Lab Experiment:

VLSI System Design Using High Level Synthesis Tools

By: Dr. Tassadaq Hussain
Instructor: Mr. Waseem Khan
Riphah International University

Date: .....

0.1 Objective

This lab is for system designer and architects to understand how High Level Synthesis (HLS) evolved from existing design methodologies and how it can help to improve current design flows.

0.1.1 Introduction to High Level Synthesis

FPGAs have traditionally been programmed with languages and methodologies that stem from the Electronic Design Automation (EDA) community. While in HPC, high level languages such as C and Fortran are used. This design gap between EDA methodologies and HPC programmers has slowed the adoption of FPGAs by the HPC community. Translation of HPC algorithms into hardware is complex and time consuming. Large applications result in complex systems with a high probability of containing design errors, and it is challenging to reuse the hardware resources. To overcome these difficulties, FPGA designers use C-to-hardware High Level Synthesis (HLS) that performs design modeling and validation at a higher level of abstraction. However, this approach has some limitations. Every application kernel requires its own accelerator. A scalar processor core is needed to manage *Local Memory* data and *Main Memory* transfers for the accelerators. Every change in the code of the application introduces a new place-and-route iteration to generate the hardware accelerator. Hence, even simple changes can lead to dramatic shifts in area usage and clock frequency. All of these factors make design closure with custom data-path accelerators very difficult.

The HLS design flow consists of a sequence of steps (see Figure 1), with each step transforming the abstraction level of the algorithm into a lower level description. The HLS *Parallelism Extraction* step extracts the Control and Data Flow Graph (CDFG) from the application to be synthesized. The CDFGs describe the computational nodes and edges between the nodes. HLS provides different methods to explore the CDFG of the input algorithm and generates the data-path structure. The generated data-path structure may contains the user-defined number of computing resources for each computing node type and the number of storage resources (registers). The *Allocation and Scheduling* step maps the CDFG algorithm onto the computing data-path and produces a Finite State Control Machine. The *Refining* step uses the appropriate board support pack-

age and performs synthesis for the *communication network*. Finally, the *HDL-RTL Generation* step produces the VHDL files to be supplied to the proprietary synthesis tools for the targeted FPGA.

Over the past few years, HLS tools have been developed that add the necessary technologies to become truly production-worthy. Initially limited to data path designs, HLS tools have now started to address complete systems, including control-logic and complex on-chip, off-chip interconnection. Xilinx Vivado HLS [1] (built on AutoESL tool technology [2]) accelerates design implementation. It takes C, C++, or SystemC as its input and produces device-specific RTL after exploring a multitude of micro-architectures based on design requirements. Impulse Accelerated Technologies develops the ImpulseC programming language [3]. The ImpulseC tools comprise a software-to-hardware compiler that translates individual Impulse C processes to hardware and generates the necessary process-to-process interface logic. Handel-C, developed by Celoxica [4], is based on the syntax of conventional C language. Programs written in Handel-C are sequential. To exploit benefits of parallelism from the target hardware, Handel-C provides parallel constructs such as pipelined communication and parallel sections. Catapult C, designed by Mentor Graphics [5], is a subset of C++. The code that is compiled through Catapult C may be general purpose and result in much different hardware implementations

