

Time Domain FIR Filter (TDFIR) Optimization Guide



© 2013 Altera Corporation—Public



Overview

- Introduction to FIR Filters
- TDFIR HPEC Benchmark
- How to Optimize TDFIR for the FPGA
- Results + Comparison

Introduction to FIR Filters

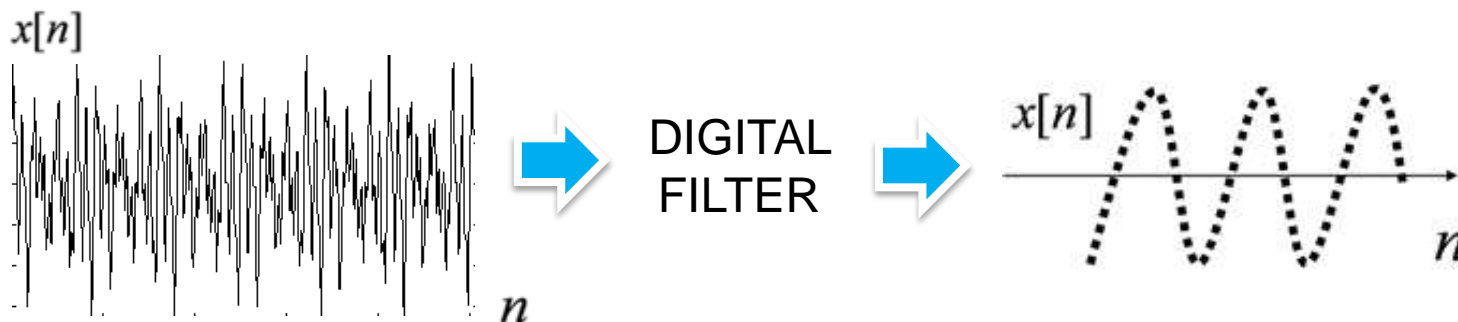


© 2013 Altera Corporation—Public



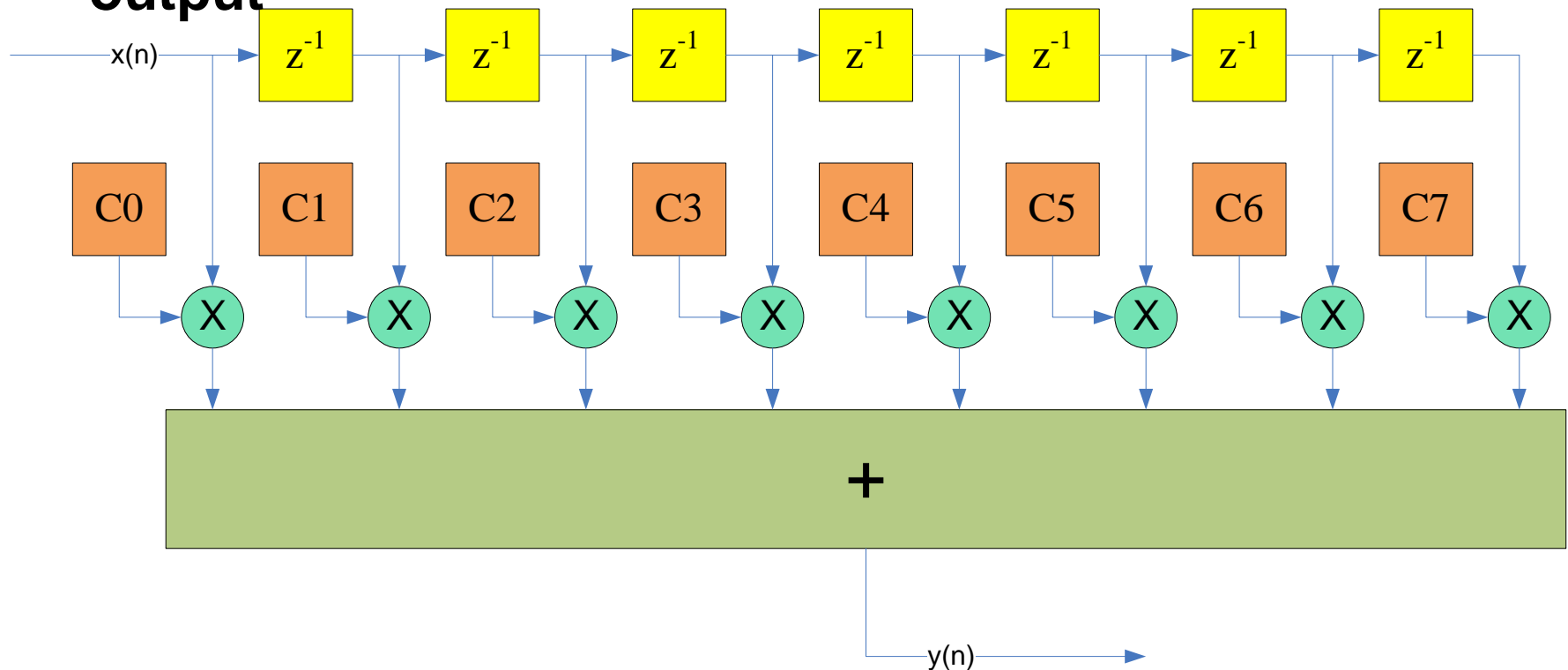
FIR Filter

- **The FIR filter is a basic building block for many market segments**
 - Wireless, video applications
 - Military and medical fields
- **It is a digital equivalent of the analog filter**
- **Purpose is to allow discrete signals in the time domain to be filtered (remove noise, high-frequency components, etc.)**



