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, 10/10**
  – 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
Vitis C-to-RTL HLS

- Function-to-IP, not Program-to-HW
  - 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)

You can use it as a better RTL

- 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
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

Later—what if you want 2 outputs?

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

- **ap_clk**: Clock signal.
- **ap_rst**: Reset signal.
- **ap_start**: Start signal.
- **ap_idle**: Idle signal.
- **ap_ready**: Ready signal.
- **ap_done**: Done signal.

**Invoke**: Starting a new process.

**Inputs used**: Signals used for processing.

**Output valid**: Output is valid.

**Ready for new ap_start**: Ready to start a new process.
Function Invocation: Latency vs Throughput

- start
- ready
- done

Minimum initiation interval

Latency
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 ap_start and ap_ready
  – output port is valid when ap_done

• 3 asynchronous handshake options on input
  – ap_vld only: consumes only if input valid
  – ap_ack only: signals back when input consumed
  – ap_hs: ap_vld + 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;
}
```
Pass-by-Pointer & Reference

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></td>
<td>Input</td>
<td>Return</td>
<td>I</td>
<td>I/O</td>
<td>O</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</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></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></td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</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**: Dark gray
- **D = Default Interface**: Red
- **Not Supported**: Light gray

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

- Pointer I/O can output before fxn ends and multiple times.
Array Arguments

#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 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

Diagram showing connections:
- n to fibi
- ap_clk to ap_ready, ap_done, ap_idle
- ap_rst to ap_done, ap_idle
- ap_start to ap_done, ap_idle
MMM (yet again)

```c
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];
            }
        }
    }
}
```
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++) {
            \( C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j] \)
        }
    }
}

*RAW hazard?*

\[ iN+j \quad \text{BRAM} \]
\[ iN+k \quad \text{BRAM} \]
\[ kN+j \quad \text{BRAM} \]

\[ \text{write_enable} \]

\[ \text{addr} \]
\[ \text{data} \]

\[ \text{addr} \]
\[ \text{data} \]

\[ \text{addr} \]
\[ \text{data} \]

FSM

\[ i \]
\[ j \]
\[ k \]
If you want to be literal for 5x5 matrices
If you want to be literal (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Control Step</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>k(phi_mux)</td>
<td></td>
</tr>
<tr>
<td>add_ln15(+)</td>
<td></td>
</tr>
<tr>
<td>zext_ln16_1(zext)</td>
<td></td>
</tr>
<tr>
<td>add_ln16_2(+)</td>
<td></td>
</tr>
<tr>
<td>zext_ln16_2(zext)</td>
<td></td>
</tr>
<tr>
<td>a_addr(getelementptr)</td>
<td></td>
</tr>
<tr>
<td>tmp_1(bitconcatenate)</td>
<td></td>
</tr>
<tr>
<td>add_ln16_3(+)</td>
<td></td>
</tr>
<tr>
<td>add_ln16_4(+)</td>
<td></td>
</tr>
<tr>
<td>zext_ln16_3(zext)</td>
<td></td>
</tr>
<tr>
<td>b_addr(getelementptr)</td>
<td></td>
</tr>
<tr>
<td>icmp_ln15(icmp)</td>
<td></td>
</tr>
<tr>
<td>br_ln15(br)</td>
<td></td>
</tr>
<tr>
<td>a_load(read)</td>
<td></td>
</tr>
<tr>
<td>b_load(read)</td>
<td></td>
</tr>
<tr>
<td>sext_ln16(sext)</td>
<td></td>
</tr>
<tr>
<td>sext_ln16_1(sext)</td>
<td></td>
</tr>
<tr>
<td>mul_ln16(*)</td>
<td></td>
</tr>
<tr>
<td>prod_load(read)</td>
<td></td>
</tr>
<tr>
<td>add_ln16(+)</td>
<td></td>
</tr>
<tr>
<td>prod_addr_write_ln16(write)</td>
<td></td>
</tr>
<tr>
<td>br_ln0(br)</td>
<td></td>
</tr>
</tbody>
</table>

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

If you want to be literal (continued)
Let’s Try Pipelining

```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],
    volatile 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
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>=2 (w. write forward)

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

- **matrix_mult** (II Violation)
  - Slack: -
  - Latency (cycles): 376
  - Latency (ns): 3.760E3
  - Iteration Latency: -
  - Interval: 377
  - Trip Count: -
  - Pipelined: no

- **Row.Col** (II Violation)
  - Slack: -
  - Latency (cycles): 375
  - Latency (ns): 3.750E3
  - Iteration Latency: 15
  - Interval: -
  - Trip Count: 25
  - Pipelined: no

- **Product** (II Violation)
  - Slack: -
  - Latency (cycles): 12
  - Latency (ns): 120.000
  - Iteration Latency: 5
  - Interval: 2
  - Trip Count: 5
  - Pipelined: yes

**Schedule**

- The II Violation in module 'matrix_mult' (loop 'Product'): Unable to enforce a carried dependence constrair operation ('prod_addr_write_Ln16', ../matrix_mult.cpp:16) of variable 'add_Ln16', ../matrix_mult.cpp:16 on array 'prod'.
- Pipelining result: Target II = 1, Final II = 2, Depth = 5, loop 'Product'

**Runtime**

- **Product**
  - prod_load(read)
  - reuse_select(select)
  - add_Ln16(+)
  - prod_addr_write_Ln16(write)
Removing “volatile”

for 5x5 matrices
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;
}
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_AROWS][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_COLS; k++) {
                prod[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

for 5x5 matrices

<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>80</td>
<td>800.000</td>
<td>-</td>
<td>81</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Row_Col</td>
<td>II Violation</td>
<td>-</td>
<td>78</td>
<td>780.000</td>
<td>7</td>
<td>-</td>
<td>3</td>
<td>yes</td>
</tr>
</tbody>
</table>
Optimizing Memory Layout

```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];
            }
        }
    }
}
```
- 25x speedup off same code
- No new functional bugs!!
- Can you do better by RTL?

[Vitis HLS 2020.2]
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
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
Additional Control thru Code Structure

for (i=...)
  for (j=...)
    for (k=...)
      C[i][j] += f(i, j, k)

unrolled inner loops

for (i=...)
  for (j=...)
    C[i][j] += f(i, j, 0) + f(i, j, 1) + ... + f(i, j, k-1)

pipelined kernel

a[i][k] + b[k][j] + c[i][j] → c[i][j]
Don’t Forget HW Basics

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

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

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

- **Clock go fast . . .**
  - 29cyc, 116ns (60x)
  - 29cyc, 87ns (79x)
  - 57cyc, 114ns (60x)
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?