L5: HLS Overview

Cong (Callie) Hao
callie.hao@ece.gatech.edu

Assistant Professor
ECE, Georgia Institute of Technology

Sharc-lab @ Georgia Tech https://sharclab.ece.gatech.edu/

Acknowledgement: some contents are borrowed from Prof. Zhiru Zhang at Cornell University
High-Level Synthesis (HLS)

What is HLS?

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

Software Specification and Program

High-Level Synthesis

Logic Synthesis

Physical Synthesis

Circuit (ASIC, FPGA) Design

- **HLS Tools**

  
  ```
  for (i=1; i<=c;)
  a = a++;
  b = x*2-a;
  a = y+b/3;
  ```

  
  **C / C++, Chisel, ...**

  **HLS Tools**

  Verilog, VHDL, ...
What is HLS?

Behavioral-level: Expressive and concise

Register-Transfer-Level (RTL):

```vhdl
for(int h = 0; h < H; h++)
    for(int w = 0; w < W; w++)
        for(int m = 0; m < K; m++)
            for(int n = 0; n < K; n++)
...```

```vhdl
44 req0 <= 0;
45 repeat (1) @ (posedge clk);
46 #10 $finish;
47 end
48
49 // Connect the DUT
50 arbiter U (
51 clk,
52 rst,
53 ...
54 repeat (1) @ (posedge clk);
55 req0 <= 1;
56 req1 <= 1;
57 repeat (1) @ (posedge clk);
58 req2 <= 1;
59 req1 <= 0;
60 repeat (1) @ (posedge clk);
61 req3 <= 1;
62 ...
63 // Clock generator
64 always #1 clk=~clk;
65 initial begin
66 $dumpfile ("arbiter.vcd");
67 $dumpvars();
68 clk = 0;
69 rst = 1;
70 ...
71 `include "xxx.v"
2 module top ();
3 ...
4 reg clk;
5 reg rst;
6 reg req3;
7 reg req2;
8 reg req1;
9 reg req0;
10 wire gnt3;
11 wire gnt2;
12 wire gnt1;
13 wire gnt0;
```
High-Level Synthesis (HLS)

Why HLS (Design at Higher Level)?

- **Productivity**
  - Lower design complexity and faster simulation speed
  - Ease-of-use: C/C++/Python v.s. Verilog

- **Portability**
  - Single source -> multiple implementations (devices)

- **Permutability**
  - **Much more optimization** opportunities at higher level
  - Rapid design space exploration -> higher quality of result (QoR)

- **Bonus:**
  - Promote device usage
  - Significant code size reduction
    - Shorter simulation/verification cycle
    - Quick / early design iterations
Why HLS (Design at Higher Level)?

High-Level Synthesis
- C/C++
- System level
- Behavior level
- RTL (Verilog)
- Register-transfer level
- Logic level
- Netlist
- Logic Synthesis
- Physical Synthesis
- Transistor level
- Layout level

Code Size
- System level
- Behavior level
- RTL (Verilog)
- Register-transfer level
- Logic level
- Netlist
- Logic Synthesis
- Physical Synthesis
- Transistor level
- Layout level

Simulation Speed
- System level
- Behavior level
- RTL (Verilog)
- Register-transfer level
- Logic level
- Netlist
- Logic Synthesis
- Physical Synthesis
- Transistor level
- Layout level

Performance Improvement
- Minutes (~ hours)
- Hours (~ days)
- Days (~ weeks)

Code Size
- 300K Line
- 40K Line
- 1M Gate
- ?? Transis.

Simulation Speed
- 100 ~ 1KHz
- ~ 1MHz
- 10K~100KHz

Performance Improvement
- 10 ~ 20X
- 2 ~ 5X
- 20 ~ 50%

[source: Wakabayashi, DAC’05 tutorial]
High-Level Synthesis (HLS)


- How to design (a better) HLS tool
- How to use HLS tool
Where does performance gain come from? **Specialization**!

- **Data type specialization**
  - Arbitrary-precision fixed-point, custom floating-point
- **Interface/communication specialization**
  - Streaming, memory-mapped I/O, etc.
- **Memory specialization**
  - Array partitioning, data reuse, etc.
- **Compute specialization**
  - Unrolling, pipelining, dataflow, multithreading, etc.
- **Architecture specialization**
  - Pipelined, recursive, hybrid, etc.

1. **System-level**
   - Architecture, interface

2. **Module-level**
   - Compute, memory

3. **Bit-level**
   - Data type
**HLS Pragmas**

- All about “pragma”s: instructions to tell your compiler how to build the hardware
- This link has all the pragmas you need:

<table>
<thead>
<tr>
<th>Category</th>
<th>Pragmas</th>
</tr>
</thead>
</table>
| Kernel Optimization    | • pragma HLS aggregate  
                         • pragma HLS bind_op  
                         • pragma HLS bind_storage  
                         • pragma HLS expression_balance  
                         • pragma HLS latency  
                         • pragma HLS reset  
                         • pragma HLS top |
| Loop Unrolling         | • pragma HLS unroll  
                         • pragma HLS dependence |
| Loop Optimization      | • pragma HLS loop_flatten  
                         • pragma HLS loop_merge  
                         • pragma HLS loop_tripcount |
| Array Optimization     | • pragma HLS array_partition  
                         • pragma HLS array_reshape |
| Structure Packing      | • pragma HLS aggregate  
                         • pragma HLS dataflow |
| Resource Optimization  | • pragma HLS allocation  
                         • pragma HLS function_instantiate |
| Function Inlining      | • pragma HLS inline |
| Interface Synthesis    | • pragma HLS interface |
| Task-level Pipeline    | • pragma HLS dataflow  
                         • pragma HLS stream |
| Pipeline               | • pragma HLS pipeline  
                         • pragma HLS occurrence |
More HLS Resources

• This link has everything you need to know about HLS...
Hardware Specialization with HLS

• Hardware is structured, hierarchical, and deterministic at compile time

• So are Verilog and HLS
Hierarchical HDL structures are achieved by defining modules (definition) and instantiating modules (instance).

- Instantiation is the process of “calling” a module.

