Cell Programming Tips & Techniques

Course Code: L3T2H1-58
Cell Ecosystem Solutions Enablement

Class Objectives – Things you will learn

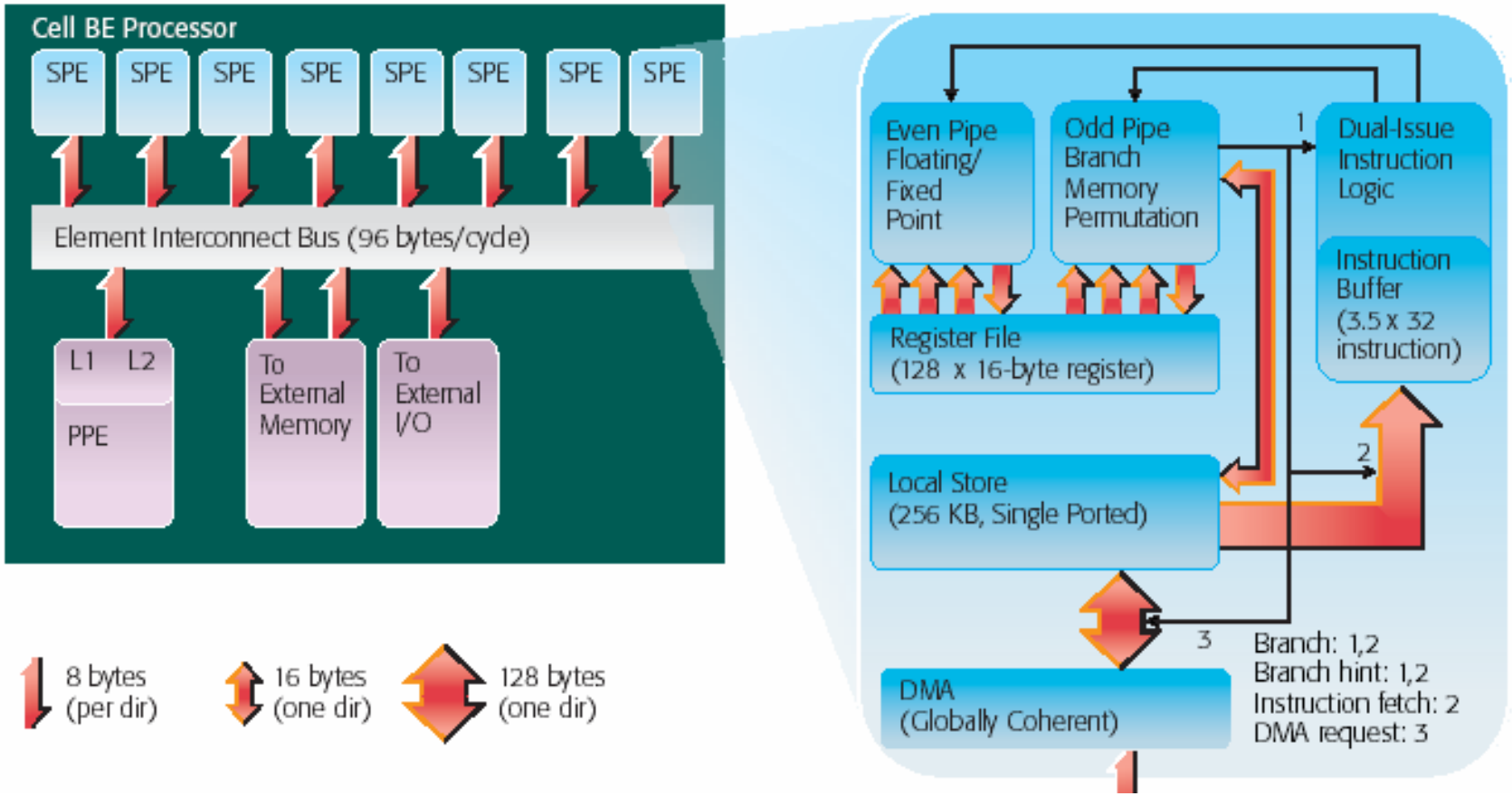
- **Key programming techniques to exploit cell hardware organization and language features for**
  - SPU
  - SIMD

