The Cell Processor Architecture & Issues

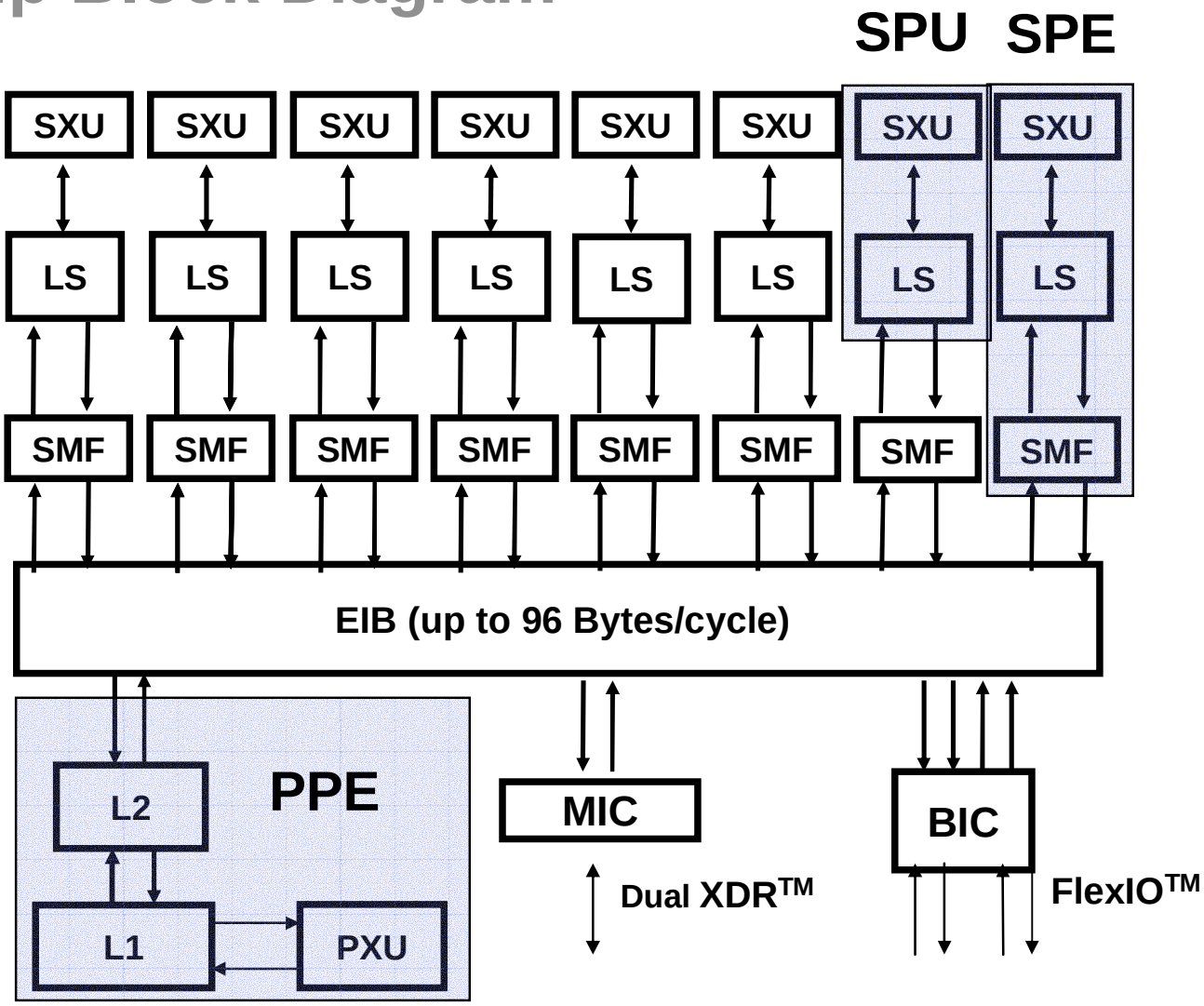
Agenda

- **Cell Processor Overview**
- **Programming the Cell Processor**
- **Concluding Remarks**

