CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Introduction

Anthony Cabrera

FL24::L02

Washington University in St. Louis

Table of contents

Introduction to High Level Synthesis (HLS)

What does HLS actually do?

From KMN[1]

- HLS analyzes and exploits the concurrency in an algorithm.
- HLS inserts registers as necessary to limit critical paths and achieve a desired clock frequency.
- HLS generates control logic that directs the data path.
- HLS implements interfaces to connect to the rest of the system.
- HLS maps data onto storage elements to balance resource usage and bandwidth.
- HLS maps computation onto logic elements performing user specified and automatic optimizations to achieve the most efficient implementation.

What are the inputs to HLS?

From KMN

- A function specified in C, C++, or SystemC
- A design testbench that calls the function and verifies its correctness by checking the results.
- A target FPGA device
- The desired clock period
- Directives guiding the implementation process

Using a C/C++ to Design Hardware(?)

- **tl;dr**: It's a tradeoff. HLS tools simultaneously limit and enhance expressiveness
- PRO: You get to author HW using a language at a higher level of abstraction
- PRO: more folks are familiar with these "higher-level" languages
- CON: C/C++ was designed as a sequential language; how do you reason about HW concurrency? (it's not always intuitive)
- CON: directives guide HW design, but more difficult to achieve fine-grained control of a design
- PRO: HLS does allow for a variety of different interfaces – DMA, streaming, on-chip memories – and optimizations – pipelining, memory partitioning, bidwidth optimization
- CON: Restrictions on what parts of C/C++ make for valid HLS designs, e.g., system calls and uncommon standard library calls are not supported

FPGA Architecture

FPGA Architecture :: General

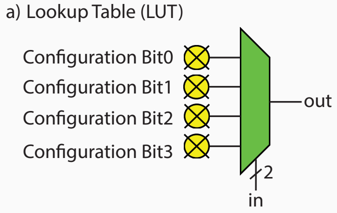
"I'm not very familiar with FPGA architecture. How much do I really need to know?"

- That's okay! You're not expected to understand the low-level details of FPGAs, nor do you need to. We just need to know enough to make sense of the FPGA elements that are targeted by high-level directives.

General FPGA Architecture

- massive arrays of programmable logic and interconnect
- on-chip memories
- custom data paths
- high speed I/O
- microprocessor cores all co-located on the same chip

LUTs



b)

in[1]	in[0]	out
0	0	0
0	1	0
1	0	0
1	1	1

out = in[1] & in[0]

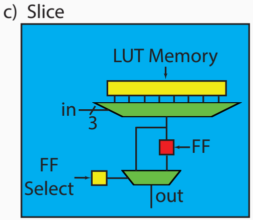


Figure 1: a) shows a 2 input lut, i.e., a 2-LUT. Each of the four configuration bits can be programmed to change the function of the 2-LUT making it a fully programmable 2 input logic gate.
b) provides a sample programming to implement an AND gate. The values in the "out" column from top to bottom correspond directly to configuration bits 0 through 3.
c) shows a simple slice that contains a slightly more complex 3-LUT with the possibility of storing the output into a ff. Note that there are nine configuration bits: eight to program the 3-LUT and one to decide whether the output should be direct from the 3-LUT or the one stored in the ff. More generally, a slice is defined as a small number of lut and ff combined with routing logic (multiplexers) to move inputs, outputs, and internal values between the lut and ff.

FPGA Architecture :: Exercise 1

Implementing logic gates from LUTs

How would you program the 2-LUT from Figure **??** to implement an XOR gate? An OR gate? How many programming bits does an *n* input (*n*-LUT) require?

FPGA Architecture :: Exercise 2

Implementing logic gates from LUTs

How many unique functions can a 2-LUT be programmed to implement? How many unique functions can a *n* input (*n*-LUT) implement?

