Vivado AXI Timer and Interrupts

Benjamin A. Blouin

*benjamin.blouin@temple.edu*

**Summary**

High-Level Synthesis (HLS) software like Vivado HLS transforms a C specification into a register transfer level (RTL) implementation that synthesizes into an FPGA, specifically for this lab, the Xilinx Zynq-7000. HLS allows for high level abstraction design implementations while creating high-performance hardware. In this lab the HLS software was explored through a matrix-multiplication function and the accompanying testbench synthesis, RTL Co-Simulation, and different iterations of optimizations. The optimizations include pipelining at different inner-loop levels, reshaping the matrices in different dimensions, and combinations of these. The HLS software can then be used to analyze feasibility of design implementation; not all optimizations can be implemented either due to algorithm or hardware deficiencies. Through synthesizable optimization directives, hardware implementations can be improved in many different categories, like throughput, latency, area, power, and many others. Some optimizations, while synthesizable, could not be implemented on the given hardware, usually due to requiring more DSP blocks than available.

**Introduction**

HLS designs can improve system performance while abstracting the physicality of the actual FPGA. HLS allows for function correctness validation through hardware emulation test output compared to host-system processor output. Optimization directives allow creation of specifications that can improve performance through differing hardware implementations. HLS allows for many different solution design explorations, improving the likelihood of optimal implementations. The HLS code, being based in C, makes the solutions portable. HLS can also create IP blocks for use in Vivado block designs.

Scheduling, in HLS, is the process that determines which clock cycle operations should occur. HLS takes into account clock frequency, timing on target hardware, and user specific optimizations. The code below is a simple example function:

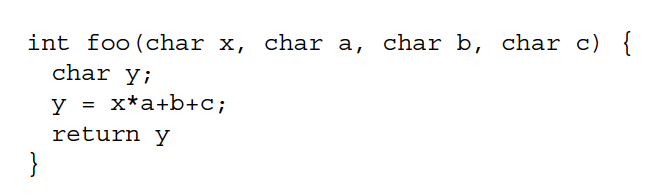


Figure Example Function

The multiplication and addition are scheduled to execute on the first clock cycle, and the next cycle performs the second addition, which means the output is available at the end of this cycle.

A screenshot of a cell phone

Description automatically generated

Figure Scheduling Example

The final hardware implementation uses the arguments of the top-level function as I/O ports. Each of the data types are 8-bits wide, so the appropriate 8-bit input ports are created. The function returns a 32-bit integer data type. The green square represents an internal register to store the intermediate result. The target binding is where the implementation of the C code is realized, which includes a DSP48 and an AddSub block. A Vivado HLS overview is shown below.

A screenshot of a cell phone

Description automatically generated

Figure Vivado HLS overview

Pipelining is a type of optimization that tries to use all resources as much as possible. The next possible transaction without pipelining would result in a logical block performing no operation; this is not optimal as the block could possibly be performing another transaction. Pipelining tries to mate the waiting block with an upcoming transaction that starts as soon as hardware resources become available. An example figure is shown below. In our labs we use loop pipelining, as there is only one function but multiple loop levels.

A screenshot of a cell phone

Description automatically generated

Figure Loop Pipelining

In the example function the latency remains the same, but the function can now process a new set of data every clock cycle. This can also be used on looped function. If a for loop from i-to-n, with some latency L is run without pipelining: For L=7 and N=1024, a new read (or write) happens every 7 clock cycles, meaning the total number of clock cycles to complete is 7\*1024=7186. With a one stage pipeline, a new read (or write) happens every 1 clock cycle, meaning the total number of clock cycles to complete is 7+(1024-1) \*1=1030. Interestingly, if a two-stage pipeline is used, 7+(1024-1) \*2=2053 results in lower performing schedule. The same optimizations can happen in dataflow also.

Another optimization example used in this lab is array reshaping. Arrays are by default synthesized into block-ram 18k-bit primitive elements. For example, an array of 1024 integer types will require 1024\*32-bit=32768 bits of block-RAM, which requires 32768/18000 = 1.8 18K block-RAM primitives. Each array is considered to be synthesized into its own block, and HLS makes no attempt to group smaller blocks into a single, larger block; nor does it attempt to partition large blocks into smaller ones by default. The array reshape directive attempts to reshape an array from one with many elements to one with greater word-width. This can be useful for improving block-RAM accesses without using more physical blocks. It attempts to create wider data ports over less blocks. The data\_pack, array\_partition, and array\_reshape directives are mutually exclusive; an example of array reshaping can be seen below in figure 6.

