FPGA-based design of a Million point Sparse FFT

Abhinav Agarwal (abhiag@mit.edu)
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Outline

- Introduction
- Motivation for hardware
- Implementation
  - 4 Blocks: Challenges and solutions
  - Resource usage and performance constraints
- Conclusion

Collaborators

- Haitham Hassanieh
- Omid Abari
- Ezzeldin Hamed
- Prof. Dina Katabi
- Prof. Arvind

SFFT implementation

- Haitham Hassanieh, Piotr Indyk, Dina Katabi, Eric Price: *Simple and Practical Algorithm for Sparse Fourier Transform,* Proceedings of the Symposium on Discrete Algorithms, 2012
- SFFT already quite fast in software: 20 ms per million pt FFT

- Special purpose hardware may offer several more benefits:
  - *power :* 100-1000x lower than software
  - *cost and form factor:* Enabling wireless, mobile or air borne applications
  - *real time:* make it possible to compute SFFT in real time on streaming data

- Design metrics: 10x higher performance than SW while consuming 100x lower power

Rapid Prototyping - FPGA

- Custom hardware – ASICs
  - High performance, low power
  - Very high costs, fixed design, no revisions
- FPGA based prototyping
  - Lower performance & efficiency than ASICs,
    - Still beats SW handily
  - Very low cost
  - Reprogrammable – useful for evolving algorithms
  - Parameterized HW for different applications

Goal & Challenges

- Goal: Perform million pt FFTs at ~1 ms throughput
  - Enable ~1 GHz line rate
- Use FPGA prototyping for hardware design & development
  - **Target Platform:** Xilinx ML605 platform with Virtex-6 FPGA
  - Power consumption: 1 W
- Challenges
  - Algorithm uses extremely large shared data structures
  - Requires a very fast and relatively large dense FFT (1024 - 4096 pts)
  - Limited HW resources on FPGA $\Rightarrow$ need modular dataflow
  - Architectural exploration needed as SFFT algorithm relatively new
- HDL: Bluespec System Verilog

