ECE 699: Lecture 9

High-Level Synthesis
Part 1
Required Reading

The ZYNQ Book

- Chapter 14: Spotlight on High-Level Synthesis
- Chapter 15: Vivado HLS: A Closer Look

S. Neuendorffer and F. Martinez-Vallina, Building Zynq Accelerators with Vivado High Level Synthesis, FPGA 2013 Tutorial (selected slides on Piazza)
Recommended Reading


Behavioral Synthesis

- Algorithm
- Target Library
- I/O Behavior
- Behavioral Synthesis
- RTL Design
- Logic Synthesis
- Gate level Netlist

Classic RTL Design Flow
Need for High-Level Design

• Higher level of abstraction
• Modeling complex designs
• Reduce design efforts
• Fast turnaround time
• Technology independence
• Ease of HW/SW partitioning
Platform Mapping
SW/HW Partitioning

Program

Software
(executed in
the microprocessor system)

Hardware
(executed in
the reconfigurable processor system)
SW/HW Partitioning & Coding
Traditional Approach

Specification

SW/HW Partitioning

SW Coding

SW Compilation

SW Profiling

HW Coding

HW Compilation

HW Profiling
SW/HW Partitioning & Coding

New Approach

- Specification
- SW/HW Coding
- SW/HW Partitioning
  - SW Compilation
  - HW Compilation
  - SW Profiling
  - HW Profiling
Advantages of Behavioral Synthesis

• Easy to model higher level of complexities
• Smaller in size source compared to RTL code
• Generates RTL much faster than manual method
• Multi-cycle functionality
• Loops
• Memory Access
Short History of High-Level Synthesis

Generation 1 (1980s-early 1990s): research period

Generation 2 (mid 1990s-early 2000s):
• Commercial tools from Synopsys, Cadence, Mentor Graphics, etc.
• Input languages: behavioral HDLs  Target: ASIC

Outcome: Commercial failure

Generation 3 (from early 2000s):
• Domain oriented commercial tools: in particular for DSP
• Input languages: C, C++, C-like languages (Impulse C, Handel C, etc.), Matlab + Simulink, Bluespec
• Target: FPGA, ASIC, or both

Outcome: First success stories
Hardware-Oriented High-Level Languages

• C-Based System level languages
  • Commercial
    • Handel C -- Celoxica Ltd.
    • Impulse C -- Impulse Accelerated Technologies
    • Carte C – SRC Computers
    • SystemC -- The Open SystemC Initiative
  • Research
    • Streams-C -- Los Alamos National Laboratory
    • SA-C -- Colorado State University, University of California, Riverside, Khoral Research, Inc.
    • SpecC – University of California, Irvine and SpecC Technology Open Consortium
Other High-Level Design Flows

- Matlab-based
  - AccelChip DSP Synthesis -- AccelChip
  - System Generator for DSP -- Xilinx
- GUI Data-Flow based
  - Corefire -- Annapolis Microsystems
- Java-based
  - Commercial
    - Forge -- Xilinx
  - Research
    - JHDL – Brigham Young University
Handel-C Overview

- High-level language based on ISO/ANSI-C for the implementation of algorithms in hardware
- Allows software engineers to design hardware without retraining
- Clean extensions for hardware design including flexible data widths, parallelism and communications
- Well defined timing model
  - Each statement takes a single clock cycle
- Includes extended operators for bit manipulation, and high-level mathematical macros (including floating point)
Handel-C/ANSI-C Comparisons

**ANSI-C**
- ANSI-C Standard Library
- Recursion
- Floating Point
- Preprocessors i.e. `#define`
- Pointers
- Structures
- ANSI-C Constructs for, while, if, switch
- Arrays
- Bitwise logical operators
- Logical operators
- Arithmetic operators
- Functions
- Handel-C Standard Library
- Parallelism
- Arbitrary width variables
- Enhanced bit manipulation
- RAM, ROM
- Signals
- Interfaces
Handel-C Design Flow

Executable Specification

Handel-C

VHDL

Synthesis

EDIF

Place & Route

EDIF
Different Levels of C/C++ Synthesis Abstraction

- **Untimed C Domain** (Non-implementation-specific)
- **Timed C Domain** (Implementation-specific)
- **RTL Domain** (Implementation-specific)

- More abstract, less implementation-specific
- Less abstract, more implementation-specific
- Pure C/C++
Pure Untimed C/C++ Design Flow

- Non-implementation-specific
- Easy to create
- Fast to simulate
- Easy to modify