Networking Slices

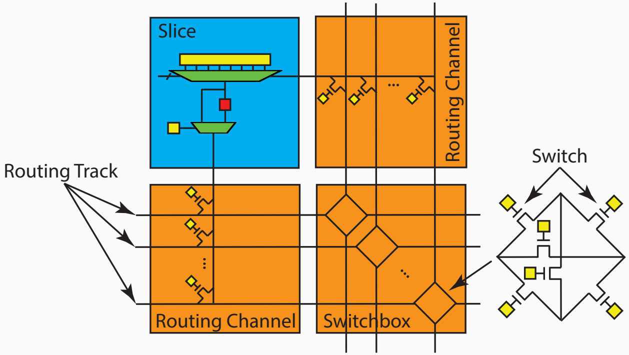


Figure 2: A slice contains a small number of lut and ff. We show a very simple slice with one lut and one ff though generally these have two or more of each. Slices are connected to one another using a routingchannel and switchbox. These two provide a programmable interconnect that provide the data movement between the programmable logic elements (slice). The switchbox has many switches (typically implemented as pass transistors) that allow for arbitrary wiring configurations between the different routing tracks in the routing tracks adjacent to the switchbox.

Island-Style FPGA Architecture

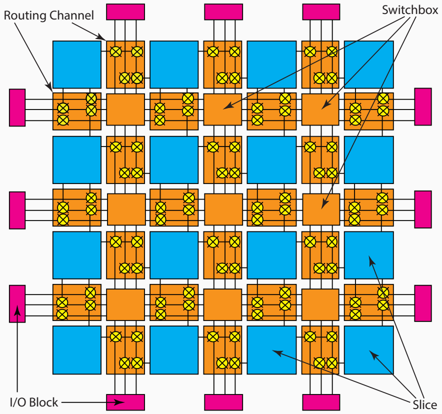


Figure 3: The 2D structure of an fpga showing an island style architecture. The logic and memory resources in the slice are interconnected using routing channels and switchboxes. The input/output (I/O) blocks provide an external interface, e.g., to a memory, microprocessor, sensor, and/or actuator. On some FPGAs, the I/O directly connects to the chip pins. Other FPGAs use the I/O to connect the programmable logic fabric to on-chip resources (e.g., a microprocessor bus or cache).

Modern FPGA Architecture

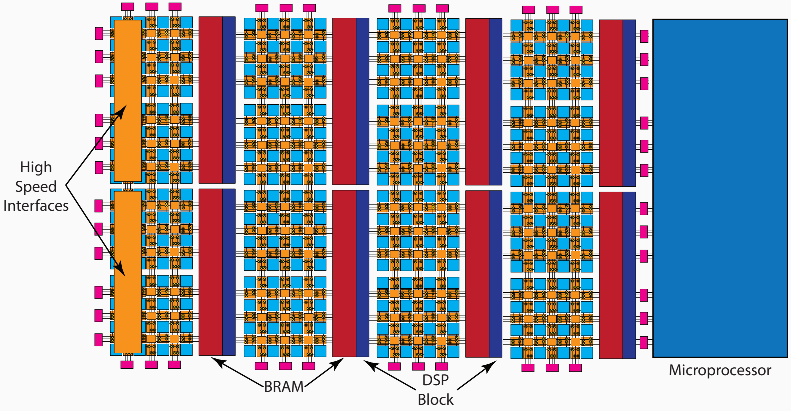


Figure 4: Modern fpga are becoming more heterogenous with a mix of programmable logic elements and “hardened” architectural elements like register files, custom datapaths, and high speed interconnect. The fpga is often paired with one or more microprocessors, e.g., ARM or x86 cores, that coordinates the control of the system.

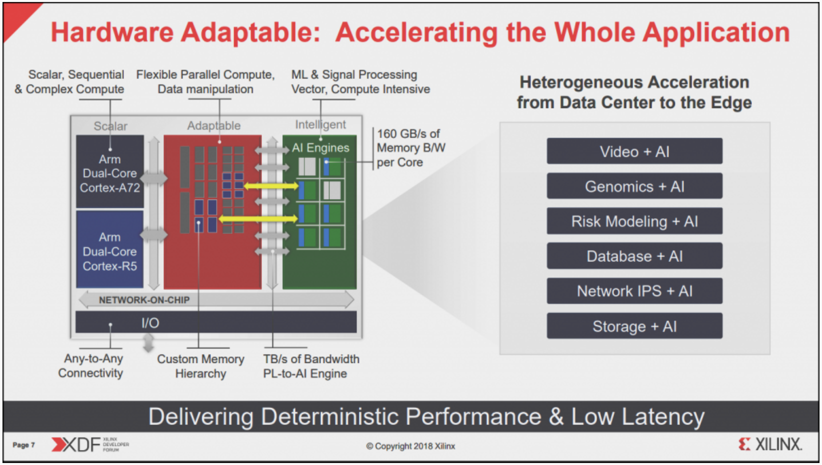
Comparing on- and off-chip memory options

