ECE 5997
Hardware Accelerator Design & Automation
Fall 2021

Tutorial on C-Based HLS
Announcements

- Lab 1 released
  - Due Wed Oct 27

- Friday lecture takes place in Rhodes 310
Agenda

- Introduction to high-level synthesis (HLS)
  - C-based synthesis
  - Common HLS optimizations

- Case study: Optimizing FIR filter using Vivado HLS
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
    • Input spec. to HLS is typically untimed or partially timed

▷ Why
  – **Productivity**: Lower design complexity & faster simulation speed
  – **Portability**: Single (untimed) source $\rightarrow$ multiple implementations
  – **Quality**: Quicker design space exploration $\rightarrow$ higher quality
Design Productivity: RTL vs. HLS

- A GCD unit with handshake

**HLS Code**

```c
void GCD ( msg& req, msg& resp ) {
    short a = req.msg_a;
    short b = req.msg_b;
    while ( a != b ) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    resp.msg = a;
}
```

**Manual RTL (partial)**

```verilog
module GcdUnitRTL ( 
    input wire [ 0:0] clk, 
    input wire [ 31:0] req_dat, 
    output wire [ 0:0] req_rdy, 
    input wire [ 0:0] req_val, 
    input wire [ 0:0] reset, 
    output wire [ 15:0] resp_dat, 
    input wire [ 0:0] resp_rdy, 
    output wire [ 0:0] resp_val );

always @ (*) begin 
    if ((curr_state__0 == STATE_IDLE))
        next_state__0 = STATE_CALC;
    if ((curr_state__0 == STATE_CALC))
        next_state__0 = STATE_DONE;
    if ((curr_state__0 == STATE_DONE))
        next_state__0 = STATE_IDLE;
end

Output logic

GCD Unit

req.dat  req.val  req.rdy

resp.dat  resp.val  resp.rdy

State transition

Module declaration

always @ (*) begin
    if ((current_state__1 == STATE_IDLE))
        req_rdy = 1; resp_val = 0; 
        a_mux_sel = A_MUX_SEL_IN; b_mux_sel = B_MUX_SEL_IN; 
        a_reg_en = 1; b_reg_en = 1;
    if ((current_state__1 == STATE_CALC))
        do_swap = is_a_lt_b; do_sub = ~is_b_zero; 
        req_rdy = 0; resp_val = 0; 
        a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB; 
        a_reg_en = 1; b_reg_en = do_swap; 
        b_mux_sel = B_MUX_SEL_A;
    else
        if ((current_state__1 == STATE_DONE))
            req_rdy = 0; resp_val = 1; 
            a_mux_sel = A_MUX_SEL_X; b_mux_sel = B_MUX_SEL_X; 
            a_reg_en = 0; b_reg_en = 0;
end
```
**Single Untimed Source to Multiple Implementations**

<table>
<thead>
<tr>
<th>Untimed</th>
<th>Combinational for Latency</th>
<th>Sequential for Area</th>
<th>Pipelined for Throughput</th>
</tr>
</thead>
</table>

Control-Data Flow Graph (CDFG)

\[ \text{out1} = f(\text{in1}, \text{in2}, \text{in3}, \text{in4}) \]

- Untimed
  - \[ t_{\text{clk}} \approx 3 \times d_{\text{add}} \]
  - \[ T_1 = \frac{1}{t_{\text{clk}}} \]
  - \[ A_1 = 3 \times A_{\text{add}} \]

- Combinational for Latency
  - \[ T_2 = \frac{1}{3 \times t_{\text{clk}}} \]

- Sequential for Area
  - \[ t_{\text{clk}} \approx d_{\text{add}} + d_{\text{setup}} \]
  - \[ A_2 = A_{\text{add}} + 2 \times A_{\text{reg}} \]

- Pipelined for Throughput
  - \[ T_3 = \frac{1}{t_{\text{clk}}} \]
  - \[ A_3 = 3 \times A_{\text{add}} + 6 \times A_{\text{reg}} \]
A Representative C-Based HLS Tool

Xilinx Vivado HLS (now Vitis HLS)

- RTL Wrapper
- VHDL Verilog System C
- Script with Constraints

Test bench

C, C++

Constraints/ Directives

RTL Simulation

RTL Synthesis

~10X code size reduction with algorithmic specification

~100X simulation speedup

Behavioral-level IP reuse

Fast architecture exploration
Typical C/C++ Synthesizable Subset

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

- No/limited support for dynamic memory allocations and recursive function calls
Arbitrary Precision Integer

- C/C++ only provides a limited set of native integer types
  - char (8b), short (16b), int (32b), long (?), long long (64b)
  - Byte aligned: efficient in processors

- Arbitrary precision integer in Vivado HLS
  - Signed: `ap_int`; Unsigned `ap_uint`
    - Two’s complement representation for signed integer
  - Templatized class `ap_int<W>` or `ap_uint<W>`
    - W is the user-specified bitwidth

```
#include <ap_int>
...
ap_int<9> x; // 9-bit
ap_uint<24> y; // 24-bit unsigned
ap_uint<512> z; // 512-bit unsigned
```
### Representing Fractional Numbers

- Binary representation can also represent fractional numbers, usually **called fixed-point numbers**, by simply extending the pattern to include negative exponents:
  - Less convenient to use compared to floating-point types
  - Efficient and cheap in application-specific hardware

