High-Level Synthesis

Creating Custom Circuits from High-Level Code

Hao Zheng
Comp Sci & Eng
University of South Florida
Existing Design Flow

→ Register-transfer (RT) synthesis
  → Specify RT structure (muxes, registers, etc)
  → Allows precise specification
  → But, time consuming, difficult, error prone
Existing Design Flow

Historically, the programming model of an FPGA was centered on register-transfer level (RTL) descriptions instead of C/C++. Although this model of design capture is completely compatible with ASIC design, it is analogous to assembly language programming in software engineering.

Figure 1-1 shows a traditional FPGA design flow with RTL as the design capture method, which illustrates how the programming model difference affects implementation time and achievable performance for different computation platforms.

As shown in Figure 1-1, arriving at an initial working version of a software program occurs relatively quickly in the project design cycle for both standard and specialized processors. After the initial working version, additional development effort must be allotted to achieve maximum performance on any implementation platform.

This figure also shows the time it takes to develop the same software application for an FPGA platform. Both the initial and optimized versions of an application provide significant performance when compared against the same stages for both standard and specialized processors. RTL coding and an FPGA optimized application result in the highest performance implementation.

However, the development time required to arrive at this implementation is beyond the scope of a typical software development effort. Therefore, FPGAs were traditionally used only for those applications requiring a performance profile that could not be achieved by any other means, such as designs with multiple processors.
Forthcoming Design Flow

C/C++, Java, etc. → High-level Synthesis → Synthesizable HDL → RT Synthesis → Netlist → Physical Design → Bitfile → ASIC

- Technology Mapping
- Placement
- Routing

FPGA
Guide Organization

There is a significant difference between the performance of an FPGA and other processors for the same C/C++ application. The following chapters in this guide describe the reasons behind this dramatic performance difference and introduce how the Vivado HLS compiler works.

Chapter 2: What is an FPGA?

Chapter 2, What is an FPGA? introduces the computational elements available in an FPGA and how they compare to a processor. It covers topics such as FPGA memory hierarchy, logic elements, and how these elements interrelate.

Chapter 3: Basic Concepts of Hardware Design

The difference between the hardware of a processor and an FPGA affects how a compiler for each target works.
HLS Overview

→ Input:
  → High-level languages (e.g., C)
  → Behavioral hardware description languages (e.g., VHDL)
  → State diagrams / logic networks

→ Tools:
  → Parser
  → Library of modules

→ Constraints:
  → Area constraints (e.g., # modules of a certain type)
  → Delay constraints (e.g., set of operations finish in # clock cycles)

→ Output – RTL models
  → Operation scheduling (time) and binding (resource)
  → Control generation and detailed interconnections
High-level Synthesis - Benefits

- Ratio of C to VHDL developers (10000:1 ?)
- Easier to specify complex functions
- Technology/architecture independent designs
- Manual HW design potentially slower
  - Similar to assembly code era
  - Programmers used to beat compiler
  - But, no longer the case
- Ease of HW/SW partitioning
  - Enhance overall system efficiency
- More efficient verification and validation
  - Easier to V & V of high-level code
High-level Synthesis

→ More challenging than SW compilation
  → Compilation maps behavior into assembly instructions
  → Architecture is known to compiler
→ HLS creates a custom architecture to execute specified behavior
  → Huge hardware exploration space
  → Best solution may include microprocessors
→ Ideally, should handle any high-level code
  → But, not all code appropriate for hardware
High-level Synthesis: An Example

First, consider how to manually convert high-level code into circuit

Steps

1) Build FSM for controller
2) Build datapath based on FSM
A Manual Example

→ Build a FSMD

```
acc = 0;
for (i=0; i < 128; i++)
    acc += a[i];
```
A Manual Example – Cont’d

→ Combine controller + datapath

Start

Controller

Done

Memory Read

In from memory

Memory address

acc = 0;
for (i=0; i < 128; i++)
    acc += a[i];
acc = 0;
for (i=0; i < 128; i++)
    acc += a[i];
A Manual Example - Optimization

→ Alternatives
   → Use one adder (plus muxes)
A Manual Example – Summary

→ Comparison with high-level synthesis
  → Determining when to perform each operation
    => Scheduling
  → Allocating resource for each operation
    => Resource allocation
  → Mapping operations to allocated resources
    => Binding
High-Level Synthesis

Could be C, C++, Java, Perl, Python, SystemC, ImpulseC, etc.