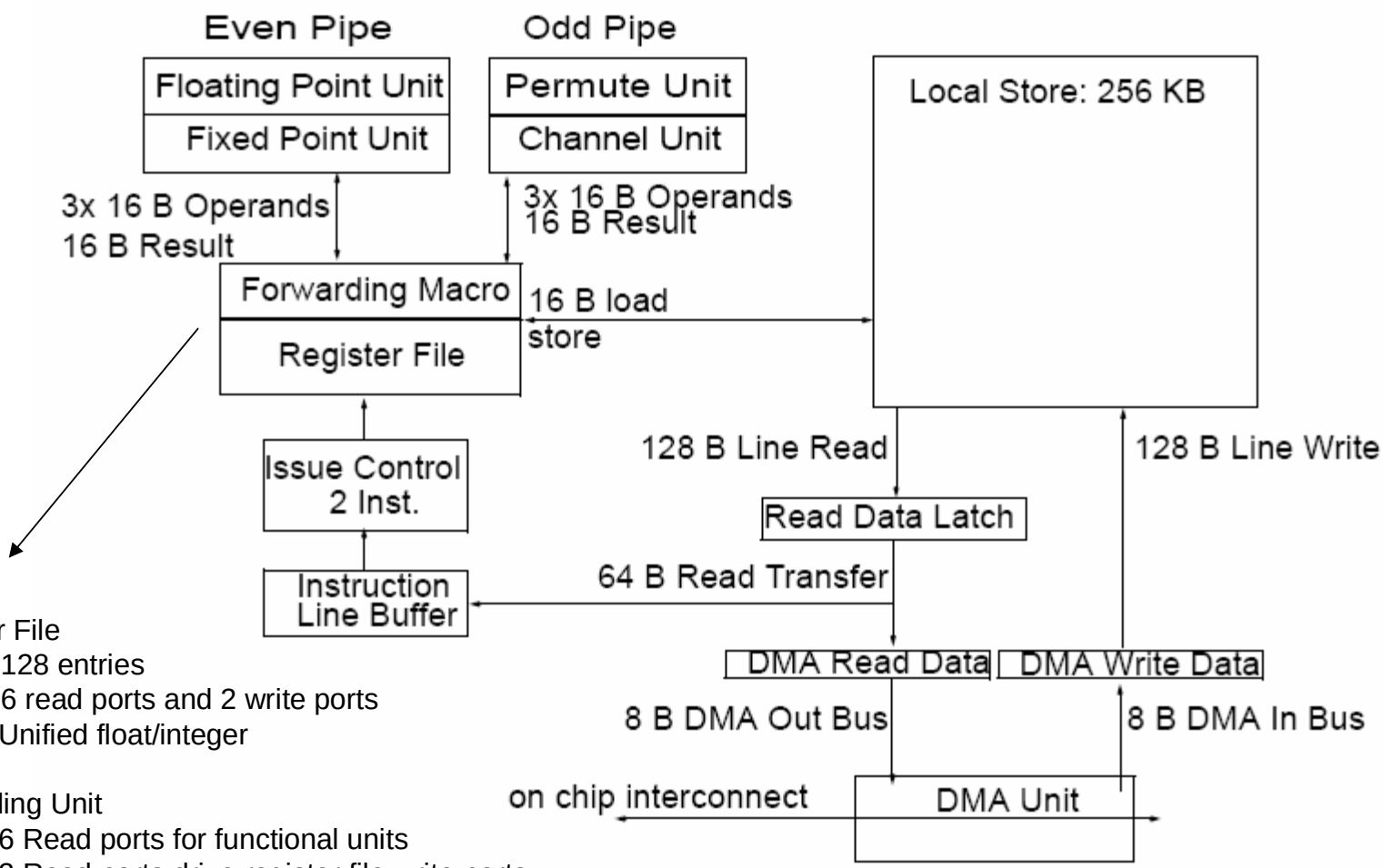
Cell Highlights

- Observed clock speed
  - > 4 GHz
- Peak performance (single precision)
  - > 256 GFlops
- Peak performance (double precision)
  - >26 GFlops
- Area 221 mm2
- Technology 90nm SOI
- Total # of transistors 234M