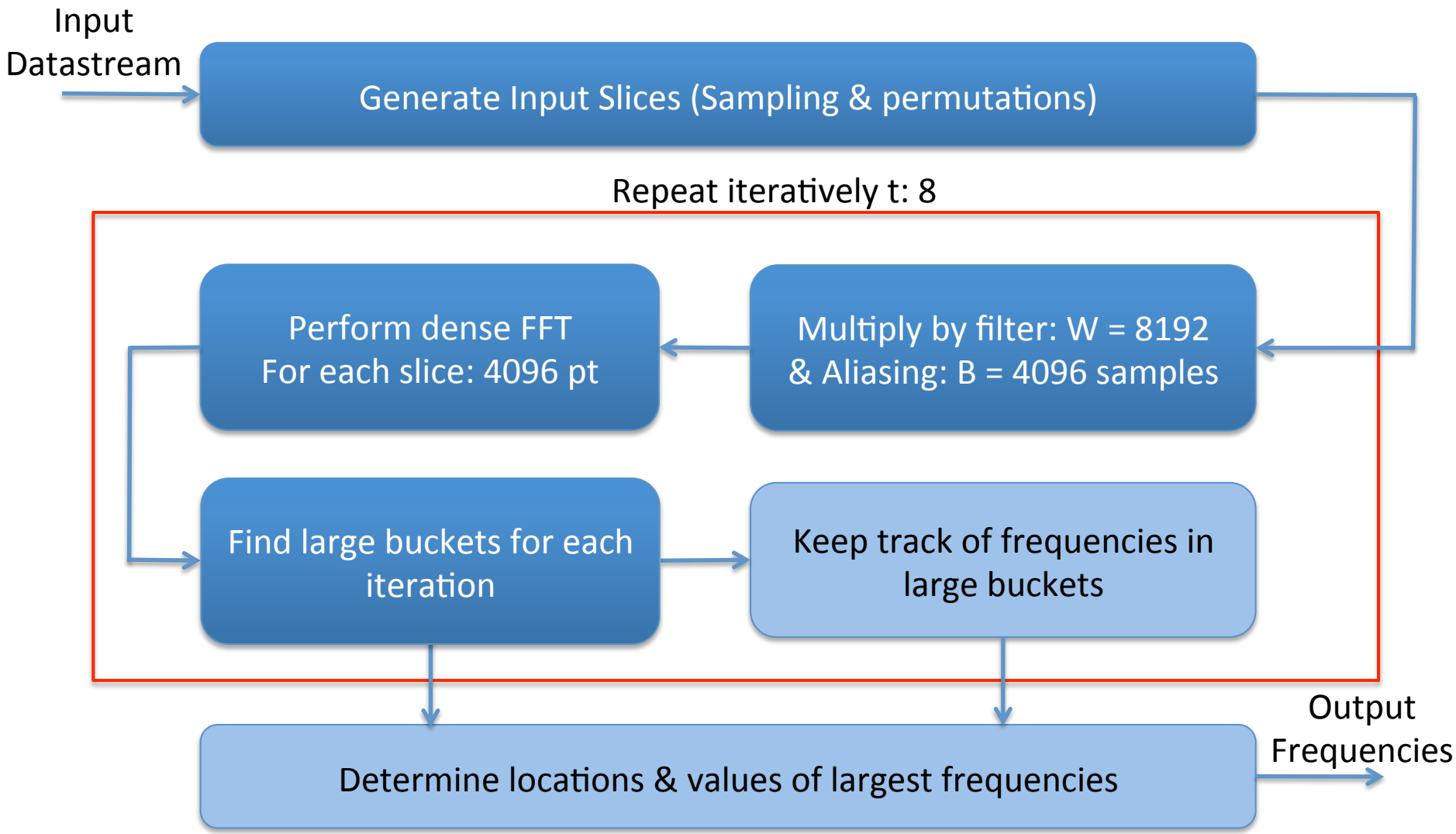
Sparse FFT algorithm parameters

- Sub-linear algorithm requires only some of the input data
- Iterative probabilistic algorithm
- Higher the no of iterations & greater the input slice sizes
  - Higher the confidence in output
  - Higher the noise tolerance
  - Lower the error
- Parameters under consideration
  - Input Slice size: W - storage requirements
  - No of buckets: B - computational requirements
  - No of iterations: t - performance & storage
- Chosen Values for $N = 2^{20}$ and $k = 500$
  - W = 8192, B = 4096, t = 8

Outline

- Introduction
- Motivation for hardware
- Implementation
  - 4 Blocks: Challenges and solutions
- Conclusion

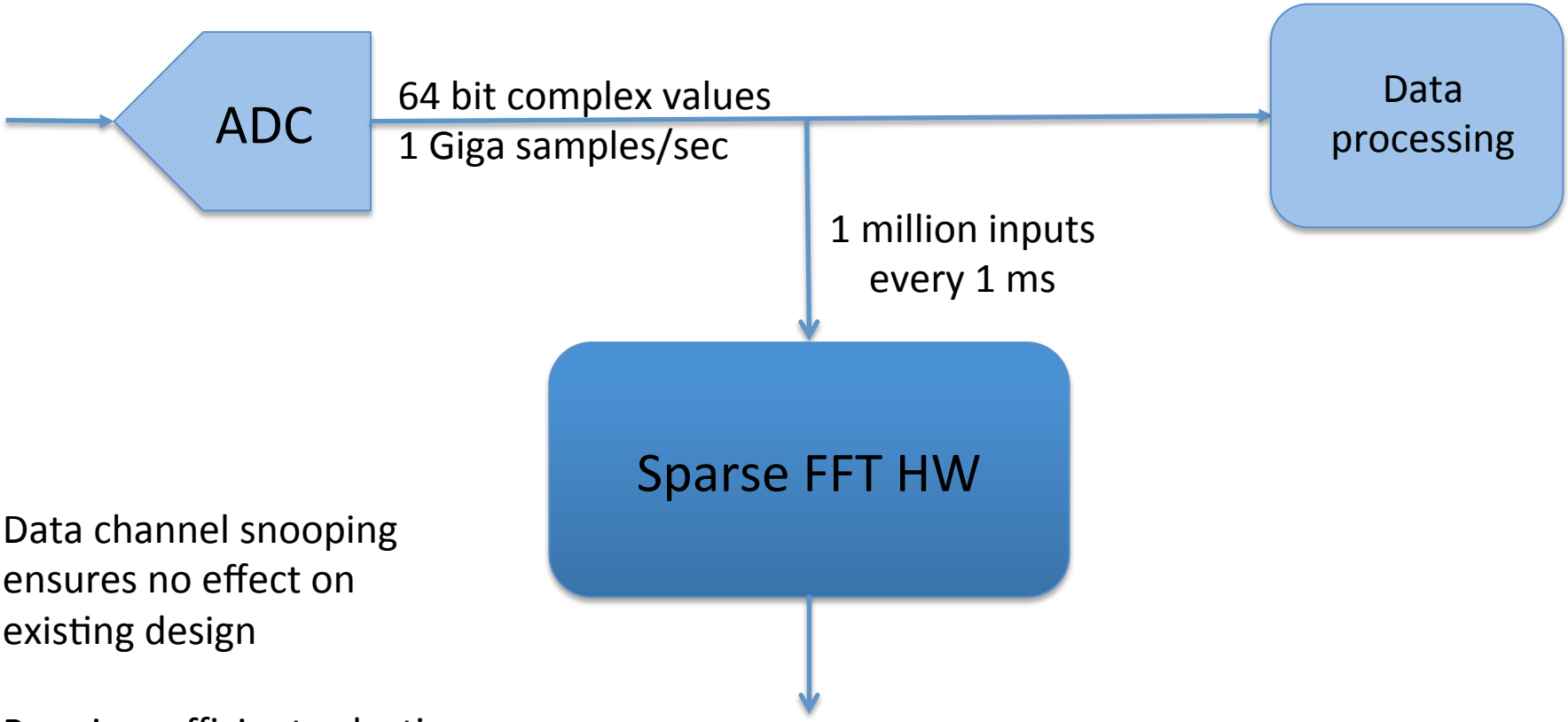
Implementation Blocks



Outline

- Introduction to the algorithm
- Motivation for hardware
- Implementation
  - Block 1: Input time slices

