

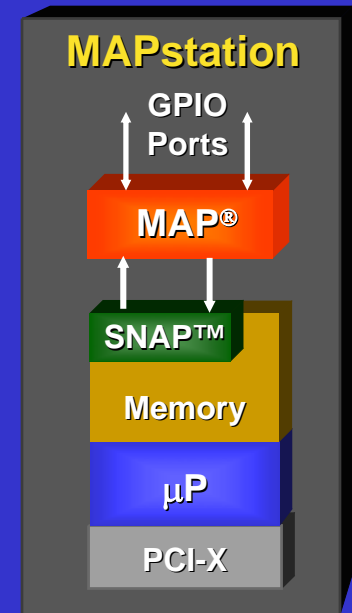
---

# **A Program Transformation Approach to High Performance Embedded Computing using the SRC MAP<sup>®</sup> Compiler**

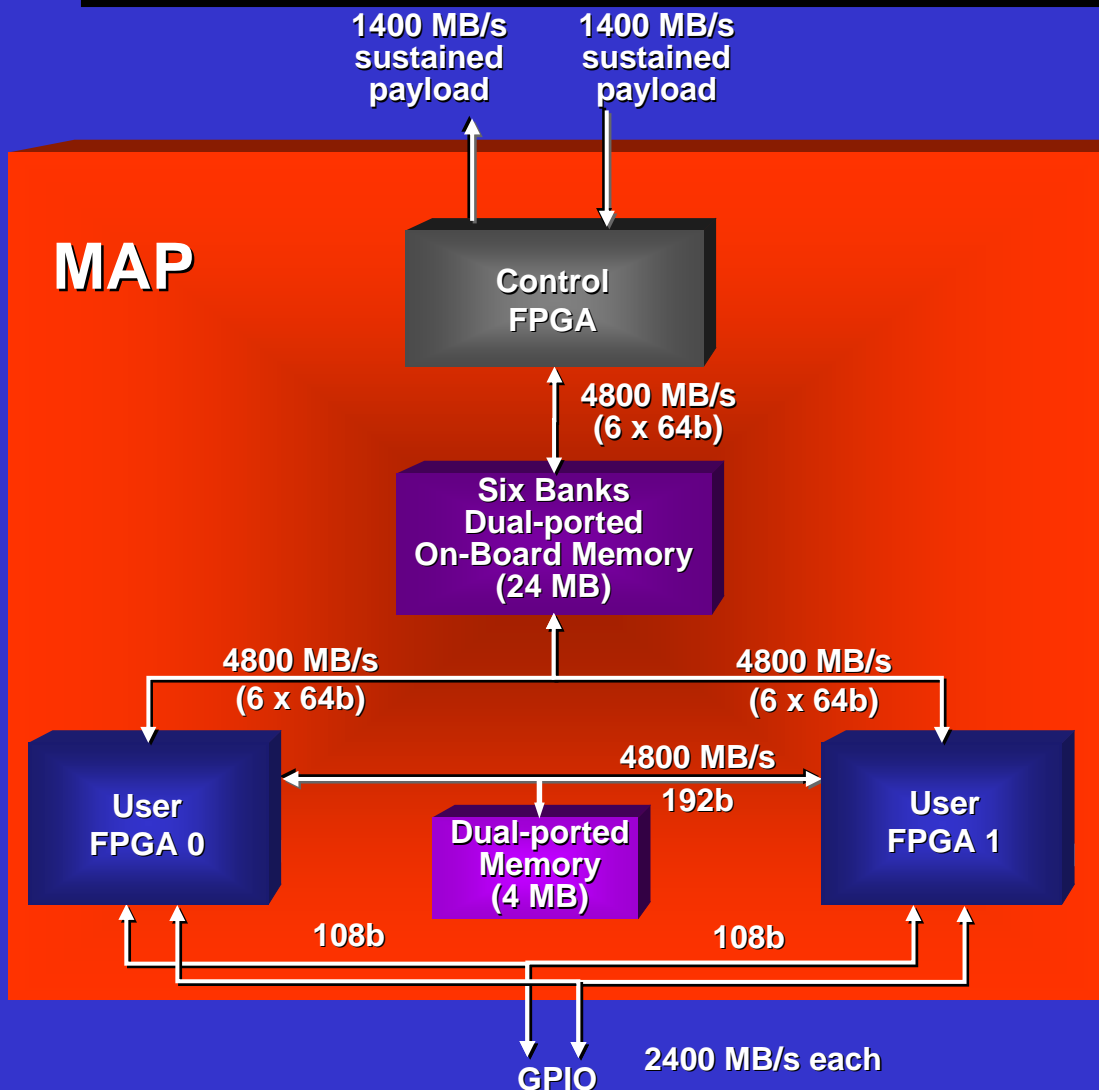
**Wim Bohm, Colorado State University  
and  
Jeff Hammes, SRC Computers, Inc.**

# SRC-6 MAP<sup>®</sup> System

- **SRC-6 MAP**
  - FPGA based High Performance architecture
  - Fortran / C compiler for the whole system
- **One Node:**
  - Microprocessor
  - MAP reconfigurable hardware board
  - SNAP  $\mu$ proc and MAP interconnected via DIM slot
  - GPIO ports allow connection to other MAPs
  - PCI-X can connect to other  $\mu$ procs
- **Multiple configurations / implementations**
  - this talk: MAPstation - one node
- **MAP C Compiler**
  - Compiler generates both  $\mu$ proc and MAP code
  - user partitions  $\mu$ proc, MAP tasks

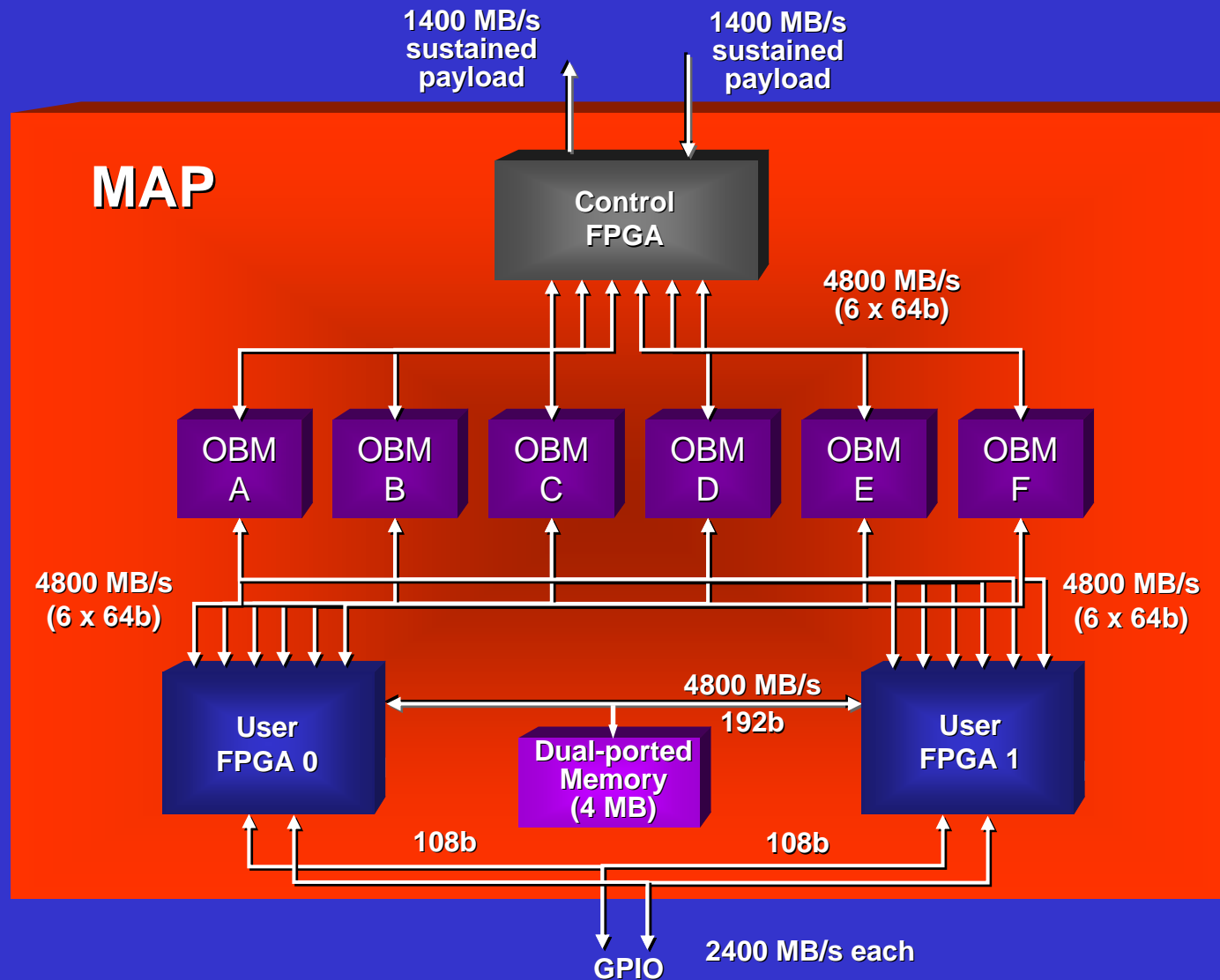


# MAP<sup>®</sup> board architecture



- **Direct Execution Logic (DEL)** made up of one or more User FPGAs
- **Control FPGA** performs off board memory access
- **Multiple banks of On-Board Memory** maximize local memory bandwidth
- **GPIO** ports allow direct MAP to MAP chain connections or direct data input
- **Multiple parallel data transports:**
  - **Distributed SRAM in FPGA**
    - 264 KB @ 844 GB/s
  - **Block SRAM in FPGA**
    - 648 KB @ 260 GB/s
  - **On-Board SRAM Memories**
    - 28 MB @ 9.6 GB/s
  - **Microprocessor Memory**
    - 8 GB @ 1400 MB/s

# MAP Programmers View



# MAP C compiler

---

- **Pure C runs on the MAP !!**
- **MAP C Compiler**
  - Intermediate form: dataflow graph of basic blocks
  - Generated code: circuits
    - Basic blocks in outer loops become special purpose hardware “function units”
    - Basic blocks in inner loop bodies are merged and become pipelined circuits
- **Sequential semantics obeyed**
  - One basic block executed at the time
  - Pipelined inner loops are slowed down to disambiguate read/write conflicts if necessary
  - MAP C compiler identifies (cause of) loop slowdown



# Execution Modes

---

- **DEBUG Mode**

- code runs on workstation
- allows debugging ( **printf** 😊 )
- allows most performance tuning (avoiding loop slow downs)
- user spends most time here

- **Two SIMULATION Modes**

- Dataflow level and Hardware level
- mostly used by compiler / hardware function unit developers
- very fine grain information

- **HARDWARE Mode**

- final stage of code development
- allows performance tuning using timer calls



# Transformational Approach

---

- **Start with pure C code**
- **Partition Code and Data**
  - distribute data over OBMs and Block RAMs
  - distribute code over two FPGAs
    - only one chip at the time can access a particular OBM
    - MPI type communication over the bridge
- **Performance tune (removing inefficiencies)**
  - avoid re-reading of data from OBMs using Delay Queues
  - avoid read / write conflicts in same iteration
  - avoid multiple accesses to a memory in one iteration
  - avoid OBM traffic by fusing loops
- **Today's transformation is tomorrow's compiler optimization**