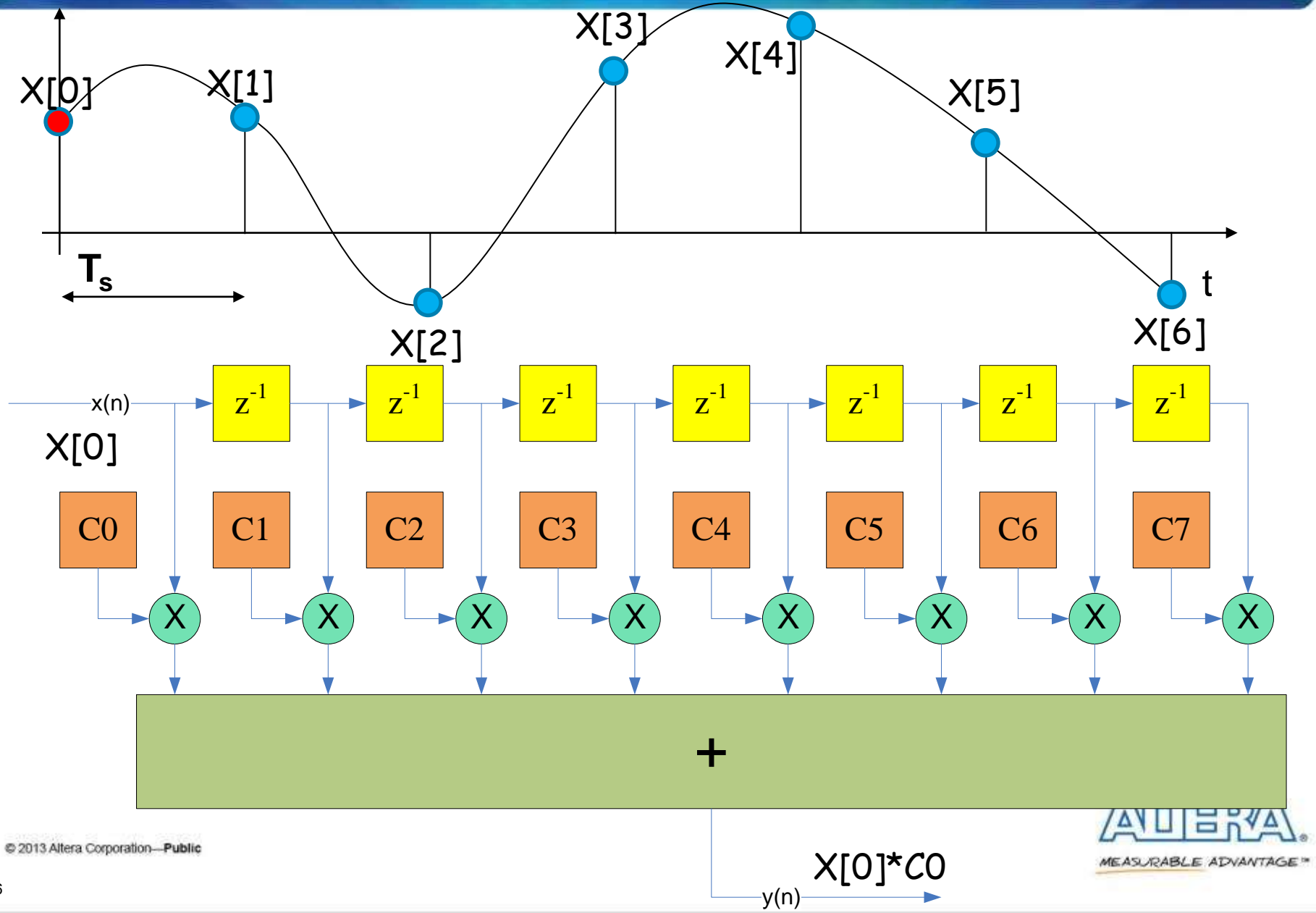
FIR Filter Structure

- Multiply a sequence of input samples by various coefficients and add their results to achieve an output

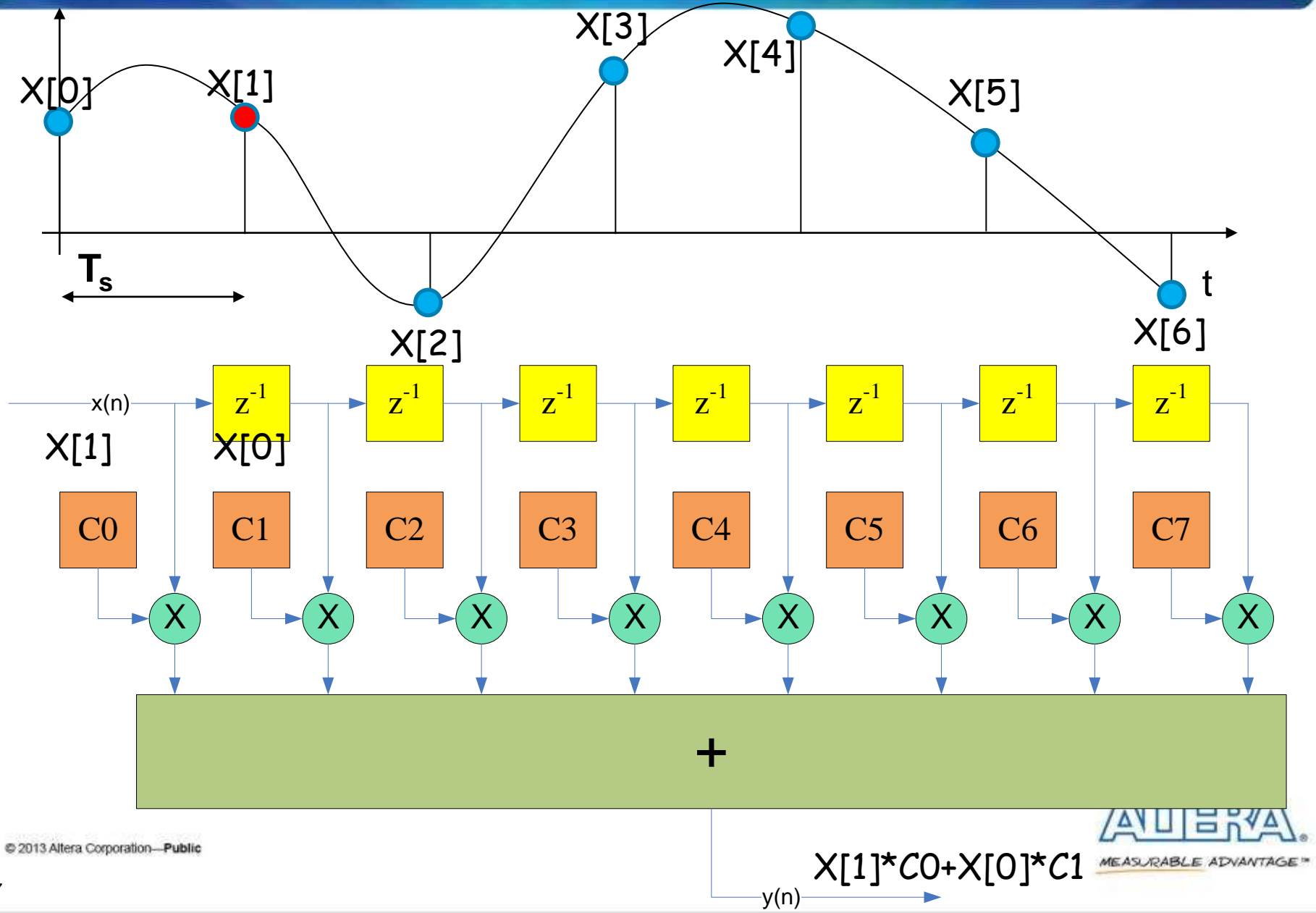


- This series of multiplying and adding will dictate how filter characteristics are shaped

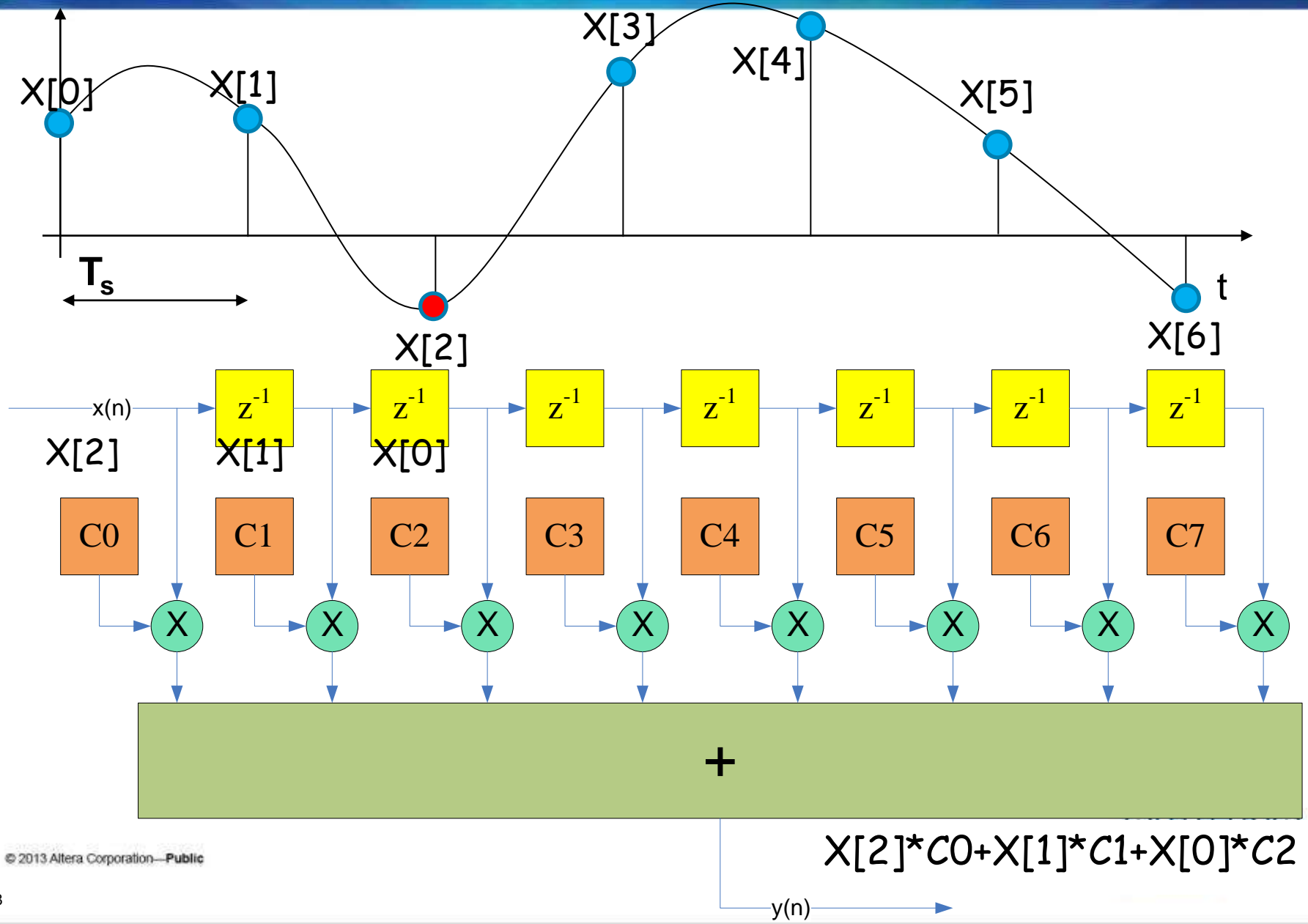
Example (Time Step 0)



