Vivado High-Level Synthesis Meet-Up June 18th 2012

Tom Schwing
Alex Paek
Vivado High-Level Synthesis Serves a Wide Range of Applications across Markets

Aerospace and Defense
- Radar, Sonar
- Signals Intelligence

Communications
- LTE MIMO receiver
- Advanced wireless antenna positioning

Industrial, Scientific, Medical
- Ultrasound systems
- Motor controllers

Audio, Video, Broadcast
- 3D cameras
- Video transport

Automotive
- Infotainment
- Driver assistance

Consumer
- 3D television
- eReaders

Test & Measurement
- Communications instruments
- Semiconductor ATE

Computing & Storage
- High performance computing
- Database acceleration

© Copyright 2013 Xilinx
The First **SoC Strength Design Suite**

IP & System Centric Next Generation Design Environment

Accelerating Integration & Implementation up to 4X
Vivado High-Level Synthesis (HLS)

Accelerate Algorithmic C to IP Integration

C, C++ or SystemC

Vivado™ HLS

VHDL or Verilog

System IP Integration

Algorithmic Specification

Micro Architecture Exploration

RTL Implementation

Comprehensive Integration with the Xilinx Design Environment

Accelerate Algorithmic C to Co-Processing Accelerator Integration

SW Spec

Iterate

Verify

HW Spec

Iterate

Verify

Available in production today for C, C++, systemC

– Proven on real customer designs

– Clear differentiator for accelerating design productivity

– Adopted across broad base of applications and markets

© Copyright 2013 Xilinx
Vivado HLS System IP Integration Flow

C based IP Creation

C, C++ or SystemC

VHDL or Verilog

C Libraries
- math.h
- Video
- DSP (2H13)

Vivado™ HLS

IP Catalog

User Preferred System Integration Environment

System Generator for DSP

Vivado IP Integrator

Vivado RTL Integration

7 Series FPGA and Zynq SoC Vivado Implementation

© Copyright 2013 Xilinx
Vivado HLS System IP Integration Flow

- **C based IP Creation**
- **Accelerator pcore**
- **Sys. Gen. Block**
- **RTL**

**User Preferred System Integration Environment**
- **Xilinx Platform Studio**
- **System Generator for DSP**
- **ISE**

**7 Series, 6 Series, Virtex-5, Virtex-4, Spartan-3 FPGA and Zynq SoC ISE Implementation**

© Copyright 2013 Xilinx
Accelerate Verification by >100X with C/C++

Optical flow video example

<table>
<thead>
<tr>
<th>Input</th>
<th>RTL Simulation Time</th>
<th>C Simulation Time</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 frames of video data</td>
<td>~2 days</td>
<td>10 seconds</td>
<td>~12,000X</td>
</tr>
</tbody>
</table>

*RTL Simulations performed using ModelSim
C to Verified RTL from Months to Weeks

“In an **HDL** design, each scenario would likely cost an **additional day of writing code** …

With Vivado **HLS** these changes took minutes”

_Nathan Jachimiec, R&D Engineer, Agilent Technologies_

“I was able to design complex linear algebra algorithms **10x faster** than before with VHDL, and yet achieved **better QoR** with Vivado HLS.”

_Design Engineer, Major A&D contractor_

“..we always use **C** to quickly build a **system-level model** for validation of **key algorithms**. .. problem .. quickly and efficiently convert C into a **HDL**”.

_Hengqi Liu, Central R&D Data Center CTO, ZTE Inc._

---

### Radar Design

<table>
<thead>
<tr>
<th>1024 x 64 QRD Floating-Point data path</th>
<th>Conventional Hand-coded HDL Approach</th>
<th>Using Vivado High Level Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Language</strong></td>
<td>VHDL (RTL)</td>
<td>C</td>
</tr>
<tr>
<td><strong>Design Time (weeks)</strong></td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td><strong>Latency (ms)</strong></td>
<td>37</td>
<td>21</td>
</tr>
<tr>
<td><strong>Memory (RAMB18E1)</strong></td>
<td>134 (16%)</td>
<td>10 (1%)</td>
</tr>
<tr>
<td><strong>Memory (RAMB36E1)</strong></td>
<td>273 (65%)</td>
<td>138 (33%)</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>29686 (9%)</td>
<td>14263 (4%)</td>
</tr>
<tr>
<td><strong>LUTs</strong></td>
<td>28152 (18%)</td>
<td>24257 (16%)</td>
</tr>
</tbody>
</table>

**Source:** Design Engineer at Major A&D contractor

“For each project where we used Vivado HLS, we **saved 2-3 weeks** of engineering time.”