User interaction and guidance

Verilog / VHDL RTL

RTL Synthesis

Gate-level netlist

RTL Synthesis

LUT/CLB-level netlist

Auto-generated, implementation-specific

ASIC target

FPGA target

Pure C/C++ Synthesis

Pure C/C++
Mentor Graphics – Catapult C

Catapult C Synthesis
Mentor Graphics – Catapult C

• Catapult C automatically converts un-timed C/C++ descriptions into synthesizable RTL.
SystemC -based design-flow alternatives

SystemC

Implementation specific, relatively slow to simulate, relatively difficult to modify

Auto-RTL Translation → Verilog / VHDL RTL → RTL Synthesis

Gate-level netlist

SystemC Synthesis

Alternative SystemC flows
SystemC Evolution

The Design Warrior’s Guide to FPGAs
Devices, Tools, and Flows. ISBN 0750676043
Copyright © 2004 Mentor Graphics Corp. (www.mentor.com)
Reconfigurable Supercomputers
What is a Reconfigurable Computer?

Microprocessor system

- μP
- μP memory
- I/O

Reconfigurable system

- FPGA
- FPGA memory
- Interface

Microprocessor system vs. Reconfigurable system: Reconfigurable systems offer more flexibility and reusability compared to microprocessor systems.
# Reconfigurable Supercomputers

<table>
<thead>
<tr>
<th>Machine</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRC 6</strong> from SRC Computers</td>
<td>2002</td>
</tr>
<tr>
<td><strong>Cray XD1</strong> from Cray</td>
<td>2005</td>
</tr>
<tr>
<td><strong>SGI Altix</strong> from SGI</td>
<td>2005</td>
</tr>
<tr>
<td><strong>SRC 7</strong> from SRC Computers, Inc.</td>
<td>2006</td>
</tr>
</tbody>
</table>
Pros and cons of reconfigurable computers

+ can be programmed using high-level programming languages, such as C, by mathematicians & scientist themselves
+ facilitates hardware/software co-design
+ shortens development time, encourages experimentation and complex optimizations
+ allows sharing costs among users of various applications

- high entry cost (~$100,000)
- hardware aware programming
- limited portability
- limited availability of libraries
- limited maturity of tools.
SRC Programming Model

Microprocessor

main.c

function_1()
function_2()

ANSI C

FPGA

function_1
macro_1(a, b, c)
macro_2(b, d)
macro_2(c, e)

function_2
macro_3(s, t)
macro_1(n, b)
macro_4(t, k)

MAP C
(subset of ANSI C)

Libraries of macros

macro_1
macro_2
macro_3
macro_4

VHDL

I/O
Macro_1
Macro_2
Macro_2
I/O

I/O
a
b
c
d
e
SRC Compilation Process

Application sources
- .c or .f files
- .mc or .mf files

μP Compiler
- .o files

MAP Compiler
- .o files

Linker
- Application executable

Macro sources
- .vhd or .v files

Logic synthesis
- .v files

Place & Route
- Netlists .ngo files
- .bin files

Configuration bitstreams

HDL sources
- .v files
Library Development - SRC

<table>
<thead>
<tr>
<th>μP system</th>
<th>FPGA system</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLL (ASM)</td>
<td>HDL (VHDL, Verilog)</td>
</tr>
<tr>
<td>HLL (C, Fortran)</td>
<td>HLL (C, Fortran)</td>
</tr>
</tbody>
</table>

Library Developer

Application Programmer
SRC Programming Environment

+ very easy to learn and use
+ standard ANSI C
+ hides implementation details
+ very well integrated environment
+ mature - in production use for over 4 years with constant improvements

- subset of C
- legacy C code requires rewriting
- C limitations in describing HW (parallelism, data types)
- closed environment, limited portability of code to HW platforms other than SRC
Application Development for Reconfigurable Computers

Program Entry

Platform mapping

Compilation

Debugging & Verification

Execution
Ideal Program Entry

Function

Program Entry
Actual Program Entry

Function

Preferred Architectures

Use of FPGA Resources (multipliers, μP cores)

Sequence of Run-time Reconfigurations

SW/HW Interface

SW/HW Partitioning

FPGA Mapping

Data Transfers & Synchronization

Use of Internal and External Memories
Cinderella Story

AutoESL Design Technologies, Inc. (25 employees)

Flagship product:

AutoPilot, translating C/C++/System C to VHDL or Verilog

• Acquired by the biggest FPGA company, Xilinx Inc., in 2011
• AutoPilot integrated into the primary Xilinx toolset, Vivado, as
  Vivado HLS, released in 2012

