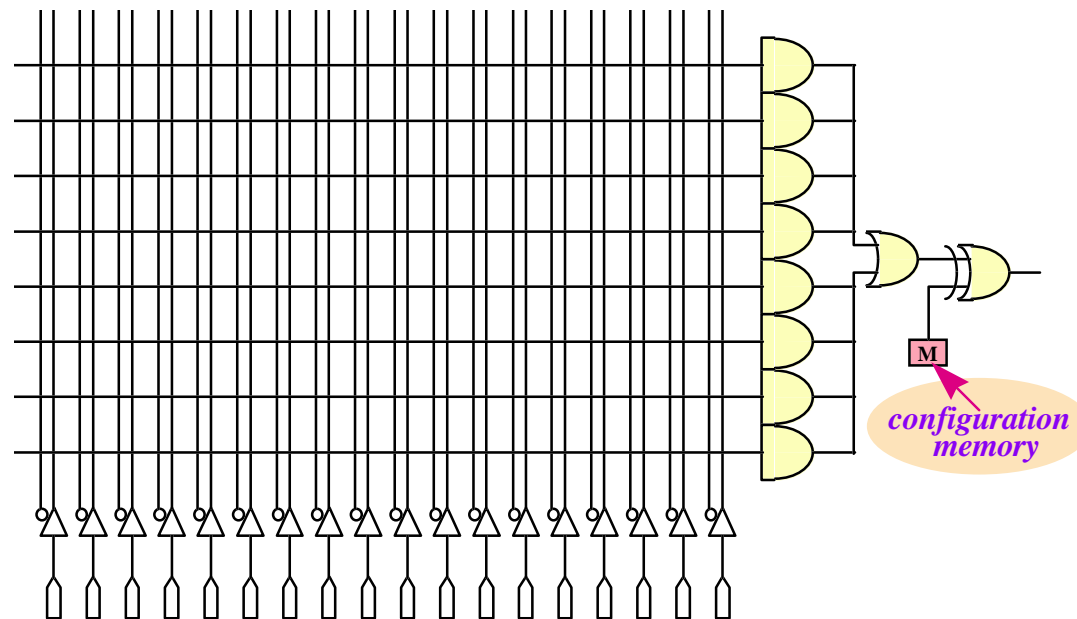


- ◆ Field-programmable
- ◆ Reprogrammable
- ◆ In-circuit design verification
- ◆ Rapid prototyping
- ◆ Fast time-to-market
- ◆ No IC-test & NRE cost
- ◆ H/W emulation instead of S/W simulation
- ◆ Good software
- ◆ ...

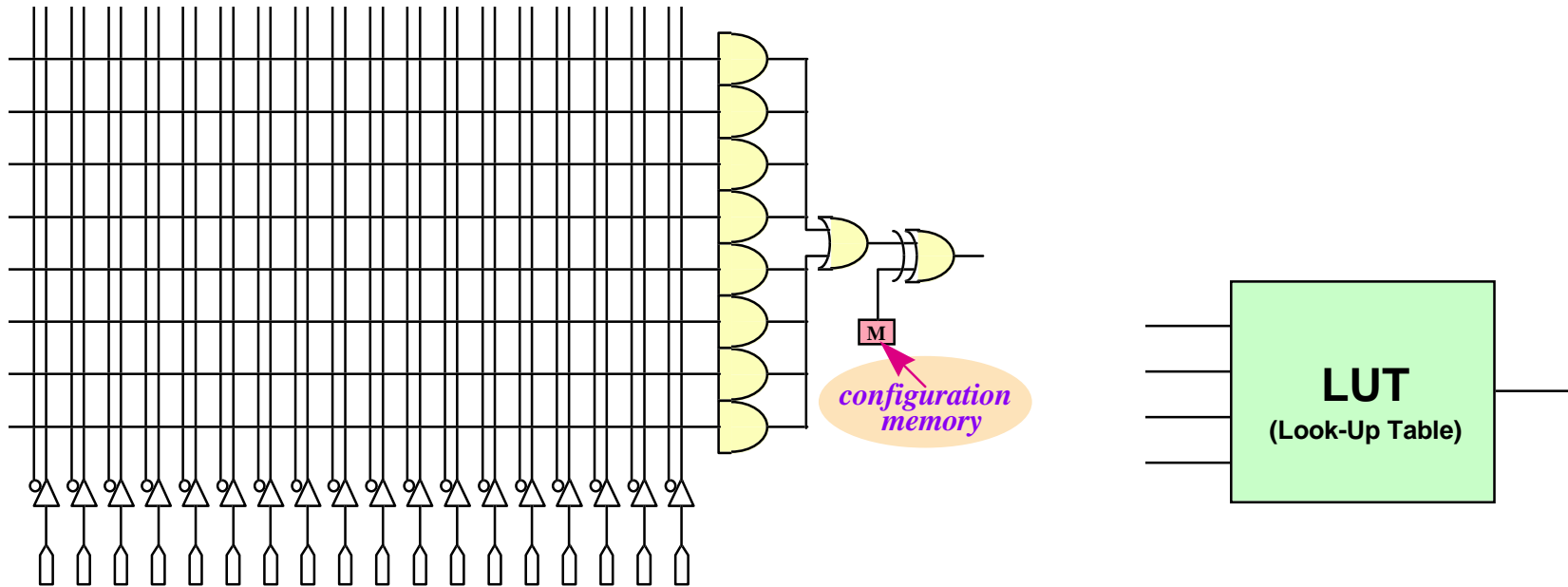
# Programmability

## ◆ Why programmable? Why reprogrammable?

- Logic is implemented by programming the “configuration memory”
- Various configuration memory technologies
  - One-Time Programmable: anti-fuse, EPROM
  - Reprogrammable: EPROM, EEPROM, Flash & SRAM



# Programmable Combinational Logic



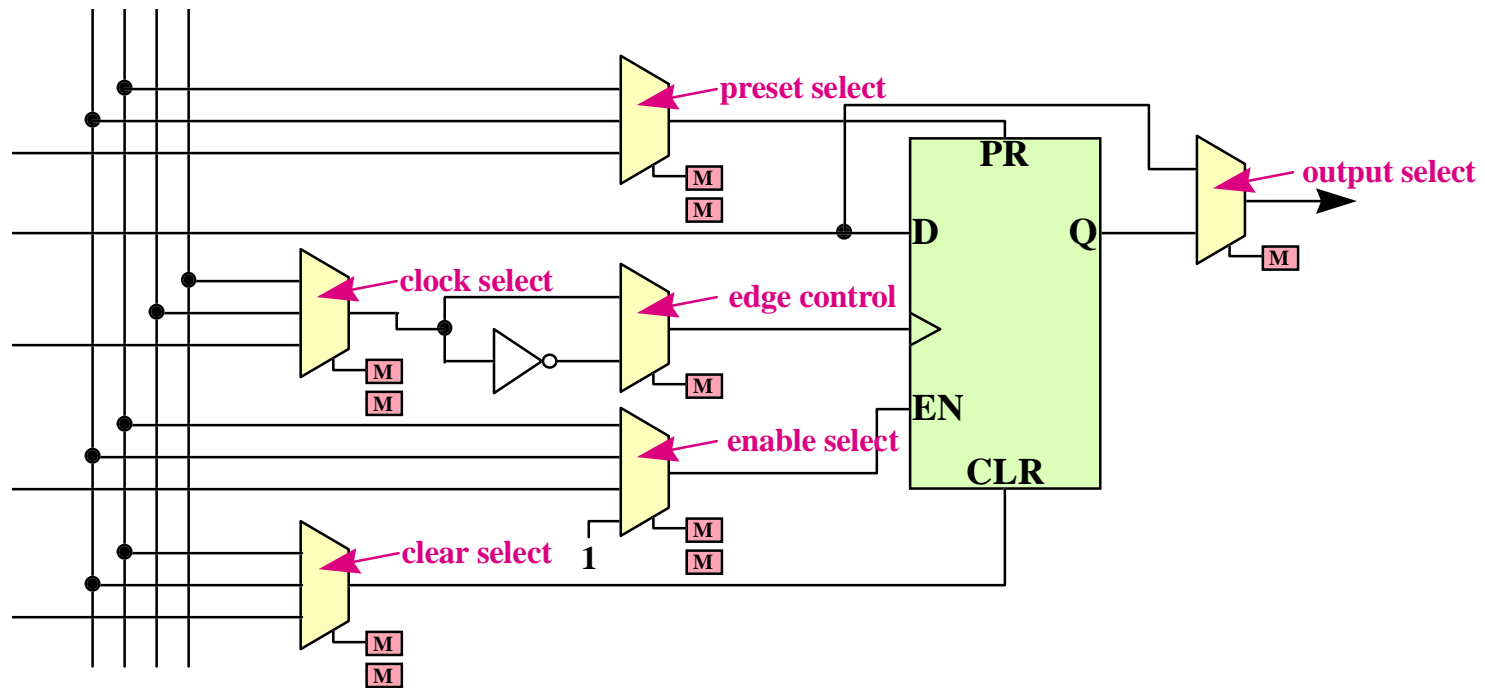
## Product Term-based Building Block

- \* 2-level logic
- \* High fan-in

## Look-up Table-based Building Block

- \* 4 to 5 inputs, fine grain architecture
- \* ROM-like

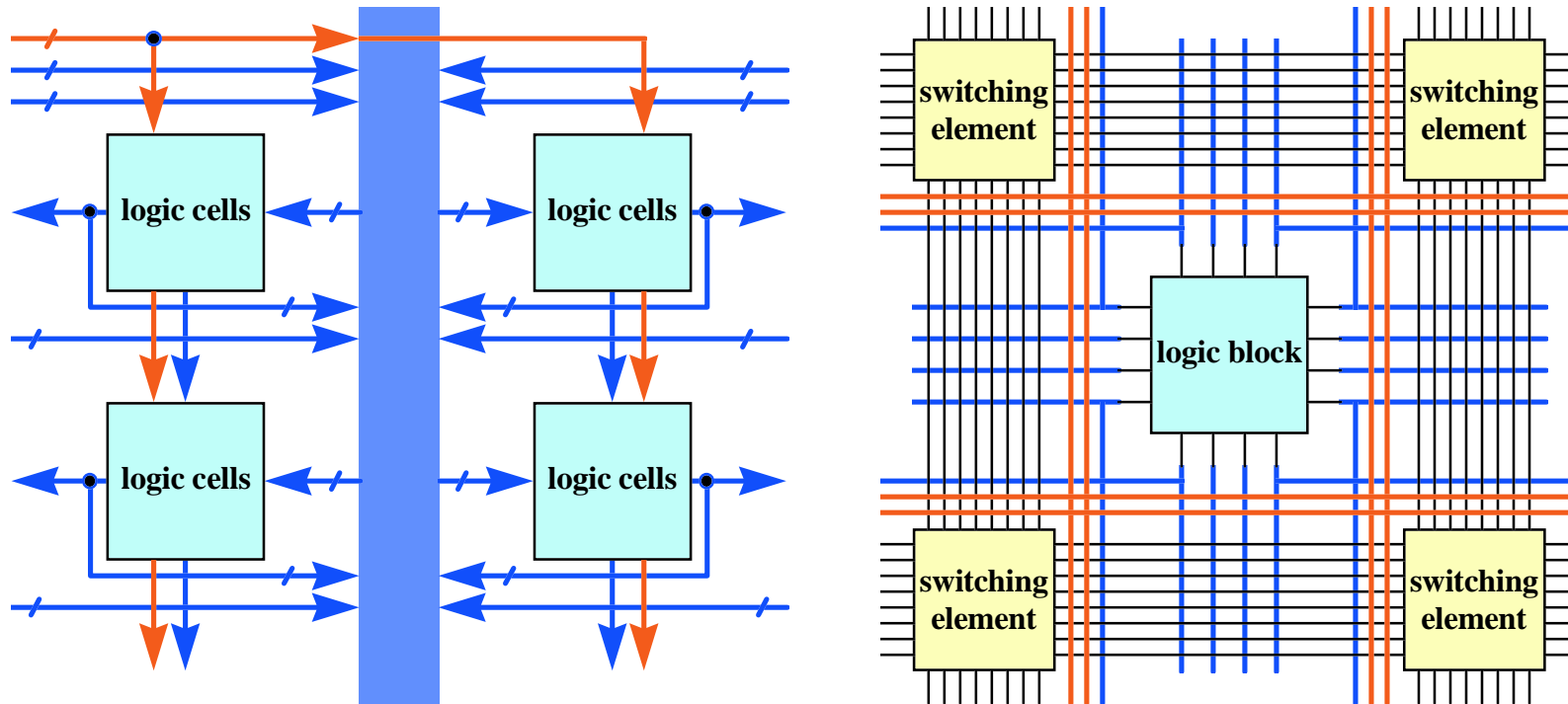
# Programmable Register



\* Typical register controls: clock, enable, preset/clear, ...

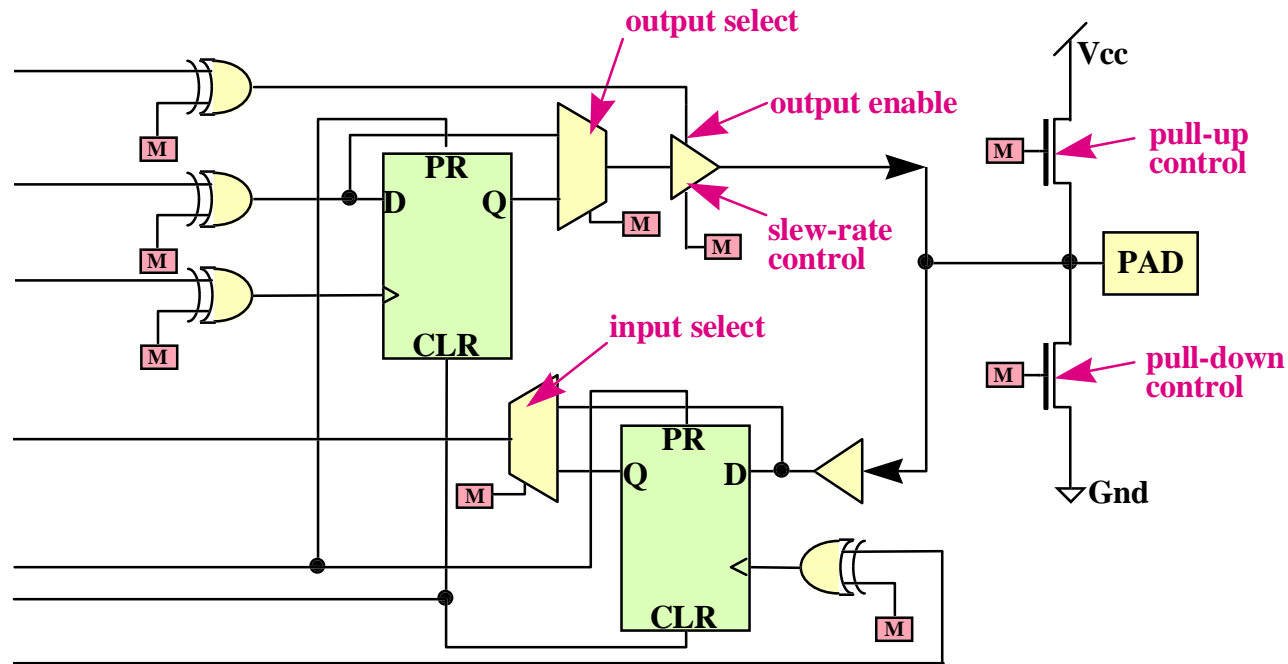


# Programmable Interconnect



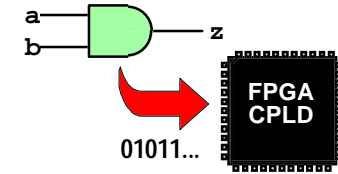
Typical routing resources: switching elements, local/global lines, clock buffers...

# Programmable I/O



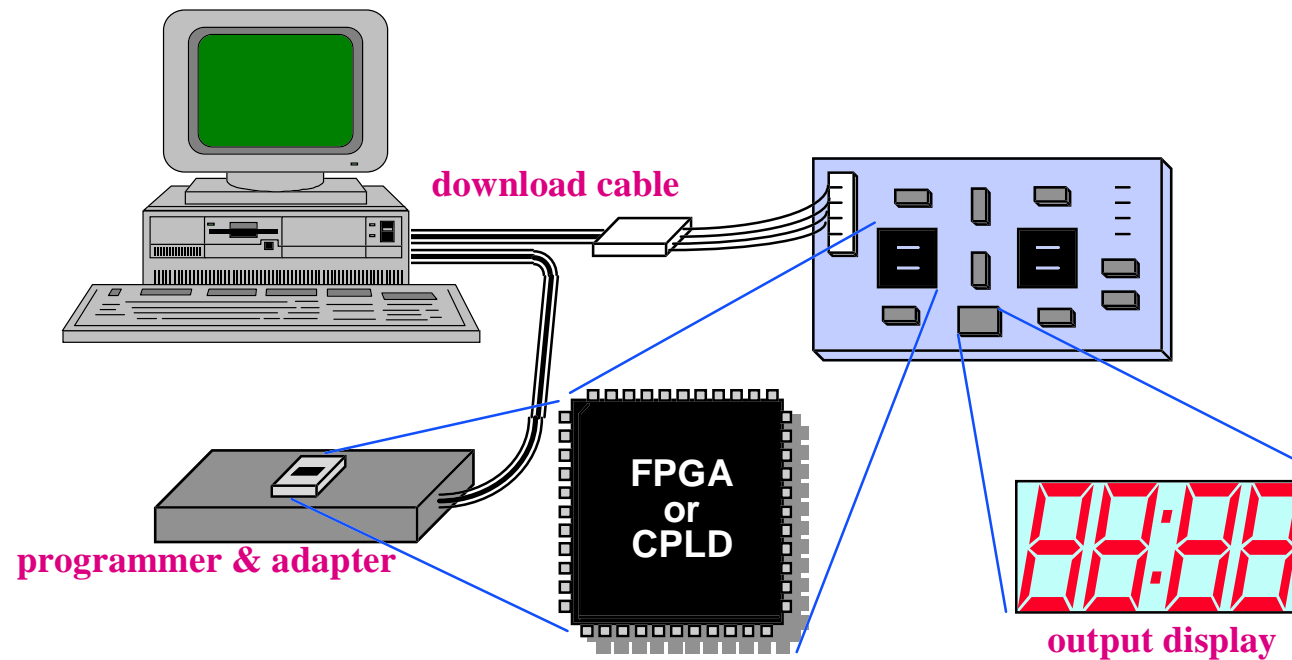
Typical I/O controls: direction, I/O registers, 3-state, slew rate, ...

# Field-Programmability



## ◆ Why filed-programmable?

- You can verify your designs at any time by configuring the FPGA/CPLD devices on board via the download cable or hardware programmer

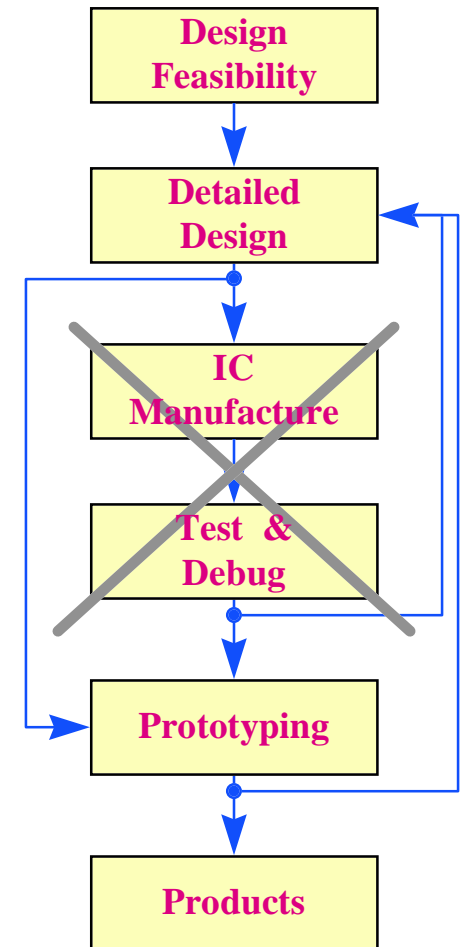
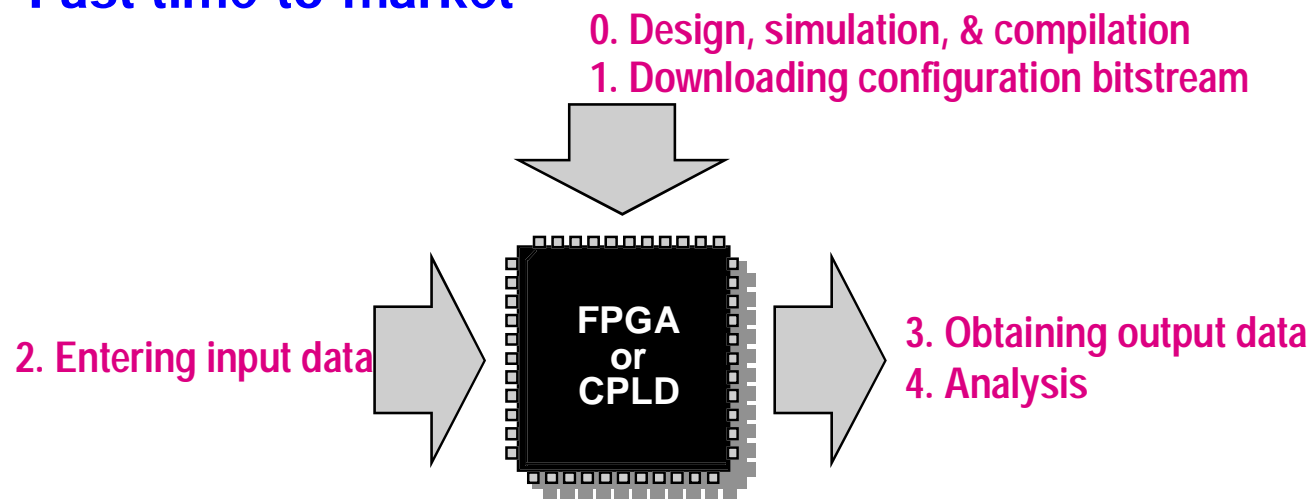


# Rapid Prototyping

## ◆ Reduce system prototyping time :

- You can see the “real” things
  - In-circuit design verification
- Quick delivery instead of IC manufacture
- No test development, no re-spin potential (i.e. no NRE cost)
- *Satisfied for educational purposes*

## ◆ Fast time-to-market



# Software Environment

## ◆ Various design entries and interfaces

- HDL: Verilog, VHDL, ABEL, ...
- Graphic: Viewlogic, OrCAD, Cadence, ...

## ◆ Primitives & macrofunctions provided

- Primitive gates, arithmetic modules, flip-flops, counters, I/O elements, ...

## ◆ Constraint-driven compilation/implementation

- Logic fitting, partition, placement & routing (P&R)

## ◆ Simulation netlist generation

- Functional simulation & timing simulation netlist extraction

## ◆ Programmer/download program

# FPGA/CPLD Benefits

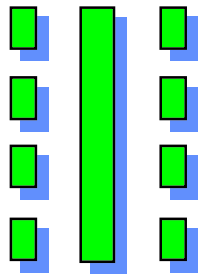
	Full-Custom ICs	Cell-Based ICs	Gate Arrays	High-Density PLDs
Speed	√ √	√	√	√
Integration Density	√ √	√	√	√
High-Volume device cost	√ √	√ √	√	√
Low-volume device cost			√	√ √
Time to Market			√	√ √
Risk Reduction				√ √
Future Modification				√ √
Development Tool	√	√	√	√ √
Educational Purpose				√ √

√ Good  
 √√ Excellent

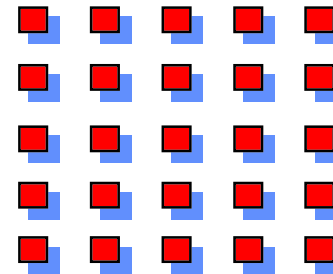
# CPLD vs. FPGA

*This training course will focus on FPGA devices.*

## CPLD



## FPGA



Architecture

PAL-like

Gate Array-like

SOP-Based Logic Cells  
Combinational-Intensive

LUT-Based Logic Cells  
Register-Intensive

Density

Low-to-medium  
Many 22V10s

Medium-to-high  
1K to 250K logic gates

Performance

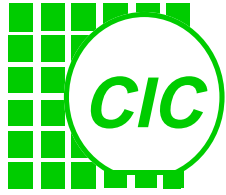
Predictable timing  
Up to 200 MHz today

Application dependent  
Up to 100MHz today

Interconnect

“Crossbar”

Incremental



# Altera & CIC



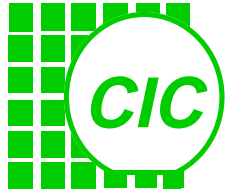
## ◆ Altera

- One of the world leaders in high-performance & high-density PLDs & associated CAE tools
- Supports university program in Taiwan via CIC

## ◆ From CIC, you can apply:

- Altera software - *it's free for educational purpose!*
  - ☞ PC : MAX+PLUS II (full design environment)
  - ☞ WS : MAX+PLUS II (full design environment since V8.1)  
Synopsys interface (Cadence & Viewlogic interfaces are optional)
- Altera hardware - *it's awarded to good software applicants!*
  - University Program Design Laboratory Package (since 9709):
    - UP1 Education Board
    - ByteBlaster download cable
    - Student Edition Software
- Of course, CIC is responsible for technical supports
- WWW: <http://www.cic.edu.tw/html/software/Altera>





# Xilinx & CIC



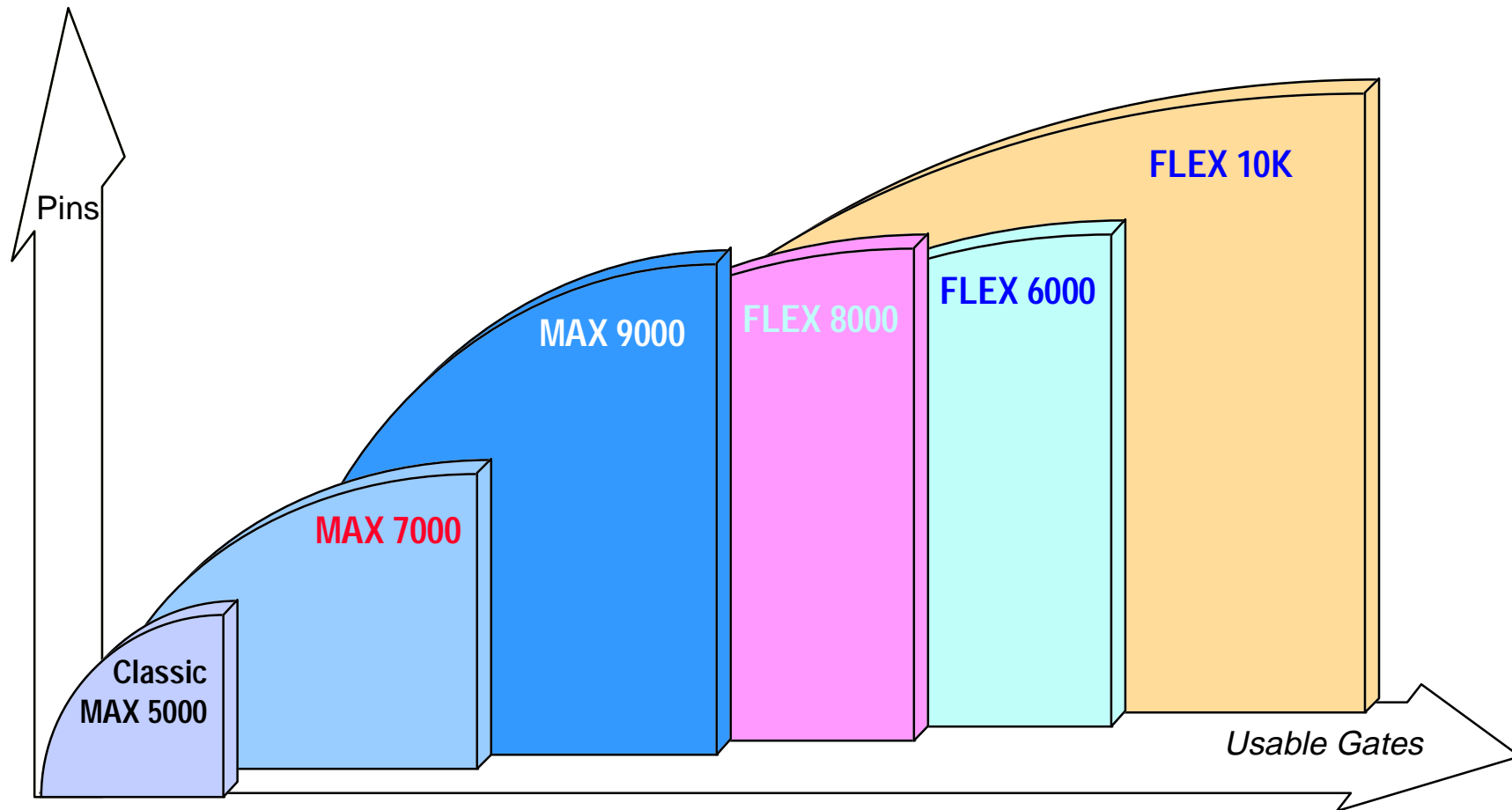
## ◆ Xilinx

- One of the world leaders in high-performance & high-density PLDs & associated CAE tools
- Supports university program in Taiwan via CIC

## ◆ From CIC, you can apply or purchase:

- Xilinx software - *it's free for educational purpose!*
  - ☞ PC : Xilinx Foundation Express (including XACTstep M1, FPGA Express)
  - ☞ WS : Xilinx Alliance Series (including XACTstep M1)  
Synopsys interface (XSI), Cadence Verilog-XL interface
- Xilinx hardware:
  - XChecker download cable
  - FPGA Demo Board
- CIC is responsible for technical supports for universities
- WWW: <http://www.cic.edu.tw/html/software/Xilinx>

# Altera Device Families



# Altera Device Families

## ◆ Altera offers 7 device families

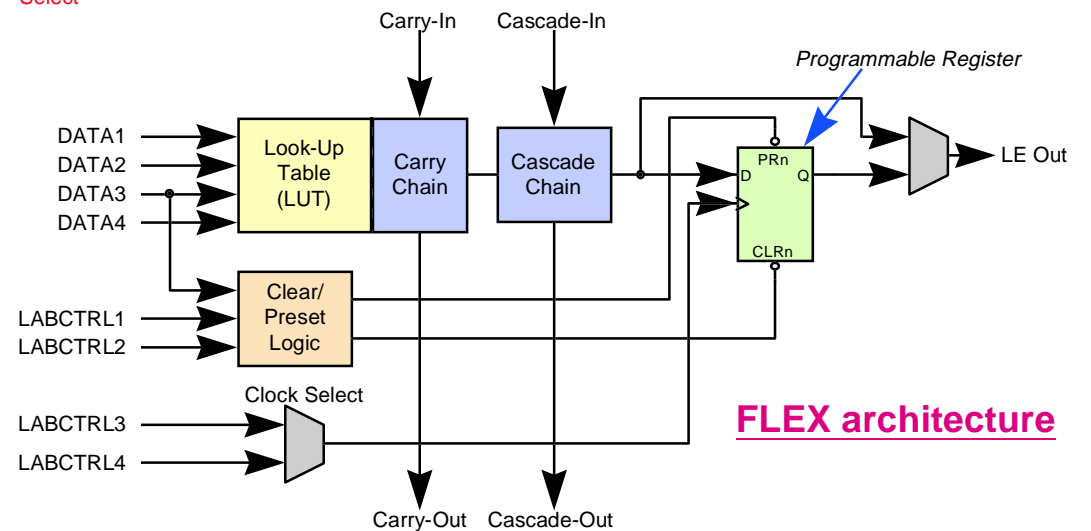
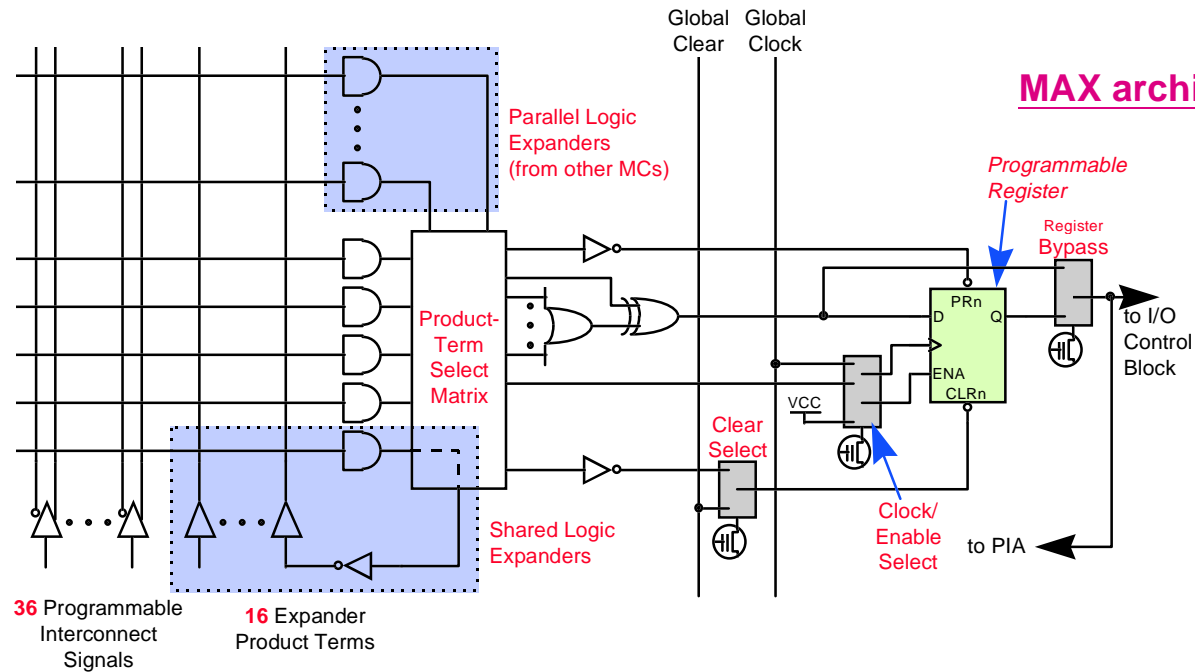
Device Family	Reconfigurable Element	Logic Cell Structure	Usable/Typical Gates	Family Members
Classic	EPROM	SOP	200 ~ 900	EP610, 910, 1810
MAX 5000	EPROM	SOP	800 ~ 3,200	EPM5032, 064, 128, 130, 192
MAX 7000/E/S <sup>(1)</sup>	EEPROM	SOP	600 ~ 5,000	EPM7032/V/S, 064/S, 096/S, EPM7128E/S, 160E/S, 192E/S, 256E/S
FLEX 6000 <sup>(1)</sup>	SRAM	LUT	10,000 ~ 24,000	EPF6010, 016/A, 024A
FLEX 8000A	SRAM	LUT	2,500 ~ 16,000	EPF8282A, 452A, 636A, 820A, 1188A, 1500A
MAX 9000/A <sup>(1)</sup>	EEPROM	SOP	6,000 ~ 12,000	EPM9320/A, 400/A, 480/A, 560/A
FLEX 10K/A <sup>(1)</sup>	SRAM	LUT	10,000 ~ 100,000	EPF10K10/A, 20/A, 30/A, 40/A, 50/V/A, EPF10K70/V/A, 100/A, 130/V/A, 250A

**Note:**

(1) Not all devices are currently available.

(2) Altera plans to ship new Michelangelo family in the near future.

# MAX & FLEX Architectures - (1)



# MAX & FLEX Architectures - (2)

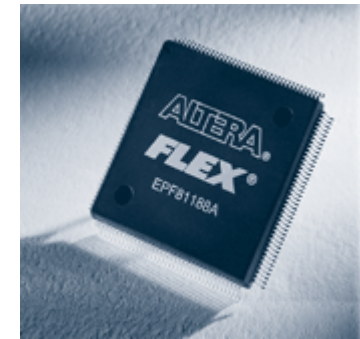
## ◆ Choose the appropriate architecture

- Different PLD architectures provide different performance & capacity results for same application

*This training course will focus on FLEX devices.*

Feature	MAX Architecture	FLEX Architecture
Basic Building Block	Course Grain	Fine Grain
Logic Cell Structure	SOP	LUT
Technology	EEPROM	SRAM
Optimization	Combinational-Intensive Logic e.g. Large Decoders, State Machines, ...	Register-Intensive, Arithmetic Functions e.g. Adders, Comparators, Counters, ...