Sparse FFT side-channel



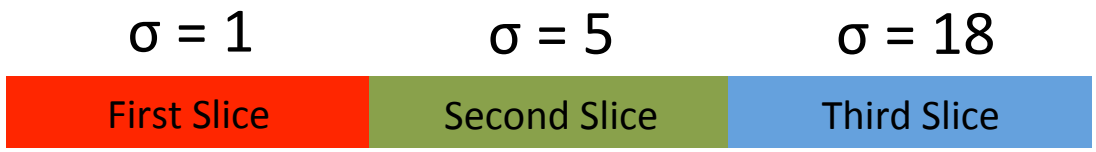
Data channel snooping ensures no effect on existing design

Requires efficient selection of the required data

Generating slices from Input Datastream

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

x[σn]
n: 1 - 8



Input indices for randomized sigma

Slice	Val1	Val2	Val3	..	..	Val8192
Slice1	0	866253	683930			843271
Slice2	0	98001	196002			677779
..	..	..	..	..	..	
Slice8	0	253835	507670			76545

- Each slice has 8192 (W) values from the input stream
- The table gives the input stream index for each value in the slice
- All these values are pre-computed

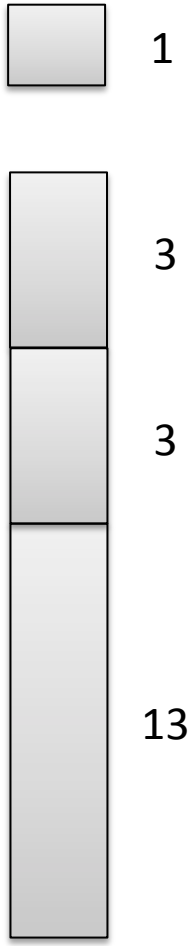
Challenge: Storage

- Each of the million input time samples
  - Can be at any place out of 8192 positions in a slice
  - 13 bits of position address
  - In any of the 8 slices: 3 bit slice address
  - 1 million * (13b + 3b) = **16 Mb**
  - For larger no of slices, increases proportionally
  - Not feasible, already larger than total BRAM storage
    - 832 16kb Blocks : 13.3 Mb

Storage: Data look up table

- Part 1: For all input time samples
  - Store a single bit denoting whether it is used
  - 1 million * 1b = 1 Mb
- Part 2: For each value that is used
  - Store the number of times it is repeated in all slices
  - Max no of repetitions = No of slices - 1 = 7 : 3 bits
  - Max no of used values = 8192 * 8
  - 8192 * 8 * 3b = 0.2 Mb
- Part 3: For each input value that is used
  - Store its 16 bit address (3b: Slice, 13b: position)
  - Max: 8192 * 8 * 16b = 1 Mb

- Total storage = 1 Mb + 0.2 Mb + 1 Mb = **2.2 Mb**

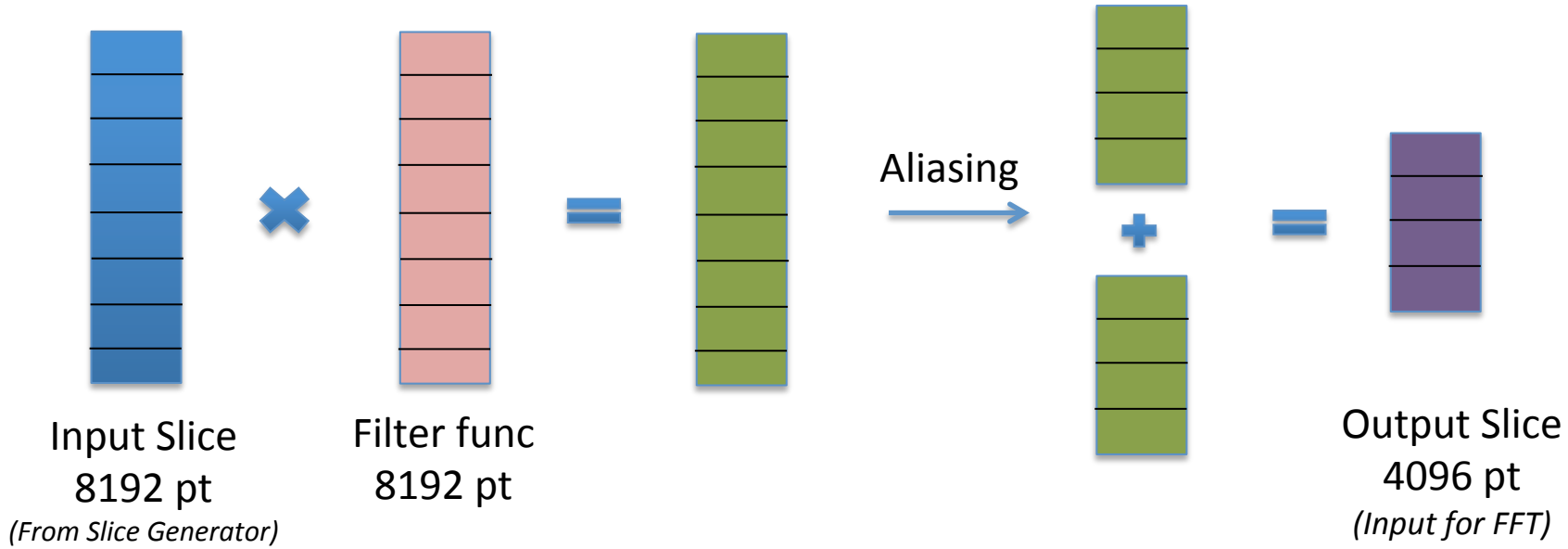


Outline

- Introduction to the algorithm
- Motivation for hardware
- Implementation
  - Block 1: Input time slices
  - Block 2: Filtering & Aliasing