Example (Time Step 1)

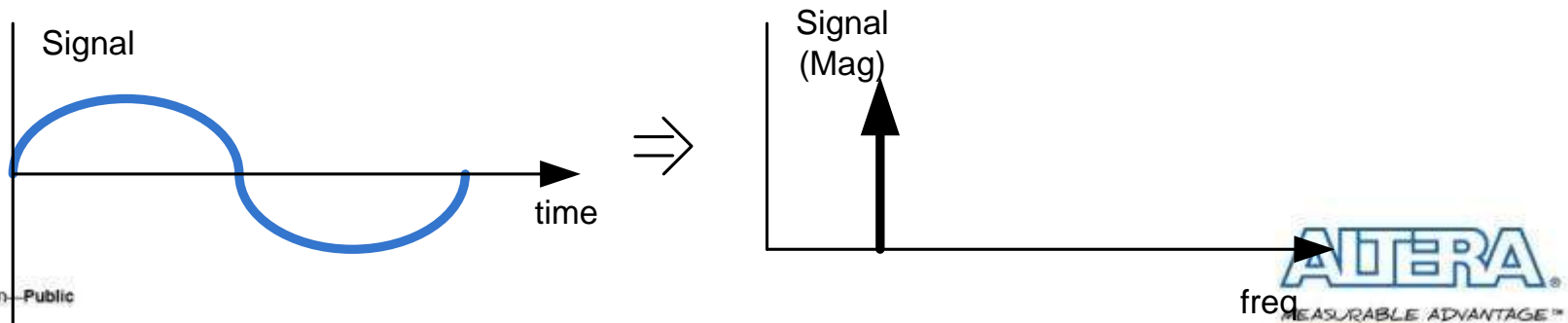
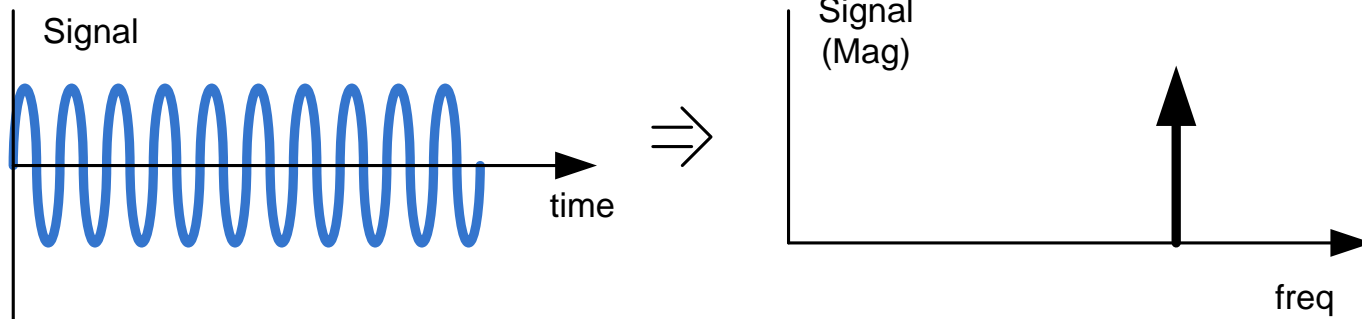


Example (Time Step 2)



Relationship between Time and Frequency

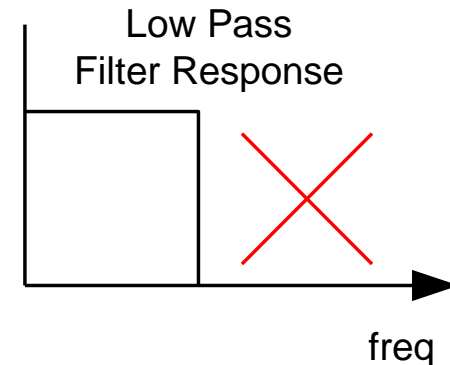
- What is a Frequency?
- Quick Changes in Time = High Frequency
- Slow Changes in Time = Low Frequencies



Example: Types of Filters

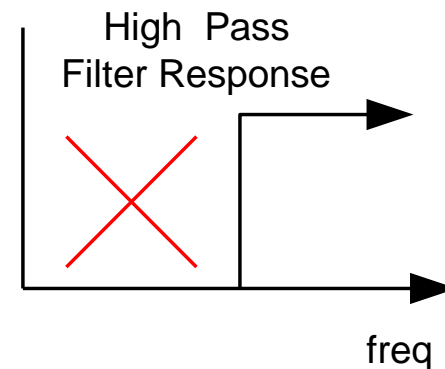
■ Low Pass Filters

- Pass All Frequencies Up to a Limit Frequency
- Reject Frequencies Greater Than Limit Frequency



■ High Pass Filters

- Pass All Frequencies Above to a Limit Frequency
- Reject Frequencies Less Than Limit Frequency



Designing Filters

- Get a desired response in the frequency domain (i.e. : low pass filter)
- Modify Desired response to possess certain “desirable” characteristics
 - I.e. : finite in length (truncate the response)
- Take the *Inverse Fourier Transform* to obtain the Impulse Response
- *The Impulse Response IS the value of the coefficients*

Filter Examples : Low Pass Filter

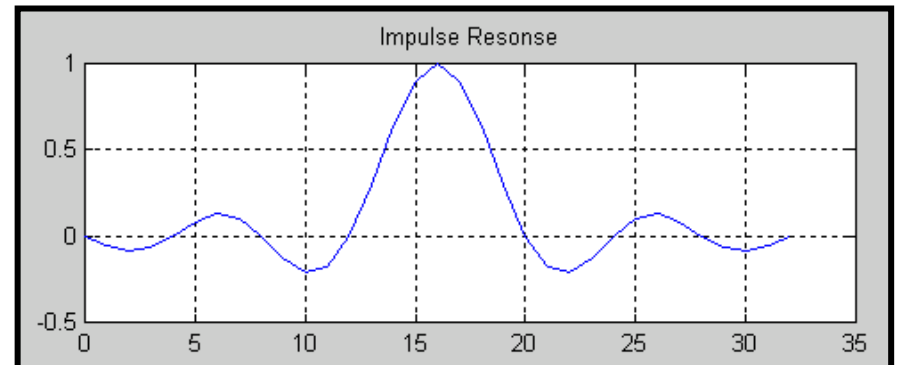
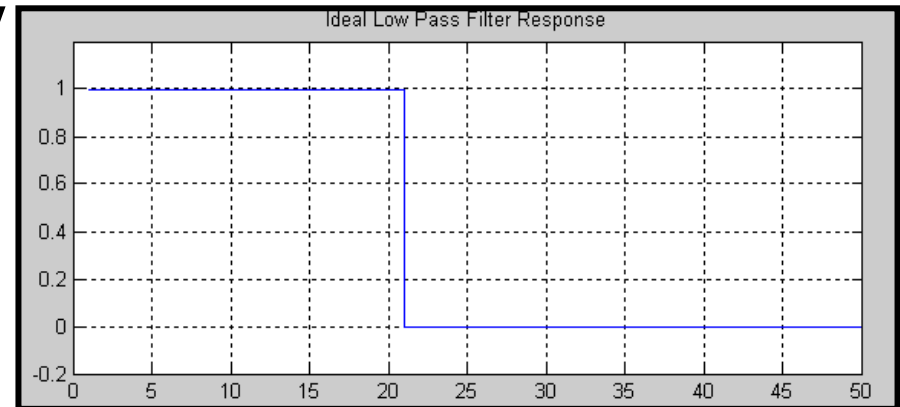
■ Low Pass Filter Frequency Response

