Using ESL Tools for FPGA Design

Aws Ismail
FPGA Research Group
University of Windsor

Acknowledgement

- All slide material Related to DK and Handel-C are courtesy of Celoxica Inc.
- Please check www.celoxica.com for more info

Outline

- ESL Overview
- Using Celoxica Handel-C & DK
  - Handel-C Building Blocks
  - DK Basics
  - Talking to the outside world
  - Aggregate Types, Advanced Types
- Handel-C & FPGAs
  - Mapping Handel-C to FPGAs
- Demos

Nomenclature

- DK
  - DK stands for Design Kit
  - This is the tool, including the GUI, the simulator, and the Hardware Compiler
- Handel-C
  - Handel-C is the programming language
  - For Hardware design
- PDK
  - PDK stands for Platform Developer's Kit
  - This is a package of libraries, tools, source code to help users design using Handel-C and target supported hardware platforms
- FPGA
  - Field Programmable Gate Array

ESL Overview

What is ESL? And why is it important ?

ESL Overview

- ESL = “Electronic System-Level”

- A new generation of EDA tools are emerging in the world of logic design
- Designers can now take their algorithms straight into hardware without having to learn traditional hardware design techniques

- What is ESL Design?
  - A collective classification of new high level tools and associated design methodologies
  - General characterization is that it refers to tools that approach the problem at a higher level of abstraction rather than the mainstream register transfer level (RTL)

ESL Overview

- ESL design languages
  - Also referred to as High-level languages (HLL)
  - Most are close in syntax and semantics to ANSI-C rather than VHDL or Verilog

- ESL for FPGAs
  - Collection of HLLs, tools, and methodologies that are specifically optimized for an FPGA platform
  - Also known as, Platform Level Design
  - Currently considered a natural evolution for FPGA design tools
  - Programmable hardware is now easily accessed by a wider and more software-centric user/designer base

Why ESL?

- First scenario
  - Most complex algorithms are captured in high-level languages like “C” and must be converted to a corresponding RTL description
  - Manually performing C-to-RTL conversion is a tedious and error prone task
  - ESL gives a direct C-to-Hardware path
- Second scenario
  - Seamless Hardware/Software implementation
  - ESL tools are used in the design of both the hardware side and the associated software side of the system
  - ESL value and appeal, therefore, extends to both HW designers and Software Programmers

Handel-C Building Block

Handel-C timing, parallel and sequential code, loops and conditions

Handel-C Concepts

- Handel-C is a C-like language
  - ANSI-C syntax and semantics
  - Extensions and restrictions for the purpose of hardware design
- Designed for synchronous hardware design
  - Optimized for FPGAs (FPGA-Centric)
  - Everything that simulates will compile to hardware
- Extensions to C allow you to produce efficient hardware
  - ***par*** to introduce parallelism
  - Arbitrary word widths
  - Synchronization
  - Hardware interfaces

A simple Handel-C Program

```
set clock = external;                       //Set clock source

void main()                                 //entry point of the design
{
 static unsigned 32 a = 238888872
 static unsigned 32 b = 12910669;           //Input variables
 unsigned 32 Result;                        //Variable for Result
 interface bus_out() OutputResult(Result);  //Output the result to pins


 while (a != b)
 {
   if(a > b)
     a = a - b;
   else
     b = b - a;
 }

 Result = a;                                //Set the output variable
}
```

Handel-C Timing

- Handel-C is implicitly sequential
- Each statement takes one clock cycle
  - ***a = b;   //clock cycle 1***
  - ***a = a + 1; //clock cycle 2***
- Delay statement to do nothing for a clock cycle
  - ***a = b;   //clock cycle 1***
  - ***delay;   //clock cycle 2***
  - ***a = a + 1; //clock cycle 3***
- Multi-expressions in a single statement is not allowd
  - ***a = b++; //not allowed***
    - Breaks the timing model of each assignment taking a clock cycle
    - Anything with side-effects can be written without them