# FLEX 8000A Family



## ◆ Today's FLEX 8000A family members

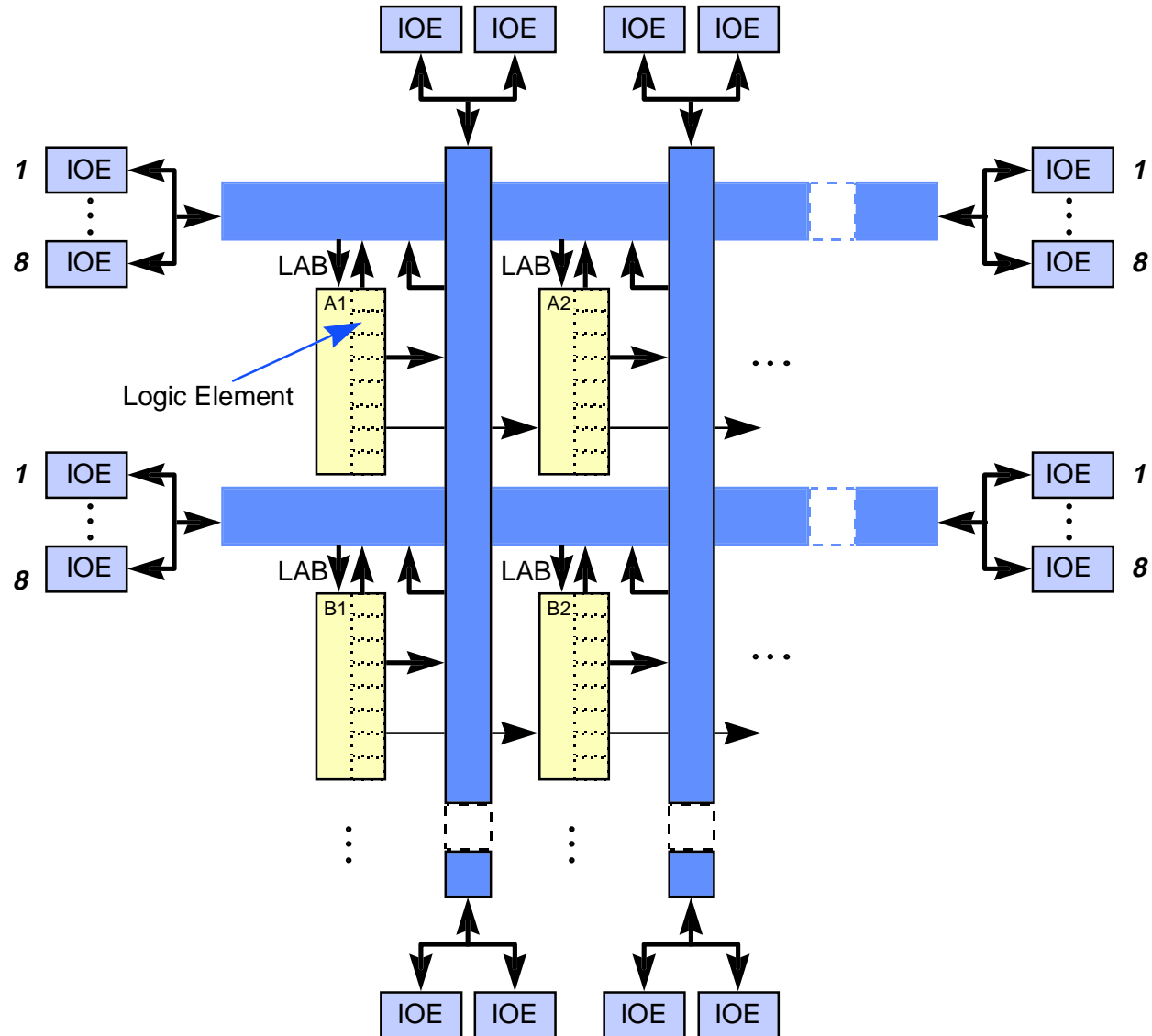
Device	Gates	LEs	FFs	Speed Grade	Package Options	I/O Pins
EPF8282A	2,500	208	282	-2,-3,-4	PLCC84, TQFP100	68,78
EPF8282AV	2,500	208	282	-4	TQFP100	68,78
EPF8452A	4,000	336	452	-2,-3,-4	PLCC84, TQFP100, PQFP160, PGA160	68,120
EPF8636A	6,000	504	636	-2,-3,-4	PLCC84, PQFP160/208, PGA192	68,118,136
EPF8820A	8,000	672	820	-2,-3,-4	TQFP144, PQFP160/208, PGA192, BGA225	120,152
EPF81188A	12,000	1,008	1,188	-2,-3,-4	PQFP208/240, PGA232	148,184
EPF81500A	16,000	1,296	1,500	-2,-3,-4	PQFP240, PGA280, RQFP304	181,208

# FLEX 8000A Features

## ◆ FLEX 8000A main features...

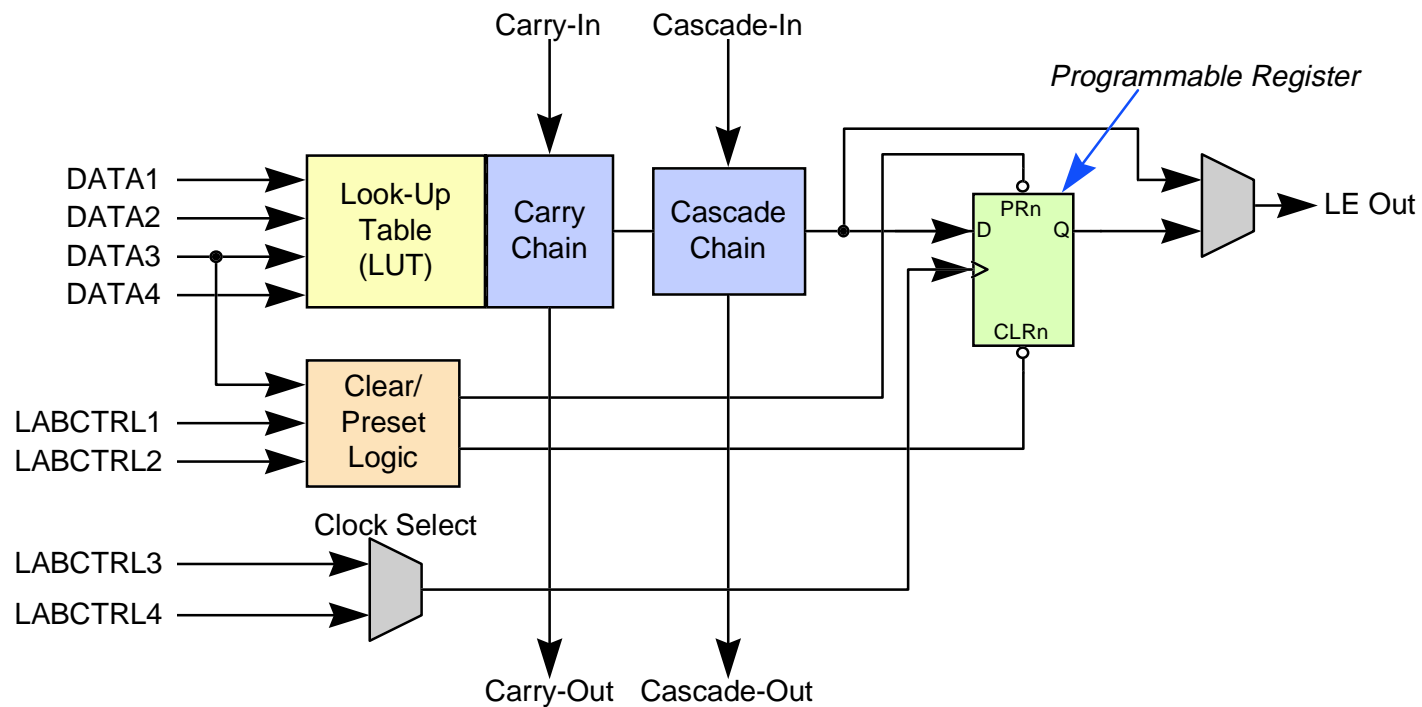
- SRAM-based devices based on Altera's FLEX architecture
- 282 ~ 1,500 registers
- 2,500 ~ 16,000 usable gates
- Programmable flip-flops with individual clear & preset controls
- Dedicated carry chain & cascade chain
- FastTrack continuous routing structure
- Programmable output slew-rate control
- Supports in-circuit reconfiguration (ICR)
- JTAG boundary-scan test circuitry
- PCI-compliant -2 speed grade
- 3.3-V or 5-V operation
  - Full 3.3-V EPF8282AV
  - 3.3-V or 5-V I/O for EPF8636A and larger devices

# FLEX 8000A Architecture

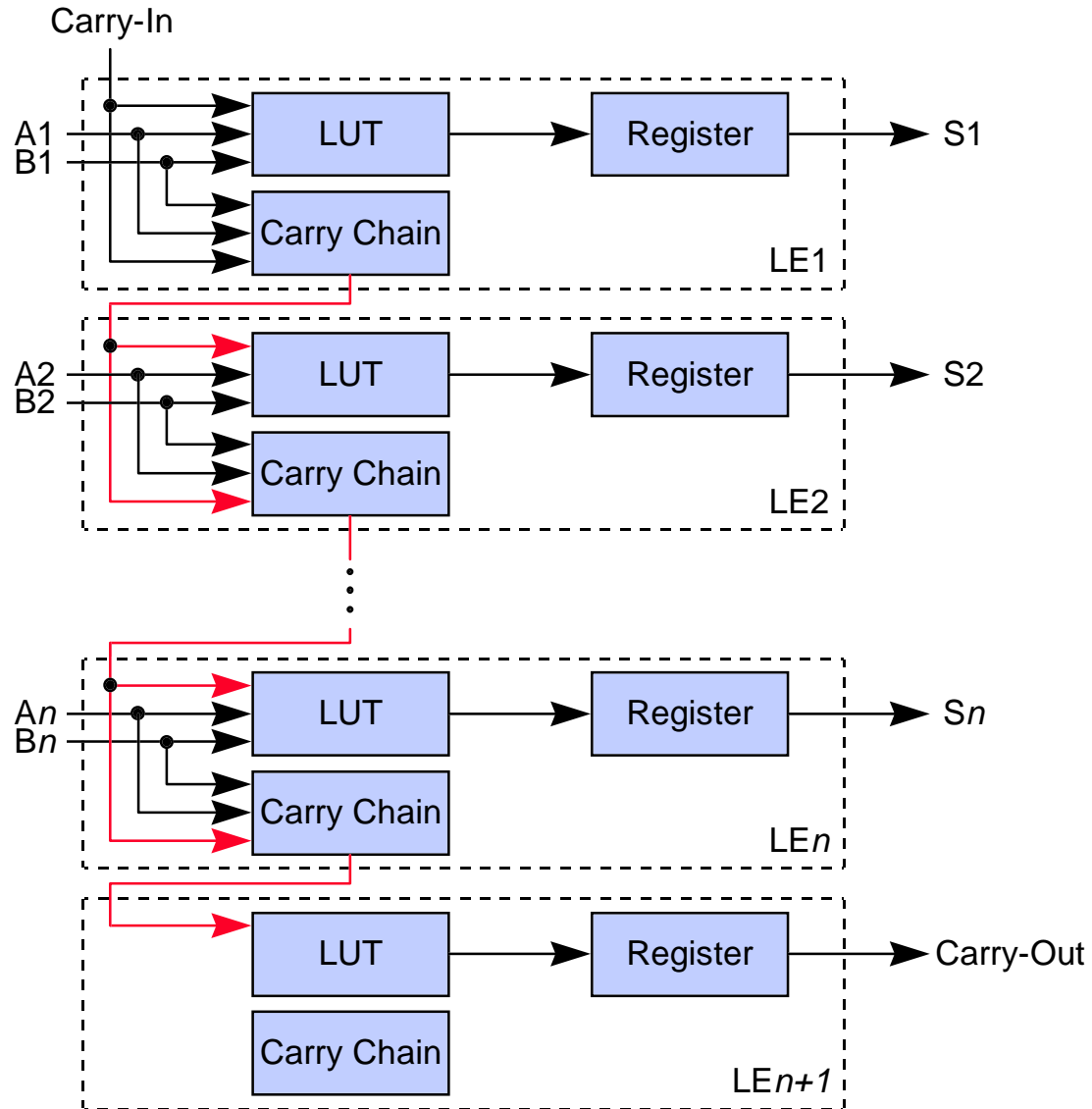




# FLEX 8000A Logic Element

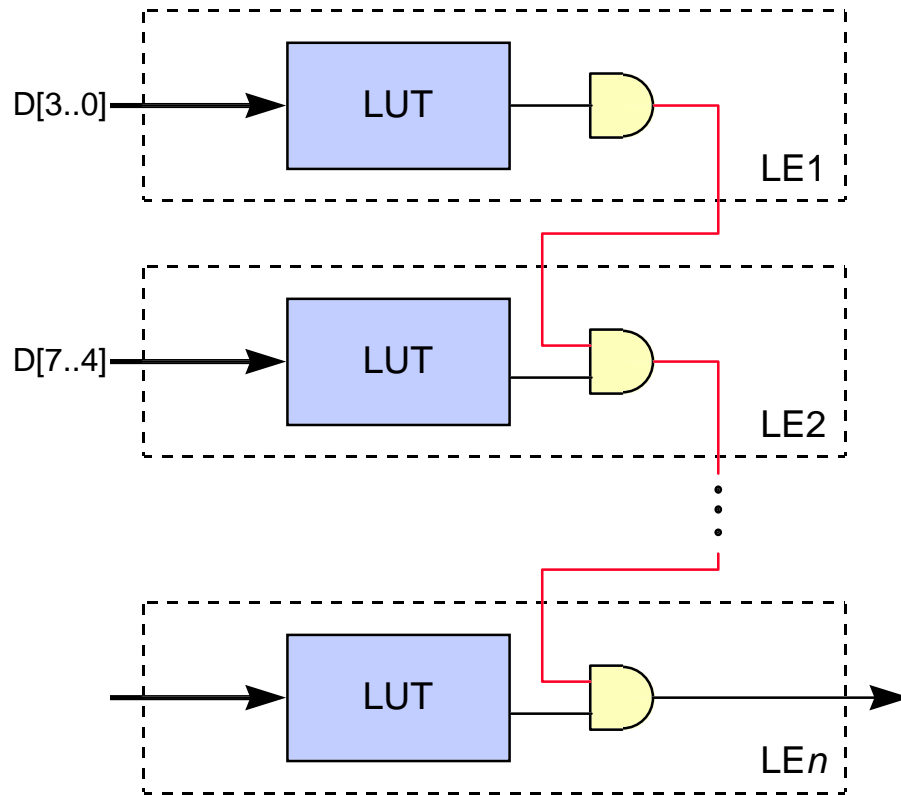


# Carry Chains

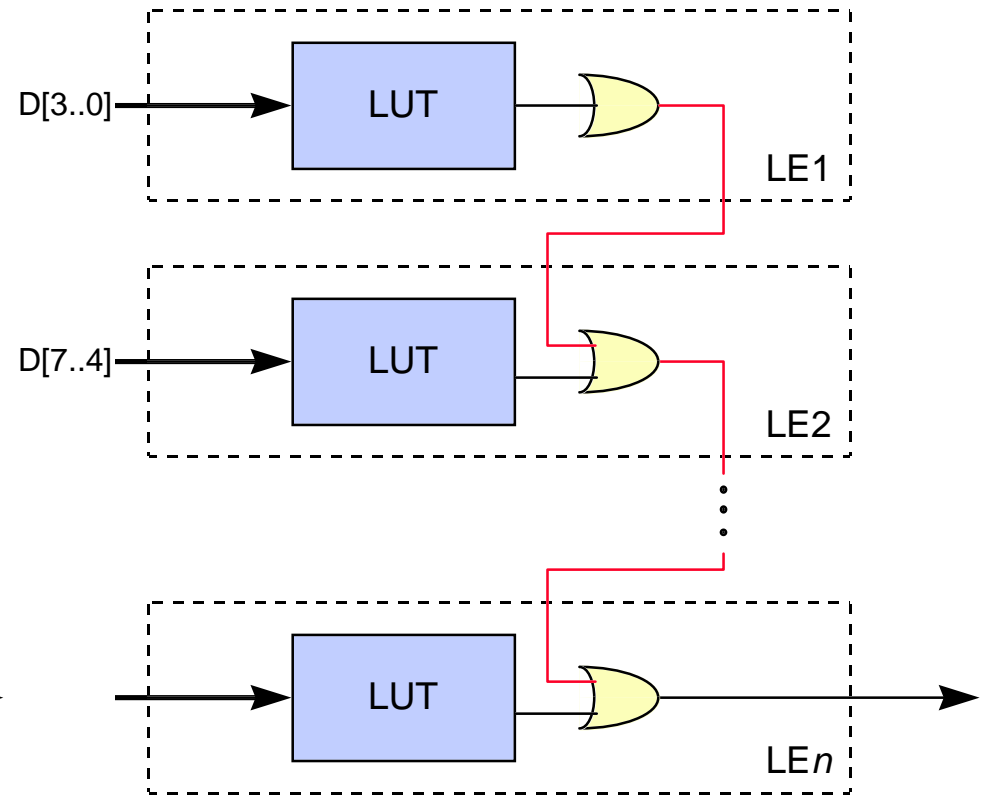


# Cascade Chains

AND Cascade Chain

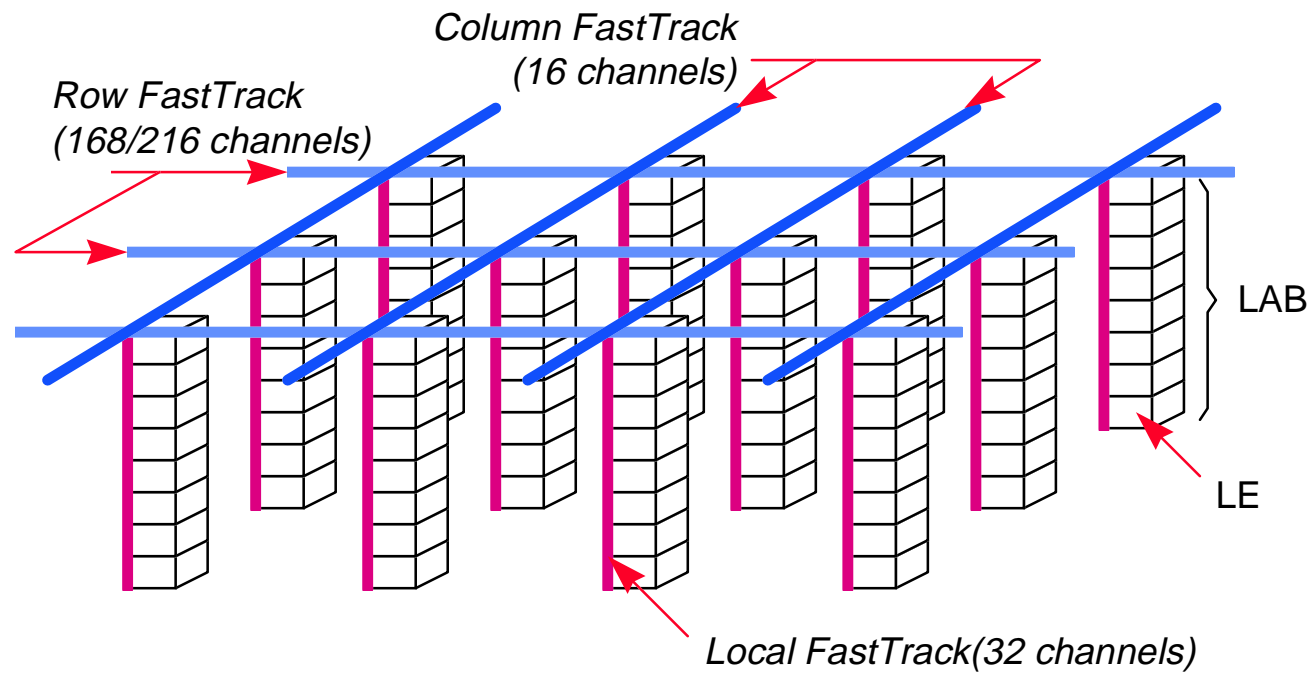


OR Cascade Chain

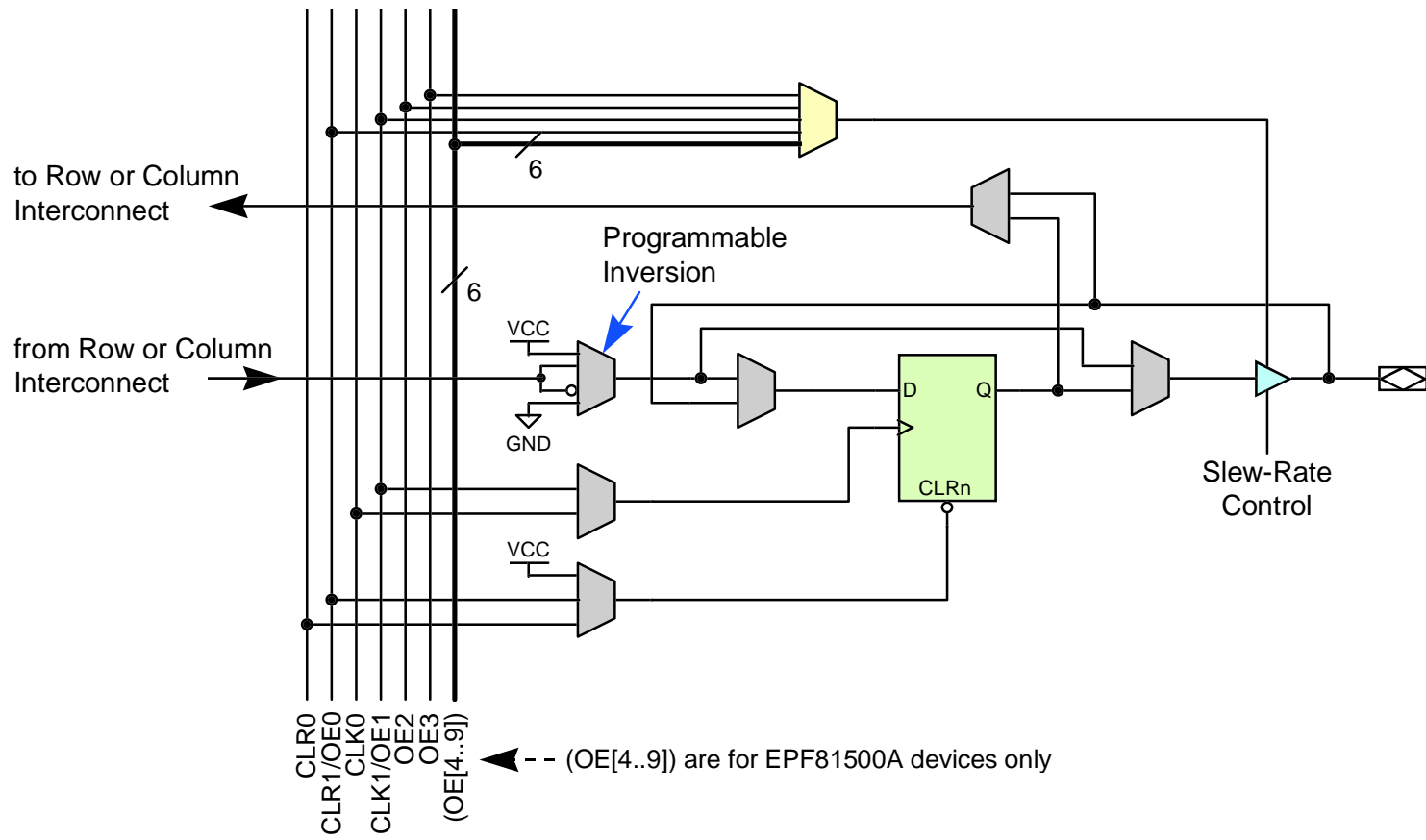




# FLEX 8000A FastTrack Interconnect



# FLEX 8000A I/O Element



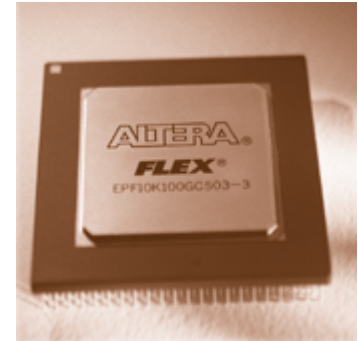
# FLEX 8000A Configuration

## ◆ Configuration schemes & data source

- Refer to Altera's *Application Notes* for details
  - AN033: *Configuring FLEX 8000 Devices*
  - AN038: *Configuring Multiple FLEX 8000 Devices*

	Configuration Scheme	Data Source
<b>AS</b>	(Active Serial)	Serial configuration EPROM
<b>APU</b>	(Active Parallel Up)	Parallel EPROM
<b>APD</b>	(Active Parallel Down)	Parallel EPROM
<b>PS</b>	(Passive Serial)	Serial data path (e.g. serial download cable)
<b>PPS</b>	(Passive Parallel Synchronous)	Intelligent host
<b>PPA</b>	(Passive Parallel Asynchronous)	Intelligent host

# FLEX 10K/A Families



## ◆ Today's FLEX 10K/A family members

Device	Gates	EAB	LEs	FFs	Speed Grade	Package Options	I/O Pins
<b>EPF10K10</b>	10,000	3	576	720	-3,-4	PLCC84, TQFP144, RQFP208	59,107,134
<b>EPF10K20</b>	20,000	6	1,152	1,344	-3,-4	TQFP144, RQFP208 /240	107,147,189
<b>EPF10K30</b>	30,000	6	1,728	1,968	-3,-4	TQFP144, RQFP208 /240, BGA356	107,147,189,246
<b>EPF10K40</b>	40,000	8	2,304	2,576	-3,-4	RQFP208/240	147,189
<b>EPF10K50</b>	50,000	10	2,880	3,184	-3,-4	RQFP240, BGA356, PGA403	189,274,310
<b>EPF10K70</b>	70,000	9	3,744	4,096	-3,-4	RQFP240, PGA503	189,358
<b>EPF10K100</b>	100,000	12	4,992	5,392	-3,-4	PGA503	406
<i>Available FLEX10KA Devices</i>							
<b>EPF10K50V</b>	50,000	10	2,880	3,184	-3,-4	RQFP240, BGA356	189,274
<b>EPF10K130V</b>	130,000	16	6,656	7,126	-3,-4	BGA596, PGA599	470



# FLEX 10K Features

## ◆ FLEX 10K/A main features...

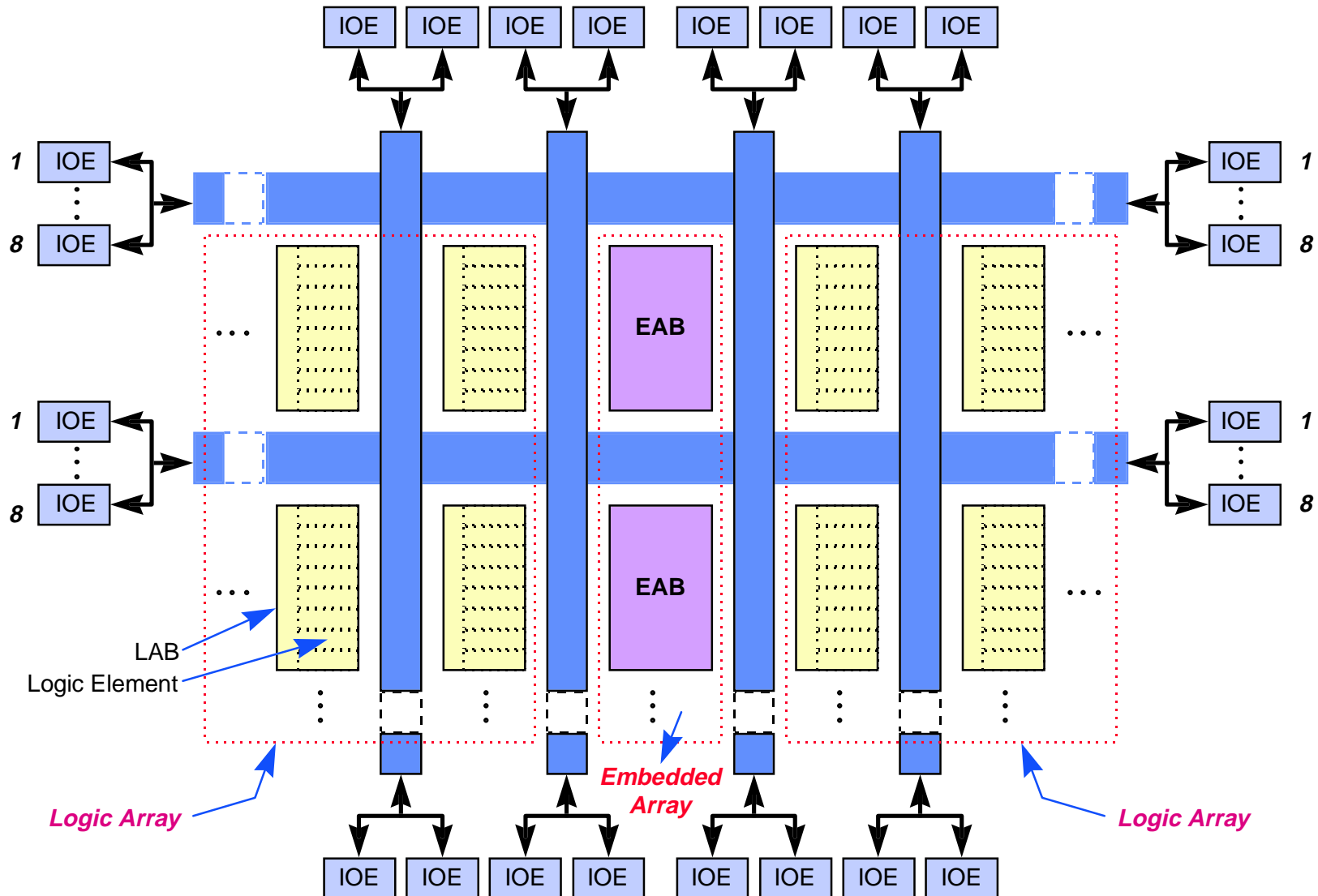
- **SRAM-based** devices based on Altera's FLEX architecture
- Embedded programmable logic family
  - **Embedded array** for implementing RAMs & specialized logic functions
  - Logic array for general logic functions
- High density
  - 10,000 ~ 100,000 typical gates (logic & RAMs)
  - 720 ~ 5,392 registers
  - 6,144 ~ 24,576 RAM bits
- Flexible interconnect
  - **FastTrack** continuous routing structure
  - Dedicated **carry chain** & **cascade chain**
  - Up to 6 global clock & 4 global clear signals

# FLEX 10K Features - (2)

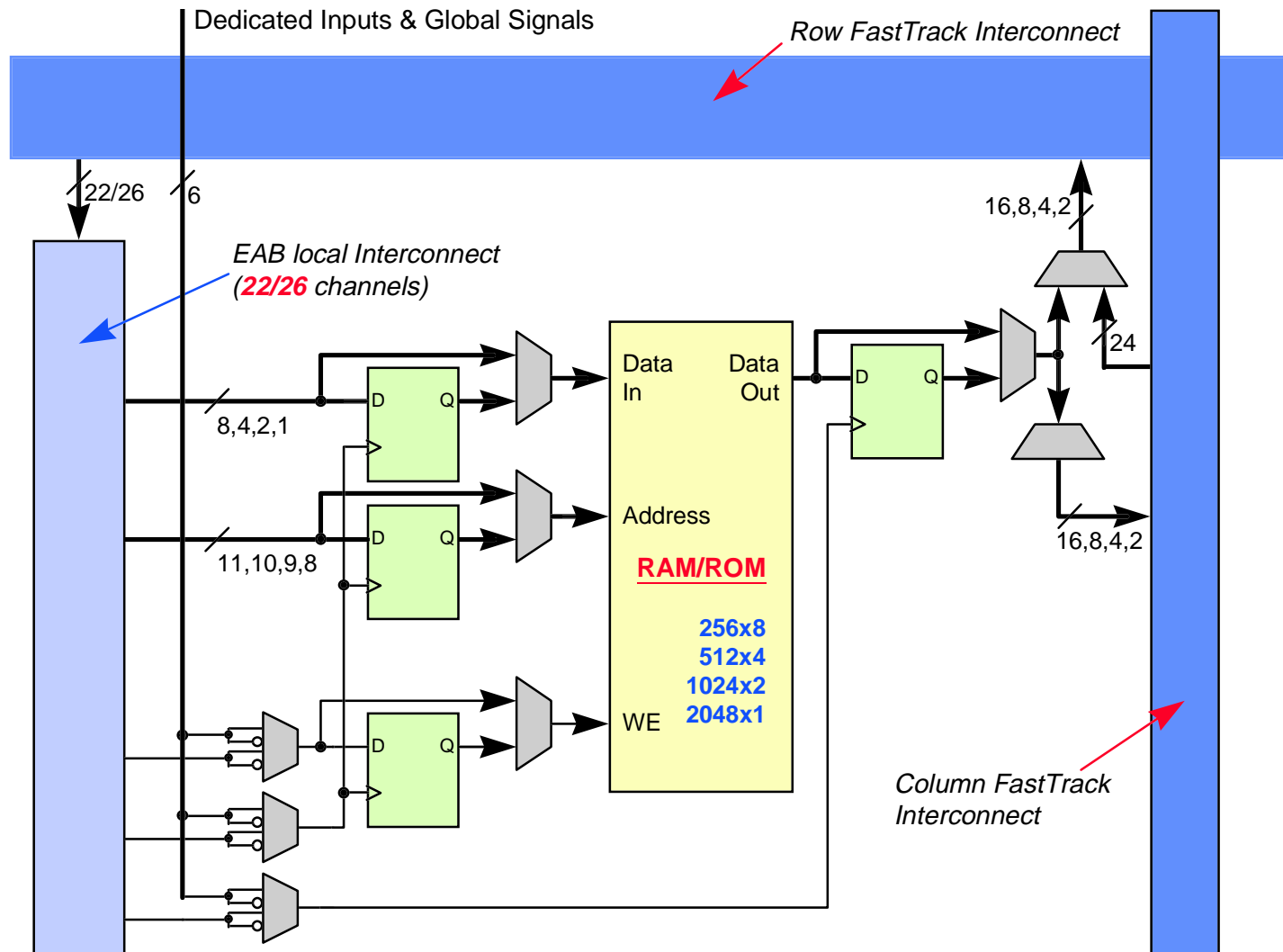
## ◆ FLEX 10K main features... (continued)

- Powerful I/O pins
  - Individual tri-state control for each pin
  - Programmable output slew-rate control
  - Open-drain option on each I/O pin
  - Peripheral register
- System-level features
  - Supports **in-circuit reconfiguration (ICR)**
  - JTAG boundary-scan test circuitry
  - PCI-compliant -3 speed grade
  - 3.3-V or 5-V I/O pins on devices in PGA, BGA & 208-pin QFP packages
  - **ClockLock & ClockBoost option** (for EPF10K100GC503-3DX device only)
- Flexible package options
  - Pin-compatibility with other FLEX 10K devices in the same packages

# FLEX 10K Architecture



# FLEX 10K Embedded Array Block



# What is the EAB?

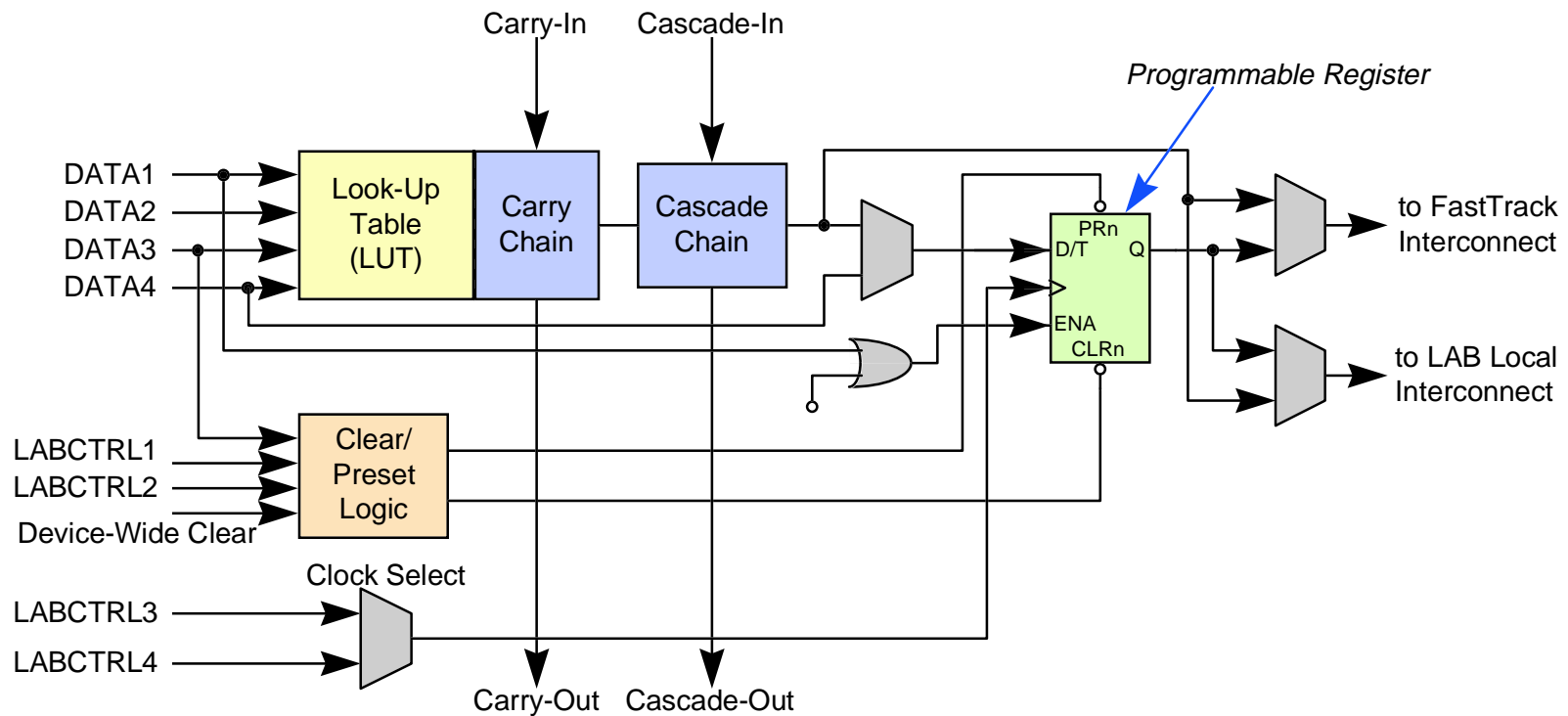
## ◆ What is the EAB?

- Larger block of RAM embedded into the PLD
- Can be preloaded with a pattern
- EAB size is flexible - 256x8 / 512x4 / 1024x2 / 2048x1
- You can combine EABs to create larger blocks
- Using RAM does not impact logic capacity

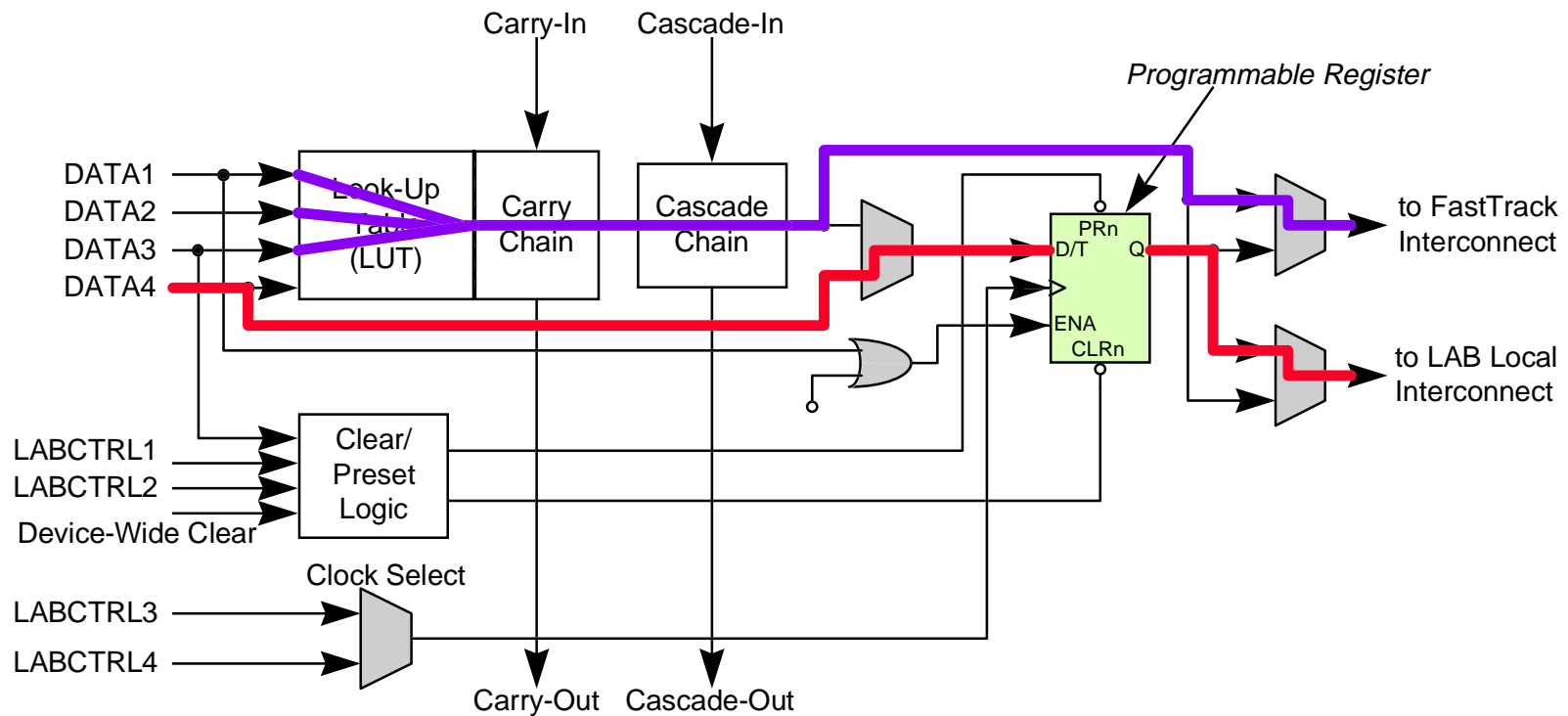
## ◆ EAB as logic

- EAB is preloadable at configuration time
- You can use EAB to create a large lookup table or ROM
- EAB is the same die size of 16 LEs, however, one EAB can perform complex functions requiring more than 16 LEs
  - Example: 4x4 Multiplier (40 LEs, 43MHz) vs. (1 EAB, 73MHz)

