18-643 Lecture 10: Vivado C-to-IP HLS

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: learn how to tell Vivado HLS what you really want and understand what Vivado HLS is telling you

• Notices
  – Handout #4: lab 2, due noon, 10/6
  – 3.5 weeks to project proposal

• Readings
  – Ch 15, The Zynq Book (skim Ch 14)
Tortoise and Hare

• Tortoise
  – delivers exact optimal implementation to a fully specified objective (functional + tuning)
  – perfection takes time
    say last 10% of quality takes up 90% of the time

• Hare
  – only gets to 90% quality
  – delivers the design 10 times faster

This hare doesn’t take a nap after one design . . .
The Design Race

- Good Enough Box
- 90% best possible
- educated guess
- hey, it works
- 1/perf

out-of-time
Why the Hare Wins

• In real design projects
  – don’t always know exact target initially
  – can’t land first shot on target anyway
  – good enough really is good enough
  – hitting schedule is everything

    show at COMDEX in Nov or bust in Dec

• There are a lot more rabbits than turtles in this world; there are not enough turtles in this world

    Even more turkeys . . . but that’s a different class

All characters appearing in this story are fictitious. Any resemblance to real persons, living or dead, is purely coincidental.
Vivado HLS
**Function-to-IP, not Program-to-HW**

- **Object of design is an IP 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**

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

1. Inputs consumed.
2. Output valid.
3. Ready for new ap_start.
Function Invocation: Latency vs Throughput

- start
- ready
- done

Latency: minimum initiation interval

start ➔ ready ➔ done

start ➔ ready ➔ done

start ➔ ready ➔ done

18-643-F17-L10-S11, James C. Hoe, CMU/ECE/CALCM, ©2017
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 port interfaces
  - 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

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;
}
void fibi(int *n, int *fib) {
    . . . .
    *n in RHS* and LHS;
    *fib in LHS only
    . . . .
} *used before assigned

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

Don’t look inside yet

n_i → fib
fib → n_o
ap_clk → ap_ready
ap_rst → ap_done
ap_start → ap_idle
### All I/O Options

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

ap_clk ap_ready
ap_rst ap_done
ap_start ap_idle

fibi

n
void mmm(char A[N][N], char B[N][N], short C[N][N]) {
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            C[i][j]=0; 
            for(int k=0; k<N; k++) {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
StructuralPragma: Pipelining

• Fully elaborate scope (e.g., unroll loops)
• Find minimum “iteration interval (II)” schedule
  – II >= num stages a resource instance is used
  – II >= RAW hazard distance
• E.g., to pipeline \( C[i][j] += A[i][k] \times B[k][j] \);
HLS Analysis and Visualization

```c
// Zynq Book Tutorial 3, Sol#2
for(int i=0; i<5; i++) {
    for(int j=0; j<5; j++) {
        C[i][j]=0;
        for(int k=0; k<5; k++) {
            #pragma HLS PIPELINE
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

[Vivado HLS Screenshots]
Design by Trial and Error

```c
// Zynq Book Tutorial 3, Sol#3
for(int i=0; i<5; i++) {
    for(int j=0; j<5; j++) {
        C[i][j]=0;
        #pragma HLS PIPELINE
        for(int k=0; k<5; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

[Vivado HLS Screenshots]
Design by Trial and Error

```c
// Zynq Book Tutorial 3, Sol#4
#program HLS ARRAY_RESHAPE variable=A, dim=2
#program HLS ARRAY_RESHAPE variable=B, dim=1
for(int i=0; i<5; i++) {
    for(int j=0; j<5; j++) {
        C[i][j]=0;
        #pragma HLS PIPELINE
        for(int k=0; k<5; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

What if \( N >> 5 \)?

A and B reshaped to read entire row/column at a time?
Recall from Last Time

```cpp
for (i=...)
  for (j=...)
    for (k=...)
      GET C[i][j]
```

parallel kernel pipelines

```
for (k=...)
  for (i=...)
    for (j=...)
      GET C[i][j]
```

fully unrolled inner loops
From here we can play with pragmas to sensibly widen concurrency if needed.

```c
// assume C initialized to 0
for(int k=0; k<5; k++)
    for(int i=0; i<5; i++)
        for(int j=0; j<5; j++) {
            #pragma HLS PIPELINE
            C[i][j] += A[i][k]*B[k][j];
        }
```

Vivado HLS Screenshots

- can fix by disable flattening
With Algo. Rewrite (Option 2)

```c
for(int i=0; i<5; i++) {
    for(int j=0; j<5; j++) {
        short Ctemp=0;
        for(int k=0; k<5; k++)
            #pragma HLS PIPELINE
            Ctemp += A[i][k]*B[k][j];
        C[i][j]=Ctemp;
    }
}
```

HLS figured out forwarding

Vivado HLS Screenshots

can fix by disable flattening
Pragma Crib Sheet: Loops

- Loop Unroll (full and partial)
  - amortize loop control overhead
  - increase loop-body size, hence “ILP” and scheduling flexibility
- Loop Merge
  - combine loop-bodies of independent loops of same control
  - improve parallelism and scheduling
- Loop Flatten
  - streamline loop-nest control
  - reduce start/finish stutter
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
  - higher bandwidth on consecutive addresses

- **Partition**
  - map 1 array to multiple BRAMs
  - multiple independent ports if no bank conflicts

A lot more you can control; must read UG902
Design by Exploration

reference algorithm & testbench

algorithm for synthesis

co-simulation validation

HLS & analysis

good enough

yes

RTL

no

pragmas

not good enough after backend

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

• Why hardware design is hard
  – reason #1: low level abstraction
  – reason #2: unrestricted design freedom
  – reason #3: massive concurrency

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

Vivado does its part fast and without mistakes
Parting Thoughts

• Vivado doesn’t turn program into HW
• Vivado 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 Vivado and Zynq!!

• We are entering a new era for FPGAs
Vivado “Software-Defined” SoC

Screenshot, page 24, SDSoC Environment Getting Started (UG1028)