Class Agenda

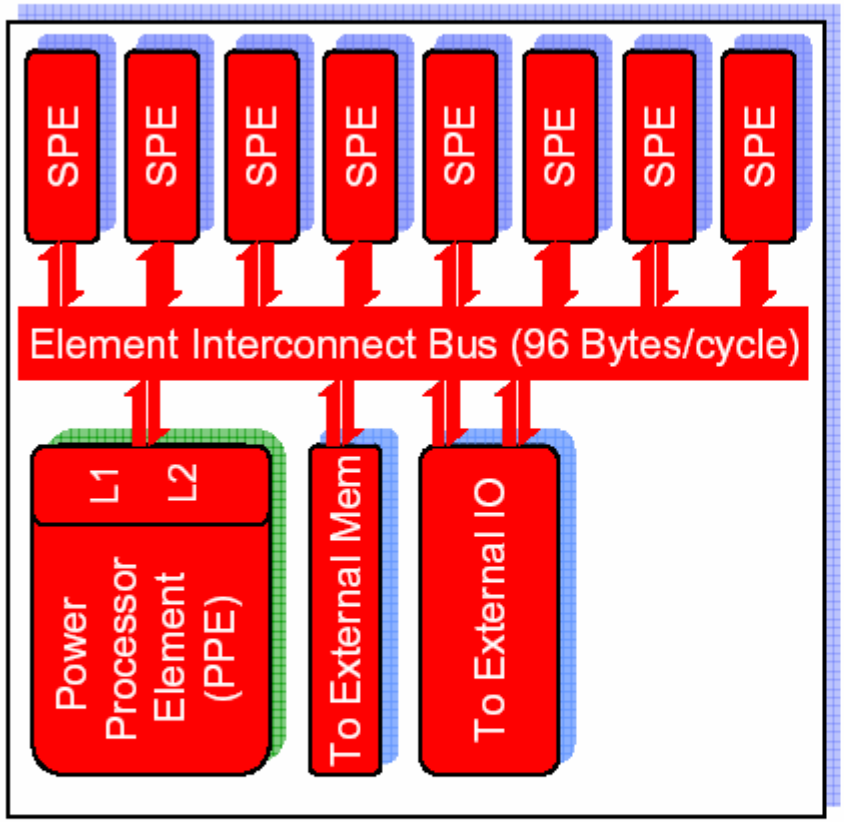
- Review relevant SPE Features
- SPU Programming Tips
  - Level of Programming (Assembler, Intrinsics, Auto-Vectorization)
  - Overlap DMA with computation (double, multiple buffering)
  - Dual Issue rate (Instruction Scheduling)
  - Design for limited local store
  - Branch hints or elimination
  - Loop unrolling and pipelining
  - Integer multiplies (avoid 32-bit integer multiplies)
  - Shuffle byte instructions for table look-ups
  - Avoid scalar code
  - Choose the right SIMD strategy
  - Load / Store only by quadword
- SIMD Programming Tips