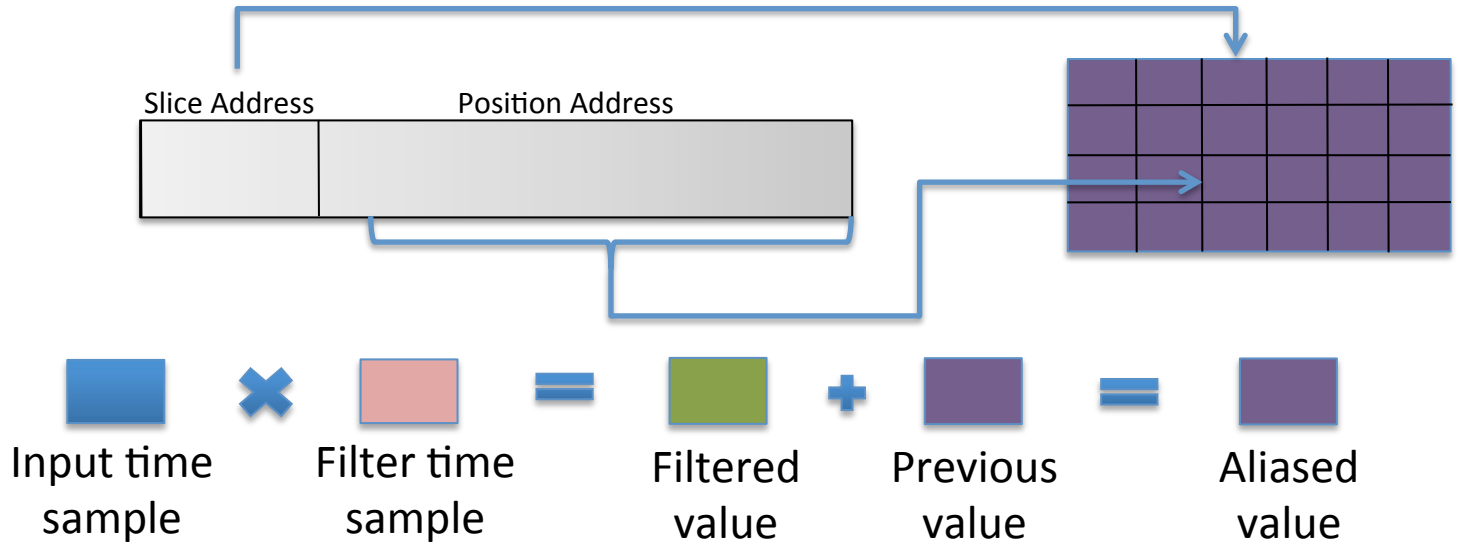
Filtering & Aliasing

- Filter function
  - Fixed at design time
  - Stored as a look up table
  - Multiply with each 8192 point slice
- Data slice storage: 8192 * 8 * 64b = 4.16 Mb
- Aliasing: Add top half of product to bottom half



Combining operations

- For each used value in input datastream
  - Multiply with appropriate filter value using position address
  - Load previously stored slice value
    - Use Slice address & 12 LSB bits of position address
    - MSB of position address determines first/second half
  - Add previously stored slice value to filtered value and store result back
  - Initialize slice values to 0, use only half the slice storage locations



Reduction of storage

- Needed storage for filtered & aliased slices
  - 4096 * 8 * 64b = 2.08 Mb
- Reduced from:
  - 8192 * 8 * 64b * 2.5 = 10.4 Mb
- Total storage for 1^{st}, 2^{nd} blocks
  - 2.2 Mb + 2.08 Mb = 4.28 Mb
  - Fits within BRAM!

Outline

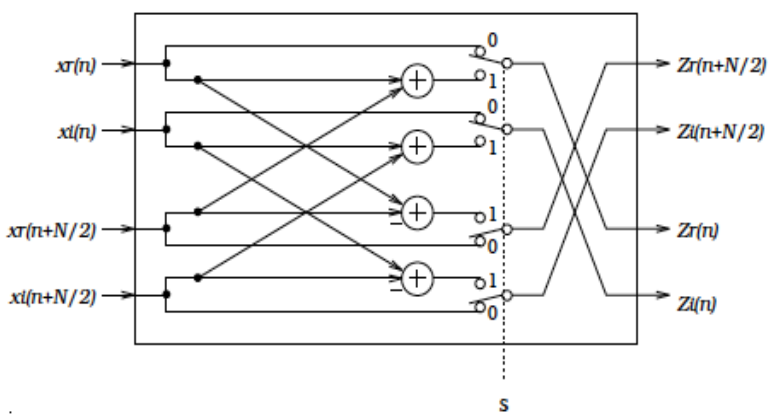
- Introduction to the algorithm
- Motivation for hardware
- Implementation
  - Block 1: Input time slices
  - Block 2: Filtering & Aliasing
  - Block 3: FFT