Variables

- Basic type is the integer
  - No floating point type in Handel-C
- Integers can be either signed or unsigned
  - Signed numbers are stored in 2's complement format
- Can be any width
  - ***signed int 8 a;*** ***// signed 8 bit variable "a"***
  - ***int 8 a;*** ***// can omit the signed keyword***
  - ***unsigned int 8 a;*** ***// unsigned 8 bit variable***
  - ***unsigned 8 a;*** ***// can omit the int keyword***
- Pre-determined widths available
  - ***chat (8), short (16), long (32), int32 (32), int 64 (64)***
  - Can specify ***unsigned***
- Behave like registers (often referred to as registers also)
  - Take new value on the clock cycle following an assignment

par Statement

- Expresses what should happen in parallel
- Everything in the subsequent block happens in parallel
- ***Seq*** statement says that a section will be sequential
  - *This is just for clarity*
  - *You can leave seq out*

```
// 2 clock cycles
{
    a = 1;
    b= 2;
}

// 1 clock cycle
par
{
    a = 1;
    b = 2;
}
```

Parallel and Sequential Examples

```
unsigned 4 a,b;

seq
{
          a = 1;              //clock cycle 1: a = 1

 par
 {
          a = a + 1;          //clock cycle 2: a = 2
          b = 5;              //clock cycle 2: b = 5
 }

 par
 {
          b = b + 1;          //clock cycle 3: a = 5
          a = b;              //clock cycle 3: b = 6
 }

}
```

par Completion

- **par** block completes when longest path completes
- The **par** statement can be used to express both coarse and fine grained parallelism
  - Individual statements in parallel
  - Functions in parallel

```
a --;  //cycle 1

par
{
    b++;      //cycle 2
    seq
    {
        a++;   //cycle 2
        a = b; //cycle 3
    }
}

b--;   //cycle 4
```

par Examples

- Can read from variable in parallel
  - Simply wires the two variables together

```
par
{
    b = a;
    c = a;
}
```

- Can't write to same variable in parallel
  - Undefined value will be written

```
par
{
    a = b; //won't work
    a = c; //won't work
}
```

- No need to use temporary variables to swap values

```
unsigned 4 a,b;
par
{
    a = b;
    b = a;
}
```

Conditional Branching

- Control the flow of your program
- Conditions take ZERO clock cycles to be evaluated
- **if**
  - Exactly like C
  - Can use a **delay** statement in the **else** clause to balance execution time

```
if (a == 0)
   a++;
else
   delay;
```

for Loops

- Syntax

  for (initialization; test; increment) body
- All expressions optional
- **initialization** takes at least a clock cycle
- **test** is evaluated before each iteration of the body
- **increment** expression takes a clock cycle at the end of each execution of **body**
  - This adds an extra clock cycle of delay to the body of any loop
- **for** is not recommended for general use
  - Use **while** or **do…while**
  - **increment** can always be done in parallel with the body

while Loops

- **while**
  - **while (condition) body**
  - **condition** evaluated before each execution of the **body**
  - **while** statement terminates if **condition** evaluates to zero
- **do…while**
  - **do body while (condition);**
  - Always executed at least once
- **while(1)**
  - Runs forever
  - Remember that any statement following a **while(1)** will never happen
- Advantages over **for**
  - Can place initialization and increments of loop counters in parallel with other code, either inside the body or elsewhere

Loops – Combinational Cycles

- Every branch within a loop must take at least 1 clock cycle

```
while(1)
{
    if(x)
      delay;
}
```

- if **x** is not true, the loop would execute in zero clock cycles, creating an invalid circuit
- Often the DK IDE will issue an error if this happen, but sometimes it will warn the user
- When a warning is issue, the compiler breaks the combinational cycle by adding an extra delay register
- Best practice is to always balance the timing and avoid comb. cycles

State Machines in Handel-C

- The state machine is implicit
- Constructed from conditional branches, loops, sequential blocks and parallel blocks
- Handel-C produces a “one-hot encoding” state machine
- You can produce very complex state machines with ease
- The final result is easy for others to understand

- It is very easy for HDL designers to get carried away by explicitly writing their state machines, which is still possible in Handel-C

Signals

signal unsigned 8 a;