# FLEX 10K Logic Element



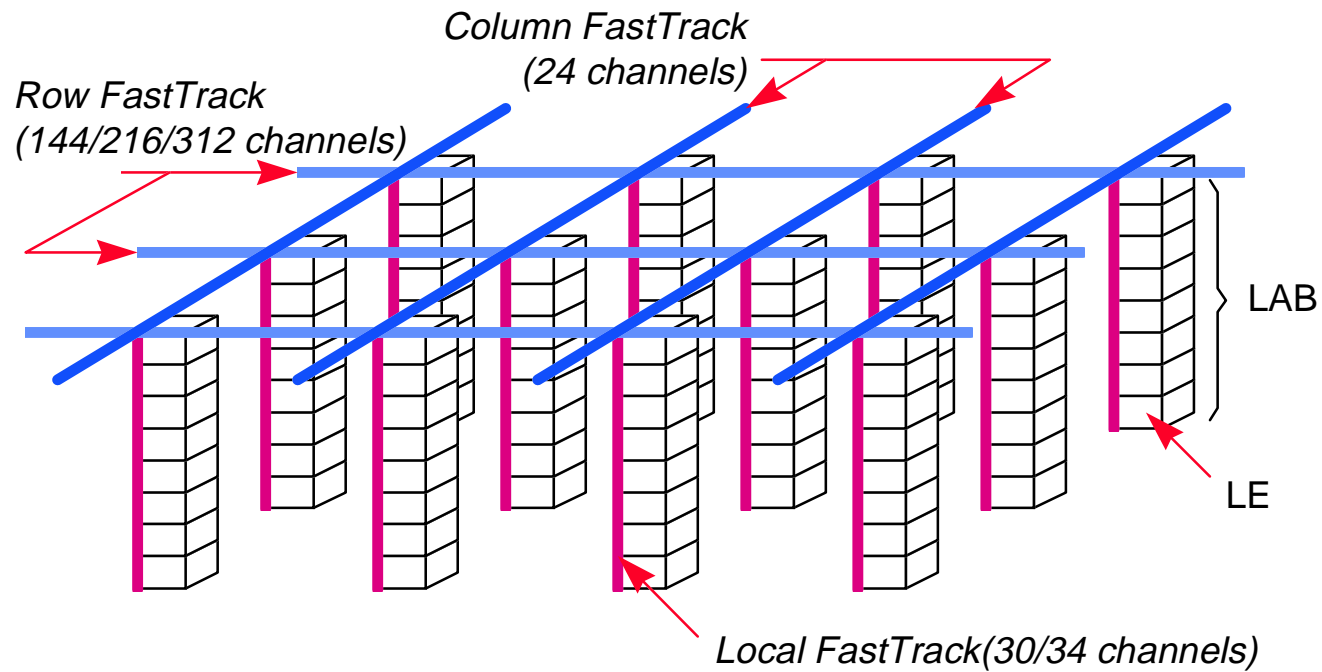
# FLEX 10K Register Packing



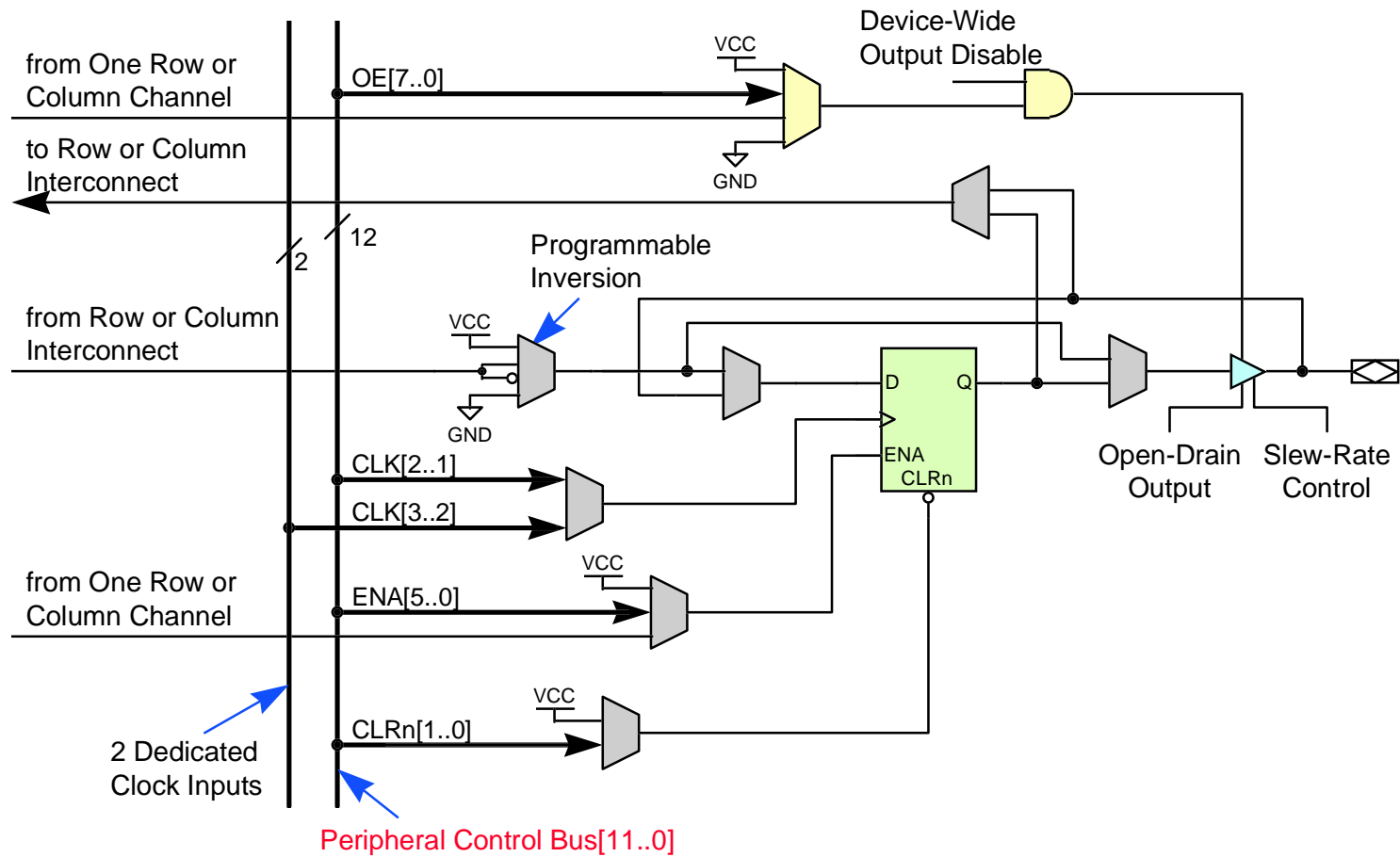




# FLEX 10K FastTrack Interconnect



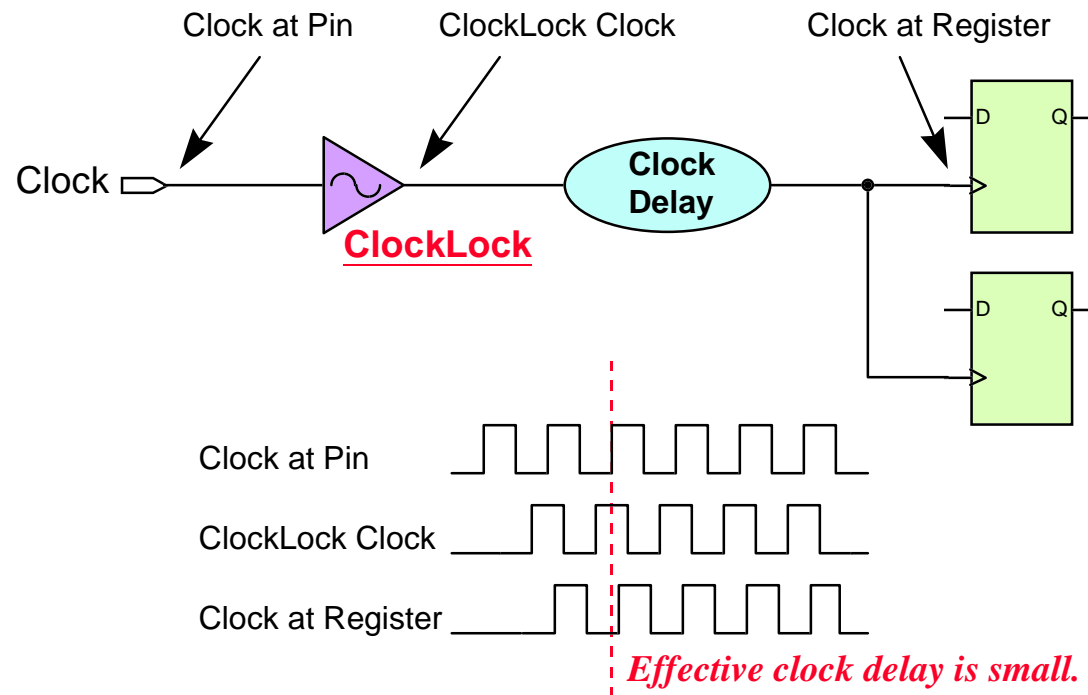
# FLEX 10K I/O Element



# ClockLock Feature

## ◆ ClockLock: faster system performance

- ClockLock feature incorporates a phase-locked loop (PLL) with a balanced clock tree to minimize on-device clock delay & skew



# ClockBoost Feature

## ◆ ClockBoost: increased system bandwidth & reduced area

- ClockBoost feature provides clock multiplication, which increases clock frequencies by as much as 4 times the incoming clock rate
- You can distribute a low-speed clock on the PCB with ClockBoost
- ClockBoost allows designers to implement time-domain multiplexed applications. The same functionality is accomplished with fewer logic resources.

– Note:

- (1) Up to now, only *EPF10K100-3DX* devices support ClockLock & ClockBoost features.
- (2) All new FLEX 10KA devices will support ClockBoost option.

# FLEX 10K Configuration

## ◆ Configuration schemes & data source

- Refer to Altera's *Application Notes* for details
  - *AN059: Configuring FLEX 10K Devices*
  - *AN039: JTAG Boundary-Scan Testing in Altera Devices*

Configuration Scheme	Data Source
<b>PS</b> (Passive Serial)	Altera's EPC1 configuration EPROM, BitBlaster or ByteBlaster download cable, serial data source
<b>PPS</b> (Passive Parallel Synchronous)	Intelligent host, parallel data source
<b>PPA</b> (Passive Parallel Asynchronous)	Intelligent host, parallel data source
<b>JTAG</b>	JTAG controller

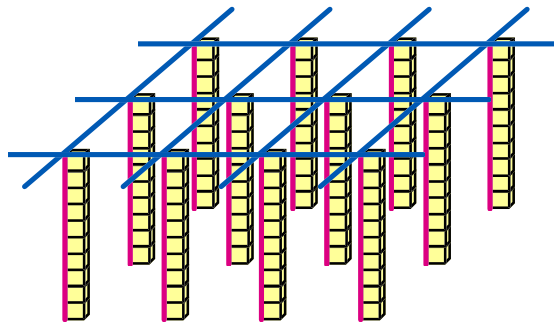
# FLEX Roadmap

## ◆ FLEX roadmap

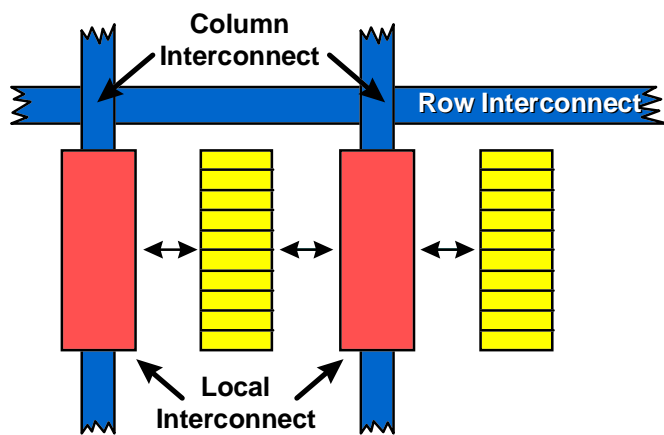
- High-density 3.3-V **FLEX10KA** family (0.35um, quad-layer-metal process)
  - Up to 250,000 gates (EPF10K250A: 12,160 LEs and 20 EABs)
  - “MultiVolt” I/O interface
- Low-cost **FLEX 6000** family (0.5um and 0.35um, triple-layer-metal process)
  - Up to 24,000 gates (EPF6024A: 1,960 LEs)
  - “OptiFLEX” architecture, advanced bond pad technology, interleaved LABs, and an optimized I/O structure to increase the level of programmable logic efficiency
  - “MultiVolt” I/O interface

# Appendix: FLEX 6000 Architecture

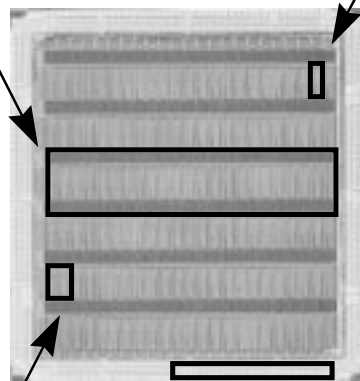
## FastTrack™ Interconnect



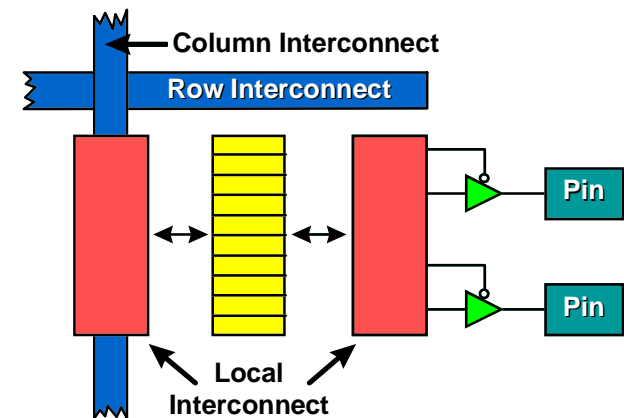
## Interleaved LABs



## FLEX 6000 Die

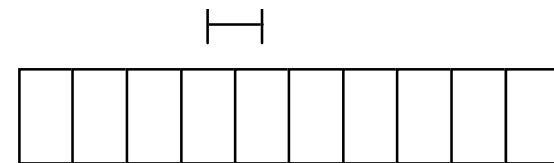


## FastFLEX™ I/O



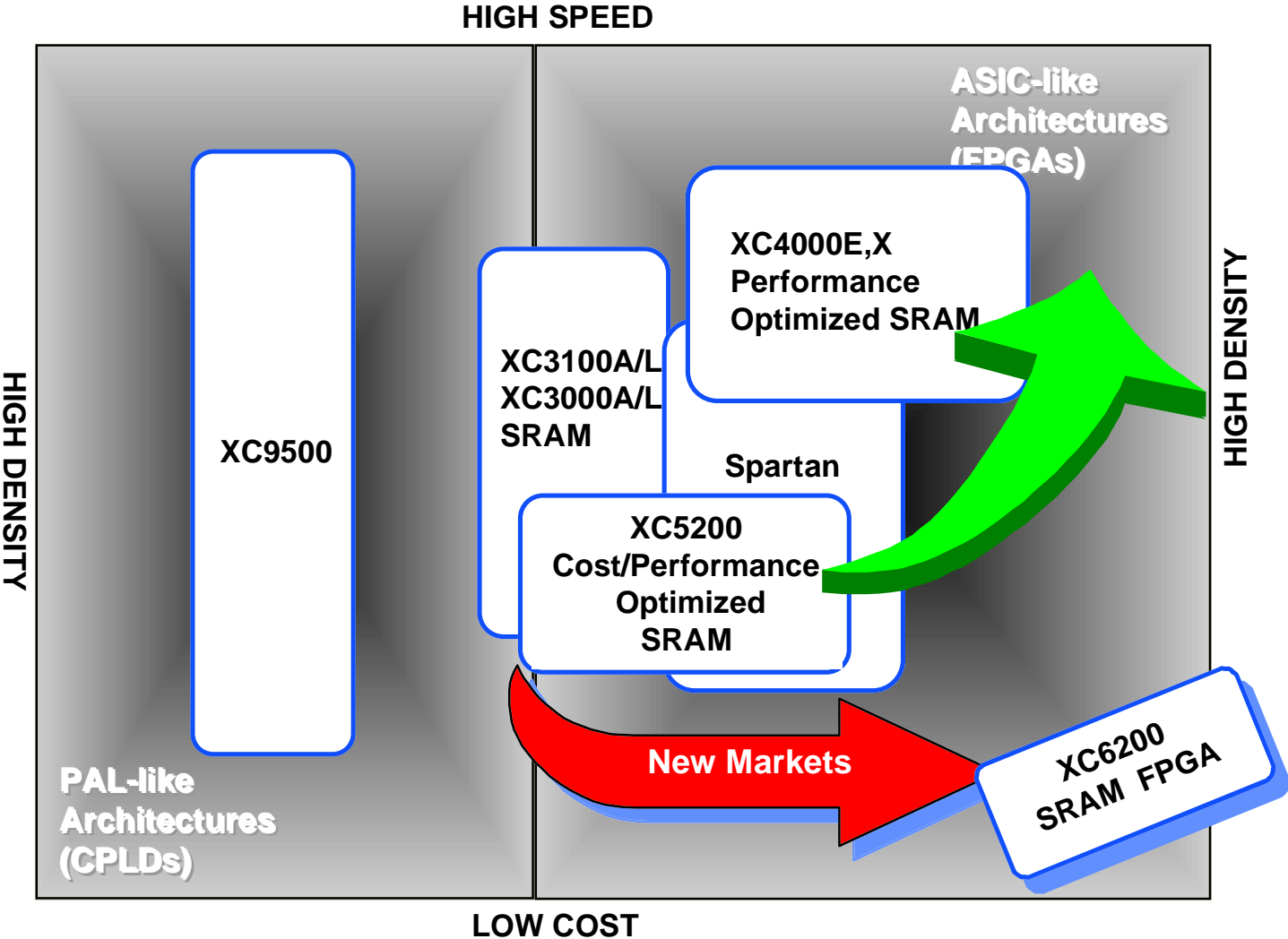
## μPitch™ Technology

3.2 mil (81 μm)



Bond Pads

# Xilinx Device Families





# Xilinx Device Families

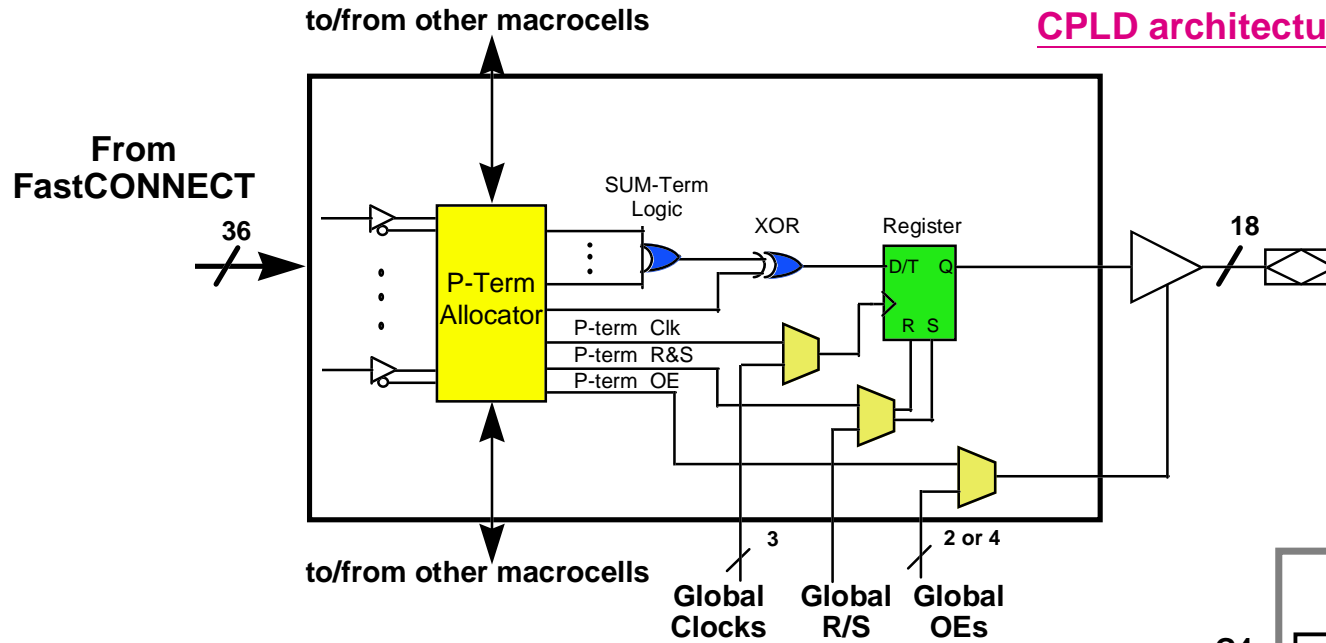
Device Family	Reconfigurable Element	Logic Cell Structure	Usable/Typical Gates	Family Members
XC3000A/L, XC3100A/L	SRAM	LUT	1,000 ~ 7,500	3020A/L, 3030A/L, 3042A/L, 3064A/L, 3090A/L 3120A, 3130A, 3142A/L, 3164A, 3190A/L, 3195A
XC4000E	SRAM	LUT	3,000 ~ 25,000	4003E, 4005E, 4006E, 4008E, 4010E, 4013E, 4020E, 4025E
XC4000EX/XL	SRAM	LUT	28,000 ~ 85,000	4005XL, 4010XL, 4013XL, 4020XL, 4028EX/XL, 4036EX/XL, 4044EX/XL, 4052XL, 4062XL, 4085XL
XC4000XV	SRAM	LUT	125,000 ~ 250,000	40125XV, 40150XV, 40200XV, 40250XV
SPARTAN/XL	SRAM	LUT	2,000 ~ 40,000	S05/XL, S10/XL, S20/XL, S30/XL, S40/XL
XC5200	SRAM	LUT	2,000 ~ 15,000	5202, 5204, 5206, 5210, 5215
XC6200 <sup>(2)</sup>	SRAM	SOG <sup>(1)</sup>	9,000 ~ 64,000	6209, 6216, 6236, 6264
XC7300	EPROM	SOP	400 ~ 3,800	7318, 7336/Q, 7354, 7372, 73108, 73144
XC9500	FLASH	SOP	800 ~ 12,800	9536, 9572, 95108, 95144, 95180, 95216, 95288, 95432, 95576

**Note:**

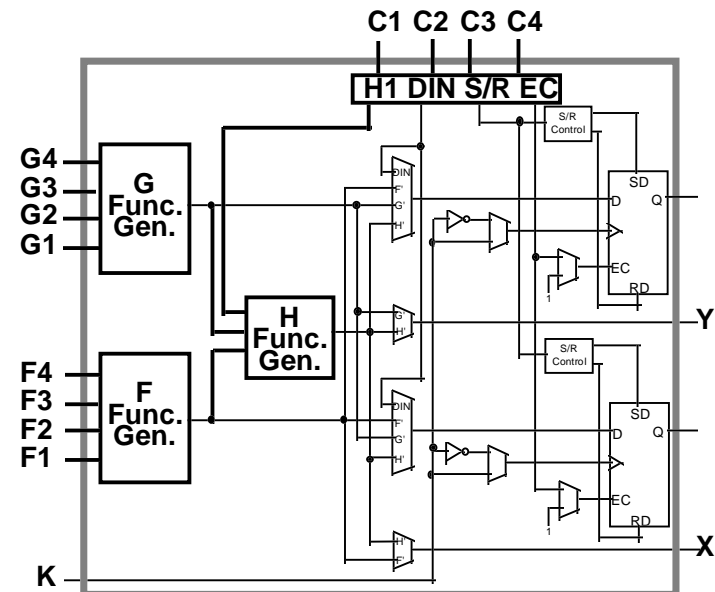
(1) XC6200 series is a special family of fine-grain, sea-of-gates FPGAs.

# Xilinx FPGA & CPLD Architectures

CPLD architecture



FPGA architecture



# Choose of FPGAs and CPLDs

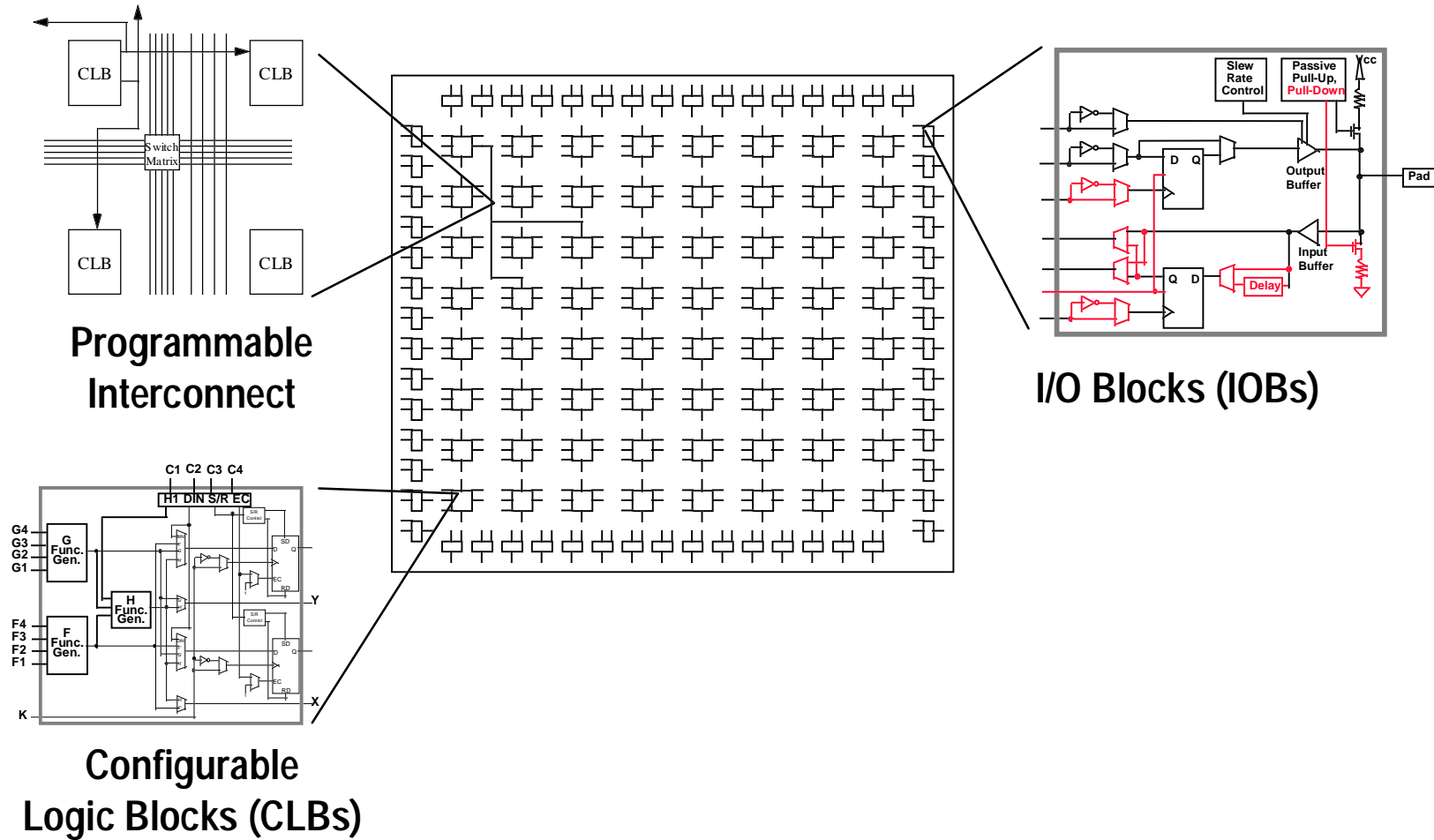
## ◆ Choose the appropriate architecture

- Different PLD architectures provide different performance & capacity results for same application

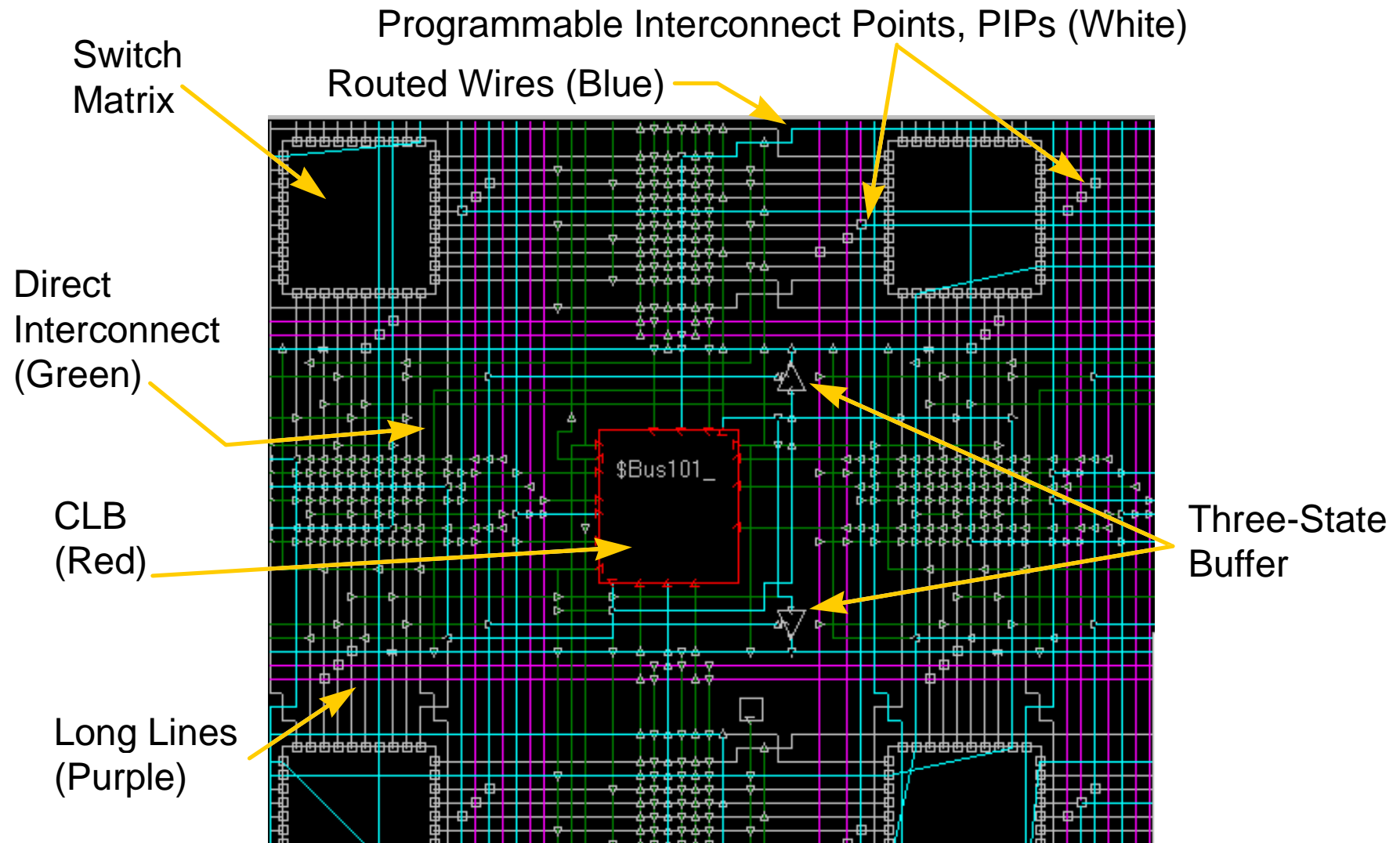
*This training course will focus on FPGA devices.*

Feature	CPLD Architecture	FPGA Architecture
Basic Building Block	Course Grain	Fine Grain
Logic Cell Structure	SOP	LUT
Technology	Non-Volatile Memory	SRAM
Optimization	Combinational-Intensive Logic e.g. Large Decoders, State Machines, ...	Register-Intensive, Arithmetic Functions e.g. Adders, Comparators, Counters, ...

# Xilinx FPGA Architecture



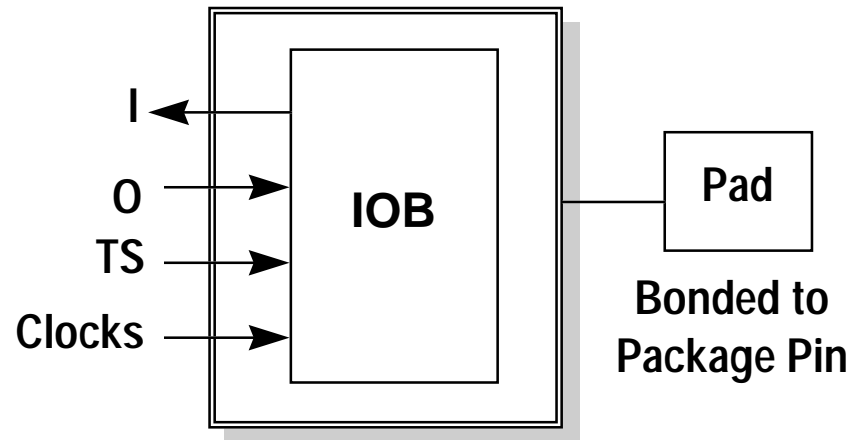
# What's Really in the Chip?



# I/O Block (IOB)

## ◆ Periphery of identical I/O blocks

- Input, output, or bi-directional
- Registered, latched, or combinational (XC3000/XC4000)
- Tri-state output
- Programmable output slew rate



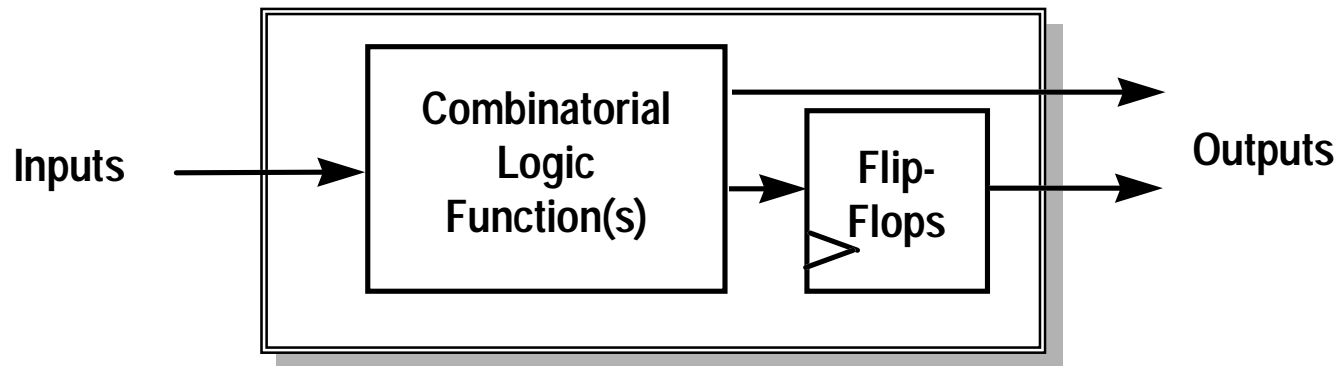
# Configurable Logic Block (CLB)

- ◆ **Combinatorial logic via lookup table**

- Any function(s) of available inputs

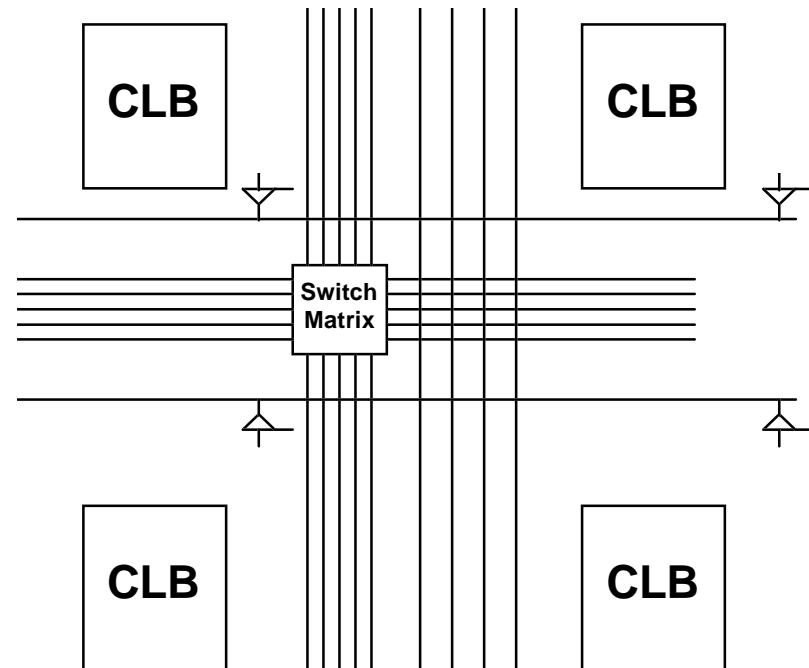
- ◆ **Output registered and/or combinatorial**

- Latches in XC5200



# Programmable Interconnect

- ◆ Resources to create arbitrary interconnection networks
- ◆ Various types of interconnect
  - Fast direct interconnect
  - Flexible general-purpose interconnect
  - Low-skew long lines
- ◆ Internal three-state buffers
  - 2 tri-state buffers per CLB for buses
    - For buses and wide functions





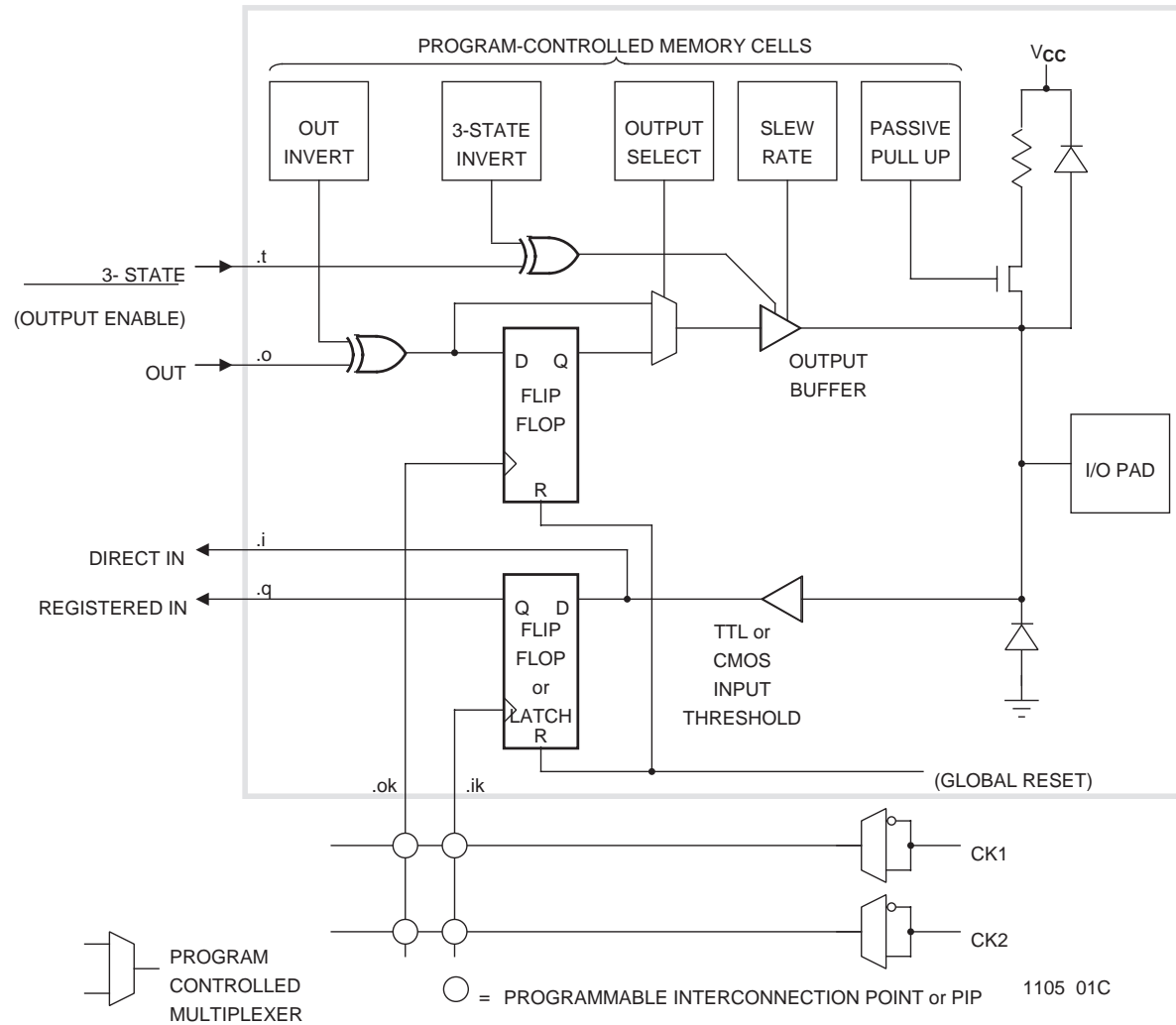
# Special Resources

- ◆ Global clock buffers
- ◆ Global reset net
- ◆ Internal oscillator
- ◆ XC4000/XC5200 special resources
  - Global three-state net
  - Arithmetic carry logic
  - Wide decode or cascade functions
  - Boundary scan

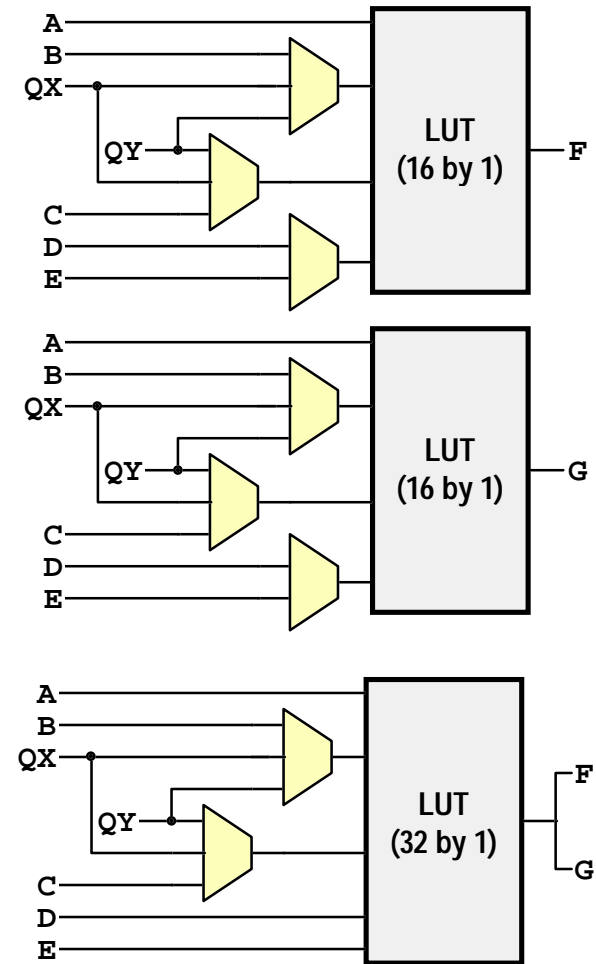
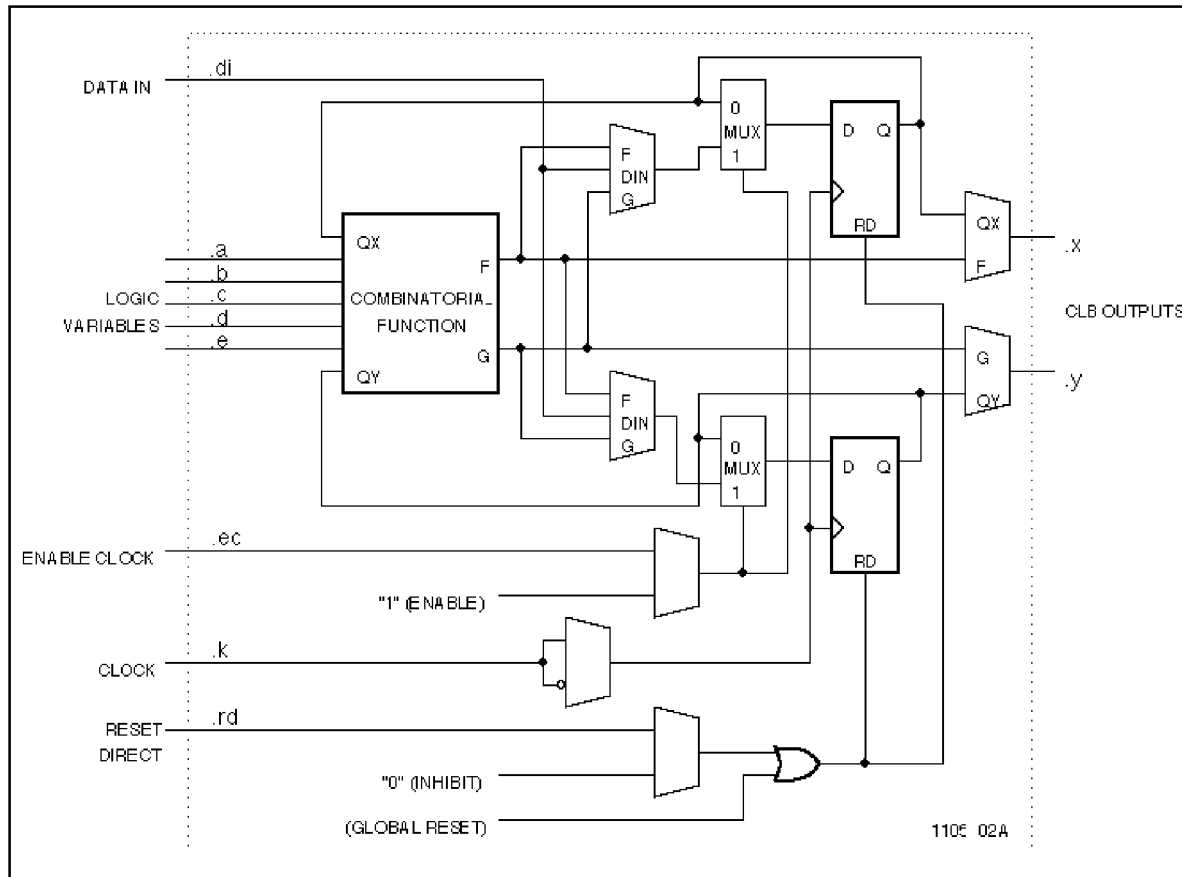
# XC3000 Series Family

