Curriculum Path for ASIC Design

- Introduction to VHDL or Introduction to Verilog (3-day course)
- FPGA and ASIC Technology Comparison (Recorded e-Learning)
  & FPGA Versus ASIC Design Flow (Recorded e-Learning)
  & ASIC to FPGA Coding Conversion (Recorded e-Learning)
- Spartan-3 Architecture Overview (Recorded e-Learning)
- Fundamentals of FPGA Design (1-day course)
- Designing for Performance (2-day course)
- Advanced FPGA Implementation* (2-day course)

*At least 6 months' design experience recommended prior to taking this course.
FPGA and ASIC Technology Comparison
Objectives

After completing this module, you will be able to:

- Describe the differences between ASIC and FPGA architectures, and describe how these differences affect coding style, implementation, and product selection
  - Gate conversion
  - Delays
  - Frequency comparison
- Discuss reconfigurability
Outline

- Architecture
- Performance
- Gate Count
- Reconfigurability
- Summary
- Appendix: Cost Estimator
Contrasting Architectures

- ASIC architecture compared to the Xilinx FPGA architecture
  - Gates versus LUTs
  - Delays
  - Performance
- Fundamental part selection considerations:
  - Cost
  - Size
  - Performance
  - Volume
  - Analog circuitry
  - Time to market
  - Reprogrammability
Standard Cell

- **Advantages:**
  - Lowest price for high volume production (greater than 200k per year)
  - Fastest clock frequency (performance)
  - Unlimited size
  - Integrated analog functions
    - Custom ASICs
    - Low power
- **Disadvantages:**
  - Highest NRE
  - Longest design cycle
  - Limited vendor IP with high cost
  - High cost for Engineering Change Order (ECO)
Embedded Array

- Advantages:
  - Low price for medium to high volume production
  - Performance only slightly slower than standard cell
  - 50+ million gates
  - Allows custom macros
  - More flexibility than FPGA
  - Low power

- Disadvantages:
  - High NRE
  - Design cycle longer than an FPGA
  - Vendor IP has high cost
  - Generally digital only
**Xilinx Field-Programmable Gate Arrays**

- **Advantages:**
  - Lowest cost for low to medium volume production
  - No NRE
  - Standard Product
  - Fastest time to market
  - Xilinx has extensive library of IP
    - Inexpensive compared to ASIC vendors
  - Ability to make bug fixes quickly and inexpensively
- **Disadvantages:**
  - Slower performance
  - Size limited ~10 million system gates
  - Digital only
Field-Programmable Gate Array Introduction

- Xilinx FPGAs are made using SRAM
- Today's FPGAs use 90-nm nine-layer metal copper process
- The largest FPGA today can accommodate approximately 10 million system gates
  - Includes RAM and logic gates
- Performance up to 420 MHz
- Integrated synthesis, simulation, and place & route tools
  - PC and UNIX
  - Inexpensive - typically $30k or less for entire tool sets*
    - * Includes third-party tools
Xilinx FPGAs are made of four primary elements:

- Configurable logic blocks
- Memory
- Input and output blocks
- Routing
Logic Cells

- Logic cells include:
  - Combinatorial logic, arithmetic logic, and a register
  - Combinatorial logic is implemented by using look-up-tables
  - Register functions may include latches, JK, SR, D, and T-type flip-flops
  - Arithmetic logic is a dedicated carry-chain for implementing fast arithmetic operations
Combinatorial Logic

- Look-up-tables (LUTs) function as a ROM, and the LUTs look up the resulting output based on the input signals
- Constant delay through LUT
- Limited by the number of inputs and outputs, not by complexity
Wide Input Functions

- For wider input functions, the LUTs can be combined by using a logic gate or multiplexer
LUT-Based Memory

- The LUTs can often be used to store 16 bits of memory as either a RAM or a ROM
- Fundamentally, the LUT is a ROM
- The LUT can become RAM by activating the configuration write strobe
- Multiple LUTs can be combined to create larger memories both in depth and width
  - 128 x 8 is not uncommon

NOTES

Sixteen bits of RAM storage in a LUT is in the depth. For example, each LUT can be configured as a 16 x 1 RAM.
Carry Logic

- The carry logic chain is dedicated logic that computes high-speed arithmetic logic functions.
- The carry chain generally consists of a multiplexer and an XOR gate:
  - The LUT computes the multiplexer selector.
  - The multiplexer determines the carry-out.
  - The XOR gate computes the addition.
Memory Blocks

- These memory blocks support single- and dual-port synchronous operations
- In dual-port mode, these RAM blocks support fully independent ports for both reading and writing
- Sizes up to 18K bits
- The blocks of memory are generally spread out across the die
The input and output blocks generally consist of a register and buffer for each path.

The output buffer can also be used as a three-state buffer.

Each path can be combinatorial or registered.