FFT Computation

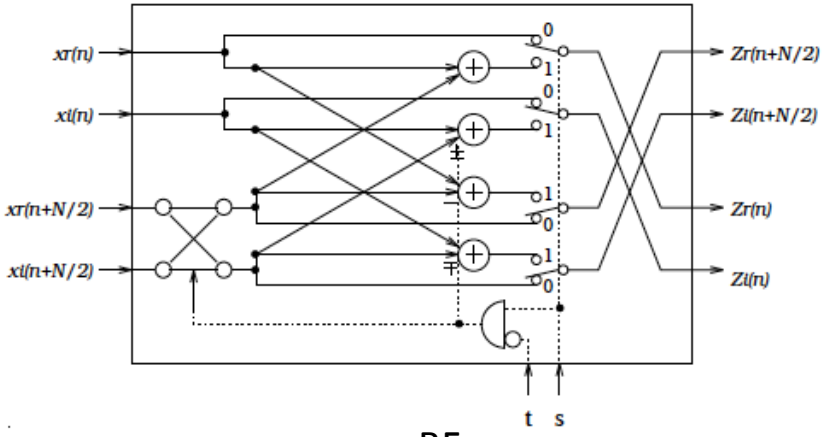
- Larger the FFT size, greater the applicability of HW
- High storage & computational costs
- Would like to go up to 4096 point FFT
  - With minimum possible resource usage
  - Conventional folded architectures not suitable for FPGAs

- Ref: *A new approach to pipeline FFT processor*, Shousheng He and Mats Torkelson, Intl Parallel Processing Symposium 1996.
  - Proposed HW oriented streaming 256 pt FFT
    - Radix-4 multiplicative complexity (fewer multiplications) using simpler radix-2 structures
  - Radix 2^2 Single-path Delay Feedback architecture

