ESE532:
System-on-a-Chip Architecture

Day 20: April 3, 2017

Pipelining, Frequency, Dataflow

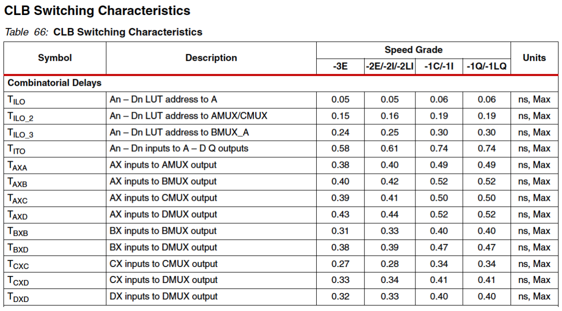
Today

- What drives cycle times
- Pipelining in Vivado HLS C
- Avoiding bottlenecks feeding data in Vivado HLS C

Message

- Should be able to run at high clock rates (e.g. 400—550MHz)
- Vivado HLS gives control over pipelining
- Code may need some care and stylization to feed data efficiently
- Read Vivado HLS Users Guide (902)
  - Methodology, Design Optimization

Clock Cycle



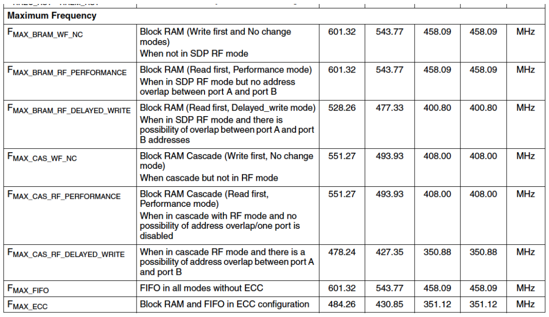
CLB Switching Characteristics

Table 66: **CLB Switching Characteristics**

Symbol	Description	Speed Grade				Units
		-3E	-2E/-2I/-2LI	-1C/-1I	-1Q/-1LQ	
Combinatorial Delays						
T_{ILO}	An – Dn LUT address to A	0.05	0.05	0.06	0.06	ns, Max
T_{ILO_2}	An – Dn LUT address to AMUX/CMUX	0.15	0.16	0.19	0.19	ns, Max
T_{ILO_3}	An – Dn LUT address to BMUX_A	0.24	0.25	0.30	0.30	ns, Max
T_{ITO}	An – Dn inputs to A – D Q outputs	0.58	0.61	0.74	0.74	ns, Max
T_{AXA}	AX inputs to AMUX output	0.38	0.40	0.49	0.49	ns, Max
T_{AXB}	AX inputs to BMUX output	0.40	0.42	0.52	0.52	ns, Max
T_{AXC}	AX inputs to CMUX output	0.39	0.41	0.50	0.50	ns, Max
T_{AXD}	AX inputs to DMUX output	0.43	0.44	0.52	0.52	ns, Max
T_{BXB}	BX inputs to BMUX output	0.31	0.33	0.40	0.40	ns, Max
T_{BXD}	BX inputs to DMUX output	0.38	0.39	0.47	0.47	ns, Max
T_{CXC}	CX inputs to CMUX output	0.27	0.28	0.34	0.34	ns, Max
T_{CXD}	CX inputs to DMUX output	0.33	0.34	0.41	0.41	ns, Max
T_{DXD}	DX inputs to DMUX output	0.32	0.33	0.40	0.40	ns, Max

Xilinx DS191 p. 54

BRAM



Maximum Frequency						
$F_{MAX\_BRAM\_WF\_NC}$	Block RAM (Write first and No change modes) When not in SDP RF mode	601.32	543.77	458.09	458.09	MHz
$F_{MAX\_BRAM\_RF\_PERFORMANCE}$	Block RAM (Read first, Performance mode) When in SDP RF mode but no address overlap between port A and port B	601.32	543.77	458.09	458.09	MHz
$F_{MAX\_BRAM\_RF\_DELAYED\_WRITE}$	Block RAM (Read first, Delayed_write mode) When in SDP RF mode and there is possibility of overlap between port A and port B addresses	528.26	477.33	400.80	400.80	MHz
$F_{MAX\_CAS\_WF\_NC}$	Block RAM Cascade (Write first, No change mode) When cascade but not in RF mode	551.27	493.93	408.00	408.00	MHz
$F_{MAX\_CAS\_RF\_PERFORMANCE}$	Block RAM Cascade (Read first, Performance mode) When in cascade with RF mode and no possibility of address overlap/one port is disabled	551.27	493.93	408.00	408.00	MHz
$F_{MAX\_CAS\_RF\_DELAYED\_WRITE}$	When in cascade RF mode and there is a possibility of address overlap between port A and port B	478.24	427.35	350.88	350.88	MHz
F_{MAX_FIFO}	FIFO in all modes without ECC	601.32	543.77	458.09	458.09	MHz
F_{MAX_ECC}	Block RAM and FIFO in ECC configuration	484.26	430.85	351.12	351.12	MHz