Cell Chip Block Diagram



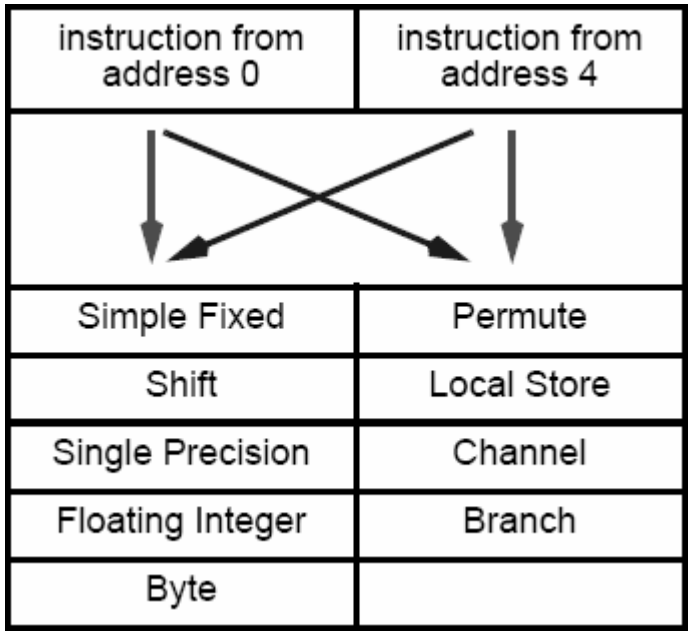
SPE Data Flow



- Register File
  - 128 entries
  - 6 read ports and 2 write ports
  - Unified float/integer
- Forwarding Unit
  - 6 Read ports for functional units
  - 2 Read ports drive register file write ports
  - Orders results from function units into program order

Issue

- In-order Issue
- Dual issue requires alignment according to type
- Instruction Swap forces single issue
- 7 units & 2 pipes



SPE ISA Summary

- 32b fixed length instruction formats.
- Up to 3 sources and 1 target register
- Instructions operate on 128b data
  - 16 x 8b , 8 x 16b, 4 x 32b , 2 x 64b

example for FMA:

OP	RT	RB	RA	RC

- 4-way sp, 2-way dp multiply-add
- Integer arithmetic, logicals & compares
- Shifts / Rotates of words, quad words
- Loads/Stores, Branches
- Channel I/O
- Media instructions

Media Instructions

- Avg bytes
- Abs differences bytes
- Sum bytes in words
- Bit-wise select
- Shuffle bytes
- Count leading zeros in words
- Pop count in bytes
- Gather bits from bytes, short, words

Programming Cell

Prototype SPE Compiler Optimizations

- Sub-word Optimizations:
  - Sub-word operations run natively on the SPE
  - 2-way 64b, 4-way 32b, 8-way 16b, 16-way 8b (ops per cycle)
- Branch Optimizations:
  - If Conversion using the Select instruction
  - Miss prediction penalty avoidance using Branch Hint Instruction
    - Via compiler analysis, profiler feedback, and/or user directive
- Instruction Scheduling Optimizations:
  - Maximizing dual issue opportunities
  - Avoiding fetch starvation using explicit instruction pre-fetch
- Automatic ‘SIMDIZATION’
  - Extract SIMD parallelism across loop iterations
  - Identify alignment requirements for SIMD operations
  - Generate code with embedded shuffle instructions to achieve alignment

Prototype Single Source Cell Compiler

- High Level Requirements
  - Partition application across multiple SPE's
  - Orchestrate Code and Data transfers via DMA
- User Directed Parallelization
  - OpenMP pragmas guide parallelization decisions.
  - Compilation Steps:
    - Outline parallel code sections
    - Apply machine independent optimizations
    - Generate optimized code for PPE and SPE's
  - A *Master Thread* runs on the PPE processor.
    - Orchestrates all work sharing constructs

Prototype Single Source Cell Compiler (contd)

- Single Shared Memory Abstraction
  - Programmers view is a single addressable memory
  - SPE program and data reside in system memory. Compiler automatically manages data movement between system memory and a compiler controlled 'Software Cache' in SPE Local Store.
  - Performance optimizations:
    - Local, stack allocated variables in the Local Store
    - Program restructuring to allow multiple elements to be fetched together
      - Loop restructuring transformations such as blocking and interchange
    - Software pipeline blocked loops
      - Overlap data movement and computation
  - Code Partitioning
    - Enables program code segment sizes larger than Local Store.
    - Integrated with the data software cache
      - Home location of code partitions is system memory
      - Partition Manager loads partitions from system memory to Local Store
        - Normally during an inter-partition function call or return