“High-Level Synthesis for the Masses”
Vivado HLS

High Level Language
C, C++, System C

Vivado
HLS

Hardware Description Language
VHDL or Verilog
HLS-Based Development and Benchmarking Flow

Reference Implementation in C

Manual Modifications (pragmas, tweaks)

HLS-ready C code

High-Level Synthesis

HDL Code

Physical Implementation

FPGA Tools

Netlist

Post Place & Route Results

Test Vectors

Functional Verification

Timing Verification

Manual Modifications (pragmas, tweaks)
LegUp – Academic Tool for HLS

- Open-source HLS Tool
  - Developed at the University of Toronto
  - Faculty supervisors: Jason H. Anderson and Stephen Brown
  - FPL Community Award 2014
- High-Level Synthesis from C to Verilog
- Targets Altera FPGAs (extension to Xilinx relatively simple)
- Two flows
  - Pure Hardware
  - Hardware/Software Hybrid
    = Tiger MIPS + hardware accelerator(s) + Avalon bus + shared on-chip and off-chip memory
Cryptol – New Language for Cryptology

- Domain specific language for cryptology: Cryptol
  - High-level programming language similar to Haskell
  - Developed by Galois Inc. based in Portland, USA
- High-Level Synthesis from Cryptol to efficient Software and Hardware
Levels of Abstraction in FPGA Design

C/C++/SystemC design entry

HDL design entry

Source: The Zynq Book
High-Level Synthesis vs. Logic Synthesis

Source: The Zynq Book
Algorithm and Interface Synthesis

Source: The Zynq Book
Vivado HLS Design Flow

Source: The Zynq Book
Design Trade-offs Explored Using HLS

Source: The Zynq Book
C Functional Verification and C/RTL Cosimulation in Vivado HLS

Source: The Zynq Book
Vivado HLS

- Starts at C
  - C
  - C++
  - SystemC

- Produces RTL
  - Verilog
  - VHDL
  - SystemC

- Automates Flow
  - Verification
  - Implementation

Vivado IP Packer
Vivado HLS
Scheduling and Binding
Vivado HLS
Scheduling and Binding

Scheduling – translation of the RTL statements interpreted from the C code into a set of operations, each with an associated duration in terms of clock cycles. Affected by the clock frequency, uncertainty, target technology, and user directives.

Binding - associating the scheduled operations with the physical resources of the target device.

Source: The Zynq Book
Three Possible Outcomes from HLS
Average of 10 numbers

Key:
- Adder (fabric)
- Multiplier (fabric)
- Adder (DSP48x)
- Multiplier (DSP48x)
Vivado HLS Synthesis Process

Source: The Zynq Book
# Native Integer Data Types of C

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Number of Bits(^a)</th>
<th>Range(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Representation of the basic character set.</td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>signed char</td>
<td></td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short int</td>
<td>A reduced precision version of int, requiring less storage.</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int</td>
<td>The basic integer data type.</td>
<td>32</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long int</td>
<td>In many cases the long int type will be the same length as int, i.e. 32 bits.</td>
<td>32</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long long int</td>
<td>An extended precision integer type.</td>
<td>64</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td></td>
<td>64</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

Source: The Zynq Book
## Arbitrary Precision Integer Data Types of C and C++ Accepted by Vivado HLS

<table>
<thead>
<tr>
<th>Language</th>
<th>Integer Data Type</th>
<th>Description</th>
<th>Required Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>intN (e.g. int7)</td>
<td>signed integer of N bits precision</td>
<td>#include “ap_cint.h”</td>
</tr>
<tr>
<td></td>
<td>uintN (e.g. uint7)</td>
<td>unsigned integer of N bits precision</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>ap_int&lt;(N)&gt; (e.g. ap_int&lt;7&gt;)</td>
<td>signed integer of (N) bits precision</td>
<td>#include “ap_int.h”</td>
</tr>
<tr>
<td></td>
<td>ap_uint&lt;(N)&gt; (e.g. ap_uint&lt;7&gt;)</td>
<td>unsigned integer of (N) bits precision</td>
<td></td>
</tr>
</tbody>
</table>

Source: The Zynq Book
Arbitrary Precision Integer Types of C and C++

// C code example
#include "ap_cint.h"

void top_level_function (...) {
    // declarations
    int6 small_signed;
    uint10 big_unsigned;
    int22 vbig_signed;
    ...
}

// C++ code example
#include "ap_int.h"

void top_level_function (...) {
    // declarations
    ap_int<6> small_signed;
    ap_uint<10> big_unsigned;
    ap_int<22> vbig_signed;
    ...
}

