Intel® FPGA Technical Training

Designing with Intel FPGAs using HLS

Course Description

This course provides all practical know-how needed to start designing with Intel FPGAs using HLS methodology.

The course starts from an overview of HLS design methodology versus the traditional FPGA design flow. Then HLS procedure is studied along with HLS interfaces such as Avalon-MM and Avalon- Streaming. The course continuous with various code optimizations for loops, data types, memory optimizations and performance optimizations.

The course provides practical tools and design methods for software and hardware engineers in order to increase their productivity with Intel FPGAs by using C language along with Intel HLS compiler.

Course Duration

3 days

Goals

1. Understand the HLS concept and design flow
2. Be familiar with HLS interfaces
3. Optimize code for better results
4. Optimize code for high performance

Intended Users

Software engineers, Digital hardware engineers and FPGA team leaders, who would like to enhance their FPGA skills and achieve higher productivity in their FPGA design

Previous Knowledge

FPGA architecture is not mandatory but recommended

C language

Course Material

1. Intel official course book (including labs)
2. Intel HLS Compiler
3. ModelSim
4. Intel Quartus Prime Pro with Arria 10 family
5. 64-bit Linux Software Development Environment with g++

Table of Contents

Day #1

- **Introduction to High Level Synthesis**
  - High level design
  - Traditional FPGA design process
  - Why HLS?
  - HLS use model
  - Intel HLS compiler
  - HLS procedure
  - Intel HLS compiler usage and output
  - HLS procedure: x86 emulation
  - Simple example program: i++ and g++ flow
  - g++ compatability
  - Example Makefile
  - x86 debugging tools
  - Using printf()
  - Debugging using gdb
  - Debugging with Valgrind
  - HLS procedure: cosimulation
  - Example component/testbench source
  - Translation from C function API to HDL module
  - The cosimulation flow
  - Cosimulation verifying HLS IP
  - Simulation framework
  - Cosimulation design process
  - i++ options: g++ compatible options

- i++ options: FPGA related options
- i++ --clock option
- C/C++ functions to dataflow circuits
- Compilation example
- HTML report
- Loop unrolling
- Loop pipelining
- Loop analysis
- Area analysis
- Compiler viewer
- Memory viewer
- Verification statistics
- The default interfaces
- Explicit simulation behavior
- Enqueue function calls
- Streaming simulation behavior
- Viewing component waveforms
- HLS procedure: integration
- Quartus generated QoR metrics for IP
- Intel Quartus software integration
- HDL instantiation
- Platform Designer system integration tool
- Platform Designer HLS component example
- HLS backed components

❖ **Lab #1:** HLS Flow

- **HLS Interfaces**
  - Intel Avalon interfaces
  - Avalon-ST interfaces
  - Avalon-MM interfaces
  - IP interfaces
  - Component invocation interfaces
  - Default interfaces for scalars
  - Pointers: implicit memory-mapped interface

- Explicit MM master interface
- ihc::mm_master class parameters
- Explicit streaming interface example
- Blocking vs non-blocking reads and writes
- Explicit streaming interface customization
- Streaming packet example
- Stable arguments
- Slaves interfaces
- MM slave component
- MM slave register argument
- Slave component and register address map
- Slave memory argument
- Streaming HLS component in a system
- Memory-mapped HLS component in a system
- MM HLS component with streaming interfaces

❖ **Lab #2:** Resource Utilization Impact Based on Interfaces Selection

Day #2

- **HLS Loop Optimizations**
  - Understanding execution models and optimization reports
    - Common types of parallelism
    - Pipeline parallel execution for loops
    - Loop pipelining and dependencies
    - Parallelizing algorithms with iteration dependency
    - Loop pipelining
    - Component treated as loop
    - Loop pipelining benefits
    - Loop pipelining in Intel HLS compiler
    - Initiation Interval (II)
    - Loop pipeline analysis
    - Loop pipelining optimization report
    - Loop pipeline status
    - Loop pipeline single loop execution

- Single loop with complex dependencies
- Memory dependency
- Data dependency
- Speed-limiting constructs
- Resolving loop exit condition at iteration initiation
- Loop block contains non-linear execution
- Understanding throughput for nested loops
- Loop pipeline with nested loops
- Out-of-order loop execution
- Out-of-order loop iterations
- Serial region execution
- Serial regions

- Resolving common dependency issues
  - Minimize pipeline stalls
  - Removing loop-carried dependency
  - Relaxing loop-carried dependency
  - Relaxing loop-carried dependency generically
  - Transferring loop-carried dependency to local memory
- Good design practices
  - Good design practices for tasks
  - Pointer aliasing
  - Avoid pointer aliasing
  - Construct “well-formed” loops
  - Minimize loop-carried dependencies
  - Avoid complex loop exit conditions
  - Convert nested loops into single loop
  - Declare variables in the deepest scope possible
  - Move unnecessary operations out of the loop
  - Loop parallelization limitation
- Loop pragmas
  - Pragmas
  - List of loop-based pragmas
  - #pragma unroll <N>
  - Effects of loop unrolling
  - Unroll factor
  - Loop unroll restrictions and recommendations
  - Loop unrolling in the HTML report
  - Removing memory access loop-carried dependency
  - ivedep pragma