```
module TOP ( port_list );
    ALU U1 ( port_connection );
    MEM U2 ( port_connection );
endmodule

module ALU ( port_list );
    FIFO S1 ( port_connection );
endmodule
```
Module Ports

- Module header starts with module keyword, contains the I/O ports
- Port declarations begins with **output**, **input** or **inout** follow by bus indices
  - Provide the interface by which a module can communicate with the environment

```verilog
module S1 (a, b, c, d, e);
input [1:0] a, b;
input c;
output reg [1:0] d;
output e;
//Verilog 1995 Style
dendmodule

module S2 (input [1:0] a, b,
           input c,
           output reg [1:0] d,
           output e);
//ANSI C Style
dendmodule
```

(a[1:0] → d[1:0])

(b[1:0] → e)

(c → e)
### Basic Mapping Rule from C/C++ to RTL

<table>
<thead>
<tr>
<th>C Constructs</th>
<th>RTL Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Modules</td>
</tr>
<tr>
<td>Arguments</td>
<td>I/O Ports</td>
</tr>
<tr>
<td>Operators (+, *)</td>
<td>Functional units (adder, multiplier)</td>
</tr>
<tr>
<td>Scalars</td>
<td>Wires or registers</td>
</tr>
<tr>
<td>Arrays</td>
<td>Memory</td>
</tr>
<tr>
<td>Control flows</td>
<td>Control logics (Finite State Machine)</td>
</tr>
</tbody>
</table>
Basic Mapping Rule from C/C++ to RTL

<table>
<thead>
<tr>
<th>C Constructs</th>
<th>RTL Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Modules</td>
</tr>
<tr>
<td>Arguments</td>
<td>I/O Ports</td>
</tr>
<tr>
<td>Operators (+, *)</td>
<td>Functional units (adder, multiplier)</td>
</tr>
<tr>
<td>Scalars</td>
<td>Wires or registers</td>
</tr>
<tr>
<td>Arrays</td>
<td>Memory</td>
</tr>
<tr>
<td>Control flows</td>
<td>Control logics (Finite State Machine)</td>
</tr>
</tbody>
</table>

**C Source Code**

```c
void Foo_C() {...}
void Foo_A() {...}
void Foo_B() {
792
```Resource sharing: only one instance of Foo_B on hardware

**RTL Hierarchy**

- `top`
- `Foo_A`
- `Foo_B`
- `Foo_C`
Basic Mapping Rule from C/C++ to RTL

### C Constructs
- Functions → Modules
- Arguments
  - Operators (+, *) → Functional units (adder, multiplier)
  - Scalars → Wires or registers
  - Arrays → Memory
  - Control flows → Control logics (Finite State Machine)

### RTL Components
- Modules
- I/O Ports

### C Source Code
```c
void top(int* in1, int* in2, int* out) {
    *out = *in1 + *in2;
}
```
Basic Mapping Rule from C/C++ to RTL

**C Constructs**
- Functions → Modules
- Arguments → I/O Ports
- Operators (+, *) → Functional units (adder, multiplier)
- Scalars → Wires or registers
- Arrays → Memory
- Control flows → Control logics (Finite State Machine)

**C Source Code**
```c
for (i = 0; i < N; i++)
```

**Diagram**
- C Source Code
- RAM
  - A[N-1]
  - A[N-2]
  - ...
  - A[1]
  - A[0]
Determined at Compile Time

- On FPGA, memory maps to BRAM
- Everything must be decided at compile time – your hardware cannot be changed while running!
  - Adding one more piece of memory after the circuit is built?