Source: The Zynq Book
## Native Floating-Point Data Types of C

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Single precision floating point (IEEE 754)</td>
<td>32</td>
<td>-3.403e+38 to 3.403e+38</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating point (IEEE 754)</td>
<td>64</td>
<td>-1.798e+308 to 1.798e+308</td>
</tr>
</tbody>
</table>

Source: The Zynq Book
Fixed-point Word Format

weighting: $\pm 2^4 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7}$

1 = 5 integer bits  B = 7 fractional bits

Source: The Zynq Book
## Arbitrary Precision Fixed-Point Data Types used in Vivado HLS

<table>
<thead>
<tr>
<th>Language</th>
<th>Fixed Point Data Type</th>
<th>Description</th>
<th>Required Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>ap_fixed(&lt;W,I,Q,O,N&gt;)</td>
<td>Signed fixed point number of I integer bits and W-I fractional bits.</td>
<td>#include “ap_fixed.h”</td>
</tr>
<tr>
<td></td>
<td>ap_ufixed(&lt;W,I,Q,O,N&gt;)</td>
<td>Unsigned fixed point number of I integer bits and W-I fractional bits.</td>
<td></td>
</tr>
</tbody>
</table>

\(W\) – total width, \(I\) – number of integer bits, 
\(Q\) – quantization mode, \(O\) – overflow mode, 
\(N\) – number of saturation bits in overflow wrap modes

Source: The Zynq Book
# Quantization modes for the C++ `ap_fixed` and `ap_ufixed` types

<table>
<thead>
<tr>
<th>Parameter (quantisation)</th>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AP_RND</td>
<td>Rounding to positive infinity</td>
</tr>
<tr>
<td></td>
<td>AP_RND_ZERO</td>
<td>Rounding to zero</td>
</tr>
<tr>
<td></td>
<td>AP_RND_MIN_INF</td>
<td>Rounding to negative infinity</td>
</tr>
<tr>
<td></td>
<td>AP_RND_INF</td>
<td>Rounding to infinity</td>
</tr>
<tr>
<td></td>
<td>AP_RND_CONV</td>
<td>Convergent rounding</td>
</tr>
<tr>
<td></td>
<td>AP_TRN</td>
<td>Truncation to negative infinity</td>
</tr>
<tr>
<td></td>
<td>AP_TRN_ZERO</td>
<td>Truncation to zero</td>
</tr>
</tbody>
</table>

Source: The Zynq Book
Truncation to zero

AP_TRN

- Round the value to the nearest representable value.
- Always round the value towards minus infinity.

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25;  // Yields: 1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.5
```

*Example 4-12: AP_TRN Examples*

AP_TRN_ZERO

Round the value to the nearest representable value.

* For positive values, the rounding is the same as mode AP_TRN.
* For negative values, round towards zero.

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25;  // Yields: 1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.0
```

*Example 4-13: AP_TRN_ZERO Examples*

## Overflow modes for
for the C++ ap_fixed and ap_ufixed types

<table>
<thead>
<tr>
<th>Overflow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP_SAT</td>
<td>Saturation</td>
</tr>
<tr>
<td>AP_SAT_ZERO</td>
<td>Saturation to zero</td>
</tr>
<tr>
<td>AP_SAT_SYM</td>
<td>Symmetrical saturation</td>
</tr>
<tr>
<td>AP_WRAP</td>
<td>Wraparound</td>
</tr>
<tr>
<td>AP_WRAP_SM</td>
<td>Sign magnitude wrap around</td>
</tr>
</tbody>
</table>

Source: The Zynq Book
Wraparound

Wrap the value around in case of overflow.

\[
\begin{align*}
ap_{\text{fixed}}\langle 4, 4, \text{AP\_RND, AP\_WRAP} \rangle & \text{ \ UAPFixed4 = 31.0; } \quad \text{\(\text{\# Yields: -1.0}\)} \\
ap_{\text{fixed}}\langle 4, 4, \text{AP\_RND, AP\_WRAP} \rangle & \text{ \ UAPFixed4 = -19.0; } \quad \text{\(\text{\# Yields: -3.0}\)} \\
ap_{\text{ufixed}}\langle 4, 4, \text{AP\_RND, AP\_WRAP} \rangle & \text{ \ UAPFixed4 = 19.0; } \quad \text{\(\text{\# Yields: 3.0}\)} \\
ap_{\text{ufixed}}\langle 4, 4, \text{AP\_RND, AP\_WRAP} \rangle & \text{ \ UAPFixed4 = -19.0; } \quad \text{\(\text{\# Yields: 13.0}\)}
\end{align*}
\]

**Example 4-17: AP\_WRAP Examples**

If the value of \(N\) is set to zero (the default overflow mode):

- All MSB bits outside the range are deleted.
- For unsigned numbers. After the maximum it wraps around to zero.
- For signed numbers. After the maximum, it wraps to the minimum values.