        - ivedep pragma with safelen
        - ivedep pragma advanced uses
        - loop_coalesce pragma
        - ii pragma
        - max_concurrency pragma
    - Resource sharing case study
        - High throughput FIR filter design
        - Resource-shared FIR filter
        - Using loops to share hardware resource

❖ **Lab #3:** Loop Optimizations for Moving Average Algorithm

- **HLS Data Types Optimization**
    - Arbitrary precision data types
        - Algorithmic C (AC) data types
        - Key advantages of AC types
        - Using ac_int in Intel HLS compiler
        - Arbitrary precision fixed point support
        - Operators for ac_int and ac_fixed
        - Conversion operators for ac_int and ac_fixed
        - Bit slicing for ac_int and ac_fixed
        - Integer promotion examples
        - Efficient bit shifting
        - Printing
        - Overflow debug flags
        - Current limitations
    - Floating point compiler optimizations
        - Need for tree-balancing
        - Tree-balancing with –fp-relaxed
        - Tree-balancing and resource savings
        - Rounding operations
        - Reduce rounding operations with --fpc
    - Other data type and math considerations
        - HLS math headers
        - Floating point common pitfalls
        - Integer promotion
        - Avoid expensive functions
        - Inexpensive functions
        - Use least-“expensive” data types
        - Floating point vs. fixed point representation
        - Data masking example
        - Arithmetic operation considerations

❖ **Lab #4:** Experience with Various Data Types

Day #3

- **HLS Local Memory Optimization**
  - On-chip memory systems
  - FPGA embedded memory blocks
  - Local memory implementation
  - On-chip memory architecture
  - Interconnect
  - Port sharing: arbitration
  - Efficient on-chip memory systems
  - Stall-free local memory accesses
  - Compiler choice of local memory system geometry
  - Double pumping
  - Double pumped memory example
  - Replication
  - Local memory replication example
  - Local memory duplication and component concurrency
  - Compiler code analysis
  - Static coalescing
  - Automatic banking
  - Automatic memory coalescing and banking
  - Automatic banking
  - Local memory limitations
  - Overcoming compiler limitations
  - Memory geometry unrelated to array shape
  - 2D possible geometries
  - Automatic memory splitting
  - Automatic port sharing optimization
  - Port sharing: mutually exclusive accesses example
  - Memory dependencies and loop pipelining
  - Minimize impact of dependencies for loop pipelining
  - Loop memory in the area report
  - Area report and local memory optimization
  - Local memory replication
  - Local memory area report: banking
  - HTML component viewer
  - Component memory viewer
  - Local memory access guidelines
  - Local memory attributes
  - Local memory attributes usage

- hls_numbanks (N) and hls_bankwidth (N)
- hls_numbanks (N) and hls_bankwidth (N) memory attribute 2D example
- Bank bits example: bankbits
- Local memory merging: depth-wise
- Local memory merging: width-wise
- Memory attributes to control replication
- Other attributes
- Static variables
- Static variable initializer example
- Control static variable initialization
- Simulate component reset
- Variables/arrays implemented as register
- Memory implemented in RAM
- Memory implemented as registers (constant access)
- Memory implemented as registers (size requirement)
- Memory describing shift registers
- Shift register implementation
- Area report: variables implemented as registers
- Area report: shift registers
- Area report: barrel shifters
- Area report: constants implemented as ROM

❖ **Lab #5:** Optimizing Loop Pipelining Performance by Relaxing Data Dependencies

❖ **Lab #6:** Local Memory Optimizations

- **The HLS Performance Optimizations**
  - Interface optimizations
    - Choosing the right interfaces
    - Vector add – pointers
    - Vector add – MM masters
    - Vector add – MM slaves
    - Vector add – streams
    - Vector add – stable
    - Vector add – scalars
  - Optimizing loops
    - Separate loops
    - Fuse loops
    - Nested loops
    - Coalescing nested loops

- Breaking data dependencies
- Breaking down dependencies with shift registers

- Getting the right memory architecture
  - Matrix multiply with component memory
  - Matrix multiply – unrolled
  - Matrix multiply with banking
- Arbitrary precision types
  - Vector add with ac_int
- Choose the right algorithm with full example
  - QR decomposition – householder algorithm
  - QRD – modified Gram Schmidt algorithm

❖ **Lab #7:** QR Decomposition Optimization