18-643 Lecture 8: C-function-to-IP HLS (Vitis HLS IP-Flow)

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: learn how to tell Vitis what you want and understand what Vitis tells you back

• Notices
  – Handout #5: lab 2, due noon, 10/11
  – Project status report due each Friday

• Readings (see lecture schedule online)
  – Ch 15, The Zynq Book (skim Ch 14)
  – for lab2, C. Zhang, et al., ISFPGA, 2015
Function-to-IP, not Program-to-HW

• **Object of design is an IP or kernel module**

• Designer still in charge (garbage in, garbage out)
  – specify functionality as algorithm (in C)
  – specify structure as pragmas (beyond C)
  – set optimization constraints (beyond C)

  Offload bit- and cycle-level design/opt. to tools

• Vitis HLS (formerly AutoESL; formerly UCLA)
  – never mind all of C (what’s main( )? what malloc?)
  – never mind all usages of allowed subset (all loops okay, but static ones actually work well)
  – what else beyond C might a HW designer need (types, interface, structural hints)
What does Vitis HLS see?

```c
int fibi(int n) {
    int last=1; int lastlast=0; int temp;

    if (n==0) return 0;
    if (n==1) return 1;

    for(;n>1;n--) {
        temp=last+lastlast;
        lastlast=last;
        last=temp;
    }

    return temp;
}
```
Function to IP Block

What if I want multiple outputs?

```c
int fibi(int n) {
    . . .
    return . . .;
}
```
AP_CTRL_HS Block Protocol

- **ap_clk**: Ap_clk signal
- **ap_rst**: Ap_RST signal
- **ap_start**: Ap_START signal
- **ap_idle**: Ap_IDLE signal
- **ap_ready**: Ap_READY signal
- **ap_done**: Ap_DONE signal

Invoke, inputs used, output valid, ready for new ap_start
Function Invocation: Latency vs Throughput

- latency
- minimum initiation interval

- start \(\leftarrow\) ready \(\rightarrow\) done

- start \(\leftarrow\) ready \(\rightarrow\) done

- start \(\leftarrow\) ready \(\rightarrow\) done
Other Block Control Options

• **ap_ctrl_chain**
  - separate input producer and output consumer
  - **ap_continue**: driven by the consumer to backpressure the block and producer
  - IF a block reaches “done” AND **ap_continue** is deasserted, the block will hold **ap_done** and keep output valid until **ap_continue** is asserted

• AXI compatible memory-mapped control
  - software on ARM interacts with the block using fxn-call-like interfaces (input, output, start, etc.)
  - IP-specific .h and routines generated automatically
Scalar I/O Port Timing

- By default (ap_none)
  - input ports should be stable between \texttt{ap\_start} and \texttt{ap\_ready}
  - output port is valid when \texttt{ap\_done}
- 3 asynchronous handshake options on input
  - \texttt{ap\_vld} only: consumes only if input valid
  - \texttt{ap\_ack} only: signals back when input consumed
  - \texttt{ap\_hs}: \texttt{ap\_vld} + \texttt{ap\_ack}
- HLS’s job to follow protocol
Pass-by-Reference Arguments

```c
void fibi(int *n, int *fib) {
    int last=1; int lastlast=0; int temp;
    int nn=*n;

    if (nn==0) { *fib=0; *n=0; return; }
    if (nn==1) { *fib=1; *n=0; return; }
    for(;nn>1;nn--) {
        temp=last+lastlast;
        lastlast=last;
        last=temp;
    }

    *fib=last; *n=lastlast;
}
```
Don’t look inside yet

They are not really “pointers”
• do not evaluate *(fib+1) or fib
• except to pretend to be a fifo

```c
void fibi(int *n, int *fib) {
    . . . .
    *n and *fib assigned to;
    *n in RHS before assigned;
    *fib in RHS after assigned;
    . . . .
}
```
## All I/O Options

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Interface Mode</th>
<th>Scalar</th>
<th></th>
<th>Array</th>
<th></th>
<th>Pointer or Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input</td>
<td>Return</td>
<td>I</td>
<td>I/O</td>
<td>O</td>
<td>I</td>
</tr>
<tr>
<td>ap_ctrl_none</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ctrl_hs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ctrl_chain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>axis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_axi_lite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m_axi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_none</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_stable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_vld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ovld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_hs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_memory</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>bram</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_fifo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_bus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Supported**
- **D = Default Interface**
- **Not Supported**