If \(N > 0\):

- When \(N > 0\), \(N\) MSB bits are saturated or set to 1.
- The sign bit is retained, so positive numbers remain positive and negative numbers remain negative.
- The bits that are not saturated are copied starting from the LSB side.

C++ code with the declaration of fixed point variables

// C++ code example
#include "ap_fixed.h"

void top_level_function (..)
{
    // declarations
    ap_ufixed<8,3> small_unsigned; // 3 int, 5 fract, defaults
    ap_fixed<10,4,AP_RND> big_signed; // round to + inf.
    ap_ufixed<10,4,AP_RND_ZERO> big_unsigned; // round to zero
    ap_fixed<21,10,AP_TRN,AP_SAT> vbig_signed; // trunc., satur.
    ap_ufixed<21,10,AP_RND_CONV> vbig_unsigned; // conv. round.
    ...
}
# System C Data Types

<table>
<thead>
<tr>
<th>SystemC Data Type</th>
<th>Description</th>
<th>Required Preamble</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_int&lt;W&gt;</td>
<td>signed integer: (up to 64 bits) (up to 512 bits)</td>
<td>#include &quot;systemc.h&quot;</td>
</tr>
<tr>
<td>sc_bigtint&lt;W&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sc_uint&lt;W&gt;</td>
<td>unsigned integer: (up to 64 bits) (up to 512 bits)</td>
<td>#define SC_INCLUDE_FX</td>
</tr>
<tr>
<td>sc_ubigtint&lt;W&gt;</td>
<td></td>
<td>[#define SC_FX_EXCLUDE_OTHER]</td>
</tr>
<tr>
<td>sc_fixed&lt;W,I,Q,O,N&gt;</td>
<td>signed fixed point</td>
<td>include &quot;systemc.h&quot;</td>
</tr>
<tr>
<td>sc_ufixed&lt;W,I,Q,O,N&gt;</td>
<td>unsigned fixed point</td>
<td></td>
</tr>
</tbody>
</table>

Source: The Zynq Book
An Example Top-Level Function for HLS

```c
void find_average_of_best_X (int *average, int samples[8], int X)
{
    // body of function (statements, sub-function calls, etc.)
}
```

Source: The Zynq Book
Simplified Interface Diagram for the Example Top-Level Function

Source: The Zynq Book
# Synthesis of Port Directions

<table>
<thead>
<tr>
<th>C/C++ Function Argument</th>
<th>RTL Port Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>An argument which is read from and never written to</td>
<td>in</td>
</tr>
<tr>
<td>An argument which is written to and never read from</td>
<td>out</td>
</tr>
<tr>
<td>A value output by the function return statement</td>
<td>out</td>
</tr>
<tr>
<td>An argument which is both written to and read from</td>
<td>inout (bidirectional)</td>
</tr>
</tbody>
</table>
Default Port Level Types and Protocols

Source: The Zynq Book
Data flow between Vivado HLS blocks

Source: The Zynq Book
RTL Interface Diagram Showing Default Block Level Ports and Protocols

Source: The Zynq Book
Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study

Ekawat Homsirikamol & Kris Gaj
George Mason University
USA

Project supported by NSF Grant #1314540
Ekawat Homsirikamol
a.k.a “Ice”

Working on the PhD Thesis entitled
“A New Approach to the Development of Cryptographic Standards Based on the Use of High-Level Synthesis Tools”
Traditional Development and Benchmarking Flow

Informal Specification

Manual Design

HDL Code

Manual Optimization

FPGA Tools

Netlist

Functional Verification

Timing Verification

Test Vectors

Post Place & Route Results
Extended Traditional Development and Benchmarking Flow

- Informal Specification
- Manual Design
- HDL Code
- Option Optimization (FPGA Tools)
- Netlist
- Functional Verification
- Timing Verification
- GMU ATHENa
- Test Vectors

Post Place & Route Results
ATHENa – Automated Tool for Hardware Evaluation

http://cryptography.gmu.edu/athena

Benchmarking open-source tool, written in Perl, aimed at an AUTOMATED generation of OPTIMIZED results for MULTIPLE hardware platforms

Currently under development at George Mason University
Generation of Results Facilitated by ATHENA

- batch mode of FPGA tools

vs.