DGEMM

$$C = \alpha A B + \beta C$$

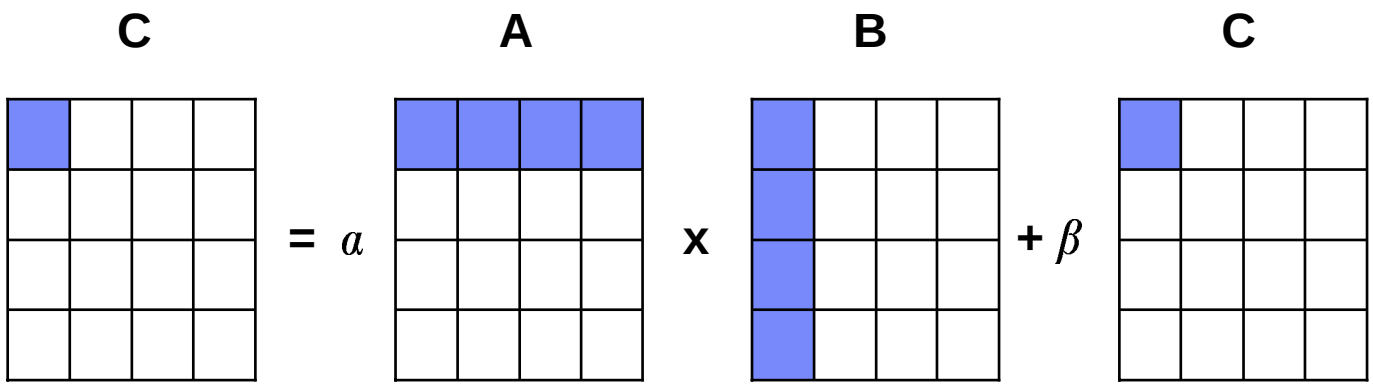
- **Where**
  - **A, B & C are double precision, NxN real matrices**
  - **α & β are double precision reals**
- **Single Cell Processor**
  - **Partition C matrix over 8 SPE's**
  - **Tile each SPE's portion into LS sized pieces**
    - **Each LS must hold at least 3 tiles at a time (A, B, C)**
    - **More tiles if multi-buffered**
    - **Example tiles: 32x32 (8KB) or 64x64 (32KB)**
  - **Tile the LS pieces in register file sized pieces**
    - **128, 16B SIMD registers per SPE**
    - **Example tile: 4x8 elements = 16 registers**
- **Compute bound: N^3 multiplies vs $3N^2$ loads + N^2 stores**

Partitioning and Tiling

- **Partition C matrix among SPE's**

SPE 0	SPE 1
SPE 2	SPE 3
SPE 4	SPE 5
SPE 6	SPE 7

- **Tile each SPE portion of C**



- **Choose blue tile size to fit in local store (256KB)**

SPE Pseudo-Code: Single Buffer

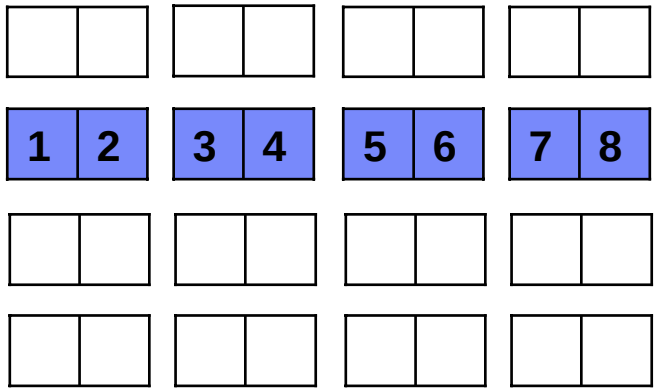
```
For (i=0; i < N; ++i)
{    For (j=0; j < N; ++j)
     {    Issue_DMA (tile C(i,j) from MS to LS);
          For (k=0; k < N; ++k)
          {    Issue_DMA (tile A(i,k) from MS to LS);
               Issue_DMA (tile B(k,j) from MS to LS);
               WaitAll ();
               For (Each sub-tile of tile C(i,j))
               {    For (Each sub-tile of tile A(i,k) and B(k,j))
                    {    Load sub-tiles of A, B, and C into registers;
                         MulAdd sub-tiles into result registers;
                         Store result registers back to sub-tile of C(i,j) in LS;
                    }
                    Clean-up partial sums;
               }
          }
          Issue_DMA (tile C(i,j) from LS to MS);
     }
}
```