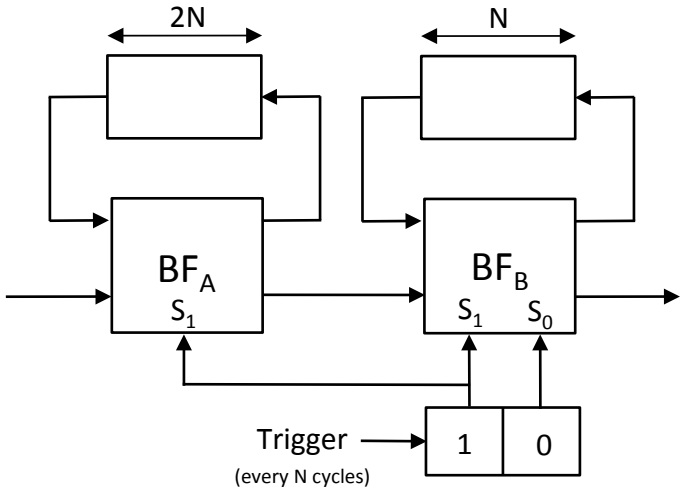
Streaming FFT - stage

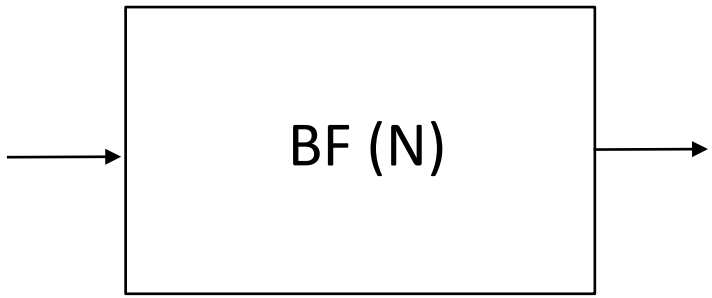


BF_A

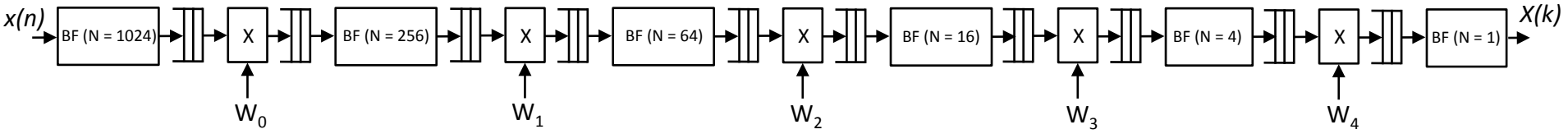


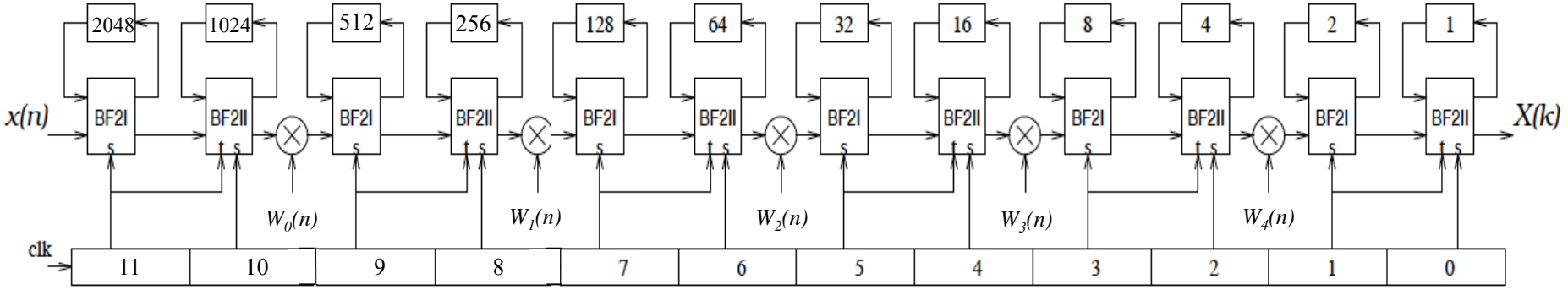
BF_B





4096 point Streaming FFT





Generation of Twiddle Factors

$$W(n) = e^{-(2\pi jn/N)}$$
$$n = k_3(k_1+2k_2)$$

Twiddle Position	Formulation	Offset
$W_0(n)$	$C_9C_8C_7C_6C_5C_4C_3C_2C_1C_0(C_{11}+2C_{10})$	3072
$W_1(n)$	$4*C_7C_6C_5C_4C_3C_2C_1C_0(C_9+2C_8)$	768
$W_2(n)$	$16*C_5C_4C_3C_2C_1C_0(C_7+2C_6)$	196
$W_3(n)$	$64*C_3C_2C_1C_0(C_5+2C_4)$	48
$W_4(n)$	$256*C_1C_0(C_3+2C_2)$	12

4096 point FFT

- Efficient design for large FFT designs
  - Minimal storage requirement
  - Reduced number of multipliers over in-place FFT
  - Simplified control structures with minimal Muxes
  - Multi-step complex multiplication to break critical path, separate counters
- Pipelined version: designed, tested & synthesized

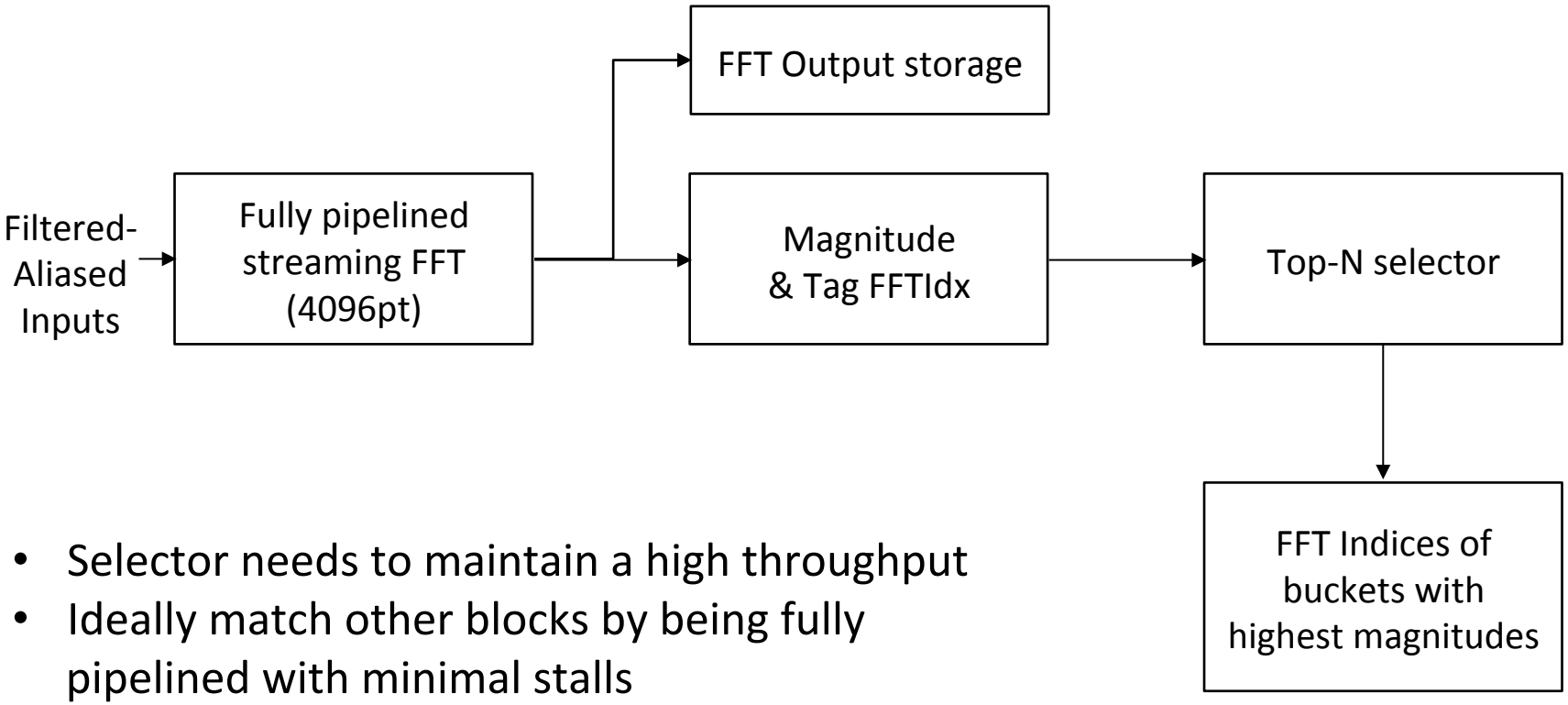
Resource Usage	Register Slices	%	LUT Slices	%	DSP48Es		BRAM	%	Frequency (MHz)	Cycles/ FFT	**FFTs/Sec**
4096pt 3step Cmplx Mult	**8579**	**3.0**	**14892**	**10**	**85**	**11**	**38**	**9**	**79.88**	**4096**	**19,502**