# How to performance tune: Macros

---

- C code can be extended using macros allowing for program transformations that cannot be expressed straightforwardly in C
- **Macros have semantics unlike C functions**
  - have a **period** (#clocks between inputs)
  - have a **pipeline delay** (#clocks between in and output)
  - MAP C compiler takes care of period and delay
  - can have **state** (kept between macro calls)
  - **two types of macros**
    - **system** provided
      - compiler knows their period and delay
    - **user** provided (written in e.g. Verilog )
      - user needs to provide period and delay





# Two Case Studies

---

## ● Wavelet Versatility Benchmark

- Image processing application (wavelet compression)
- Part of DARPA/ITO ACS (Adaptive Computing Systems) benchmark suite
- **Versatile**: Four phases of **different computational nature**
  - 1: wavelet transform: *window access, multiple outputs*
  - 2: quantization: *sum, min, max reductions*
  - 3: run length encoding: *while loop, irregular output*
  - 4: Huffman encoding: *table lookups*

## ● Gauss Seidel Linear Equation Solver

- Numerical (Floating Point) kernel
- Iterative nature: *non-perfect loop structure*
- Many applications

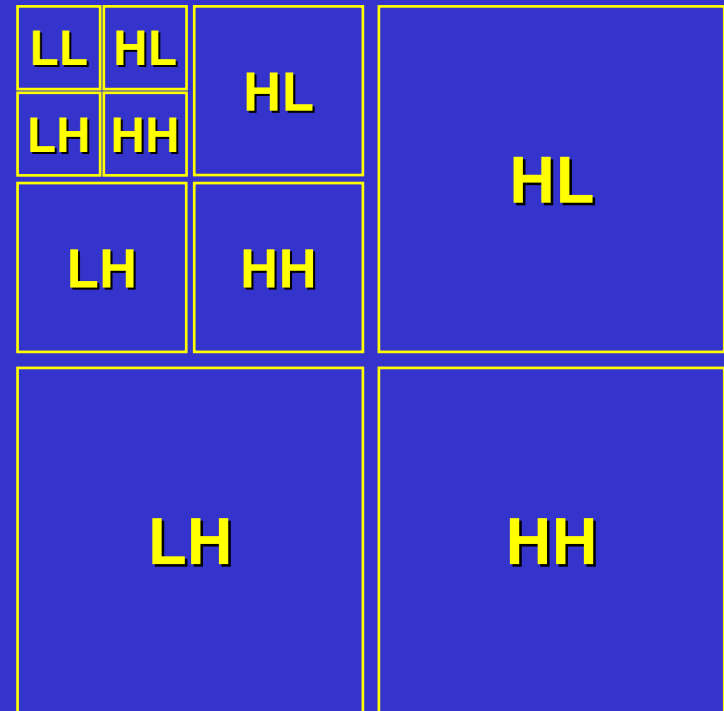
# Wavelet Versatility Benchmark

- **Wavelet transform**

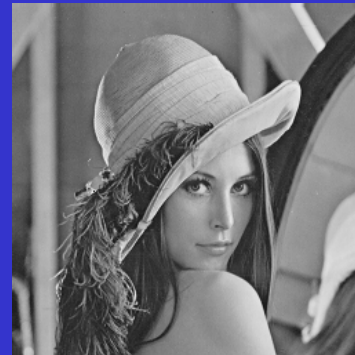
- Applied three times
  - Second and third passes use upper left quadrant of previous pass
- L: Low pass filter (average)
- H: High pass filter (derivative)

- **Wavelet does not compress but enables compression in further stages (many 0s in H)**

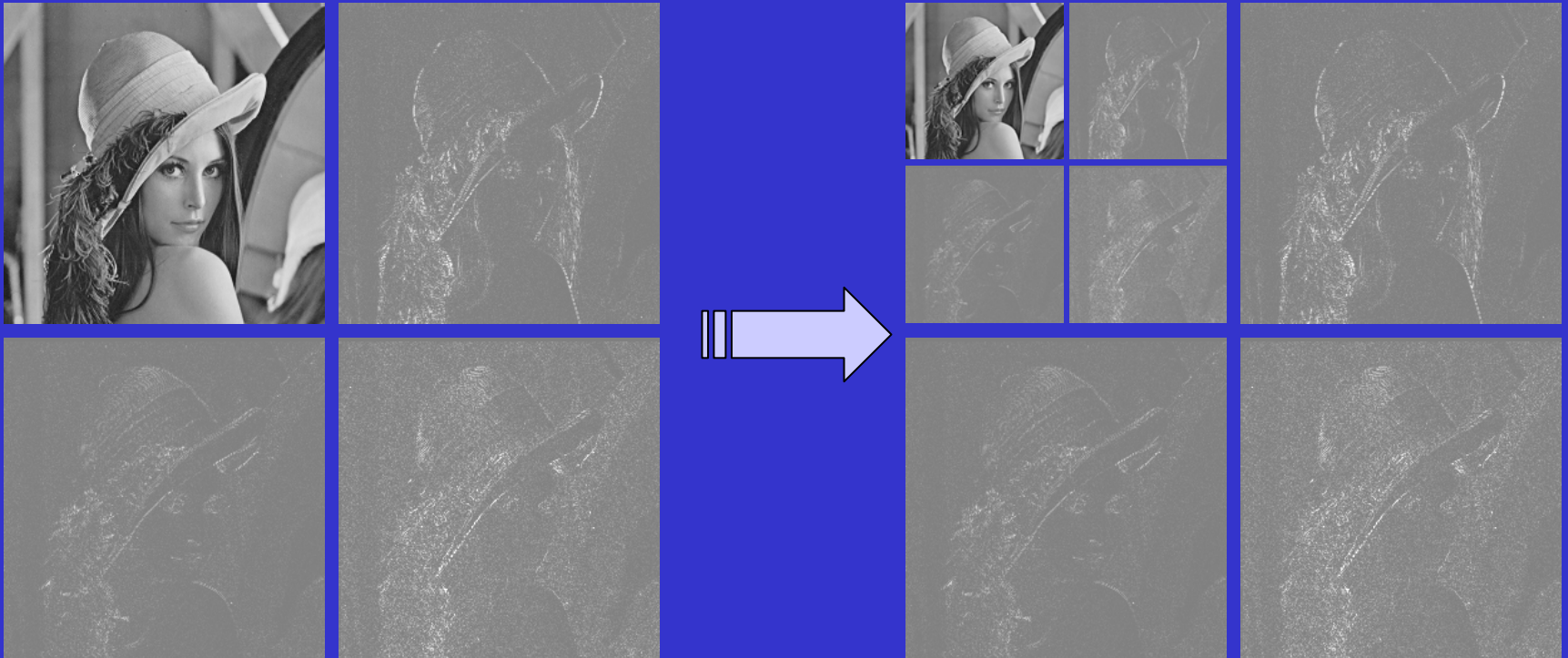
- Quantization
- Run-Length Encoding
- Huffman Encoding



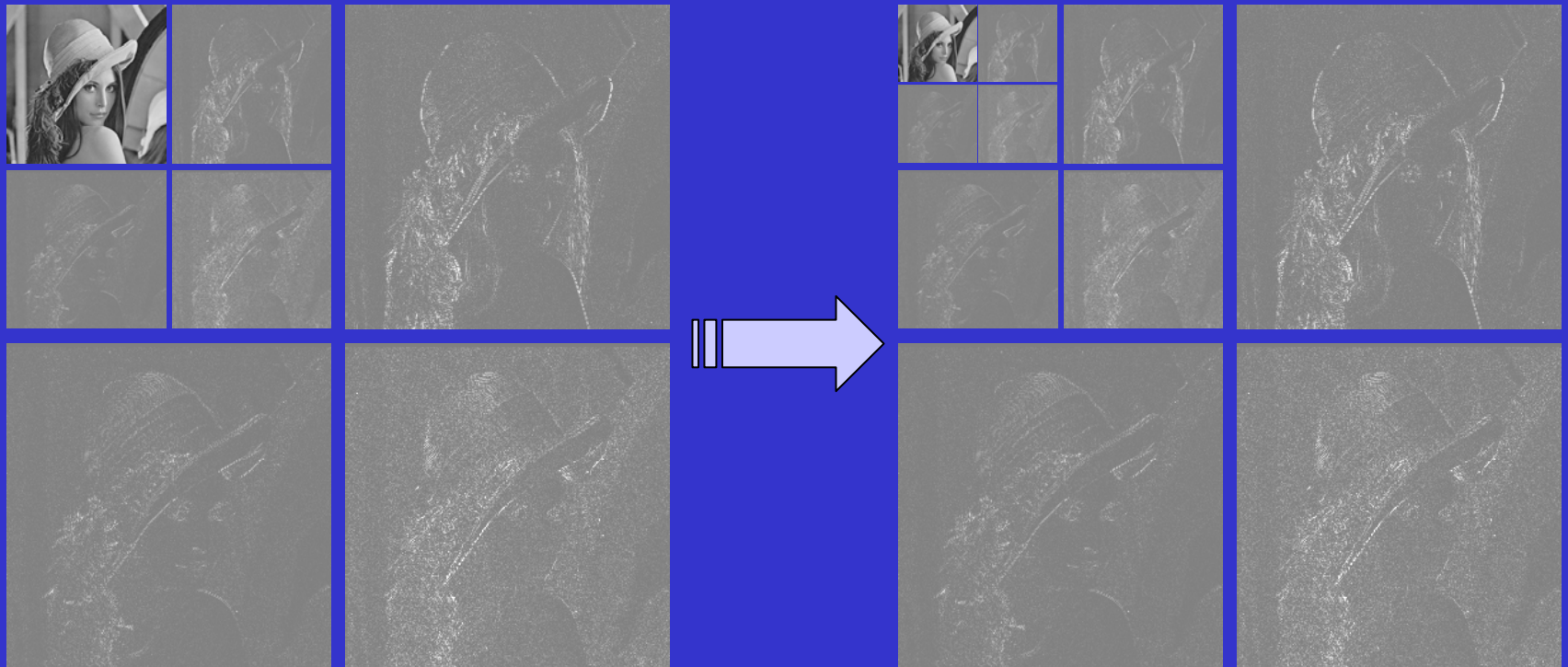
# First wavelet step



# Second wavelet step



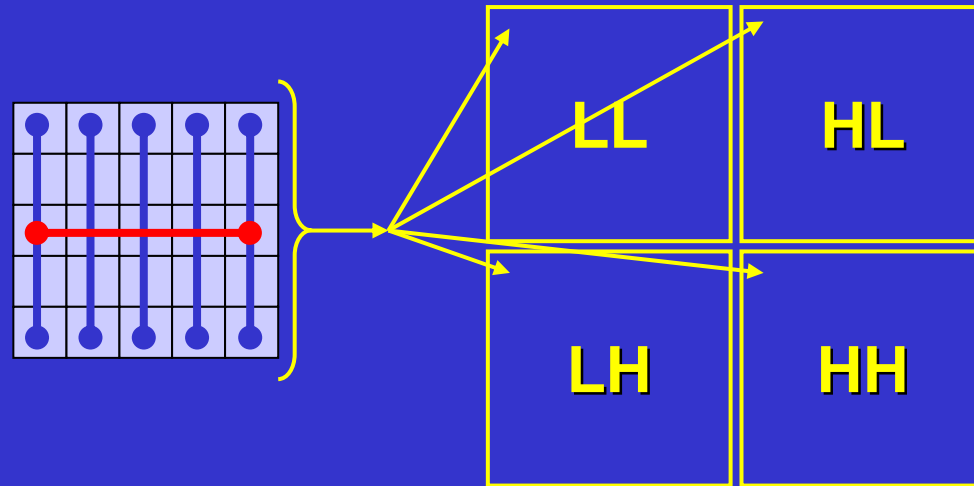
# Final wavelet step



# MAP C Algorithm

**One 5x5 window stepping by 2 in both directions**

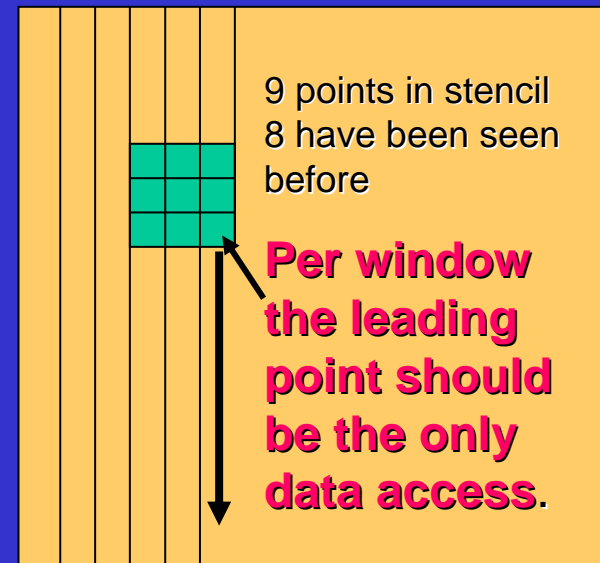
- Computes LL, LH, HL, and HH simultaneously