<u>Part</u>	<u>Typical Gates</u>	<u>CLB Matrix</u>	<u>CLBs</u>	<u>Flip-Flops</u>	<u>IOBs</u>
3120A	1,300-1,800	8x8	64	256	64
3130A	2,000-2,700	10x10	100	360	80
3142A	2,500-3,700	12x12	144	480	96
3164A	4,000-5,500	14x16	224	688	120
3190A	5,000-7,000	16x20	320	928	144
3195A	6,500-8,500	22x22	484	1,320	176

# XC3000 IOB



# XC3000 CLB



XC3000 Configurable Function Generator

# XC3000 Interconnect Resources

## ◆ General-purpose interconnect

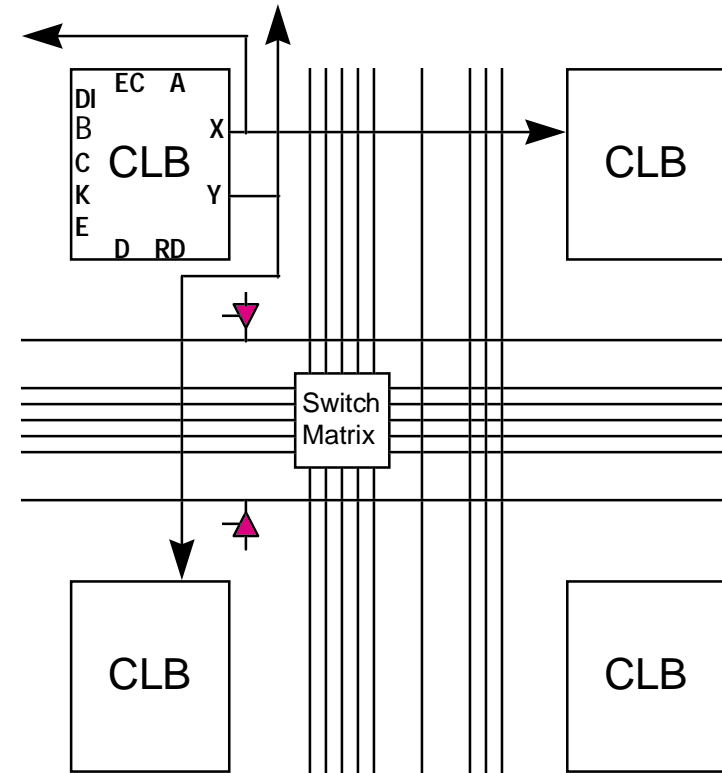
- Switching matrix
- PIPs (Programmable Interconnection Point)

## ◆ Direct interconnect

- X -> right B / left C
- Y -> above D / below A
- Die edge direct interconnect

## ◆ Longlines

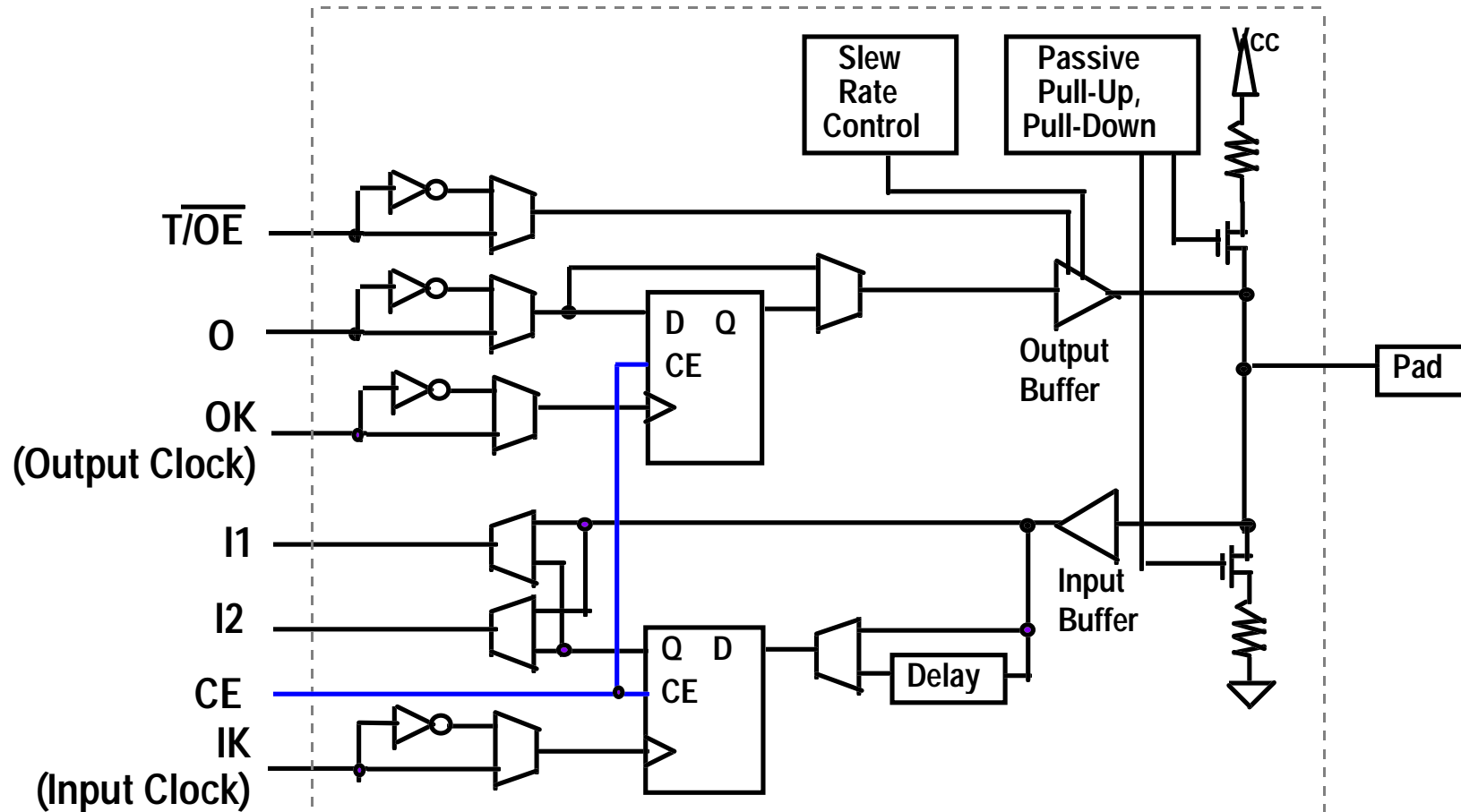
- 3 vertical longlines per column
- 2 horizontal longlines per row
- 2 additional outer longlines
- 2 Global nets : global buffer, alternate buffer & global reset



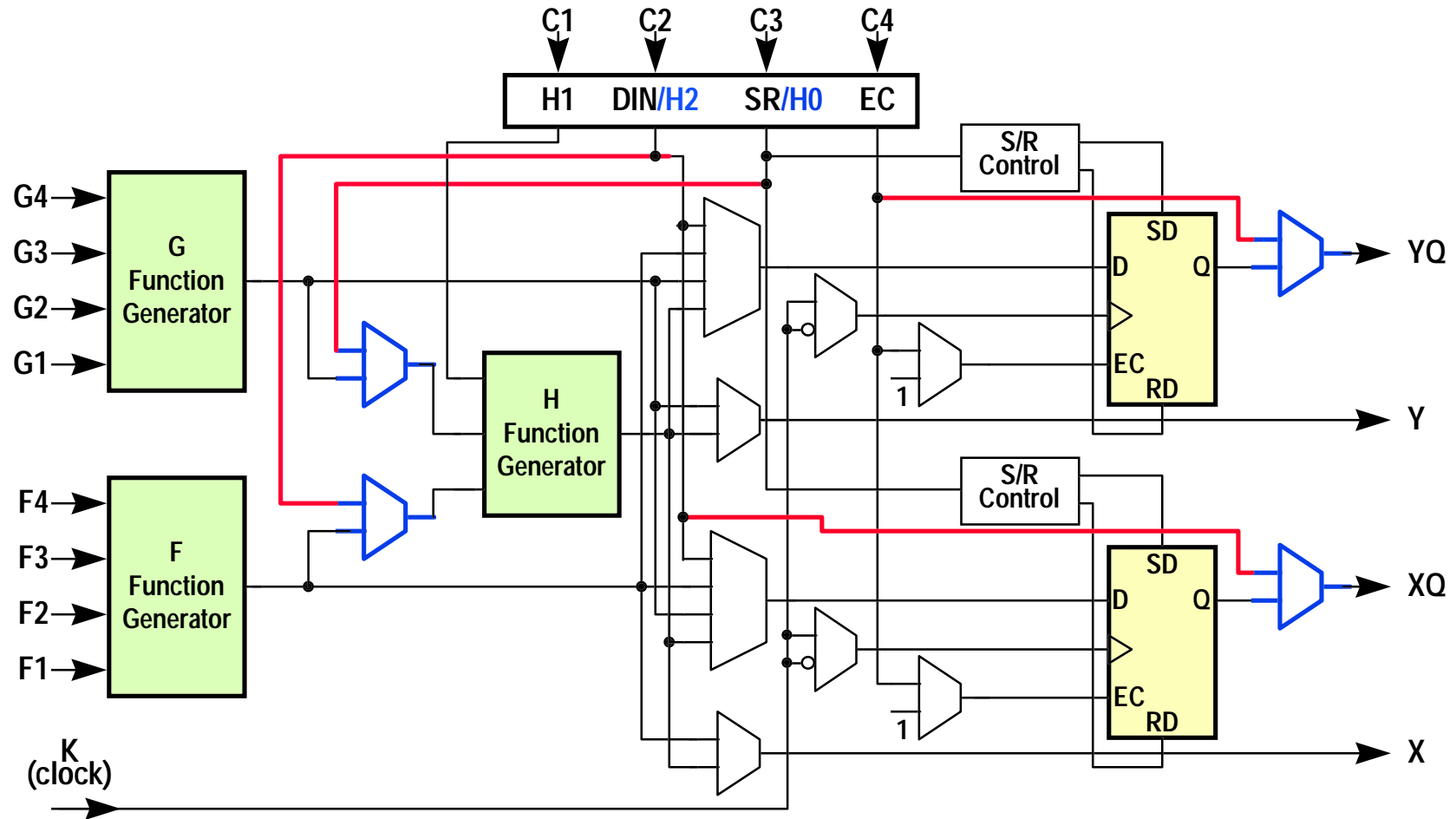
# XC4000E Family

<u>Part</u>	<u>MAX Logic Gates</u>	<u>CLB Matrix</u>	<u>CLBs</u>	<u>Flip-Flops</u>	<u>IOBs</u>
4003E	3,000	10x10	100	360	80
4005E	5,000	14x14	196	616	112
4006E	6,000	16x16	256	768	128
4008E	8,000	18x18	324	936	144
4010E	10,000	20x20	400	1,120	160
4013E	13,000	24x24	576	1,536	192
4020E	20,000	28x28	784	2,016	224
4025E	25,000	32x32	1,024	2,560	256

# XC4000E IOB

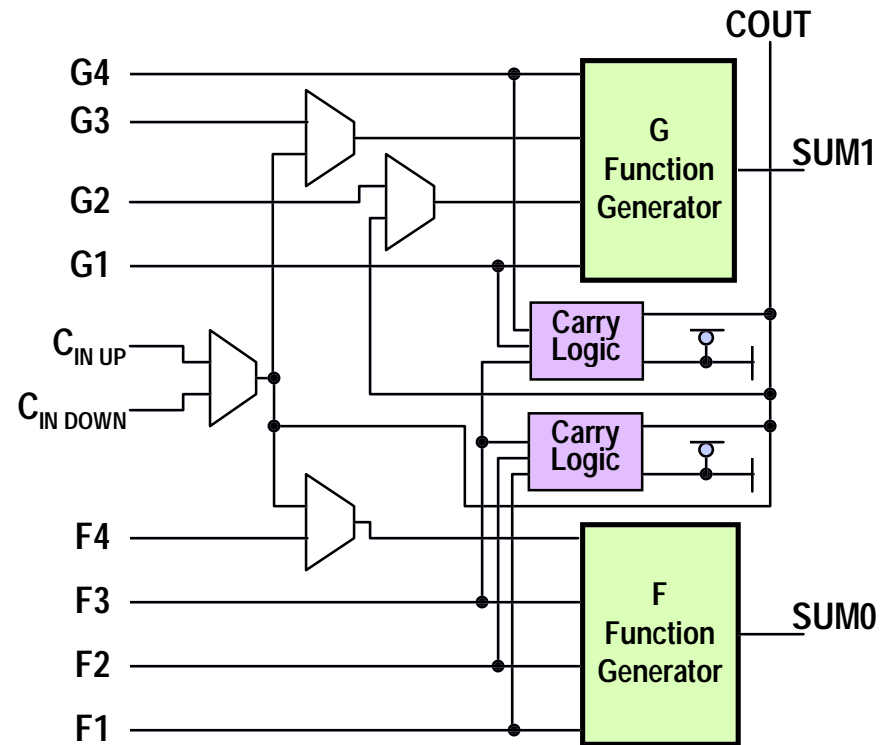


# XC4000E CLB





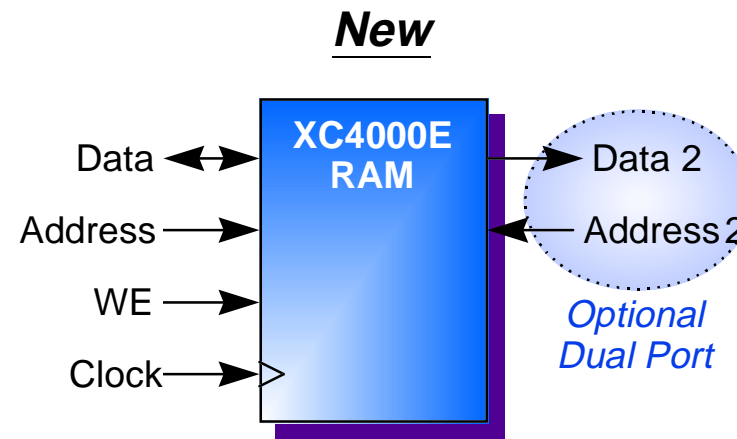
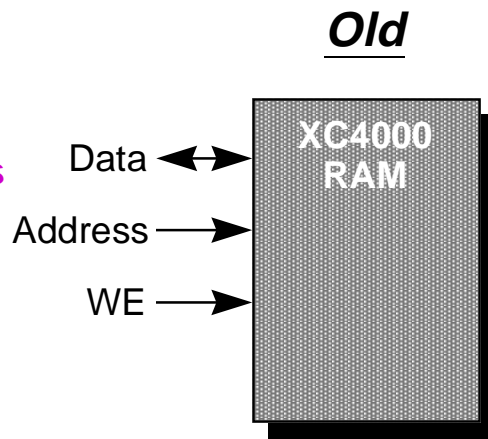
# XC4000E Dedicated Carry Logic



# XC4000E Select-RAM Memory

## ◆ CLB can be used as RAM

- Asynchronous
- Timing critical
- Longer design times
- Single port
- Programmable only during device operation
- 4ns read time



- Synchronous
- Simple timing
- Simple to use
- Dual port support
- Programmable during device operation or at start-up configuration
- 3ns read time

RAM Size	No. of Read Port	Timing Modes	XC4000/A/D/H	XC4000E
16x1	Single-Port	Level-Sensitive	✓	✓
		Edge-Triggered		✓
16x2	Dual-Port	Edge-Triggered		✓
	Single-Port	Level-Sensitive	✓	✓
32x1	Single-Port	Edge-Triggered		✓
		Level-Sensitive		✓
		Edge-Triggered	✓	✓

# XC4000E Interconnect Resources

## ◆ Single-length lines

- Switch matrix
- PIPs

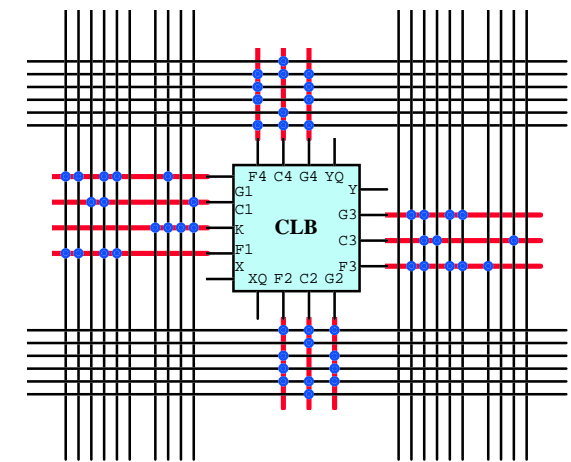
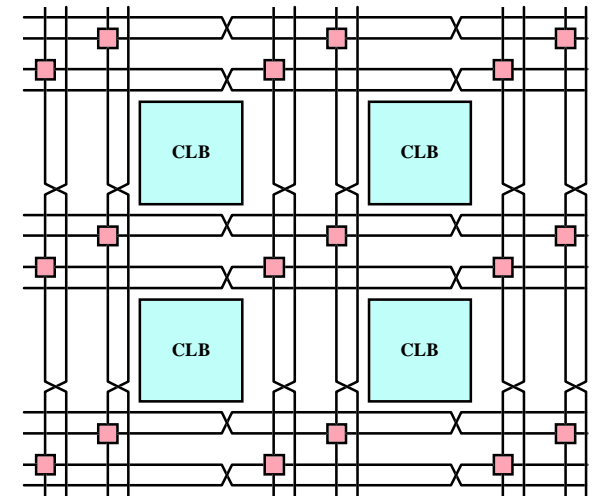
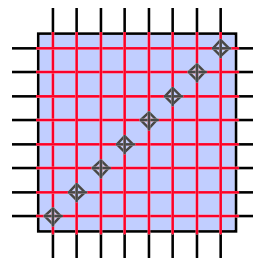
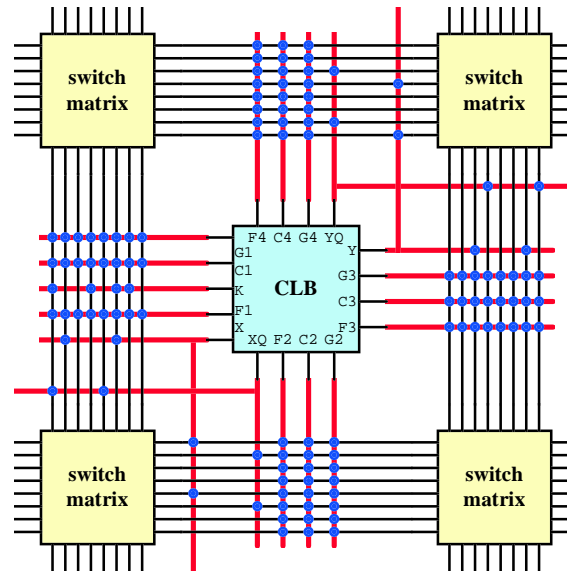
## ◆ Double-length lines

## ◆ Longlines

- Vertical longlines
- Horizontal longlines
- Global longlines :  
global reset & 8 global buffers

## ◆ Dedicated paths

- Fast carry paths



XC4000E Routing Resources

# XC4000X Series (EX/XL) Families

<u>Part</u>	<u>MAX Logic Gates</u>	<u>CLB Matrix</u>	<u>CLBs</u>	<u>Flip-Flops</u>	<u>IOBs</u>
4005XL	5,000	14x14	196	616	112
4010XL	10,000	20x20	400	1,120	160
4013XL	13,000	24x24	576	1,536	192
4020XL	20,000	28x28	784	2,016	224
4028EX/XL	28,000	32x32	1,024	2,560	256
4036EX/XL	36,000	36x36	1,296	3,168	288
4044XL	44,000	40x40	1,600	3,840	320
4052XL	52,000	44x44	1,936	4,576	352
4062XL	62,000	48x48	2,304	5,376	384
4085XL	85,000	56x56	3,136	7,168	448

# XC4000X Series Enhancements

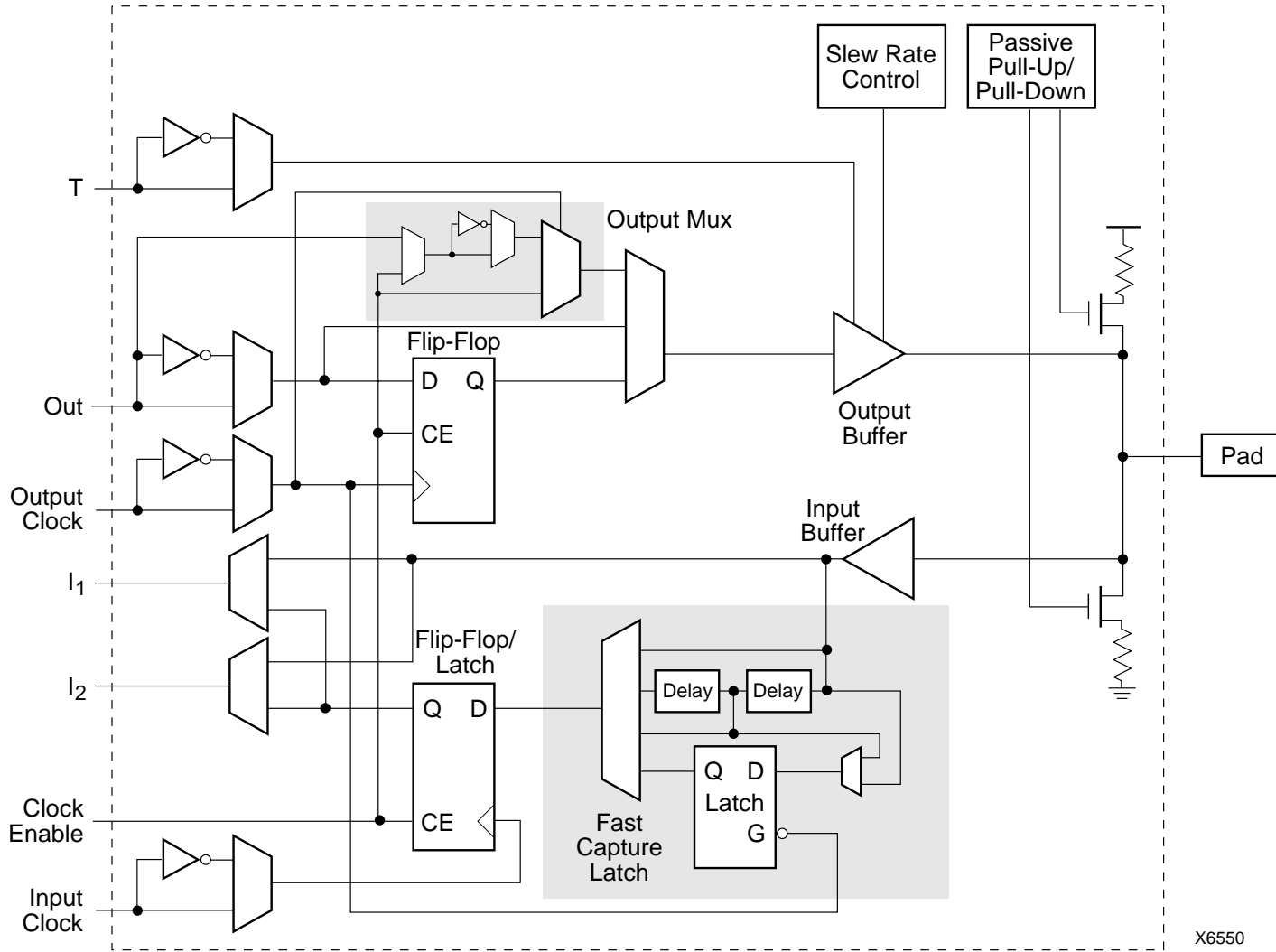
## ◆ The XC4000X Series contains XC4000E features:

- Synchronous, single and dual-port Select RAM memory
- Dedicated JTAG boundary scan logic
- High speed carry logic
- Wide Edge decoders

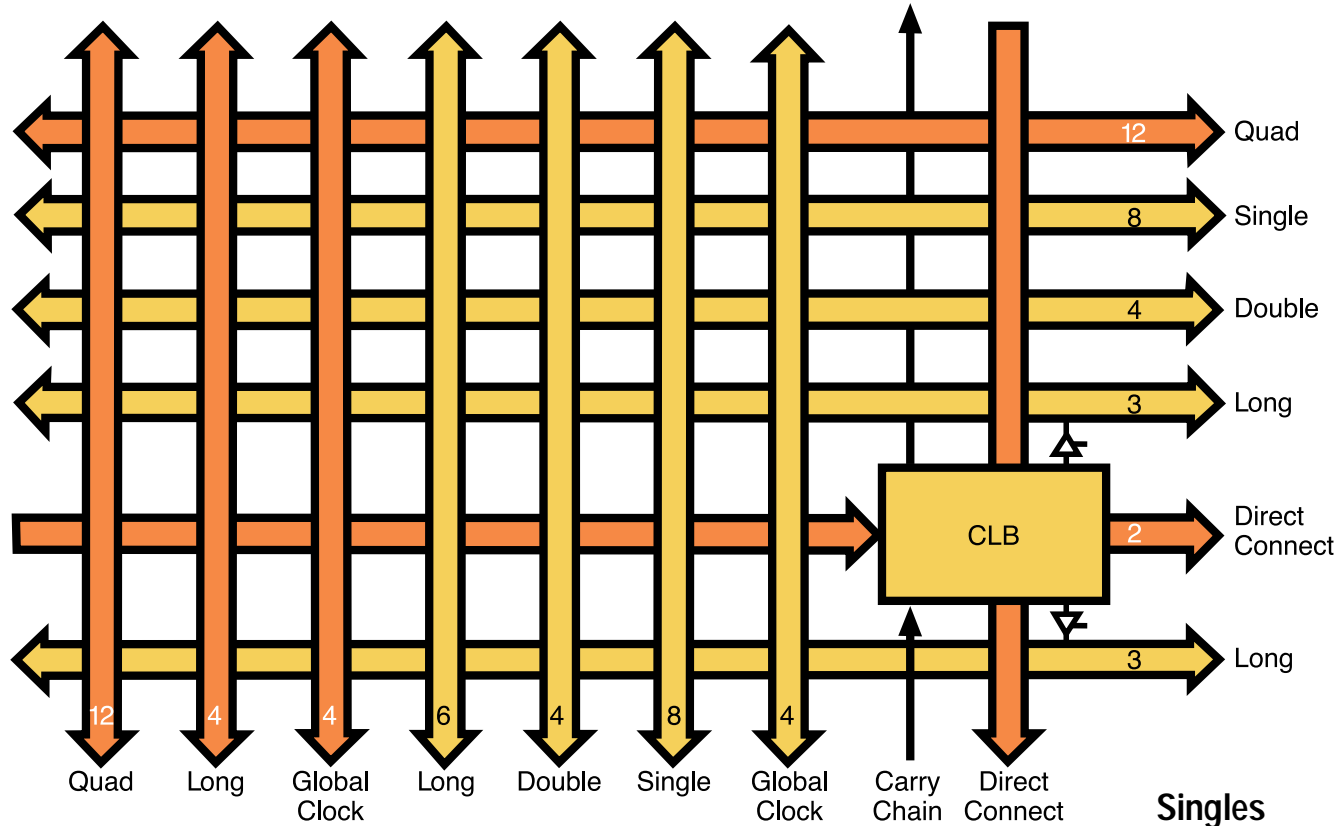
## ◆ In addition to:



- Twice the routing resources
- New high speed quad interconnect resources
- High speed 3 level clocking network
- VersaRing I/O added for pin assignment flexibility
- Latch capability in CLBs
- Improved carry logic for faster adders, multipliers and DSP functions

# XC4000X IOB



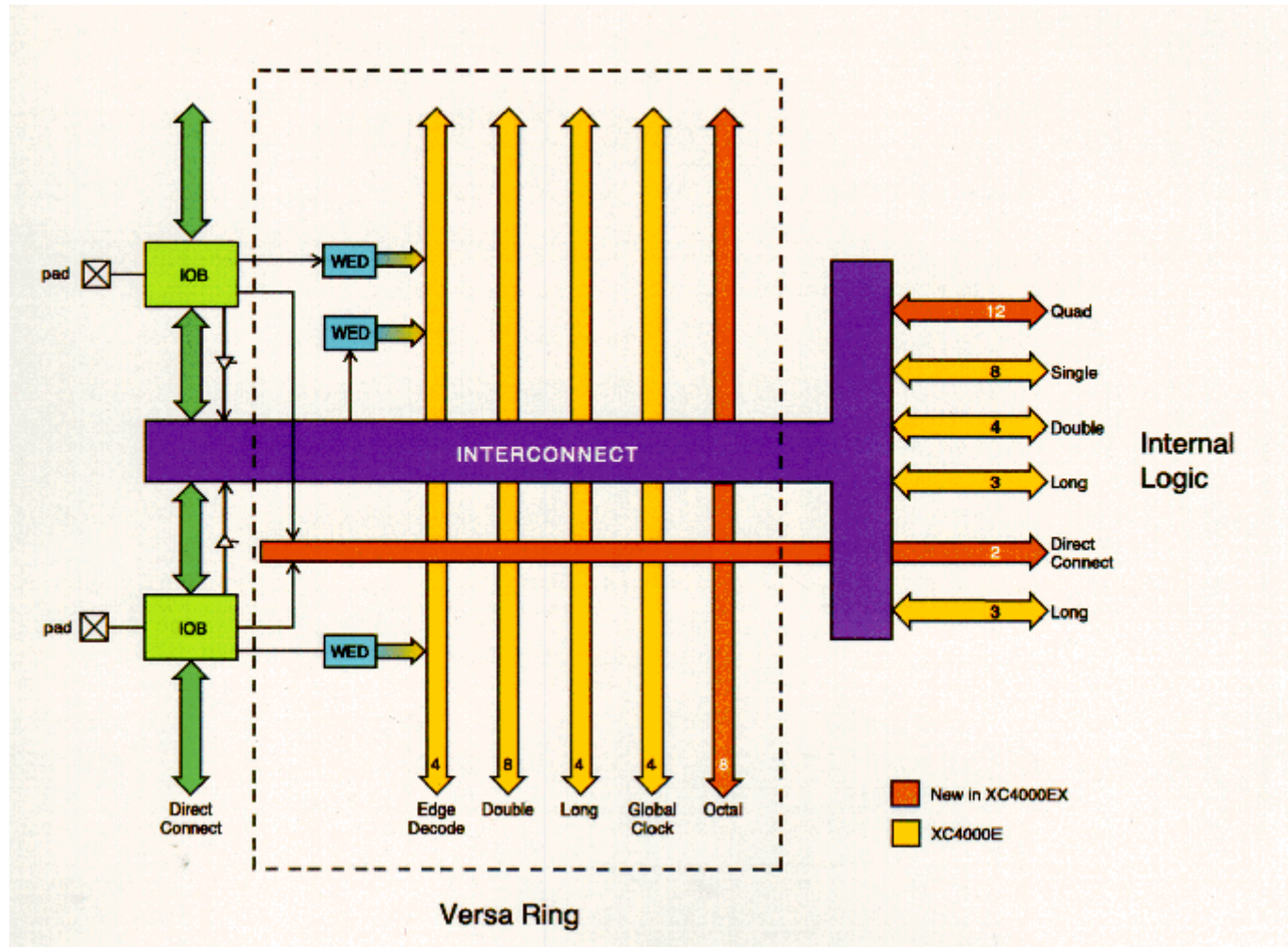
# XC4000X Interconnect Resources



 New in XC4000EX  
 XC4000 E

	XC4000E		XC4000X	
	Vertical	Horizontal	Vertical	Horizontal
<b>Singles</b>	8	8	8	8
<b>Doubles</b>	4	4	4	4
<b>Quads</b>	0	0	12	12
<b>Longlines</b>	6	6	10	6
<b>Direct Connects</b>	0	0	2	2
<b>Globals</b>	4	0	8	0
<b>Carry Logic</b>	2	0	1	0
<b>Total</b>	24	18	45	32

# XC4000X VersaRing

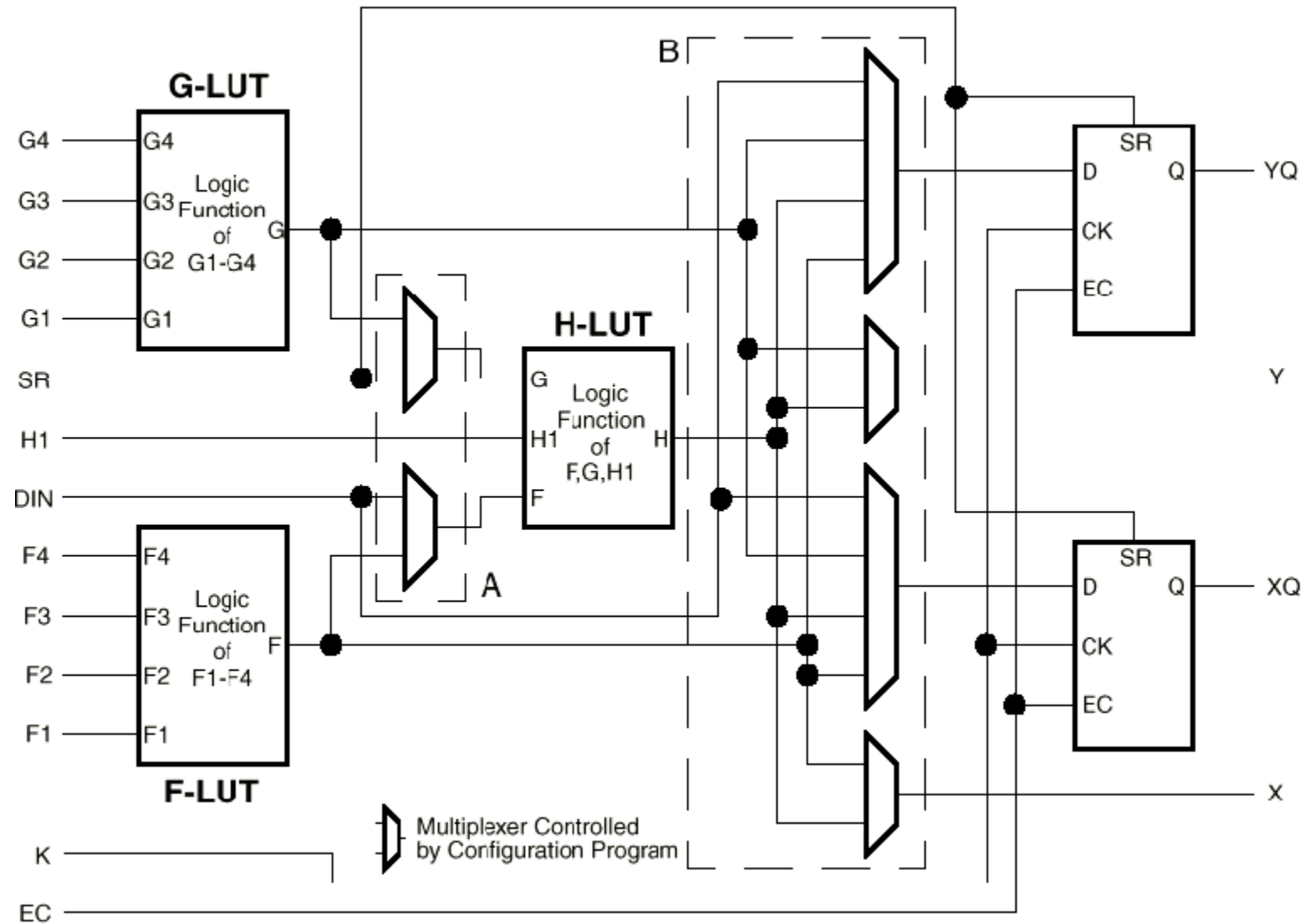




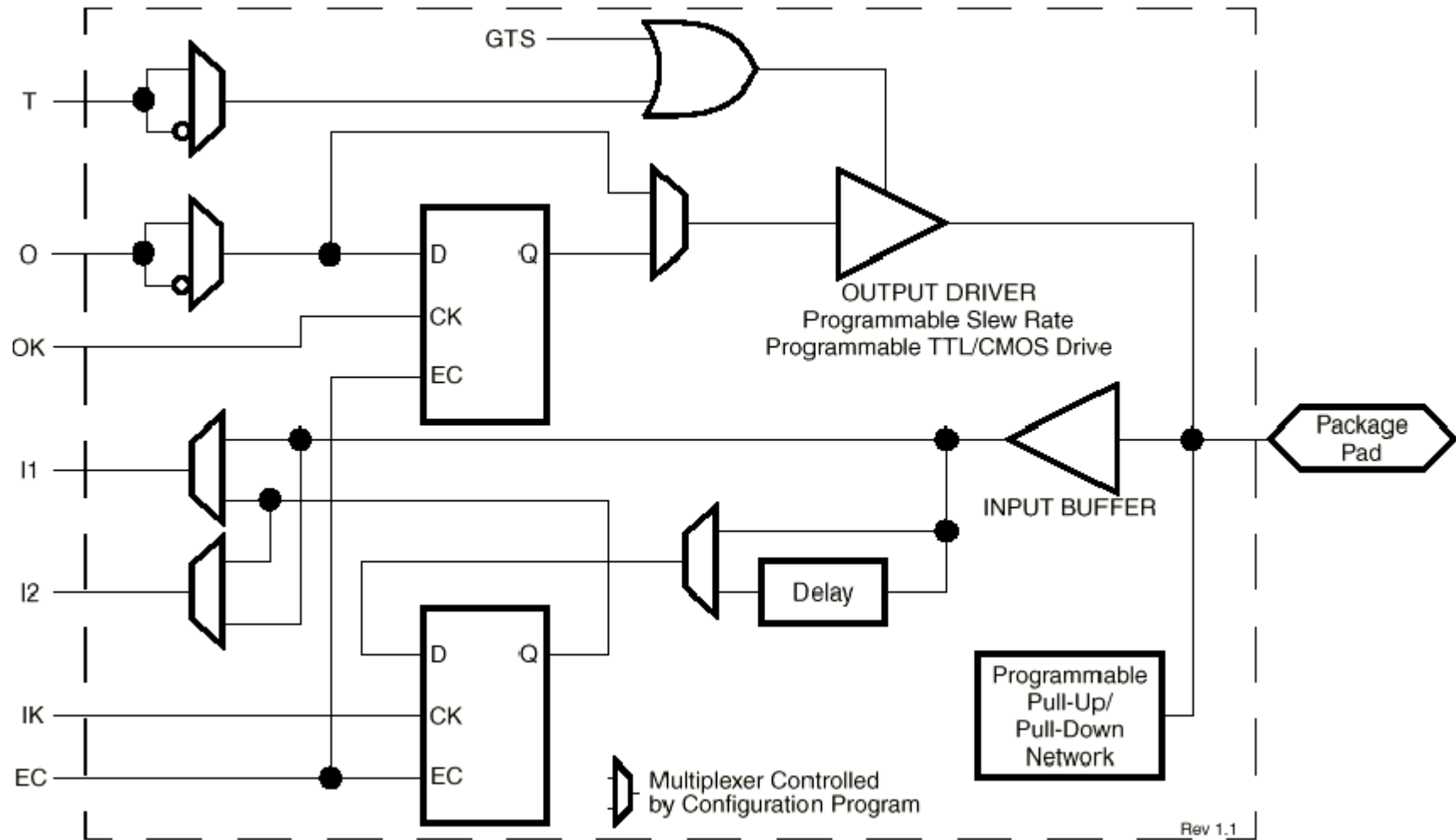
# Spartan Family

<u>Part</u>	<u>Gates</u>	<u>CLB Matrix</u>	<u>CLBs</u>	<u>Flip-Flops</u>	<u>IOBs</u>
S05/XL	2,000 ~ 5,000	10x10	100	360	80
S10/XL	3,000 ~ 10,000	14x14	196	616	112
S20/XL	7,000 ~20,000	20x20	400	1,120	160
S30/XL	10,000 ~ 30,000	24x24	576	1,536	192
S40/XL	13,000 ~ 40,000	28x28	784	2,016	224

