NEW FPGA DESIGN AND VERIFICATION TECHNIQUES

MICHAL HUSEJKO
IT-PES-ES
Design:

- Part 1 – High Level Synthesis (Xilinx Vivado HLS)
- Part 2 – SDSoC (Xilinx, HLS + ARM)
- Part 3 – OpenCL (Altera OpenCL SDK)

Verification:

- Part 4 – SystemVerilog and Universal Verification Methodology (UVM)
- Part 5 – Automatic build systems and Continuous Integration
- Part 6 – Open Source VHDL Verification Methodology (OSVVM)
NEW FPGA DESIGN AND VERIFICATION TECHNIQUES

PART 1 – HIGH-LEVEL SYNTHESIS (XILINX VIVADO HLS)
AGENDA

• Xilinx High-Level Productivity Design Methodology
• Case study 1 – Matrix Multiplication in FPGA (Physics)
• Overview of Vivado HLS tool
• HLS optimization methodology
• Case study 2 – CMS ECAL Data Concentrator Card (DCC)
• Conclusions
AGENDA

• Xilinx High-Level Productivity Design Methodology
• Case study 1 – Matrix Multiplication in FPGA (Physics)
• Overview of Vivado HLS tool
• HLS optimization methodology
• Case study 2 – CMS ECAL Data Concentrator Card (DCC)
• Conclusions
HIGH-LEVEL PRODUCTIVITY DESIGN METHODOLOGY

Xilinx UG1197, Figure 1-1
SYSTEM DESIGN

• System partitioning
  • Platform IP (I/O logic, pre/post-processing)
  • Design IP

• Platform design
  • Separation of Platform and Design development
  • Creating re-usable design, or platform to quickly create derivatives
IP DESIGN

• The key productivity benefit is being able to simulate as many C IP blocks as one C simulation during development.

• IP developed from C/C++ is verified using the C/RTL co-simulation feature of Vivado HLS, allowing the RTL to be verified using the same C test bench used to verify the C test bench.

• IP developed from System Generator is verified using the MathWorks Simulink design environment provided in System Generator.

• For IP generated from RTL, you must create an RTL test bench to verify the IP.
AGENDA

• Xilinx High-Level Productivity Design Methodology
• **Case study 1 – Matrix Multiplication in FPGA (Physics)**
• Overview of Vivado HLS tool
• HLS optimization methodology
• **Case study 2 – CMS ECAL Data Concentrator Card (DCC)**
• Conclusions
CASE STUDY 1 – PLATFORM FOR PHYSICS SIMULATIONS ON A FPGA

• FPGA accelerator for physics simulations

• Team work:
  • Platform Integrator: Michal
  • Design IP: Wojtek

• Initial algorithm: matrix multiplication on a FPGA
CASE STUDY 1

- Platform Integrator: Michal
  - Developing and integrating IPs.
    - RIFFA (Verilog) – An open source IP from UCSA
    - RIFFA to AXIS/HLS bridge (VHDL) – “developed” by Michal
  - Platform testbenches written in SystemVerilog
  - Few days of work (while learning RIFFA)
- Design IP: Wojtek
  - Developing Design IP and handing it over to Platform integrator
  - Matrix multiplication (C++) – based on XAPP 1170 from Xilinx
  - IP testbench written in C++
  - Few hours of work (while learning HLS)
CASE STUDY 1 - HLS DESIGN SPACE EXPLORATION

• More details in XAPP 1170
CASE STUDY 1

- Wojtek is interested in physics and not in VHDL/PCIe/FPGA/etc.
- A platform with container for HLS core has been developed for him.
- Verified with matrix multiplication
- Now that platform is verified it can be re-used for more complicated algorithms.
- Design IP designed and verified before Platform was ready thanks to HLS
AGENDA

• Xilinx High-Level Productivity Design Methodology
• Case study 1 – Matrix Multiplication in FPGA (Physics)
• **Overview of Vivado HLS tool**
• HLS optimization methodology
• Case study 2 – CMS ECAL Data Concentrator Card (DCC)
• Conclusions
INTRODUCTION TO VIVADO HLS

• High-Level Synthesis
  • Creates an RTL implementation from C level source code
  • Implements the design based on defaults and user applied directives
• Many implementation are possible from the same source description
  • Smaller designs, faster designs, optimal designs
  • Enables design exploration
C LANGUAGE SUPPORT

• Vivado HLS provides comprehensive support for C, C++, and SystemC. Everything is supported for C simulation; however, it is not possible to synthesize every description into an equivalent RTL implementation.