- Start with Ideal Low Pass Filter

■ Take the inverse discrete Fourier Transform

■ Resulting Filter Coefficients

- All Frequencies Up to desired Frequency are present



TDFIR HPEC Benchmark

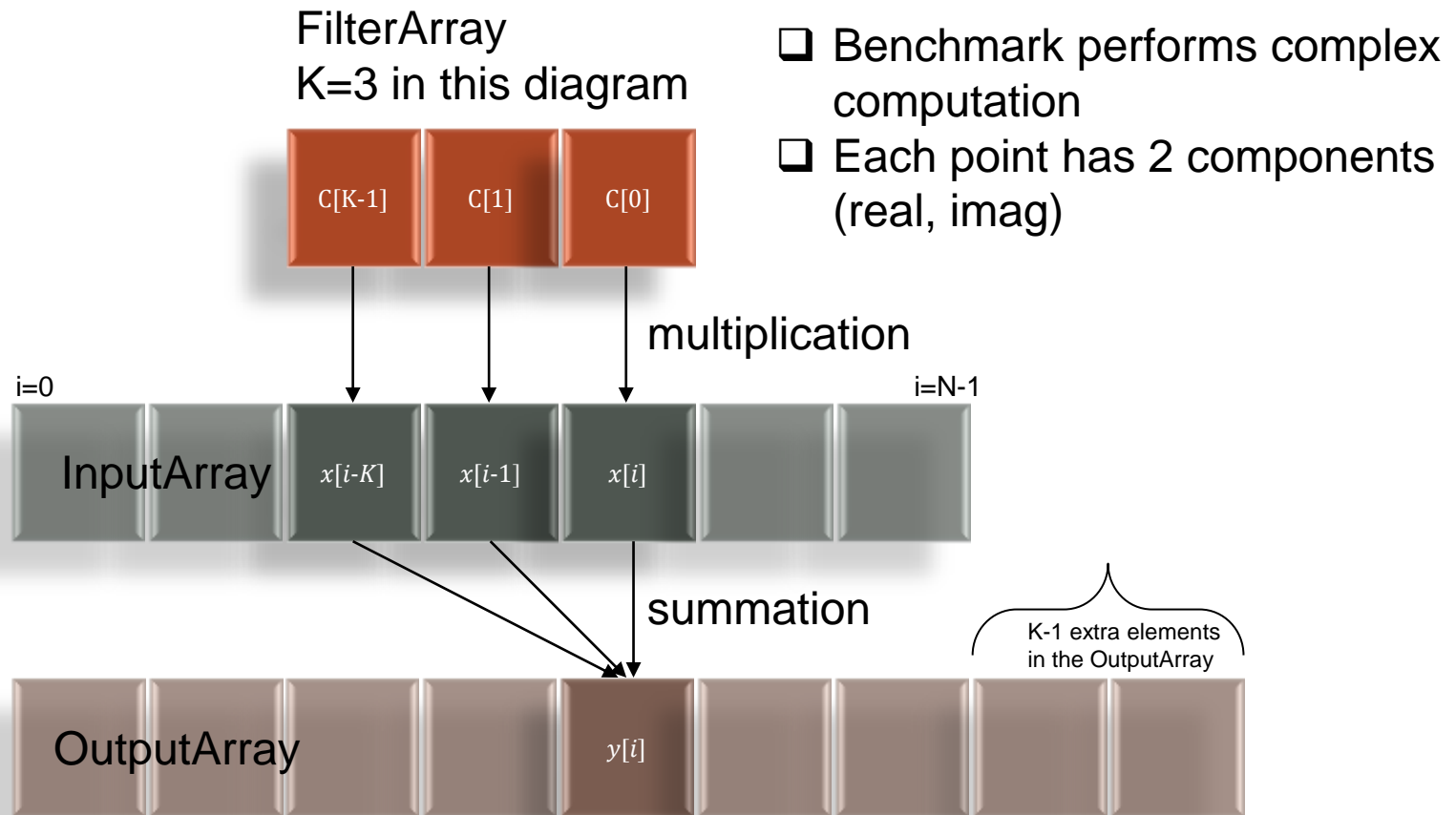
Slides courtesy of Craig Lund




© 2013 Altera Corporation—Public



Graphical Representation of the TDFIR



Benchmark Description

	Parameter	Description	Data Set 1	Data Set 2
	M	Number of filters to stream	64	20
	N	Length of InputArray in complex elements	4096	1024
	K	Length of FilterArray in complex elements	128	12
	Workload in MFLOP		268.44	1.97

We need some real-world data sizes if we are going to show results. For that we turn to the RADAR-oriented, HPEC Challenge benchmark suite. See the TDFIR kernel

<http://www.ll.mit.edu/HPECchallenge/tdfir.html>

Their benchmark offers two datasets. The timing data we present in later slides uses Data Set 1.

- ❖ Note that you won't find direct convolution implementations optimized for giant arrays. It is faster to use an FFT algorithm in that case. Using an FFT for that is benchmarked using hpecChallenge's FDFIR kernel.

Using HPEC Challenge Data

```
% TDFIR comes with a file containing separate input data for each filter.  
% The input data provided for each filter is exactly the same.  
% The benchmark implementation ignores that fact and reads in all the copies.  
  
% For this illustration we assume the data for each filter was read into rows  
% and a separate row used for each filter  
for f=1:M  
    OutputStream(f,:) = conv (InputStream(f,:), FilterStream(f,:));  
end
```

- This slide illustrates how the HPEC Challenge tdFIR benchmark uses the data that it supplies, expressed in MATLAB.
- In words, the benchmark presents a batch of M separate filters to compute.

Simple C Implementation

- The obvious implementation in C is very simple.
 - Often “good enough.”
- This implementation exactly matches the fundamental equation of the FIR filter.
- Upcoming slides show alternative implementation methods that lead to an efficient FPGA realization.

```
// Loop assumes OutputArray starts out full of zeros.  
for ( int FilterElement = 0; FilterElement < K; FilterElement++ )  
{ for ( int InputElement = 0; InputElement < N; InputElement++ )  
  { OutputArray [ InputElement + FilterElement ]  
    += InputArray [ InputElement ] * FilterArray[ FilterElement ];  
  }  
}
```

How to Optimize TDFIR for FPGA



Key Factors in Improving FPGA Throughput

- Restructuring data input and output
- Using local memory
- Implement single work-item execution

Optimization #1: Data Restructuring

- DataSet #1 contains 64 sets of filter data to process
- Each set of data contains 4096 complex points
- Original implementation has a double nested for loop

```
for ( int FilterElement = 0; FilterElement < K; FilterElement++ )  
{ for ( int InputElement = 0; InputElement < N; InputElement++ )  
  {  
    perform_computation(FilterElement, InputElement);  
  }  
}
```