The default standard is LVTTL:
- Fast and slow slew rates
- Programmable drive strength
- Other thresholds are discussed in the following slides.
Routing

- The routing is generally a combination of both programmable and dedicated routing lines
- Dedicated routing:
  - Global clocks with predefined clock tree
  - Global asynchronous set and reset
  - Global low-skew routing resources for other high-fanout signals
  - Carry chain routing
  - Dedicated routing between logic blocks
- General interconnect:
  - Routing of local signals from one logic block to another
Dedicated and Special Resources

- Clock management
  - DCM
  - Dedicated clock trees
- Test Logic
  - Built-in JTAG
- I/O translators
  - Supporting many different thresholds
- Other Resources
  - Embedded processors
  - Gigabit transceivers
  - Dual-Data Rate (DDR) registers in IOB
Clock Management

- The dedicated clock trees are pre-optimized clock networks that balance the skew and minimize delay
  - Virtex-II device has 16 separate clock networks
  - Spartan-3 device has 8 separate clock networks
- Advanced clock management:
  - DCM (Digital Clock Manager) consists of:
    - DLL (Digital delay locked loop)
    - DFS (Digital Frequency Synthesis)
    - DPS (Digital Phase Shifter)
I/O Translators

- Programmable input and output thresholds
- The supported standards include:
  - LVTTL, LVCMOS, LVCMOS2, AGP, HSTL (and several classes), SSTL (and several classes), PCI, PCI-X, LVDS, LVPECL, GTL, GTL+, and CTT
- Different I/O standards require a separate input and output reference voltage for each bank supporting a separate I/O standard
- Generally, each bank can support several standards, as long as they share the same vref (input) or vcco (output)
Other Resources

- Embedded processor cores (soft and hard)
  - 32-bit PowerPC processor core (hard)
  - MicroBlaze processor core (soft)
- Digitally controlled termination resistance (DCI)
- Dedicated multipliers
- Dedicated DDR I/O registers
Future FPGA Enhancements

- Mixed digital and analog
- Mixed ASIC and FPGA blocks
- Memory BIST
- 15 million gates by 2004
Outline

- Architecture
- Performance
- Gate Count
- Reconfigurability
- Summary
- Appendix: Cost Estimator
AND-Gate Example

- Eight-input and-gate

**VHDL** for vec(7 downto 0)
and_out <= vec(0) AND vec(1) AND vec(2) AND vec(3)
 AND vec(4) AND vec(5) AND vec(6) AND vec(7);

**Verilog** for vec[7:0]
assign and_out = & vec;
ASIC Implementation

- Eight-input and-gate
  - Two four-input NAND gates feeding a two-input NOR gate

Approximate delay in a standard-cell ASIC with .13 µ process = .47 ns
Approximate gate count = 14
Xilinx Implementation

- Implemented in three four-input Look Up Tables (LUTs)

Approximate max delay in a Virtex -II -6 device = 0.7 ns Approximate gate count = 18 gates
Registered I/O

VHDL
process (clk)
begin
if rising_edge(clk) then
  vec_q <= vec;
  and_out <= vec_q(0) AND vec_q(1) AND vec_q(2) AND vec_q(3) AND vec_q(4) AND vec_q(5) AND vec_q(6) AND vec_q(7);
end if;
end process;

Verilog
always @ (posedge clk)
begin
  vec_q <= vec;
  and_out <= & vec_q;
end
Performance Comparison

- A comparison of the achieved performance for the registered eight-input and-gate:
  - Xilinx Virtex-II -6 device
    - ~420 MHz
    - ~88 Gates
  - .13 µ standard cell ASIC
    - ~850 MHz
    - ~77 Gates

- Typical high-performance frequencies:
  - Xilinx Virtex-II -6
    - ~250 MHz for four-levels of LUT (combinatorial) logic
  - .13 µ standard cell ASIC
    - ~550 MHz for equivalent logic
ASIC Versus FPGA

- Combinatorial logic implemented in an ASIC is typically faster than in an FPGA implementation
  - ASIC's fine-grain architecture allows wider input functions to be implemented with significantly less delay
  - ASICs have a dedicated routing structure rather than a programmable routing structure
- Critical paths typically include I/O, RAM, PCI, and DSP resources
  - Xilinx has dedicated FPGA resources to implement these functions, making these paths equivalent to an ASIC implementation
    - Remember: the Xilinx Virtex -II and Spartan -3 FPGAs are a cutting-edge ASIC

Virtex-based FPGAs: Virtex, Virtex-E, Virtex-EM, Spartan -II
Synchronous Design

- The essence of achieving performance in a Xilinx device is using synchronous design techniques.
- Xilinx has a *register-rich* architecture that accommodates synchronous circuits.
  - For Xilinx FPGAs, the resources exist on the chip.
- For combinational logic paths, FPGAs generally cannot achieve the frequencies that are possible in a custom ASIC.
  - However, code optimization for Xilinx will increase performance.
Coding Style

- How do you get high-performance out of an FPGA?

- The programmability of the FPGA inherently makes it slower than an ASIC architecture

- Coding style has a large impact on the performance
  - Because FPGAs combinatorial and routing resources are inherently slower, the coding style needs to be more stringent
  - Write your code to limit the number of logic levels inferred
Sequential Design

- How do you get high performance out of an FPGA?

- Pipelining
  - For large combinatorial paths, additional registers may need to be inferred to break up combinatorial paths to increase performance

- Timing Constraints
  - Proper timing constraints need to be added to constrain multi-cycle paths, false paths, and to communicate the performance goals to the implementation tools
Optimize Combinatorial Paths

- How do you get high performance out of an FPGA?

- Evaluate combinatorial paths
  - Comparators often can be replaced with AND-OR logic for a faster implementation
  - Large multiplexers may get a higher performance implemented with three-states
  - Use one-hot encoded state machines for higher performance
  - Instantiate cores from the CORE Generator system or instantiate primitives
    - These are pre-optimized for the Xilinx architecture
  - Use one-hot, Johnson, pre-scaled, or LFSR counters
Outline

- Architecture
- Performance
  - **Gate Count**
  - Reconfigurability
- Summary
- Appendix: Cost Estimator
In re-targeting ASIC code to Xilinx FPGAs, gate conversion is rarely one:one.