Xilinx DS191 p. 57

DSP

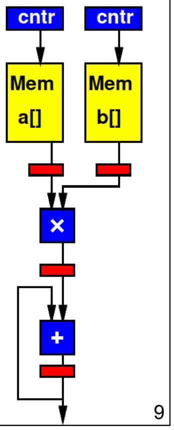
Maximum Frequency						
F_{MAX}	With all registers used	741.84	650.20	547.95	547.95	MHz
F_{MAX_PATDET}	With pattern detector	627.35	549.75	463.61	463.61	MHz
$F_{MAX\_MULT\_NOMREG}$	Two register multiply without MREG	412.20	360.75	303.77	303.77	MHz
$F_{MAX\_MULT\_NOMREG\_PATDET}$	Two register multiply without MREG with pattern detect	374.25	327.65	276.01	276.01	MHz
$F_{MAX\_PREADD\_MULT\_NOADREG}$	Without ADREG	468.82	408.66	342.70	342.70	MHz
$F_{MAX\_PREADD\_MULT\_NOADREG\_PATDET}$	Without ADREG with pattern detect	468.82	408.66	342.70	342.70	MHz
$F_{MAX\_NOPIPELINEREG}$	Without pipeline registers (MREG, ADREG)	306.84	267.81	225.02	225.02	MHz
$F_{MAX\_NOPIPELINEREG\_PATDET}$	Without pipeline registers (MREG, ADREG) with pattern detect	285.23	249.13	209.38	209.38	MHz

Zynq Primitives

- DSP block – run around 550MHz
- BRAM – run around 400—460MHz
- CLB – latency around 0.5ns
  - Carry cascade around 9ps

Preclass 1

- How fast can pipeline?



Acyclic Datapath

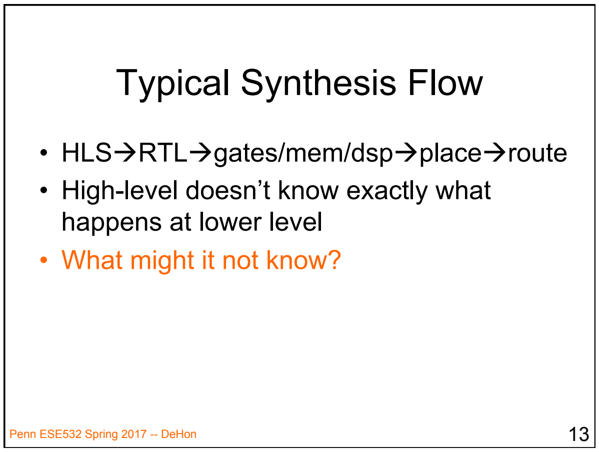
- With no feedback cycles
  - Should be able to pipeline datapath down to level of operators
    - Slowest operator
  - May need to add more pipeline levels
    - Takes more clocks to get from input to output

Interconnect

- If we pipeline operators ignoring interconnect delays,
  what can happen?

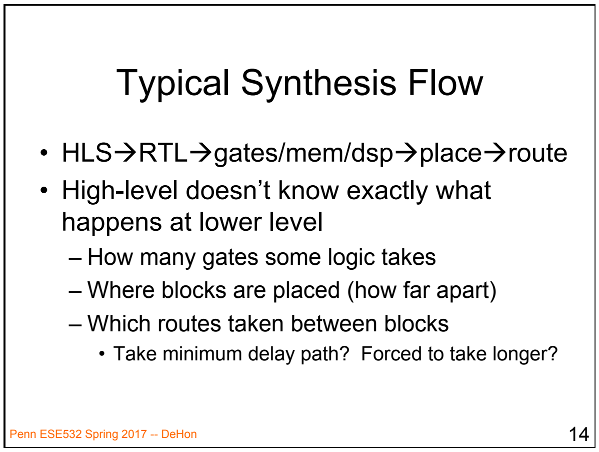
Interconnect Delays

- How would we fix?
- What does it cost us?



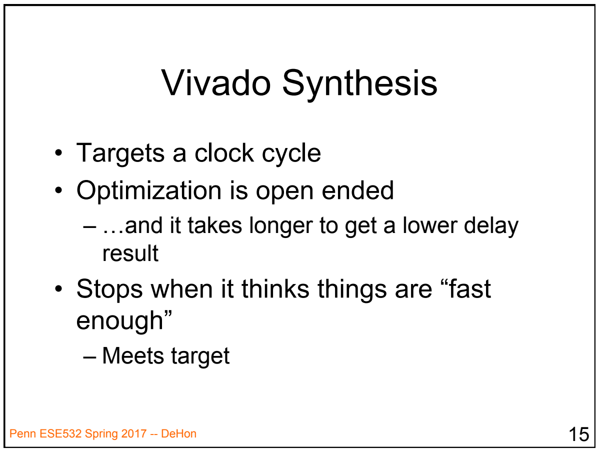
Typical Synthesis Flow

- HLS→RTL→gates/mem/dsp→place→route
- High-level doesn't know exactly what happens at lower level
- What might it not know?