- **Inefficiency:** naive first implementation re-accesses overlapping image elements

# Efficient Window Access

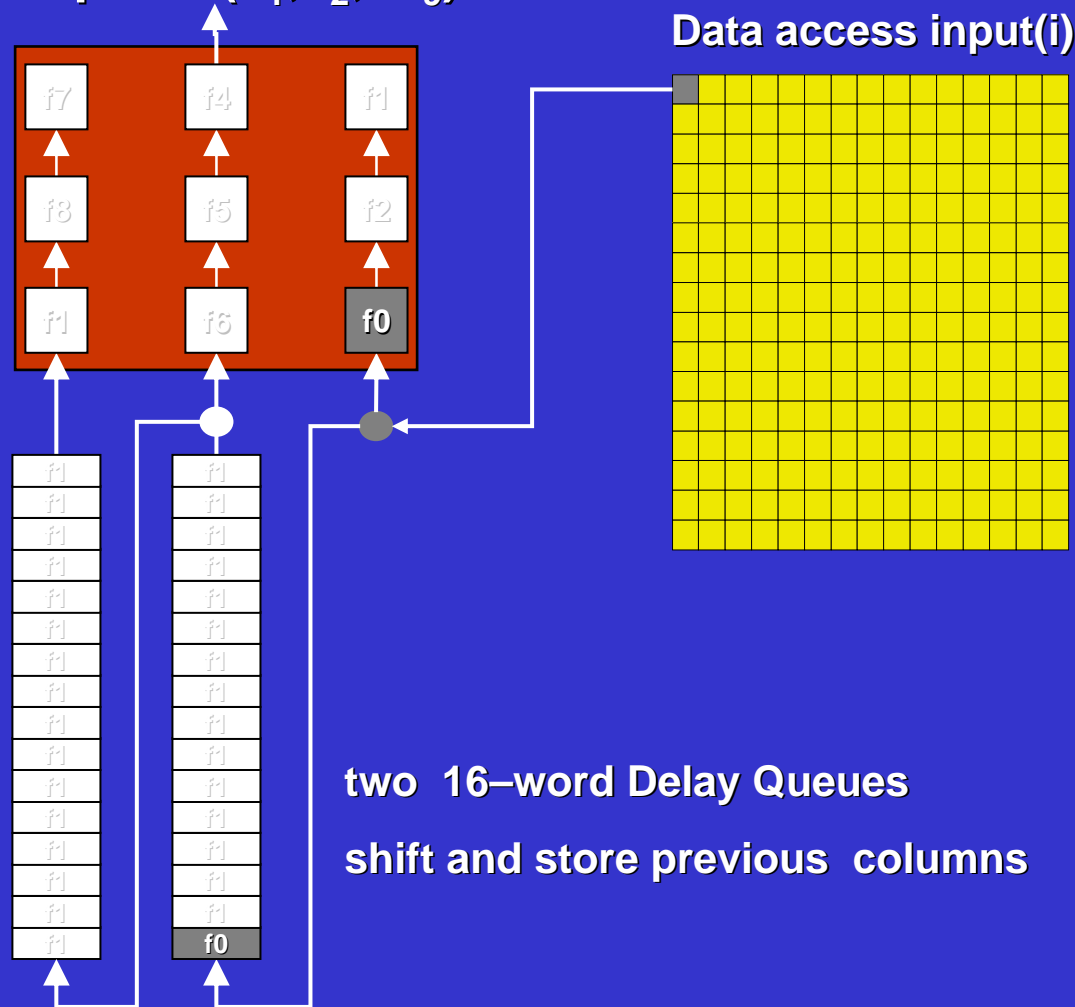
- **Keep data on chip using Delay Queues**
  - E.g. 16 deep (using efficient hardware SLR16 shifters)
- **Simplified example:**
  - 3x3 window
    - stepping 1 by 1
    - in column major order
  - image 16 deep
    - general case divides the Image in 16 deep strips