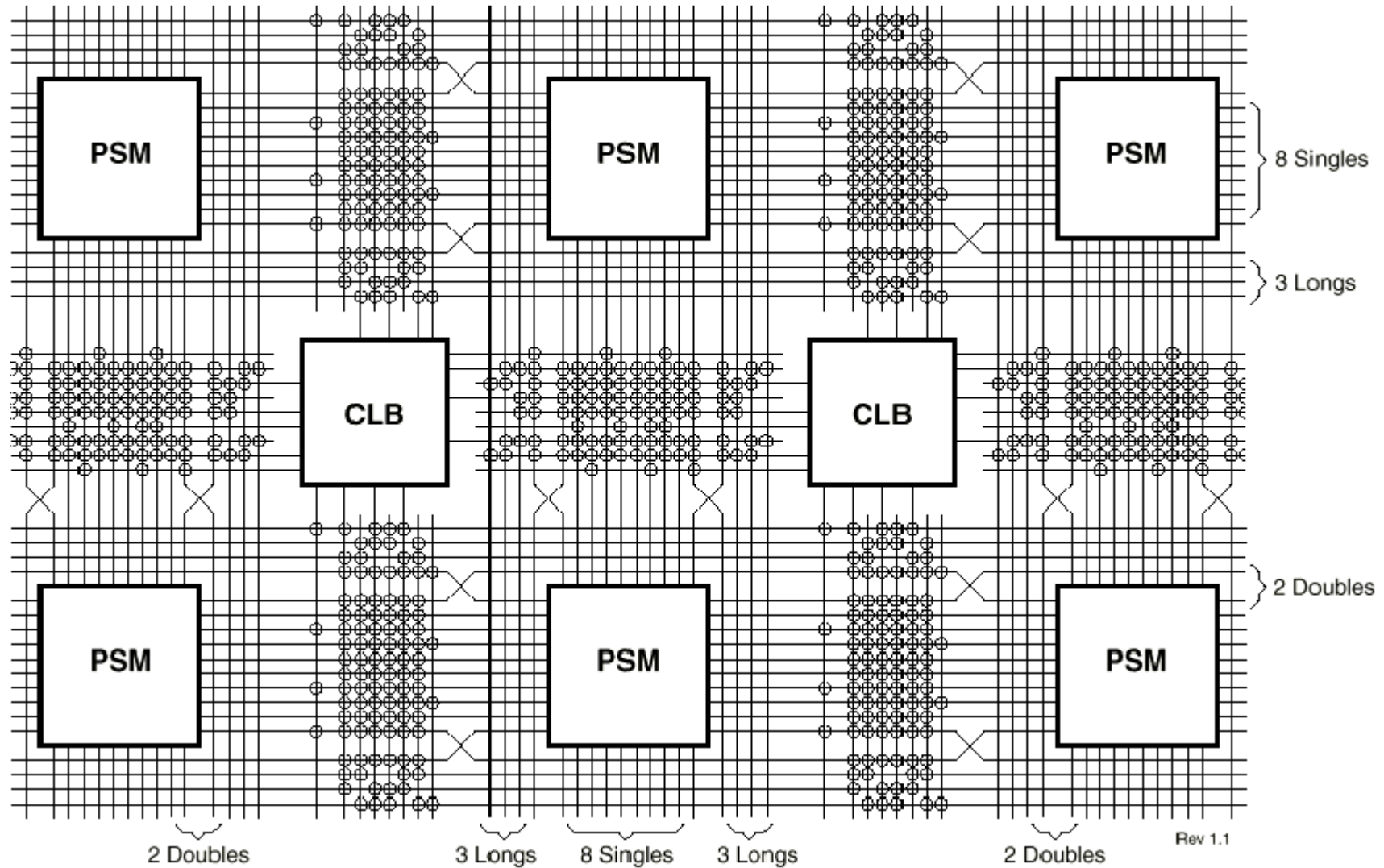
# Spartan Configurable Logic Block (CLB)



# Spartan I/O Block Diagram



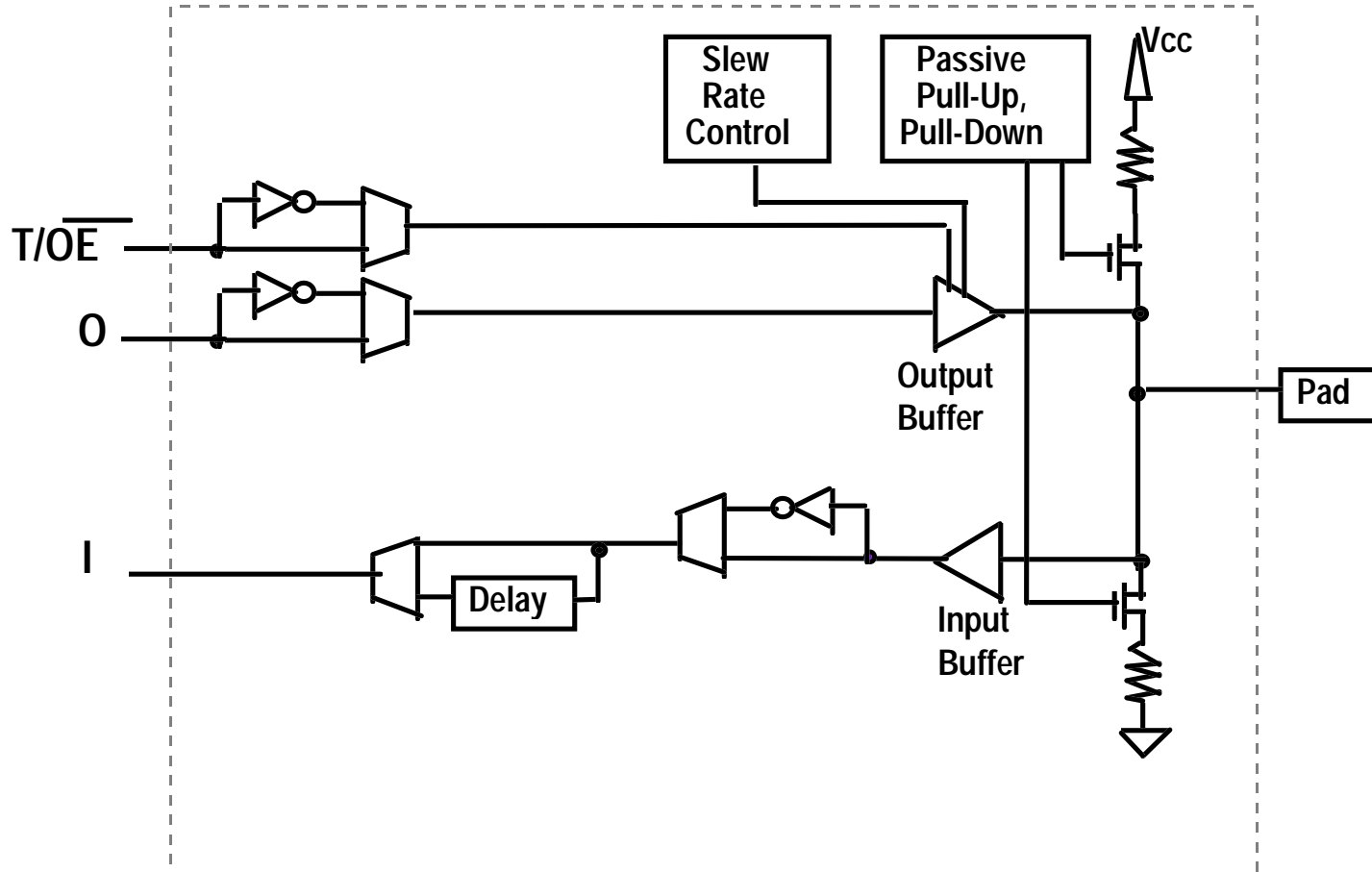
# Spartan Interconnect Resources



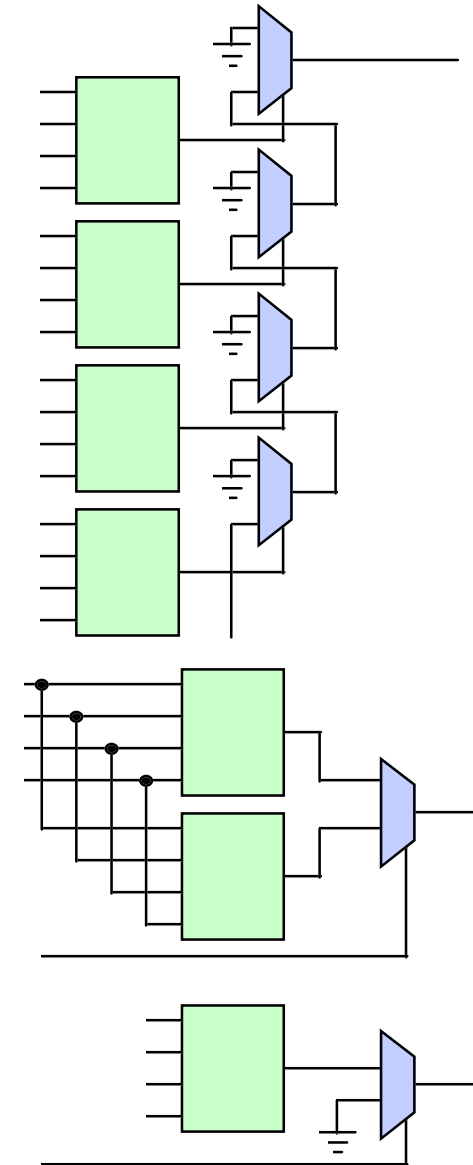
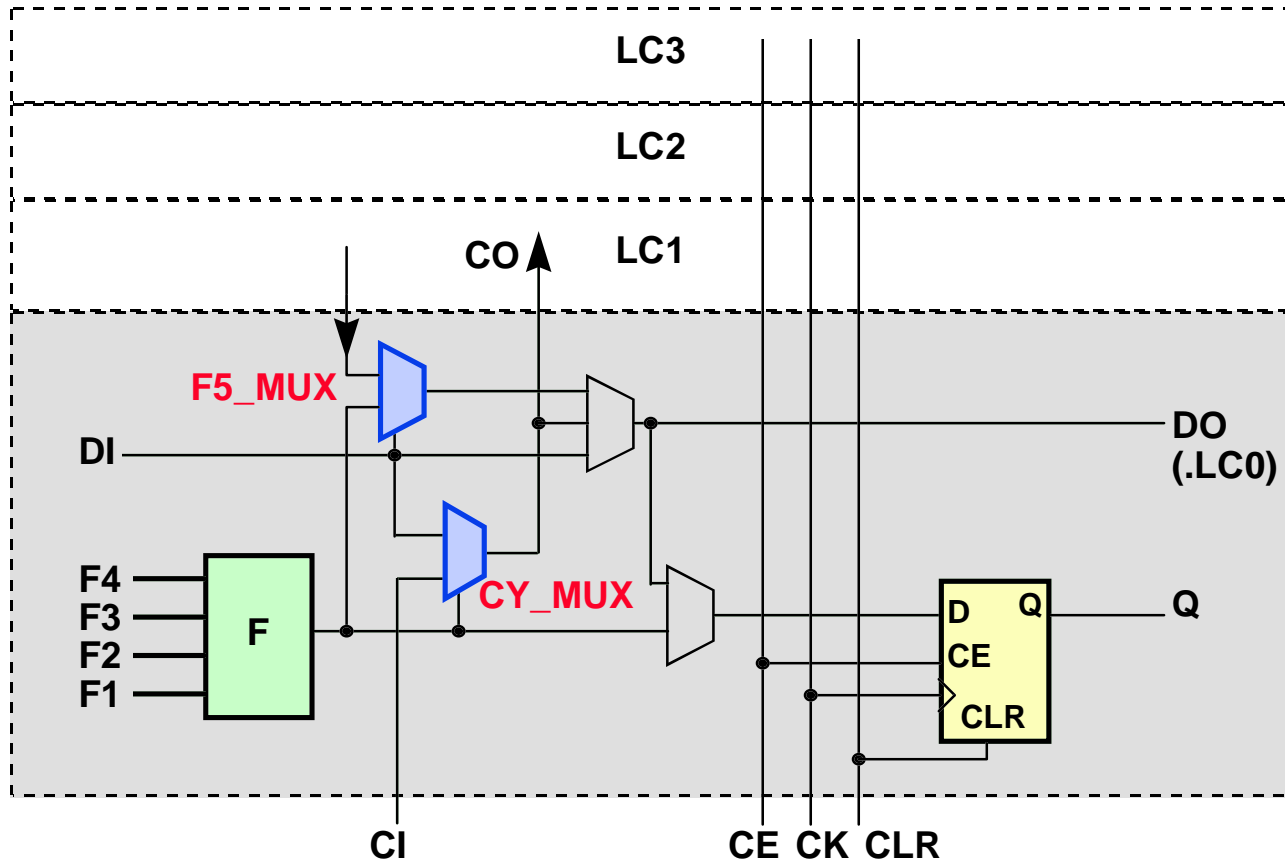
# XC5200 Families

<u>Part</u>	<u>Typical Gates</u>	<u>CLB Matrix</u>	<u>CLBs</u>	<u>Flip-Flops</u>	<u>IOBs</u>
5202	2,000-3,000	8x8	64	256	84
5204	4,000-6,000	10x12	120	480	124
5206	6,000-10,000	14x14	196	784	148
5210	10,000-16,000	18x18	324	1,296	196
5215	15,000-23,000	22x22	484	1,936	244

# XC5200 IOB



# XC5200 CLB

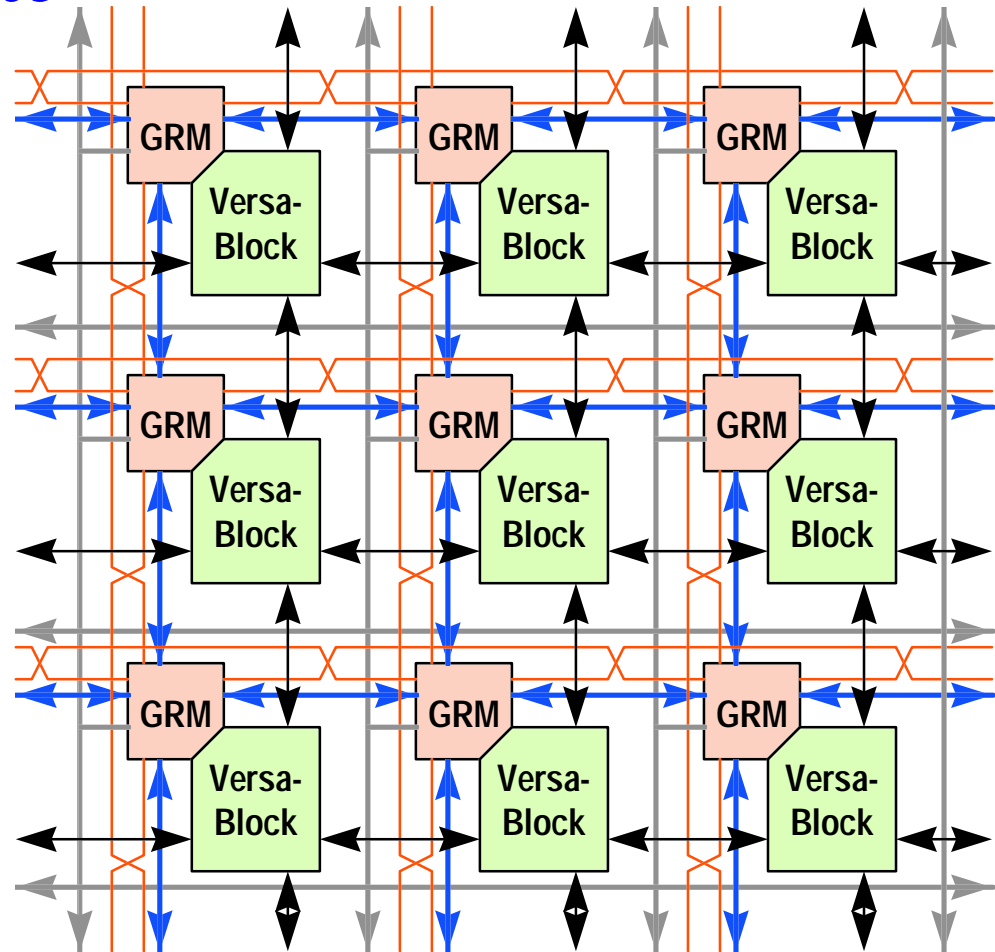


XC5200 Configurable Function Generator

# XC5200 Interconnect Resources

## ◆ Hierarchical routing resources

- Logic cell feedthrough path
- VersaBlock routing
  - Local interconnect matrix
  - Direct connects
- GRM: general routing matrix
  - Single-length lines
  - Double-length lines
  - Longlines
  - Global lines
- VersaRing I/O interface





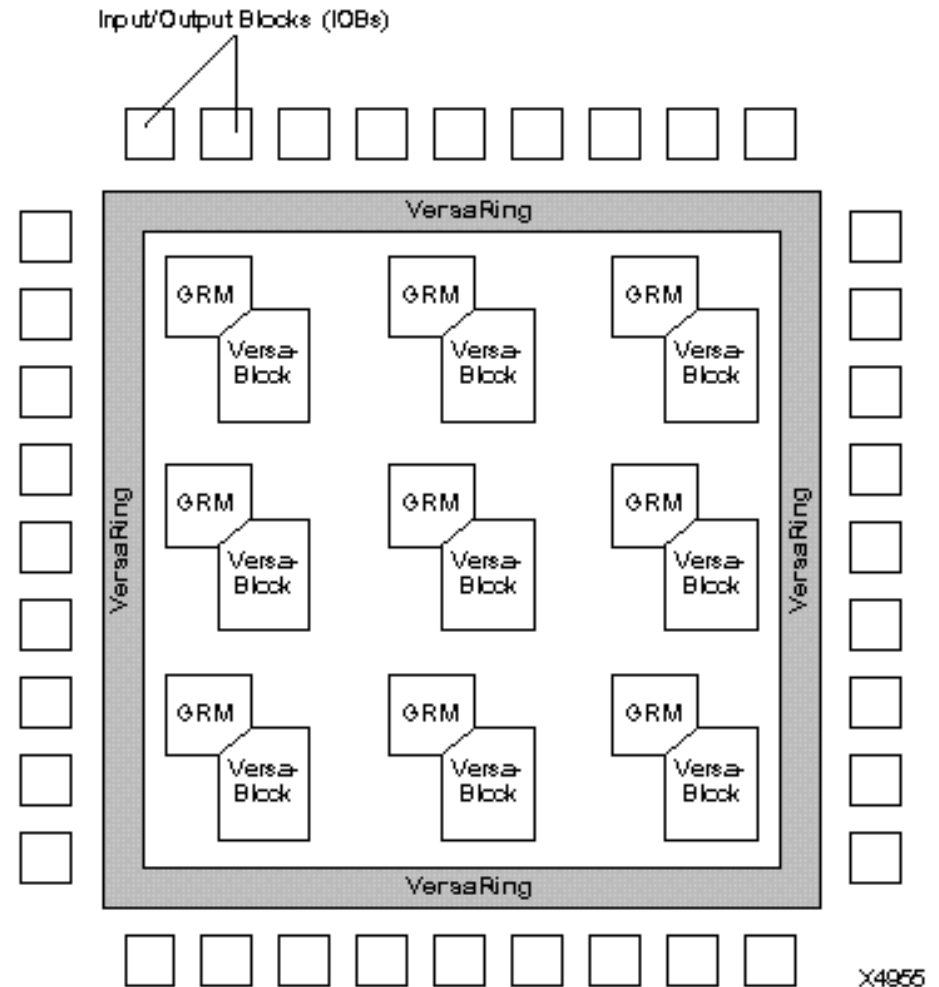
# XC5200 VersaBlock & VersaRing

## ◆ VersaBlock

- Abundant local routing plus versatile logic
- 5 independent inputs & 3 outputs to each LC
- Each LC contains a direct feedthrough path

## ◆ VersaRing I/O interface

- Abundant connections from each IOB to the nearest VersaBlock, in addition to Longline connections surrounding the device
- Increase pin-locking flexibility



# Xilinx FPGA Family Comparison

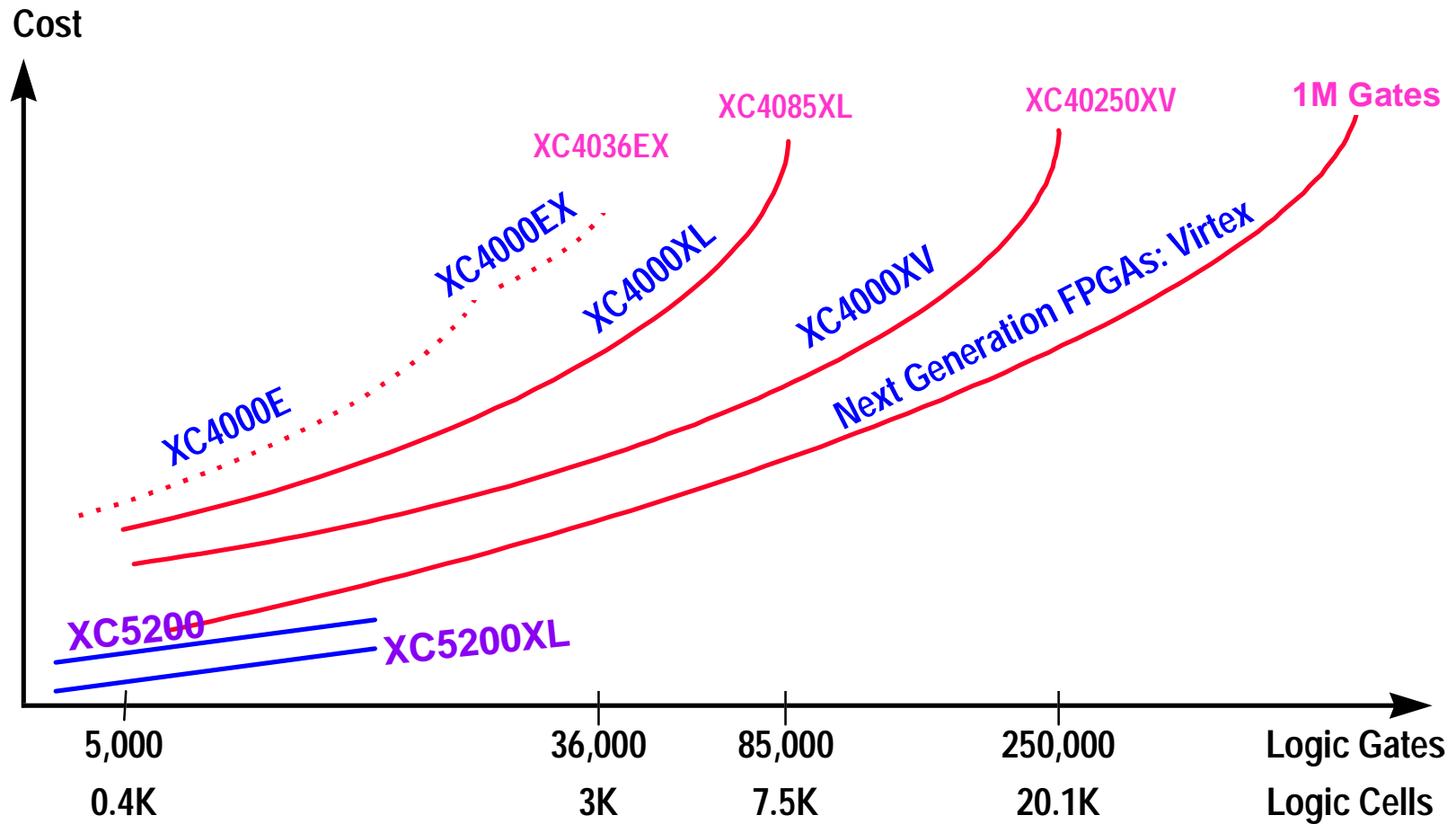
<u>Family</u>	<u>XC3100A</u>	<u>XC4000E/X</u>	<u>Spartan/XL</u>	<u>XC5200</u>
Key Feature	Speed	Speed	Low Cost	Low Cost
		Density	Density	Density
		RAM	RAM	
Speed	to 95 MHz	to 66-100 MHz	to 80 MHz	to 50 MHz
Density	1K-7.5K gates	3K-250K gates	2K-40K gates	2K-23K gates
I/O	64-176	80-448	80-224	84-244
Flip-flops	256-1320	360-18400	360-2016	256-1936
Features	2 global clocks	4-8 global clocks	8 global clocks	4 global clocks
	Internal buses	(same)	(same)	(same)
		Edge decode		Cascade
		Carry	Carry	Carry
		Scan	Scan	Scan
		RAM	RAM	

# The Roadmap

## ◆ Xilinx FPGA roadmap

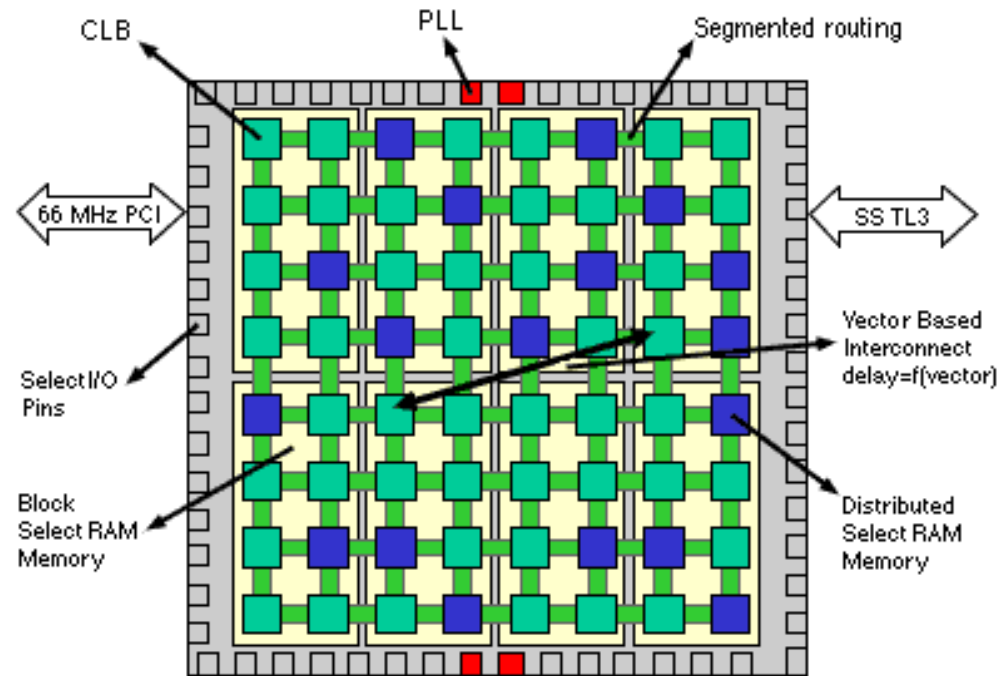
- High-density 2.5-V **XC4000XV** family (0.25um, 5 layer metal process, 2.5V)
  - Up to 500,000 gates or higher
  - 5-V, 3.3-V compatible
- High-density 2.5-V **Virtex** family (0.25um, 5 layer metal process, 2.5V)
  - Alternative to system-level ASIC design
  - 100+ MHz system performance
  - Up to 1,000,000 system gates
  - Highly flexible I/O buffers, multiple PLLs, dedicated 4-k bit, dual-ported SRAM blocks
- Low-cost **XC5200XL** family (0.35um, 3.3V)
  - Up to 15,000 gates or higher
  - 5-V compatible

# Density & Cost Roadmap



# Appendix: Virtex Architecture

## Functional Block Diagram



The New Xilinx Virtex architecture features Xilinx SelectRAM+memory of distributed, block, and high-speed access to external RAM via the SSTL3 standard, phase locked loops (PLL), Xilinx Select I/O pins, Xilinx segmented-routing and vector based interconnect

# HDL Design Flow & Tools

## ◆ FPGA Design Flow

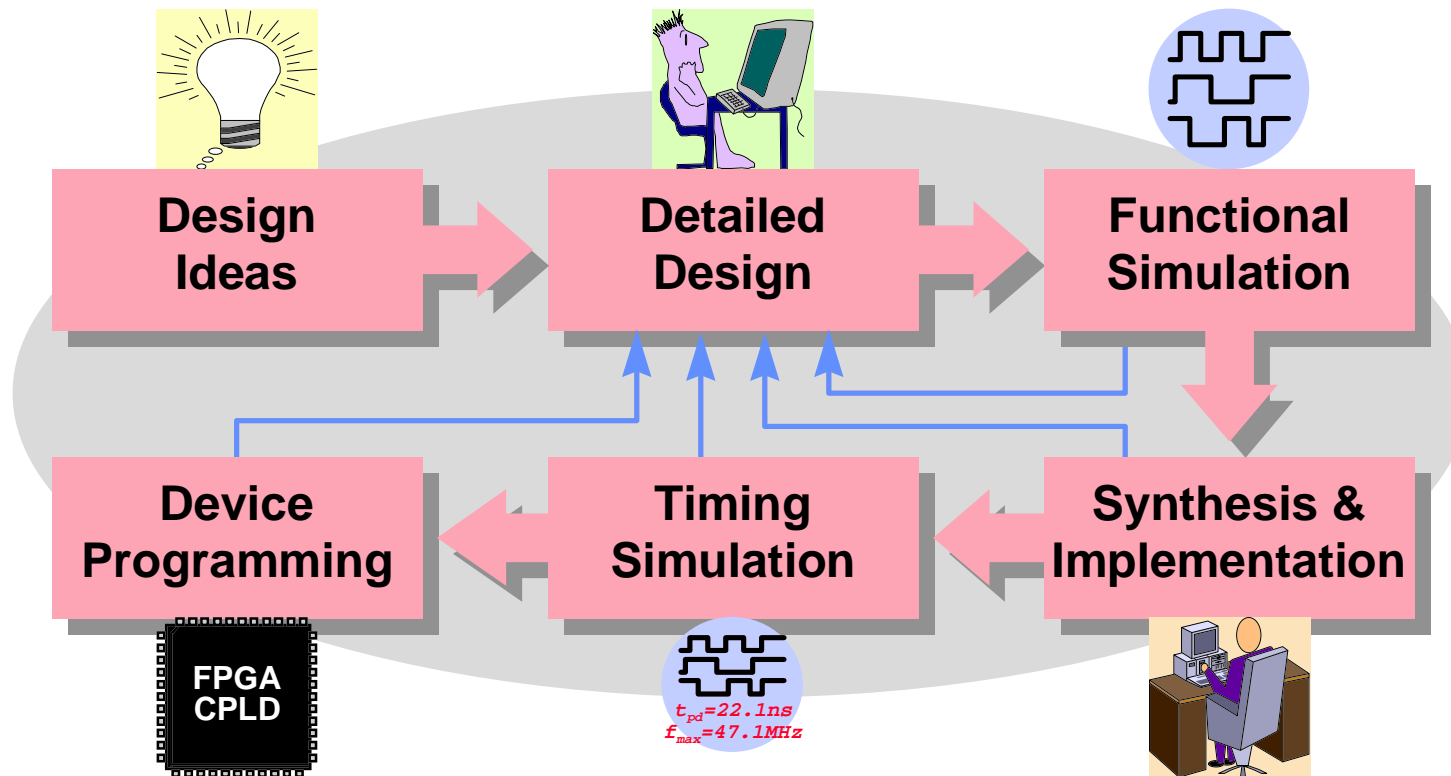
- Design Ideas
- Detailed Design
- Functional Simulation
- Synthesis & Implementation
- Timing Simulation
- Device Programming

## ◆ Altera HDL Design Flow & Tools

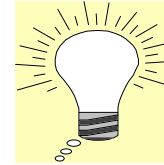
## ◆ Xilinx HDL Design Flow & Tools

## ◆ A Simple Tutorial

# FPGA/CPLD Design Flow



# Design Ideas

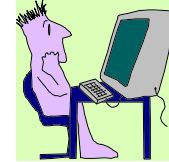


## ◆ What are the main design considerations?

- Design feasibility?
- Design spec?
- Cost?
- FPGA/CPLD or ASIC?
- Which FPGA/CPLD vendor?
- Which device family?
- Development time?



# Detailed Design



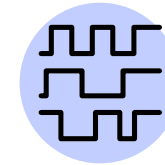
## ◆ Choose the design entry method

- Schematic
  - Gate level design
  - Intuitive & easy to debug
- HDL (Hardware Description Language), e.g. Verilog & VHDL
  - Descriptive & portable
  - Easy to modify
- Mixed HDL & schematic

## ◆ Manage the design hierarchy

- Design partitioning
  - Chip partitioning
  - Logic partitioning
- Use vendor-supplied libraries or parameterized libraries to reduce design time
- Create & manage user-created libraries (circuits)

# Functional Simulation



## ◆ Preparation for simulation

- Generate simulation patterns
  - Waveform entry
  - HDL testbench
- Generate simulation netlist

## ◆ Functional simulation

- To verify the functionality of your design only

## ◆ Simulation results

- Waveform display
- Text output

## ◆ Challenge

- Sufficient & efficient test patterns

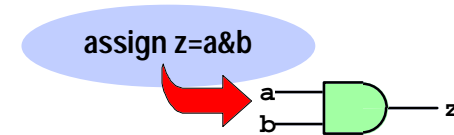
# HDL Synthesis

## ◆ Synthesis = Translation + Optimization

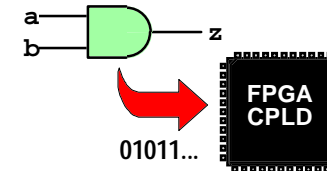
- Translate HDL design files into gate-level netlist
- Optimize according to your design constraints
  - Area constraints
  - Timing constraints
  - Power constraints
  - ...

## ◆ Main challenges

- Learn synthesizable coding style
- Write correct & synthesizable HDL design files
- Specify reasonable design constraints
- Use HDL synthesis tool efficiently



# Design Implementation



## ◆ Implementation flow

- Netlist merging, flattening, data base building
- Design rule checking
- Logic optimization
- Block mapping & placement
- Net routing
- Configuration bitstream generation

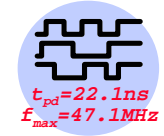
## ◆ Implementation results

- Design error or warnings
- Device utilization
- Timing reports

## ◆ Challenge

- How to reach high performance & high utilization implementation?