SPE Pseudo-Code: Double Buffer

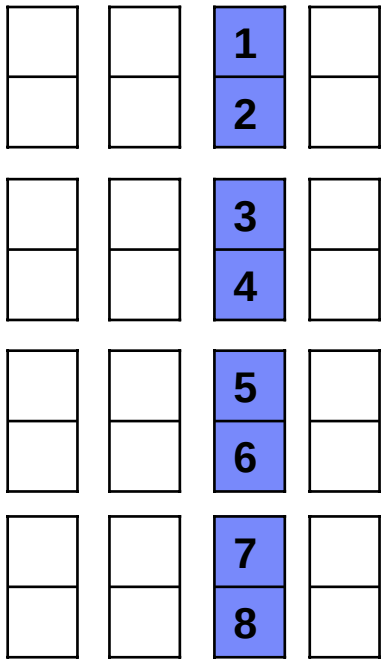
```
For (i=0; i < N; ++i)
{     For (j=0; j < N; ++j)
    {     Issue_DMA (InitialGroup, tile C(i,j) from MS to LS);
          Issue_DMA (CurrentGroup, tile A(i,0) from MS to LS);
          Issue_DMA (CurrentGroup, tile B(0,j) from MS to LS);
          For (k=1; k < N; k++)
          {   if (k+1 < N)
              {     Issue_DMA (NextGroup, tile A(i,k) from MS to LS);
                    Issue_DMA (NextGroup, tile B(k,j) from MS to LS);
              }
              if (k == 1) WaitAll (InitialGroup);
              WaitAll (CurrentGroup);
              For (Each sub-tile of tile C(i,j))
              {     For (Each sub-tile of tile A(i,k-1) and B(k-1,j))
                    {     Load sub-tiles of A, B, and C into registers;
                          MulAdd sub-tiles into result registers;
                          Store result registers back to sub-tile of C(i,j) in LS;
                    }
                    Clean-up partial sums;
              }
              Temp = CurrentGroup;
              CurrentGroup = NextGroup;
              NextGroup = Temp;
          }
          Issue_DMA (InitialGroup, tile C(i,j) from LS to MS);
     }
}
```

Register file tiling

Tile from A matrix



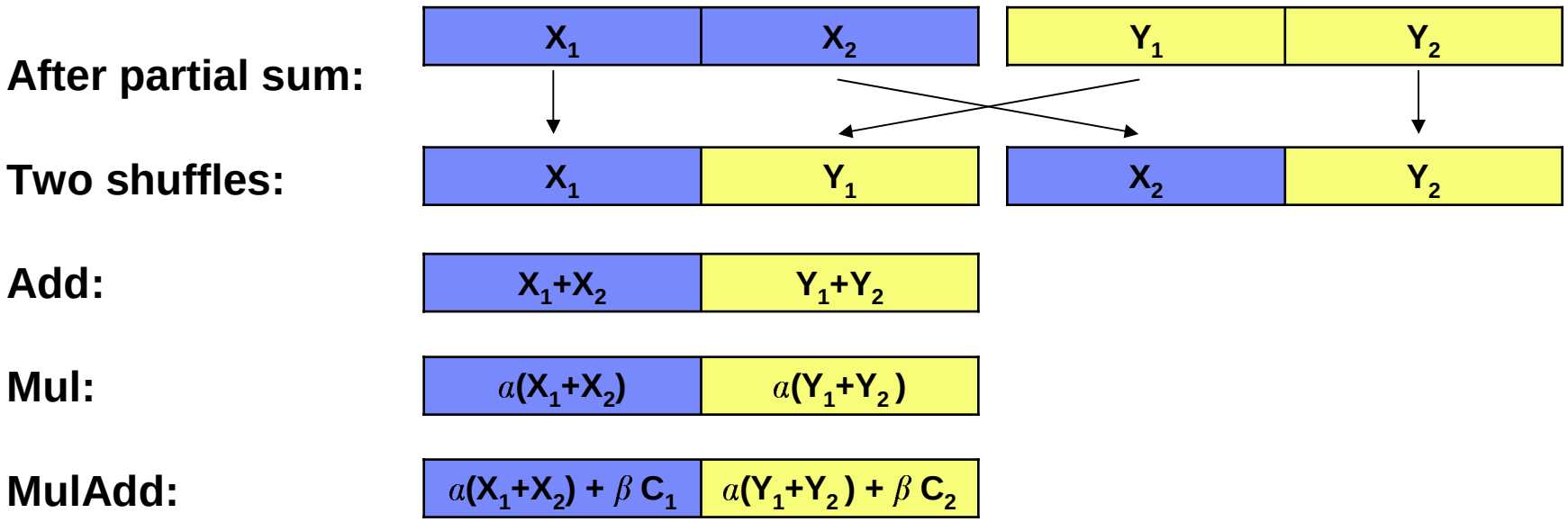
Tile from B matrix



- SIMD Register – 2 DP floats
- Tile A is 8x4
- Tile B is 4x8
- Matrix B must be stored accordingly or reorganized at load time
- Each register pair will be muladd'ed to a temp register
- Temp registers summed after all register file tiles processed
- Gives one register with two partial sums