input array traversal

# Delay Queues 1

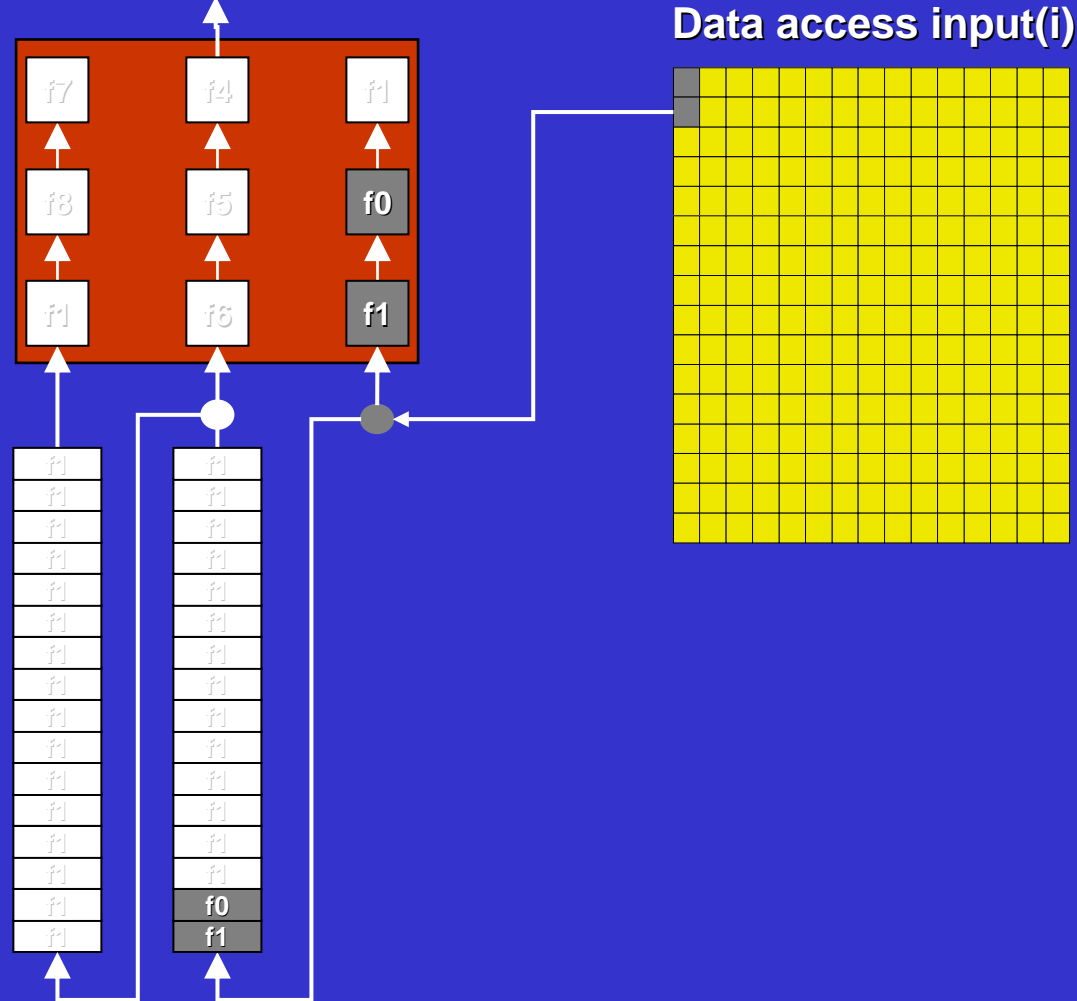
Compute  $f(x_1, x_2, \dots, x_9)$





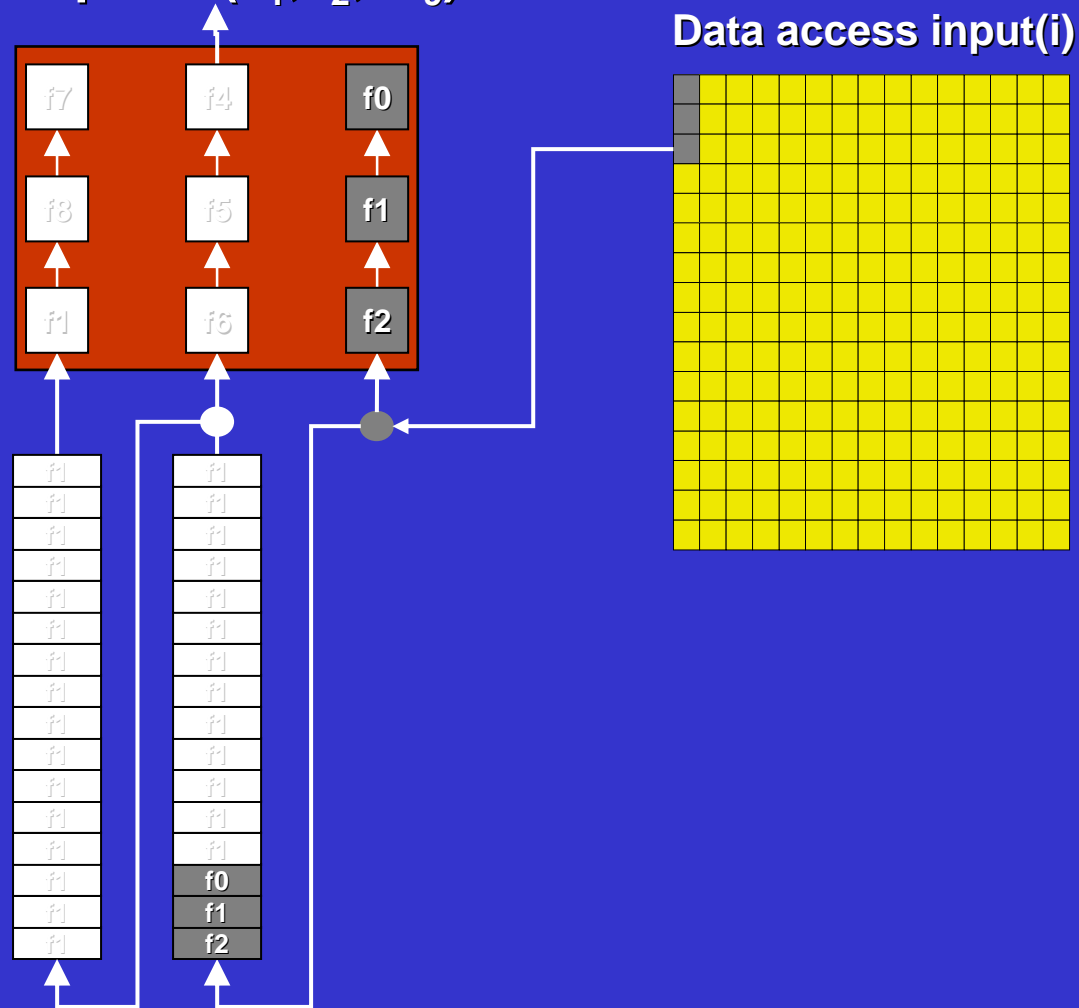
# Delay Queues 2

Compute  $f(x_1, x_2, \dots, x_9)$



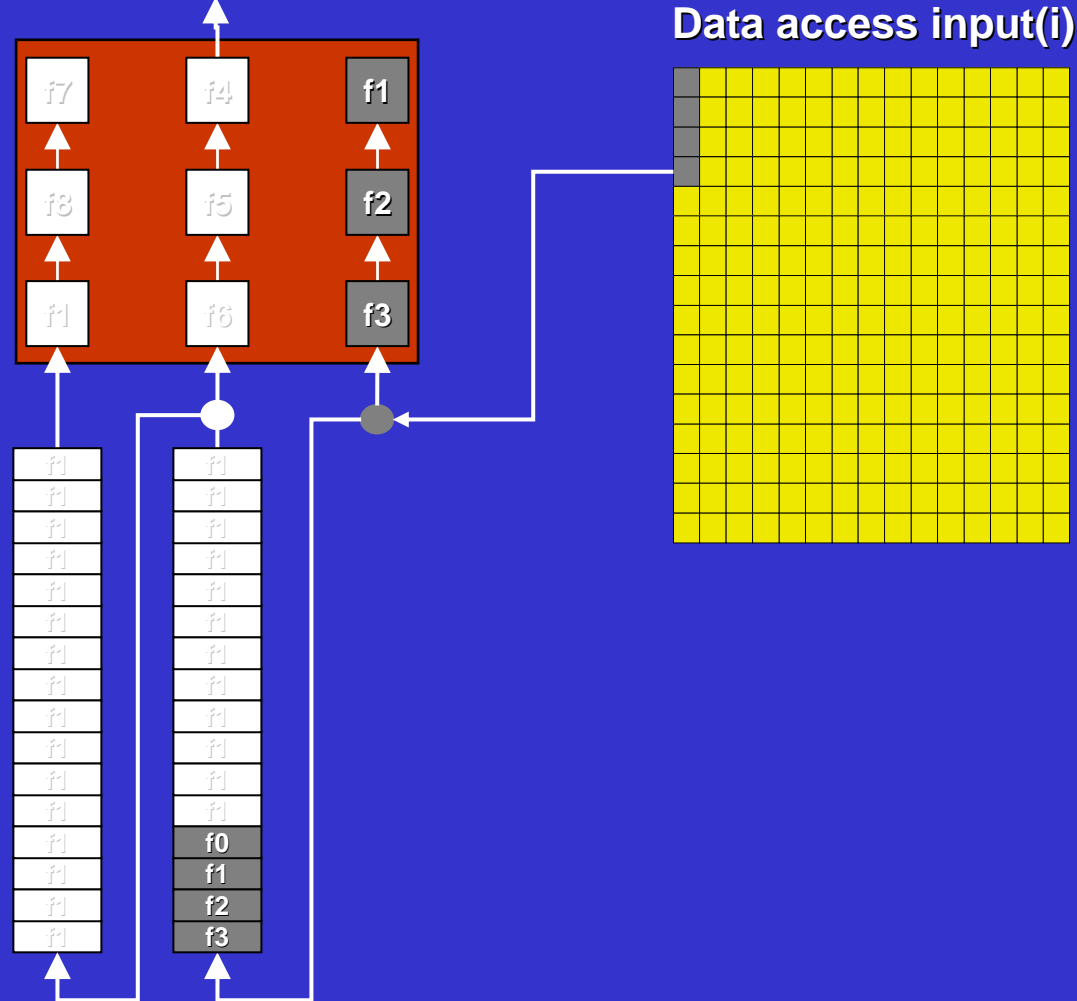
# Delay Queues 3

Compute  $f(x_1, x_2, \dots, x_9)$



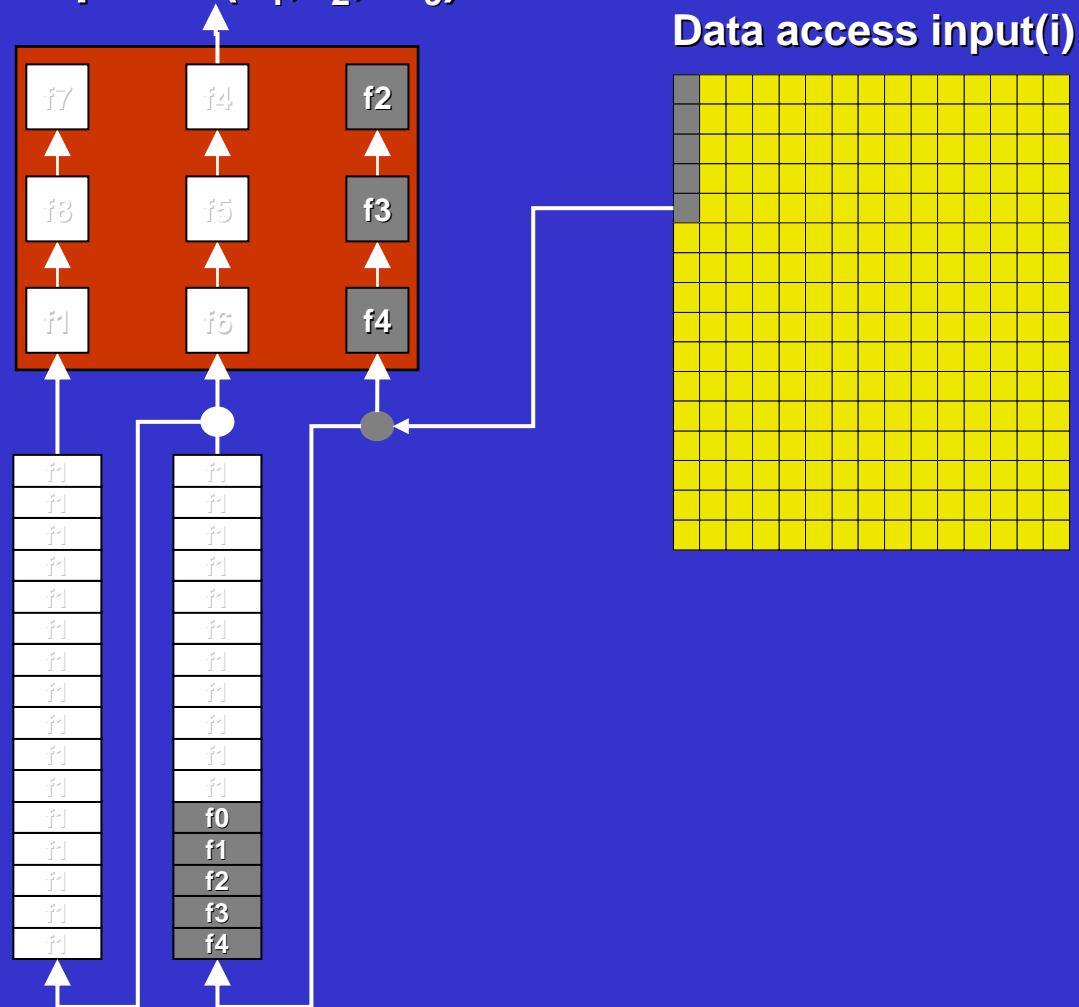
# Delay Queues 4

Compute  $f(x_1, x_2, \dots, x_9)$



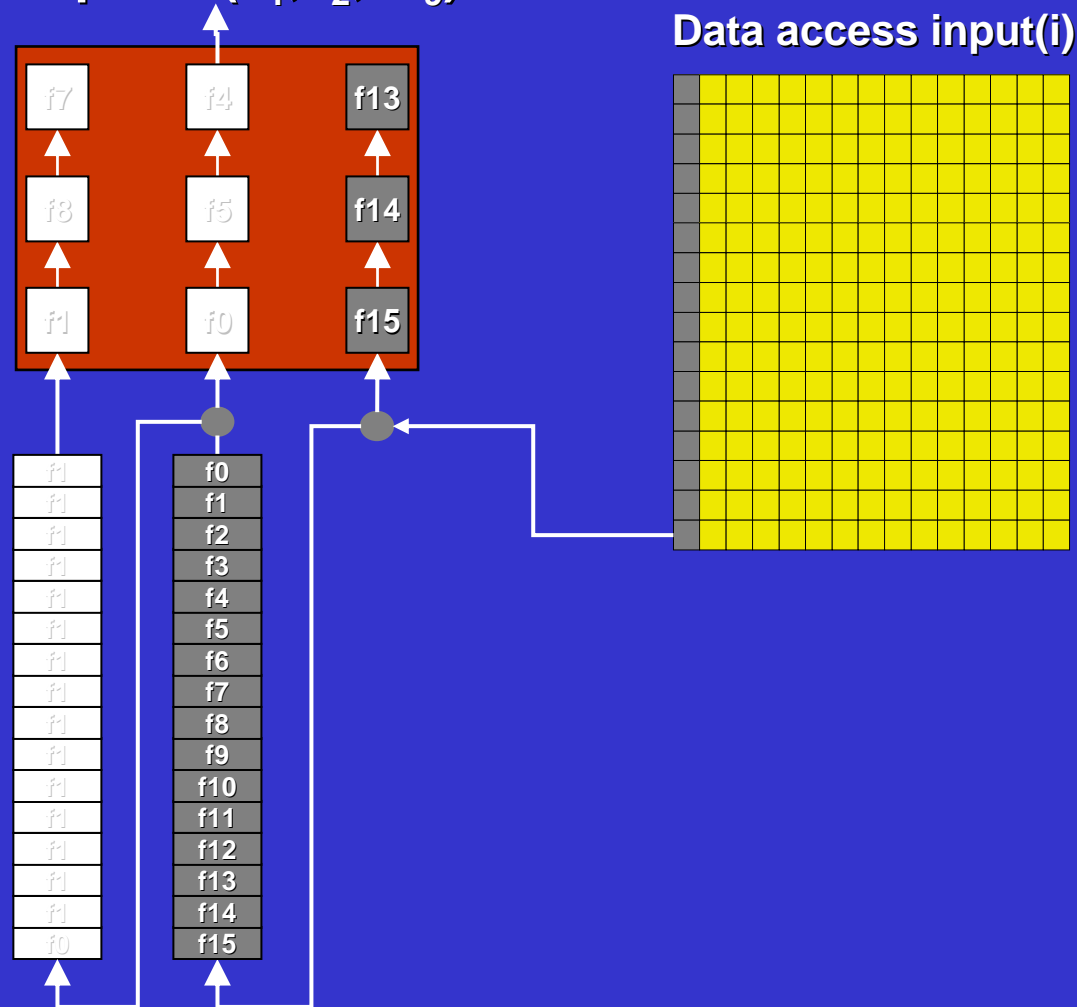
# Delay Queues 5

Compute  $f(x_1, x_2, \dots, x_9)$



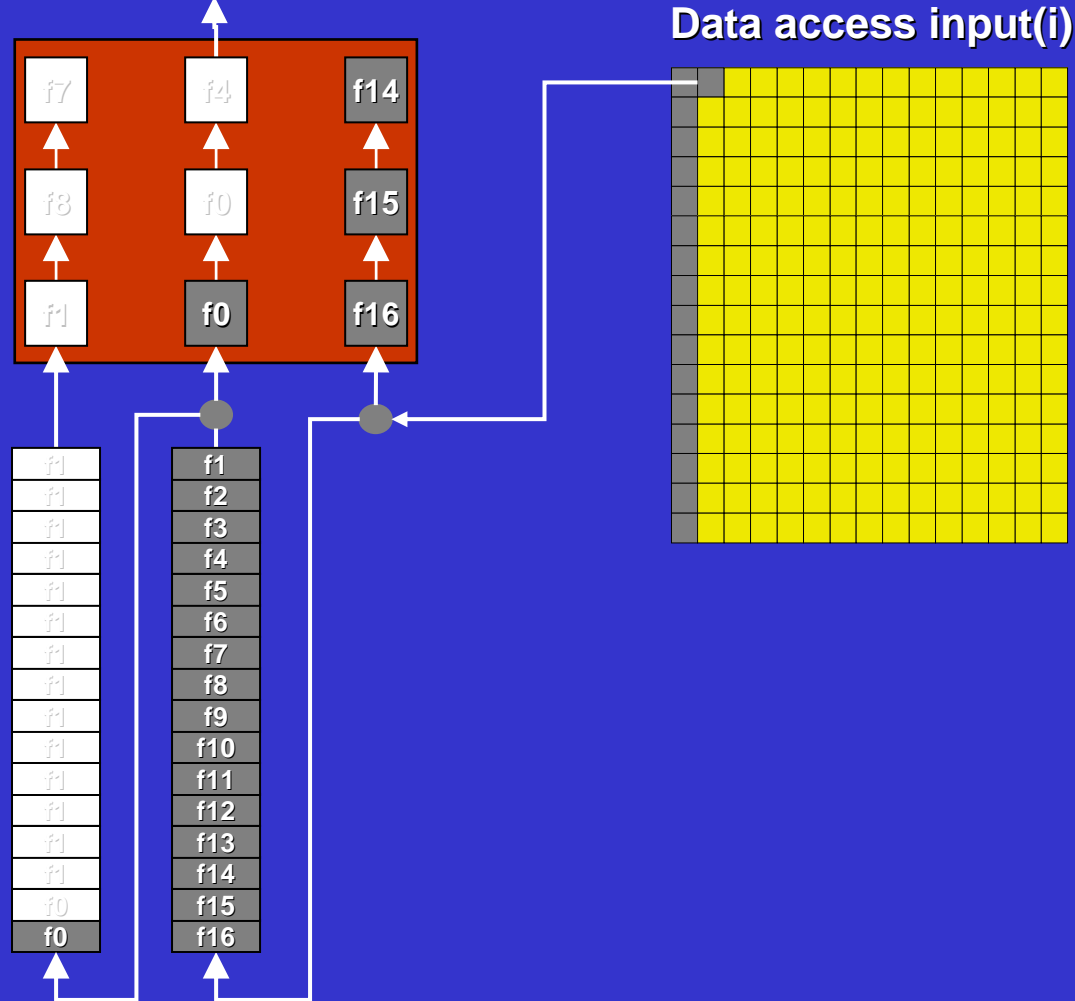
# ... Delay Queues 16

Compute  $f(x_1, x_2, \dots, x_9)$



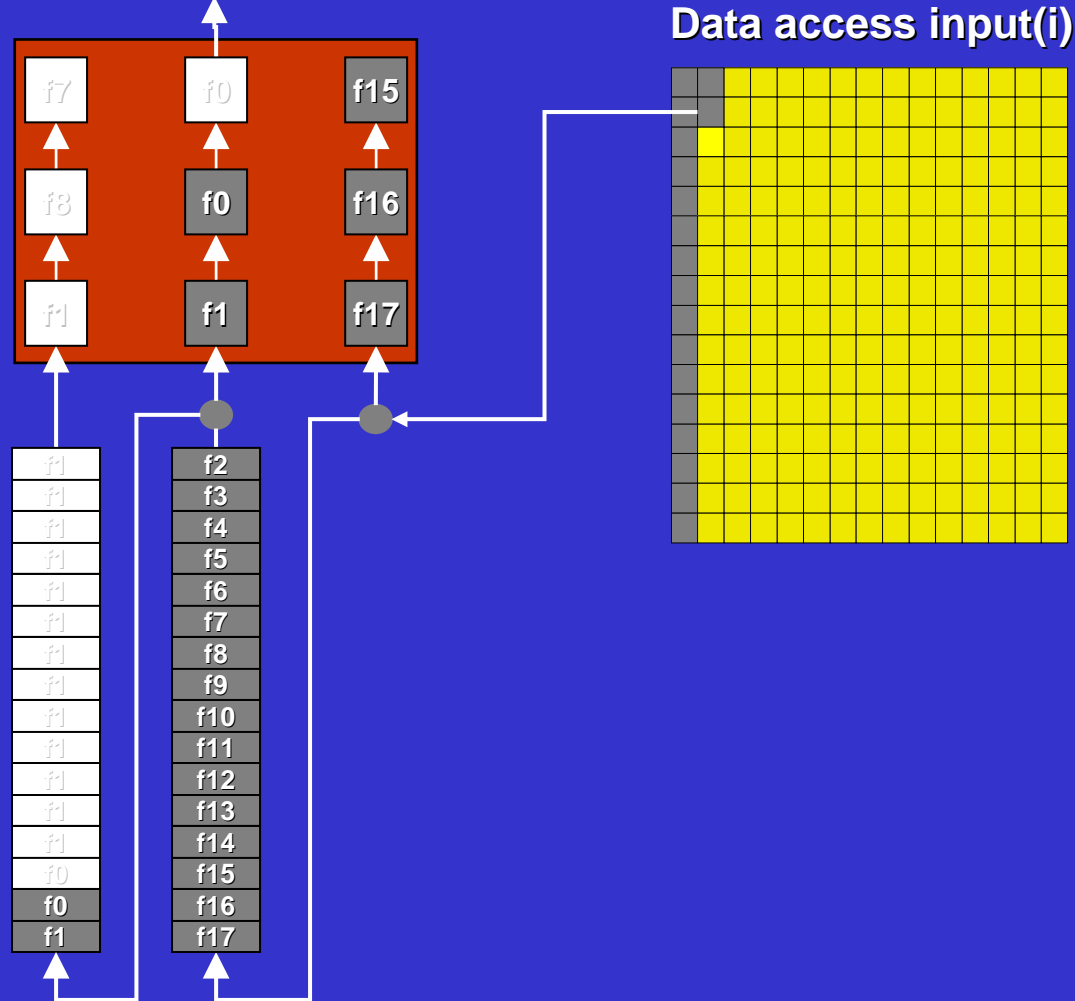