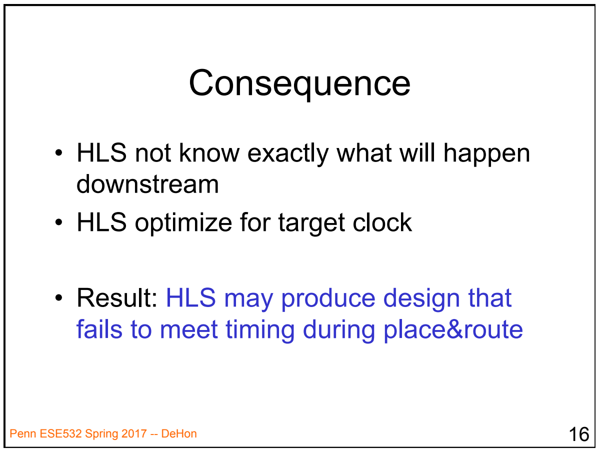
Typical Synthesis Flow

- HLS→RTL→gates/mem/dsp→place→route
- High-level doesn't know exactly what happens at lower level
  - How many gates some logic takes
  - Where blocks are placed (how far apart)
  - Which routes taken between blocks
    - Take minimum delay path? Forced to take longer?



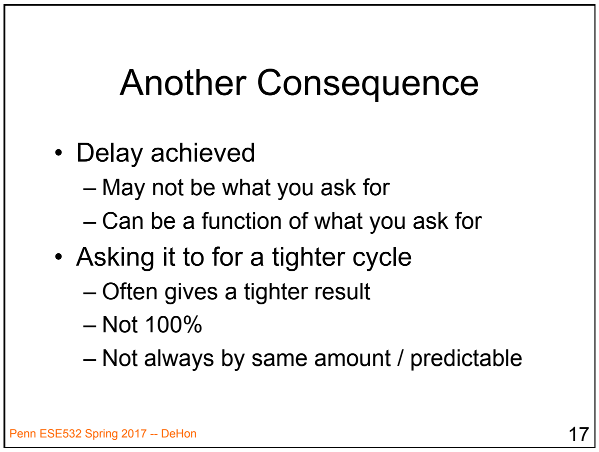
Vivado Synthesis

- Targets a clock cycle
- Optimization is open ended
  - …and it takes longer to get a lower delay result
- Stops when it thinks things are "fast enough"
  - Meets target



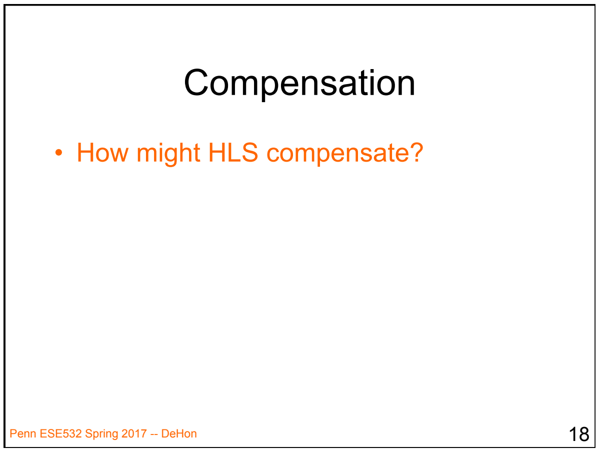
Consequence

- HLS not know exactly what will happen downstream
- HLS optimize for target clock
- Result: HLS may produce design that fails to meet timing during place&route



Another Consequence

- Delay achieved
  - May not be what you ask for
  - Can be a function of what you ask for
- Asking it to for a tighter cycle
  - Often gives a tighter result
  - Not 100%
  - Not always by same amount / predictable



Compensation

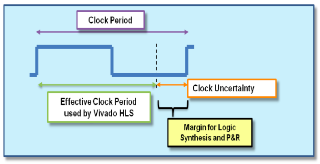
- How might HLS compensate?

Compensating

- Add margin for downstream
  - How much?
- Iterate
  - If downstream margin inadequate, increase margin and try again
- Try at multiple targets to explore possible frequencies

Compensating

- Add margin for downstream
  - How much?
  - Vivado HLS – allows specification of “clock uncertainty”


Clock Period
Clock Uncertainty
Effective Clock Period used by Vivado HLS
Margin for Logic Synthesis and P&R

Xilinx UG902 p. 184

Compensating

- Add margin for downstream
  - How much?
  - Vivado HLS – allows specification of “clock uncertainty”
- Iterate
  - If downstream margin inadequate, increase margin and try again
  - Vivado HLS – doesn’t do this automatically,
    - But you can…