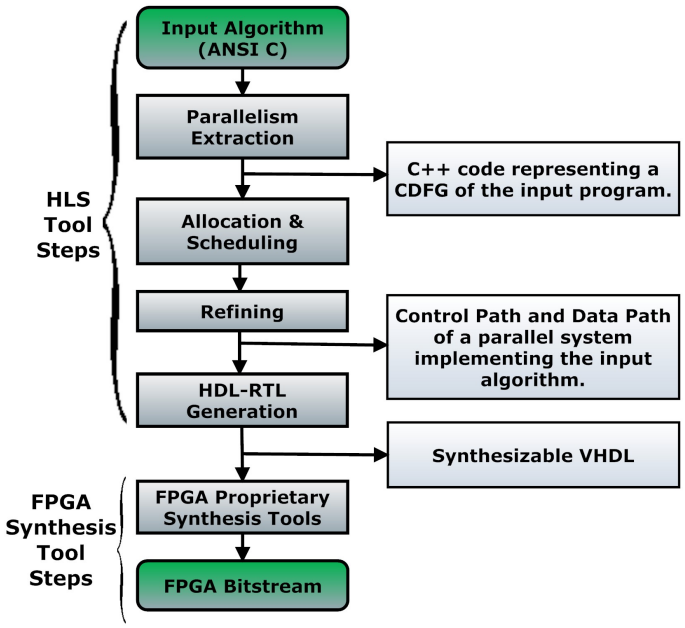

Figure 1: Conventional High Level Synthesis Tool Flow

with different timing and resource constraints. The Catapult C environment takes constraints and platform details in order to generate a set of optimal implementations. Ylichron (now PLDA Italia) developed a source-to-source C to VHDL compiler toolchain targeting system designers called HCE (Hardware Compiling Environment). The HCE toolchain [6] takes ANSI-C language as input, which describes the hardware architecture with some limitations and extensions. ROCCC 2.0 [7] is a free and open source tool that focuses on FPGA-based code acceleration from a subset of the C language. ROCCC tries to exploit parallelism within the constraints of the target device, optimize clock cycle time by pipelining, and minimize area. ROCCC is one of the few HLS tools that does memory access optimization. LegUp [8] is an open srouce tool which allows different software techniques to be used for hardware design. It accepts a standard C program as input and automatically compiles the program to a hybrid architecture containing an FPGA-based MIPS soft processor and custom hardware accelerators that communicate through a standard bus interfaces.

0.2 Reverse Time Migration

RTM is the most advance seismic data processing method used to recover inside subsurface images of the Earth. The complete system is shown in Figure 2. It is based on wave equation through a volume representing the earth subsurface. RTM's main strength is the ability of showing the bottom of salt bodies at several kilometers of the earth's interior. Figure 3(a) presents the pseudo-code of the RTM algorithm with partial differential equation (PDE) solver. RTM works by running the WFC, forward in time for the source and backwards in time for the receiver. In RTM, both forward and backward propagation parts of the method utilizes the

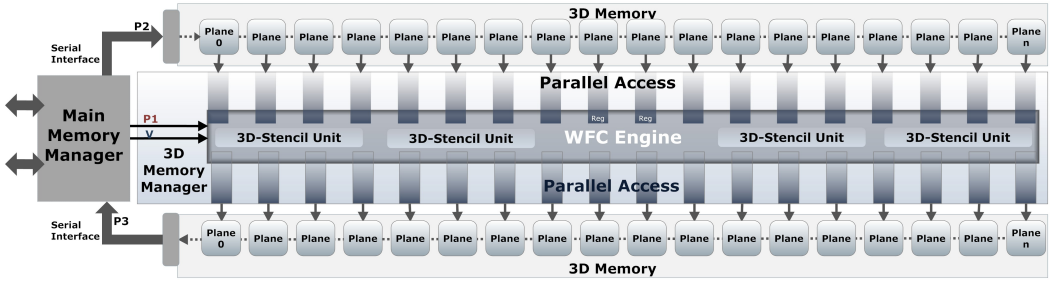

Figure 2: Wave Field Computation Architecture

Forward propagation	Backward propagation
input: velocity model, shots	input: velocity model, receivers' traces, forward wavefield
output: forward wavefield	output: image

```
1: for all time steps do
2:     for all main grid do
3:         compute wavefield
4:     end for
5:     for all source location do
6:         add source wavelet
7:     end for
8:     for all ABC area do
9:         apply ABC
10:    end for
11:    for all main grid do
12:        store wavefield
13:    end for
14: end for
```

```
1: for all time steps do
2:     for all main grid do
3:         compute wavefield
4:     end for
5:     for all receivers location do
6:         add receivers data
7:     end for
8:     for all ABC area do
9:         apply ABC
10:    end for
11:    for all main grid do
12:        load forward wavefield,
           correlate wavefields
13:    end for
14: end for
```