_CTO, Major broadcast equipment company_
C Synthesis Design Flow Overview

- Overview of HLS
- Design Example
HLS Premise:
One C Code – Multiple HW Implementations

One body of code:
Many hardware outcomes

The same hardware is used for each iteration of the loop:
- Small area
- Long latency
- Low throughput

Before going into details, let’s look under the hood ....

Different hardware is used for each iteration of the loop:
- Higher area
- Short latency
- Better throughput

Different iterations are executed concurrently:
- Higher area
- Short latency
- Best throughput

Default Design

Unrolled Loop Design

 Pipelined Design
Attributes of a Program for Synthesis

- **Functions**
  - Functions define hierarchy and control regions

- **Function Parameters:**
  - Define the RTL I/O Ports

- **Types:**
  - Data types define bitwidth requirements
  - HLS optimizes bitwidth except for function parameters

- **Loops:**
  - Define iterative execution regions that can share HW resources

- **Arrays:**
  - Main way of defining memory and data storage

- **Operators:**
  - Implementations optimized for performance
  - Automatically shared where possible to reduce area
Combining Control and Operations

From any C code example...

Operations are extracted...

The control is known

A unified control dataflow behavior is created

© Copyright 2013 Xilinx
Optimizing and Sizing Program Operations

```c
void fir(
    data_t *y,
    coef_t c[4],
    data_t x
){
    static data_t shift_reg[4];
    acc_t acc;
    int i;
    acc=0;
    loop: for (i=3;i>=0;i--){
       if (i==0) {
         acc+=x*c[0];
         shift_reg[0]=x;
       } else {
         shift_reg[i]=shift_reg[i-1];
         acc+=shift_reg[i]*c[i];
       }
    }
    y=acc;
}
```

**Code**

**Operations**
- RDx
- RDC
- >=
- -
- ==
- +
- *
- *
- WRy

**Types**

**Standard C Types**
- long long (64-bit)
- short (16-bit)
- unsigned types
- int (32-bit)
- char (8-bit)
- float (32-bit)
- double (64-bit)

For floats and doubles, there must be an FP core in the library binding can map to; else cannot be synthesized.

**Arbitrary Precision Types**

- C:
  - (u)inttypes (1-1024)
- C++:
  - ap_(u)inttypes (1-1024)
  - ap_fixedtypes
- C++/SystemC:
  - sc_(u)inttypes (1-1024)
  - sc_fixedtypes

Can be used to define any variable to be a specific bit width (e.g., 17-bit, 47-bit, etc.)

From any C code example

Operations are extracted...

The C types define the size of the hardware used: handled automatically

© Copyright 2013 Xilinx
Completing a Design – I/O Port Creation

- Function parameters define data I/O ports and default protocols
  - Pointers → AXI4-Master interface
  - Scalars → AXI4-Lite interface or raw wires
  - Arrays → AXI4-Lite or AXI4 stream interface

- Protocol in generated HW are controlled through user directives

```c
void foo_top(int *in1, int *in2, int *out1) {
    *out1 = *in1 + *in2;
}
```
Two steps to verifying the design
- Pre-synthesis: C validation
- Post-synthesis: RTL verification

C validation
- Fast and free verification on any Operating System
- Prove algorithm correctness before RTL generation

RTL Verification
- RTL Co-Simulation against the original program testbench
Coding Restrictions

- **Data Types**
  - Forward declared data types
  - Recursive type definitions

- **Pointers**
  - General casting between user defined data types
  - Pointers to dynamically allocated memory regions

- **System Calls**
  - Dynamic memory allocation – must be replaced with static allocation
  - Standard I/O and file I/O – automatically ignored by the compiler
  - System calls: time(), sleep(), …

- **Recursive functions that are not compile time bounded**

- **STL lib calls**
  - Not supported due to dynamic memory allocation
  - Have compile time unbounded recursion
VHLS Design Example
Matrix Multiplication