Usually a RT VHDL/Verilog description, but could as low level as a bit file for FPGA, or a gate netlist.
Main Steps

Front-end

High-level Code

Syntactic Analysis

Intermediate Representation

Optimization

Scheduling/Resource Allocation

Back-end

Binding/Resource Sharing

Cycle accurate RTL code

Converts code to intermediate representation - allows all following steps to use language independent format.

Determines when each operation will execute, and resources used

Maps operations onto physical resources
Parsing & Syntactic Analysis
Syntactic Analysis

• Definition: Analysis of code to verify syntactic correctness
  - Converts code into intermediate representation

• Steps: similar to SW compilation
  1) Lexical analysis (Lexing)
  2) Parsing
  3) Code generation – intermediate representation
Intermediate Representation

→ Parser converts an input program to intermediate representation

→ Why use intermediate representation?
  → Easier to analyze/optimize than source code
  → Theoretically can be used for all languages
    → Makes synthesis back end language independent

(Schematic diagram showing flow from syntactic analysis to intermediate representation to back end)
Intermediate Representation

- Different Types
  - Abstract Syntax Tree
  - Control/Data Flow Graph (CDFG)
  - Sequencing Graph

- We will focus on CDFG
  - Combines control flow graph (CFG) and data flow graph (DFG)
    - CFG ---> controller
    - DFG ---> datapath
Control Flow Graphs (CFGs)

- Represents control flow dependencies of *basic blocks*

- A basic block is a section of code that always executes from beginning to end
  - i.e. no jumps into or out of block, nor branching

```c
acc = 0;
for (i=0; i < 128; i++)
acc += a[i];
```
Control Flow Graphs: Your Turn

• Find a CFG for the following code.

```java
i = 0;
while (i < 10) {
    if (x < 5)
        y = 2;
    else if (z < 10)
        y = 6;
    i++;
}
```
Data Flow Graphs

→ Represents data dependencies between operations within a single basic block

\[ x = a + b; \]
\[ y = c \times d; \]
\[ z = x - y; \]

\[
\begin{align*}
\text{Data Flow Graph:} & \\
\text{Operations:} & +, -, \times \\
\text{Inputs:} & a, b, c, d \\
\text{Outputs:} & x, z, y
\end{align*}
\]
Control/Data Flow Graph

→ Combines CFG and DFG
→ Maintains DFG for each node of CFG

```c
acc = 0;
for (i=0; i < 128; i++)
    acc += a[i];
```

Diagram: [Control/Data Flow Graph Diagram]
Transformation/Optimization
Synthesis Optimizations

→ After creating CDFG, HLS optimizes it with the following goals
   → Reduce area
   → Reduce latency
   → Increase parallelism
   → Reduce power/energy

→ 2 types of optimizations
   → Data flow optimizations
   → Control flow optimizations
Data Flow Optimizations

→ Tree-height reduction

→ Generally made possible from commutativity, associativity, and distributivity

\[ x = a + b + c + d \]
Data Flow Optimizations

→ **Operator Strength Reduction**
  → Replacing an expensive ("strong") operation with a faster one
  → Common example: replacing multiply/divide with shift

```
b[i] = a[i] * 8;
b[i] = a[i] << 3;
a = b * 5;
c = b << 2;
a = b + c;
```

1 multiplication

```
b[i] = a[i] * 13;
c = b << 2;
c = d << 3;
d = b << 3;
a = c + d + b;
```

0 multiplications
Data Flow Optimizations

• **Constant propagation**
  
  - Statically evaluate expressions with constants

```
x = 0;
y = x * 15;
z = y + 10;
```

```
x = 0;
y = 0;
z = 10;
```
Data Flow Optimizations