(a)

$$for\ (y = stencil;\ y < NY - stencil;\ y++;)$$
$$for\ (x = stencil;\ x < NX - stencil;\ x++;)$$
$$for\ (z = stencil;\ z < NZ - stencil;\ z++;)$$

$$P_3(x,y,z) = \sum_{l}^{s} w_l^1\ [P_2(x-l,y,z) + P_2(x+l,y,z)]$$
$$+ \sum_{l}^{s} w_l^2\ [P_2(x,y-l,z) + P_2(x,y+l,z)]$$
$$+ \sum_{l}^{s} w_l^3\ [P_2(x,y,z-l) + |P_2(x,y,z+l)]$$
$$+ c^{\circ} P_2(x,y,z)$$
$$+(V(x,y,z) \times dt)^2 + (2 \times P_2(x,y,z)) - P_1(x,y,z)$$

(b)

Figure 3: (a) Reverse Time Migration Algorithm (b) Wave Field Computation's Mathematical Representation

same PDE solver for computing the Wave Field Computation (see Figure 3(b)). It is analyzed by [9] that 90% and 70% execution time is utilized while compute the Wave Field computation (WFC) part. This shows the significance in accelerating this part of code. Our RTM implementation is based on an explicit 3D high order Finite Difference numerical scheme for the wave equation.

-

Mathematical representation of Wave Filed Computation part is mentioned in Figure 3(b). We can see the actual code of the compute intense kernel for WFC called Kernel_rivera in Figure 5. WFC uses four volumes of data `v,p1,p2` (input volume) and `p3` (output volume). We separated WFC into two parts.

- Integration over time
- Stencil computation

0.2.1 Integration over time

Integration over time needs one operand from each `v` and `p1` volumes, output of stencil unit and two constants. This unit requires few computations and data access of `p1` and `v` volumes is streaming.

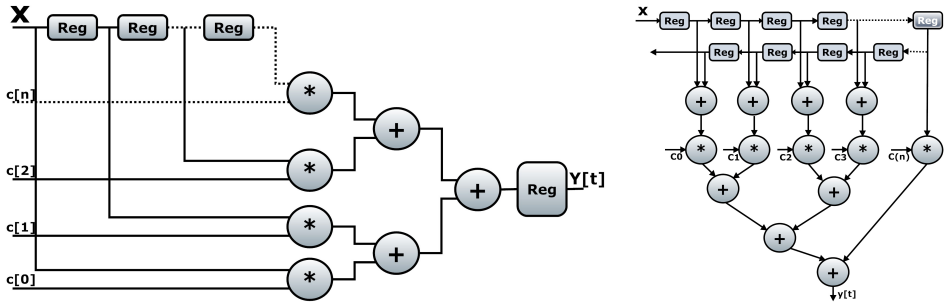

Figure 4: FIR[n]=$\sum_{k=0}^{N} h_k x\,[n-k]$: (a) Fully Parallel FIR Filter (b) Odd Symmetric FIR Filter

0.2.2 Stencil Computation

The stencil computation unit consists of 3 Symmetric finite impulse response (FIR) filters. For every stencil output point 25 operands are required from p2 volume. In addition to these data variables there are 14 constant operands are used. All these operands are of single precision floating point i.e. 4 bytes/operand is required. Due to sparse access pattern of Stencil compute unit it needs special consideration. In order to know the approaching details it is important to revisit the 3-Stencil explanations.

3D-Stencil

Stencils are used in numerous scientific applications like, computational fluid dynamics, geometric modeling, electromagnetic, diffusion and image processing. These applications are often implemented using iterative finite-difference techniques that sweep over a two dimensional (2D) or 3D grid, while performing computations called stencil on the nearest neighbors of current point in the grid. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space. Stencils can be either non-symmetric or symmetric with even or odd symmetry.