Time required per FFT: 51.3 µs
8 iterations: 410.4 µs

Outline

- Introduction to the algorithm
- Motivation for hardware
- Implementation
  - Block 1: Input time slices
  - Block 2: Filtering & Aliasing
  - Block 3: FFT
  - Block 4: Top N Selector

Pipeline - One element at a time

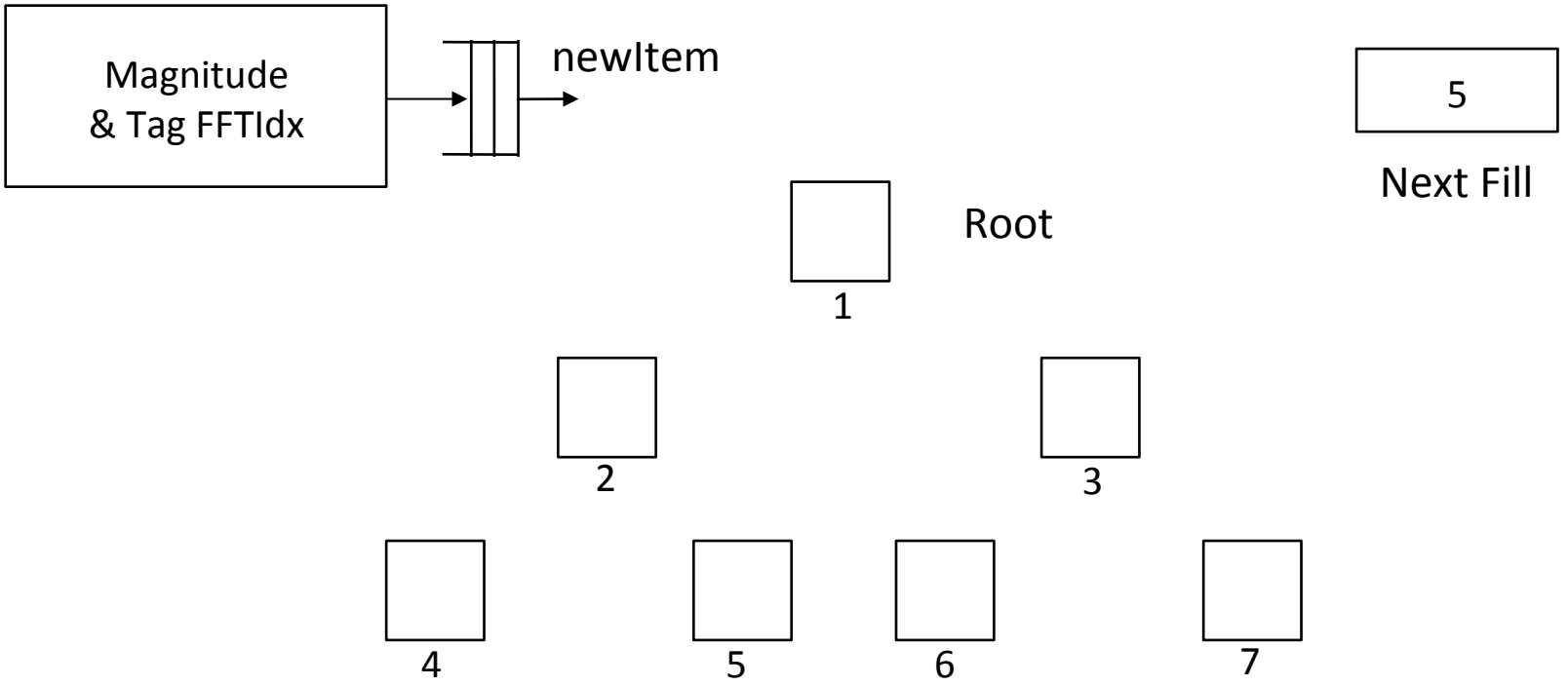


- Selector needs to maintain a high throughput
- Ideally match other blocks by being fully pipelined with minimal stalls

Top N Selector

- Heap priority queue
  - Entries stored at 1,2,3,..,N-1 (N=512)
  - Children of k^{th} node located at 2k & 2k+1
  - Root is minimum of all tree entries
  - Each parent is smaller than its two children