Review Cell Architecture

Cell Processor



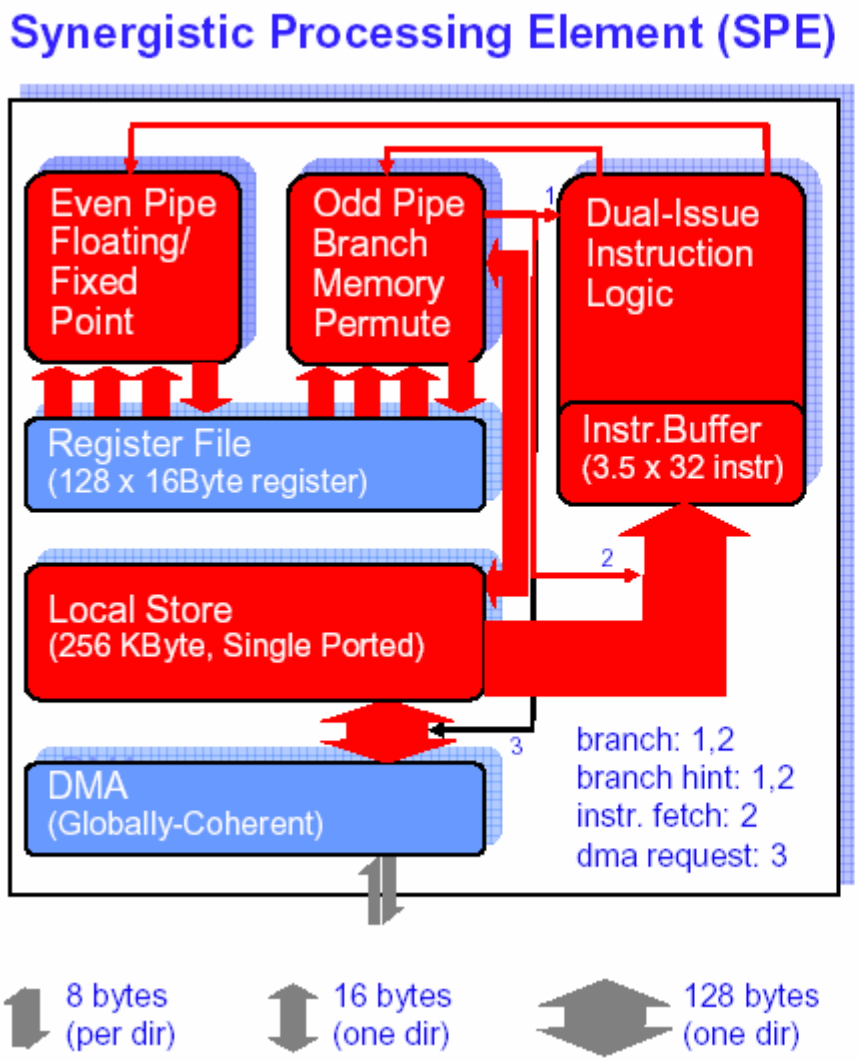
Cell Broadband Engine Overview



- **Heterogeneous, multi-core engine**
  - 1 multi-threaded power processor
  - up to 8 compute-intensive-ISA engines
- **Local Memories**
  - fast access to 256KB local memories
  - globally coherent DMA to transfer data
- **Pervasive SIMD**
  - PPE has VMX
  - SPEs are SIMD-only engines
- **High bandwidth**
  - fast internal bus (200GB/s)
  - dual XDR™ controller (25.6GB/s)
  - two configurable interfaces (76.8GB/s)
  - numbers based on 3.2GHz clock rate

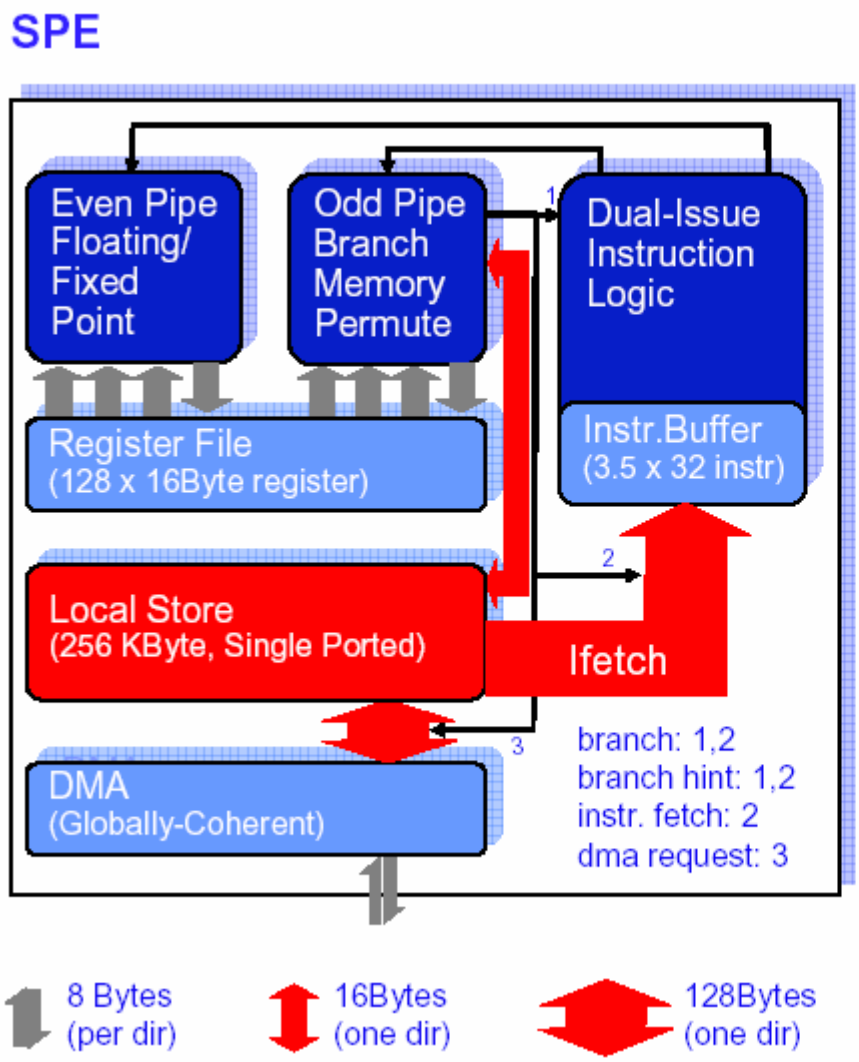
8 Bytes (per dir) 16Bytes (one dir) 128Bytes (one dir)