# Delay Queues 17

Compute  $f(x_1, x_2, \dots, x_9)$



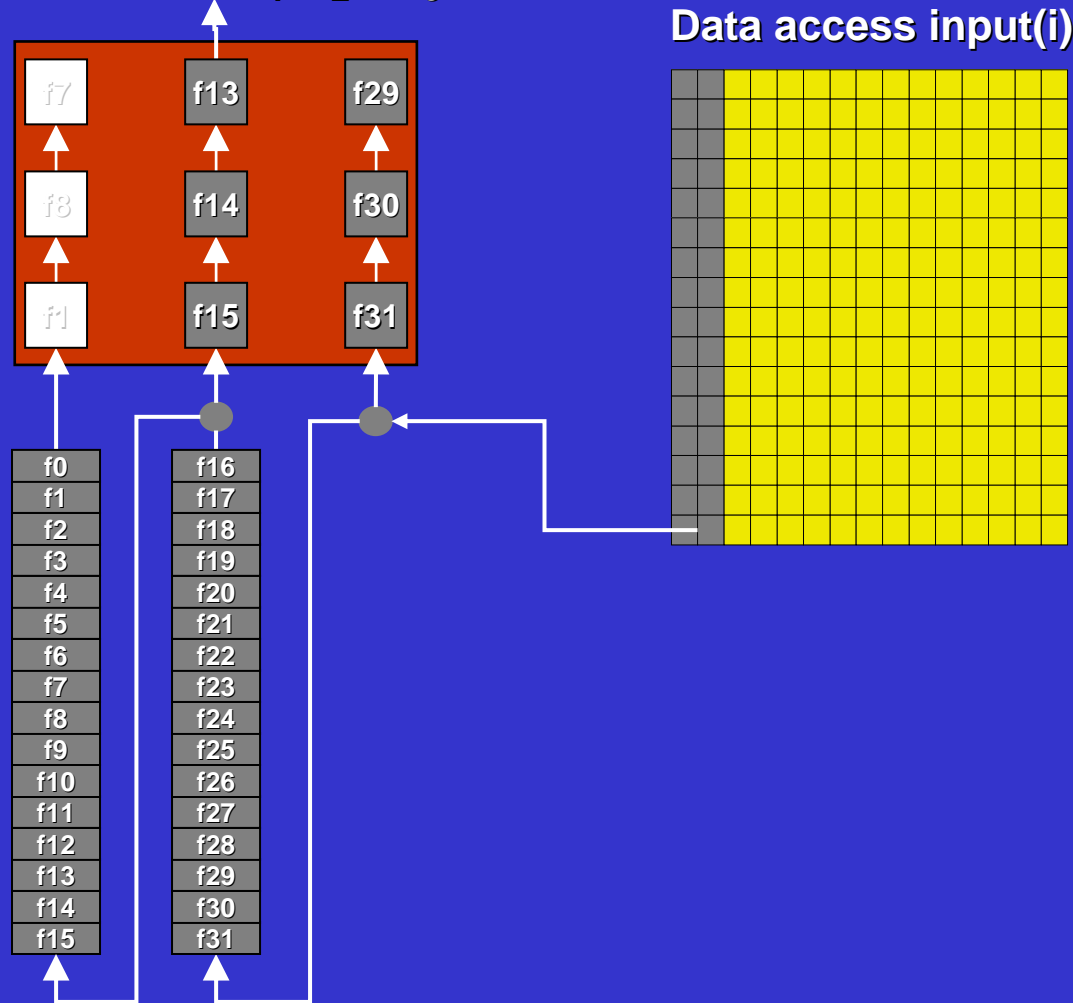
# Delay Queues 18

Compute  $f(x_1, x_2, \dots, x_9)$



# ...Delay Queues 32

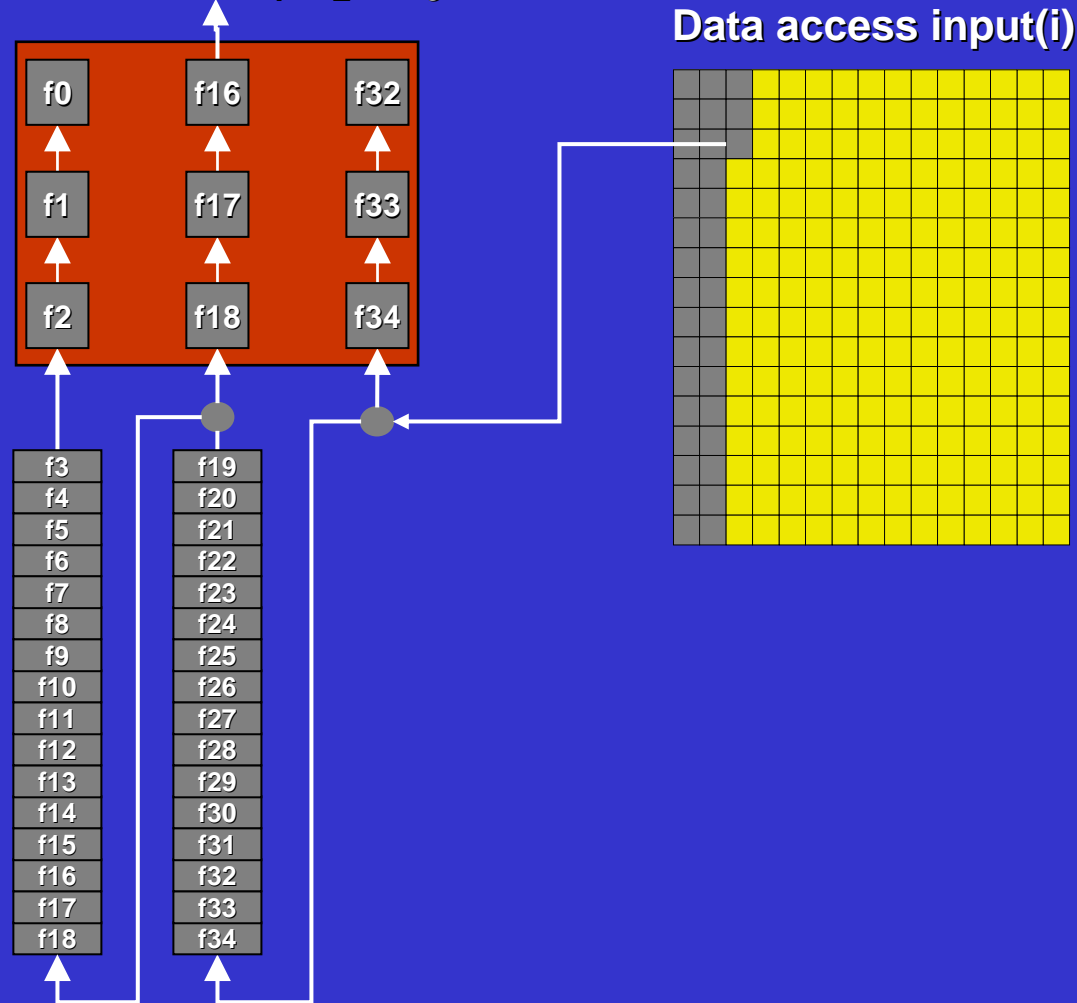
Compute  $f(x_1, x_2, \dots, x_9)$





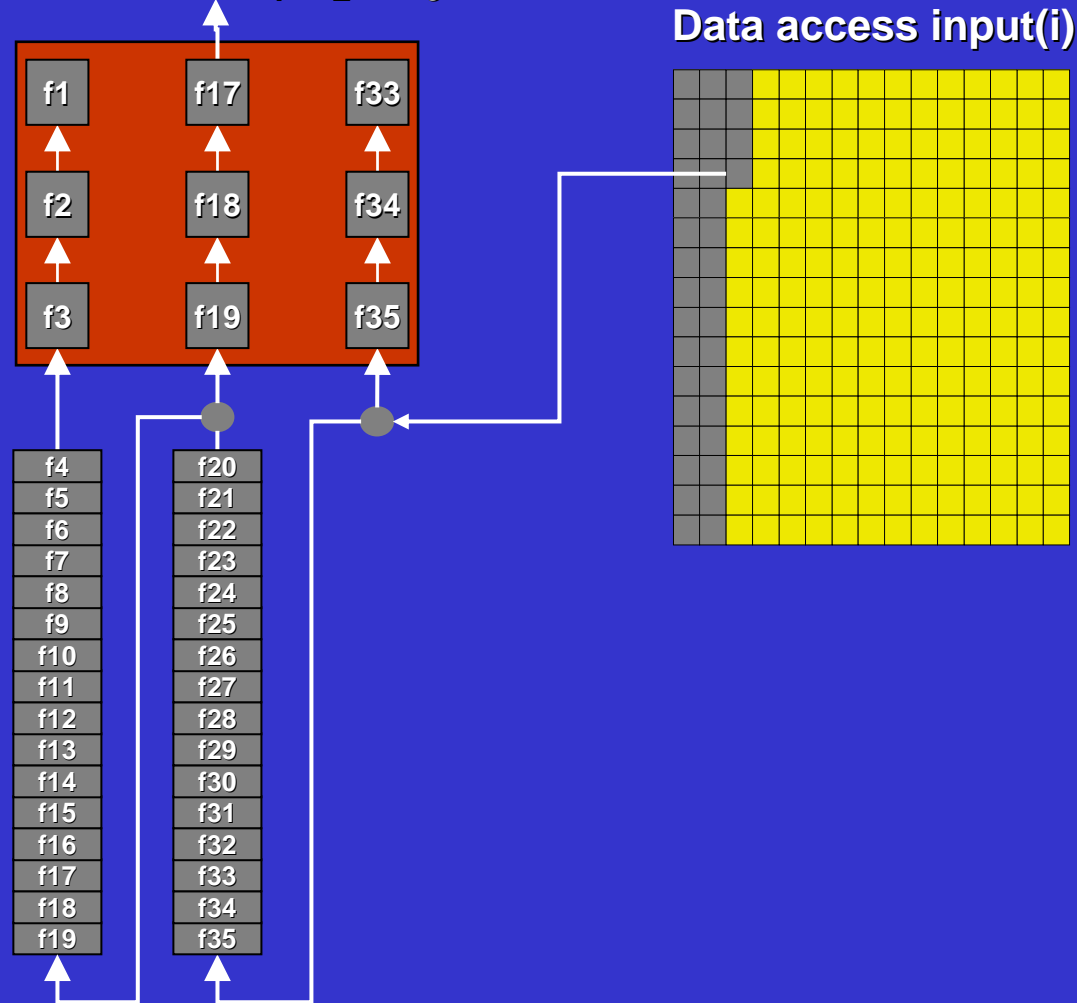
# ...Delay Queues 35

Compute  $f(x_1, x_2, \dots, x_9)$



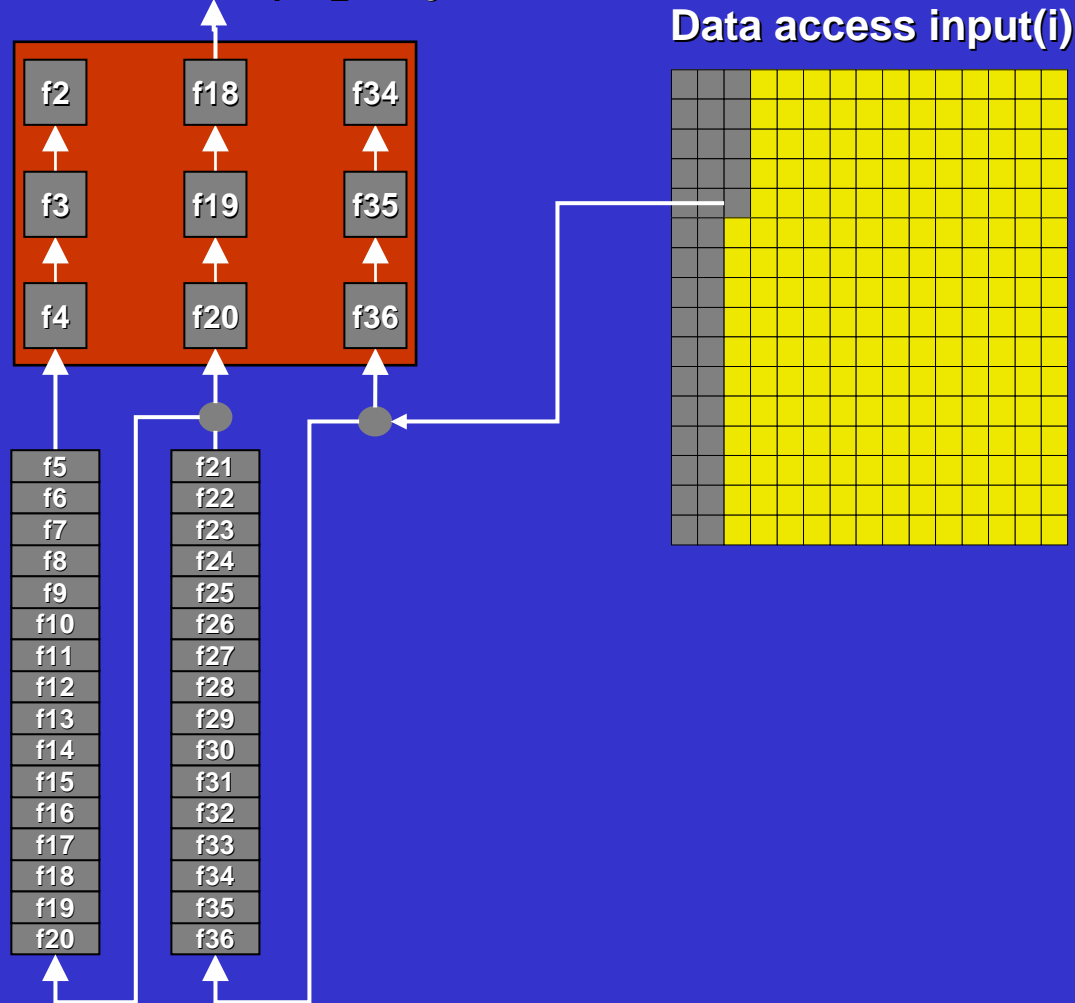
# Delay Queues 36

Compute  $f(x_1, x_2, \dots, x_9)$



# Delay Queues 37

Compute  $f(x_1, x_2, \dots, x_9)$



# Delay Queues: Performance

## *3x3 window access code*

512 x 512 pixel image

Routine Style	Number of clocks	Comment
Straight Window	2,376,617	close to 9 clocks per iteration 2,340,900: the difference is pipeline prime effect
Delay Queue	279,999	close to 1 clock per iteration 262144: theoretical limit
<i>FPGA timing behavior is very predictable</i>		

# Wavelet Benchmark cont'

---

- **Rest of the code:**

- Quantize each block in 16 bins per block
- Run Length Encode zeroes
  - Occur frequently in derivative blocks
- Huffman Encode

- **Three transformations**

- Fuse the three loops avoiding OBM traffic
- Use accumulator macros to avoid R / W conflicts
  - (see Gauss Seidel case study)
- Task parallelize the complete code over two FPGAs