Vivado HLS Pipelining

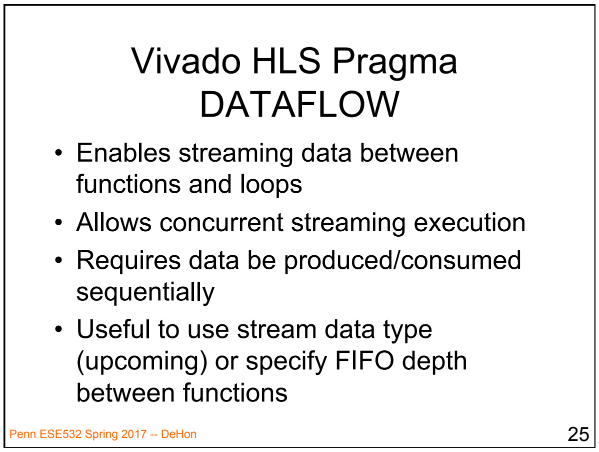
Design Mapping

- Streaming operators
- Area-Time tuning
  - Initiation Intervals for loop for streaming operator
  - Unrolling
- Benefits and tradeoffs of each?

Preclass 2

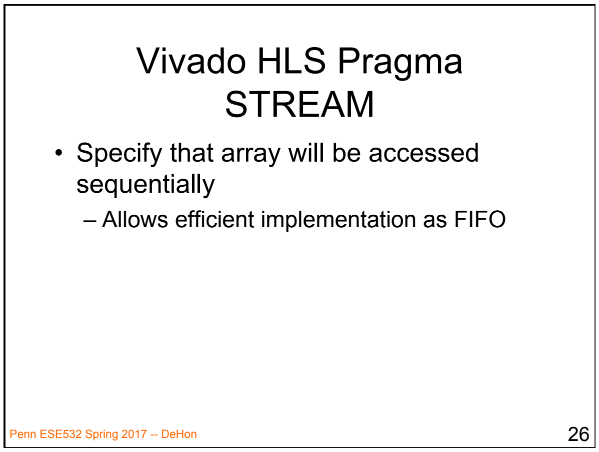
- What dataflow graph does this describe?

```
while(true) {
        i=read_input();
        fA(i,t1);
        fB(t1,t2);
        fC(t2,out);
        write_output(out);
}
```



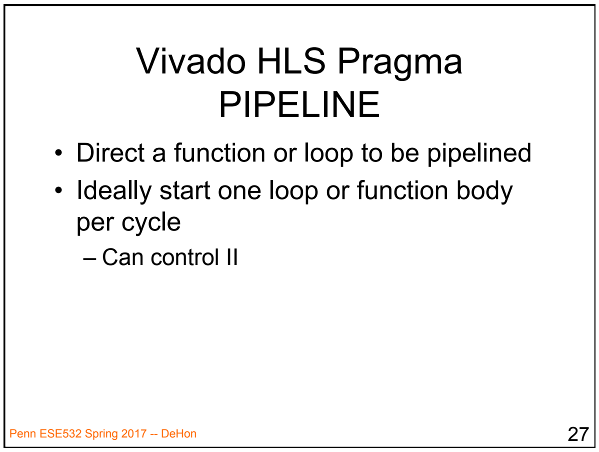
Vivado HLS Pragma DATAFLOW

- Enables streaming data between functions and loops
- Allows concurrent streaming execution
- Requires data be produced/consumed sequentially
- Useful to use stream data type (upcoming) or specify FIFO depth between functions



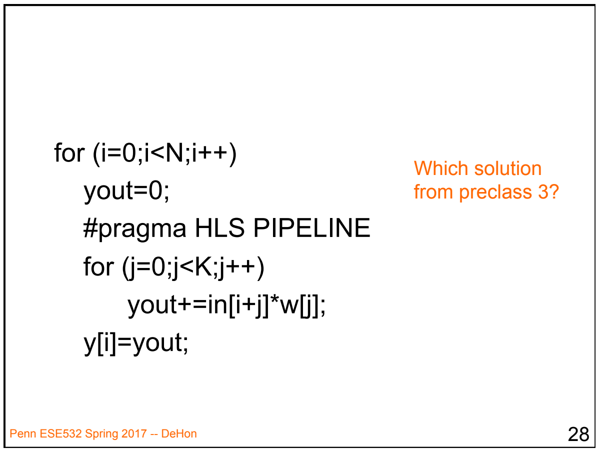
Vivado HLS Pragma STREAM

- Specify that array will be accessed sequentially
  - Allows efficient implementation as FIFO



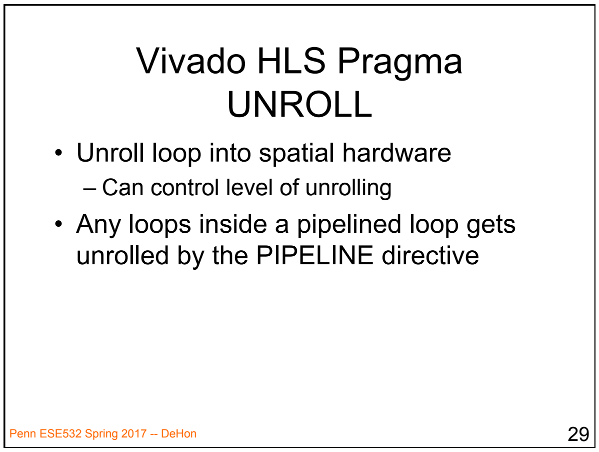
Vivado HLS Pragma PIPELINE

- Direct a function or loop to be pipelined
- Ideally start one loop or function body per cycle
  - Can control II