*Fig 1-49, Vivado Design Suite User Guide: High-Level Synthesis*

 Pointer I/O can output before function ends and multiple times
Array Arguments

```c
#define N (1<<10)
void D2XPY (double Y[N], double X[N]) {
    for(i=0; i<N; i++) {
        Y[i]=2*X[i]+Y[i];
    }
}
```

*could ask to use separate read and write ports*
Array Arg Options

• By default, array args become BRAM ports
  – array must be fixed size
  – can use 2 ports for bandwidth or split read/write
• If array arg is accessed **always consecutively** AND only either either read or written
  – can become **ap_fifo** port
  – i.e., no address wires, just push or pop
• Array args can also become AXI or a generic bus master ports

Scheduler handles port sharing and dynamic delays
Time to Look Inside
void mmm(char A[N][N], char B[N][N], short C[N][N]) {
    Row: for (int i=0; i<N; i++) {
        Col: for (int j=0; j<N; j++) {
            C[i][j] = 0;
            Product: for (int k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

keep it simple
If you want to be literal

\[
C[i][j] += A[i][k] \times B[k][j]
\]

for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++) {
            \text{RAW hazard?}
        }
    }
}

18-643-F21-L8-S17, James C. Hoe, CMU/ECE/CALCM, ©2021
If you want to be literal

for 5x5 matrices
If you want to be literal (continued)

```c
for(int k=0; k<N; k++) {
    C[i][j] += A[i][k]*B[k][j];
}
```

[Operation Control Step]

[Vitis HLS 2020.2]
Let’s Try Pipelining

for 5x5 matrices
Structural Pragma: Pipelining

- Find minimum “iteration interval (II)” schedule
  - II >= num stages a resource instance is used
  - II >= RAW hazard distance
- E.g., to pipeline \( C[i][j] += A[i][k] * B[k][j]; \)

RAW hazard, II >= 3

structural conflict, II >= 2 (II >= 1 if 2-port)
What Vitis HLS tells you . . .

<table>
<thead>
<tr>
<th>Modules &amp; Loops</th>
<th>Issue Type</th>
<th>Slack</th>
<th>Latency(cycles)</th>
<th>Latency(ns)</th>
<th>Iteration Latency</th>
<th>Interval</th>
<th>Trip Count</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_mult</td>
<td>II Violation</td>
<td>-</td>
<td>376</td>
<td>3.760E3</td>
<td>-</td>
<td>377</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Row_Col</td>
<td>II Violation</td>
<td>-</td>
<td>375</td>
<td>3.750E3</td>
<td>15</td>
<td>-</td>
<td>25</td>
<td>yes</td>
</tr>
<tr>
<td>Product</td>
<td>II Violation</td>
<td>-</td>
<td>12</td>
<td>120.000</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>yes</td>
</tr>
</tbody>
</table>

The II Violation in module 'matrix_mult' (loop 'Product'): Unable to enforce a carried dependence constrair operation ('prod_addr_write_in16',../matrix_mult.cpp:16) of variable 'add_in16',../matrix_mult.cpp:16 on array 'prod'.

Pipelining result: Target II = 1, Final II = 2, Depth = 5, loop 'Product'
Removing “volatile”

for 5x5 matrices
Inferring Accumulation Register

int temp;
for(int cnt=0; cnt<125; cnt++) {
    int i=cnt/25, j=(cnt/5)%5, k=cnt%5;
    if (k==0) temp=C[i][j];
    temp += A[i][k]*B[k][j];
    if (k==4) C[i][j]=temp;
}