- A .13μm standard cell can have up to 100K gates per mm², an FPGA has about 10K usable gates per mm².
- Why the difference?
Why the difference?

Xilinx has programmable logic in addition to the functional logic
  - Routing
  - Multiplexers
  - Configuration memory registers
  - Etc.
Virtex-II Slice

- Can you identify the usable resources in this slice?
  - Configuration logic
  - LUTs
  - Registers
Virtex-II Slice

- Configuration logic
- Registers
- Look-up tables

Data In Multiplex Logic
Gate Translation

- Separate out logic, RAM, cores, and I/O
  - Partition cores into logic and RAM
- Assume:
  - Six to twelve gates per LUT (four-input LUT)
  - RAM bits are equivalent
  - Up to 100 ASIC gates per I/O; translate to IOBs
  - Seven gates per register
Example

- **ASIC**
  - 250K logic gates
- Four 16KB blocks of RAM
- 243 pads, including power and ground

- **FPGA**
  - 20,833 to 41,666 LUTs
  - Equivalent
  - Equivalent number of IOBs

This could require a Virtex-II 2000 device or up to a Virtex-II 4000 device
Gate Counts

Gate counts are influenced by:
- Coding style
- Metal layers
- Process geometry
- Library quality
- Placement and routing algorithms
- Core contents (RAM versus gates)
- I/O requirements
- special features used

Conclusion:
ASIC-to-FPGA gate counting has few common denominators. Taking ASIC code directly to an FPGA will not use the advantages of the FPGA.
What Happened?

- A design that is not optimized for an FPGA will not take advantage of the architectural resources of the FPGA
- The design needs to be re-targeted for a Xilinx FPGA
  - DCM and DLL
  - SRL (Shift Register LUT) instead of registers
  - Distributed RAM
  - Block RAM
  - Comparators
  - Coding style
  - Cores
  - Pipelining
  - Three-state buffers
  - Clock enables

NOTES

Optimizing code for an FPGA will be discussed in the following section: ASIC to FPGA Coding Conversion.

The Xilinx system gate count assumes that you use a large portion of the available block RAMs and approximately 20 percent of the logic resources as distributed RAM.
Outline

- Architecture
- Performance
- Gate Count
- Reconfigurability
- Summary
- Appendix: Cost Estimator
Remote Configuration
Xilinx IRL

- Remote configuration can be accomplished through the use of any network - Internet Reconfigurable Logic (IRL)
- Cost of ownership is reduced with the ability to reconfigure the hardware, therefore extending the life of the product
  - Reduces the costly physical deployment of repair technicians
  - Extends the life of the product
    - Upgrades
    - Bug fixes
    - Adding additional functionality
    - Faster time to market
    - Partial reconfiguration
Remote Upgrades

- This capability needs to be taken into account at the beginning of the design cycle
  - The board needs to be designed to allow remote upgrades
    - A microprocessor or CPLD can control reconfiguration of the part directly or by reloading the data into flash or EEPROM memory
- One of the most common reasons for using an FPGA is faster time to market, and IRL enhances that capability
  - Remotely upgrading the hardware gives you faster time to market
    - For standards in the state of fluctuation, the hardware can be deployed with the most current known standard and upgraded remotely at a later date as the standard becomes static
Remote Functional Changes

- Bug fixes and additional functionality can now be accomplished electronically by transmitting a new bitstream over a network.

- The hardware can be completely reconfigured or partially reconfigured.
  - For more information about partial reconfiguration, see the following Xilinx application notes on http://support.xilinx.com:
    - XAPP 151
    - XAPP 153
    - XAPP 216
    - XAPP 290
Partial Reconfiguration

- Partial reconfiguration allows users to change functionality, add functionality, or both to the FPGA during operation
  - Changes are specified for specific areas on the die
  - Does not otherwise change the in-circuit operation
- Most Virtex-based devices provide the capability of partial reconfiguration

- Reconfiguration is done on a frame by frame basis
  - A frame primarily consists of a column in the FPGA
Outline

- Architecture
- Performance
- Gate Count
- Reconfigurability
- Summary
- Appendix: Cost Estimator
Review Questions

- Describe why the performance of ASIC combinatorial resources is different than that of an FPGA.

- Why are the Xilinx dedicated resources as fast as equivalent ASIC resources?

- In what ways can code be optimized to take advantage of the Xilinx architectural resources?

- What advantages does reconfigurability have, besides quick turnarounds for bug fixes?
Answers

- Describe why the performance of ASIC combinatorial resources is different than that of an FPGA
  - ASICs have a fine-grain architecture and a dedicated routing structure

- Why are the Xilinx dedicated resources as fast as equivalent ASIC resources?
  - Because Xilinx Virtex-based FPGAs are at the cutting edge of ASIC technology
Answers

- In what ways can code be optimized to take advantage of the Xilinx architectural resources?
  - Synchronous design, pipelining, optimizing code for Xilinx resources, use of DCM, block RAM, distributed RAM, cores, three-states, and clock enables

- What advantages does reconfigurability have besides quick turnarounds for bug fixes?
  - Faster time-to-market
  - Extends the life of the product through IRL
Summary

- Converting HDL code from targeting ASIC technology to targeting a Xilinx FPGA architecture is rarely as simple as re-targeting the code.
- Xilinx combinatorial resources use flexible LUTs.
- Xilinx has dedicated resources for arithmetic logic, RAM, PCI, and I/O that make these critical paths equivalent to a custom ASIC.
- Reconfigurability reduces the cost of ownership.
- FPGA Flexibility:
  - Remotely reconfigurable
  - Time to market
  - Lowest “cost-of-change”
Where Can I Learn More?