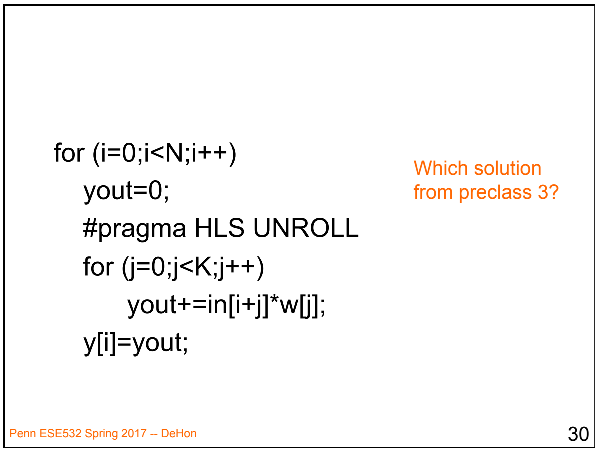
```
for (i=0;i<N;i++)
   yout=0;
   #pragma HLS PIPELINE
   for (j=0;j<K;j++)
        yout+=in[i+j]*w[j];
   y[i]=yout;
```

Which solution from preclass 3?



Vivado HLS Pragma UNROLL

- Unroll loop into spatial hardware
  - Can control level of unrolling
- Any loops inside a pipelined loop gets unrolled by the PIPELINE directive



```
for (i=0;i<N;i++)
   yout=0;
   #pragma HLS UNROLL
   for (j=0;j<K;j++)
        yout+=in[i+j]*w[j];
   y[i]=yout;
```

Which solution from preclass 3?

Vivado HLS Pragma INLINE

- Collapse function body into caller
  - Eliminates interface code
  - Allows optimization of inline code
- recursive option to inline a hierarchy
  - Maybe useful when explore granularity of accelerator

Vivado HLS Pragma ARRAY_PARTITION

- Spread out array over multiple BRAMs
  - By default placed in single BRAM
  - Use to remove memory bottleneck that prevents pipelining (limits II)

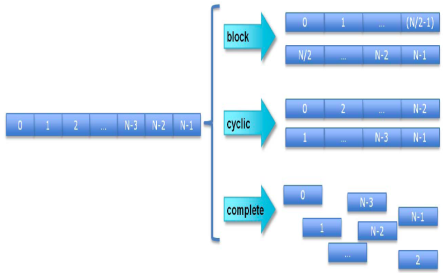
Memory Bottleneck Example

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

What problem if put mem in single BRAM?

Xilinx UG902 p. 91

Array Partition



Xilinx UG902 p. 195

Array Partition Example

#pragma ARRAY_PARTITION variable=mem cylic factor=4

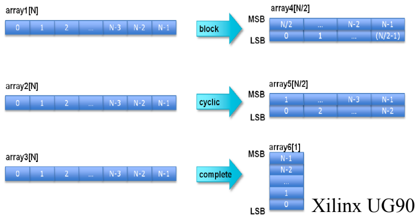
```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

Xilinx UG902 p. 91

Vivado HLS Pragma ARRAY_RESHAPE

- Pack data into BRAM to improve access (reduce BRAMs)
  - May provide similar benefit to partitioning without using more BRAMs

```
void foo (...) {
int  array1[N];
int  array2[N];
int  array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```



Xilinx UG902 p. 218

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

How fix if dint_t is 16b?

Xilinx UG902 p. 91

Array Reshape Example

#pragma ARRAY_RESHAPE variable=mem cylic factor=4 dim=1
(if din_t 16b)

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

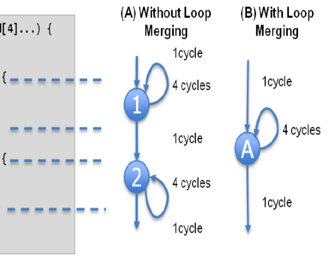
Xilinx UG902 p. 91

Vivado HLS Pragma LOOP_MERGE

- Combine loops to reduce delay

Loop Merge

```
void top (a[4],b[4],c[4],d[4]...) {

  ...
  Add: for (i=3;i>=0;i--) {
     if (d[i])
       a[i] = b[i] + c[i];
  }

  Sub: for (i=3;i>=0;i--) {
     if (!d[i])
       a[i] = b[i] - c[i];
  }
  ...
}
```

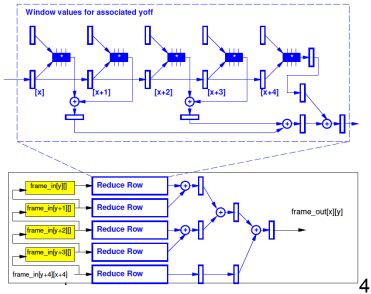


Xilinx UG902 p. 208

Feeding Data

Challenge

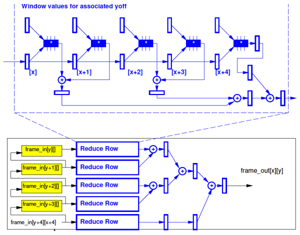
- How get Vivado HLS to provide a streaming, fully unrolled version like midterm 2f?