# Timing Analysis & Simulation



## ◆ Timing analysis

- Timing analysis is static, i.e., independent of input & output patterns
- To examine the timing constraints
- To show the detailed timing paths
- Can find the critical path

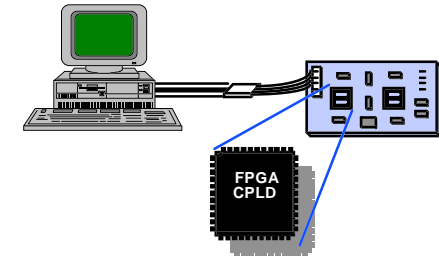
## ◆ Timing simulation

- To verify both the functionality & timing of the design

# Device Programming

## ◆ Choose the appropriate configuration scheme

- SRAM-based FPGA/CPLD devices
  - Downloading the bitstream via a download cable
  - Programming onto a non-volatile memory device & attaching it on the circuit board
- OTP, EPROM, EEPROM or Flash-based FPGA/CPLD devices
  - Using hardware programmer
  - ISP



## ◆ Finish the board design

## ◆ Program the device

## ◆ Challenge

- Board design
- System considerations

# Our Focus: HDL Design Flow

## ◆ Why HDL?

- Can express digital systems in behavior or structure domain, shortening the design time
- Can support all level of abstraction, including algorithm, RTL, gate and switch level
- Both VHDL & Verilog are formal hardware description languages, thus portable

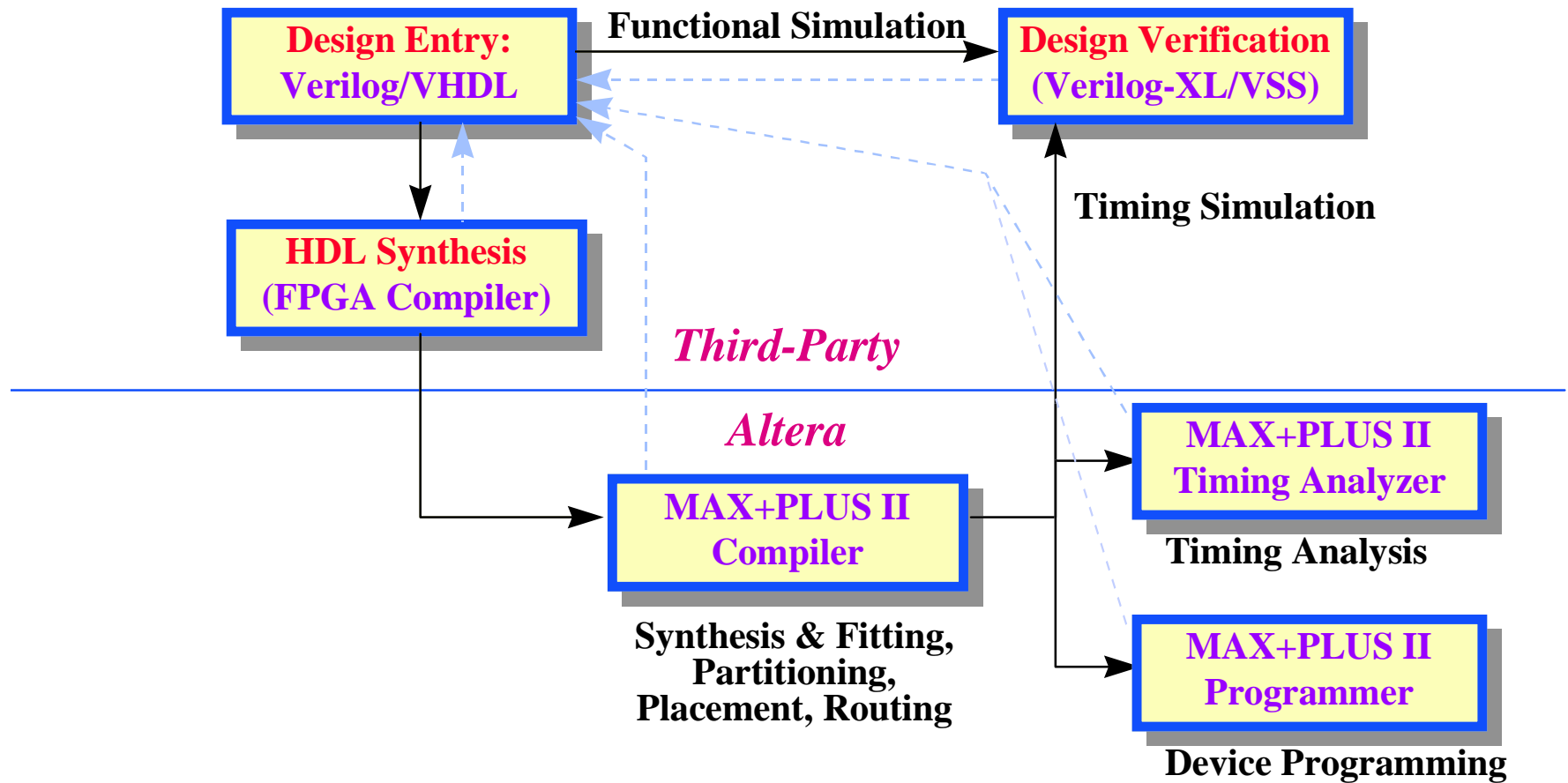
## ◆ Typical HDL design flow

- Use VHDL or Verilog to express digital systems
  - VHDL or Verilog simulation tool is required to simulate your project
- Use high-level synthesis tool to obtain structural level design
- Then use FPGA placement & routing tools to obtain physical FPGA netlist

## ◆ We assume you are familiar with VHDL or Verilog...

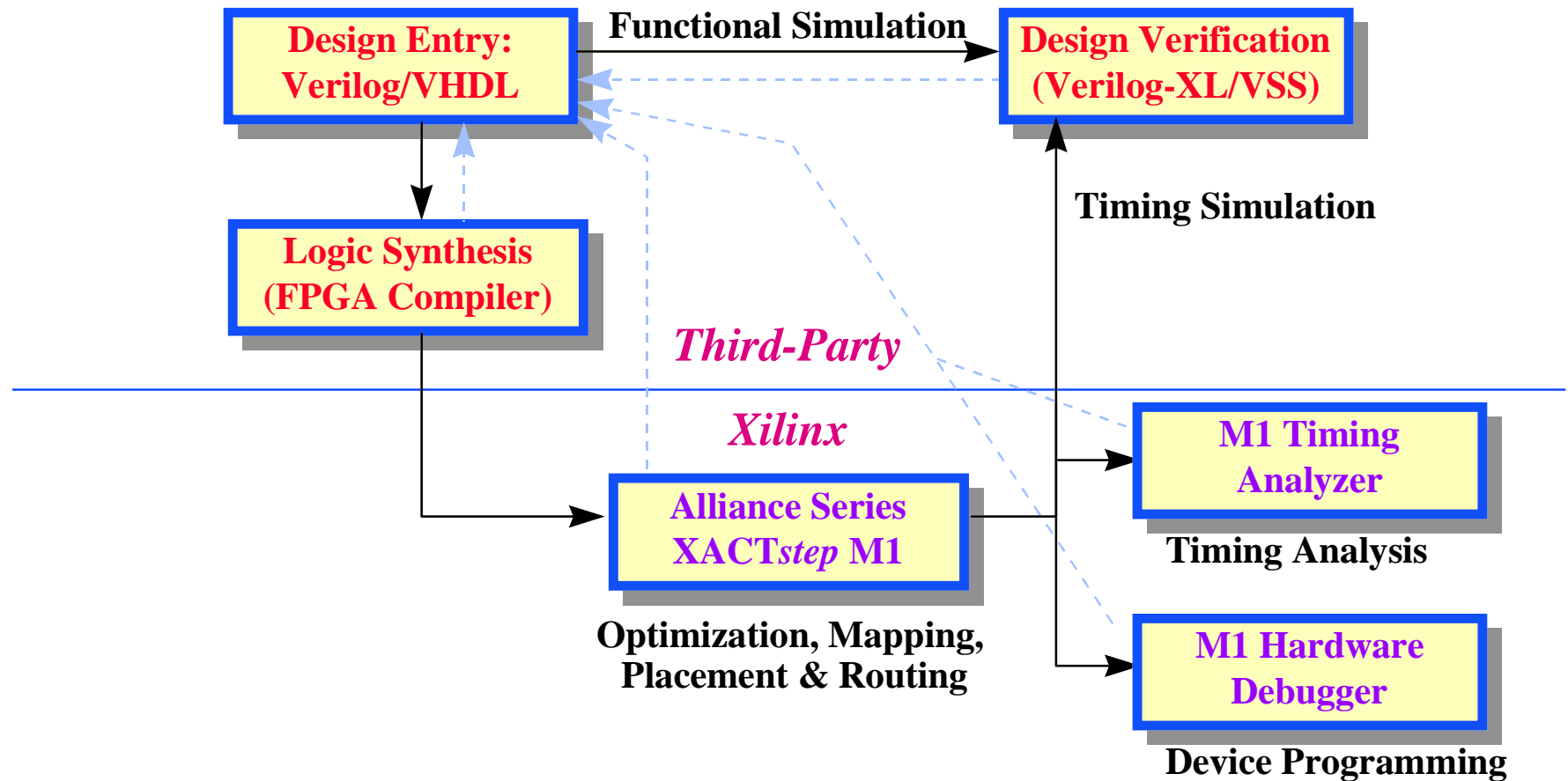
- In this course, we'll emphasize on FPGA HDL coding techniques for synthesis
  - It's the key issue to reduce area and achieve high performance for your project
- We assume you know how to use VHDL or Verilog simulator too

# Altera HDL Design Flow





# Xilinx HDL Design Flow



# A Simple Tutorial

- ◆ Prepare Verilog or VHDL design files
- ◆ Perform Verilog or VHDL functional simulation
- ◆ Use Synopsys FPGA Compiler to synthesize your Verilog or VHDL codes
- ◆ Implement the design into the specified FPGA part
- ◆ Perform Verilog or VHDL timing analysis & simulation

# Design Entry

## ◆ Write HDL design files

- Must learn synthesizable RTL Verilog or VHDL coding style for the synthesis tool
- Tool: text editor
  - xedit, textedit, vi, joe, ...

```
module converter(i3,i2,i1,i0,a,b,c,d,e,f,g);
  input i3, i2, i1, i0 ;
  output a, b, c, d, e, f, g;
  reg a,b,c,d,e,f,g;

  always @(i3 or i2 or i1 or i0) begin
    case({i3,i2,i1,i0})
      4'b0000: {a,b,c,d,e,f,g} = 7'b11111110;
      4'b0001: {a,b,c,d,e,f,g} = 7'b11000000;
      4'b0010: {a,b,c,d,e,f,g} = 7'b10110111;
      4'b0011: {a,b,c,d,e,f,g} = 7'b11100111;
      4'b0100: {a,b,c,d,e,f,g} = 7'b11001011;
      4'b0101: {a,b,c,d,e,f,g} = 7'b01101111;
      4'b0110: {a,b,c,d,e,f,g} = 7'b01111111;
      4'b0111: {a,b,c,d,e,f,g} = 7'b11000101;
      4'b1000: {a,b,c,d,e,f,g} = 7'b11111111;
      4'b1001: {a,b,c,d,e,f,g} = 7'b11101111;
      4'b1010: {a,b,c,d,e,f,g} = 7'b11011111;
      4'b1011: {a,b,c,d,e,f,g} = 7'b01111011;
      4'b1100: {a,b,c,d,e,f,g} = 7'b00111110;
      4'b1101: {a,b,c,d,e,f,g} = 7'b11110011;
      4'b1110: {a,b,c,d,e,f,g} = 7'b00111111;
      4'b1111: {a,b,c,d,e,f,g} = 7'b00011111;
    endcase
  end
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity converter is
  port ( i3, i2, i1, i0: in STD_LOGIC;
         a, b, c, d, e, f, g: out STD_LOGIC);
end converter;

architecture case_description of converter is
begin
  P1: process(i3, i2, i1, i0)
    variable tmp_in: STD_LOGIC_VECTOR(3 downto 0);
  begin
    tmp_in := i3 & i2 & i1 & i0;
    case tmp_in is
      when "0000" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11111110");
      when "0001" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11000000");
      when "0010" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("10110111");
      when "0011" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11100111");
      when "0100" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11001011");
      when "0101" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("01101111");
      when "0110" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("01111111");
      when "0111" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11000101");
      when "1000" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11111111");
      when "1001" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11101111");
      when "1010" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11011111");
      when "1011" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("01111011");
      when "1100" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("00111110");
      when "1101" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("11110011");
      when "1110" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("00111111");
      when "1111" => (a,b,c,d,e,f,g) <= STD_LOGIC_VECTOR("00011111");
      when others => (a,b,c,d,e,f,g) <= STD_LOGIC_vector("00000000");
    end case;
  end process P1;
end case_description;
```

# HDL Functional Simulation

- ◆ Write HDL testbench files
- ◆ Prepare technology-dependent simulation model, if necessary
- ◆ Verilog functional simulation
  - Tool: Verilog simulator
    - ☞ Cadence Verilog-XL
    - Viewlogic VCS
- ◆ VHDL functional simulation
  - Tool: VHDL simulator
    - ☞ Synopsys VSS
    - Viewlogic SpeedWave
    - Cadence LeapFrog

# Verilog Functional Simulation

The image displays two windows from a Verilog simulation environment. The left window, titled 'STCHEN', shows the compilation process and simulation results. The right window, titled 'Cwaves 2.2', shows a timing diagram for a circuit named './converter.shm'.

**Terminal Output (STCHEN):**

```
Compiling source file "case1_tl
Highest level modules:
test

*WARNING* Global timer will no
input: 0000 output: 1111110
input: 0001 output: 1100000
input: 0010 output: 1011011
input: 0011 output: 1110011
input: 0100 output: 1100101
input: 0101 output: 0110111
input: 0110 output: 0111111
input: 0111 output: 1100010
input: 1000 output: 1111111
input: 1001 output: 1110111
input: 1010 output: 1101111
input: 1011 output: 0111101
input: 1100 output: 0011110
input: 1101 output: 1111001
input: 1110 output: 0011111
input: 1111 output: 0001111
251 simulation events
CPU time: 0.4 secs to compile
End of VERILOG-XL 2.2.1 Jul
cic02/stchen>
```

**Waveform Viewer (Cwaves 2.2):**

The waveform viewer shows a timing diagram for the circuit './converter.shm'. The time axis ranges from 0 to 300 seconds. The signals are:

- test.i3: 0
- test.i2: 0
- test.i1: 0
- test.i0: 0
- test.a: St1
- test.b: St1
- test.c: St1
- test.d: St1
- test.e: St1
- test.f: St1
- test.g: St0

The signals test.a through test.g show a complex pattern of transitions, while test.i0 through test.i3 are constant at 0. The waveform viewer includes a menu (File, Edit, View, Options, Help) and a toolbar with various icons for zooming and navigation. The status bar at the bottom shows 'Time in sec', 'Cursor = 0', and 'Delta = 0'.

# VHDL Functional Simulation

The image displays two windows from the Synopsys VHDL simulation environment. The top window, 'Synopsys VHDL Debugger (Vhdlbvx)', shows a VHDL code snippet with an assertion failure at 320.0 NS. The code includes a loop for i from 0 to 19, with assignments for temp, i3, i2, i1, and i0, and a wait for 20 ns. The assertion failure is indicated by a red arrow pointing to 'assert FALSE' on line 36. The bottom window, 'Synopsys Waveform Viewer - TEST.cic02.13346.ow:0 - [Untitled]', shows a timing diagram with a time axis from 0 to 3000 ns. The signal list on the left includes /TEST/I3, /TEST/I2, /TEST/I1, /TEST/I0, /TEST/A, /TEST/B, /TEST/C, /TEST/D, /TEST/E, /TEST/F, and /TEST/G. The waveform shows various digital signals over time. A red vertical line at 320.0 ns indicates the point of failure. The status bar at the bottom shows 'Ready' and 'Time = 3200 Wif=11 Wfc=11 Sel=0'.

```
28 for i in 0 to 19
29   temp := CONV_S
30   i3 <= temp(3);
31   i2 <= temp(2);
32   i1 <= temp(1);
33   i0 <= temp(0);
34   wait for 20 ns;
35 end loop;
36 →assert FALSE
37   report "TEST
38 end process;
39 end test_converter;
```

CWR : /TEST/\_P0  
Time : 320.0 NS

Stop at Clear Trace Event BK

```
# run
320.0 NS
Assertion FAILURE at 320.0 NS
"TEST COMPLETE!"
#
```

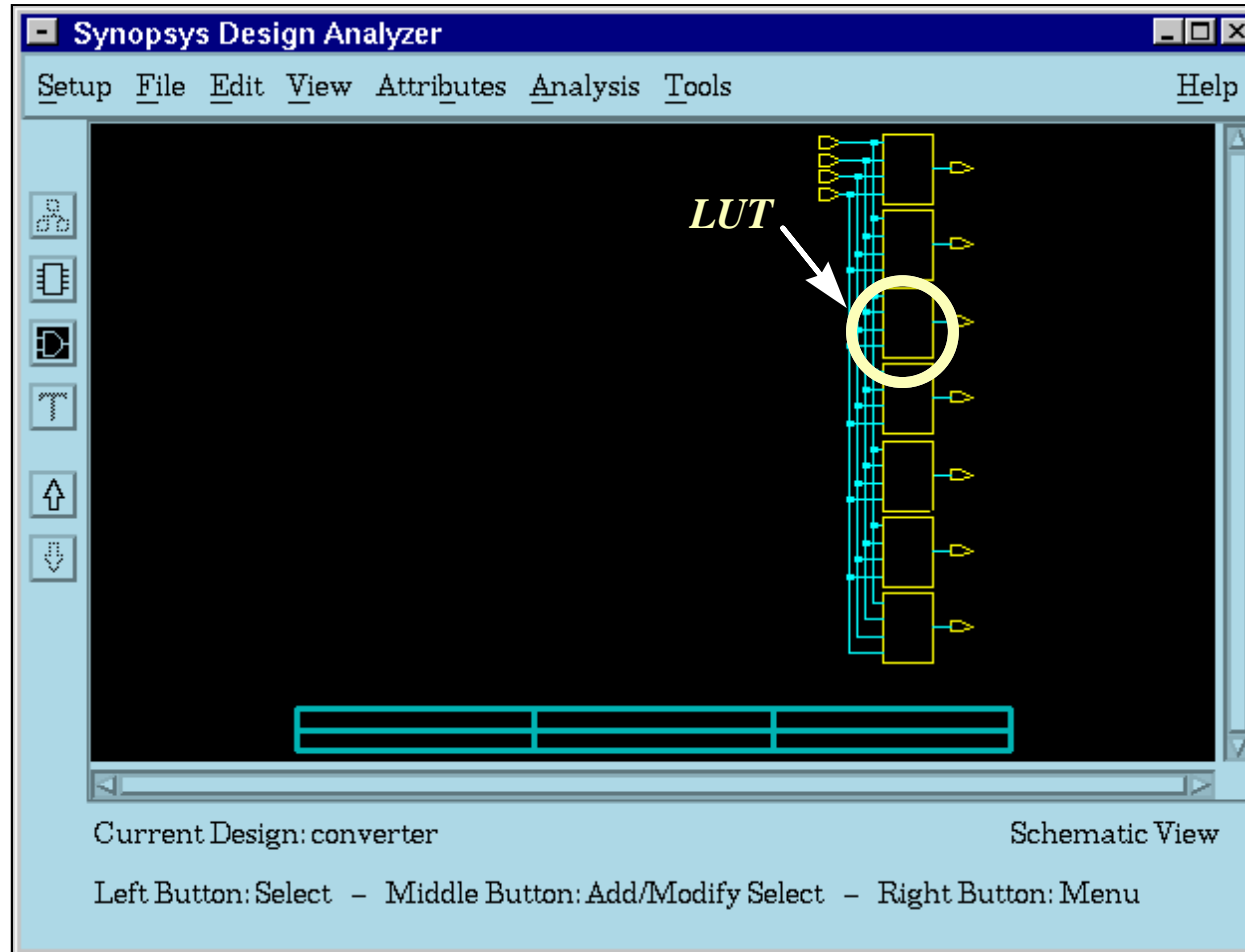
Signal	Value at 320.0 NS
/TEST/I3	1
/TEST/I2	1
/TEST/I1	1
/TEST/I0	1
/TEST/A	0
/TEST/B	0
/TEST/C	1
/TEST/D	1
/TEST/E	1
/TEST/F	1
/TEST/G	1

Ready Time = 3200 Wif=11 Wfc=11 Sel=0

# HDL Synthesis

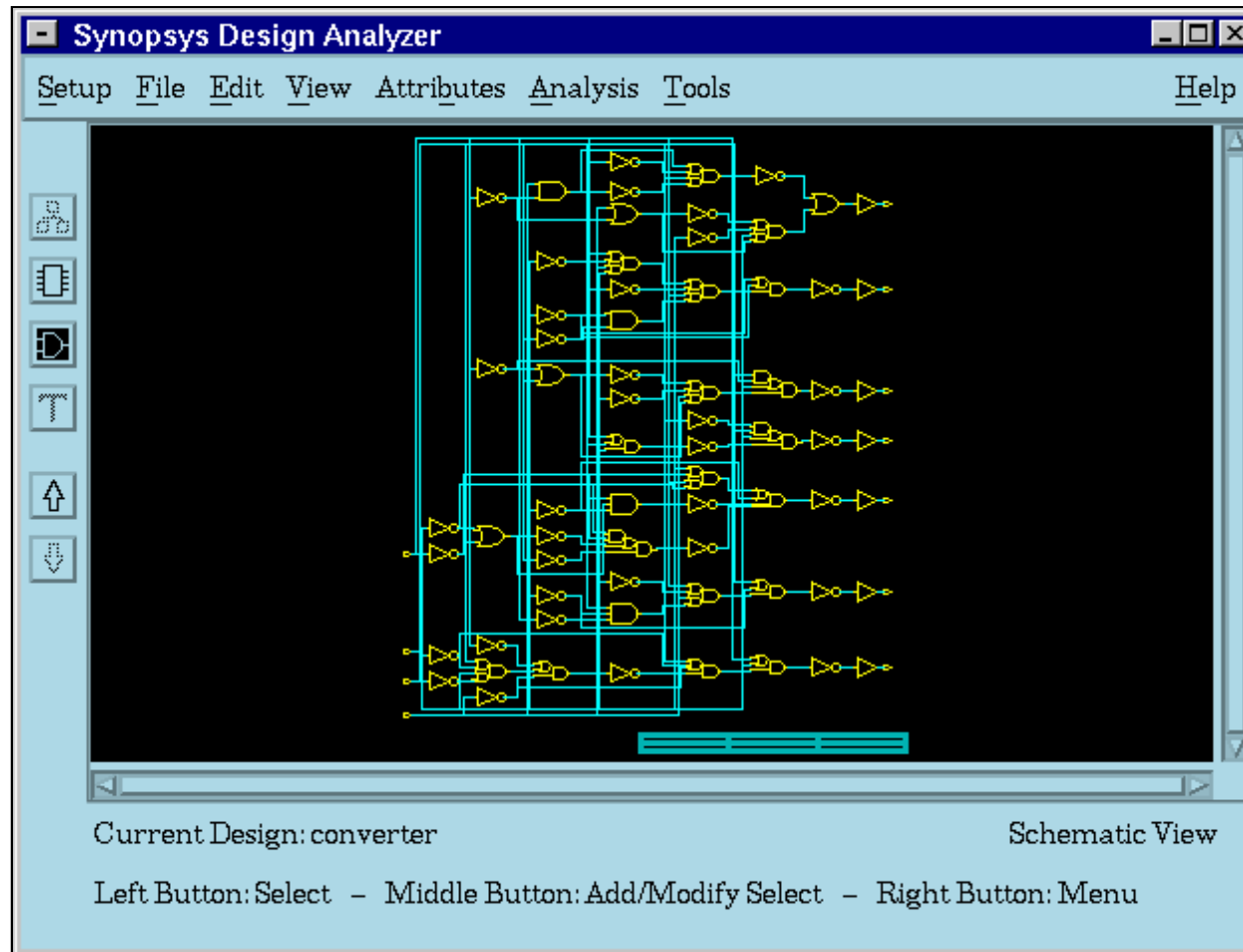
- ◆ Prepare synthesis library
- ◆ Transfer HDL design file into gate-level netlist
  - Tool: HDL synthesis software
    - ☞ Synopsys: Design Analyzer, HDL/VHDL Compiler & FPGA Compiler
      - Viewlogic ViewSynthesis (for VHDL only)
      - Cadence Synergy
  - Generate EDIF netlist file (\*.edf) for Altera design
  - Generate XNF netlist files (\*.sxnf) for Xilinx design

# LUT Optimization for Altera

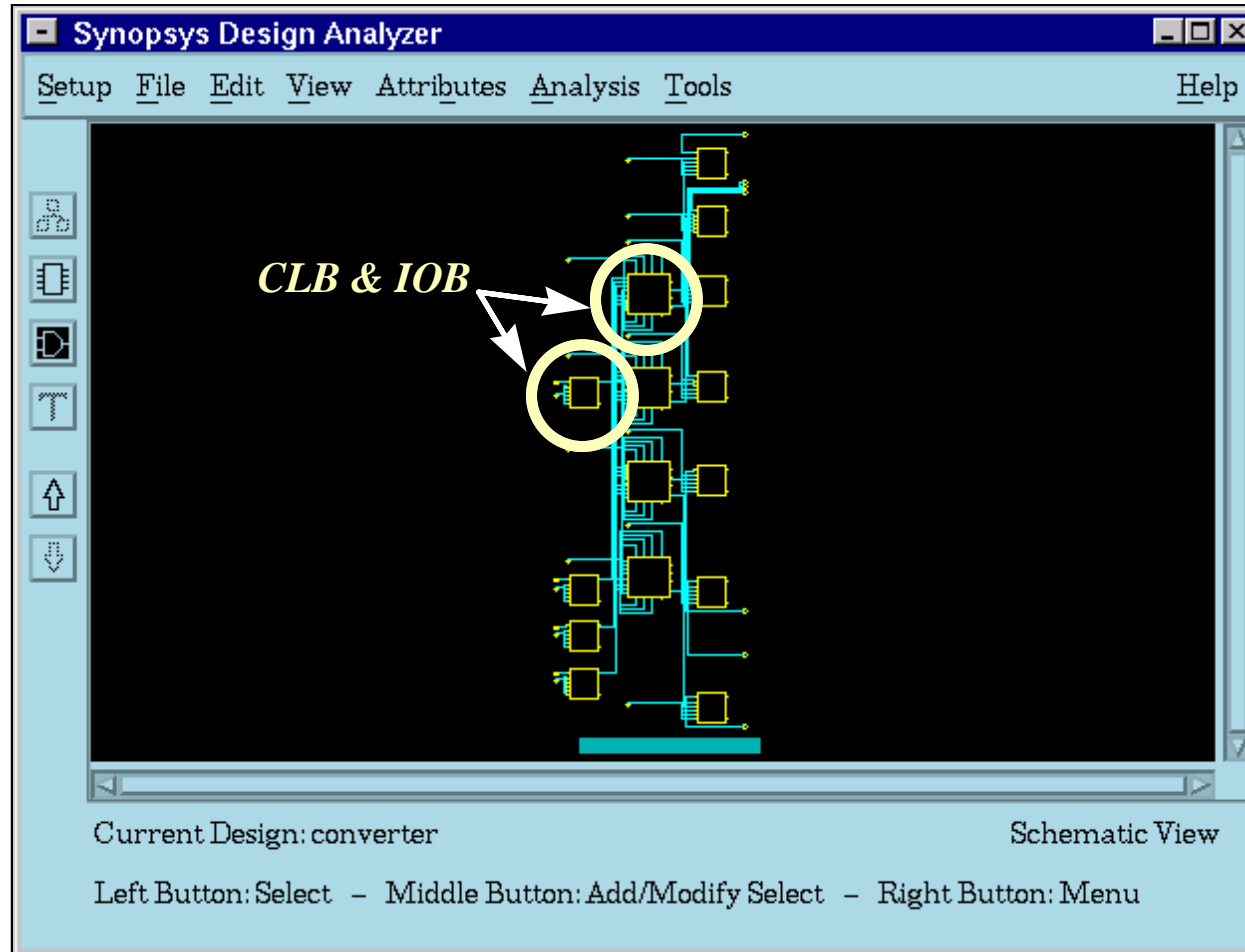




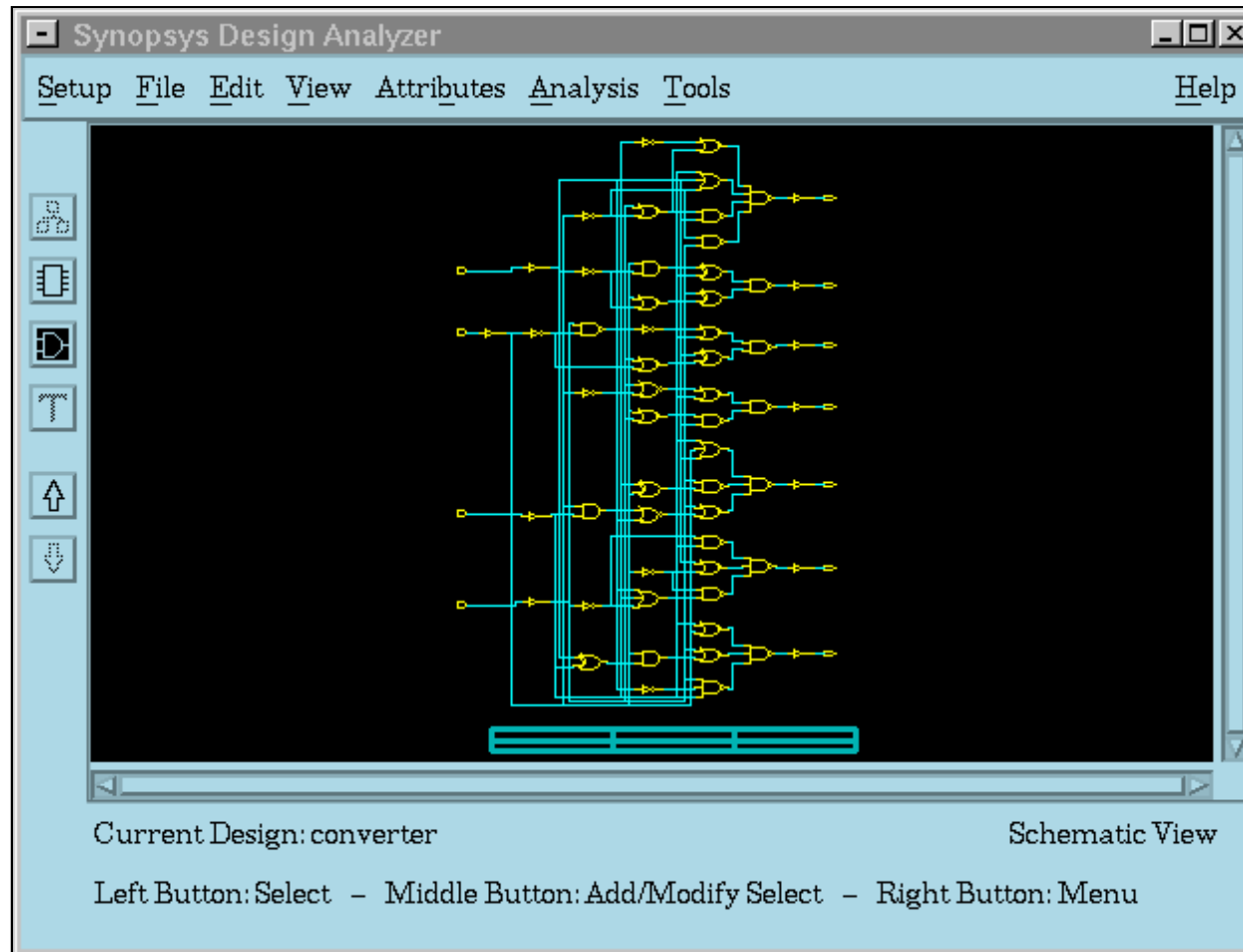
# Gate-Level Netlist for Altera



# LUT Optimization for Xilinx



# Gate-Level Netlist for Xilinx

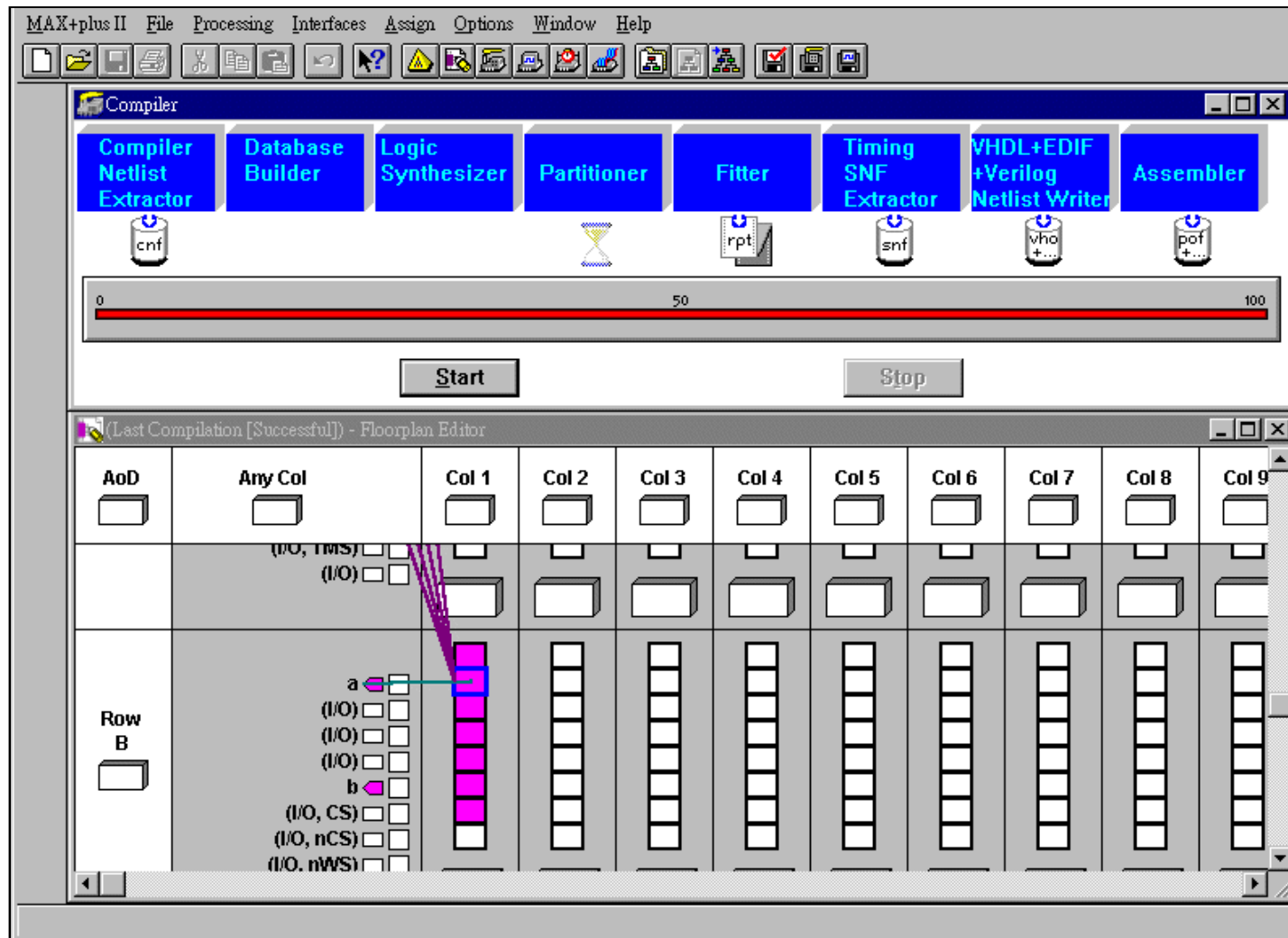


# FPGA Implementation

## ◆ Gate-level netlist -> configuration bitstream & timing information

- Altera development tool: Altera MAX+PLUS II software
  - MAX+PLUS II Compiler
  - MAX+PLUS II Floorplan Editor
- Xilinx development tool: Xilinx XACT *step* M1 software
  - Xilinx Design Manager
  - Flow Engine
  - EPIC Design Editor

# Altera Implementation



# Xilinx Implementation

Flow Engine – converter(ver1->rev1)

Flow View Setup Utilities Help

XC4000E Design Flow (rev1) Status: OK

Translate	Map	Place&Route	Timing	Configure
Completed	Completed	Completed	Completed	Completed

```
Saving ll file in "converter.ll".
Creating bit map...
Saving bit stream in "converter.bit".

xcpy converter.bit /users/cic/stchen/synopsys/xilinx00/simple1/converter.bit
xcpy converter.ll /users/cic/stchen/synopsys/xilinx00/simple1/converter.ll
```

XC4003E-3-PC84 None None

Script playback completed.  
Initialization completed.  
Copyright (c) 1995-1997 Xilinx, Inc. All rights reserved.  
EPIC M1.3.7 - ready for input.

# Timing Analysis

## ◆ Check critical timing path & clock rate

- Altera timing analysis tool: Altera MAX+PLUS II Timing Analyzer
- Xilinx timing analysis tool: Xilinx Timing Analyzer

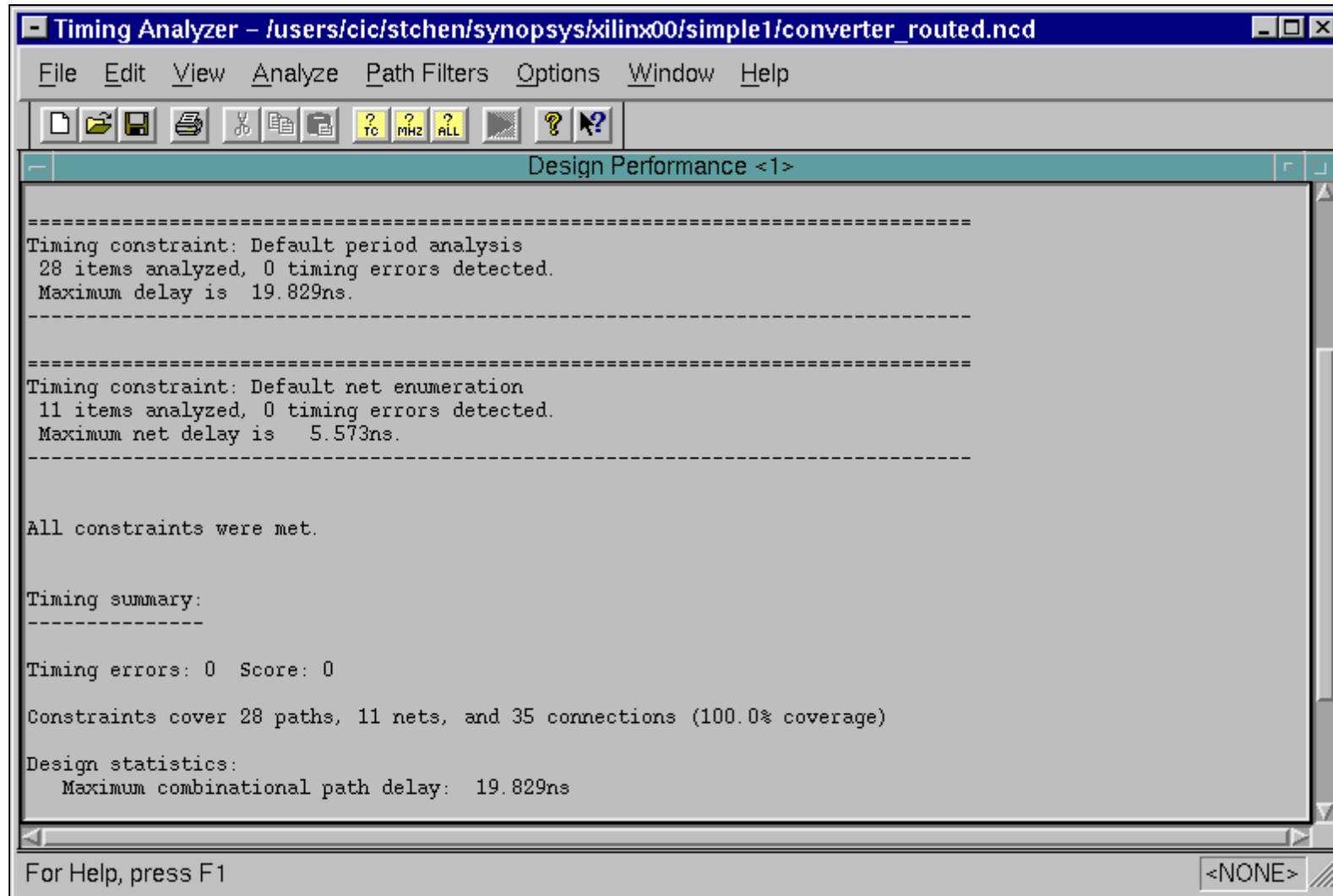
# Altera Timing Analysis

The screenshot displays the Altera MAX+plus II software interface. The main window is titled "Compiler" and contains several tool buttons: Compiler Netlist Extract, Database Builder, Logic Synthesizer, Partitioner, Fitter, Timing SNF, VHDL+EDIF+Verilog Writer, and Assembler. A "Timing Analyzer" window is open in the foreground, displaying a "Delay Matrix" for a destination. The matrix shows delay values in nanoseconds (ns) for sources i0, i1, i2, and i3 across destinations a, b, c, d, e, f, and g. Below the matrix is a progress bar and three buttons: Start, Stop, and List Paths. The background shows a partial view of a logic device configuration grid with columns labeled Col 7, Col 8, and Col 9.

		Destination						
		a	b	c	d	e	f	g
Source	i0	14.2ns	14.2ns	14.5ns	14.5ns	14.5ns	14.5ns	14.5ns
	i1	14.2ns	14.2ns	14.5ns	14.5ns	14.5ns	14.5ns	14.5ns
	i2	14.2ns	14.2ns	14.5ns	14.5ns	14.5ns	14.5ns	14.5ns
	i3	13.3ns	13.3ns	13.6ns	13.6ns	13.6ns	13.6ns	13.6ns



# Xilinx Timing Analysis



The screenshot shows the Xilinx Timing Analyzer interface. The window title is "Timing Analyzer - /users/cic/stchen/synopsys/xilinx00/simple1/converter\_routed.ncd". The menu bar includes File, Edit, View, Analyze, Path Filters, Options, Window, and Help. The toolbar contains icons for file operations and analysis settings. The main display area, titled "Design Performance <1>", shows the following text:

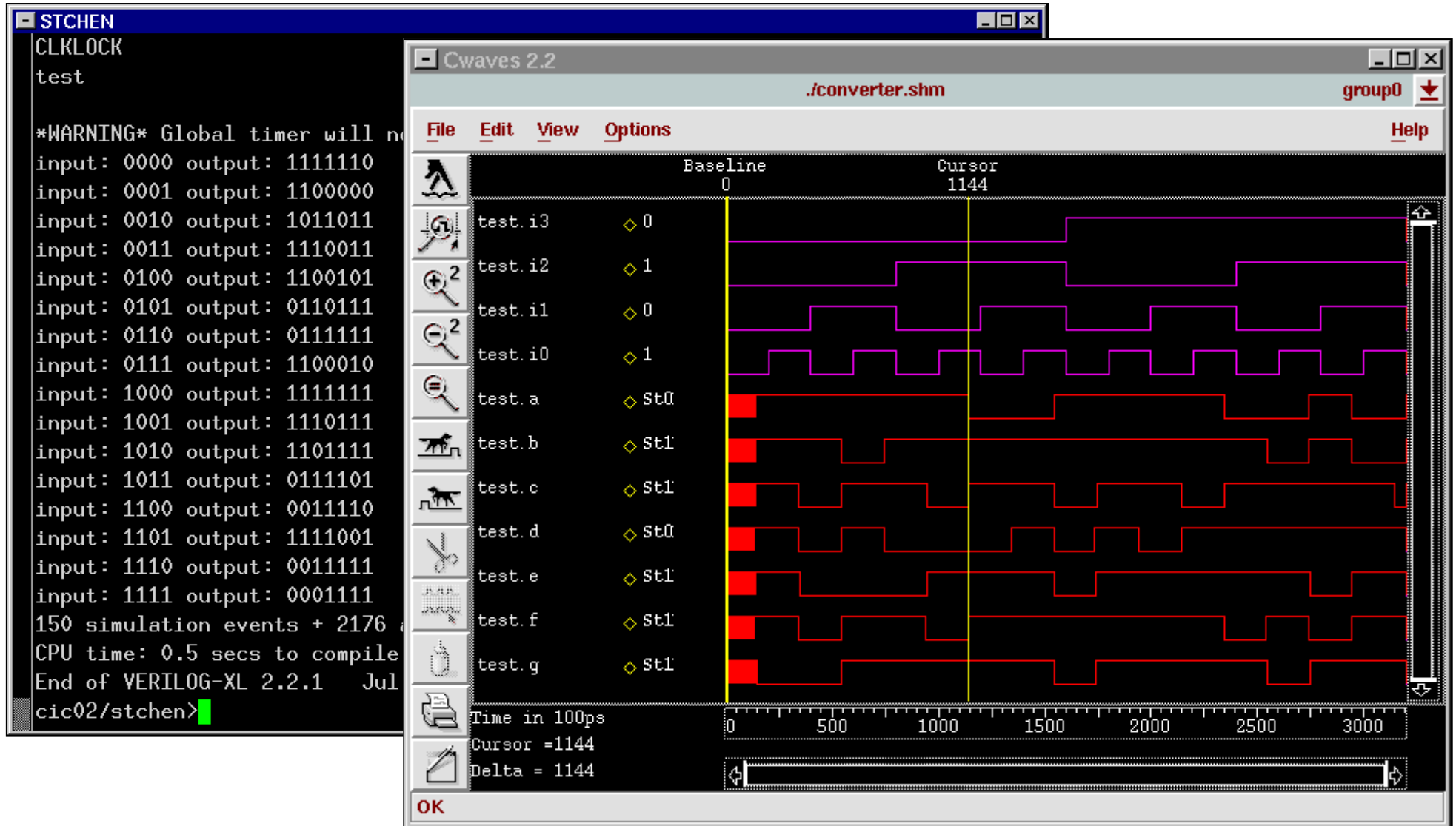
```
-----  
Timing constraint: Default period analysis  
28 items analyzed, 0 timing errors detected.  
Maximum delay is 19.829ns.  
-----  
-----  
Timing constraint: Default net enumeration  
11 items analyzed, 0 timing errors detected.  
Maximum net delay is 5.573ns.  
-----  
  
All constraints were met.  
  
Timing summary:  
-----  
Timing errors: 0 Score: 0  
Constraints cover 28 paths, 11 nets, and 35 connections (100.0% coverage)  
  
Design statistics:  
Maximum combinational path delay: 19.829ns
```

At the bottom of the window, it says "For Help, press F1" and a button labeled "<NONE>".