- A signal behaves like a wire
  - Take the value assigned for the current clock cycle only
- Default value is undefined
  - May get different behavior in simulation compared to hardware
- Has to be declared as Static when user decides to give it a default value
  - Will take this value if not assigned to in a clock cycle
  - A static signal without an explicit initialization will default to 0
- Assignment evaluated before read in a clock cycle
- You can use the signal keyword on an array declaration to create an array of signals
- Usually used to break complex expressions into simple, more readable chunks

Signals - Example

```
signal unsigned 8 A;
static signal unsigned 8 B = 5;
static unsigned 8 X = 1, Y = 2;

par
{
            A = X * 2;
            X = A;
            Y = A + 1;

}           //X = 2, Y = 3

X = B;      //use default value of B, X = 5

Y = A;      //Won't work. A is not static and not initialized
            //Y now has an undefined value
```

Synchronization and Communication

- Many programs have independent processes running in parallel
- They often need to communicate and synchronize with each other
- They often need to share resources such as functions and RAMs
- This type of code can be complicated and convoluted to write

- Handel-C has two features to make these problems easier to solve
  - Channels to communicate between processes
  - Semaphores to control access to critical sections of the code

Channels

- Blocking communication between two sections of code
  - Both sides block until the other is ready

  chan unsigned 8 ChannelA; //ChannelA is an 8 bit channel

  unsigned 8 VarX;

  channelA ! 3 //send 3 down ChannelA

  ChannelA ? VarX //read from ChannelA into VarX
- Channel communication takes at least one clock cycle
- Can only read from or write once to a channel in parallel
- Can use zero width for synchronization only
- Channels between clock domains are possible
- ***chanin*** and ***chanout*** for debug
  - Can attach to text files

Channels - Example

```
chan unsigned 4 myChan;
//channel between the two processes

static unsigned 4 Val = 1;

while(1)
{
	Val = Val + 1;
	MyChan ! Val; //send

	delay;
	//Delay always happen on
	//the same clock cycle
}
```

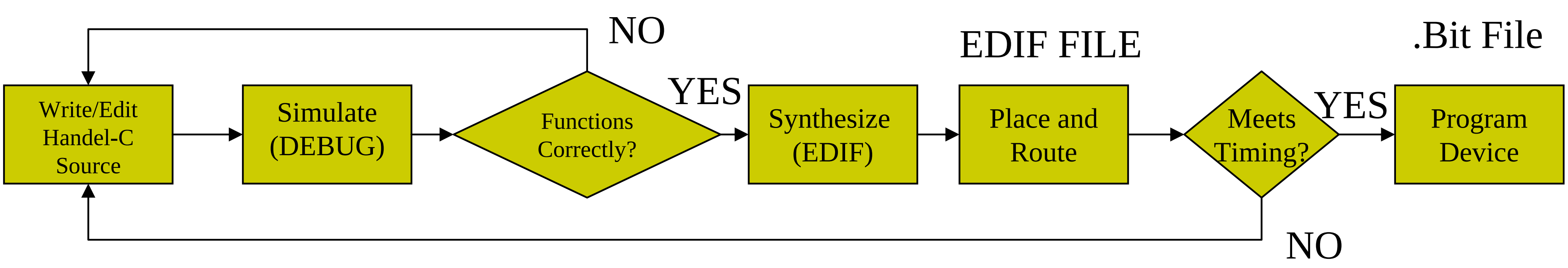
```
static unsigned 4 Count = 1;

while(1)
{	//wait for 0 or more cycles

	while(Count != 1)
	{
	 Count--;
	}

	MyChan ? Count //receive

	delay;
	//Delay always happen on
	//the same clock cycle

}
```

DK Basics

Introduction to DK Flow, Project Types, DK options, Compiling to HW

DK Basics

- DK is an Integrated Development Environment (IDE)
  - Familiar look and feel
  - Compile, Simulate and Debug
  - Compile to Hardware (Synthesize)
  - Run other tools, e.g. Place and Route tools, software compilers
- Simplified DK design flow:



Project Types

- Chip
  - Generic chip – does not use device specific resources
- Specific Chip
  - Targeted toward a particular device
  - Use device-specific resources
- Core
  - Discrete piece of code e.g. filter, FIFO
  - Targeted toward a particular device architecture
- Library
  - Defines functions that can be used in other projects
  - Like a library file in C (i.e. .lib)
  - Can be generic, which allows user to target simulation, EDIF, VHDL and Verilog
  - Can be limited to a particular output e.g. simulation
- Other
  - Board can contain multiple chip projects
  - System can contain multiple board projects