- ease of extraction and tabulation of results
  - Text Reports, Excel, CSV (Comma-Separated Values)
- optimized choice of tool options
  - GMU_optimization_1 strategy
HLS-Based Development and Benchmarking Flow

Reference Implementation in C

Manual Modifications (pragmas, tweaks)

HLS-ready C code

High-Level Synthesis

HDL Code

Option Optimization

FPGA Tools

Netlist

Test Vectors

Functional Verification

GMU ATHENA

Timing Verification

Post Place & Route Results
Case Study

- **Algorithm:** AES-128
- **Mode of operation:** Counter (CTR)
- **Protocol and interface:** GMU proposal
- **Two vendors:** Xilinx & Altera
- **Four different FPGA families**
  - ✓ Xilinx Spartan-6 (X-S6)
  - ✓ Xilinx Virtex-7 (X-V7)
  - ✓ Altera Cyclone IV (A-CIV)
  - ✓ Altera Stratix V (A-SV)
Tools & Tool Versions

- Vivado HLS 2014.1
- Xilinx ISE v14.7
- Altera Quartus II v13.0sp1
- ATHENa v0.6.4 (with GMU_optimization_1)
Interface & Protocol

CRYPTO CORE

w
pdi
pdi_ready
pdi_read
do
w
sdi
do_ready
do_write
w
sdi_ready
sdi_read

a)

<table>
<thead>
<tr>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg(hdr)</td>
</tr>
<tr>
<td>seg(dat)</td>
</tr>
</tbody>
</table>

b)

<table>
<thead>
<tr>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg(hdr)</td>
</tr>
<tr>
<td>seg(dat)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>seg(hdr)</td>
</tr>
<tr>
<td>seg(dat)</td>
</tr>
</tbody>
</table>
Top-Level
Reference Hardware Design in RTL VHDL

Note: All buses are 128-bit wide
RTL Result

Latency = 11 cycles

Time between two consecutive outputs = 10 cycles
Software Design

Reference Code


HLSv0

• Removed support for decryption
• Removed support for different AES variants
HLSv0: Xilinx Results

Latency = 7367 cycles
HLSv1: Code Refactoring

Refactor the code to match the target AES architecture

- KeyScheduling is performed once per round
- Improved Galois field multiplication operation
- Included last round as part of the core loop
HLSv1: Xilinx Results

Latency = 3224 cycles
HLSv2: Optimization directives: ARRAY_RESHAPE

- Change an array shape in the output hardware

```c
void AES_encrypt (word8 a[4][4], word8 k[4][4], word8 b[4][4])
{
#pragma HLS ARRAY_RESHAPE variable=a[0] complete dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[1] complete dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[2] complete dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[3] complete dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a complete dim =1 reshape
```
HLSv2: Optimization directives: UNROLL & INLINE

- **Unroll a loop**
  
  OutputLoop: for (i = 0; i < 4; i ++)
  
  #pragma HLS UNROLL
  
  for (j = 0; j < 4; j ++)
  
  #pragma HLS UNROLL
  
  b[i][j] = s[i][j];

- **Flatten a function's hierarchy for improved performance**

  void KeyUpdate (word8 k[4][4], word8 round) {
  
  #pragma HLS INLINE
  
  ...
  
  }


HLSv2: Optimization directives: RESOURCE & INTERFACE

- Specify the type of FPGA resource to be used by the target variable

```plaintext
word32 rcon[10] = {
    0x01, 0x02, 0x04, 0x08, 0x10,
    0x20, 0x40, 0x80, 0x1b, 0x36
};
#pragma HLS RESOURCE variable=Rcon0 core=ROM_1P_1S
```

- Direct how an input/output port should behave, i.e., registered or handshake mode

```plaintext
void AES_encrypt (word8 a[4][4], word8 k[4][4], word8 b[4][4])
{
    #pragma HLS INTERFACE register port=b
```
HLSv2: Xilinx Results

Latency = 11 cycles
HLSv2: HLS vs. RTL, Frequency - Area

![Graphs showing frequency vs. area for Altera and Xilinx devices.
Altera Cyclone IV: HLS vs. RTL
Altera Stratix V: HLS vs. RTL
Xilinx Spartan 6: HLS vs. RTL
Xilinx Virtex 7: HLS vs. RTL]
HLSv2: HLS vs. RTL, Throughput - Area

![Graphs showing throughput and area comparison between HLS and RTL for different FPGA types: Altera Cyclone IV, Altera Stratix V, Xilinx Spartan 6, Xilinx Virtex 7. Each graph compares throughput (Mbit/s) on the y-axis with area (LEs/ALMs/SLICEs) on the x-axis, with points indicating HLS (blue) and RTL (red) performance.]