Partial Sum Clean-Up

- Each sum results in single register with two partial sums
- These must be combined and merged with C
- We do this two sums at a time:



HPCC Random Access Benchmark (GUPS)

- **Measures random memory update throughput**
- **Algorithm**
  - Initialize large table of 64-bit elements in main memory
  - Generate 64-bit pseudo-random number
  - Use random number to generate table address
  - Update corresponding address using random number
  - Iterate…
- **Focus: Using SIMD to generate random numbers** R_j

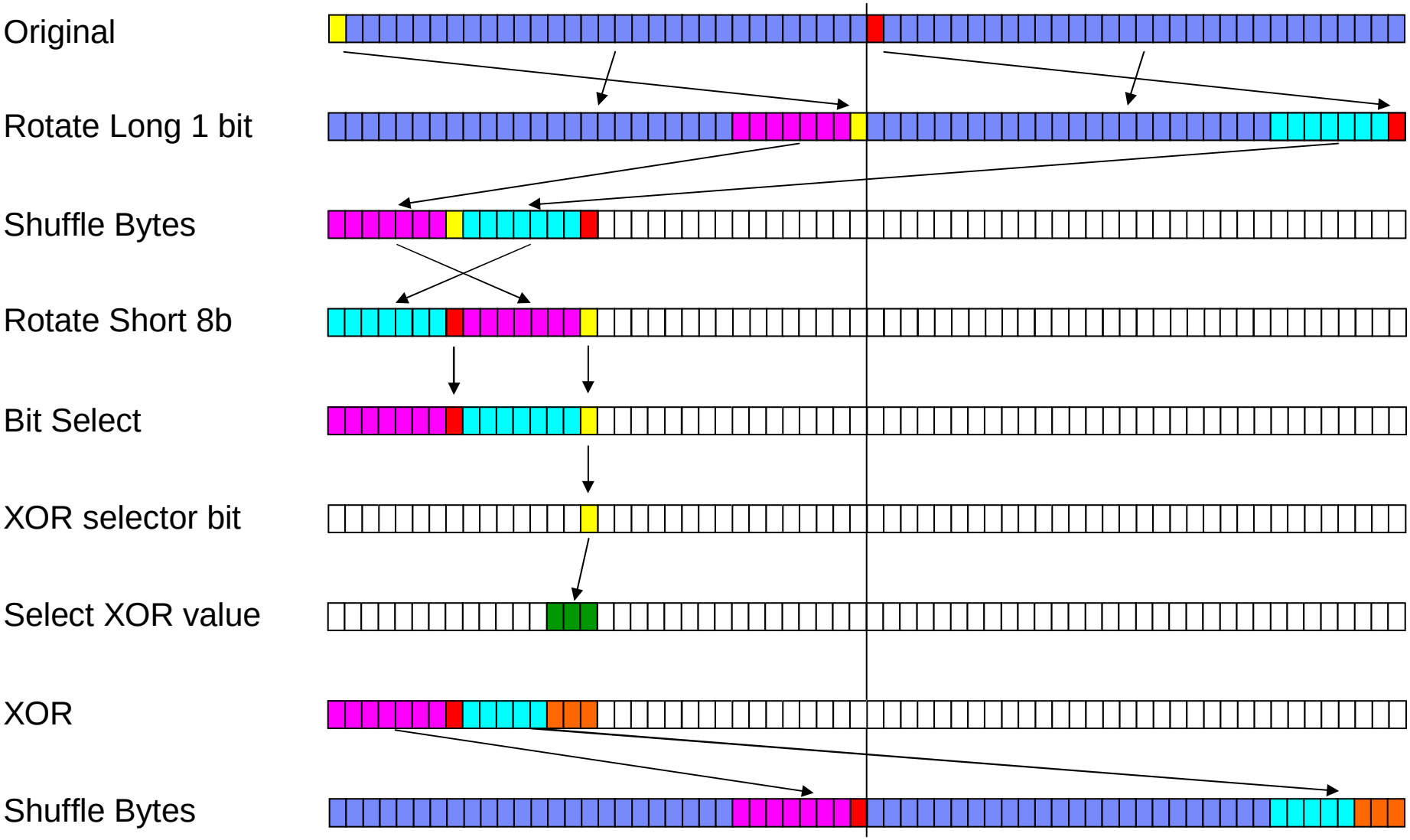
$$R_j = (R_{j-1} << 1) \wedge [\, (R_{j-1} < 0) \; ? \; 7 : 0 \,]$$