→ Function Specialization

→ Create specialized code for common inputs
  → Treat common inputs as constants
  → If inputs not known statically, must include if statement for each call to specialized function

```c
int f (int x) {
    y = x * 15;
    return y + 10;
}
```

```c
for (I=0; I < 1000; I++)
    f(0);
...}
```

Treat frequent input as a constant

```c
int f_opt () {
    return 10;
}
```

```c
for (I=0; I < 1000; I++)
    f_opt();
...}
```
Data Flow Optimizations

→ **Common sub-expression elimination**

→ If expression appears more than once, repetitions can be replaced

\[
\begin{align*}
a &= x + y; \\
\cdots & \\
\cdots & \\
b &= c \times 25 + x + y;
\end{align*}
\]

\[
\begin{align*}
a &= x + y; \\
\cdots & \\
\cdots & \\
b &= c \times 25 + a;
\end{align*}
\]

\[x + y\text{ already determined}\]
Data Flow Optimizations

→ Dead code elimination
  → Remove code that is never executed
    → May seem like stupid code, but often comes from constant propagation or function specialization

```c
int f (int x) {
    if (x > 0 )
        a = b * 15;
    else
        a = b / 4;
    return a;
}
```

```c
int f_opt () {
    a = b * 15;
    return a;
}
```

Specialized version for \( x > 0 \) does not need else branch - “dead code”
Data Flow Optimizations

→ **Code motion** (hoisting/sinking)
→ Avoid same repeated computation

```java
for (I=0; I < 100; I++) {
    z = x + y;
    b[i] = a[i] + z;
}
```

```
z = x + y;
for (I=0; I < 100; I++) {
    b[i] = a[i] + z;
}
```
Control Flow Optimizations

→ Loop Unrolling
  → Replicate body of loop
  → May increase parallelism

```
for (i=0; i < 128; i++)
a[i] = b[i] + c[i];
```

```
for (i=0; i < 128; i+=2) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1]
}
```
Control Flow Optimizations

**→ Function inlining** – replace function call with body of function

- Common for both SW and HW
- SW: Eliminates function call instructions
- HW: Eliminates unnecessary control states

```c
for (i=0; i < 128; i++)
    a[i] = f( b[i], c[i] );

... 
int f (int a, int b) {
    return a + b * 15;
}
```

```c
for (i=0; i < 128; i++)
    a[i] = b[i] + c[i] * 15;
```
Control Flow Optimizations

→ **Conditional Expansion** – replace if with logic expression

→ Execute **if/else** bodies in parallel

```
y = ab
if (a)
  x = b+d
else
  x = bd
```

Can be further optimized to:

```
y = ab
x = y + d(a+b)
```
Example

→ Optimize this

```plaintext
x = 0;
y = a + b;
if (x < 15)
   z = a + b - c;
else
   z = x + 12;
output = z * 12;
```
Scheduling/Resource Allocation
Scheduling

• Scheduling assigns a start time to each operation in DFG
  – Start times must not violate dependencies in DFG
  – Start times must meet performance constraints
    + Alternatively, resource constraints

• Performed on the DFG of each CFG node
  – Cannot execute multiple CFG nodes in parallel
Scheduling – Examples

Cycle 1
+ 
Cycle 2
+ 
Cycle 3
+

Cycle 1
+ 
Cycle 2
+ 
Cycle 3
+

Cycle 1
+ 
Cycle 2
+ 
Cycle 3
+

Cycle 1
+ 
Cycle 2
+
Scheduling Problems

→ Several types of scheduling problems
  → Usually some combination of performance and resource constraints

→ Problems:
  → Unconstrained
    → Not very useful, every schedule is valid
  → Minimum latency
  → Latency constrained
  → Minimum-latency, resource constrained
    → i.e. find the schedule with the shortest latency, that uses less than a specified # of resources
    → NP-Complete
  → Minimum-resource, latency constrained
    → i.e. find the schedule that meets the latency constraint (which may be anything), and uses the minimum # of resources
    → NP-Complete
Minimum Latency Scheduling