- Each output element = dot product (each row of A, each col of B)

\[
\begin{array}{ccc}
A & \times & B \\
\begin{array}{ccc}
11 & 12 & 13 \\
14 & 15 & 16 \\
17 & 18 & 19 \\
\end{array} & \times & \begin{array}{ccc}
21 & 22 & 23 \\
24 & 25 & 26 \\
27 & 28 & 29 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{ccc}
C \\
\begin{array}{ccc}
870 & 906 & 942 \\
1086 & 1131 & 1176 \\
1302 & 1356 & 1410 \\
\end{array} \\
\end{array}
\]

\[
\begin{align*}
r_{00} &= 11 \times 21 + 12 \times 24 + 13 \times 27 \\
&= 870 \\
r_{01} &= 11 \times 22 + 12 \times 25 + 13 \times 28 \\
&= 906 \\
\end{align*}
\]

Etc…
Design Goals

- Matrix size: 16x16
- Complex elements of 16 bit integer
- One output element per clock cycles, i.e., overall throughput of ~256 cycles per matrix multiplication operation
- Using version 2013.1 Vivado HLS
- Target device: V7 485 -2 “mid speed” grade
- Target clock rate: <3 ns (333 Mhz)

To consider:
- FPGA resources, IO size
- Optimization process, trading resource with speed
- Fixed point analysis
- Coding style, advantage of using C++ for class, templates, overloading
```c
#include "mat_mult.h"

void mat_mult(complex_t c[M][M], complex_t a[M][M], complex_t b[M][M])
{
    int i, j, k;
    complex_t tmp;
    fx_data_t ar, ai, br, bi, cr, ci;

    loop_i:
        for (i = 0; i < M; i++) {
            loop_j:
                for (j = 0; j < M; j++) {
                    tmp.r = 0;
                    tmp.i = 0;

                    loop_k:
                        for (k = 0; k < M; k++) {
                            // #pragma AP PIPELINE
                            ar = a[i][k].r;
                            ai = a[i][k].i;
                            br = b[k][j].r;
                            bi = b[k][j].i;
                            cr = ar*br - ai*bi;
                            ci = ar*bi + ai*br;
                            tmp.r += cr;
                            tmp.i += ci;
                        }
                    c[i][j].r = tmp.r;
                    c[i][j].i = tmp.i;
                }
            }
        }

    #ifndef FIXED
    #define FIXED
    // #pragma AP PIPELINE
    // #define FIXED
    // typedef ap_fixed<16,16, AP_RND_CONV, AP_SAT> fx_data_t;
    // typedef ap_fixed<16,16> fx_data_t;
    typedef ap_int<16> fx_data_t;
    #else
    typedef float fx_data_t;
    #endif

    struct complex_t {
        fx_data_t r, i;
    };

    // dimension of square matrix
    #define SMALL
    #ifdef SMALL
        #define M 4
    #else
        #define M 16
    #endif

    void mat_mult(complex_t c[M][M], complex_t a[M][M], complex_t b[M][M]);
    #endif
```
Results (fixed point)

- Results after implementation
  - One output per cycle, latency of 13
  - Clock period = 2.63 ns
Results (floating point)

- The same code as the fixed point except the variables are declared float or double type
- Automatically instantiates Xilinx floating point cores
- Results after implementation
  - One output per cycle, latency of 71
  - Clock period = 2.7 ns
Initiation Interval (II)

Definition: the number of clock cycles between new input samples (applied to loops)

- II=1: one loop body per clock cycle
  - a ‘fully pipelined’ datapath for the loop body

- II=2: one loop body every 2 clock cycles
  - Allows for resource sharing of operators.
Optimizing the loop: pipelining

- run sequentially
  - Throughput = 3 clock cycles
  - Latency
    - 3 cycles per iteration
    - 6 cycles for entire loop

- run in parallel
  - Throughput = 1 clock cycle
  - Latency
    - 3 cycles per iteration
    - 4 cycles for entire loop
Data access

- Increase the data bandwidth by reshaping the input array
- No change to the original C code by use of synthesis directives