# Timing Simulation

- ◆ **Generate timing HDL files and delay back-annotation files**
  - Altera tool: MAX+PLUS II Compiler
  - Xilinx tool: ngdanno, ngd2vhd, ngd2ver utilities
- ◆ **Prepare testbench files**
- ◆ **Prepare technology-dependent simulation model, if necessary**
- ◆ **Verilog timing simulation**
  - Tool: Verilog simulator
- ◆ **VHDL timing simulation**
  - Tool: VHDL simulator

# Verilog Timing Simulation



# VHDL Timing Simulation

The image displays two windows from the Synopsys VHDL Debugger. The left window, titled "Synopsys VHDL Debugger (Vhdlb)", shows VHDL code for a testbench. The code includes a loop from 0 to 15, variable assignments, and an assertion at line 34: `assert FALSE`. The current cursor position is at line 34, and the time is 320000000 FS. The right window, titled "Synopsys Waveform Viewer - TEST.cic02.14602.ow:0 - [Untitled]", shows a timing diagram for signals /TEST/I3 through /TEST/G. A red vertical line marks the time 320000000 FS, where an assertion failure occurred. The waveform shows various digital signals, with some signals marked with 'U' (unknown) at the time of the failure.

**Synopsys VHDL Debugger (Vhdlb) Code:**

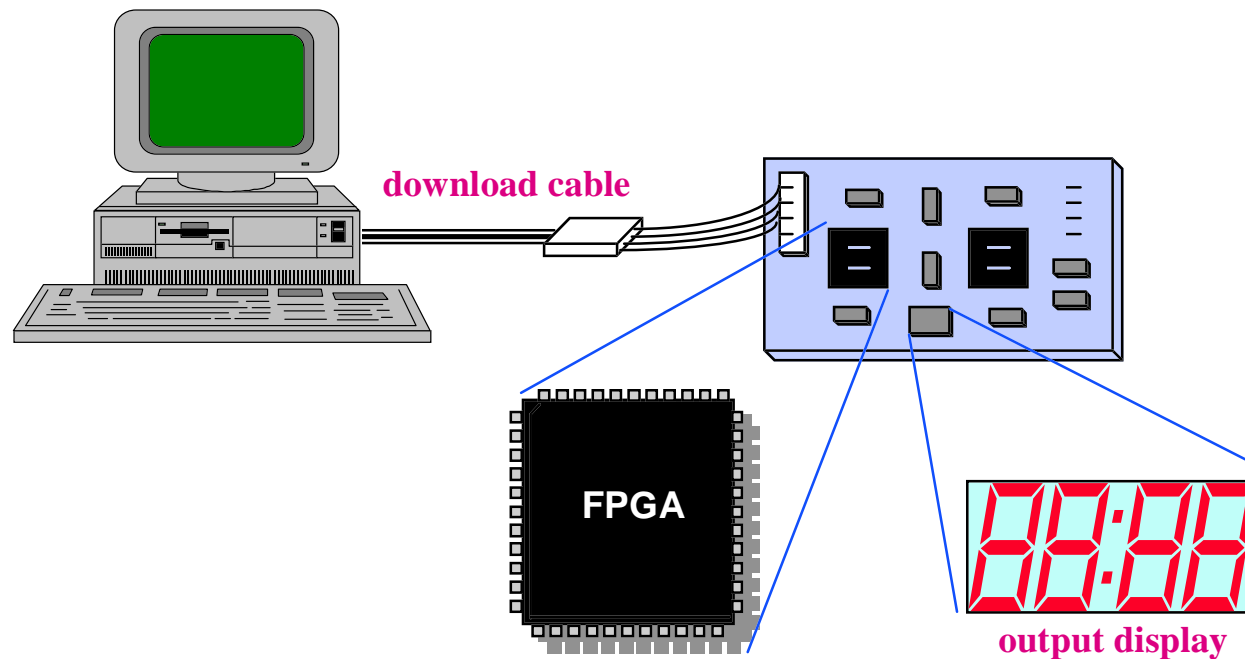
```
26   for i in 0 to 15 loop
27     temp := CONV_STD_LOGIC_V
28     i3 <= temp(3);
29     i2 <= temp(2);
30     i1 <= temp(1);
31     i0 <= temp(0);
32     wait for 20 ns;
33   end loop;
34   assert FALSE
35     report "TEST COMPLETE!"
36   end process;
37 end test_converter;
38
39 configuration CFG_converter_tb
40 for test_converter
41 end for;
42 end CFG_converter_tb;
```

**Waveform Viewer Details:**

- Time: 320000000 FS
- Assertion Failure at 320000000 FS in des "TEST COMPLETE!"
- Signals: /TEST/I3, /TEST/I2, /TEST/I1, /TEST/I0, /TEST/A, /TEST/B, /TEST/C, /TEST/D, /TEST/E, /TEST/F, /TEST/G
- Time markers: 140000000, 0, 200000000, 440000000

# Device Programming

- ◆ Prepare the configuration bitstream file
- ◆ Configure FPGA device(s)
  - By downloading the configuration bitstream via a download cable
  - By programming the configuration bitstream onto a non-volatile memory device & attaching it on the circuit board



# HDL Coding Hints

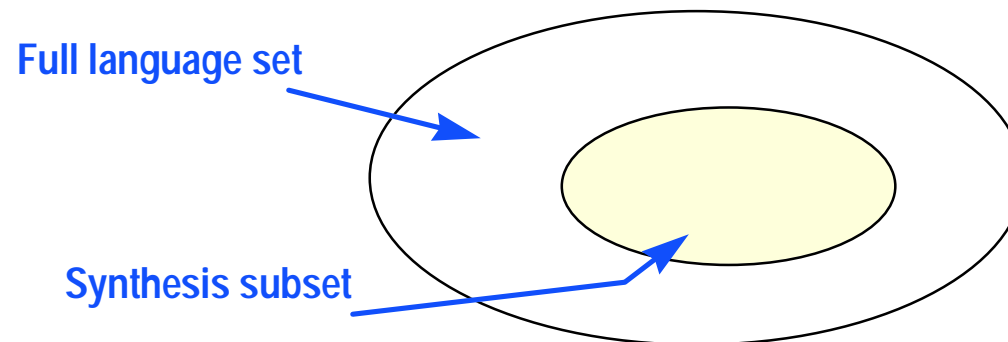
## Generic Coding Techniques & Considerations

- ◆ HDL Synthesis Subset
- ◆ HDL Coding Techniques & Examples
- ◆ Considerations for Synthesis

# HDL Synthesis Subset

## ◆ Verilog & VHDL are hardware description and simulation language and was not originally intended as an input to synthesis

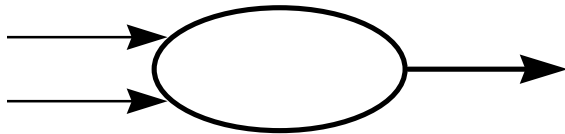
- Many hardware description and simulation constructs are not supported by synthesis tools
- Various synthesis tools use different subset of HDL -- *synthesis subset*
  - Each synthesis tool has its own synthesis subset
- Up to now, most synthesis tools can read, translate & optimize *RTL* (Register-Transfer Level) HDL well
  - Behavioral synthesis is under development



# RTL Coding Style

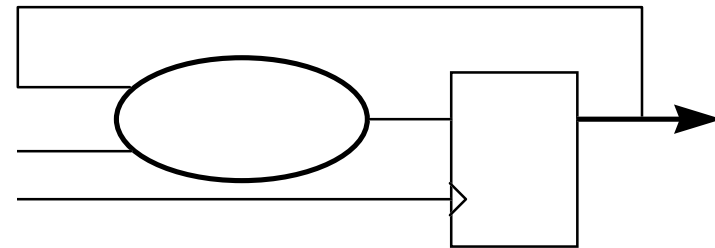

## Combinational process

```
process(      )  
begin  
-----  
-----  
-----  
end process;
```



## Clocked process

```
process(      )  
begin  
-----  
-----  
-----  
end process;
```





# RTL Modeling Issues

- ◆ **Consider implementation effects**
  - Propagation delays
  - Signal glitches
- ◆ **Define registers (synchronous)**
  - Store values
  - Clocking
  - Reset
  - Amount of logic between flip-flops
- ◆ **Operations must be scheduled**

# Synopsys Verilog/VHDL Synthesis Subset

- ◆ **We are going to learn Synopsys Verilog/VHDL synthesis subset**
  - Synopsys RTL coding style is widely accepted as a guideline to write a synthesizable Verilog or VHDL design file
    - Learning Synopsys Verilog or VHDL coding style can get the best synthesis results
    - It also can help avoid synthesis mismatch with simulation

# Synopsys RTL Synthesis Tools

## ◆ Design Analyzer

- User graphical interface of Synopsys synthesis tool

## ◆ HDL Compiler

- Translate Verilog descriptions into Design Compiler/FPGA Compiler

## ◆ VHDL Compiler

- Translate VHDL descriptions into Design Compiler/FPGA Compiler

## ◆ Design Compiler

- Constraint-driven logic optimizer

## ◆ FPGA Compiler

- Design Compiler Family synthesis tool that targets FPGA technologies
- Optimization for LUT architectures

# HDL Coding Techniques

## ◆ Why we need to learn HDL coding techniques?

- The designer has maximum influence on the final performance & density of the target device in the original coding style of the HDL
- The effect of coding style on final circuit performance can be very dramatic
  - In some cases, circuit speed can be improved by a factor of three and utilization by a factor of two by making simple changes to source code
- Ideally, you should code the design as abstractly as possible to preserve technology independence
  - However, the design can become faster and smaller as you design closer to the specific technology or device architecture

# Naming Conventions for Identifiers

- ◆ **User-defined names can not violate HDL naming rules**
  - Using HDL keywords as identifiers for objects is not allowed
- ◆ **FPGA tools may have other limitations on the character set**
  - You should ordinarily use identifiers consisting of letters and digits, with underscores ("\_") included where necessary to separate parts of a name
    - Do not use "/", "-", "\$", "<", ">" as name separator
    - FPGA resource names are reserved and should **not** be used

# Case Considerations

- ◆ **FPGA tools may use formats in which case does not matter**
  - Although Verilog considers upper and lower case letters to be different, you should **not** have two identifiers in the same Verilog module that differ only in case
    - For example, don't name one module "adder" and another one "Adder"

# Omitting Delay Statements

- ◆ **Avoid using any delay statements to create code that simulates the same way before and after synthesis**
  - Synthesis tools always ignore delay statements
  - Do **not** use delay statement in the design files
    - Do **not** use VHDL `wait for xx ns` or `after xx ns` statement
    - Do **not** use Verilog `#` delay token

# Omitting Initial Values

- ◆ **Do not assign signals and variables initial values to avoid synthesis mismatch with simulation**
  - Most synthesis tools will ignore signal and variable initial values
  - For flip-flops, use clear or reset signal to initialize their values



# Creating Readable Code

- ◆ **Create code that is easy to read, debug and maintain**
  - Indent blocks of code to align related statements
  - Use empty lines to separate top-level constructs, designs, architecture, process, ...
  - Use space to make your code easier to read
  - Break long lines of code at an appropriate point or a colon
  - Add comments to your code

# Using Labels

## ◆ Use labels to group logic

- You can use optional labels on flow control constructs to make the code structure more obvious
  - However, the labels are not exactly translated to gate or register names in your implemented design
- It's suggested to label all VHDL processes, functions, and procedures and Verilog blocks

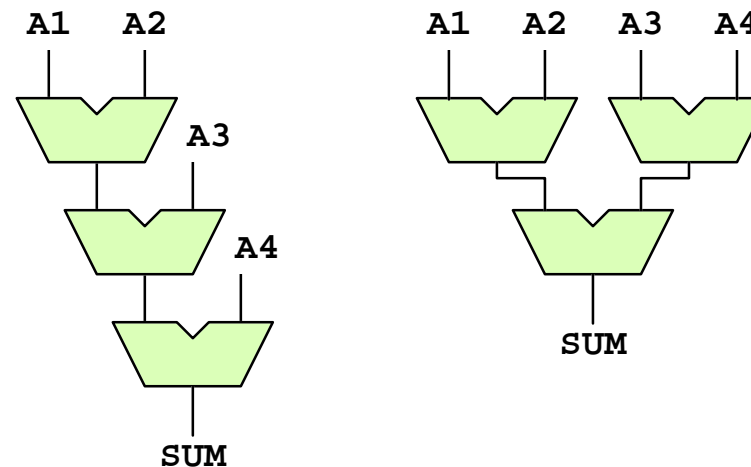
# Ordering & Grouping Arithmetic Functions

- ◆ The ordering & grouping of arithmetic functions may influence design performance, depending on the bit width of input signals

- Example1 (VHDL): `SUM <= A1 + A2 + A3 + A4;`

- Example2 (VHDL): `SUM <= (A1 + A2) + (A3 + A4);`

- Example1 cascades 3 adders in series
- Example2 creates two adders in parallel: (A1+A2) and (A3 + A4), and the results are combined with a third adder



# Synopsys DesignWare Library

## ◆ Synopsys DesignWare parts are ready to use

- Predefined, synthesizable resource blocks for ALU, mathematics, sequential, data integrity operations
  - These blocks facilitate the designer's ability to perform more high-level design trade-offs, and spend less time focusing on detailed design
- DesignWare components can be invoked by inference or instantiation
  - Default DesignWare component: adder, subtractor, comparator and multiplier circuit will be inferred automatically by Synopsys
  - Foundation DesignWare library (DW01 through DW05)
- DesignWare component is scaleable, technology-independent
  - The implementation of DesignWare component is decided by the optimization constraints
- Most FPGA vendors provide their own DesignWare library too
  - You can use it to obtain the optimal performance and increase the accuracy of the area and timing prediction

# VHDL IEEE Library

## ◆ Available packages for synthesis in the Synopsys IEEE library

- std\_logic\_1164
- std\_logic\_arith
- std\_logic\_unsigned
- std\_logic\_signed

# VHDL Data Type

## ◆ Std\_logic (IEEE 1164) type is recommended for synthesis

- This type is effective for hardware descriptions because it has nine different values
- Std\_logic type is automatically initialized to an unknown value
  - It forces you to initialize your design to a known state
  - Do not override this feature by initializing signals and variables to a known value when they are declared because the result may be a gate-level circuit that cannot be initialized to a known value
- Use Std\_logic package for all entity port declarations to make it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports

# VHDL Buffer Port Declaration

## ◆ Minimize the use of ports declared as buffers

- Declare a buffer when a signal is used internally and as an output port
  - Every level of hierarchy in your design that connects to a buffer port must be declared as a buffer
- To reduce the amount of coding in hierarchical designs, you may insert a dummy signal and declare the port as an output
  - It's often the best method because it does not artificially create a bidirectional port

```
P1: process begin
  wait until CLK'event and CLK = '1';
  C <= A + B + C;
end process P1;
```

```
C <= C_INT;
P1: process begin
  wait until CLK'event and CLK = '1';
  C_INT <= A + B + C_INT;
end process P1;
```

# VHDL Signals & Variables

## ◆ Compare signals and variables

- Signals are similar to hardware and are not updated until the end of a process
- Variables are immediately updated
  - They can mask glitches that may impact how your design functions
- It's better to use signals for hardware descriptions
  - However, variables allow quick simulation

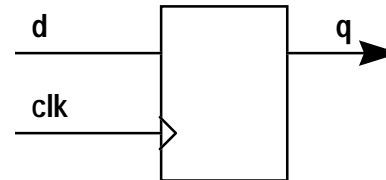


# VHDL Variable Assignments

## ◆ VHDL variable assignments execute in zero simulation time

- The value of a variable assignment takes effect immediately after execution of the assignment statement

```
architecture BEHAV of SHIFTER is
begin
    VAR_ASSIGNMENT: process(CLK)
        variable VAR_D: STD_LOGIC;
    begin
        if (CLK'EVENT and CLK = '1') then
            VAR_D := D;
            Q <= VAR_D;
        end if;
    end process VAR_ASSIGNMENT;
end BEHAV;
```

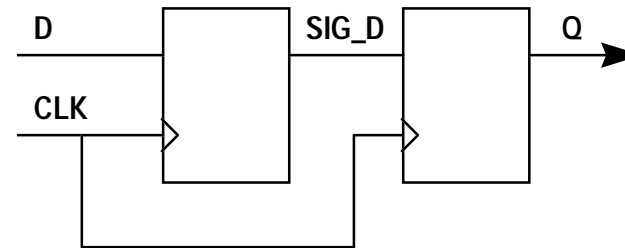


*The variable VAR\_D is optimized.*

# VHDL Signal Assignments

- ◆ VHDL signal assignments evaluate the right side expression and cause the simulator to schedule an update of the target
  - The actual assignment takes effect only after an actual or implied, (such as in a process with a sensitivity list), WAIT statement

```
architecture BEHAV of SHIFTER is  
  
    signal SIG_D: STD_LOGIC;  
  
begin  
  
    SIG_ASSIGNMENT: process(CLK)  
    begin  
        if (CLK'EVENT and CLK = '1') then  
            SIG_D <= D;  
            Q <= SIG_D;  
        end if;  
    end process SIG_ASSIGNMENT;  
  
end BEHAV;
```



# Verilog Procedural Assignments

## ◆ Blocking procedural assignment (=)

- The assignment is executed before the statement following the assignment is executed
- This is a very essential form of assignment that should be used in testbenches

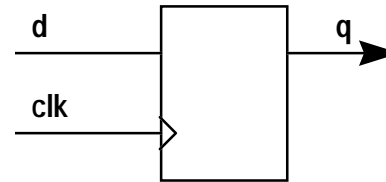
## ◆ Non-blocking procedural assignment (<=)

- Non-blocking assignments allow the simulator to schedule assignments without blocking the procedural flow
  - The assignments are performed within the same time step without any order dependence
- This is a very essential form of assignment that should be used in all design modules

# Verilog Blocking Assignments

- ◆ Verilog immediately assign the values in blocking assignments

```
module nbk1(q, d, clk);  
  
input d, clk;  
output q;  
  
reg q, bk_d;  
  
always @(posedge clk) begin  
    bk_d = d;  
    q = bk_d;  
end  
  
endmodule
```

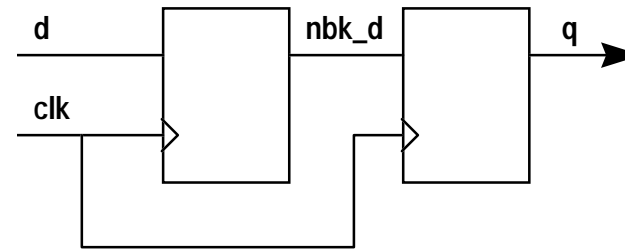


*The reg bk\_d is optimized.*

# Verilog Non-Blocking Assignments

- ◆ Verilog schedules the update of the target in non-blocking assignments

```
module nbk2(q, d, clk);  
  
  input d, clk;  
  output q;  
  
  reg q, nbk_d;  
  
  always @(posedge clk) begin  
    nbk_d <= d;  
    q <= nbk_d;  
  end  
  
endmodule
```



# Component Instantiation

## ◆ Named association vs. positional association

- Use positional association in function and procedures calls and in port lists only when you assign all items in the list
- Use named association when you assign only some of the items in the list
- Using named association is suggested

# Positional Association

- ◆ The signals are connected up in the order in which the ports were declared

```
-- VHDL example

architecture STRUCT of INC is

    signal X,Y,S,C : bit;

    component FADD
        port(A,B : in bit;
            SUM, CARRY : out bit);
    end component;

begin

    U1: FADD port map (X,Y,S,C);

    -- other statements

end STRUCT;
```

```
// Verilog example

module fadd(a, b, sum, carry);
    :
    :
    :
endmodule

module inc(...)
    :
    :
    wire x, y, s, c;
    :
    :
    fadd u1(x,y,s,c);
    :
    // other statements

endmodule
```

# Named Association

- ◆ The signals are connected up where the ports are explicitly referenced and order is not important

```
-- VHDL example

architecture STRUCT of INC is
    signal X,Y,S,C : bit;

    component FADD
        port(A,B : in bit;
            SUM, CARRY : out bit);
    end component;

begin

    ADDER1: FADD port map
        (B => Y, A => X, SUM => S, CARRY => C);

    -- other statements

end STRUCT;
```

```
// Verilog example

module fadd(a, b, sum, carry);
    :
    :
    :
endmodule

module inc(...)
    :
    :
    wire x, y, s, c;
    :
    :
    fadd adder1(.a(x), .b(y), .sum(s), .carry(c));
    :
    // other statements

endmodule
```



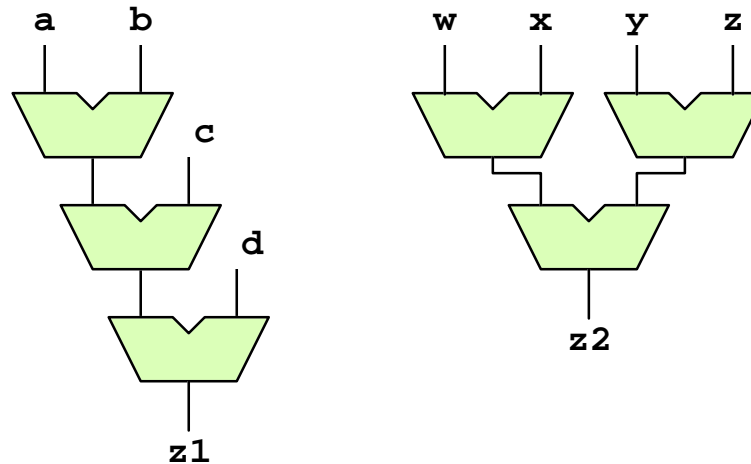
# Using Parentheses

## ◆ Use parentheses efficiently

- You may use parentheses to control the structure of a complex design

```
// Verilog  
assign z1 = a + b + c + d;  
assign z2 = (w + x) + (y + z);
```

```
-- VHDL  
Z1 <= A + B + C + D;  
Z2 <= (W + X) + (Y + Z);
```



# Specifying Bit-Width

## ◆ Use operator bit-width efficiently

- You may directly specify the operand bits to reduce area

```
// Verilog
input [7:0] a, b;
output [7:0] z1, z2;

assign z1 = a + b;
assign z2 = a + b[4:0];

-- VHDL, must use std_logic_unsigned package
A, B: in STD_LOGIC_VECTOR(7 downto 0);
Z1, Z2: out STD_LOGIC_VECTOR(8 downto 0);
...
Z1 <= '0'&A + B;
Z2 <= '0'&A + B(4 downto 0);
```

# Level-Sensitive Latches

## ◆ Level-sensitive latch

- A variable assigned within an always block that is not fully specified in all branches will infer a latch
- In some FPGA devices, latches are not available in the logic function block
  - Latches described in RTL HDL are implemented with gates, and will result in combination loops and hold-time requirements

# Latch Inference - Verilog

```
// Ex1. simple D-latch
always @(datain or enable)
begin
    if (enable) dataout = datain;
end

// Ex2. simple D-latch with active-low RESET
always @(datain or enable or reset)
begin
    if (!reset) dataout = 0;
    else if (enable) dataout = datain;
end
```

# Latch Inference - VHDL

```
-- Ex1. simple D-latch
DLAT: process(DATAIN, ENABLE)
begin
  if (ENABLE = '1') then DATAOUT <= DATAIN;
  end if;
end process DLAT;

-- Ex2. simple D-latch with active-low RESET
DLAT_R: process(DATAIN, ENABLE, RESET)
begin
  if (RESET = '0') then DATAOUT <= '0';
  elsif (ENABLE = '1') then DATAOUT <= DATAIN;
  end if;
end process DLAT_R;
```

# Edge-Sensitive D Flip-Flops

## ◆ Edge-sensitive D flip-flops

- In contrast to latches, flip-flops(registers) are inferred by the detection of the clock edge
  - Verilog: `@(posedge clk)` or `@(negedge clk)` in a always block
  - VHDL: `(clk'event and clk = '1')` or `(clk'event and clk = '0')`

# D Flip-Flop Inference - Verilog

```
// Ex1. simple DFF
always @(posedge clk)
begin
    dataout = datain;
end

// Ex2. DFF with asynchronous active-low RESET
always @(posedge clk or negedge reset)
begin
    if (!reset) dataout = 0;
    else dataout = datain;
end

// Ex3. DFF with synchronous active-low RESET
always @(posedge clk)
begin
    if (!reset) dataout = 0;
    else dataout = datain;
end

// Ex4. DFF with active-high clock ENABLE
always @(posedge clk)
begin
    if (enable) dataout = datain;
end
```

# D Flip-Flop Inference - VHDL

```
-- Ex1. simple DFF
DFF1: process(CLK)
begin
  if (CLK'EVENT and CLK = '1') then DATAOUT <= DATAIN;
  end if;
end process DFF1;

-- Ex2. DFF with asynchronous active-low RESET
DFF2_R: process(RESET, CLK)
begin
  if (RESET = '0') then DATAOUT <= '0';
  elsif (CLK'EVENT and CLK = '1') then DATAOUT <= DATAIN;
  end if;
end process DFF2_R;

-- Ex3. DFF with synchronous active-high ENABLE
DFF3_EN: process(CLK)
begin
  if (CLK'EVENT and CLK = '1') then
    if (ENABLE = '1') then DATAOUT <= DATAIN;
    end if;
  end if;
end process DFF3_EN;

-- Another style
-- if (ENABLE = '0') then null;
-- elsif (CLK'EVENT and CLK = '1') then
--   DATAOUT <= DATAIN;
-- end if;
```



# Output Pad Tri-State Buffers

## ◆ Output pad tri-state buffers

- Output tri-state buffers are available in I/O block of FPGA devices
  - To drive an output port
  - To drive internal logic connected to an *inout* port
- Can implement tri-state registered output in some FPGA architectures
  - The flip-flop must directly drive the 3-state signal

# Tri-State Buffer Inference - Verilog

```
// Ex1. Output pad tri-state buffers (for top-level output port)
assign out1 = (oe1) ? data1 : 1'bz;

// Ex2. Bidirectional pad
assign io1 = (oe1) ? data1 : 1'bz;
assign data2 = io1 & in1 & in2;

// Ex3. 3-state registered output
always @(posedge clk) bus_q = data3;
always @(oe2 or bus_q) begin
    if (!oe2) bus_out = bus_q;
    else bus_out = 1'bz;
end
```

# Tri-State Buffer Inference - VHDL

```
-- Ex1. Output pad tri-state buffers (for top-level output port)
TRI1: process(DATA1, OE1)
begin
  if (OE1 = '1') then OUT1 <= DATA1;
    else OUT1 <= 'Z';
  end if;
end process TRI1;

-- Ex2. Bidirectional pad
TRI2: process(DATA1, OE1)
begin
  if (OE1 = '1') then IO1 <= DATA1;  -- IO1 is declared as an inout port
    else IO1 <= 'Z';
  end if;
end process TRI2;
DATA2 <= IO1 & IN1 & IN2;

-- Ex3. 3-state registered output
TRI_REG: process(CLK)
begin
  if (CLK'EVENT and CLK = '1') then BUS_Q <= DATA3;
  end if;
end process TRI_REG;
BUS_OUT <= BUS_Q when (OE2 = '0') else 'Z';
```

# Internal Tri-State Buses

## ◆ Internal tri-state buses

- Xilinx devices support many internal tri-state buffers
  - Easy to implement bus functions
  - Can be used to implement some wide functions
  - Xilinx internal tri-state buffers (TBUFs) are active-low enable
- Altera tri-state emulation
  - Altera devices do not support internal tri-state buffers
  - Altera software will convert internal tri-state buffers into multiplexers

# Tri-State Bus Inference - Verilog

```
// Ex1. Internal tri-state buffers  
assign busa = (!oe_a) ? a : 8'bz;  
assign busa = (!oe_b) ? b : 8'bz;
```

# Tri-State Bus Inference - VHDL

```
-- Ex1. Internal tri-state buffers
BUS_IN1: process(A, OE_A)
begin
  if (OE_A = '0') then BUSA <= A;
    else BUSA <= "ZZZZZZZZ";
  end if;
end process BUS_IN1;

BUS_IN2: process(B, OE_B)
begin
  if (OE_B = '0') then BUSA <= B;
    else BUSA <= "ZZZZZZZZ";
  end if;
end process BUS_IN2;
```

# Using If Statements

## ◆ Use `if` statements

- `if` statement provides more complex conditional actions
- An `if` statement generally builds a priority encoder that gives the shortest path delay to the first branch in the `if` statement
- You must specify all the branches in a `if` statement, otherwise Synopsys may infer latches
- Do not use deeply nested `if` statement
  - Improper use of the nested-if statement can result in an increase in area and longer delays in your designs

# If Statement Examples - Verilog

```
// Ex1. Use if statements
always @(a or b or c or d or e or f or g or sel)
begin
    if (sel == 3'b000) mux_out = a;
    else if (sel == 3'b001) mux_out = b;
    else if (sel == 3'b010) mux_out = c;
    else if (sel == 3'b011) mux_out = d;
    else if (sel == 3'b100) mux_out = e;
    else if (sel == 3'b101) mux_out = f;
    else if (sel == 3'b110) mux_out = g;
    else mux_out = 1'b0;
end
```



# If Statement Examples - VHDL

```
-- Ex1. use if statement
IF_1: process (A,B,C,D,E,F,G,SEL)
begin
  if (SEL = "000") then MUX_OUT <= A;
  elsif (SEL = "001") then MUX_OUT <= B;
  elsif (SEL = "010") then MUX_OUT <= C;
  elsif (SEL = "011") then MUX_OUT <= D;
  elsif (SEL = "100") then MUX_OUT <= E;
  elsif (SEL = "101") then MUX_OUT <= F;
  elsif (SEL = "110") then MUX_OUT <= G;
  else mux_out <= '0';
  end if;
end process IF_1;
```

# Using Case Statements

## ◆ Use case statements

- A `case` statement generally builds a multiplexer, where all the paths have similar delays
- If a `case` statement is not a “full case”, it will infer latches
  - Use `default` statements or use “don't care” inference
- If Synopsys can't determine that case branches are parallel, it may build a priority decoder

# Case Statement Examples - Verilog

```
// Ex1. Use case and default statements
always @(a or b or c or d or e or f or g or sel)
begin
  case (sel)
    3'b000: mux_out = a;
    3'b001: mux_out = b;
    3'b010: mux_out = c;
    3'b011: mux_out = d;
    3'b100: mux_out = e;
    3'b101: mux_out = f;
    3'b110: mux_out = g;
    default: mux_out = 1'b0; // You may assign to 1'bx for specifying a don't care
  endcase
end

// Ex2. Use don't cares in case statements
always @(bcd) begin
  case (bcd)
    4'd0: bout = 3'b001;
    4'd1: bout = 3'b010;
    4'd2: bout = 3'b100;
    default: bout = 3'bxxx;
  endcase
end
```

# Case Statement Examples - VHDL

```
-- Ex1. Use case & default statements
CASE_1P: process (A,B,C,D,E,F,G,SEL)
begin
  case SEL is
    when "000" => MUX_OUT <= A;
    when "001" => MUX_OUT <= B;
    when "010" => MUX_OUT <= C;
    when "011" => MUX_OUT <= D;
    when "100" => MUX_OUT <= E;
    when "101" => MUX_OUT <= F;
    when "110" => MUX_OUT <= G;
    when others => MUX_OUT <= '0'; -- You may assign a '-' for specifying a don't care
  end case;
end process CASE_1P;

-- Ex2. Use don't cares in case statements
CASE_2P: process (BCD)
begin
  case BCD is
    when "0000" => BOUT <= "001";
    when "0001" => BOUT <= "010";
    when "0010" => BOUT <= "100";
    when others => BOUT <= "----";
  end case;
end process CASE_2P;
```

# The `full_case` Directive

## ◆ You can attach the `full_case` directive to tell the synthesis tool the Verilog `case` item expressions cover all the possible cases

- Synopsys does not do an exhaustive data flow analysis for the case statements, and so cannot tell that all the possible case items have been enumerated
  - Consequently, Synopsys implements extra logic to keep the value when it has some values other than those listed in the case statements
- If you are sure the unspecified branches will never occur, you may use Synopsys `full_case` directive to specify a full case in Verilog and eliminate the extra logic
  - The `full_case` directive should appear immediately following the test expression of the `case` statement
- It is recommended that you still use `full_case` if the case items do not include a `default` item
  - This may improve the efficiency of the synthesis process

# The `parallel_case` Directive

- ◆ Attach the `parallel_case` directive to improve the efficiency if no two case item expressions ever match the test expression at the same time
  - You may use Synopsys `parallel_case` to specify a parallel case in Verilog, which force to generate multiplexer logic instead of priority decoder
    - The `full_case` directive should appear immediately following the test expression of the case statement

# Using full\_case & parallel\_case

```
module johnson (clock, reset, count);

input clock, reset;
output [2:0] count;

wire clock, reset;
reg [2:0] count;

always @(negedge clock or posedge reset)
  if (reset == 1'b1)
    count <= 3'b000;
  else
    case (count) // synopsys full_case parallel_case
      3'b000: count <= 3'b001;
      3'b001: count <= 3'b011;
      3'b011: count <= 3'b111;
      3'b111: count <= 3'b110;
      3'b110: count <= 3'b100;
      3'b100: count <= 3'b000;
    endcase
endmodule
```

# Using For Loop Statements

## ◆ Use `For` loop statements

- Use `for` loop to duplicate statement
  - Loop index variable must be integer type
  - Step, start & end value must be constant



# For Loop Statement Examples - Verilog

```
-- Ex1. parity tree
always @(word)
begin
  is_odd = 0;
  for (i=0; i<=7; i=i+1) begin
    is_odd = is_odd xor word[i];
  end
end

assign parity = is_odd;
```

# For Loop Statement Examples - VHDL

```
-- Ex1. parity tree
FOR_1: process (WORD)
  variable IS_ODD: STD_LOGIC;
begin
  IS_ODD := '0';
  LOOP1: for I in 0 to 7 loop
    IS_ODD := IS_ODD xor WORD(I);
  end loop LOOP1;
  PARITY <= IS_ODD;
end process FOR_1;
```

# Designing Counters

## ◆ Counters can be built by inference or instantiation

- Infer a simple counter
  - Synopsys will build it out of gates
- Infer a Synopsys DesignWare counter
- Instantiate the scaleable Synopsys DesignWare counter: `DW03_updn_ctr` or others
- Instantiate the optimized counter macrofunctions provided by FPGA vendors

# Counter Examples - Verilog

```
// Ex1. Infer a loadable up-down counter with asynchronous reset
always @(negedge reset or posedge clk)
begin : upcnt_1
  if (!reset) count = 8'b00000000;
  else if (load) count = datain;
  else if (en)
    if (updn) count = count + 1;
    else count = count - 1;
end
assign dataout = count;

// Ex2. Instantiate a DesignWare counter
DW03_updn_ctr # (8) U0 (.data(datain), .load(load), .up_dn(updn),
    .cen(en), .clk(clk), .reset(reset), .count(dataout));
```

# Counter Examples - VHDL

```
-- Ex1. Infer a loadable up-down counter with asynchronous reset
UPCNT_1: process(RESET, CLK)
begin
  if (RESET = '0') then COUNT <= "00000000";
  elsif (CLK'EVENT and CLK = '1') then
    if (LOAD = '1') then COUNT <= DATAIN;
    elsif (EN = '1') then
      if (UPDN = '1') then COUNT <= COUNT + 1;
      else COUNT <= COUNT - 1;
      end if;
    end if;
  end if;
  DATAOUT <= COUNT;
end process UPCNT_1;

-- Ex2. Instantiate a DesignWare counter
-- DW03 library and package must be included for simulation and synthesis
U0: DW03_updn_ctr
  generic map (width => 8)
  port map( data => DATAIN, load => LOAD, up_dn => UPDN, cen => EN, clk => CLK,
           reset => RESET, count => DATAOUT);
```

# Designing Multipliers

## ◆ Design multipliers

- Infer a simple, non-DesignWare multiplier using the HDL operator "\*", which will be built out of gates and optimized
- Instantiate the multiplier as a sequence of adders, each of which can be mapped to the DesignWare adder
- Instantiate Synopsys DesignWare multiplier, `DW02_mult` or others
- Instantiate the optimized multiplier macrofunctions provided by FPGA vendors

# Multiplier Examples - Verilog

```
// Ex1. Infer a simple non-pipelined multiplier
assign multout = a * b;

// Instantiate DW02_mult_5_stage
DW02_mult_5_stage #(8,8) U1(in1,in2,control,clk,product);
```

# Multiplier Examples - VHDL

```
-- Ex1. Infer a simple non-pipelined multiplier
multout <= a * b;

-- Instantiate DW02_mult_5_stage
U1: DW02_mult_5_stage
    generic map(A_width => 8, B_width => 8)
    port map(A => in1, B => in2, TC => control, CLK => clk, PRODUCT => product);
```



# Designing FSMs

## ◆ Design FSMs (Finite State Machines)

- A FSM is a design that has a clearly defined set of internal states and a defined method of moving between them
- Most synthesis tools recognize any clocked process as an implicit state machine, that is, as a state machine for which the state variable is implicit
- You can explicitly define a Verilog register or VHDL signal to be used as the state variable for state machine
  - FSMs designed for synthesis are usually represented as two processes/blocks, one calculates the next state (combinational logic), and one assigns the next state to the state vector register(s) on the clock edge (storage element)

# FSM Optimization

- ◆ **Most synthesis tools identify certain forms as representing state machines and provide mechanisms for controlling the encoding of the associated state variable**
  - For example, Synopsys FSM Extraction tool

# Using Synopsys FSM Directives

## ◆ You must explicitly specify FSM blocks in your code if you want to use Synopsys FSM Extraction tool in the later design stage

- In Verilog, you must use Synopsys FSM directives to specify FSM blocks:

```
/* synopsys enum enum_name */ : specifies default state machine encoding
```

```
// synopsys state_vector vector_name : indicates the state vector variable (reg)
```

- In VHDL, you must use VHDL attributes to specify the default FSM encoding:

```
attribute ENUM_ENCODING: string;
```

```
attribute ENUM_ENCODING of state_type : type is "... ";
```

And you must use VHDL attributes to specify the state vector signal:

```
attribute STATE_VECTOR : STRING;
```

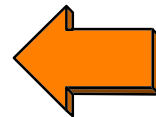
```
attribute STATE_VECTOR of arch_name : ARCHITECTURE is "sig_name";
```

# FSM Example - Verilog

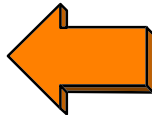
```
parameter [4:0] /* synopsys enum day_type */
    sun = 5'b10000, mon = 5'b01000, tue = 5'b00100, wed = 5'b00010,
    thu = 5'b00001, fri = 5'b10001, sat = 5'b01110;
reg [4:0] /* synopsys enum day_type */ day, next_day;

// synopsys state_vector day

always @(posedge clk negedge clear)
begin : SYNC
    if (!clear) day = sun;
    else day = next_day;
end
always @(day)
begin : COMB
    bell_ring = 0;
    case (day) // synopsys full_case
        sun: next_day = mon;
        mon: next_day = tue;
        tue: next_day = wed;
        wed: next_day = thu;
        thu: next_day = fri;
        fri: next_day = sat;
        sat: begin
            next_day = sun; bell_ring = 1;
        end
    endcase
end
```



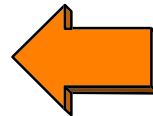
update state vector



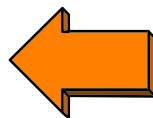
combinational logic

# FSM Example - VHDL

```
architecture BEHAVIOR of FSM2 is
  type DAY_TYPE is (SUN, MON, TUE, WED, THU, FRI, SAT);
  attribute ENUM_ENCODING : string;
  attribute ENUM_ENCODING of DAY_TYPE : type is "10000 01000 00100 00010 00001 10001 01110";
  signal DAY, NEXT_DAY: DAY_TYPE;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of BEHAVIOR : ARCHITECTURE is "DAY";
begin
  SYNC: process(CLK, CLEAR)
  begin
    if (CLEAR = '0') then DAY <= SUN;
    elsif (CLK'EVENT and CLK = '1') then DAY <= NEXT_DAY;
    end if;
  end process SYNC;
  COMB: process (DAY)
  begin
    BELL_RING <= '0';
    case DAY is
      when SUN => NEXT_DAY <= MON;
      when MON => NEXT_DAY <= TUE;
      when TUE => NEXT_DAY <= WED;
      when WED => NEXT_DAY <= THU;
      when THU => NEXT_DAY <= FRI;
      when FRI => NEXT_DAY <= SAT;
      when SAT => NEXT_DAY <= SUN; BELL_RING <= '1';
    end case;
  end process COMB;
```



update state vector



combinational logic

# Designing Memories

- ◆ **Do not behaviorally describe RAMS in your code because the synthesis tool will create combinational loops and a lot of registers**
  - None of synthesis tools can judge a RAM block or process
  - Instead, you should instantiate a RAM module or primitive provided by the vendor
- ◆ **FPGA vendors often provide the memory generator tool to create the following items for simulation and synthesis:**
  - Behavioral Verilog/VHDL simulation model files
    - They can not be used as the input to the synthesis tool
  - Example files to tell you how to instantiate the RAM/ROM module in your design file
  - Timing model files for the synthesis tool
  - The netlist files required by the FPGA implementation tools

# Synchronous & Asynchronous RAM

- ◆ **Some FPGA devices provides synchronous and asynchronous RAM blocks**
  - Use synchronous RAM instead of asynchronous RAM if possible
    - FPGA RAM block usually has a self-timed write pulse
    - Address can be change after it is clocked into RAM
    - Besides, you may register RAM outputs for pipelining

# Considerations for Synthesis

- ◆ **The main consideration for synthesis: overall cost**
  - Time is money!
  - Area is money!
  - Quality is money!