- Xilinx online documents
  - http://support.xilinx.com/ → Documentation
    - Software manuals
    - Data sheets
    - Application notes

- Xilinx Education Services courses
  - http://support.xilinx.com/ → Education
    - Xilinx tools and architecture
    - Hardware description languages

- IRL
  - http://support.xilinx.com → Products → System Resources → Design for Upgradability → PAVE
Curriculum Path for ASIC Design

- Introduction to VHDL or Introduction to Verilog (3-day course)
- FPGA and ASIC Technology Comparison (Recorded e-Learning)
- FPGA Versus ASIC Design Flow (Recorded e-Learning)
- ASIC to FPGA Coding Conversion (Recorded e-Learning)
- Spartan-3 Architecture Overview (Recorded e-Learning)
- Fundamentals of FPGA Design (1-day course)
- Designing for Performance (2-day course)
- Advanced FPGA Implementation* (2-day course)

*At least 6 months' design experience recommended prior to taking this course.
Please provide us your feedback on this module
- http://www.support.xilinx.com/support/training/eval-asic.htm

To complete the evaluation later and go directly to the lab, go to
Outline

- Architecture
- Performance
- Gate Count
- Reconfigurability
- Summary
- Appendix: Cost Estimator
Breakeven Analysis

- The ASIC Cost Estimator (ACE) allows you to enter critical information about your design, the market, etc., to estimate the breakeven point for choosing an ASIC or an FPGA

- (Price x Volume) does not take into account time-to-market and extended product life (IRL)

- Xilinx ASIC Cost Estimator white paper, tutorial, and downloadable program can be found at
# Total Cost Includes Ownership

<table>
<thead>
<tr>
<th>Balance Sheet Costs</th>
<th>Ownership Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRE</td>
<td>Time-to-market</td>
</tr>
<tr>
<td>Unit Price</td>
<td>Solution Risk</td>
</tr>
<tr>
<td>Tools Expense</td>
<td>Inventory Risk</td>
</tr>
<tr>
<td>Pre-Production Parts</td>
<td>Upgradability</td>
</tr>
<tr>
<td>Engineering Time</td>
<td>Cost Reduction Potential</td>
</tr>
<tr>
<td>Expedite Fees</td>
<td>Reliability</td>
</tr>
<tr>
<td>Cost of Upgrades</td>
<td></td>
</tr>
</tbody>
</table>
The Base Equation

Measures Cost

\[
\text{BALANCE SHEET COSTS} + \text{OWNERSHIP COSTS} + (\text{UNIT VOLUME} \times \text{PRICE}) = \text{TOTAL PROJECT COST}
\]
Delayed market entry results in lost revenue

\[
\text{Lost revenue} = \frac{(\text{delay}(3W-\text{delay})/2W^2)(100 \text{ percent})}{\quad}
\]

This assumes that development starts at the beginning of the market window and that being late in a competitive market will reduce potential revenue even if ramp is on the same slope. This also assumes that the FPGA can get to market at the beginning of market window and the ASIC will require a longer design cycle.

Sources: Synopsys Inc. and McKinsey and Co.
# Xilinx ASIC Cost Estimator: Parameters

<table>
<thead>
<tr>
<th>Project Information</th>
<th>FPGA Parameters</th>
<th>ASIC Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Size (K)</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>Purchase Type</td>
<td>PDKs</td>
<td>25</td>
</tr>
<tr>
<td>Chip Estim (K)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Logic Estim (K)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Required Core</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Period (Years)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Annual Sales Forecast</td>
<td>125K</td>
<td>125K</td>
</tr>
<tr>
<td>FTE Value</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Total Value</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Product Adoption</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Product Introduction</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPGA Parameters</th>
<th>ASIC Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engineering Cost per MCM</td>
<td>$500</td>
</tr>
<tr>
<td>Target FPGA</td>
<td>265K</td>
</tr>
<tr>
<td>FPGA Price ($)</td>
<td>265</td>
</tr>
<tr>
<td>FTE Value</td>
<td>500</td>
</tr>
<tr>
<td>ASIC Logic Value</td>
<td>500</td>
</tr>
<tr>
<td>FTE Value</td>
<td>500</td>
</tr>
<tr>
<td>ASIC Design Value</td>
<td>500</td>
</tr>
<tr>
<td>Sales Price ($)</td>
<td>265K</td>
</tr>
<tr>
<td>ASIC Logic Design</td>
<td>500</td>
</tr>
<tr>
<td>Sales Price ($)</td>
<td>265K</td>
</tr>
<tr>
<td>ASIC Cost ($)</td>
<td>500</td>
</tr>
<tr>
<td>ASIC Sales Price</td>
<td>265K</td>
</tr>
<tr>
<td>ASIC Logic Design</td>
<td>500</td>
</tr>
<tr>
<td>ASIC Logic Design</td>
<td>500</td>
</tr>
<tr>
<td>ASIC Logic Design</td>
<td>500</td>
</tr>
</tbody>
</table>
# Design Resources: Parameters