Stencil operation can be understood by taking a look at Figure 4 which mathematically explains one dimensional (1D) stencils. A fully parallel FIR n-tap filter uses multipliers equal to the *number of coefficients (n)* or *n-taps* and *(n-1)* number of adders. Coefficients are often symmetrical around the center tap or two innermost taps of the filter. This symmetry allows the number of multiplication to reduce to half. Even Symmetry eliminates the number of multiplication to half by pre-adding the top val-

```
#define MX 64
#define MY 64
#define MZ 64
for ( k = Stencil ; k < MY - Stencil ; k++ )
for ( j = Stencil ; j < MZ - Stencil ; j++ )
for ( i = Stencil ; i < MX - Stencil ; i++ )
{
iter = k*(MX*MZ) + (j*MX) + i;
tmp =
Y1*(P2_linear[i+j*iter_j+(k-1)*iter_k] + P2_linear[i+j*iter_j+(k+1)*iter_k]) +
Y2*(P2_linear[i+j*iter_j+(k-2)*iter_k] + P2_linear[i+j*iter_j+(k+2)*iter_k]) +
Y3*(P2_linear[i+j*iter_j+(k-3)*iter_k] + P2_linear[i+j*iter_j+(k+3)*iter_k]) +
Y4*(P2_linear[i+j*iter_j+(k-4)*iter_k] + P2_linear[i+j*iter_j+(k+4)*iter_k]) +
c00 * P2_linear[iter] +
X4*(P2_linear[i+(j-4)*iter_j+k*iter_k] + P2_linear[i+(j+4)*iter_j+k*iter_k]) +
X3*(P2_linear[i+(j-3)*iter_j+k*iter_k] + P2_linear[i+(j+3)*iter_j+k*iter_k]) +
X2*(P2_linear[i+(j-2)*iter_j+k*iter_k] + P2_linear[i+(j+2)*iter_j+k*iter_k]) +
X1*(P2_linear[i+(j-1)*iter_j+k*iter_k] + P2_linear[i+(j+1)*iter_j+k*iter_k]) +
Z4*(P2_linear[(i-4)+j*iter_j+k*iter_k] + P2_linear[(i+4)+j*iter_j+k*iter_k]) +
Z3*(P2_linear[(i-3)+j*iter_j+k*iter_k] + P2_linear[(i+3)+j*iter_j+k*iter_k]) +
Z2*(P2_linear[(i-2)+j*iter_j+k*iter_k] + P2_linear[(i+2)+j*iter_j+k*iter_k]) +
Z1*(P2_linear[(i-1)+j*iter_j+k*iter_k] + P2_linear[(i+1)+j*iter_j+k*iter_k]);
P3_linear[iter] = tmp ;
}
```

Figure 5: Non-Synthesizable WFC Code used for Functional Verification

ues together. Odd symmetry is slightly different. In odd symmetry center tap is handled separately because there is no pre-adder. Other than odd symmetry value rest of the architecture is similar to even symmetry. Odd symmetry 1D-FIR is shown in Figure 4(b).

In 3D stencil, each computed point needs to access data from the three axis of a volume. Therefore, the 3D stencil computation increases the complexity not only by increasing number of computation 3 times but also due to sparse data access pattern arising from a volume linearly laid out in memory.

0.3 Functional Verification

High-level functional verification provides substantial decrease in the test generation time, test application time. By utilizing debug/varify facility it increases the fault coverage and decrease area/delay overheads. In this section we test and verify behavior of *WFC system* prior to synthesis. In order to determine intent of the *WFC system* HLS provides facility to perform functional verification and fault simulation at higher level. To verify and compare intent of *WFC system* we used non-synthesizeable C code of kernel-rivera kernel (see Figure 5).