<table>
<thead>
<tr>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11.25</td>
</tr>
</tbody>
</table>

Binary point

<table>
<thead>
<tr>
<th>$2’c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Fixed-Point Type in Vivado HLS

- Arbitrary precision fixed-point type
  - Signed: \texttt{ap\_fixed}; Unsigned \texttt{ap\_ufixed}
  - Templatized class \texttt{ap\_fixed<W, I, Q, O>}
    - \texttt{W}: total word length
    - \texttt{I}: integer word length
    - \texttt{Q}: quantization mode
    - \texttt{O}: overflow mode

\begin{center}
\begin{tikzpicture}
  \node (W) at (0,0) {W};
  \node (I) at (0,-2) {I};
  \node (Binary_point) at (2,-2) {Binary point};
  \draw[->] (W) -- (1,-1);
  \draw[->] (I) -- (1,-3);
\end{tikzpicture}
\end{center}
Example: Fixed-Point Modeling

- `ap_ufixed<11, 8, AP_TRN, AP_WRAP> x;`

- 11 is the total number of bits in the type
- 8 bits to the left of the decimal point
- `AP_WRAP` defines wrapping behavior for overflow
- `AP_TRN` defines truncation behavior for quantization
Handling Overflow/Underflow

One common (& efficient) way of handling overflow / underflow is to drop the most significant bits (MSBs) of the original number, often called wrapping.

\[
\begin{array}{ccccccc}
-2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^c \\
1 & 0 & 1 & 1 & 0 & 1 & = -4.75 \\
\end{array}
\]

Reduce integer width by 1
Wrap if overflows

Wrapping can cause a negative number to become positive, or a positive to negative.
Fixed-Point Type: Quantization Behavior

- **ap_fixed quantization mode**
  - Determines the behavior of the fixed point type when the result of an operation generates more precision in the LSBs than is available
  - Default mode: AP_TRN (truncation)
  - Other rounding modes: AP_RND, AP_RND_ZERO, AP_RND_INF, …

```
ap_fixed<4, 2, AP_TRN> x = 1.25;  \ (b'01.01)
ap_fixed<3, 2, AP_TRN> y = x;    \ 1.0  \ (b'01.0)
```

```
ap_fixed<4, 2, AP_TRN> x = -1.25; \ (b'10.11)
ap_fixed<3, 2, AP_TRN> y = x;     \ -1.5 \ (b'10.1)
```
Typical C/C++ Constructs to RTL Mapping

<table>
<thead>
<tr>
<th>C/C++ Constructs</th>
<th>RTL 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>
Design Hierarchy

- Each function is usually translated into an RTL module
  - Functions may be inlined to dissolve their hierarchy

```c
void A() { ... /* body of A */ }
void C() { ... /* body of C */ }
void B() {
    C();
}
void TOP() { A(); B(); }
```
Arrays

- By default, an array in C code is typically implemented by a memory module in the RTL
  - Read & write array -> RAM; Constant array -> ROM

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

- An array can be partitioned and map to multiple RAMs
  - An array can be partitioned into individual elements and map to registers (complete partition)
- Multiples arrays can be merged and map to one RAM
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++)
        sum += A[i];
}
```
Loop Unrolling

- Loop unrolling to expose higher parallelism and achieve shorter latency
  - Pros
    - Decrease loop overhead
    - Increase parallelism for scheduling
  - Cons
    - Increase operation count, which may negatively impact area, power, and timing

```c
for (int i = 0; i < 8; i++)
    A[i] = C[i] + D[i];
```

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

- Pipelining is one of the most important optimizations for HLS
  - Key factor: **Initiation Interval (II)**
  - Allows a new iteration to begin processing, II cycles after the start of the previous iteration (II=1 means the loop is fully pipelined)

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

**Pipelined schedule**

- **ld** – Load (memory read)
- **st** – Store (memory write)

**Dataflow of the loop body**
A Tutorial on Vivado HLS (led TAs)
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

```
#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/ece5997/setup-ece5997.sh

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

> cd ~
> cp -r /classes/ece5997/fir-tutorial/ .
> ls

- **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 –reset fir.prj

# 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

# export design
export_design

# exit Vivado HLS
exit

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

```
> vivado_hls -f run.tcl
```

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.

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

INFO: [HLS 200-10]---------------------------------------------------------------
INFO: [HLS 200-10] -- Scheduling module 'fir'
INFO: [HLS 200-10]---------------------------------------------------------------
INFO: [HLS 200-10]---------------------------------------------------------------
INFO: [HLS 200-10] -- Exploring micro-architecture for module 'fir'
INFO: [HLS 200-10]---------------------------------------------------------------
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 ***
Synthesis Directory Structure

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

Latency: 7948 cycles
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];
        }
    }
}
Architecture 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
    ...
}

Latency: 897 cycles
Microarchitecture after Partitioning

Unrolled

Unrolled + Partitioned
void fir(int input[SIZE], int output[SIZE]) {
    ...

    // loop through each output
    for (int i = 0; i < SIZE; i++) {
        // pipeline the entire outer loop
        #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

[Figure credit] Ryan Kastner et. al., Parallel Programming for FPGAs, page 44.
Next Lecture

- Specialized computing