<table>
<thead>
<tr>
<th>FPGA DEVELOPMENT TIME RESOURCES (in weeks)</th>
<th>ASIC DEVELOPMENT TIME RESOURCES (in weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Authoring:</strong></td>
<td><strong>Design Authoring:</strong></td>
</tr>
<tr>
<td>- VHDL Development</td>
<td>- VHDL Development</td>
</tr>
<tr>
<td>- Functional Simulation</td>
<td>- Functional Simulation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Timing Closure:</strong></td>
<td><strong>Timing Closure:</strong></td>
</tr>
<tr>
<td>- Synthesis</td>
<td>- Synthesis</td>
</tr>
<tr>
<td>- Floor Planning</td>
<td>- Floor Planning</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Final Verification:</strong></td>
<td><strong>Final Verification:</strong></td>
</tr>
<tr>
<td>- Test Development</td>
<td>- Test Development</td>
</tr>
<tr>
<td>- Static Timing</td>
<td>- Static Timing</td>
</tr>
<tr>
<td>- Power Analysis</td>
<td>- Power Analysis</td>
</tr>
<tr>
<td>- Full Timing Simulation</td>
<td>- Full Timing Simulation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FPGA Qualification:</strong></td>
<td><strong>FPGA Qualification:</strong></td>
</tr>
<tr>
<td>- Other</td>
<td>- Other</td>
</tr>
<tr>
<td>- Total Time</td>
<td>- Total Time</td>
</tr>
</tbody>
</table>

- VHDL Development
- Functional Simulation
- Timing Closure
- Synthesis
- Floor Planning
- Final Verification
- Test Development
- Static Timing
- Power Analysis
- FPGA Qualification
- Other
- Total Time

- ASIC Development
- Functional Simulation
- Timing Closure
- Synthesis
- Floor Planning
- final Verification
- Test Development
- Static Timing
- Power Analysis
- FPGA Qualification
- Other
- Total Time
Gate array models will have different NRE scales and different time-to-market.
The cost for an FPGA solution, considering the total cost of ownership, is $2,931,192.

The cost for an FPGA solution, considering only the parts and development cost, is $3,400,000.

The cost for an ASIC solution, considering only the parts and development cost, is $944,999.

The cost for an ASIC solution, considering the total cost of ownership, is $8,681,146.
Breakeven Analysis Graph

This graph shows the volume at which it is more cost-effective to use an FPGA instead of an ASIC. The calculation considers parts and development cost PLUS the total cost of ownership (lost market share, inventory costs, re-spin). Including parts and development cost and the total cost of ownership, an FPGA solution is cost-effective if the volume < 44,230 units.

This graph shows the volume at which it is more cost-effective to use an FPGA instead of an ASIC. The calculation considers parts and development cost but NOT the total cost of ownership. Including only the parts and development cost, an ASIC solution is more cost-effective than an FPGA.
Computing Costs Yourself

- Percent likelihood of ECO
  - Re-spins in the ASIC industry occur about 30 percent of the time
  - *Expedite fees are often paid to expedite the ASIC process*

- Cost Reductions
  - ASIC prices reduce each year by about 3 percent
  - FPGA prices reduce each year by about 12 percent

- Upgrade and bug fixes
  - Future developments, such as enhancements and bug fixes, can occur at a very high rate
    - *Depends largely on the market, fluctuating standards, and competition*

- Lead times, inventory costs, expedite fees, risk
TTM Considerations

Delayed Market Model
Maximum Available Revenue

Market Rise
Market Fall

Maximum Revenue From Delayed Entry

Profit Loss Factors

- 50% development cost overrun
- Product cost 9% too high
- Ship product six months late

Sources: Synopsys Inc. and McKinsey and Co.

Delayed market entry results in lost revenue
Breakeven Analysis

- Sample Design:
  - 300K Logic Gates
  - 256 Kb RAM
  - PCI Core
  - BG560
  - Product Selling Cost: $1.5K
- Forecast Total = 315K
  - Year 1 35K
  - Year 2 70K
  - Year 3 105K
  - Year 4 70K
  - Year 5 35K
Analysis of Delayed Market Entry

Lost revenue of EA or SC implementation versus FPGA implementation

Embedded array:
\[
\text{Lost revenue} = \frac{5\text{mo}(3\times30\text{mo} - 5\text{mo})}{(2\times30\text{mo})^2} = \frac{5\times85}{60^2} = .118
\]

Standard cell:
\[
\text{Lost revenue} = \frac{7\text{mo}(3\times30\text{mo} - 7\text{mo})}{(2\times30\text{mo})^2} = \frac{7\times83}{60^2} = .16
\]

This assumes that entry into the market with the FPGA solution would capture the full market potential. The delays after that are applied for the embedded array and standard cell solutions.

Numbers come from data on following slide.
## Development Costs with TTM Pressures

(Using ASIC industry re-spin probability = 30 percent)

### NRE

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$0</td>
<td>$200k</td>
<td>$350k</td>
</tr>
</tbody>
</table>

### Tools and Maintenance

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRE</td>
<td>$20k/10k</td>
<td>$60k/10k</td>
<td>$60k/10k</td>
</tr>
</tbody>
</table>

### Engineering

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 weeks @ $5k per week = $100k</td>
<td>$750 x 500 units = $375k</td>
<td>$600 x 500 units = $300k</td>
<td></td>
</tr>
</tbody>
</table>

### Preproduction unit price

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>$240 * 35k) + ($240 * 70k * .97) + ($240 * 105k * .94) + ($240 * 180 * .91)</td>
<td>$225k * 30% = $67,5k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Production Price* Volume - With Cost Reduction