# Considering the Design

## ◆ The main considerations in the design stage are:

- Level of abstraction in HDL modeling
- Levels of hierarchy
- Technology-dependent or independent
- Reliability
- Easy to maintain and debug your code
- Development time
- Simulation time
- Easy to upgrade
- Migration to ASIC

# Considering Synthesis Process

- ◆ **The main considerations in the synthesis process are:**
  - Optimization constraints
  - Hierarchy structure of your design
  - CPU time
  - Memory requirements

# Considering Synthesis Results

- ◆ **The main considerations for the synthesis results are:**
  - Performance
  - Critical timing paths
  - Area
  - FPGA resource utilization

# Instantiation vs. Inference

## ◆ Instantiation vs. inference

- Instantiation: explicitly select a technology-specific element in the target library
- Inference: direct the HDL Compiler to infer latches or flip-flops or other resource blocks from your Verilog or VHDL description

## ◆ Technology-independent vs. technology-dependent

- Keeping the pre-synthesis design source code technology-independent allows you to re-target to other technologies at a later time, with a minimum of re-design effort
- Technology-dependent design uses dedicated functions or design techniques optimized for speed or area
  - The designer may require detailed understanding on device architectures

# Synchronous vs. Asynchronous

- ◆ **Synchronous designs run smoothly through synthesis, simulation, test and implementation**
  - Synchronous designs uses more registers, which are rich in FPGAs
  - Watch for unintentional latch inference
    - Remember completely specify all branches for every case or if statement
  - If asynchronous logic blocks are required, put them into separate blocks

# Thinking Hardware

## ◆ The most straightforward coding of a design is not always the best choice for actual hardware implementation

- Think if you can reduce logic from the architecture view before starting coding
- Pre-schedule your RTL design if possible

a[31:0]

(shiftcnt=5)

if a[31] == 0, result = {5{1'b0}, a[31], ..., a[5]}

if a[31] == 1, result = {5{1'b1}, a[31], ..., a[5]}

```
module shift(a, shiftcnt, z);  
input [31:0] a;  
input [4:0] shiftcnt;  
output [31:0] z;  
  
assign z = {{31{a[31]}}, a} >> shiftcnt;  
endmodule
```

190 Altera LCs

```
module shift(a, shiftcnt, z);  
input [31:0] a;  
input [4:0] shiftcnt;  
output [31:0] z;  
  
assign z = a[31] ? ((a >> shiftcnt) |  
                  (((32'b1 << shiftcnt) - 1)) << (32 - shiftcnt))  
            : (a >> shiftcnt);  
endmodule
```

405 Altera LCs

# Design Partitioning

## ◆ Design partitioning

- Manage your design hierarchy to reduce CPU time during synthesis, and also achieve higher performance
  - A very big design without hierarchy may require 24 hours or above of run time
  - A design with many levels of hierarchy may reduce efficiency during synthesis process

# Considering Design Partitioning

## ◆ Strategies for design partitioning

- Partition each design block to target < 5,000 gates size
- Do **not** partition a basic logic element, for example, a flip-flop or one-bit adder, into a basic block in a large design
  - The number of the hierarchy levels will be too large and the compile time will increase very much
- Separate datapath with control logic
- Separate structured circuit with random logic
  - Structured circuit: adder, multiplier, comparator, MUXs, ALU, ...
  - Unstructured circuit: random control logic, PLA, ...
- Merge blocks that can share resources
- Isolate state machines
- Keep related combinational logic together in a single block
- Keep critical paths in a single block and remove circuit that is not speed critical from the critical path block



# Registered Outputs

## ◆ Registered outputs

- Register the outputs of lower-level files in the design hierarchy whenever possible
  - This process generates code that is easier to maintain and it can be more successfully synthesized
- Registered outputs also make it easier to provide accurate timing constraints
- Use a single clock for each design clock
- Separate combinational and sequential assignments

# Resource Sharing

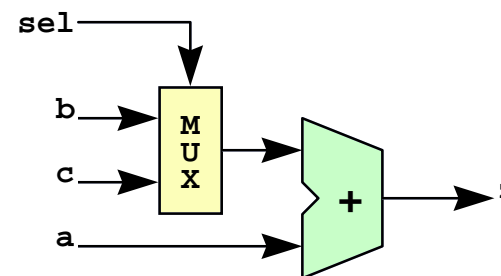
## ◆ Resource sharing

- Operations can be shared if they lie in the same block
- May improve resource utilization & design performance

## ◆ Strategies

- Keep resources that might be shared between different functions or that might require a similar configuration in the same block of the design hierarchy
- Group similar designs into the same hierarchy to allow Synopsys to share resources when practical

```
always @(a or b or c or sel)
  if (sel) z = a + b;
  else z = a + c;
```



# Pipelining & Pipeline Re-Timing

## ◆ Pipelining & pipeline re-timing

- To pipeline high-speed sequential designs whenever possible
- FPGA devices are register-rich
  - FPGA devices provide a high ratio of flip-flops to combinational logic
  - The architecture makes pipelining & pipeline re-timing especially advantageous techniques for FPGAs
  - There is often little or no area penalty associated with using extra registers
- You can make a pipelined design to achieve higher performance & higher device utilization simultaneously in the FPGA architecture

# Designing FSMs

## ◆ FSM (Finite State Machine) encoding is the key issue

- The number of available registers affects the design of FSMs
  - Binary encoding of FSMs is designed to minimize the usage of flip-flops
  - Gray encoding is designed to avoid glitches
  - “One-hot” encoding is typically the fastest state machine implementation available in FPGA architecture
- All state flip-flops must be driven by the same clock
- In cases where “fail-safe” behavior is required, specify default clause in every **case** statement
  - Synopsys will generate the required logic to reset the state machine to a known state when it goes to an unknown state

## ◆ Using Synopsys FSM extraction & optimization tool

- Follow Synopsys FSM coding style so that you can use Synopsys FSM extraction & optimization tool to get the best FSM implementation

# State Machine Partitioning

## ◆ State machine partitioning

- A complex state machine can be partitioned into smaller state machines
  - For example, you may be able to reduce a 100-state state machine into 3 state machines of 6 states each
  - With good designs, there can be a positive effect on both performance & area
  - Each sub-machine needs an *IDLE* state during which the control is passed to another sub-machine
- Partition your design into control logic & datapath elements and use a state machine for the control blocks

# Altera-Specific Issues

- ◆ Using Flip-Flops Instead of Latches
- ◆ Designing Tri-State Buses
- ◆ Altera Primitives
- ◆ Altera Macrofunctions
- ◆ Altera DesignWare Modules
- ◆ FLEX 10K RAMs & ROMs

*Please refer to "Altera & Synopsys" training manual to know how to simulate a Altera-specific designs.*

# Using Flip-Flops Instead of Latches

## ◆ Level-sensitive latch vs. edge-sensitive flip-flop

- Altera recommends using flip-flops rather than latches
  - Latches are built by combining the combinational logic in the LE with feedback, and will result in combination loops and hold-time requirements
- Be careful when using HDL `case` or `if` statements
  - Specify all the branches in a `if` or `case` statement if possible

# Designing Tri-State Buses

## ◆ Tri-state emulation in Altera devices

- You can infer tri-state buffers in HDL modeling and synthesis
- However, Altera devices do not have real internal tri-state buffers
- Altera software will *emulate* tri-state buses by using multiplexers and by routing the bidirectional line outside of the device and then back in through another pin
  - When tri-state buses are used to multiplex signals, Altera MAX+PLUS II Compiler will convert the logic to a combinatorial multiplexer
  - When tri-state buses are used for bidirectional communication, you can rout this bidirectional line outside of the device, which uses the tri-states present at the I/O pins, or you can convert the tri-state bus into a multiplexer



# Altera-Provided Primitives

## ◆ Altera provides many primitives you can instantiate in your HDL design files

- Buffer primitives:
  - `AGLOBAL`, `ALCELL`, `ASOFT`, `ATRIBUF`, `ACARRY`, `ACASCADE`, `EXP`, `OPNDRN`, ...
- Latch & flip-flop primitives:
  - `LATCH`, `DFP`, `DFFS`, `DFFE`, `DFFC`, `DFFP`, `TFF`, `TFFS`, `TFFC`, `TFFE`, ...
- Logic primitives:
  - `ATBL_x`, `INV`, `ANDx`, `NANDx`, `ORx`, `NORx`, `XOR2`, `XNOR2`, ...

## ◆ In most circumstances, you do not need to instantiate Altera primitives

- Latch, flip-flop and logic primitives are automatically inferred by the synthesis tools
- Only some buffer primitives may be used to control the logic synthesis process

# Special Buffer Primitives - (1)

## ◆ **AGLOBAL (GLOBAL)**

- To indicate that a signal must use a global clock, clear, preset or output enable signal, instead of signals generated with internal logic or driven by ordinary I/O pins
- A inverter gate may be inserted between the input pin and GLOBAL

## ◆ **ALCELL (LCELL)**

- To allocate a logic cell for the project (an LCELL buffer always consumes one logic cell)
  - It's not removed from a project during Altera implementation
- It can be used to create an intentional delay or asynchronous pulse
  - However, race conditions can occur and create an unreliable circuit because the delay of these elements varies with temperature, power supply voltage and device fabrication process

## ◆ **ASOFT (SOFT)**

- To specify that a logic cell **may** be needed in the project
- During implementation, MAX+PLUS II Compiler examines the logic feeding the primitive and determines whether a logic cell is needed

# Special Buffer Primitives - (2)

## ◆ **ATRIBUF (ATRI, TRI)**

- A active-high tri-state buffer

## ◆ **OPNDRN**

- An open-drain buffer, equivalent to a `TRI` primitive whose output enable input is fed by an signal, but whose primary input is fed by a `GND` primitive
- Only supported for the FLEX 10K and MAX 7000S device families

# Buffer Primitives Instantiation

## ◆ Note when instantiating Altera buffer primitives

- Component names without an "A" prefixed (such as `GLOBAL`, `LCELL`, `SOFT`, `TRI`)
  - The input, output and output enable port names are `A_IN` and `A_OUT` and `OE`
  - They are set to "don't touch" and Synopsys will not optimize them
  - Only compliant with VHDL VITAL 2.2b
  - **Not** suggested to use for FLEX designs
- Component names with an "A" prefixed (such as `AGLOBAL`, `ALCELL`, `ASOFT`, `ATRIBUF`)
  - The input, output and output enable port names become `IN1`, `Y`, and `OE`
  - `ALCELL` are not set to "don't touch"
  - Compliant with VHDL VITAL 3.0

# Simulation Models of Primitives

## ◆ Simulation models have been included in the software

- Verilog simulation model for Altera's primitives:  
`/usr/maxplus2/synopsys/library/alt_pre/verilog/src/altera.v`
- VHDL source files: `flex.vhd` and `flex.cmp` under the directory  
`/usr/maxplus2/synopsys/library/alt_pre/vital/src`
- Synopsys VSS library directory:  
`/usr/maxplus2/synopsys/library/alt_pre/vital/lib/flex_vtl/`  
(Library name: `flex_vtl` ; Package name: `vcomponents`)

# Using Altera Primitives - Verilog

```
// Ex1. a parity tree
wire [7:1] parityout;
wire [7:1] parity;
wire ww0;
integer i;

AGLOBAL U0(.Y(ww0), .IN1(word[0]));
assign parity[1] = ww0 ^ word[1];
ASOFT U1(.Y(parityout[1]), .IN1(parity[1]));
assign parity[2] = parityout[1] ^ word[2];
ASOFT U2(.Y(parityout[2]), .IN1(parity[2]));
assign parity[3] = parityout[2] ^ word[3];
ASOFT U3(.Y(parityout[3]), .IN1(parity[3]));
assign parity[4] = parityout[3] ^ word[4];
ASOFT U4(.Y(parityout[4]), .IN1(parity[4]));
assign parity[5] = parityout[4] ^ word[5];
ASOFT U5(.Y(parityout[5]), .IN1(parity[5]));
assign parity[6] = parityout[5] ^ word[6];
ASOFT U6(.Y(parityout[6]), .IN1(parity[6]));
assign parity[7] = parityout[6] ^ word[7];
ASOFT U7(.Y(parityout[7]), .IN1(parity[7]));

assign is_odd = parityout[7];
```

# Using Altera Primitives - VHDL

```
-- Ex1. a parity tree
architecture BEHAVIOR of GENERATE3 is
  component ASOFT                                -- Must include FLEX_VTL library
    port (Y: out STD_LOGIC;                      -- and FLEX_VTL.VCOMPONENTS package
          IN1: in STD_LOGIC);                   -- for simulation purpose
  end component;
  component AGLOBAL
    port (Y: out STD_LOGIC;
          IN1: in STD_LOGIC);
  end component;
  signal PARITY, PARITYOUT: STD_LOGIC_VECTOR(7 downto 0);
  signal AA0: STD_LOGIC;
begin
  U0: AGLOBAL port map(Y => AA0, IN1 => A(0)); -- Named Association
  FOR_1: for I in 1 to 7 generate
    G1: if (I = 1) generate
      PARITY(I) <= AA0 xor A(I);
      U1: ASOFT port map(PARITYOUT(I), PARITY(I)); -- Positional Association
    end generate;
    G2: if (I > 1) generate
      PARITY(I) <= PARITYOUT(I-1) xor A(I);
      UGEN: ASOFT port map(PARITYOUT(I), PARITY(I)); -- Positional Association
    end generate;
  end generate;
  PARITY_OUT <= PARITYOUT(7);
end BEHAVIOR;
```

# Altera Macrofunctions

## ◆ Altera supplies 4 macrofunctions which are compact & run at high speed

- `A_81MUX` : multiplexer (Altera's 81mux macrofunction)
- `A_8COUNT` : up/down counter (Altera's 8count macrofunction)
- `A_8FADD` : full adder (Altera's 8fadd or 8faddb macrofunction)
- `A_8MCOMP` : magnitude comparator (Altera's 8mcomp or 8mcompb macrofunction)
  - Refer to Altera on-line help or their simulation models for detailed function descriptions



# Simulation Models of Macrofunctions

## ◆ Simulation models have been included in the software

- Verilog simulation model for Altera's macrofunctions:  
`/usr/maxplus2/synopsys/library/alt_mf/src/mf.v`
- VHDL source files: `mf.vhd` and `mf_components.vhd` under the directory  
`/usr/maxplus2/synopsys/library/alt_mf/src/`
- Synopsys VSS library directory:  
`/usr/maxplus2/synopsys/library/alt_mf/lib`  
(Library name: `altera` ; Package name: `maxplus2`)

# Using Altera Macrofunctions - Verilog

```
/* The port declarations of Altera Macrofunction
module A_81MUX (A, B, C, D0, D1, D2, D3, D4, D5, D6, D7, GN, Y, WN);
module A_8COUNT (A, B, C, D, E, F, G, H, LDN, GN, DNUP, SETN, CLRN, CLK,
                 QA, QB, QC, QD, QE, QF, QG, QH, COUT);
module A_8FADD (A8, A7, A6, A5, A4, A3, A2, A1, B8, B7, B6, B5, B4, B3, B2, B1, CIN,
              SUM8, SUM7, SUM6, SUM5, SUM4, SUM3, SUM2, SUM1, COUT);
module A_8MCOMP (A7, A6, A5, A4, A3, A2, A1, A0, B7, B6, B5, B4, B3, B2, B1, B0,
               ALTB, AEQB, AGTB, AEB7, AEB6, AEB5, AEB4, AEB3, AEB2, AEB1, AEB0);
*/

// An example to instantiate A_8COUNT function
input clear, clk;
output [7:0] out;
supply1 VDD;

A_8COUNT U1 (.A(), .B(), .C(), .D(), .E(), .F(), .G(), .H(),
             .LDN(VDD), .GN(), .DNUP(), .SETN(VDD), .CLRN(clear), .CLK(clk),
             .QA(out[0]), .QB(out[1]), .QC(out[2]), .QD(out[3]), .QE(out[4]),
             .QF(out[5]), .QG(out[6]), .QH(out[7]), .COUT());
```

# Using Altera Macrofunctions - VHDL

```
library IEEE;
use IEEE.STD_LOGIC.1164;
library ALTERA;
use ALTERA.MAXPLUS2.all;

-- An example using Altera-supplied macrofunction to create a complex 8-bit counter
-- (active-low clear, async and sync load & enable)

entity COUNTER3 is
  port (COUNT_OUT: out STD_LOGIC_VECTOR(7 downto 0);
        COUNT_IN: in STD_LOGIC_VECTOR(7 downto 0);
        CLK, UPDOWN, ENABLE, LOAD, CLEAR, SET: in STD_LOGIC);
end COUNTER3;

architecture STRUCTURE of COUNTER3 is
  signal LDN, GN, DNUP: STD_LOGIC;
begin
  LDN <= not LOAD;  GN <= not ENABLE;  DNUP <= not UPDOWN;
  U1: A_8COUNT port map
    (A => COUNT_IN(0), B => COUNT_IN(1), C => COUNT_IN(2), D => COUNT_IN(3),
     E => COUNT_IN(4), F => COUNT_IN(5), G => COUNT_IN(6), H => COUNT_IN(7),
     LDN => LDN, GN => GN, DNUP => DNUP, SETN => SET, CLRN => CLEAR, CLK => CLK,
     QA => COUNT_OUT(0), QB => COUNT_OUT(1), QC => COUNT_OUT(2), QD => COUNT_OUT(3),
     QE => COUNT_OUT(4), QF => COUNT_OUT(5), QG => COUNT_OUT(6), QH =>COUNT_OUT(7));
end STRUCTURE;
```

# Altera DesignWare Modules

- ◆ **Altera provides DesignWare interface to Synopsys for all FLEX device families**
  - Altera provides optimized support for the following HDL operators:
    - + , - , > , < , >= , <= , +1 , -1
    - Automatic access to FLEX carry & cascade chain functions
    - Optimal routing of FLEX designs
    - Improved area & performance prediction capability in Synopsys tools
- ◆ **Therefore, if you need adders, subtractors, and comparators, just use the above HDL operators instead of creating functions of gates**

# FLEX 10K RAMs & ROMs

- ◆ **FLEX 10K devices support EABs that can be used as memory**
  - EAB: 2Kb of RAM block
    - Can be configured as 2048x1, 1024x2, 512x4, or 256x8 RAM or ROM (without WE)
    - Input and output ports can be registered
- ◆ **Do not behaviorally describe your RAM functions**
  - Synthesis tools can't make difference between RAM and large register files
- ◆ **Do not behaviorally describe your ROM functions either**
  - Synopsys will translate your ROM descriptions into a large combinational circuit

# GENMEM Utility

- ◆ You can use Altera-provided `GENMEM` utility to create RAM/ROM models that you can instantiate in your HDL design files
  - `GENMEM` creates:
    - Functional simulation model
    - Library timing model
    - Component declaration (for VHDL users)
  - `GENMEM` supports the following RAM/ROM configurations:
    - Asynchronous RAM/ROM
    - Synchronous RAM/ROM
    - Cycle-shared FIFO
    - Cycle-shared dual-port RAM

# GENMEM Usage

Usage: `genmem memory_type memory_size [-vhdl] [-verilog] [-viewlogic] [-o]`

`memory_type` Specifies a valid memory type.  
memory\_type can be one of these values:

<code>ASYNRAM</code>	Asynchronous RAM
<code>ASYNROM</code>	Asynchronous ROM
<code>SYNRAM</code>	Synchronous RAM
<code>SYNROM</code>	Synchronous ROM
<code>CSFIFO</code>	Cycle-Shared FIFO
<code>CSDPRAM</code>	Cycle-Shared Dual-Port RAM

`memory_size` Specifies the size of the memory model. memory\_size consists of two values:

`word` Must be between 2 and 32768 (32k).  
`width` Must be between 1 and 32.

The word and width values must be separated by an "x". ( `<word>x<width>` )

<code>-vhdl</code>	Generate vhdl output(default)
<code>-verilog</code>	Generate verilog output
<code>-viewlogic</code>	Generate VHDL model for Viewlogic
<code>-o</code>	Overwrite the output file

Example: `genmem asynrom 256x15`

This example generates `asyn_rom_256x15.vhd`, `asyn_rom_256x15.cmp`, and `asyn_rom_256x15.lib`

# GENMEM Example - Verilog

```
unix> genmem synram 256x16 -verilog
```

```
GenMem -- Generate Memory Simulation Model Version 2.3
```

```
Copyright (C) 1996 Altera Corporation
```

	Sync.	Sync.	Sync.
RAM	Address	Input	Output
Type	Control	Data	Data
1	Yes	Yes	Yes
2	Yes	Yes	No
3	Yes	No	Yes
4	Yes	No	No
5	No	Yes	No
6	No	Yes	Yes
7	No	No	Yes

```
Select a RAM type number (1 ~ 7) or 8 to quit
```

```
2
```

```
unix> ls syn_ram_256x16*
```

```
syn_ram_256x16_irou.lib syn_ram_256x16_irou.v
```

*.lib : library timing model for synthesis*

*.v : Verilog functional simulation model*





# Verilog File Created by GENMEM

```
unix> more syn_ram_256x16_irou.v
```

```
module syn_ram_256x16_irou (Q, Data, WE, Address, Inclock);
```



port list

```
parameter LPM_FILE = "UNUSED";  
parameter Width = 16;  
parameter WidthAd = 8;  
parameter NumWords = 256;
```



**LPM\_FILE** parameter specifies RAM/ROM initialization file.

- It's optional for a RAM module
- It must be set to a Intel-Hex file (\*.hex) that defines memory content
- The memory file must reside in the current working directory

```
input [WidthAd-1:0] Address;  
input [Width-1:0] Data;  
input WE;  
output [Width-1:0] Q;  
input Inclock;
```

```
    :  
    :
```

```
if (LPM_FILE != "UNUSED")  
    begin
```



```
        $convert_hex2ver(LPM_FILE, Width, mem_initf);  
        $readmemh(mem_initf, mem_data);  
    end
```

## Note:

Verilog-XL does not support this system task by default. Please use another Verilog executable that can be get from CIC FTP site to do functional simulation.

CIC FTP site: ftp.cic.edu.tw (140.126.24.62)

File: /pub/cad/tools/Altera\_unix/verilog/bin/sunos/alt\_verilog (for SunOS 4.1.x)

File: /pub/cad/tools/Altera\_unix/verilog/bin/solaris/alt\_verilog (for Solaris 2.5)

# Instantiating RAM/ROM in Verilog

```
unix> more use_ram1.v
```

```
// An example to use syn_ram_256x16_irou module
```

```
module use_ram1(Q, ram_din, ram_wen, Address, clk, Count_En, Reset);
```

```
parameter LPM_FILE = "ramblk1.hex";
```

```
parameter Width = 16;
```

```
parameter WidthAd = 8;
```

```
parameter NumWords = 256;
```

```
input [WidthAd-1:0] Address;
```

```
input [Width-1:0] ram_din, Q;
```

```
input ram_wen, clk, Reset, Count_En;
```

```
reg [WidthAd-1:0] ram_addr;
```

```
:
```

```
:
```

```
syn_ram_256x16_irou
```

```
// synopsys translate_off
```

```
 #(LPM_FILE)
```

```
// synopsys translate_on
```

```
ramblk1 (.Q(Q), .Data(ram_din), .WE(ram_wen), .Address(ram_addr), .Inclock(clk));
```

```
:
```

```
endmodule
```



Specify LPM\_FILE parameter if necessary

## Note:

The filename must be the same as the the instance name



Use Synopsys directives to skip checking and translation for LPM\_FILE parameter when instantiate RAM/ROM block



# GENMEM Example - VHDL

```
unix> genmem synram 256x16 -vhdl
```

```
GenMem -- Generate Memory Simulation Model Version 2.3
```

```
Copyright (C) 1996 Altera Corporation
```

	Sync.	Sync.	Sync.
RAM	Address	Input	Output
Type	Control	Data	Data
1	Yes	Yes	Yes
2	Yes	Yes	No
3	Yes	No	Yes
4	Yes	No	No
5	No	Yes	No
6	No	Yes	Yes
7	No	No	Yes

Select a RAM type number (1 ~ 7) or 8 to quit

2

```
unix> ls syn_ram_256x16*
```

```
syn_ram_256x16_irou.cmp syn_ram_256x16_irou.lib syn_ram_256x16_irou.vhd
```

*.lib : library timing model for synthesis*  
*.vhd : VHDL functional simulation model*  
*.cmp : VHDL component declaration*



# VHDL File Created by GENMEM

```
unix> more syn_ram_256x16_irou.vhd
```

```
entity syn_ram_256x16_irou is
  --pragma translate_off
  generic ( LPM_FILE : string := "UNUSED" );
  --pragma translate_on
  port (  Data      : in std_logic_vector(15 downto 0);
        Address   : in std_logic_vector(7  downto 0);
        WE        : in std_logic;
        Q         : out std_logic_vector(15 downto 0);
        Inclock   : in std_logic
  );
  :
  :
end syn_ram_256x16_irou;
:
:
```



Synopsys directives are used to skip checking and translation for LPM\_FILE parameter when instantiate a RAM/ROM block

# Instantiating RAM/ROM in VHDL

```
unix> more use_ram1.vhd
```

```
architecture STRUCTURE of USE_RAM1 is
```

```
  component syn_ram_256x16_irou
```

```
    --pragma translate_off
```

```
      generic ( LPM_FILE : string );
```

```
    --pragma translate_on
```

```
    port ( Data      : in std_logic_vector(15 downto 0);
```

```
          Address    : in std_logic_vector(7  downto 0);
```

```
          WE         : in std_logic;
```

```
          Q          : out std_logic_vector(15 downto 0);
```

```
          Inclock   : in std_logic);
```

```
  end component;
```

```
begin
```

```
  ramblk1: syn_ram_256x16_irou
```

```
    --pragma translate_off
```

```
    generic map ( LPM_FILE => "ramblk1.hex" )
```

```
    --pragma translate_on
```

```
    port map (Data => DIN, Address => Addr, WE => WEN, Q => Q, Inclock => CLK);
```

```
end STRUCTURE;
```

Specify LPM\_FILE parameter if necessary

**Note:**

The filename must be the same as the the instance name



Use Synopsys directives to skip checking and translation for LPM\_FILE parameter when instantiate a RAM/ROM block

# Xilinx-Specific Issues

- ◆ Using Flip-Flops or Latches
- ◆ Designing Tri-State Buses
- ◆ Xilinx Primitives
- ◆ Xilinx DesignWare Modules

*Please refer to "Xilinx & Synopsys" training manual to know how to simulate a Xilinx-specific designs.*

# Using Flip-Flops or Latches

## ◆ Level-sensitive latch vs. edge-sensitive flip-flop

- In XC4000/E devices, only IOBs contain input latch
  - A latch can be built by combining the combinational logic in the CLB with feedback, and will result in combination loops and hold-time requirements
  - You may implement a simple latch (without reset/set or enable) by instantiating a `RAM16x1` primitive in which all address input pins are connected to ground
- In XC5200 and XC4000X Series devices, the storage elements in CLBs can be configured as flip-flops or latches individually
- Be careful when using HDL `case` or `if` statements
  - Specify all the branches in a `if` or `case` statement if possible

# Designing Tri-State Buses

## ◆ Internal tri-state buffers in Xilinx devices

- Xilinx FPGA devices have many internal tri-state buffers (BUFT)
  - You can infer tri-state buffers in HDL modeling and synthesis
- Multiplexer that are larger than 4-to-1 exceed the capacity of one CLB
  - You can build multiplexers that have one-hot encoded selector inputs by using internal tri-state buffers



# Xilinx Primitives

## ◆ Xilinx provides many primitives you can instantiate in your HDL design files

- Please refer to “Synopsys (XSI) Interface/Tutorial Guide” Appendix for detailed function descriptions of Xilinx primitives
- Note when instantiating Xilinx primitives:
  - Xilinx has I/O primitives that can be inferred or instantiated by Synopsys
  - Instantiating Xilinx primitives may not allow you to migrate between FPGA families
  - Not all primitives have simulation models

## ◆ In most circumstances, you do not need to instantiate Xilinx primitives expect for the following cases:

- Instantiate a registered-bidirectional I/O
- Instantiate the STARTUP symbol to utilize the GSR/GR net
- Use on-chip clock generator & clock buffers (OSC4, OSC5, BUFG, ...)
- Use readback and/or boundary-scan primitives
- Use other special functions of specific FPGA device architecture

# Registered Bidirectional I/Os

## ◆ Synopsys can handle various I/O configuration

- Synopsys can handle uni-directional I/O (registered or non-registered) and non-registered bidirectional ports well
  - For a non-registered bidirectional port, its 3-state signal that drives the output buffer must be described in the same hierarchy level as the input signal
- However, Synopsys cannot automatically infer the 3-state registered output buffer cells (`OFDT`, `OFDT_F`, ...) in a bidirectional I/Os
  - These cells and the `IBUF` must be instantiated into the top-level design
- Note: you must place a "dont\_touch" attribute on all instantiated I/O buffers before the design is compiled by Synopsys

# Dedicated GSR & GTS Nets

## ◆ XC4000 devices provides special GSR & GTS nets

- All XC4000 device families have a dedicated Global Set/Reset (**GSR**) net that initialize all CLBs and IOBs
  - This function of GSR net is separate from the individual preset and clear pin
  - When GSR is asserted, every flip-flop in the FPGA is simultaneously preset or cleared
- Besides, a dedicated Global Tri-state (**GTS**) net is supported to force all of the IOB outputs to high-impedance mode
- You do not need to use general purpose routing resources to connect to the preset or clear pins of the flip-flops, or to force all of the IOB outputs to high-impedance mode
  - Use the GSR and GTS nets to increase performance and reduce routing congestion
- You can access the GSR or GTS nets from the GSR or GTS pin on the **STARTUP** primitive

## ◆ XC5200 devices also provides special GR & GTS nets

- Similar to XC4000 GSR/GTS nets, XC5200 devices provide a Global Reset (**GR**) and **GTS** nets on the **STARTUP** primitive

# Clock Generators

## ◆ XC4000 & XC5200 devices provides on-chip clock generators

- XC4000 devices provide internal clock generators (`osc4`) to provide internal clock signals in applications where timing is not critical
  - Frequency: 8M, 500K, 16K, 490, 15Hz
- XC5200 devices also provide internal clock generators (`osc5`)
  - "`DIVIDE1_by`" or "`DIVIDE2_by`" attributes are required to set

# Global Buffers

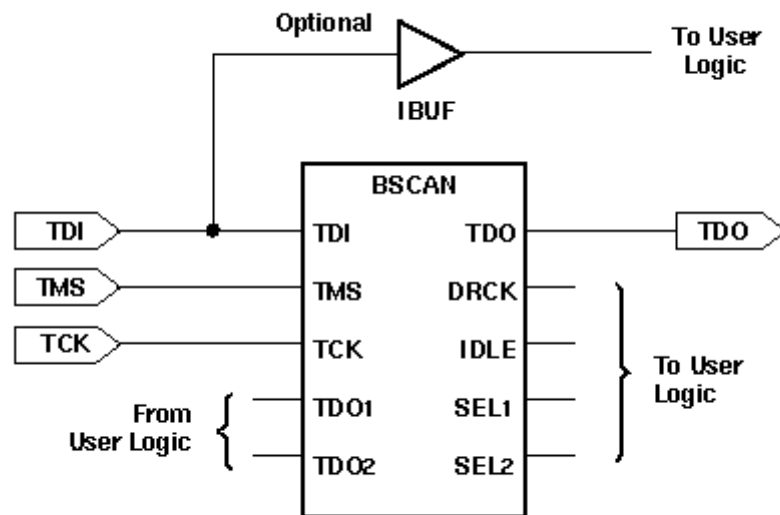
## ◆ You can manually connect internal clock signal to a global buffer provided by XC4000 & XC5200 devices to reduce clock skew

- Each XC4000E device contains 8 global buffers
  - 8 `BUFG_F` (= 4 `BUFGP_F` + 4 `BUFGS/BUFGS_F`)
- Each XC4000X device contains more fast buffers
  - 8 `BUFGLS/BUFGE`, 4 `BUFFCLK`
  - `BUFG_F`, `BUFFGS`, `BUFG_F` primitives still can be used
- Each XC5200 device contains 4 global buffers
  - 4 `BUFG/BUFG_F`

# Boundary Scan

## ◆ XC4000 and XC5200 devices contain boundary-scan facilities

- XC4000/E and XC5200 devices contain boundary-scan facilities that are compatible with IEEE Standard 1149.1
- You can instantiate the boundary-scan symbol, **BSCAN**, and the boundary scan I/O pads, **TDI**, **TMS**, **TCK** and **TDO** to access the boundary scan logic after configuration
  - Activation of the boundary-scan logic is part of the design process
- Do not use Synopsys tools to insert boundary-scan logic because they do not work with FPGA devices



# Using Dedicated I/O Decoders

## ◆ XC4000 devices provide dedicated I/O decoders (edge decoders)

- The periphery of the XC4000 devices has 4 wide decoder circuits at each edge (2 in XC4000A devices)
- The inputs to each decoder are any of the IOB signals on that edge (using `DECODE1_IO`) plus one local interconnect per CLB row or column (using `DECODE1_INT`)
  - Each decoder generates a high output (using a pull-up resistor, `PULLUP`) when the AND condition of the selected inputs or their complements is true
- Edge decoder and pull-up resistor primitives include
  - `DECODE1_IO`
  - `DECODE1_INT`
  - `DECODE4`, `DECODE8`, `DECODE16`
  - `PULLUP`

# Memory Primitives

- ◆ **XC4000 device families can efficiently implement RAM & ROM using CLB functional generators**
  - You can instantiate 16x1 and 32x1 RAM and ROM primitives
    - Asynchronous RAM: `RAM16X1`, `RAM32X1`
    - Synchronous RAM: `RAM16X1S`, `RAM32X1S`
    - Dual port RAM: `RAM16X1D`
    - ROM: `ROM16X1`, `ROM32X1` (must add ROM value)
  - You can use LogiBLOX utility to create RAM and ROM netlist instead of using primitives
    - Easier, faster, and more flexible
    - Refer to “Xilinx & Synopsys” training manual for details



# Simulation Models of Primitives

## ◆ Simulation models have been included in the software

- Verilog simulation model for Xilinx's primitives:

XC4000E: `/usr/xilinx/cadence/data/verilogxce4000e/*.v`

XC4000X: `/usr/xilinx/cadence/data/verilogxce4000ex/*.v`

XC5200: `/usr/xilinx/cadence/data/verilogxce5200/*.v`

XC3000: `/usr/xilinx/cadence/data/verilogxce3000/*.v`

- VHDL source files: `<family>_FTGS.vhd` and `<family>_components.vhd` under the directory

`/usr/xilinx/synopsys/libraries/sim/src/<family>/ftgs`

- `<family>` includes: xc3000a, xc4000e, xc5200

- Synopsys VSS library directory:

`/usr/xilinx/synopsys/libraries/sim/lib/<family>/ftgs`

(Library name: `<family>`; Package name: `components`)

- `<family>` includes: xc3000a, xc4000e, xc5200

# Using Xilinx Primitives - Verilog

```
// Instantiating a registered bidirectional I/O on XC4000/E IOB
// OFDT_F: 3-state registered output buffers with a fast slew rate
// IBUF : input buffer
OFDT_F U0 (.D(outa), .C(clock), .T(loada), .O(siga));
IBUF UI0 (.I(siga), .O(ina));

// Instantiating STARTUP symbol
STARTUP U1 (.GSR(gsr_in));

// Instantiate OSC4 on-chip clock generator - 15Hz and a global buffer for the clock
OSC4 U2 (.F15(f15_out));
BUFGS U3 (.I(f15_out), .O(clk_f15));

// Instantiate BSCAN symbol
BSCAN U4 (.TDI(tdi_sig), .TMS(tms_sig), .TCK(tck_clk), .TDO(tdo_sig));
TDI U5 (.I(tdi_sig));
TMS U6 (.I(tms_sig));
TCK U7 (.I(tck_sig));
TDO U8 (.O(tco_sig));

// Instantiate XC4000/E edge decoders
ADR_INV = ~ADR;
DECODE4 A1 (.A3(adr[3]) , .A2(adr[2]), .A1(adr[1]), .A0(adr_inv[0]), .O(dec_out[0]));
DECODE1_IO A2 (.I(adr_inv[4], .O(dec_out[0]));
PULLUP A3 (.O(dec_out[0]));
```

# Using Xilinx Primitives - VHDL

```
library XC4000E;
use XC4000E.COMPONENTS.all;
...
-- Instantiating a registered bidirectional I/O on XC4000/E IOB
-- OFDT_F: 3-state registered output buffers with a fast slew rate; IBUF: input buffer
U0 : OFDT_F port map (D => OUTA, .C => CLOCK, T => LOADA, O => SIGA);
UI0 : IBUF port map (I => SIGA, .O => INA);

-- Instantiating STARTUP symbol
U1: STARTUP port map (GSR => gsr_in);

-- Instantiate OSC4 on-chip clock generator - 15Hz and a global buffer
U2: OSC4 port map (F15 => f15_out);
U3: BUFGS port map (I => f15_out, O => clk_f15);

-- Instantiate BSCAN symbol
U4: BSCAN port map (TDI => TDI_SIG, TMS => TMS_SIG, TCK => TCK_CLK, TDO => TDO_SIG);
U5: TDI port map (I => TDI_SIG);
U6: TMS port map (I => TMS_SIG);
U7: TCK port map (I => TCK_SIG);
U8: TDO port map (O => TDO_SIG);

-- Instantiate XC4000/E edge decoders
ADR_INV <= not ADR(4 downto 0);
A1: DECODE4 port map
    (A3 => ADR(3), A2 => ADR(2), A1 => ADR(1), A0 => ADR_INV(0), O => DEC_OUT(0));
A2: DECODE1_IO port map (I => ADR_INV(4), O => DEC_OUT(0));
A3: PULLUP port map (O => DEC_OUT(0));
```

# Xilinx DesignWare Modules

- ◆ Xilinx provides DesignWare interface to Synopsys for all XC4000 & XC5200 device families (XDW library)
  - Xilinx provides optimized support for the following HDL operators:  
+, -, >, <, >=, <=, +1, -1
    - Automatically utilize carry logic to improve both the area & speed
    - Optimal implementation for specific device family
    - The XDW library contains area and speed information for its modules
    - X-BLOX modules are synthesized in the design implementation stage
- ◆ Therefore, if you need adders, subtractors, and comparators, just use the above HDL operators instead of creating functions of gates

# XDW Module Implementation

- ◆ **XC4000 devices accommodate 2 bits fo arithmetic function per CLB, and XC4200 devices accommodate 4 bits per CLB**

- In XC4000 devices, arithmetic functions are implemented in one vertical column of CLBs
  - The carry propagation direction is upward in XC4000X devices, and up or down in XC4000E devices
- In XC5200 devices, arithmetic functions are implemented in two vertical columns of CLBs
  - The carry propagation direction is upward

- ◆ **In general, choose a target device that can accommodate the “tallest” arithmetic structure in your design**

- For example, if you want fastest implementation of a 33-bit two’s complement adder, you should select XC4008E or larger part type
  - XC4006E device contains 16 CLBs per column that can only implement 32-bit unsigned adders a column. Using a XC4008E (18 CLBs per column) or larger device would give you the fastest implementation since the adder would not have to wrap into the next column

# Summary

## ◆ Why programmable logic?

- In-circuit verification
- Shorter time-to-market
- Reduced risk
- ...

## ◆ Why HDL?

- Descriptive
- IEEE standardized & portable
- Success of synthesis tools
- ...

## ◆ What to consider?

- Cost: unit cost, development time, resource utilization, performance, ...
- HDL coding style: synthesis subset, technology independent or dependent, efficiency for synthesis, hierarchy, FPGA family-specific issues, ...

# What's Next?

## ◆ Now, you've prepared a HDL design file

- Go to functionally simulate it
- Use FPGA Compiler to synthesis it
- Use FPGA vendor tools to P&R the design
- Program the device on system

## ◆ Now, you can continue the learning process from the following training course:

- Xilinx-Synopsys Design Flow Training Course
  - For Xilinx users, learn FPGA Compiler and Xilinx tools
- Altera-Synopsys Design Flow Training Course
  - For Altera users, learn FPGA Compiler and Altera tools

# Recommended Reference Manuals


## ◆ Synopsys manuals

- HDL Compiler for Verilog Reference Manual
- VHDL Compiler Reference

 Synopsys on-line document


## ◆ Altera manuals

- Synopsys & MAX+PLUS II Software Interface Guide
- Altera/Synopsys User Guide

 as\_sig.pdf and as\_ug.pdf on Internet

## ◆ Xilinx manuals

- Quick Start Guide for Xilinx Alliance Series
- Synopsys (XSI) Interface/Tutorial Guide

 Xilinx on-line books

## ◆ CIC Training Manuals

- VHDL Training Manual
- Verilog Training Manual



# Getting Help

## ◆ CIC technical support for FPGAs: 林岡正

- Phone : (03)5773693 ext. 146
- Email : max@mbox.cic.edu.tw
- News : nsc.cic
- ftp-site : ftp://ftp.cic.edu.tw/pub (140.126.24.62)
- WWW : http://www.cic.edu.tw



**Thank You!**