	External Memory	BRAM	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
width	8-64	1-16	1
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

Figure 5: A comparison of three different on- and off-chip memory storage options. External memory provides the most density but has limited total bandwidth. Moving on-chip there are two options: ff and bram. ff have the best total bandwidth but only a limited amount of total data storage capability. bram provide an intermediate value between external memory and ff.

Analagous to the memory hierarchy in typical CPUs.

AMD Versal Adaptive SoC


Hardware Adaptable: Accelerating the Whole Application

Scalar, Sequential & Complex Compute
Flexible Parallel Compute, Data manipulation
ML & Signal Processing Vector, Compute Intensive
Scalar
Adaptable
Intelligent
Arm Dual-Core Cortex-A72
Arm Dual-Core Cortex-R5
AI Engines
160 GB/s of Memory B/W per Core
NETWORK-ON-CHIP
I/O
Any-to-Any Connectivity
Custom Memory Hierarchy
TB/s of Bandwidth PL-to-AI Engine

Heterogeneous Acceleration from Data Center to the Edge

- Video + AI
- Genomics + AI
- Risk Modeling + AI
- Database + AI
- Network IPS + AI
- Storage + AI

Delivering Deterministic Performance & Low Latency

Page 7 XDF XILINX DEVELOPER FORUM © Copyright 2018 Xilinx XILINX.

Figure 6: AMD Versal Adaptive SoC. Combines Programmable logic with AI Engine and DSP

FPGA Design Processs

A Hypothetical Embedded FPGA Design

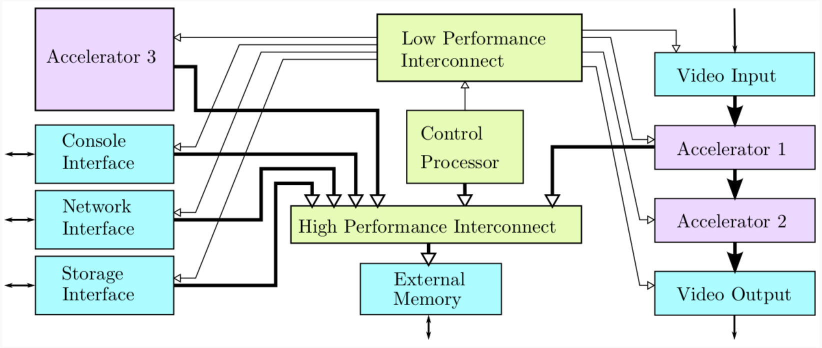


Figure 7: A block diagram showing a hypothetical embedded FPGA design, consisting of I/O interface cores (shown in blue), standard cores (shown in green), and application specific accelerator cores (shown in purple). Note that accelerator cores might have streaming interfaces (Accelerator 2), memory-mapped interfaces (Accelerator 3), or both (Accelerator 1).

Design Optimization

The Clock and Performance Characterization

It is possible to optimize the design by changing the clock frequency. The tool takes as input a target clock frequency, and changing this frequency target will likely result in the tool generating different implementations. We will revisit this concept throughout the semester.. For example, there are constraints the are imposed on the HLS tool depending on the clock period. Further, increasing the clock period can increase the throughput by employing operation chaining (e.g., pipelining).