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>(500 * 35k) + (500 * 70k * .8) + (500 * 105k * .84) + (500 * 35k * .52 = 119.7M = 24% reduction)</td>
<td>$225k * 30% = $67,5k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Re-spin costs = Re-spin cost * Probability (ASIC Industry)

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$100k * 30% = $30k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Expedite Fees

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$35k</td>
<td>$50k</td>
<td></td>
</tr>
</tbody>
</table>

### Cost of development delay (engineering + production time versus FPGA solution)

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Month delay = 11.8% profit loss</td>
<td>7 Month delay = 16% profit loss</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Profit Analysis

(Using ASIC industry re-spin probability = 30 percent)

Delayed market entry profit (takes into account only this chip's cost):

Lost Profit (TTM) = TTM Lost Revenue Factor * Profit Potential
Total Profit = Volume * Selling Price - Lost Profit (TTM) - Development Costs
Cost of Delayed Market Entry = Profit Potential (original) - Total Profit

• FPGA:
  • Total Profit = 315k * 1.5k - 119.83M = 352.67M

• Embedded Array:
  • Lost Profit (TTM) = .118 * $396.05M = $46.73M
  • Total Profit = 315k * $1.5k - $46.73M - $71.98M = $353.79M
  • Cost of Delayed Market Entry = $396.05M - $353.79M = $42.26M

• Standard Cell:
  • Lost Profit (TTM) = .16 * $414.82M = $66.37M
  • Total Profit = 315k * $1.5k - $66.37M - $54.39M = $351.74M
  • Cost of Delayed Market Entry = $414.82M - $351.74M = $63.08M
Breakeven Analysis

- (Using ASIC Industry re-spin probability = 30 percent)

<table>
<thead>
<tr>
<th>Total Cost with TTM Pressures:</th>
<th>Development Costs + Cost of Delayed Market Entry + Unit Cost * Volume * Price Reduction = Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA versus embedded array, breakeven analysis with TTM pressures</td>
<td>$130k + (500 \times 0.76)x = 920k + 42.26M + (240 \times 0.94)x$</td>
</tr>
<tr>
<td></td>
<td>$154x = 43.05M \quad x = 279.5k$</td>
</tr>
<tr>
<td>FPGA versus standard cell, breakeven analysis with TTM pressures</td>
<td>$130k + (500 \times 0.76)x = 1.20M + 63.08M + (180 \times 0.94)x$</td>
</tr>
<tr>
<td></td>
<td>$210x = 64.15M \quad x = 305.5k$</td>
</tr>
</tbody>
</table>
## Development Costs with TTM Pressures

(Using ASIC industry re-spin probability = 30 percent)

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>Embedded Array</th>
<th>Standard Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRE</td>
<td>$0</td>
<td>$200k</td>
<td>$350k</td>
</tr>
<tr>
<td>Tools/Maintenance</td>
<td>$30k</td>
<td>$70k</td>
<td>$70k</td>
</tr>
<tr>
<td>Engineering</td>
<td>$100k</td>
<td>$210k</td>
<td>$260k</td>
</tr>
<tr>
<td>Pre-production unit price</td>
<td>NA</td>
<td>$375k</td>
<td>$300k</td>
</tr>
<tr>
<td>Production Price* Volume - With Cost Reduction</td>
<td>$119.7M</td>
<td>$71.06M</td>
<td>$53.30M</td>
</tr>
<tr>
<td>Re-spin costs = Re-spin cost * Probability(ASIC Industry)</td>
<td>$0</td>
<td>$30k</td>
<td>$67.5k</td>
</tr>
<tr>
<td>Expedite Fees</td>
<td>$0</td>
<td>$35k</td>
<td>$50k</td>
</tr>
<tr>
<td>Cost of development delay (engineering + production time Vs. FPGA solution)</td>
<td>-</td>
<td>$46.73M</td>
<td>$66.37M</td>
</tr>
<tr>
<td>Total Development Cost</td>
<td>$119.83M</td>
<td>$118.71M</td>
<td>$120.77M</td>
</tr>
<tr>
<td>Total Profit</td>
<td>$352.67M</td>
<td>$353.79M</td>
<td>$351.73M</td>
</tr>
</tbody>
</table>
TTM Analysis of Breakeven

- (Using ASIC industry re-spin probability)
- An FPGA is more cost-effective versus an EA through 279K units
- An FPGA is more cost-effective versus an SC through 305K units
Flexibility is Free!

- Xilinx FPGA features and densities enable system-level design without fixed logic
  - Lowest “cost-of-change”
  - IRL enabled
  - Standard product economies

- Time-to-market is maximized through Xilinx FPGAs
Pipelining Lab

Introduction

This lab illustrates how the Xilinx architecture is built for high performance by taking advantage of the registers that exist on the die. Pipelining a few simple designs illustrates that high performance can be achieved in Xilinx devices by pipelining your design.

Objectives

After completing this lab, you will be able to:

- Describe how to pipeline your code
- Create a CORE Generator pipelined multiplier

Procedure

During this procedure, you will evaluate code and analyze its performance. You will then create a CORE Generator multiplier core and implement the core.

This lab comprises four primary steps. Below each general instruction for a given procedure, you will find accompanying step-by-step directions and illustrated figures providing more detail for performing the general instruction. If you feel confident about a specific instruction, feel free to skip the step-by-step directions and move on to the next general instruction in the procedure.