Compiling to Hardware

- Two routes to HW
  - EDIF (a netlist description format), where DK does all the synthesis
  - RTL VHDL/Verilog where third-party tools does synthesis
- EDIF is recommended route for FPGAs
  - Offers tightest integration
  - Fast and easy to use
  - Produces good results
  - Easier to debug for timing issues
  - VHDL/Verilog for use with favoured synthesis tool, simulation tool, combining into a larger HDL project
- Celoxica’s tools are geared toward FPGAs rather than ASIC
  - For ASIC the only path is to generate VHDL/Verilog from Handel-C and then use third-party tool (i.e. Synopsys DC)

Place and Route

- What is P&R ?
  - Automatic process to place the logic components and determine a path between them through the dedicated routing resources
- Timing constraints are used to get efficient P&R results
- Xilinx
  - DK produces an EDIF file (.edf) and a timing constraints file (.ncf)
  - Xilinx ISE is used to place and route
  - *edifmake.bat* (supplied with PDK), project navigator in ISE
- Altera
  - DK produces an EDIF file (.edf), a TCL script (.tcl) and memory inisialization file (.mif)
  - Quartus II software is used to place and route
  - *softmake.bat* (supplied with PDK)

Creating Hardware Output

- All Handel-C programs must have
  - A main function
    - This is the start point of the design
    - No integer return value in Handel-C
  - A clock specification
    - For example:

      set clock = external “A12” with (rate = 50);
    - “A12” is the name of a clock pin on the device
    - 50 is the clock rate in MHz of the input clock
      - This passes on timing constraints to the place and route tool
  - Output interfaces in order to synthesize
    - For example:

      interface bus_out( ) OutputBus (OutputPort);
    - Without it, the design will be optimized away to nothing because your design would sit inside the FPGA and never affect the outside world

Talking to the Outside World

Input and Output interfaces, PDK introduction

Interfaces

- 3 basic types
  - **Bus** for interfacing to external devices via pins
  - **Port** for when Handel-C is not the top-level module in a design
  - **User-defined** for talking to external code (e.g. VHDL, Verilog, EDIF) with Handel-C as the top-level
- Interfaces declarations appear with variable declarations before any statements
- All interfaces have the same basic syntax

  interface InterfaceType (InputsToDK) InstanceName (OutputFromDK);
  - Each interface type has restrictions to the inputs and outputs
- Only signed and unsigned types maybe passed over interfaces

Buses - Examples

```
#define PinList {"A1", "A2", "A3", "A4"}
unsigned 4 a;
interface bus_out ( ) MyOutBus (unsigned 4  Out = a) with {data = PinList};
//-----------------------------------------------------------------------------------------
interface bus_clock_in (unsigned  8 In) MyInBus( );
unsigned 8 B;
B = MyInBus.In;
//-----------------------------------------------------------------------------------------
interface bus_ts (InputPort) Name(OutputPort, ConditionPort);
```

Platform Developer's Kit

- PDK has been conceived to accelerate the design process
  - Lets the designer concentrate on implementing algorithms rather than spend time dealing with low-level complexities
- PDK offers three layers of functionality
  - Platform Abstraction Layer (PAL): API for portable projects
  - Platform Support Libraries (PSL): board specific support
  - Data Stream Manager (DSM): integration between processors and FPGAs
- PDK also includes
  - Support for co-simulating Handel-C with VHDL, Verilog, SystemC, and Matlab designs
  - Support for reconfigurable platforms (other than Xilinx and Altera)

Aggregate Types and Advanced Types

RAMs, ROMs, Multiport RAMs

RAMs and ROMs

- Designers often need efficient random access storage for hardware design
- Often only one value per a single clock cycle is needed
- FPGAs provide dedicated on-chip RAM resources
  - Block RAM or Distributed RAM on Xilinx FPGAs
  - Tri-Matrix RAM or LUT ROM on latest Altera FPGAs
- Handel-C therefore provides RAMs and ROMs data types
- RAMs and ROMs are not inferred from Arrays