```c
int mem[var];
int mem* = malloc(var * sizeof(int));
```

```vhdl
reg [0:7] mem [var:0];
```

8-bit element
8-bit element
8-bit element
8-bit element

---

How many??
Hardware Specialization with HLS

- Where does performance gain come from? **Specialization**!

- **Data type** specialization
  - Arbitrary-precision fixed-point, custom floating-point

- Interface/communication specialization

- Memory specialization
  - Array partitioning, data reuse, etc.

- Compute specialization
  - Unrolling, pipelining, dataflow, multithreading, etc.

- Architecture specialization
  - Pipelined, recursive, hybrid, etc.

- **Discuss data type next week**
- **Remember**: FPGA doesn’t like floating point!
  Use integer at least :D

1. System-level
   - Architecture, interface

2. Module-level
   - Compute, memory

3. Bit-level
   - Data type

---

Callie Hao | Sharclab @ Georgia Institute of Technology  
https://sharclab.ece.gatech.edu/
Hardware Specialization with HLS

- Where does performance gain come from? **Specialization**!
  - **Data type specialization**
    - Arbitrary-precision fixed-point, custom floating-point
  - **Interface/communication specialization**
    - Streaming, memory-mapped I/O, etc.
  - **Memory specialization**
    - Array partitioning, data reuse, etc.
  - **Compute specialization**
    - Unrolling, pipelining, dataflow, multithreading, etc.
  - **Architecture specialization**
    - Pipelined, recursive, hybrid, etc.

The Three Musketeers
(i) Array partition
(ii) Loop unroll
(iii) Loop pipeline
Initially, an array is mapped to one (or more) block(s) of RAM (or BRAM on FPGA)
- One block of RAM has at most two ports
- At most two read/write operations can be done in one clock cycle – Parallelism is 2 (too low)
- An array can be partitioned and mapped to multiple blocks of RAMs

Array Partition – Memory Parallelism

1 RAM block

4 RAM blocks can be accessed simultaneously!
Array Partition – Memory Parallelism

- Initially, an array is mapped to one (or more) block(s) of RAM (or BRAM on FPGA)
  - One block of RAM has at most **two ports**
  - At most **two** read/write operations can be done in one clock cycle – Parallelism is **2** (too low)
- An array can be **partitioned and mapped to multiple** blocks of RAMs
  - Can also be partitioned into individual elements and mapped to registers
    - Only if your array is small otherwise the tool will give up

|------|------|------|------|

4 RAM blocks can be accessed simultaneously!

OR

|------|------|------|

OR

|------|------|

OR

|------|------|

All registers
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

**Original Loop**

```c
for (int i = 0; i < N; i++)
    #pragma HLS unroll
    A[i] = B[i] + C[i];
```

**Unrolled Loop**

```c
A[0] = B[0] + C[0];
...
```

**N x m cycles**

*Assume A[i] = B[i] + C[i] takes m cycle*

**m cycle**

*Only if A, B, and C are fully partitioned!*
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.

For (i = 0; i < N; ++i)

#pragma HLS pipeline

\[ p[i] = x[i] \times y[i]; \]
The three techniques are frequently used together to boost computation efficiency.

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        A[i][j] = B[i][j] * C[i][j];
    }
}
```
The three techniques are frequently used together to boost computation efficiency

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        #pragma HLS unroll
        A[i][j] = B[i][j] * C[i][j];
    }
}
```

Memory ports limited by 2 → Need to partition
The three techniques are frequently used together to boost computation efficiency.

```c
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        #pragma HLS unroll
        A[i][j] = B[i][j] * C[i][j];
    }
}
```
The three techniques are frequently used together to boost computation efficiency. Put-together: Pipeline + Unroll + Partition

```c
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        #pragma HLS unroll
        A[i][j] = B[i][j] * C[i][j];
    }
}
```
• The three techniques are frequently used together to boost computation efficiency

```c
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {
    #pragma HLS pipeline II=1
    for (int j = 0; j < 32; j++) {
        #pragma HLS unroll factor=8
        A[i][j] = B[i][j] * C[i][j];
    }
}
```
Hardware Specialization with HLS

• Where does performance gain come from? **Specialization**!

  - Data type specialization
    o Arbitrary-precision fixed-point, custom floating-point
  - Interface/communication specialization
    o Streaming, memory-mapped I/O, etc.
  - Memory specialization
    o Array partitioning, data reuse, etc.
  - Compute specialization
    o Unrolling, pipelining, dataflow, multithreading, etc.
  - **Architecture** specialization
    o Pipelined, recursive, hybrid, etc.

1. **System-level**
   • Architecture, interface

2. **Module-level**
   • Compute, memory

3. **Bit-level**
   • Data type

Talk more in the future
• HLS is good...!
• HLS is all about pragmas
• Optimization starting point:
  o Memory partition + loop unrolling + loop pipelining
• Next lecture:
  o More about loop optimization