Note: You can download the lab files for this module from the Xilinx FTP site at ftp://ftp.xilinx.com/pub/documentation/education/asic25001-5-prnt.zip.
### VHDL: Non-Pipelined
process (clk, reset) begin
  if reset = '1' then
    accum <= (others => '0');
    a_q <= (others => '0');
    b_q <= (others => '0');
    c_q <= (others => '0');
    d_q <= (others => '0');
  elsif rising_edge(clk) then
    a_q <= a;
    b_q <= b;
    c_q <= c;
    d_q <= d;
    accum <= accum + (a_q + b_q + c_q + d_q);
  end if;
end process;

### VHDL: Pipelined
process (clk, reset) begin
  if reset = '1' then
    accum <= (others => '0');
    a_q <= (others => '0');
    b_q <= (others => '0');
    c_q <= (others => '0');
    d_q <= (others => '0');
    ab <= (others => '0');
    cd <= (others => '0');
    abcd <= (others => '0');
  elsif rising_edge(clk) then
    a_q <= a;
    b_q <= b;
    c_q <= c;
    d_q <= d;
    ab <= ('0' & a_q) + ('0' & b_q);
    cd <= ('0' & c_q) + ('0' & d_q);
    abcd <= ('0' & ab) + ('0' & cd);
    accum <= accum + abcd;
  end if;
end process;

### Verilog: Non-Pipelined
always @ (posedge clk or posedge reset) begin
  if (reset)
    begin
      accum <= 0;
      a_q <= 0;
      b_q <= 0;
      c_q <= 0;
      d_q <= 0;
    end
  else
    begin
      a_q <= a;
      b_q <= b;
      c_q <= c;
      d_q <= d;
      accum <= accum + (a_q + b_q + c_q + d_q);
    end
end

### Verilog: Pipelined
always @ (posedge clk or posedge reset) begin
  if (reset)
    begin
      accum <= 0;
      a_q <= 0;
      b_q <= 0;
      c_q <= 0;
      d_q <= 0;
      ab <= 0;
      cd <= 0;
      abcd <= 0;
    end
  else
    begin
      a_q <= a;
      b_q <= b;
      c_q <= c;
      d_q <= d;
      ab <= a_q + b_q;
      cd <= c_q + d_q;
      abcd <= ab + cd;
      accum <= accum + abcd;
    end
end

Figure 2-1. Non-Pipelined Accumulator.

Figure 2-2. Pipelined Accumulator.
1. The code in Figure 2-1 is a basic addition-accumulation circuit. For a high-performance implementation, the code in Figure 2-2 could be used. Why is the pipelined version in Figure 2-2 beneficial in an FPGA?

________________________________________________________________

________________________________________________________________

2. What is the drawback of implementing the pipelined version?

________________________________________________________________

________________________________________________________________

Implementation Results for Addition-Accumulation Circuits

The code for Figures 2-1 and 2-2 has been implemented for you. The code for each example resides in c:\training\desperf\labs\pipeline\add_accum. The name of the HDL code for Figure 2-1 is add_accum_slow.vhd and the code for Figure 2-2 is add_accum_pipeline.vhd. The code was synthesized with XST targeting a XC2V40 FG456 –5. The resulting frequencies and resource utilization of each is shown:

Figure 2-1 with a period specification of 10 ns and effort level of 3
Frequency: 88 MHz
Resources: FFS: 96  LUTs: 76

Figure 2-2 with a period specification of 10 ns and effort level of 3
Frequency: 200 MHz
Resources: FFS: 148 LUTs: 74

Evaluate the code and answer the questions that follow.

<table>
<thead>
<tr>
<th>VHDL:</th>
<th>Verilog:</th>
</tr>
</thead>
<tbody>
<tr>
<td>process (clk, reset) begin if reset = '1' then c &lt;= (others =&gt; '0'); a_q &lt;= (others =&gt; '0'); b_q &lt;= (others =&gt; '0'); elsif rising_edge(clk) then a_q &lt;= a; b_q &lt;= b; c &lt;= a_q * b_q; end if; end process;</td>
<td>always @ (posedge clk or posedge reset) begin if (reset) begin c &lt;= 0; a_q &lt;= 0; b_q &lt;= 0; end else begin a_q &lt;= a; b_q &lt;= b; c &lt;= a_q * b_q; end end</td>
</tr>
</tbody>
</table>

Figure 2-3. Basic Multiplier.

3. The code in Figure 2-3 is a basic 16 x 16 multiplier. Can this be easily pipelined for a high-performance implementation?

________________________________________________________________
4. What are the benefits to using the CORE Generator system to implement a pipelined multiplier?

Implement the Slow Multiplier Step 2

Open the project slow_pipe_lab.npl from:
- Spartan -3 users:
  c:\training\desperf\labs\pipelining\slow_mult\s3_slow_pipe_lab.
- Virtex -II users:
  c:\training\desperf\labs\pipelining\slow_mult\v2_slow_pipe_lab.

Implement the design.

For the slow multiplier project, XST will implement the multiplier by using the dedicated MULT18X18 block. In the next step (Step 3), we will try to exceed the resultant period in this step (Step 2) by using a pipelined LUT implementation.

1. Select Start → Programs → Xilinx ISE 5 → Project Navigator
2. Select File → Open Project
3. Browse to:
   - Spartan-3: c:\training\desperf\labs\pipelining\slow_mult\s3_slow_pipe_lab
   - Virtex-II: c:\training\desperf\labs\pipelining\slow_mult\v2_slow_pipe_lab
4. Select slow_pipe_lab.npl. Click Open
5. In the Sources in Project window, select slow_mult.v
6. Double-click Implement Design

The design has been synthesized and a Period constraint of 8.3 ns has already been entered for you. XST has used a block multiplier (MULT18X18) to implement the multiplier.