Midterm 2f

```
for(int y=0;y<(FSIZE-WSIZE+1);y++)
  for(int x=0;x<(FSIZE-WSIZE+1);x++) {
     frame_out[y][x]=0;
     for (int xoff=0;xoff<WSIZE;xoff++)
         for (int yoff=0;yoff<WSIZE;yoff++)
             frame_out[y][x]+=window[yoff][xoff]*frame_in[y+yoff][x+xoff];
     }
```

- Each pixel needed WSIZE^2 times
- Only want to read once from big memory
- (move once to array)



Three Coding Tricks

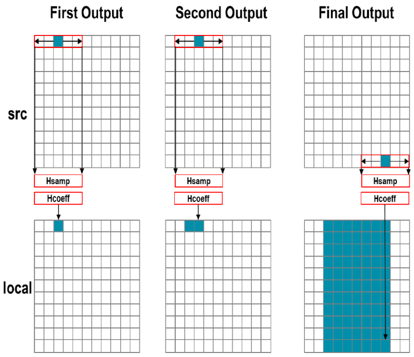
1. Setup input for streaming
2. Setup local window for X shift register
3. Setup linebuffer for Y

- Use Vivado HLS convolution example (pp. 104—121)
- …likely need for efficient full search…

Unoptimized Convolution

```
template<typename T, int K>
static void convolution_orig(
  int width,
  int height,
  const T *src,
  T *dst,
  const T *hcoeff,
const T *vcoeff) {

T local[MAX_IMG_ROWS*MAX_IMG_COLS];

// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
  HconvWfor(int row = border_width; row < width - border_width; row++){
    Hconv:for(int i = - border_width; i <= border_width; i++){
    }
  }
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
  VconvW:for(int row = 0; row < width; row++){
    Vconv:for(int i = - border_width; i <= border_width; i++){
    }
  }
```

Xilinx UG902 p. 104



Xilinx UG902 p. 105

Unoptimized Horizontal Conv.

```
// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
  HconvWfor(int row = border_width; row < width - border_width; row++){
    int pixel = col * width + row;
    Hconv:for(int i = - border_width; i <= border_width; i++){
      local[pixel] += src[pixel + i] * hcoeff[i + border_width];
    }
  }
}
```

- What’s inefficient here with src?

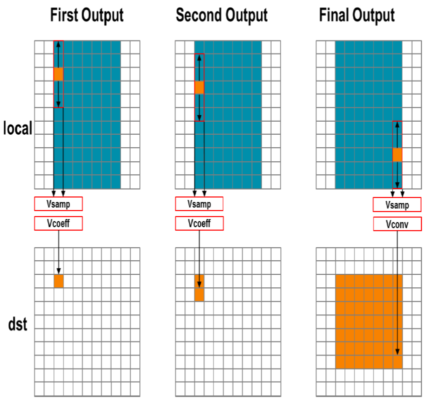
Xilinx UG902 p. 106

Unoptimized Horizontal Conv.

```
// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
  HconvWfor(int row = border_width; row < width - border_width; row++){
    int pixel = col * width + row;
    Hconv:for(int i = - border_width; i <= border_width; i++){
      local[pixel] += src[pixel + i] * hcoeff[i + border_width];
    }
  }
}
```

- Forces src data to be reread within window
  - Doesn't automatically figure out and localize

Xilinx UG902 p. 106



Xilinx UG902 p. 108

Unoptimized Vertical

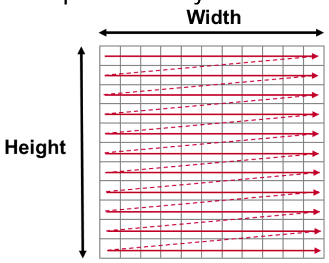
```
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
  VconvW:for(int row = 0; row < width; row++){
    int pixel = col * width + row;
    Vconv:for(int i = - border_width; i <= border_width; i++){
      int offset = i * width;
      dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
    }
  }
}
```