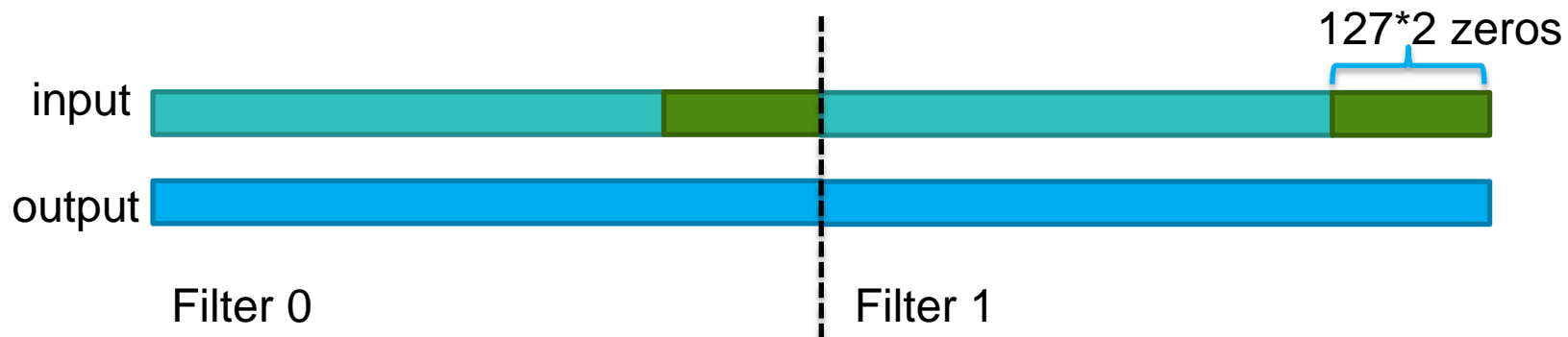
- For simplicity, we combined this into a single for loop

```
for ( int ilen = 0; ilen < TotalInputLength; ilen++)  
{  
  perform_computation(ilen);  
}
```

- This simplifies control flow

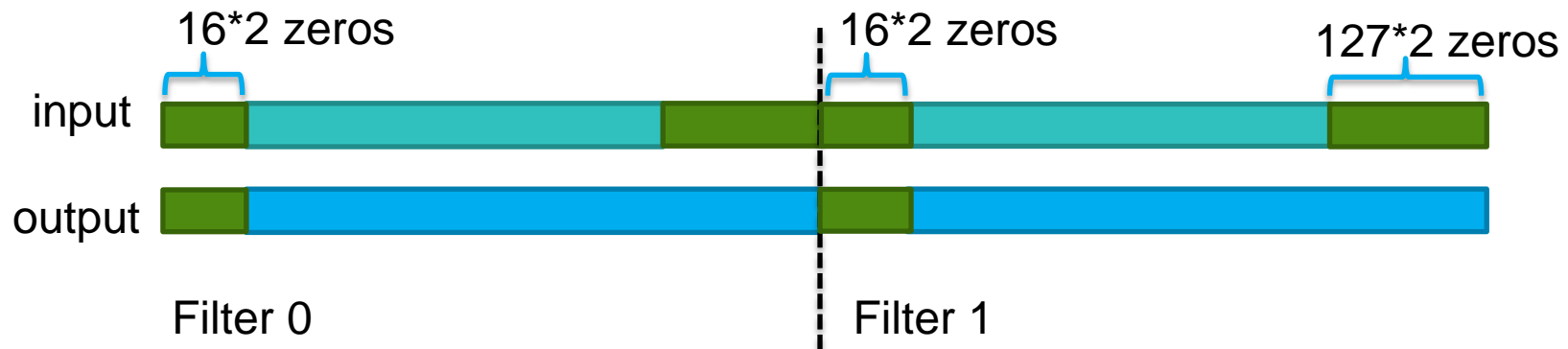
Alignment of input and output arrays

- **InputLength = 4096**
- **ResultLength = InputLength + FilterLength – 1**
 - $4096 + 128 - 1 = 4223$
- **Thus to maintain the expected behaviour, we PAD the input array by 127 complex points of zero**



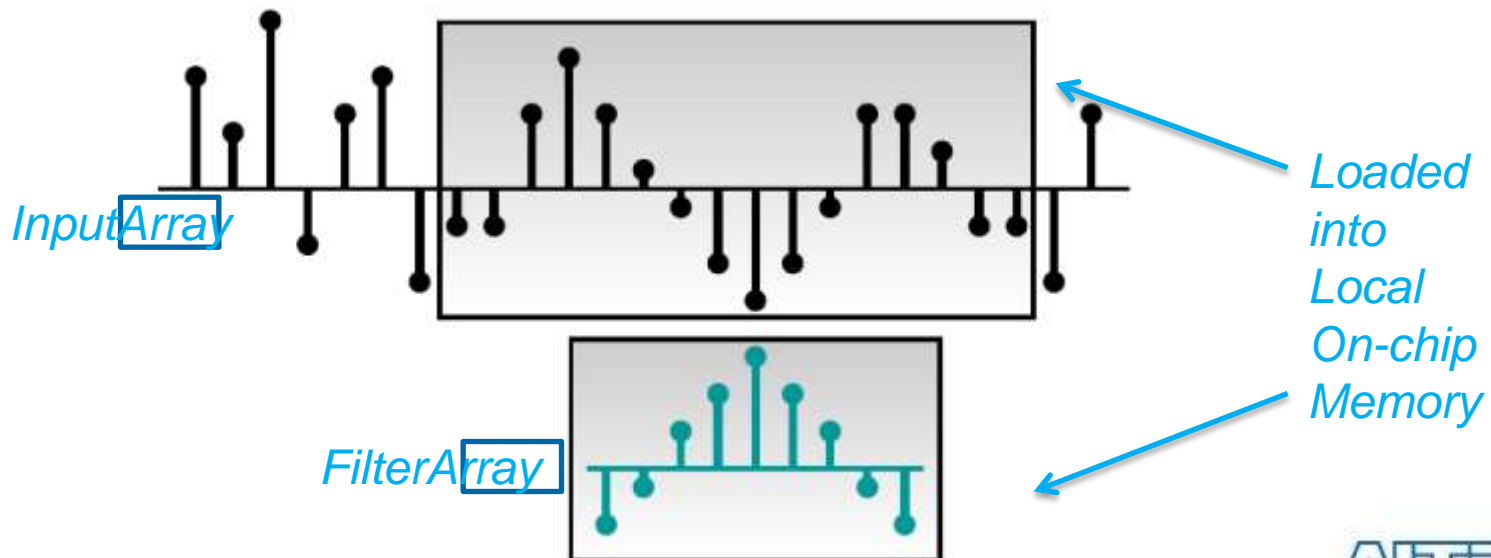
Loading the Filter Coefficients

- Each new filter to process has different filter coefficients.
- Must load these in before we can perform computation
- We chose to load these constants 8 complex points at a time
 - For the 128-tap filter, need to spend $128/8 = 16$ clock cycles to load in the filter coefficients
- To account for this, while at the same time maintaining our simplified control flow, we simply add more padding to the ***beginning*** of both the input and the output array



Optimization #2: Using Local Memory

- Accessing global memory is slow, so a more efficient implementation involves breaking the problem up into smaller segments that can fit in *local memory*
- The *InputArray* needs to be broken up into smaller pieces and the convolution of each of these subregions can be computed independently



Simplified Code Structure

```
// Hardcode these for efficiency
#define N 4096
#define K 128

__kernel void tdFIR
( __global const float *restrict InputArray, // Length N
  __global const float *restrict FilterArray, // Length K
  __global float *restrict OutputArray // Length N+K-1
)
{
  __local float local_copy_input_array[2*K+N];
  __local float local_copy_filter_array[K];

  InputArray += get_group_id(0) * N;
  FilterArray += get_group_id(0) * K;
  OutputArray += get_group_id(0) * (N+K);

  // Copy from global to local
  local_copy_input_array[get_local_id(0)] = InputArray[get_local_id(0)];
  if (get_local_id(0) < K)
    local_copy_filter_array[get_local_id(0)] = FilterArray[get_local_id(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  // Perform Compute
  float result=0.0f;
  for (int i=0; i<K; i++) {
    result += local_copy_of_filter_array[K-1-i]*local_copy_of_input_array[get_local_id(0)+i];
  }
  OutputArray[get_local_id(0)] = result;
}
```