- **Altera Cyclone IV**
- **Altera Stratix V**
- **Xilinx Spartan 6**
- **Xilinx Virtex 7**
Source of Inefficiencies: Datapath vs. Control Unit

Determines
• Area
• Clock Frequency

Determines
• Number of clock cycles
Source of Inefficiencies

Datapath inferred correctly

- **Frequency and area** within 10% of manual designs

Control Unit suboptimal

- Difficulty in inferring an overlap between completing the last round and reading the next input block
- One additional clock cycle used for initialization of the state at the beginning of each round
- The formulas for throughput:

  RTL:  Throughput = \( \text{Block\_size} / (\#\text{Rounds} \times T_{\text{CLK}}) \)

  HLS:  Throughput = \( \text{Block\_size} / ((\#\text{Rounds}+2) \times T_{\text{CLK}}) \)
AES-ECB-ENC x2: HLS vs. RTL, Frequency - Area
AES-ECB-ENC x2: HLS vs. RTL, Throughput - Area

- Altera Cyclone IV
- Altera Stratix V
- Xilinx Spartan 6
- Xilinx Virtex 7

Throughput (Mbit/s) vs. Area (LEs/ALMs/SliceEs)

- Iterative
- x2 Unroll

- RTL
- HLS
AES-CTR

AES-CTR

AES
ECB-ENC
AES-CTR Results

![Graphs showing throughput vs area for Altera Stratix V and Xilinx Virtex 7.]
Full AES-CTR with I/O processors
AES-CTR with IO Results

![Graphs showing throughput vs. area for different FPGA types (Cyclone IV, Stratix V, Spartan 6, Virtex 7) with data points for RTL and HLS.]
## Results for AES

<table>
<thead>
<tr>
<th>Imp.</th>
<th>Area (LEs/ALMs/SLICEs)</th>
<th>Memory/BRAMs</th>
<th>Latency</th>
<th>Freq. (MHz)</th>
<th>TP (Mbits/s)</th>
<th>TP/Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Altera Cyclone IV</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td>4925</td>
<td>0</td>
<td>11</td>
<td>143</td>
<td>1833</td>
<td>0.37</td>
</tr>
<tr>
<td>HLSv0</td>
<td>N/A</td>
<td>N/A</td>
<td>7367</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>HLSv1</td>
<td>N/A</td>
<td>N/A</td>
<td>3224</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>HLSv2</td>
<td>4934</td>
<td>0</td>
<td>11</td>
<td>144</td>
<td>1540</td>
<td>0.31</td>
</tr>
<tr>
<td><strong>Altera Stratix V</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td>1134</td>
<td>0</td>
<td>11</td>
<td>413</td>
<td>5291</td>
<td>4.67</td>
</tr>
<tr>
<td>HLSv0</td>
<td>N/A</td>
<td>N/A</td>
<td>7367</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>HLSv1</td>
<td>N/A</td>
<td>N/A</td>
<td>3224</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>HLSv2</td>
<td>1169</td>
<td>0</td>
<td>11</td>
<td>424</td>
<td>4520</td>
<td>3.87</td>
</tr>
<tr>
<td><strong>Xilinx Spartan 6</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td>354</td>
<td>0</td>
<td>11</td>
<td>230</td>
<td>2943</td>
<td>8.31</td>
</tr>
<tr>
<td>HLSv0</td>
<td>374</td>
<td>0</td>
<td>7367</td>
<td>183</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td>HLSv1</td>
<td>211</td>
<td>0</td>
<td>3224</td>
<td>193</td>
<td>8</td>
<td>0.04</td>
</tr>
<tr>
<td>HLSv2</td>
<td>343</td>
<td>0</td>
<td>11</td>
<td>231</td>
<td>2467</td>
<td>7.19</td>
</tr>
<tr>
<td><strong>Xilinx Virtex 7</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td>317</td>
<td>0</td>
<td>11</td>
<td>336</td>
<td>4298</td>
<td>13.56</td>
</tr>
<tr>
<td>HLSv0</td>
<td>318</td>
<td>1</td>
<td>7367</td>
<td>265</td>
<td>5</td>
<td>0.01</td>
</tr>
<tr>
<td>HLSv1</td>
<td>236</td>
<td>1</td>
<td>3224</td>
<td>334</td>
<td>13</td>
<td>0.06</td>
</tr>
<tr>
<td>HLSv2</td>
<td>344</td>
<td>0</td>
<td>11</td>
<td>352</td>
<td>3760</td>
<td>10.93</td>
</tr>
</tbody>
</table>
Conclusions