RAMs and ROMs - Details

ram unsigned 8 a[n];

- Can only read from or write to one location in a clock cycle
  - Shared address bus for read and write
- ROM has has no write data bus
- Built out of dedicated RAM resources
  - Use distributed RAM by default (for Xilinx)
  - Use ***with { block = "X" }*** to use specific RAM resource

  e.g. ***ram unsigned 8 a[8] with { block = "M512" };*** *// use an M512 block in an Altera Stratix*
  - Use ***with {block = 1}*** to allow place and route tools to choose an appropriate resource

Arrays versus RAMs

- Use arrays for
  - Parallel access to all elements, for example in pipelined FIR
- Use RAMs for
  - Random access
  - Large data storage
- Use ROM for
  - Decoding or encoding signals, for example Seven Segment Display
  - Lookup table of coefficients
  - Lookup table for results like sine/cosine/tangent, using input as address
- Can’t read, modify, and write to a RAM in the same clock cycle

MyRAM[0] += 2; *//Won’t work*

Multi-port RAM

```
mpram
{
    wom unsigned 4 Write[32]; //write only port
    rom unsigned 8 Read[16]; //read only port
}
```

- Devices have entirely independent read/write ports
  - Can use both port during the same clock cycle
  - Virtex-II has two read/write ports on BRAM, one read/write port and one read port on distributed dual-port RAM
  - Stratix has two read/write ports on M4K and M-RAM, one read and one write port on M512 blocks
- Example: line buffers in image processing
  - One pixel in and one out every clock cycle

Handel-C & FPGAs

Technology mapping

Technology Mapping

- The gate-level netlist output of DK contains basic logic gates: OR,XOR, NOT, and AND
- The FPGA itself is built up of lookup tables (LUTs) and other components
- Technology mapping is the process of packing gates into these components
- Each LUT has a fixed propagation delay associated with it, regardless of what it is doing
- DK has its own technology mapping scheme
- Example:

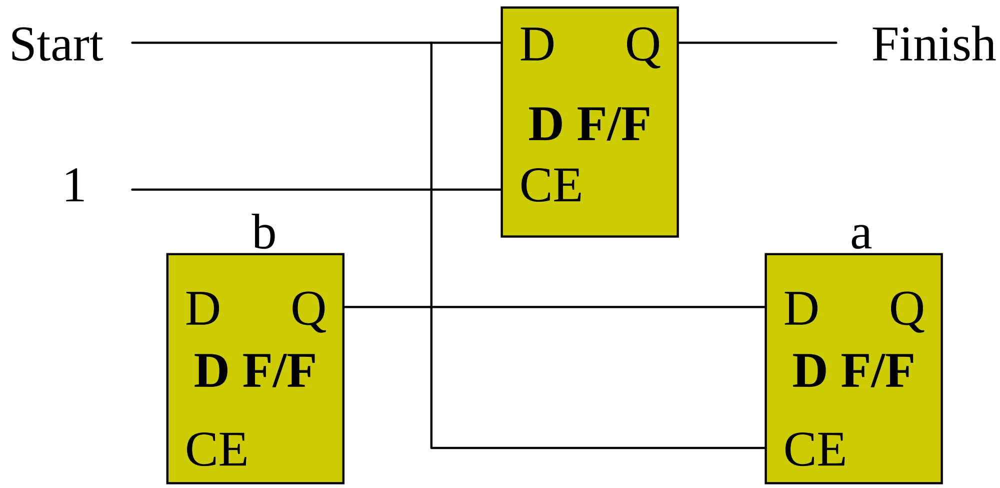
unsigned 1 a, b, c, d, e, f;

f = (a ^ b ^ c ^ d) & e; *//this will require 2 4-input LUTs*

How Handel-C construct map to HW

- Each block has an input signal **START** and output signal **FINISH**
- The **start** signal comes from the enclosing block
- The main function has its own special **start** signal
- The start signal goes high for one clock cycle when the statement is executed

a = b;



Demos

BCD Counter for simulation (DEBUG), BCD Counter for HW, Simple FIR Filter