Logical view



If newItem < parent
  swap
If swapped continue checking

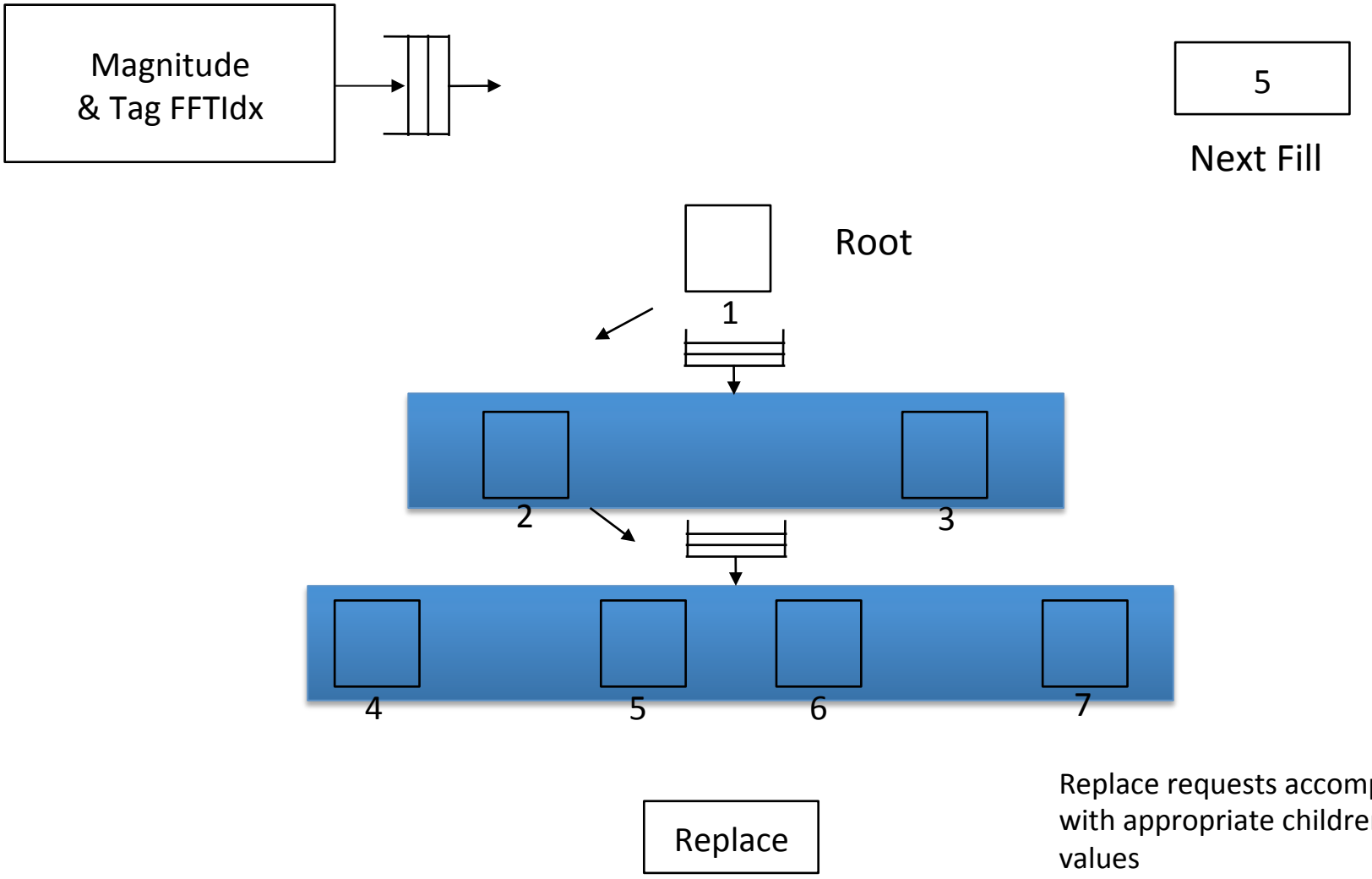
Replace

If newItem > Root
  discard Root
  if newItem == min(newItem,2,3)
    root = newItem; done
  else swap with min
If swapped continue checking

Issues with Initial Implementation

- Two phases
  - Insert into unfilled tree at next empty cell
    - If Parent is larger, swap and check with new parent until false or root
  - Replace root of filled tree if larger
    - Read both children to find new minimum as new parent
    - New child checks with its own children until last level
- Storage not an issue: Only 7% Slice registers
- 54% LUTs used – Gigantic Muxes reading and writing throughout the array
- Pipelining not possible as the two phases move in different directions

Pipelined design



Replace requests accompanied with appropriate children values

Fully Pipelined Heap queue

- Insert from top using address of nextEmpty cell
  - If newItem > root, use address to select appropriate child for comparison
- Insertions can be started every cycle
  - 511 insertions done for every slice at the start to fill up the heap
- Once heap filled, newItems lead to replacements only if greater than root
  - Can be checked every cycle
- Replacements can lead to bubbles (stall one cycle) only if:
  - Previous cycle, newItem > root but not the new minimum
  - And, this cycle again the newItem > root, as check for minimum requires the previous replacement to settle
  - Number of bubbles on average very low for randomized input

Pipelined design

- Satisfies performance requirement
  - Top-N selector operation starts immediately as the first FFT slice output is available, element by element.
  - Ends operation after 4096 + number of bubble cycles later
- Satisfies area requirements
  - 7% Registers and 25% LUTs
  - Dramatic reduction in Muxes
  - Further optimization possible later using other storage options

Outline

- Introduction to the algorithm
- Motivation for hardware
- Implementation
  - Block 1: Input time slices
  - Block 2: Filtering & Aliasing
  - Block 3: FFT
  - Block 4: Top N Selector
- Conclusion

Conclusion

- Sparse FFT algorithm an opportunity to process Big Data in real time
- HW implementation needed to improve throughput, power, mobile apps
- Choice of parameters impacts resource usage and performance
- Demonstrated implementation of 4 major blocks
  - How to meet resource and performance constraints
  - 2 other blocks are currently being implemented
- Applications will determine appropriate parameters for design & in turn architectural choices
- Focus on parameterization to allow variety of designs for varying application requirements