**Ignoring Complex Math for simplicity*

Not the most efficient for FPGA

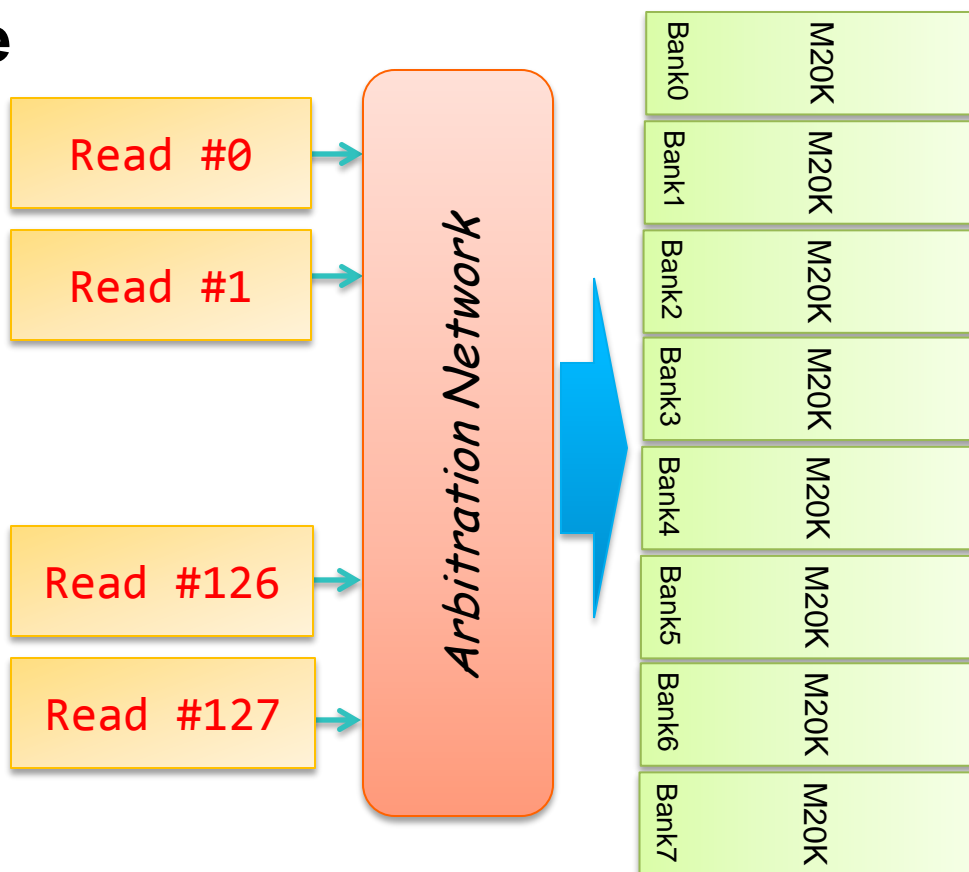
- Consider if we unrolled the compute loop

```
result = local_copy_of_filter_array[K-1]*local_copy_of_input_array[get_local_id(0)] +  
        local_copy_of_filter_array[K-2]*local_copy_of_input_array[get_local_id(0)+1] +  
        local_copy_of_filter_array[K-3]*local_copy_of_input_array[get_local_id(0)+2] +  
        ...+  
        local_copy_of_filter_array[0]*local_copy_of_input_array[get_local_id(0)+K-1];
```

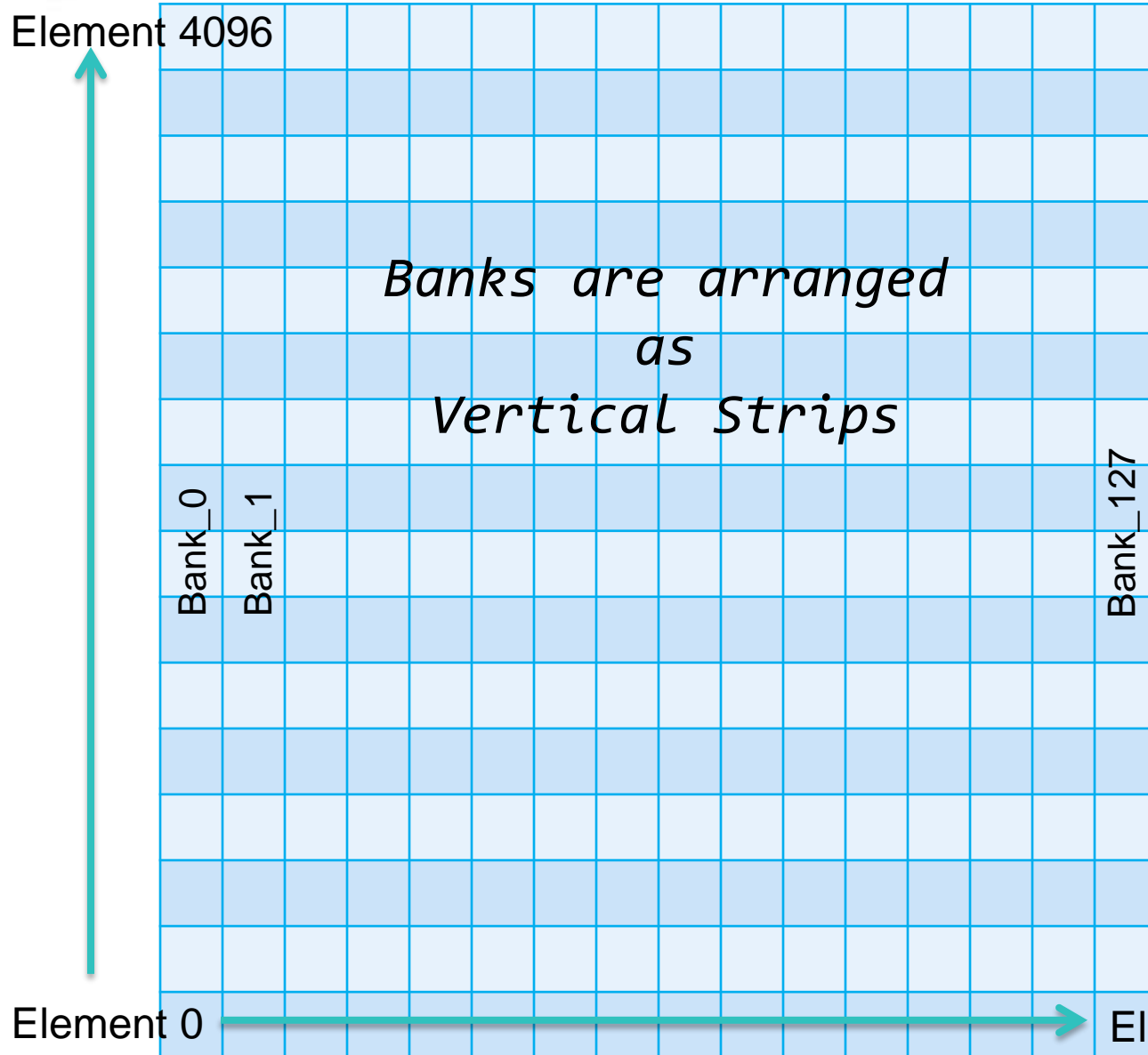
- We have something that looks somewhat like a good circuit for a FIR filter
- Several loads from memory fetch 128 coefficients from the *local FilterArray* and 128 elements from the *local InputArray*

Reading from a banked Memory

- On any given clock cycle, as long as all requests are asking for elements in different banks → we can service ALL requests!
- Altera's OpenCL compiler automatically builds this structure



Using Banking to service all Reads



Disadvantages

- **Banked local memory structures are an inefficient way to handle FIR data reuse.**
 - Consumes LOTS of area and on-chip logic and memory resources
- *Notice that every thread accesses almost the same data, but shifted by one position*
 - We really need to create a shift register structure!
 - It is the ultimate form of expressing the data reuse pattern for the FIR filter