→ ASAP (as soon as possible) algorithm
  → Find a candidate node
    → Candidate is a node whose predecessors have been scheduled and completed (or has no predecessors)
  → Schedule node one cycle later than max cycle of predecessor
  → Repeat until all nodes scheduled

Minimum possible latency - 4 cycles
Minimum Latency Scheduling

→ ALAP (as late as possible) algorithm
  → Run ASAP, get minimum latency L
  → Find a candidate
    → Candidate is node whose successors are scheduled (or has none)
  → Schedule node one cycle before min cycle of successor
    → Nodes with no successors scheduled to cycle L
  → Repeat until all nodes scheduled

L = 4 cycles
Latency-Constrained Scheduling

→ Instead of finding the minimum latency, find latency less than $L$

→ Solutions:
  → Use ASAP, verify that minimum latency $\leq L$.
  → Use ALAP starting with cycle $L$ instead of minimum latency (don’t need ASAP)
Scheduling with Resource Constraints

Schedule must use less than specified number of resources

Constraints: 1 ALU (+/-), 1 Multiplier
Scheduling with Resource Constraints

Schedule must use less than specified number of resources

Constraints: 2 ALU (+/-), 1 Multiplier
Minimum-Latency, Resource-Constrained Scheduling

→ Definition: Given resource constraints, find schedule that has the minimum latency

→ Example:

Constraints: 1 ALU (+/-), 1 Multiplier
Minimum-Latency, Resource-Constrained Scheduling

→ Definition: Given resource constraints, find schedule that has the minimum latency

→ Example:

Constraints: 1 ALU (+/-), 1 Multiplier

Diagram: [Diagram showing resource constraints and operations]
Minimum-Latency, Resource-Constrained Scheduling

→ Definition: Given resource constraints, find schedule that has the minimum latency

→ Example:

Constraints: 1 ALU (+/-), 1 Multiplier
Binding/Resource Sharing
Binding

During scheduling, we determine:

→ When operations will execute
→ How many resources are needed

We still need to decide which operations execute on which resources – binding

→ If multiple operations use the same resource, we need to decide how resources are shared - resource sharing.
Map operations onto resources such that operations in same cycle do not use same resource
Binding

Many possibilities

Bad binding may increase resources, require huge steering logic, reduce clock, etc.
Binding

→ Cannot do this

→ 1 resource can’t perform multiple ops simultaneously!

2 ALUs (+/-), 2 Multipliers

Cycle1

Cycle2

Cycle3

Cycle4
Translation to Datapath

1) Add resources and registers
2) Add mux for each input
3) Add input to left mux for each left input in DFG
4) Do same for right mux
5) If only 1 input, remove mux
Summary
Main Steps

→ Front-end (lexing/parsing) converts code into intermediate representation – CDFG

→ Scheduling assigns a start time for each operation in DFG
  → CFG node start times defined by control dependencies
  → Resource allocation determined by schedule

→ Binding maps scheduled operations onto physical resources
  → Determines how resources are shared

→ Big picture:
  → Scheduled/Bound DFG can be translated into a datapath
  → CFG can be translated to a controller
  → High-level synthesis can create a custom circuit for any CDFG!
Limitations

→ Task-level parallelism
  → Parallelism in CDFG limited to individual control states
    → Cannot have multiple states executing concurrently
  → Potential solution: use model other than CDFG
    → Kahn Process Networks
      → Nodes represents parallel processes/tasks
      → Edges represent communication between processes
    → High-level synthesis can create a controller+datapath for each process
      → Must also consider communication buffers

→ Challenge:
  → Most high-level code does not have explicit parallelism
    → Difficult/impossible to extract task-level parallelism from code
Limitations

→ Coding practices limit circuit performance
  → Very often, languages contain constructs not appropriate for circuit implementation
    → Recursion, pointers, virtual functions, etc.