Tasks as atomic units of behavior

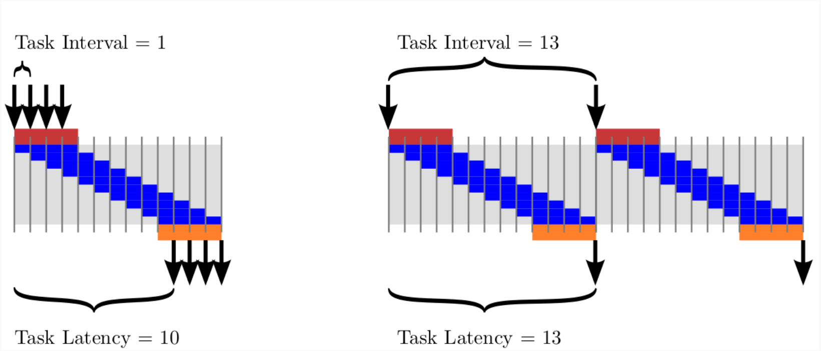


Figure 8: A block diagram showing a hypothetical embedded FPGA design, consisting of I/O interface cores (shown in blue), standard cores (shown in green), and application specific accelerator cores (shown in purple). Note that accelerator cores might have streaming interfaces (Accelerator 2), memory-mapped interfaces (Accelerator 3), or both (Accelerator 1).

Helpful to think about this like you would a pipeline diagram in Computer Architecture.

Area/Throughput Tradeoffs

```c
#define NUM_TAPS 4

void fir(int input, int *output, int taps[NUM_TAPS])
{
  static int delay_line[NUM_TAPS] = {};

  int result = 0;
  for (int i = NUM_TAPS - 1; i > 0; i--) {
    delay_line[i] = delay_line[i - 1];
  }
  delay_line[0] = input;

  for (int i = 0; i < NUM_TAPS; i++) {
    result += delay_line[i] * taps[i];
  }

  *output = result;
}
```

Figure 10: Code for a four tap FIR filter. Given this HLS description, what circuit does this actually create?

The Vitis tools will generate an optimized but largely sequential architecture in the

Area/Throughput Tradeoffs

```
fir:
.frame r1,0,r15 # vars= 0, regs= 0, args= 0
.mask 0x00000000
addik r3,r0,delay_line.1450
lwi r4,r3,8 # Unrolled loop to shift the delay line
swi r4,r3,12
lwi r4,r3,4
swi r4,r3,8
lwi r4,r3,0
swi r4,r3,4
swi r5,r3,0 # Store the new input sample into the delay line
addik r5,r0,4  # Initialize the loop counter
addk r8,r0,r0 # Initialize accumulator to zero
addk r4,r8,r0 # Initialize index expression to zero
$L2:
muli r3,r4,4 # Compute a byte offset into the delay_line array
addik r9,r3,delay_line.1450
lw r3,r3,r7 # Load filter tap
lwi r9,r9,0 # Load value from delay line
mul r3,r3,r9 # Filter Multiply
addk r8,r8,r3 # Filter Accumulate
addik r5,r5,-1 # update the loop counter
bneid r5,$L2
addik r4,r4,1 # branch delay slot, update index expression

rtsd r15, 8
swi r8,r6,0  # branch delay slot, store the output
.end fir
```

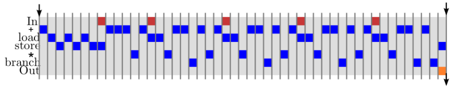


Figure 11: Example of a resulting sequentially generated architecture.

Area/Throughput Tradeoffs

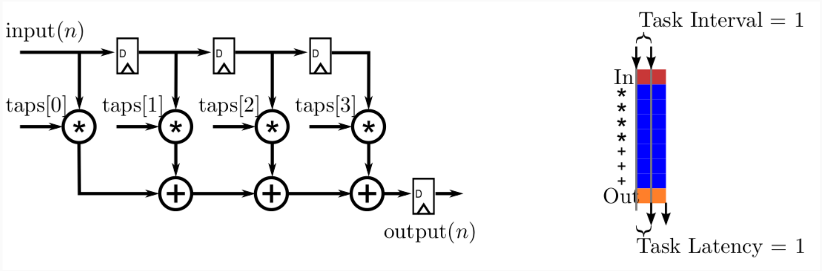


Figure 12: Trading area for throughput

References i

R. Kastner, J. Matai, and S. Neuendorffer.
Parallel Programming for FPGAs.
ArXiv e-prints, May 2018.