• The two key principles to keep in mind when reviewing the code for implementation in an FPGA are:
  • An FPGA is a fixed size resource. The functionality must be fixed at compile time. Objects in hardware cannot be dynamically created and destroyed.
  • All communication with the FPGA must be performed through the input and output ports. There is no underlying Operating System (OS) or OS resources in an FPGA.
UNSUPPORTED CONSTRUCTS

• Synthesis does not support ...
• Systems calls: abort(), exit(), printf(), etc
• Dynamic objects: malloc(), alloc(), free(), new, delete
CONSTRUCTS WITH LIMITED SUPPORT

• Top-level function: templates are supported for synthesis but not for a top level function
• Pointer supports: some limitations to pointer casting and pointer arrays
• Recursion: supported through use of templates, you have to use termination class.
• Memory functions: memcpy() abd memset() supported but only with cost values.

• Any code which is not supported for synthesis, or for which only limited support is provided, must be modified before it can be synthesized
HARDWARE OPTIMIZED C LIBRARIES

• Arbitrary Precision Data Types
• HLS Stream Library
• Math Functions
• Linear Algebra Functions
• DSP Functions
• Video Functions
• IP Library
HIGH LEVEL SYNTHESIS BASICS

• High-level synthesis includes the following phases:
  • Control logic extraction
  • Scheduling
  • Binding

• High-level synthesis synthesizes the C code as follows
  • Top-level function arguments synthesize into RTL I/O ports
  • C functions synthesis into blocks in the RTL hierarchy
  • Loops in the C functions are kept rolled by default
  • Arrays in the C code synthesize into block RAM
Design Source
(C, C++, SystemC)

Scheduling

Technology Library

User Directives

Binding

RTL
(Verilog, VHDL)
CONTROL LOGIC EXTRACTION

```c
void foo(int in[3], char a, char b, char c, int out[3]) {
    int x, y;
    for(int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x + b + c;
        out[i] = y;
    }
}
```
SCHEDULING AND BINDING

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```
UNDERSTANDING VIVADO HLS
VIVADO HLS DESIGN FLOW

• Compile, execute (simulate), and debug the C algorithm
  • Note: In high-level synthesis, running the compiled C program is referred to as C simulation. Executing the C algorithm simulates the function to validate that the algorithm is functionally correct.

• Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives

• Generate comprehensive reports and analyse the design

• Verify the RTL implementation using pushbutton flow

• Package the RTL implementation into a selection of IP formats
IMPORTANCE OF TESTBENCH

• Post-synthesis verification is automated through the C/RTL co-simulation feature which reuses the pre-synthesis C test bench to perform verification on the output RTL

• The following is required to use C/RTL co-simulation feature successfully:
  • The test bench must be self-checking and return a value of 0 if the test passes or returns a non-zero value if the test fails
  • The correct interface synthesis options must be selected
  • Any simulators must be available in the search path
AGENDA

• Xilinx High-Level Productivity Design Methodology
• Case study 1 – Matrix Multiplication in FPGA (Physics)
• Overview of Vivado HLS tool
• HLS optimization methodology
• Case study 2 – CMS ECAL Data Concentrator Card (DCC)
• Conclusions
HLS OPTIMIZATION METHODOLOGY

- Default constraints are usually leading to an RTL which is not exactly what you want ...
  - Constraints can be provided either as Tcl constraints file or as pragmas inside C/C++ source file
- Step 1 – Initial optimizations
- Step 2 – Pipeline for performance
- Step 3 – Optimize structures for performance
- Step 4 – Reduce latency
- Step 5 – Reduce area
STEP 1 – INITIAL OPTIMIZATIONS

- **INTERFACE** – specifies how RTL ports are created from function description
- **DATA_PACK** – packs the data fields of a struct into a single scalar with a wider word width
- **LOOP_TRIPCOUNT** – used for loops which have variable bounds
# INTERFACE

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Scalar</th>
<th>Array</th>
<th>Pointer or Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Mode</td>
<td>Input</td>
<td>Return</td>
<td>I</td>
</tr>
<tr>
<td><strong>ap_ctrl_none</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_ctrl_hs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_ctrl_chain</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>axis</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_axiite</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>m_axi</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_none</strong></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td><strong>ap_stable</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_ack</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_vld</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_ovld</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_hs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_memory</strong></td>
<td></td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td><strong>bmem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_fifo</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ap_bus</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STEP 2 – PIPELINE FOR PERFORMANCE

