Vivado HLS Tutorial

Steve Dai, Sean Lai, Zhiru Zhang
School of Electrical and Computer Engineering
Agenda

▸ Logistics and questions

▸ What is high-level synthesis?

▸ C-based synthesis

▸ Case study: FIR filter
High-Level Synthesis (HLS)

- **What**
  - *Automated* design process that transforms a **high-level functional specification to optimized register-transfer level (RTL)** descriptions for efficient hardware implementation

- **Why**
  - **Productivity**
    - lower design complexity and faster simulation speed
  - **Permutability**
    - rapid design space exploration -> higher quality of result (QoR)
  - **Portability**
    - single source -> multiple implementations
Permutability: Faster Design Space Exploration

Control-Data Flow Graph

Latency

Area

Throughput

\[ t_{\text{clk}} = 3 \quad d_{\text{add}} \]
\[ T_1 = 1 / t_{\text{clk}} \]
\[ A_1 = 3 * A_{\text{add}} \]

\[ t_{\text{clk}} \approx d_{\text{add}} + d_{\text{setup}} \]
\[ T_2 = 1 / (3 * t_{\text{clk}}) \]
\[ A_2 = A_{\text{add}} + 2 * A_{\text{reg}} \]

\[ t_{\text{clk}} = d_{\text{add}} + d_{\text{setup}} \]
\[ T_3 = 1 / t_{\text{clk}} \]
\[ A_3 = 3 * A_{\text{add}} + 6 * A_{\text{reg}} \]
Typical C/C++ Synthesizable Subset

- **Data types:**
  - Primitive types: (u)char, (u)short, (u)int, (u)long, float, double
  - Arbitrary precision integer or fixed-point types
  - Composite types: array, struct, class
  - Templated types: template
  - Statically determinable pointers

- No support for dynamic memory allocations

- No support for recursive function calls
<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 usually translated into an RTL module
  - Functions may be inlined to dissolve their hierarchy

Source code

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

RTL hierarchy

```
  TOP
  ├── A
  │    ├── B
  │    └── C
```

6
Function Arguments

- Function arguments become ports on the RTL blocks

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

- Additional control ports are added to the design

Input/output (I/O) protocols
- Allow RTL blocks to automatically synchronize data exchange
HLS generates datapath circuits mostly from expressions

- Timing constraints influence the degree of registering

```c
char A, B, C, D, int P;
P = (A+B)*C+D
```
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

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

- 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
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];
```

```
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

```plaintext
for (i = 0; i < N; ++i)
    p[i] = x[i] * y[i];
```

```
ld  st  st  st  st
i=0

ld  ld  st  st
i=1

ld  ld  ld  st
i=2

ld  ld  ld  ld  st
i=3
```

**ld** – Load  
**st** – Store
Case Study:
Finite Impulse Response (FIR) Filter
Finite Impulse Response (FIR) Filter

\[ y[n] = \sum_{i=0}^{N} b_i x[n - i] \]

- \( x[n] \) input signal
- \( y[n] \) output signal
- \( N \) filter order
- \( b_i \) \( i \)th filter coefficient

// original, non-optimized version of FIR

```c
#define SIZE 128
#define N 10

void fir(int input[SIZE], int output[SIZE]) {
    // FIR coefficients
    int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};

    // exact translation from FIR formula above
    for (int n = 0; n < SIZE; n++) {
        int acc = 0;
        for (int i = 0; i < N; i++) {
            if (n - i >= 0)
                acc += coeff[i] * input[n - i];
        }
        output[n] = acc;
    }
}
```
Server Setup

- Log into ece-linux server
  - Host name: ecelinux.ece.cornell.edu
  - User name and password: [Your NetID credentials]

- Setup tools for this class
  - Source class setup script to setup Vivado HLS
    ```
    > source /classes/ece5775/setup-ece5775.sh
    ```

- Test Vivado HLS
  - Open Vivado HLS interactive environment
    ```
    > vivado_hls -i
    ```
  - List the available commands
    ```
    > help
    ```
Copy FIR Example to Your Home Directory

- Design files
  - fir.h: function prototypes
  - fir_.c: function definitions
- Testbench files
  - fir-top.c: function used to test the design
- Synthesis configuration files
  - run.tcl: script for configuring and running Vivado HLS
Project Tcl Script

#===================================
# run.tcl for FIR
#===================================

# open the HLS project fir.prj
open_project fir.prj -reset

# set the top-level function of the design to be fir
set_top fir

# add design and testbench files
add_files fir_initial.c
add_files -tb fir-top.c

open_solution "solution1"

# use Zynq device
set_part xc7z020clg484-1

# target clock period is 10 ns
create_clock -period 10

# do a c simulation
csim_design

# synthesize the design
csynth_design

# do a co-simulation
cosim_design

# close project and quit
close_project

# exit Vivado HLS
quit

You can use multiple Tcl scripts to automate different runs with different configurations.
Synthesize and Simulate the Design

> vivado_hls -f run.tcl

Generating csim.exe
128/128 correct values!
INFO: [SIM 211-1] CSim done with 0 errors.

INFO: [HLS 200-10] -- Scheduling module 'fir'
INFO: [HLS 200-10]

INFO: [HLS 200-10] -- Exploring micro-architecture for module 'fir'
INFO: [HLS 200-10]

INFO: [HLS 200-10] -- Generating RTL for module 'fir'
INFO: [HLS 200-10]

INFO: [COSIM 212-14] Instrumenting C test bench ...