  - Use highest order bit to select XOR mask: 7 or 0
  - Shift number left 1 bit and XOR with mask

HPCC Random Number Generation

- Naive approach
  - 128b register, 64b elements ➔ 2-way SIMD
- Better: Process 64 random numbers in 32 registers
  - No 64b rotate so…
  - 32 **integer rotate left** 1 bit
    - Bottom bits of the 2 words in each double-word now reversed
  - 16 **shuffles** & 8 **or**'s
    - Gather the bottom bytes of all words into 8 registers
  - 8 **rotate shorts** by 8 bits to swap all byte pairs
    - Least significant bits now in their proper bytes
  - 8 **select** ops to merge the top 7 bits with properly aligned least significant bits
  - 8 **ands** to mask all but lsb's
  - 8 **shuffle byte** ops to replace lsb's with "7" or "0"
  - 8 **xor** ops to merge the "7" or "0"
  - 32 **shuffle byte** ops merge updated lower order bytes back into random numbers

HPCC Random Number Generation



Concluding Remarks

- **Cell has characteristics that:**
  - **Can map well to HPC applications**
  - **Support a variety of programming models**
  - **Present significant opportunities and challenges to:**
    - **Programmers, Compilers and Run time systems**
- **A few enhancements HPC enthusiasts would like some day:**
  - **64b Mask & Rotate**
  - **2-way DP PF MADD's per cycle per SPE**
  - **Main memory expansion capability**

Backup

SIMD “IF” Without Branching

- **SIMD compare**
  - 2 input registers, 1 output
  - Compare input elements
    - On success: Corresponding bits of output register set to 1
    - Else: Corresponding bits of output register set to 0
- **SIMD bit select**
  - 3 input registers (A, B, C), 1 output
  - Copy bits from A to output if corresponding bits in C are 1, else Copy bits from B to output if corresponding bits in C are 0
- **Example: 16b shorts**

```
for (i=0; i < 8; ++i)
   if     ( x[i] == y[i] )  a[i] = b[i]
   else                     a[i] = c[i]
```

- **Step 1 - Compare**

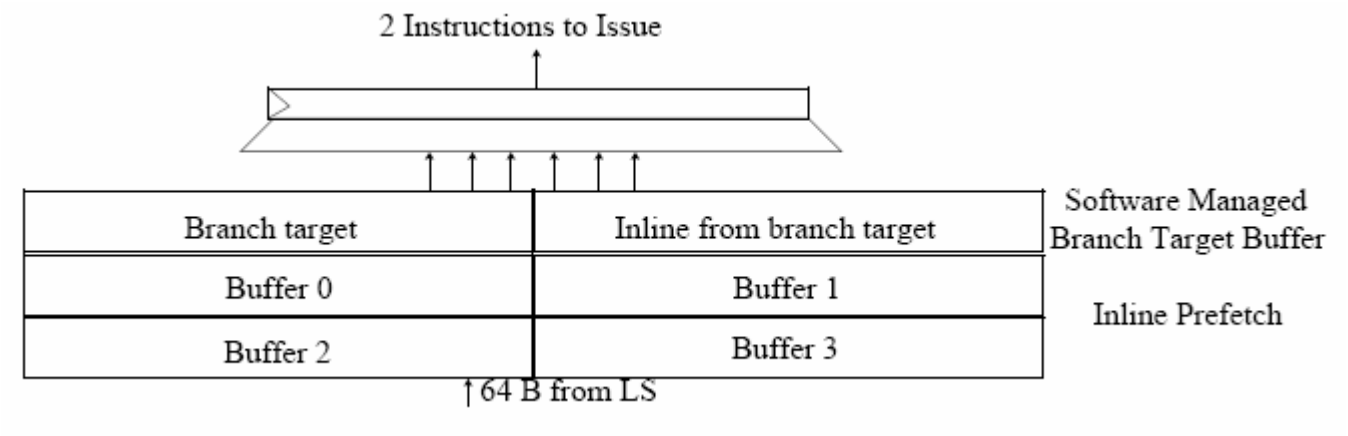
Input X	x0	x1	x2	x3	x4	x5	x6	x7
Input Y	y0	y1	y2	y3	y4	y5	y6	y7
Output	1	0	1	1	0	1	0	0

Step 2 - Select

Input B	b0	b1	b2	b3	b4	b5	b6	b7
Input C	c0	c1	c2	c3	c4	c5	c6	c7
Input	1	0	1	1	0	1	0	0
Output A	b0	c1	b2	b3	c4	b5	c6	c7