A screenshot of a cell phone

Description automatically generated

Figure Array Reshaping

This lab will be demonstrated in the discussion section, with explanations of the results.

**Discussion**

The lab manual dictates creating a project with a specific filename, importing source code, header file, and a testbench source file. These files were given along with a TCL script, and the project defines selecting the specific part identification based on the Zybo board chip: xc7z010clg400-1. The tutorial dictates, after creating the project, exiting HLS and running the given script file from the HLS command prompt. This command window includes TCL scripting abilities not normally available with the default Window’s path. Running the script produces either a pass status, fail, or a compilation error. The project can now be opened again, a C-Simulation, synthesis, and C/RTL Co-Simulation.

C-Simulation requires a testbench to validate that an algorithm is functionally correct. Simulating in C can be orders of magnitude faster than RTL simulation. All solutions during testing passed C-simulation.

The next step is C Synthesis, which is used to create an initial solution for an RTL implementation. A report is generated, as well as Verilog, VHDL, and systemC output RTL files. The analysis perspective provides tabular and graphical view of the design performance and resources for cross-referencing. The tutorial, on the first solution, requests running the C/RTL cosimulation. The performance profile provides details of block and sub-block latencies, initiation intervals (II), and pipelining information. The scheduler shows how the operations in a particular block are scheduled over clock cycles. The resources profile shows what resources are associated with which operations.

Each lab and subsequent solutions apply differing optimization directives. Solutions implement pipelining at different loop layers, array reshaping of different variables and with differing reshaping dimensions, and lastly pipelining the entire solution’s top function. Many of these choices end in solutions that can’t be implemented in our hardware, but the ones that can show increased performance on many orders of magnitude.

Errors found during synthesis will now be described.

1. WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('b\_load\_2', matrix\_mult\_10x10.cpp:16) on array 'b' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'b'.
   1. The memory block has a maximum of two data ports. The third and fourth memory reads start before the second has finished. This can be fixed either by adding a pipeline if the hardware can support the needed resources or partitioning the arrays. Neither option is viable, as the hardware can’t support another pipeline, and the array has already been reshaped, which means it can’t then also be partitioned. An example is that during some pipelining the specification would need 240 DSP48E blocks, but the Xilinx chip used only has 80, for a 300% utilization rate.
2. WARNING: [SCHED 204-68] The II Violation in module 'matrix\_mult': Unable to enforce a carried dependence constraint (II = 2, distance = 1, offset = 1). between 'store' operation (matrix\_mult\_10x10.cpp:16) of variable 'tmp\_8', matrix\_mult\_10x10.cpp:16 on array 'prod' and 'load' operation ('prod\_load', matrix\_mult\_10x10.cpp:16) on array 'prod'.
   1. This warning describes the pipeline not being filled due to latency from previous operation. A high-performance pipeline is able to start one new iteration at every clock cycle, with a write following a read. This warning shows that at least one clock cycle must be used for the pipeline read to finish before the next write continues. Loops with II = 1 represent cycle execution with each loop iteration performed in in a single clock cycle. Because the optimization technique used is automatic, manual optimization of the code could be achieved by static array declarations in an epilogue/prologue format. Another way to word this, is we need to move more operations to the prologue and epilogue and to have all the operations in the loop body executing simultaneously per loop iteration.

**Conclusions**

**Appendices**

Bazeghi, C., 2016. IP Flows, in: Designing with Xilinx® FPGAs. Springer International Publishing, pp. 23–33. <https://doi.org/10.1007/978-3-319-42438-5_3>

J. Liu, J. Wickerson and G. A. Constantinides, "Loop Splitting for Efficient Pipelining in High-Level Synthesis," 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, 2016, pp. 72-79. doi: 10.1109/FCCM.2016.27. Accessed 2/21/2019 <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7544750&isnumber=7544722>

Koch, D., Hannig, F., Ziener, D. (Eds.), 2016. FPGAs for Software Programmers. Springer International Publishing. <https://doi.org/10.1007/978-3-319-26408-0>

**Code**