→ Potential solution: use specialized languages
  → Remove problematic constructs, add task-level parallelism

→ Challenge:
  → Difficult to learn new languages
  → Many designers resist changes to tool flow
Limitations

→ Expert designers can achieve better circuits
  → High-level synthesis has to work with specification in code
    → Can be difficult to automatically create efficient pipeline
    → May require dozens of optimizations applied in a particular order
  → Expert designer can transform algorithm
    → Synthesis can transform code, but can’t change algorithm

→ Potential Solution: ???
  → New language?
  → New methodology?
  → New tools?
Vivado HLS Highlights
Overview

High-Level Synthesis

www.xilinx.com

Chapter 1: High-Level Synthesis

In any C program the top-level function is called `main()`. In the Vivado HLS design flow, any sub-function below the level of `main()` can be specified as the top-level function for synthesis. The function `main()` cannot be synthesized.

- Only one function can be selected as the top-level function for synthesis.
- Any sub-functions in the hierarchy below the function marked for synthesis will also be synthesized.

X-Ref Target - Figure 1-4

Figure 1-4: Vivado HLS Overview
## Typical C/C++ Construct to RTL Mapping

<table>
<thead>
<tr>
<th>C Constructs</th>
<th>HW Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Modules</td>
</tr>
<tr>
<td>Arguments</td>
<td>Input/output ports</td>
</tr>
<tr>
<td>Operators</td>
<td>Functional units</td>
</tr>
<tr>
<td>Scalars</td>
<td>Wires or registers</td>
</tr>
<tr>
<td>Arrays</td>
<td>Memories</td>
</tr>
<tr>
<td>Control flows</td>
<td>Control logics</td>
</tr>
</tbody>
</table>
Function Hierarchy

→ Each function is synthesized to a RTL module
  → Function inlining eliminates hierarchy
→ The function main() cannot be synthesized
  → Used to develop C-testbench

Source code

```c
void A() { .. body A .. }
void C() { .. body C .. }
void B() {
    C();
}
void TOP() {
    A(...);
    B(...);
}
```

RTL hierarchy
Function Arguments

→ Function arguments become module ports
→ Interface follows certain protocol to synchronize data exchange

```c
void TOP(int* in1, int* in2, int* out1)
{
    *out1 = *in1 + *in2;
}
```
Expressions

Expresssions and operations are synthesized to datapath

Timing constraints influence the degree of registering
Arrays

→ By default, an array in C code is typically implemented by a memory block in the RTL
  → Read & write array -> RAM; Constant array -> ROM
→ An array can be partitioned and map to multiple RAMs
→ Multiples arrays can be merged and map to one RAM
→ An array can be partitioned into individual elements and map to registers

```c
void TOP(int)
{
  int A[N];
  for (i = 0; i < N; i++)
}
```
Loops

- By default, loops are rolled
  - Each loop iteration corresponds to a “sequence” of states (possibly a DAG)
  - This state sequence will be repeated multiple times based on the loop trip count

```c
void TOP (...) {
  ...
  for (i = 0; i < N; i++)
    b += a[i];
}
```
Loop Unrolling

→ Loop unrolling to expose higher parallelism and achieve shorter latency

→ Pros
  → Decrease loop overhead
  → Increase parallelism for scheduling
  → Facilitate constant propagation and array-to-scalar promotion

→ Cons – increase operation count, which may negatively impact area, power, and timing

```c
for (int i = 0; i < N; i++)
    A[i] = C[i] + D[i];
```

```c
A[0] = C[0] + D[0];
.....
```
Loop Pipelining

→ Loop pipelining is one of the most important optimizations for high-level synthesis
  → Allows a new iteration to begin processing before the previous iteration is complete
  → Key metric: **Initiation Interval (II)** in # cycles

\[
\text{for } (i = 0; i < N; ++i) \quad p[i] = x[i] \times y[i];
\]

\[
\begin{array}{c}
\text{i=0} \\
\text{i=1} \\
\text{i=2} \\
\text{i=3}
\end{array}
\]

- **ld** – Load
- **st** – Store