Open the Place & Route report, and answer the questions that follow.

1. When the design has completed the Place & Route implementation phase, expand → Implement Design, expand → Place & Route, and double-click Place & Route Report
2. Near the bottom of the file, look for the Constraint Summary, as shown in Figure 2-4.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Requested</th>
<th>Actual</th>
<th>Logic</th>
<th>Levels</th>
</tr>
</thead>
</table>

Figure 2-4. Constraint Summary.
5. What is the resulting Period for the slow multiplier?

Create the Fast Multiplier Core  Step 3

Open the ISE project fast_pipe_lab.npl from:
- Spartan-3 users:
  c:\training\desperf\labs\pipelining\fast_mult\s3_fast_pipe_lab
- Virtex-II users:
  c:\training\desperf\labs\pipelining\fast_mult\v2_fast_pipe_lab

Open the COREGen GUI from within ISE. Name the core mult_16x16.

1. In the ISE 5 Project Navigator, select File → Open Project
2. Browse to:
   - Spartan-3 users: c:\training\desperf\labs\pipelining\fast_mult\s3_fast_pipe_lab
   - Virtex-II users: c:\training\desperf\labs\pipelining\fast_mult\v2_fast_pipe_lab
3. Select fast_pipe_lab.npl. Click Open
4. Click Project menu → New Source
5. In the New Source window, select Coregen IP. For File Name enter mult_16x16, as shown in Figure 2-5

![Figure 2-5. New Source Window.](image)

6. Click Next, then click Finish

This CORE Generator GUI opens.
Under Math Functions folder, under Multipliers folder, double-click Multiplier. Enter the following information:

- Component Name: mult_16x16
- Pipeline: Maximum Pipelining
- Register Options: Asynchronous Clear

1. In the Catalog window on the left, double-click Math Functions, and click Multipliers

2. In the window on the right, double-click Multiplier, as shown in Figure 2-6

![Figure 2-6. CORE Generator GUI.](image)

On the first page enter this data:

- Component Name → mult_16x16

For this implementation, we want to use LUTs for the implementation. In the slow multiplier project, the synthesis tool used a block multiplier. Find out if a pipelined LUT implementation is faster…

3. Click Next

4. On the next two pages, review the data and leave them at their default values. Click Next on pages 2 and 3.

5. On page 4, select the following:
   - Pipeline → Maximum Pipelining
   - Register Options → Asynchronous Clear

6. Click Generate

7. When the Successfully Generated window appears, click OK
Implement the Fast Multiplier Project

Implement the fast multiplier project. Review the Place & Route report and answer the questions that follow.

1. In the ISE Project Navigator, in the Sources in Project window, double-click fast_mult.v

   Note the CORE Generator core for the mult_16x16 has already been instantiated for you.

2. In the Processes for Current Source window, double-click Implement Design

   A Period constraint of 8.3 ns has already been entered for you.

3. When the design has completed the Place & Route implementation phase, expand Implement Design, expand Place & Route, and double-click Place & Route Report

4. Near the bottom of the file, look for the Constraint Summary, as shown in Figure 2-7.

   ![](constraint_summary.png)

   **Figure 2-7. Constraint Summary.**

   6. What is the resulting Period for the fast multiplier?

   ![](constraint_summary.png)

### Conclusion

Coding styles have a large impact on the resulting performance of any design. However, even the best coding style will not always take advantage of the architectural resources. In this lab, you found that a multiplier implemented by using LUT resources with pipeline stages can be faster than using the dedicated multiplier block. This simply reminds you that you need to consider the required performance and area results, then consider the best implementation style for many different resources, including, of course, multipliers.
Answers

1. The code in Figure 2-1 is a basic addition-accumulation circuit. For a high performance implementation, the code in Figure 2-2 could be used. Why is the pipelined version in Figure 2-2 beneficial in an FPGA?

   It will help to increase the frequency at which the design can be run. Also, there is no negative impact on the area because the registers exist on the die. Xilinx FPGAs have a “register-rich” architecture, making it ideal for highly pipelined application.

2. What is the drawback of implementing the pipelined version?

   The only possible drawback is if the latency requirements cannot be met for the design by adding the additional pipeline stages.

3. The code in Figure 2-3 is a basic 16 x 16 multiplier. Can this be easily pipelined for a high-performance implementation?

   Pipelining a multiplier is not easily done in code. However, if you add pipeline stages in the code after a multiplier, Synplicity’s Synplify will insert them into the multiplier if you use the Pipeline synthesis option.

4. What are the benefits to using CORE Generator system to implement a pipelined multiplier?

   The CORE Generator system will create a pipelined multiplier that has relationally placed macros; that is, the placement of the logic within the multiplier will remain together, resulting in a consistent implementation with high performance. The CORE Generator system also comes with the files for simulating a multiplier.

5. What is the resulting Period for the slow multiplier?

   *Your results may vary depending on your software environment*
   Spartan -3: ~11.4 ns  
   Virtex -II: ~8.3 ns

6. What is the resulting Period for the fast multiplier?

   *Your results may vary depending on your software environment*
   Spartan-3: ~7.8 ns  
   Virtex-II: ~4.7 ns