Optimization #3: Implementing Single Work Item Execution

```
// Hardcode these for efficiency
```

```
#define N 4096
```

```
#define K 128
```

```
__kernel __attribute__((task)) void tdFIR
(
    __global const float *restrict InputArray, // Length N
    __global const float *restrict FilterArray, // Length K
    __global float *restrict OutputArray, // Length N+K-1
    unsigned int totalLengthOfInputsConcatenated
)
{
    float data_shift_register[K];
    for (int i=0; i<totalLengthOfInputsConcatenated; i++) {
        #pragma unroll
        for (int j=0; j<K; j++)
            data_shift_register[j]=data_shift_register[j+1];
        data_shift_register[K-1]=InputArray[i];

        float result=0.0f;
        #pragma unroll
        for (int j=0; j<K; j++)
            result += data_shift_register[K-1-j]*FilterArray[j];

        OutputArray[i] = result;
    }
}
```

Shift register
implementation

**For Simplicity Assume the FilterArray doesn't change*

Consider the unrolled loops ... Again

■ The first unrolled loop looks like:

```
data_shift_register[0]=data_shift_register[1];  
data_shift_register[1]=data_shift_register[2];  
data_shift_register[2]=data_shift_register[3];  
...  
data_shift_register[K-1]=InputArray[i];
```

■ The second unrolled loop:

```
result = data_shift_register[K-1]*FilterArray[0] +  
         data_shift_register[K-2]*FilterArray[1] +  
         data_shift_register[K-3]*FilterArray[2] +  
         ...  
         data_shift_register[0]*FilterArray[K-1];
```

■ *The key observation is that all array accesses are from a constant address*

- Altera's OpenCL compiler will now build 128 floating point registers instead of a complex memory system

Shift Register Implementation

Iteration i	Iteration i+1
data_shift_register[0]	data_shift_register[0]
data_shift_register[1]	data_shift_register[1]
data_shift_register[2]	data_shift_register[2]
data_shift_register[3]	data_shift_register[3]
data_shift_register[4]	data_shift_register[4]
...	...
data_shift_register[K-1]	data_shift_register[K-1]

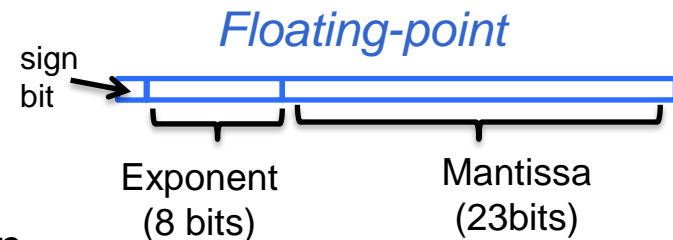
- Pipelining iterations in this loop is very simple because the dependencies are trivial
- Essentially becomes a shift register in hardware.

We handle Floating-Point !

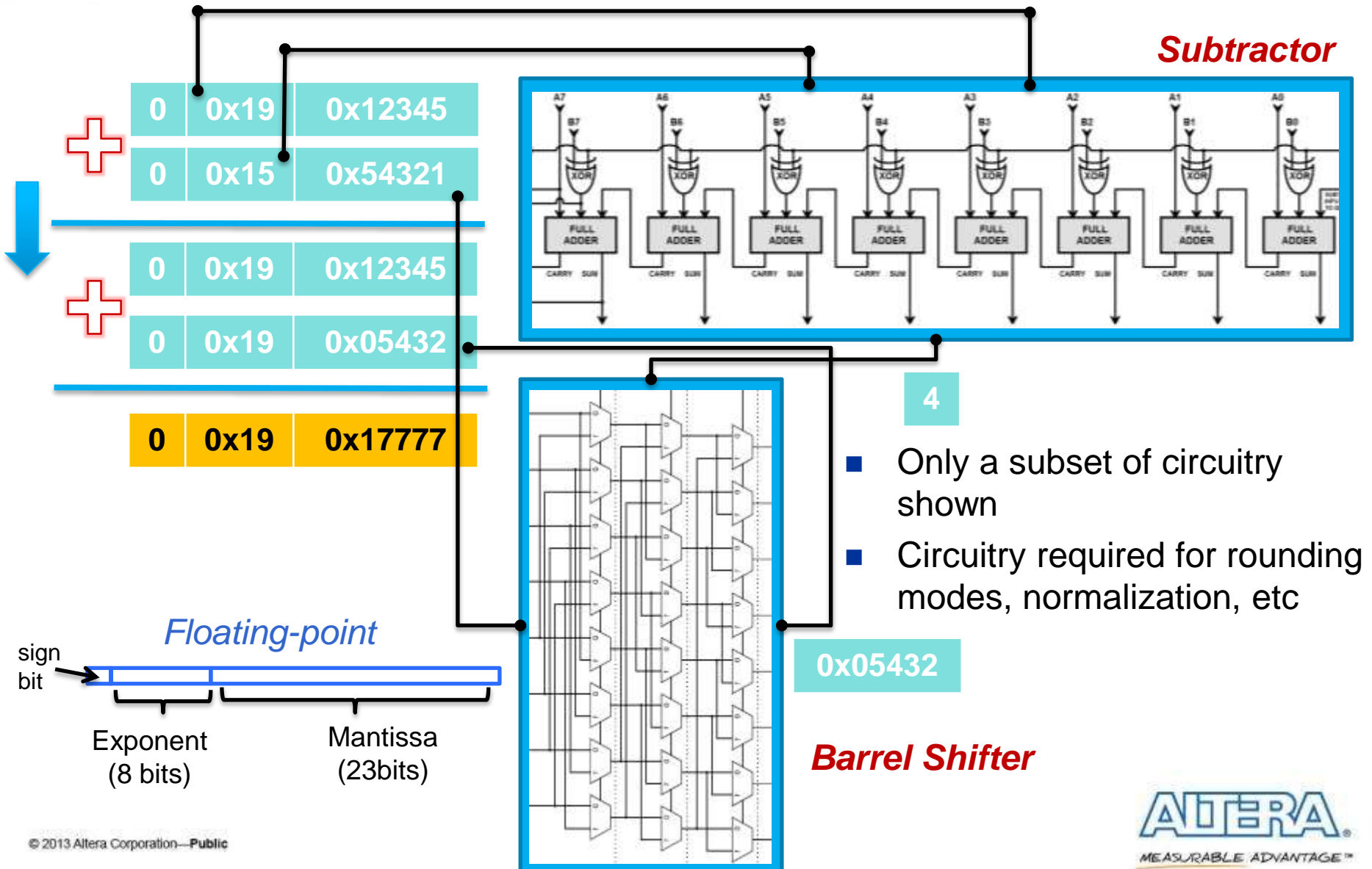
- Notice that the TDFIR is a Floating Point Application
- Floating-point formats represent a large range of real numbers with a finite number of bits

- Extreme care is required to handle OpenCL compliant floating point on the FPGA

- Handling of Infs, NaNs
- Denormalized Numbers
- Rounding



Example: Floating Point Addition



OpenCL builds on Altera's Floating Point Technology

Function	ALUTs	Register	Multipliers (27x27)	Latency	Performance
Add/Subtract	541	611	n/a	14	497 MHz
Multiplier	150	391	1	11	431 MHz
Divider	254	288	4	14	316 MHz
Inverse	470	683	4	20	401 MHz
SQRT	503	932	n/a	28	478 MHz
Inverse SQRT	435	705	6	26	401 MHz
EXP	626	533	5	17	279 MHz
LOG	1,889	1,821	2	21	394 MHz

We can implement the full range of OpenCL math functions

TDFIR Results



© 2013 Altera Corporation—Public



Results of Task TDFIR

- Data collected using the BittWare S5-PCle-HQ (S5PH-Q) board
- Shipped .aocx yields 170 GFLOP/s