- **PIPELINE** – allow concurrent execution of operations with a loop or function
- **DATAFLOW** – enables task level pipelining, allowing functions and loops to execute concurrently
- **RESOURCE** – specifies a resource to implement a variable in the RTL
void func(...) {
    op_Read;
    op_Compute;
    op_Write;
}

(A) Without Function Pipelining

(B) With Function Pipelining
STEP 3 – OPTIMIZE STRUCTURES FOR PERFORMANCE

• **ARRAY_PARTITION** – partitions large arrays into multiple smaller arrays or into individual registers

• **DEPENDENCE** - used to provide additional information that can overcome loop-carry dependencies

• **INLINE** – inlines function, removing all function hierarchy. Used to enable logic optimization across function boundaries.

• **UNROLL** – Unroll for-loops
ARRAY_PARTITION

block
0 1 ... (N/2-1)
N/2 ... N-2 N-1

cyclic
0 2 ... N-2
1 ... N-3 N-1

complete
0 ...
N-3
1 ...
N-2
N-1
...
2

my_array[10][6][4] → partition dimension 3
  my_array_0[10][6]
  my_array_1[10][6]
  my_array_2[10][6]
  my_array_3[10][6]

my_array[10][6][4] → partition dimension 1
  my_array_[16][4]
  my_array_16[4]
  my_array_26[4]
  my_array_36[4]
  my_array_46[4]
  my_array_56[4]
  my_array_66[4]
  my_array_76[4]
  my_array_86[4]
  my_array_96[4]

my_array[10][6][4] → partition dimension 0
  10x6x4 = 240 registers

my_array[10][6][4] → partition dimension 0
  10x6x4 = 240 registers
void top(...) {
  ...
  for_mult:for (i=3;i>0;i--) {
    a[i] = b[i] * c[i];
  }
  ...
}
STEP 4 – REDUCE LATENCY

• **LATENCY** – allows a minimum and maximum latency constraint to be specified
• **LOOP_FLATTEN** – allows nested loops to be collapsed
• **LOOP_MERGE** – merge consecutive loops and reduce overall latency
```c
void top (a[4], b[4], c[4], d[4]...) {
    ...
    Add: for (i=3; i>=0; i--) {
        if (d[i])
            a[i] = b[i] + c[i];
    }
    ...
    Sub: for (i=3; i>=0; i--) {
        if (!d[i])
            a[i] = b[i] - c[i];
    }
    ...
}
```

Diagram:

(A) Without Loop Merging

1 cycle

1 cycle

4 cycles

4 cycles

(B) With Loop Merging

1 cycle

1 cycle

4 cycle

1 cycle

4 cycle
STEP 5 – REDUCE AREA

• **ALLOCATION** – specifies a limit for the number of operations, cores or functions
• **ARRAY_MAP** – combines multiple smaller arrays into a single large array
• **ARRAY_RESHAPE** – reshapes an array from one with many elements to one with greater word-width
• ... and more ...
dout_t array_arith (dio_t d[317]) {
    static int acc;
    int i;
    #pragma HLS ALLOCATION instances=mul limit=256 operation
    for (i=0; i<317; i++) {
        #pragma HLS UNROLL
        acc += acc * d[i];
    }
    rerun acc;
}
AGENDA

- Xilinx High-Level Productivity Design Methodology
- Case study 1 – Matrix Multiplication in FPGA (Physics)
- Overview of Vivado HLS tool
- HLS optimization methodology
- Case study 2 – CMS ECAL Data Concentrator Card (DCC)
- Conclusions
CASE STUDY 2 – CMS ECAL DATA CONCENTRATOR CARD (DCC)
DCC – PRODUCTION SYSTEM

• Firmware in: 9x VirtexII Pro, 2x Stratix, and 1x Acex FPGAs
• Production design described in mixture of SystemVerilog, VHDL and Quartus Schematics
• DCC design SV/VHDL ~ 17’500 lines of code
• DCC testbench in SV ~ 3000 lines of code
DCC FIRMWARE – HLS IMPLEMENTATIONS

• Targeted for Zynq and Virtex-7 FPGA devices

• Written in C and C++ languages, and compiled to Verilog, then instantiated inside FPGA as a single component and connected to Platform (PCIe, VC709) through AXIS interfaces.