Read modify write is not allowed: read the whole word or write the whole word

Similar to converting a RAM into a very wide register: great access, high throughput
Applying synthesis directives

```c
#include "mat_mult.h"

void mat_mult(complex_t c[M][M], complex_t a[M][M], complex_t b[M][M])
{
    int i,j,k;
    complex_t tmp;
    fx_data_t ar,ai, br,bi, cr, ci;

    loop_i:
    for (i=0;i<M;i++) {
        loop_j:
        for (j=0;j<M;j++) {
            tmp.r = 0;
            tmp.i = 0;

            loop_k:
            for (k=0;k<M;k++) {
                //pragma AP PIPELINE
                ar = a[i][k].r;
                ai = a[i][k].i;
                br = b[k][j].r;
                bi = b[k][j].i;

                cr = ar*br - ai*bi;
                ci = ar*bi + ai*br;

                tmp.r += cr;
                tmp.i += ci;
            }
            c[i][j].r = tmp.r;
            c[i][j].i = tmp.i;
        }
    }
}
```

```c
#ifndef mat_mult_H_
#define mat_mult_H_

//include <math.h>
//float sqrtf(float);

#include "ap_int.h"
#include "ap_fixed.h"

#define FIXED
#ifndef FIXED
//typedef ap_fixed<16,16, AP_RND_CONV, AP_SAT> fx_data_t;
#endif
typedef ap_int<16> fx_data_t;

#define float fx_data_t;
#endif

struct complex_t {
    fx_data_t r, i;
};

// dimension of square matrix
#define SMALL
#ifndef SMALL
# define M 4
#else
# define M 16
#endif
#endif

void mat_mult(complex_t c[M][M], complex_t a[M][M], complex_t b[M][M]);
```

© Copyright 2013 Xilinx
The generated module becomes a memory master, interfacing the external (to the module) BRAM.

The memory interface can be “translated” to AXI interface, to interface to processors - Zynq ARM and microblaze.
Vivado HLS Resources
Where to Learn More

**Xilinx.com**
- Vivado HLS Landing Page
  - Quick Links
  - Key Documents (White Papers, User Guides, Tutorials)
  - Application Notes

**Training**
- Training Landing Page
- Vivado HLS
  - C-Based Design: High Level Synthesis with Vivado HLS
  - C-Based HLS Coding for Hardware Designers
  - C-Based HLS Coding for Software Designers
- Classes also available for:
  - Vivado Design Suite Tool
  - All Programmable 7-Series FPGAs and Zynq SOCs
  - Languages (VHDL, Verilog, Tcl, XDC)

**Xilinx Authorized Training Provider**
- Bottom Line Technologies
  - Training Class Locations
    - Parsippany, NJ
    - Hauppauge, NY
    - Rochester, NU
    - Marlton, NJ
    - Columbia, MD
    - Private ‘in-house’ classes also available
On-demand webinars

» C code to Co-processing Accelerator on Zynq-7000 All Programmable SoCs

» Floating-point C based blocks in System Generator for DSP
Quick-take Videos:  [www.xilinx.com/training/vivado](http://www.xilinx.com/training/vivado)

1. **Getting Started with Vivado High-Level Synthesis**
2. **Verifying your Vivado HLS Design**
3. **Packaging Vivado HLS IP for use from Vivado IP Catalog**
4. **Generating Vivado HLS block for use in System Generator for DSP**
5. **Generating Vivado HLS pcore for use in Xilinx Platform Studio**
6. **Analyzing your Vivado HLS design**
7. **Specifying AXI4 interfaces for your Vivado HLS design**
8. **Using Vivado HLS C/C++/SystemC block in System Generator**
9. **Using Vivado HLS C/C++/SystemC based pcores in XPS**
10. **Floating-Point Design with Vivado HLS**
11. **Using Vivado HLS SW libraries in your C, C++, SystemC code**
12. **Using the Vivado HLS Tcl interface**
13. **Leveraging OpenCV and High Level Synthesis with Vivado**
Application Notes: [www.xilinx.com/hls](http://www.xilinx.com/hls)

- XAPP599 Floating Point Design with Vivado HLS
- XAPP745 Processor Control of Vivado HLS Designs
- XAPP793 Implementing Memory Structures for Video Processing in the Vivado HLS Tool
- XAPP890 Zynq All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool
- XAPP1167 Accelerating OpenCV Applications with Zynq using Vivado HLS Video Libraries
- XAPP1163 - Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP
Articles

- Using OpenCV and Vivado™ HLS to Accelerate Embedded Vision Applications in the Zynq SoC
- Xilinx High-Level Synthesis Tool Speeds FPGA Design
- Software programmable Digital Predistortion on Zynq SoC
- Floating-Point Design with Xilinx’s Vivado HLS
- Vivado HLS Eases Design of Floating-Point PID Controller
- Vivado HLS: Agilent packet engine case study
- Using HLS and Programmable SoCs to Drive Real-Time Digital Signal Processing
Thank You