Sample Data Output

```
Loading tdfir_131_s5phq_d8.aocx...  
tdFirVars: inputLength = 4096, resultLength = 4223, filterLen = 128  
Done.  
Latency: 0.001575 s.  
Buffer Setup Time: 0.004548 s.  
Throughput: 170.491 GFLOPs.  
Verification: PASS
```

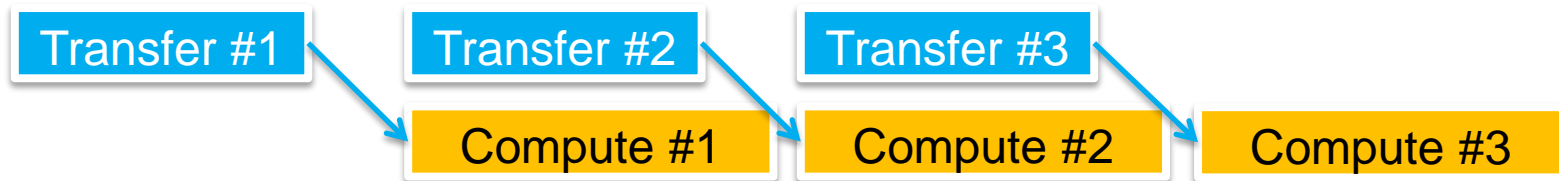
OpenCL Overhead

Dataset	M	N	K	Input Bytes	Filter Bytes	Output Bytes	Total MB transferred	Transfer Time (ms)
Large	64	4096	128	2097152	65536	2162176	4.35 MB	1.36
Small	20	1024	12	163840	1920	165600	0.36 MB	0.11

- **OpenCL is usually associated with PCIe attached acceleration hardware.**
- **The DMA of data over PCIe consumes time.**
- **Transfer Time is the amount of time to transfer all of the data back and forth to the FPGA using PCIe Gen2x8.**

Can overlap transfers and computation

- For the most common case in HPEC, we likely want to continuously process batches of filters as time progresses.
- Can overlap transfer and compute.



- This pipelining approach can be used to hide transfer latency

Comparisons



© 2013 Altera Corporation—Public



Evaluating the Potential of Graphics Processors for High Performance Embedded Computing

Shuai Mu¹, Chenxi Wang¹, Ming Liu², Dongdong Li², Maohua Zhu¹, Xiaoliang Chen³, Xiang Xie¹, Yangdong Deng¹

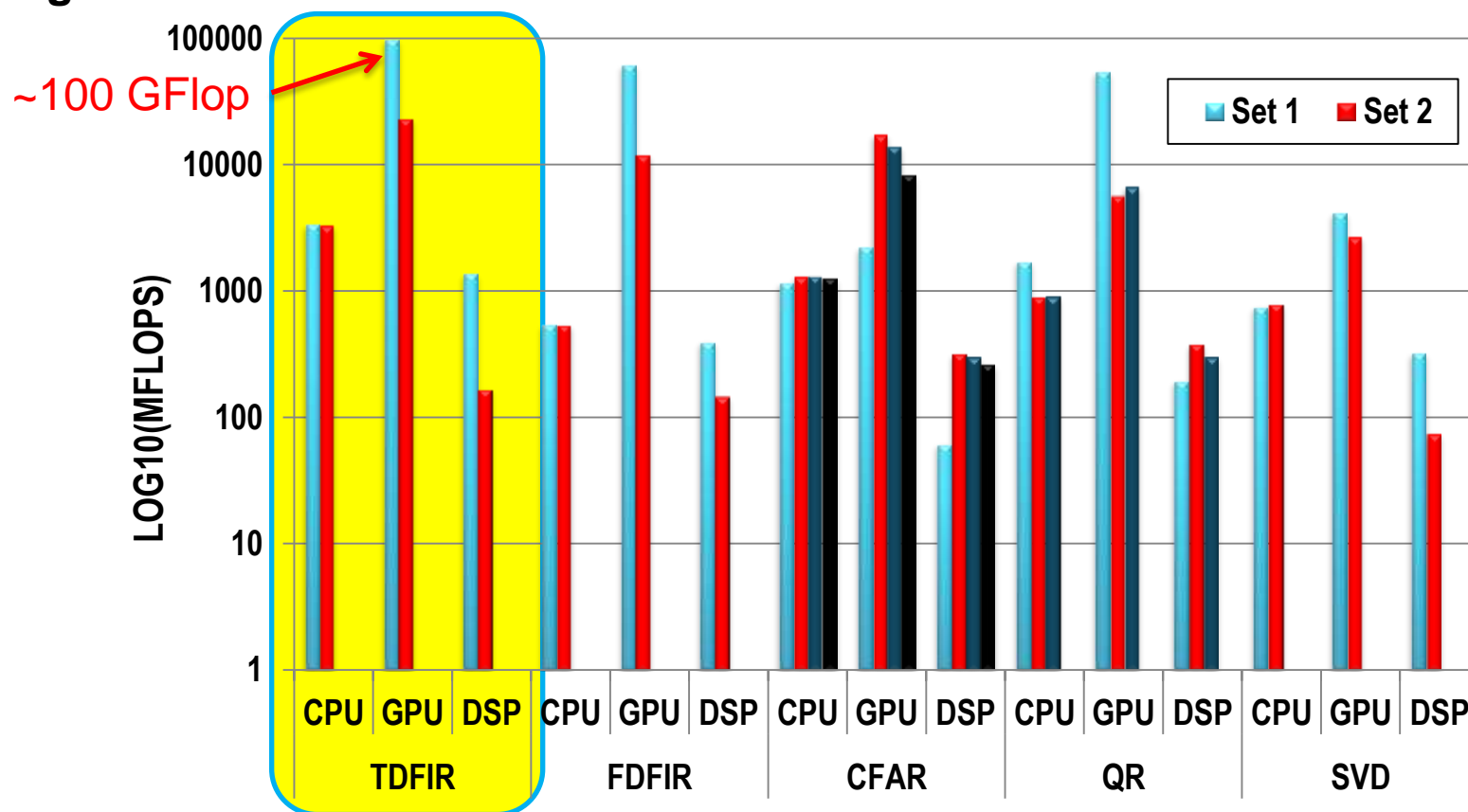
¹Tsinghua University, ²BeiHang University, ³Chinese Academy of Sciences

{mus04ster, zhumaohua, yangdong.deng, lmingesce}@gmail.com, wx09@mails.tsinghua.edu.cn, xiexiang@tsinghua.org.cn,

Kernels	Data Set	CPU Throughput (GFLOPS) *	GPU Throughput (GFLOPS) *	Speedup
TDFIR	Set 1	3.382	97.506	28.8
	Set 2	3.326	23.130	6.9
FDFIR	Set 1	0.541	61.681	114.1
	Set 2	0.542	11.955	22.1
CT	Set 1	1.194	17.177	14.3
	Set 2	0.501	35.545	70.9
PM	Set 1	0.871	7.761	8.9
	Set 2	0.281	21.241	75.6
CFAR	Set 1	1.154	2.234	1.9
	Set 2	1.314	17.319	13.1
	Set 3	1.313	13.962	10.6
	Set 4	1.261	8.301	6.6
GA	Set 1	0.562	1.177	2.1
	Set 2	0.683	8.571	12.5
	Set 3	0.441	0.589	1.4
	Set 4	0.373	2.249	6.0
QR	Set 1	1.704	54.309	31.8
	Set 2	0.901	5.679	6.3
	Set 3	0.904	6.686	7.4
SVD	Set 1	0.747	4.175	5.6
	Set 2	0.791	2.684	3.4
DB	Set 1	112.3	126.8	1.13
	Set 2	5.794	8.459	1.46

Performance Comparison

GPU: NVIDIA Fermi, CPU: Intel Core 2 Duo (3.33GHz), DSP AD TigerSHARC 101



Our FPGA implementation achieves 170-190 GFlops depending on FMax



Thank You



© 2013 Altera Corporation—Public

ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/legal.