- Area and frequency of designs produced by High-Level Synthesis are comparable to handwritten RTL code.
- Small increase in the number of clock cycles reduces throughput of HLS-based approach.
- Complex I/O units can be created by HLS-based approach.
- HLS-based design can compete against handwritten RTL code when we have a specific architecture and latency in mind while preparing an HLS-ready HLL code.
Hardware Benchmarking of SHA-3 Finalists using High-Level Synthesis

Ekawat Homsirikamol & Kris Gaj
George Mason University
Our Test Case

- 5 final SHA-3 candidates + old standard SHA-2
- Most efficient sequential architectures
  (/2h for BLAKE, x4 for Skein, x1 for others)
- GMU VHDL codes developed during SHA-3 contest
- Reference software implementations in C included in the submission packages

Hypotheses:

- Ranking of candidates will remain the same
- Performance ratios HDL/HLS similar across candidates
Manual RTL vs. HLS-based Results: Altera Stratix III
Manual RTL vs. HLS-based Results: Altera Stratix IV

![Graph showing throughput vs. area for manual RTL and HLS](image)

**RTL**

**HLS**
Lack of Correlation for Xilinx Virtex 6
Lack of Correlation for Xilinx Virtex 6

**Diagram Description:**
- **X-axis:** Area
- **Y-axis:** Throughput
- **Legend:**
  - **BLAKE** (Red Circle)
  - **Groestl** (Green Circle)
  - **JH** (Light Blue Circle)
  - **Keccak** (Purple Square)
  - **Skein** (Blue Square)
  - **SHA-2** (Black Triangle)

**Comparative Analysis:**
- **RTL** cluster shows lower throughput compared to **HLS**.
- **HLS** generally exhibits higher throughput across different algorithms.

**Implications:**
The lack of correlation suggests that **RTL** implementations do not scale as effectively as their **HLS** counterparts, particularly in terms of throughput efficiency.
Lack of Correlation for Xilinx Virtex 7

[Graph showing throughput vs. area for different algorithms in RTL and HLS]
Ratios of Major Results RTL/HLS for Altera Stratix IV
Ratios of Major Results RTL/HLS for Xilinx Virtex 6
Datapath vs. Control Unit

Datapath

- Data Inputs
  - Data Outputs

Control Unit

- Control Inputs
  - Control Signals
  - Status Signals
  - Control Outputs

Determines
- Area
- Clock Frequency

Determines
- Number of clock cycles
Encountered Problems

Datapath inferred correctly

• Frequency and area within 30% of manual designs

Control Unit suboptimal

• Difficulty in inferring an overlap between completing the last round and reading the next input block
• One additional clock cycle used for initialization of the state at the beginning of each round
• The formulas for throughput:
  
  RTL: Throughput = Block_size / (#Rounds * T_{CLK})
  
  HLS: Throughput = Block_size / ((#Rounds+2) * T_{CLK})
Hypothesis I:
- Ranking of candidates in terms of throughput, area, and throughput/area ratio will remain the same
  - TRUE for Altera Stratix III, Stratix IV
  - FALSE for Xilinx Virtex 5, Virtex 6, and Virtex 7

Hypothesis II:
- Performance ratios HDL/HLS similar across candidates

<table>
<thead>
<tr>
<th></th>
<th>Stratix III</th>
<th>Stratix IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.99-1.30</td>
<td>0.98-1.19</td>
</tr>
<tr>
<td>Area</td>
<td>0.71-1.01</td>
<td>0.68-1.02</td>
</tr>
<tr>
<td>Throughput</td>
<td>1.10-1.33</td>
<td>1.09-1.27</td>
</tr>
<tr>
<td>Throughput/Area</td>
<td>1.14-1.55</td>
<td>1.17-1.59</td>
</tr>
</tbody>
</table>
Correlation Between Altera FPGA Results and ASICs

Stratix III FPGA

ASIC
Most Promising Methodology & Toolset

Reference Implementation in C

Manual Modifications

HLS-ready C code

High-Level Synthesis
Xilinx Vivado HLS

HDL Code

Option Optimization
GMU ATHENa

FPGA Tools
Altera Quartus II

Results

Frequency & Throughput decrease
Area increases
by no more than 30%
compared to manual RTL