- Also forces multiple reads of pixel data

Xilinx UG902 p. 109

Optimizing I/O

- Want to read each pixel exactly once in scan order
  - Store locally

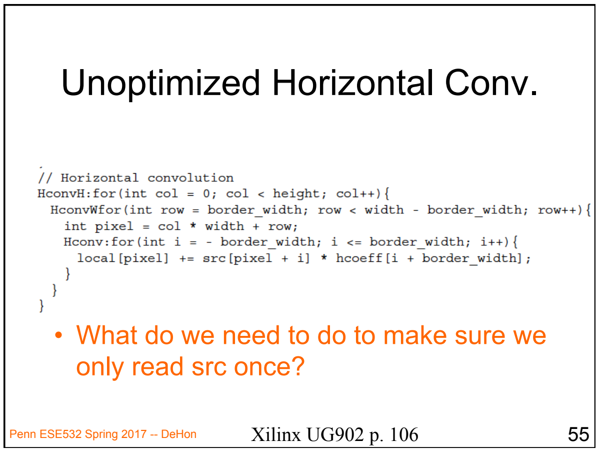


hls::stream

- Data accessed sequentially
- Can only read data item once
- Forces discipline for optimal I/O
  - Tells compiler that is what you are doing so it can optimize implementation
  - Infers a FIFO of depth 1
- Useful with function dataflow streams

```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
hls::stream<T> &src,
hls::stream<T> &dst,
const T *hcoeff,
const T *vcoeff)
{

hls::stream<T> hconv("hconv");
hls::stream<T> vconv("vconv");
// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
  HConvW:for(int row = 0; row < width; row++) {
    HConv:for(int i = 0; i < K; i++) {
    }
  }
}
```

Xilinx UG902 p. 114

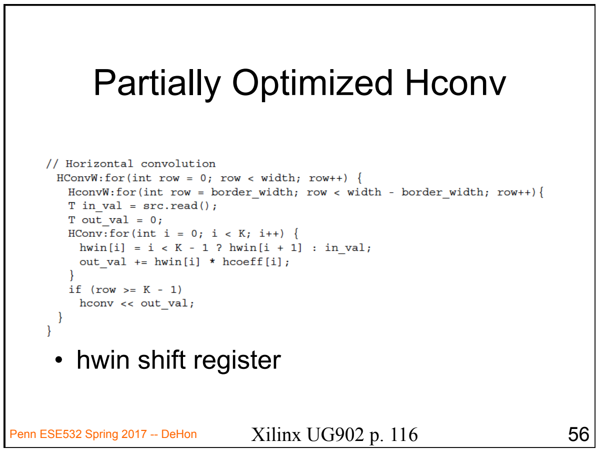


Unoptimized Horizontal Conv.

```
// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
  HconvWfor(int row = border_width; row < width - border_width; row++){
    int pixel = col * width + row;
    Hconv:for(int i = - border_width; i <= border_width; i++){
      local[pixel] += src[pixel + i] * hcoeff[i + border_width];
    }
  }
}
```

- What do we need to do to make sure we only read src once?

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 106   55

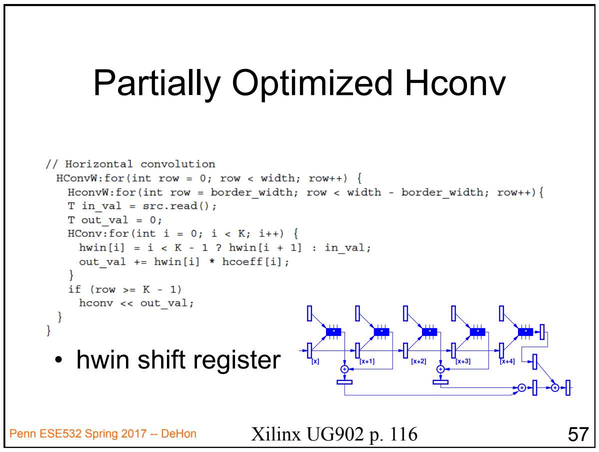


Partially Optimized Hconv

```
// Horizontal convolution
  HConvW:for(int row = 0; row < width; row++) {
    HconvW:for(int row = border_width; row < width - border_width; row++){
    T in_val = src.read();
    T out_val = 0;
    HConv:for(int i = 0; i < K; i++) {
      hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
      out_val += hwin[i] * hcoeff[i];
    }
    if (row >= K - 1)
      hconv << out_val;
  }
}
```

- hwin shift register

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 116   56

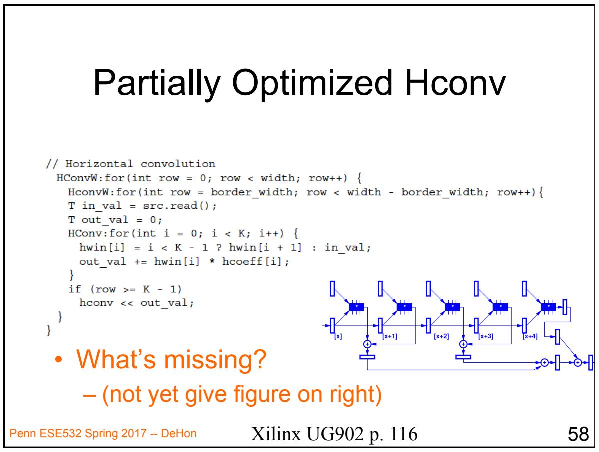


Partially Optimized Hconv

```
// Horizontal convolution
  HConvW:for(int row = 0; row < width; row++) {
    HconvW:for(int row = border_width; row < width - border_width; row++){
    T in_val = src.read();
    T out_val = 0;
    HConv:for(int i = 0; i < K; i++) {
      hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
      out_val += hwin[i] * hcoeff[i];
    }
    if (row >= K - 1)
      hconv << out_val;
  }
}
```