# Versatility Benchmark: Performance

---

- 512x512 image
- Bit true results as compared to reference code
- Full implementation: All phases run on FPGAs
- Reference code compiled using Intel C compiler  
executed on 2.8 GHz Pentium IV: 76.0 milli-sec
- MAP execution time: 2.0 milli-sec
- MAP Speedup vs. Pentium 38

# Gauss Seidel Iterative Solver

- Scientific Floating Point Kernel (single precision for now)
- Works for diagonally dominant matrices
- Some math manipulation to create an iterative solver:

$$Ax = b \rightarrow (L+D+U)x = b \rightarrow x = D^{-1}b - D^{-1}(L+U)x \rightarrow x_{n+1} = (Ab)x_n$$

```
while(maxerror > tolerance) {           // do a next iteration
    maxerror = 0.0;
    for(i=0;i<n;i++) {                   // compute new x[ i ]
        sxi = x[ i ];
        xi = 0.0;
        for(j=0;j<n+1;j++)
            xi += Ab[ i*COL+j ] * x[ j ];    // in product
        error = abs(xi - sxi);
    }
    maxerror = max(maxerror, error);
}
```

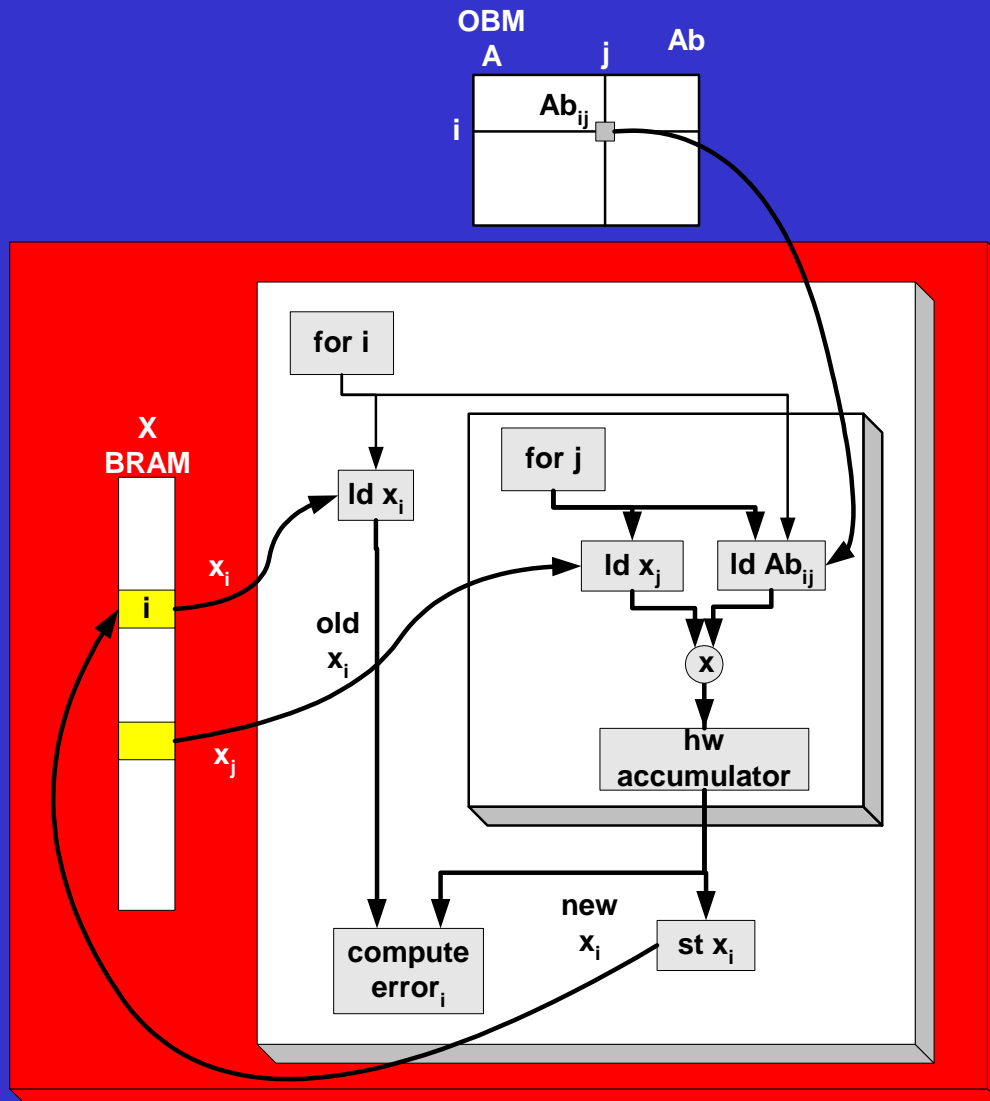


\_\_\_\_\_





# Accumulator Macro



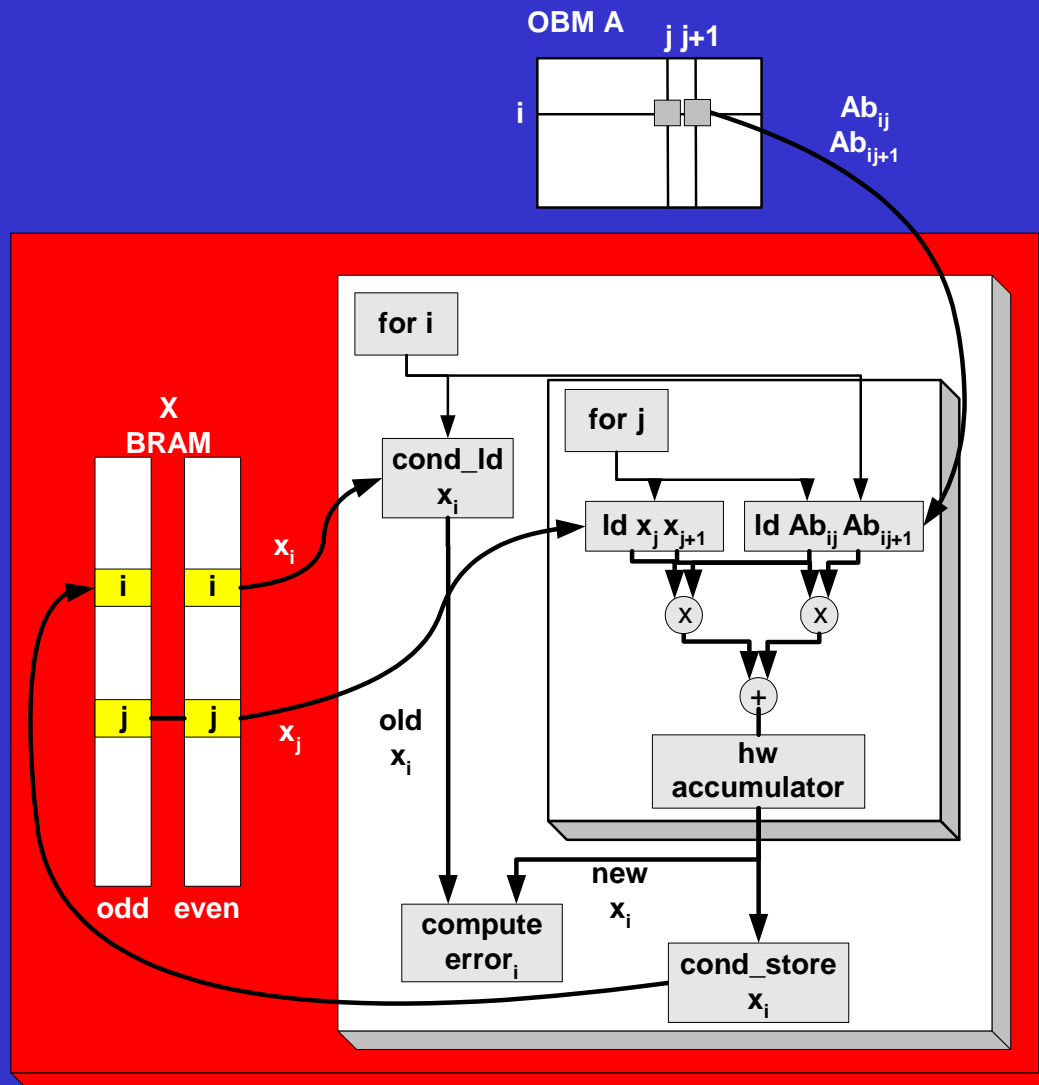
**Hardware  
Accumulator  
macro  
resolves  
read / write  
conflict**

# Packing the data

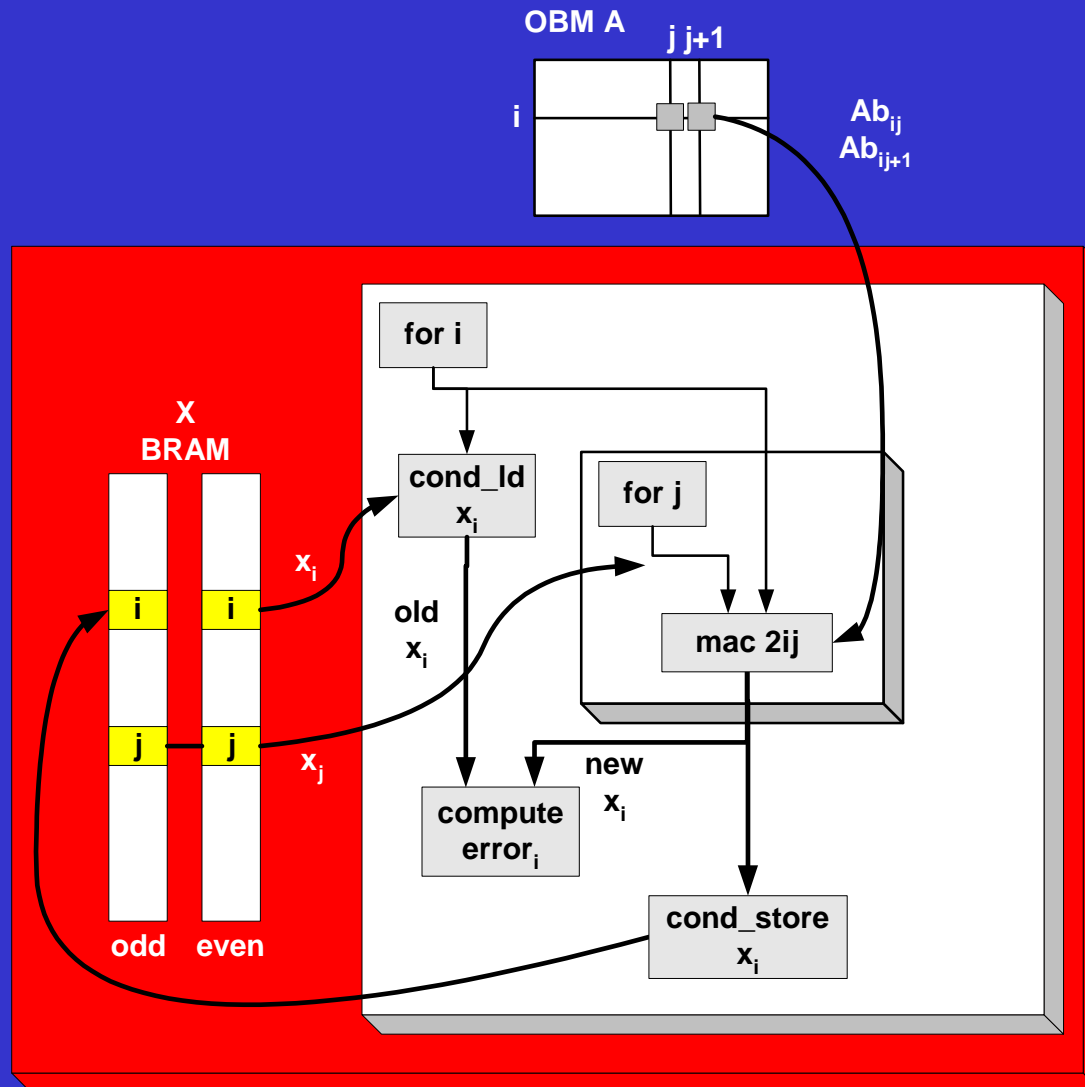
**64 bit OBM word  
can contain two floats**