[Image of operation control step table and diagram with annotations:
- **read A[i][k]**
- **read B[k][j]**
- **mult**
- **accumulate**
- **add**
- **only single write to C[i][j]**]
Letting Vitis HLS just do its own thing

```cpp
#include "matrix_mult.h"

void matrix_mult(
    mat_a a[IN_A_ROWS][IN_A_COLS],
    mat_b b[IN_B_ROWS][IN_B_COLS],
    mat_prod prod[IN_A_ROWS][IN_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < IN_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < IN_B_COLS; j++) {
            prod[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < IN_B_ROWS; k++) {
                prod[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

for 5x5 matrices
Optimizing Memory Layout
- 25x speedup off same code
- No new functional bugs!!
- Can you do better by RTL?
Pragma Crib Sheet: Loops

- **Loop Unroll (full and partial)**
  - amortize loop control overhead
  - increase loop-body size, hence “ILP” and scheduling flexibility

- **Loop Flatten**
  - streamline loop-nest control
  - reduce start/finish stutter

- **Loop Merge**
  - combine loop-bodies of independent loops of same control
  - improve parallelism and scheduling
Pragma Crib Sheet: Arrays

• Map
  – multiple arrays in same BRAM
  – no perf loss if no scheduling conflicts

• Reshape
  – change BRAM aspect ratio to widen ports
  – change linear address to location mapping
  – higher bandwidth on consecutive locations

• Partition
  – map 1 array to multiple BRAMs
  – multiple independent ports if no bank conflicts

A lot more you can control; must read UG902
Additional Control thru Code Structure

\[
\text{for } (k = \ldots)
\quad \text{for } (i = \ldots)
\quad \text{for } (j = \ldots)
\quad C[i][j] += f(i,j,k)
\]

\[
\text{pipelined kernel}
\]

\[
\text{unrolled inner loops}
\]

\[
\text{for } (i = \ldots)
\quad \text{for } (j = \ldots)
\quad C[i][j] += f(i,j,0) + f(i,j,1) + \ldots + f(i,j,k-1)
\]
## Don’t Forget HW Basics

- **Literal (forced)**
  - 686cyc, 6860ns (1x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00</td>
<td>1.878</td>
<td>2.70</td>
<td>0</td>
<td>1</td>
<td>39</td>
<td>214</td>
</tr>
</tbody>
</table>

- **Vitis Default**
  - 80cyc, 800ns (8.6x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00</td>
<td>3.772</td>
<td>2.70</td>
<td>0</td>
<td>3</td>
<td>128</td>
<td>467</td>
</tr>
</tbody>
</table>

- **Pragma Directed**
  - 28cyc, 280ns (25x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00</td>
<td>4.195</td>
<td>2.70</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>190</td>
</tr>
</tbody>
</table>

- **Clock go fast . . .**
  - 29cyc, 116ns (60x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>2.170</td>
<td>1.08</td>
<td>0</td>
<td>0</td>
<td>132</td>
<td>231</td>
</tr>
</tbody>
</table>

29cyc, 87ns (79x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00</td>
<td>2.170</td>
<td>0.81</td>
<td>0</td>
<td>0</td>
<td>250</td>
<td>295</td>
</tr>
</tbody>
</table>

57cyc, 114ns (60x)

<table>
<thead>
<tr>
<th>Target (ns)</th>
<th>Estimated (ns)</th>
<th>Uncertainty (ns)</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>1.287</td>
<td>0.54</td>
<td>0</td>
<td>1</td>
<td>211</td>
<td>263</td>
</tr>
</tbody>
</table>
Design by Exploration

When this takes only minutes, a little trial-and-error is okay (just a little!!!!)
Putting it in context (from last time)

• For you to produce “good” structural RTL
  – identify suitable “temporal and spatial pattern”
  – flesh out concrete datapath (bit/cycle exact)
  – develop correct and efficient control sequencing

• C-to-HW (i.e., C-to-RTL) compiler bridges the gap between functionality and implementation
  – extract parallelism from a sequential specification
  – fill in the details below the functional abstraction
  – make good decisions when filling in the details

    Vitis HLS does its part (under your direction) fast and without mistakes
Parting Thoughts

- Vitis HLS doesn’t turn program into HW
- Vitis HLS doesn’t turn programmer into HW designer
- Multifaceted benefits to HW designer
  - algo. development/debug/validate in SW
  - pragma steering (no RTL hacking, machine tuning)
  - fast analysis and visualization
  - data type support
  
  it is about more than adding “double” to Verilog

- built-in, stylized IP interfaces
  
  integration with the rest of Vitis and Zynq!!

Can we turn HPC programmers into HW designers?