INFO: [COSIM 212-12] Generating RTL test bench ...
INFO: [COSIM 212-15] Starting XSIM ...

INFO: [COSIM 212-316] Starting C post checking ...
128/128 correct values!

INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***

SW simulation only. Same as simply running a software program.

HLS
Synthesizing C to RTL

HW-SW co-simulation. SW test bench invokes RTL simulation.
Synthesis reports of each function in the design, except those inlined.
Default Microarchitecture

```c
void fir(int input[SIZE], int output[SIZE]) {
    // FIR coefficients
    int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
    // Shift registers
    int shift_reg[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    // loop through each output
    for (int i = 0; i < SIZE; i++) {
        int acc = 0;
        // shift registers
        for (int j = N - 1; j > 0; j--) {
            shift_reg[j] = shift_reg[j - 1];
        }
        // put the new input value into the first register
        shift_reg[0] = input[i];
        // do multiply-accumulate operation
        for (j = 0; j < N; j++) {
            acc += shift_reg[j] * coeff[j];
        }
        output[i] = acc;
    }
}
```

Possible optimizations
- Loop unrolling
- Array partitioning
- Pipelining
void fir(int input[SIZE], int output[SIZE]) {
    
    // loop through each output
    for (int i = 0; i < SIZE; i++) {
        int acc = 0;
        // shift the registers
        for (int j = N-1; j > 0; j--) {
            #pragma HLS unroll
            shift_reg[j] = shift_reg[j-1];
        }
        ... 
        // do multiply-accumulate operation
        for (j = 0; j < N; j++) {
            #pragma HLS unroll
            acc += shift_reg[j] * coeff[j];
        }
    }
}

// unrolled shift registers
shift_reg[9] = shift_reg[8];
shift_reg[8] = shift_reg[7];
...
shift_reg[1] = shift_reg[0];

// unrolled multiply-accumulate
acc += shift_reg[0] * coeff[0];
acc += shift_reg[1] * coeff[1];
...
acc += shift_reg[9] * coeff[9];
Microarchitecture after Unrolling

![Diagram showing microarchitecture after unrolling]

Default

Unrolled
void fir(int input[SIZE], int output[SIZE]) {
    // FIR coefficients
    int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
    // Shift registers
    int shift_reg[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=0
    ...
}
Microarchitecture after Partitioning

\[
\begin{align*}
    &x_n \rightarrow \text{shift\_reg}[0] \\
    &\times \quad \text{coeff}[0] \quad \times \quad \text{coeff}[1] \quad \times \quad \text{coeff}[2] \quad \times \quad \text{coeff}[8] \quad \times \quad \text{coeff}[9] \\
    &+ \quad \text{shift\_reg}[1] \quad + \quad \text{shift\_reg}[2] \quad + \quad \text{shift\_reg}[8] \quad + \quad \text{shift\_reg}[9] \\
    &\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \\
    &y_n \quad \text{Unrolled} \\
\end{align*}
\]

\[
\begin{align*}
    &x_n \rightarrow \text{shift\_reg}[0] \\
    &\times \quad \text{coeff}[0] \quad \times \quad \text{coeff}[1] \quad \times \quad \text{coeff}[2] \quad \times \quad \text{coeff}[8] \quad \times \quad \text{coeff}[9] \\
    &+ \quad \text{shift\_reg}[1] \quad + \quad \text{shift\_reg}[2] \quad + \quad \text{shift\_reg}[8] \quad + \quad \text{shift\_reg}[9] \\
    &\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \\
    &y_n \quad \text{Partitioned} \\
\end{align*}
\]
void fir(int input[SIZE], int output[SIZE]) {
  ...

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    #pragma HLS pipeline II=1
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j - 1];
    }
    ...
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += shift_reg[j] * coeff[j];
    }
    ...
  }
  ...
}
Fully Pipelined Implementation

\[
\begin{align*}
\text{Previous sample} & \quad x_{n-1} & \text{Current sample} & \quad x_n \\
\text{shift_reg}[0] & \quad \times \quad \text{coeff}[0] & \quad \text{shift_reg}[0] & \quad \times \quad \text{coeff}[0] \\
\text{shift_reg}[1] & \quad \times \quad \text{coeff}[1] & \quad \text{shift_reg}[1] & \quad \times \quad \text{coeff}[1] \\
\text{shift_reg}[9] & \quad \times \quad \text{coeff}[2] & \quad \text{shift_reg}[9] & \quad \times \quad \text{coeff}[2] \\
\text{shift_reg}[9] & \quad \times \quad \text{coeff}[8] & \quad \text{shift_reg}[9] & \quad \times \quad \text{coeff}[8] \\
\text{shift_reg}[9] & \quad \times \quad \text{coeff}[9] & \quad \text{shift_reg}[9] & \quad \times \quad \text{coeff}[9] \\
\end{align*}
\]
void fir(int input[SIZE], int output[SIZE]) {
    …
    // loop through each output
    for (int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        int acc = 0;
        // shift the registers
        for (int j = N - 1; j > 0; j--) {
            #pragma HLS unroll
            shift_reg[j] = shift_reg[j - 1];
        }
        …
        // do multiply-accumulate operation
        for (j = 0; j < N; j++) {
            #pragma HLS unroll
            acc += shift_reg[j] * coeff[j];
        }
        …
    }
}