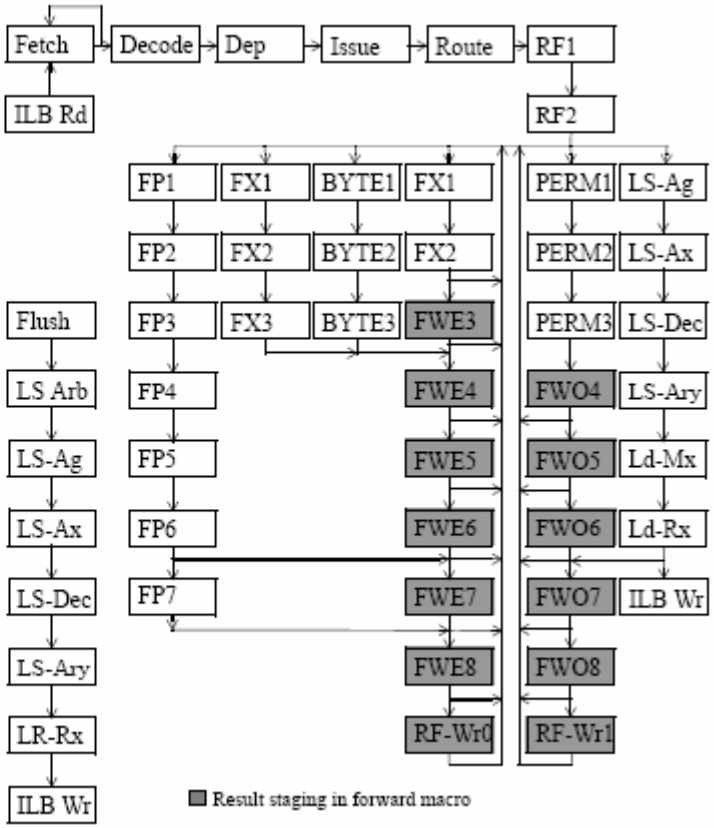
Instruction Line Buffer

- I-Fetch aligned to 512 bit boundary
  - improves effective bandwidth
- Software Managed Branch Target Buffer (SMBTB)
  - loaded by branch hint instructions



Pipeline Diagram

Latencies	
Simple Fixed (FX)	2
Shift (FX3)	4
Single Precision (FP1-6)	6
Floating Integer (FP7)	7
Byte	4
Permute	4
Local Store (LS)	6
Channel (not shown)	6
Branch (not shown)	4



SPE DMA Unit

- Data transfer
  - Between Main Memory and SPE Local Store
  - Full virtual DMA (64b effective PowerPC addresses).
  - Data transfer commands
    - put, get, put_list, get_list
    - groups of commands associated by tags
- Synchronization
  - Fence: order DMA's within a tag group.
  - Barrier: order DMA's within the DMA queue
  - Signals: preemption via external agents
  - Atomics: cache coherent memory based locks

Channel Interface

- The interface from SPU core to rest of system
- Instructions: read, write, read capacity
- Effects appear at SPU core interface in instruction order
- Blocking reads & writes stall SPU in low power wait mode
- Channel Facilities Include:
  - DMA control / status
  - Counter-timer
  - Interrupt controller
  - SPE to SPE, PPE to SPE Mailboxes

Desired Characteristics for Efficiency on an SPE

- Loops that can be unrolled deeply
  - Ideally, LS accesses that are small powers of 2.
  - Can user supply alignment hints to compiler?
- Limited pointer and branch usage
  - SPEs are not well-suited to ‘branchy’ code
  - Can the user supply effective compile time branch hints?

Desired Characteristics for Efficiency on CELL

- Explicit Data Parallelism
  - A data set that can be decomposed into reasonably well balanced parts.
  - A computation phase long enough to:
    - Amortize the cost of memory latency and synchronization overheads.
    - Software pipeline to hide memory latency and synchronization overheads.
- Function Offloading
  - Core computations that can be handled by functions that run efficiently on an SPE.
  - Enough independent functions to allow for adequate parallelism.
  - Function Pipelining
    - Balanced pipeline stages (code size and path length)
    - Volume of data passed in each stage consistent with size of the SPE LS

DGEMM Comments and Caveats

- **Main memory alignment issues**
  - **Preceding analysis ignores poor alignment and transposes**
  - **Most alignment can be hidden behind muladds (on different pipe)**
- **Assumed no border clean-up code required**
  - **Tile size choice can leave partial tiles at matrix edge**
  - **Border clean-up can be on PX or SPE's**

Cell Die