0.4 Experimental Data

This section deals with exploiting parallelism from the proposed architecture. In this experiment, we are using ROCCC [7] HLS tools to generate hardware accelerators. At first we will design 8-tap FIR filter hardware accelerator (Shown in Figure 4). The synthesizable source code for FIR is mentioned in Figure 6.

```
//  A 8-tap FIR filter.
//  By Dr. Tassadaq Hussain

typedef struct
{ // Inputs
  int A0_in ;
  int A1_in ;
  int A2_in ;
  int A3_in ;
  int A4_in ;
  int A5_in ;
  int A6_in ;
  int A7_in ;
  // Outputs
  int result_out ;
} FIR_t ;

FIR_t FIR(FIR_t FIR_8)
{ // Should be propagated
  const int filer[8] = { 3, 5, 7, 9, 11, 12, 21, 33};
  FIR_8.result_out = FIR_8.A0_in * filer[0] +
   FIR_8.A1_in * filer[1] +
   FIR_8.A2_in * filer[2] +
   FIR_8.A3_in * filer[3] +
   FIR_8.A4_in * filer[4] +
   FIR_8.A5_in * filer[5]+
   FIR_8.A6_in * filer[6]+
   FIR_8.A7_in * filer[7] ;
  return FIR_8 ;
}
```

Figure 6: Synthesizable 8-tap FIR Filter Hardware Accelerator

Table 1: Achieved GFLOPS by HLS Hardware Accelerator

System Name	Floating Point Operation per Output	Number Of Clocks	Achieved GFLOPs @ ..... MHz Clock
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....

```
#include "roccc-library.h"
void FIR_SYSTEM_RIU(int* A, int* B)
{
   int i;
   int my_tmp;

   for(i = 0; i < 100; ++i)
   {
      // Example code to pass stream A into stream B
      FIR(A[i],A[i+1],A[i+2],A[i+3],A[i+4],A[i+5],A[i+6],A[i+7],my_tmp);
      B[i]=my_tmp;
   }
}
```

Figure 7: Synthesizable FIR System

$$freq * \frac{\#ofClocks}{Single\ Output\ Point} * \frac{Floating\ Point\ Operations}{Single\ Output\ Point}$$

Figure 8: HLS Hardware Accelerator GFLOPs

Once the necessary computation and input outputs are finalized, focus is transferred to resource allocator and scheduler.

Design different FIR systems and count their floating point operates, number of clocks required to generate a single output and calculate GFLOPS using 8.

1. We will use a single instance of FIR and FIR system shown in Figure 7 and calculate GFLOPS of the system.
2. Add two instances of FIR and calculate GFLOPS.
3. Add different hardware accelerators such as FFT, FIR, etc. and calculate GFLOPS of system.

0.5 Discussion and Analysis

1. How many Hardware Accelerators block (functions) we can instantiate in the main system (i.e. BRAM of FPGA, etc.).
   .....
   .....
   .....

2. What affects the theocratical and practical GFLOPs of system? (i.e. Memory Read/Write, etc.)
   .....

.....
.....

3. What are the differences between handwritten HDL and HLS generated HDL (i.e. Design Time, etc.).

.....
.....
.....

Questions and Suggestions:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Bibliography

[1] Feist Tom, "Vivado design suite," *Xilinx, White Paper Version*, vol. 1, 2012.

[2] "AutoESL High-Level Synthesis Tool." [Online]. Available: {http://www.xilinx.com/tools/autoesl.htm}

[3] "Impulse CoDeveloper Overview," 3,April 2011. [Online]. Available: {http://www.impulseaccelerated.com/}

[4] Matthew Bowen, "Handel-C Language Reference Manual."

[5] Graphics, Mentor, "Catapult C synthesis overview," 2012.

[6] Alessandro Marongiu and Paolo Palazzari, "The HARWEST Compiling Environment: Accessing the FPGA World through ANSI-C Programs." *CUG 2008 Proceedings*, 2008.

[7] "Riverside Optimizing Compiler for Configurable Computing (ROCCC 2.0)," http://roccc.cs.ucr.edu/, 23,March 2015.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[9] Francisco Ortigosa, Hongbo Zhou, Santiago Fernandez, Repsol-YPF, Mauricio Hanzich, Mauricio Araya-Polo, Flix Rubio, Ral de la Cruz and Jos Mara Cela, "BENCHMARKING 3D RTM ON HPC PLATFORMS," , . [Online]. Available: {http://www.bsc.es/projects/kaleidoskope_tmp/pdf/106%20BENCHMARKING%203D%20RTM%20ON%20HPC%20PLATFORMS_2008.pdf}