- hwin shift register

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 116   57

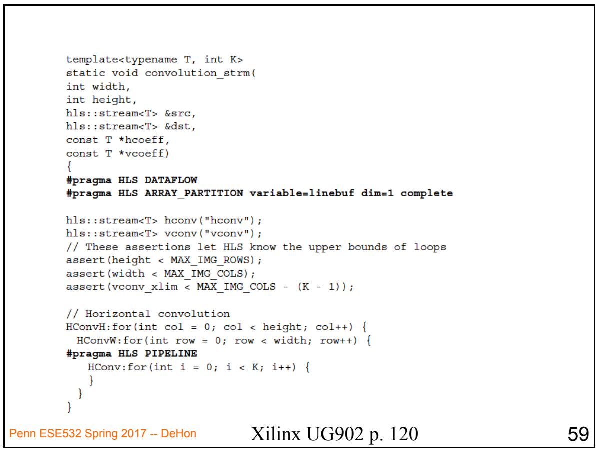


Partially Optimized Hconv

```
// Horizontal convolution
  HConvW:for(int row = 0; row < width; row++) {
    HconvW:for(int row = border_width; row < width - border_width; row++){
    T in_val = src.read();
    T out_val = 0;
    HConv:for(int i = 0; i < K; i++) {
      hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
      out_val += hwin[i] * hcoeff[i];
    }
    if (row >= K - 1)
      hconv << out_val;
  }
}
```

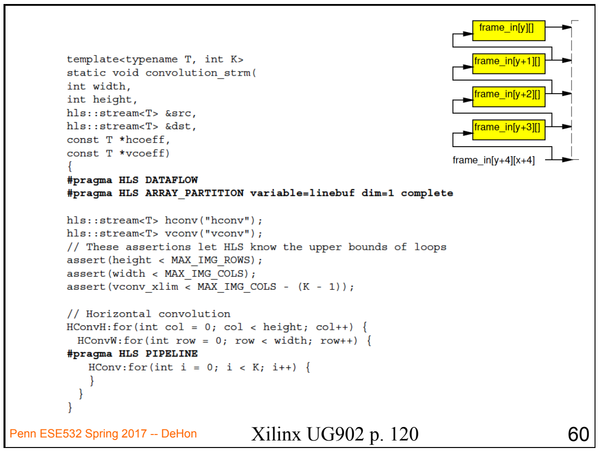
- What's missing?
  - (not yet give figure on right)

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 116   58



```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
hls::stream<T> &src,
hls::stream<T> &dst,
const T *hcoeff,
const T *vcoeff)
{
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITION variable=linebuf dim=1 complete

hls::stream<T> hconv("hconv");
hls::stream<T> vconv("vconv");
// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
 HConvW:for(int row = 0; row < width; row++) {
#pragma HLS PIPELINE
   HConv:for(int i = 0; i < K; i++) {
   }
  }
}
```

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 120   59



```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
hls::stream<T> &src,
hls::stream<T> &dst,
const T *hcoeff,
const T *vcoeff)
{
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITION variable=linebuf dim=1 complete

hls::stream<T> hconv("hconv");
hls::stream<T> vconv("vconv");
// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
 HConvW:for(int row = 0; row < width; row++) {
#pragma HLS PIPELINE
   HConv:for(int i = 0; i < K; i++) {
   }
  }
}
```

Penn ESE532 Spring 2017 -- DeHon   Xilinx UG902 p. 120   60

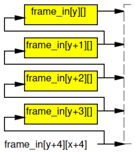
Unoptimized Vertical

```
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
  VconvW:for(int row = 0; row < width; row++){
    int pixel = col * width + row;
    Vconv:for(int i = - border_width; i <= border_width; i++){
      int offset = i * width;
      dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
    }
  }
}
```

- What do we need to do to only read from local once?

Xilinx UG902 p. 109

```
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
 VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS DEPENDENCE variable=linebuf inter false
#pragma HLS PIPELINE
    T in_val = hconv.read();
    T out_val = 0;
    VConv:for(int i = 0; i < K; i++) {
      T vwin_val = i < K - 1 ? linebuf[i][row] : in_val;
      out_val += vwin_val * vcoeff[i];
      if (i > 0)
        linebuf[i - 1][row] = vwin_val;
    }
    if (col >= K - 1)
      vconv << out_val;
  }
}
```



Xilinx UG902 p. 118

Three Coding Tricks

1. Setup input for streaming
2. Setup local window for X shift register
3. Setup linebuffer for Y

- …likely need for efficient full search…

Big Ideas

- Should be able to run at high clock rates (e.g. 400—550MHz)
- Vivado HLS gives control over pipelining
- Code may need some care and stylization to feed data efficiently
- Read Vivado HLS Users Guide (902)
  - Methodology, Design Optimization

Admin

- Project 4x and area Milestone
  - Due Friday