• Do not include some other functionality of a production DCC (TCC, TTS, VME).
DCC HLS – TESTING PLATFORM

• Re-used MMULT platform for VC709
• Performed DCC HLS functional tests in hardware
DCC HLS

• DCCv1 HLS design:
  • Contains around ~ 1000 lines of code + 30 pragmas
  • Code was not modified after initial coding, only additional compiler pragmas were added (inside external pragma file) for design space exploration

• DCCv2 HLS design – complete code rewrite of DCCv1
  • Uses data streaming interfaces instead of arrays (DCCv1)
  • Contains around ~1000 lines of code and 20 pragmas
  • Coding style was tailored towards processing of data streams
void dcc_top (...) {

dcc_sr(...) 

for(i=0; i<TOWER_NUM; i++)
    dcc_ih (...)

dcc_em(...) 

dccEb(...) 
}
DCCV1 HLS – STEP 1 – DEFAULT HLS CONSTRAINTS

- Serial implementation
- C functions synthetized into HDL hierarchical blocks
- No initiation interval specified, minimize latency then minimize area.
- Loops are “rolled” – serial execution
- Arrays synthetized into BRAMs
- Serial execution of tasks – very high latency, but small device utilization
DCCV1 HLS – STEP 2 – PARALLELIZE TASKS

- Execute tasks in parallel - Loop unrolling to create multiple independent operations, rather than single collection of operations
DCCV1 HLS – STEP 3 – PIPELINE FUNCTIONS

- Loop pipelining - concurrently execute the operations
- Loop flattening - flatten nested loops
- Loop rewinding - if the loop runs "continuously", rewind consecutive appearances to fill the gaps
DCCV1 HLS – STEP 4 – PARTITION ARRAYS

• FPGA has thousands of dual port BRAM memories – utilize them to improve throughput (more RAM ports, vectorized operations) and lower latency

• Step 3 + Apply array partitioning on internal arrays, splitting single array onto Nx BRAMs, virtually creating N port BRAM
DCCV1 HLS – STEP 5 – PIPELINE TASKS

- Pipeline tasks’ execution
- Partially overlapping computations
DCCV1 HLS – RESULTS

- All design space exploration done with HLS compiler directives stored in external Tcl file
- Not a single line of C code was changed

<table>
<thead>
<tr>
<th>Design space exploration</th>
<th>Latency</th>
<th>Init Inter</th>
<th>BRAM</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step1</td>
<td>26590</td>
<td>26590</td>
<td>12</td>
<td>4</td>
<td>988</td>
<td>1951</td>
</tr>
<tr>
<td>Step2</td>
<td>7960</td>
<td>7960</td>
<td>20</td>
<td>7</td>
<td>3555</td>
<td>6089</td>
</tr>
<tr>
<td>Step3</td>
<td>5192</td>
<td>5192</td>
<td>20</td>
<td>350</td>
<td>22203</td>
<td>27385</td>
</tr>
<tr>
<td>Step4</td>
<td>1734</td>
<td>1734</td>
<td>52</td>
<td>351</td>
<td>25966</td>
<td>25091</td>
</tr>
<tr>
<td>Step5</td>
<td>1443</td>
<td>603</td>
<td>104</td>
<td>401</td>
<td>29999</td>
<td>28366</td>
</tr>
<tr>
<td>FPGA Device utilization (%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>
DCCV2 HLS – COMPLETE CODE REWRITE

- Interfaces: multi-dimensional arrays converted to hls::stream
- All functions rewritten – migrated from loops (FOR) to FSMs (SWITCH)
- Resource usage (8x FE channels): LL/FF=4k, DSP=8, BRAM=0;
AGENDA

- Xilinx High-Level Productivity Design Methodology
- Case study 1 – Matrix Multiplication in FPGA (Physics)
- Overview of Vivado HLS tool
- HLS optimization methodology
- Case study 2 – CMS ECAL Data Concentrator Card (DCC)
- Conclusions
SUMMARY

• It seems that Vivado HLS is working 😊
  • Proven with some algebra (mmult) and DSP (FIR)
  • Does also work for packet processing
• The tool has still some bugs, which are blocking full adoption of High-Level Productivity Design Methodology (i.e. array of hls::stream)
• If there is an interest in community we could try to organize some training
HLS LEARNING RESOURCES/TRAINING

• Vivado High-Level Productivity Design Methodology Guide (UG11977)
• Vivado HLS User Guide (UG902)
• Vivado HLS Tutorial (UG871)
• Application notes (XAPP 1170, 1209)
• Vivado Design Suite Puzzlebook – HLS (UG1170) – Xilinx non-public document
HLS LEARNING RESOURCES/TRAINING

• High-Level Synthesis using Vivado HLS Course (a XUP course)
• System design using Vivado /Zynq (a XUP course)
• SDS0C course (a XUP course)