**This requires unrolling j loop  
which now accesses  
 $Ab_{ij}$   $Ab_{ij+1}$   $X_j$   $X_{j+1}$**

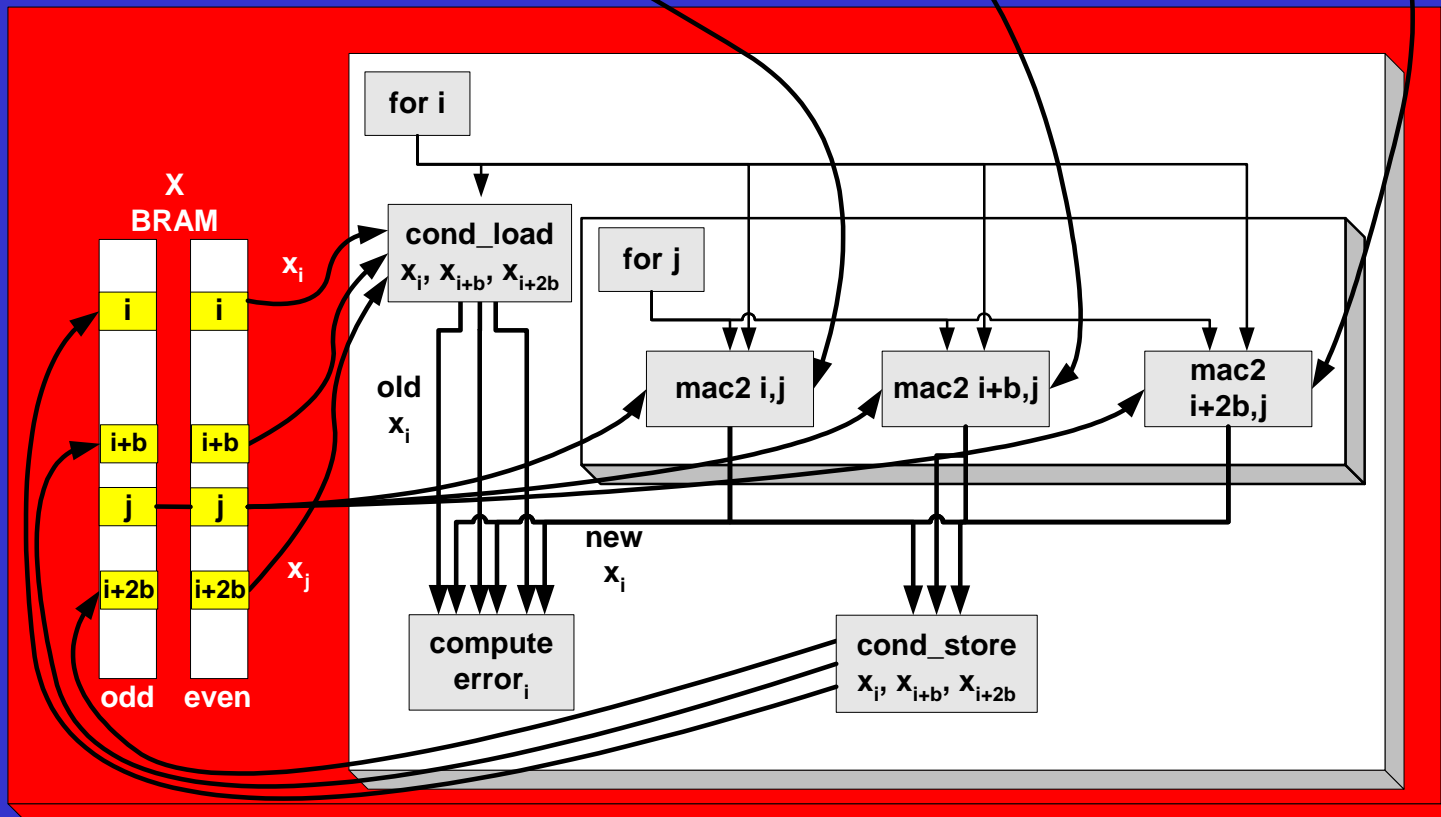
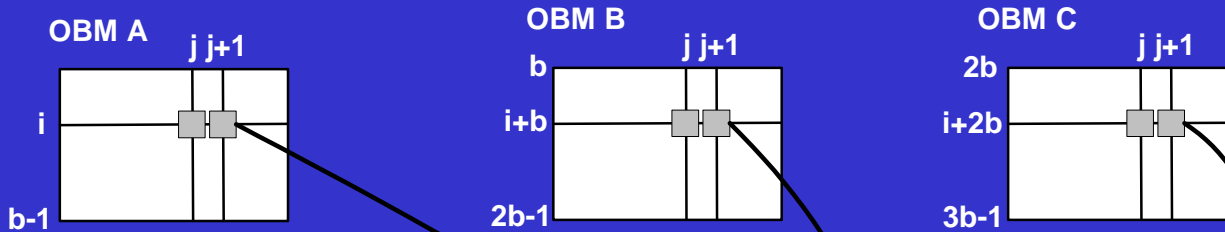
**To avoid multiple Block  
RAM reads, X is stripe  
partitioned over two Block  
RAM arrays**



# Pack Abstracted



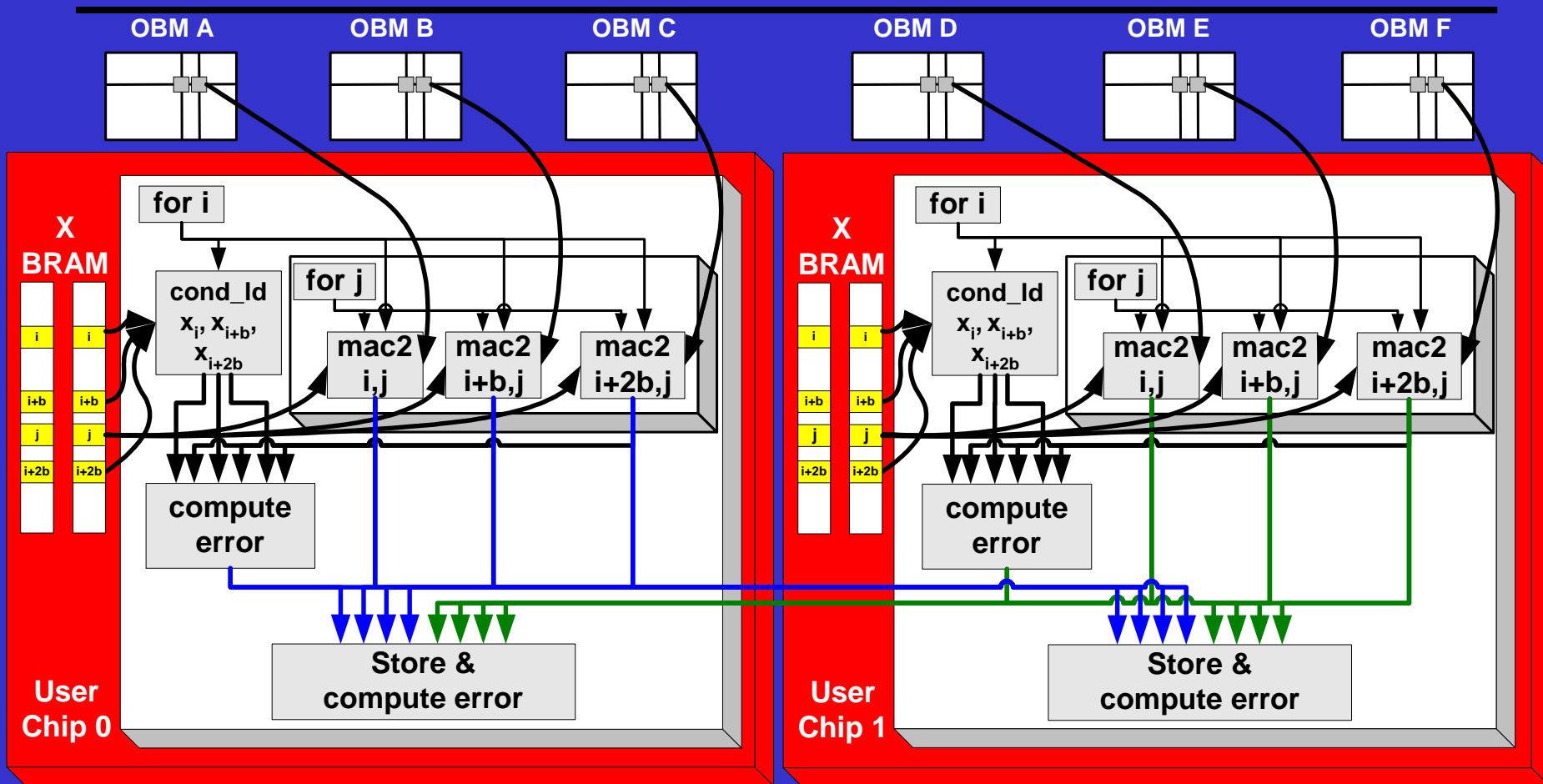
# Data Partitioned into 3 blocks



**Ab is now  
row-block  
distributed  
(3 blocks in  
3 OBMs)**

**j loop now  
computes  
3 new Xs**

# Two FPGAs



**Ab is row block distributed (6 blocks in 6 OBMs)**  
**The j-loops perform 24 Floating Ops in each clock**  
**FPGA0 and FPGA1 exchange 3 Xs, 1 error**

# Gauss Seidel Performance

	<b>n=500</b>	<b>n=1000</b>	<b>n=2000</b>
<b>No. Iterations</b>	<b>27</b>	<b>6</b>	<b>7</b>
<b>Pentium IV (2.8 GHz )</b>	<b>0.19 secs 65 MFlops</b>	<b>0.48 secs 26 MFlops</b>	<b>1.90 secs 28 MFlops</b>
<b>MAP</b>	<b>0.013 secs 830 MFlops</b>	<b>0.008 secs 1.23 GFlops</b>	<b>0.031 secs 1.65 GFlops</b>
<b>MAP speedup vs. Pentium</b>	<b>14</b>	<b>57</b>	<b>60</b>

# Conclusions

---

- **High Level Algorithmic Language runs on FPGA based HPEC system**
  - DEBUG Mode allows most development on workstation
- **We can apply standard software design methodologies**
  - stepwise refinement
    - currently using macros
    - later using (user directed?) compiler optimizations
- **Bandwidth is key to FPGA performance**
  - Often, more operations are available in the FPGA fabric than can be supplied by the available off-chip I/O
  - FPGA capability is improving rapidly
- **Currently speedups ~50 vs. Pentium IV**
- **Future: Multiple MAPs**
  - More complex, streaming applications