Key SPE Features



- **SIMD-only functional units**
  - 16-bytes register/memory accesses
- **Simplified branch architecture**
  - no hardware branch predictor
  - compiler managed hint/predication
- **Dual-issue for instructions**
  - full dependence check in hardware
  - must be parallel & properly aligned
- **Single-ported local memory**
  - aligned accesses only
  - contentions alleviated by compiler

SPE – Single-Ported Local Memory



- **Local store is single ported**
  - less expensive hardware
  - asymmetric port
    - 16 bytes for load/store ops
    - 128 bytes for IFETCH/DMA
  - static priority
    - DMA > MEM > IFETCH
- **If we are not careful, we may starve for instructions**

SPU Programming Tips

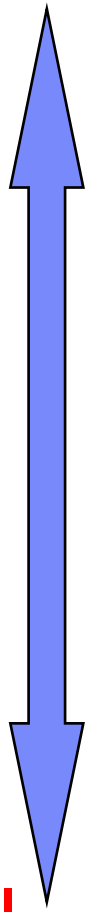
SPU Programming Tips

- Level of Programming (Assembler, Intrinsics, Auto-Vectorization)
- Overlap DMA with computation (double, multiple buffering)
- Dual Issue rate (Instruction Scheduling)
- Design for limited local store
- Branch hints or elimination
- Loop unrolling and pipelining
- Integer multiplies (avoid 32-bit integer multiplies)
- Shuffle byte instructions for table look-ups
- Avoid scalar code
- Choose the right SIMD strategy
- Load / Store only by quadword

Programming Levels on Cell BE

Trade-Off

Performance vs. Effort

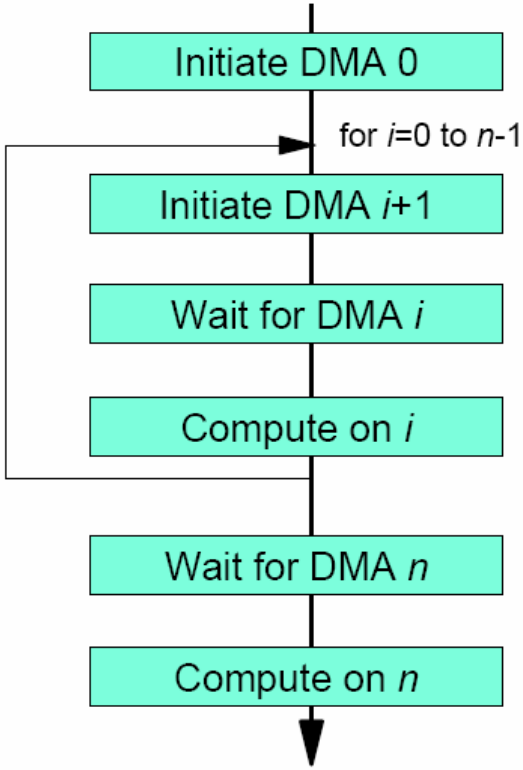


- **Expert level**
  - Assembler, high performance, high efforts
- **More ease of programming**
  - C compiler, vector data types, intrinsics, compiler schedules instructions + allocates registers
- **Auto-SIMDization**
  - for scalar loops, user should support by alignment directives, compiler provides feedback about SIMDization
- **Highest degree of ease of use**
  - user-guided parallelization necessary, Cell BE looks like a single processor

Requirements for Compiler increasing with each level

Overlap DMA with computation

- **Double or multi-buffer code or (typically) data**
- **Example for double bufferign n+1 data blcoks:**
  - Use multiple buffers in local store
  - Use unique DMA tag ID for each buffer
  - Use fence commands to order DMAs within a tag group
  - Use barrier commands to ordr DMAs within a queue



Start DMAs from SPU

- **Use SPE-initiated DMA transfers rather than PPE-initiated DMA transfers, because**
  - there are more SPEs than the one PPE
  - the PPE can enqueue only eight DMA requests whereas each SPE can enqueue 16

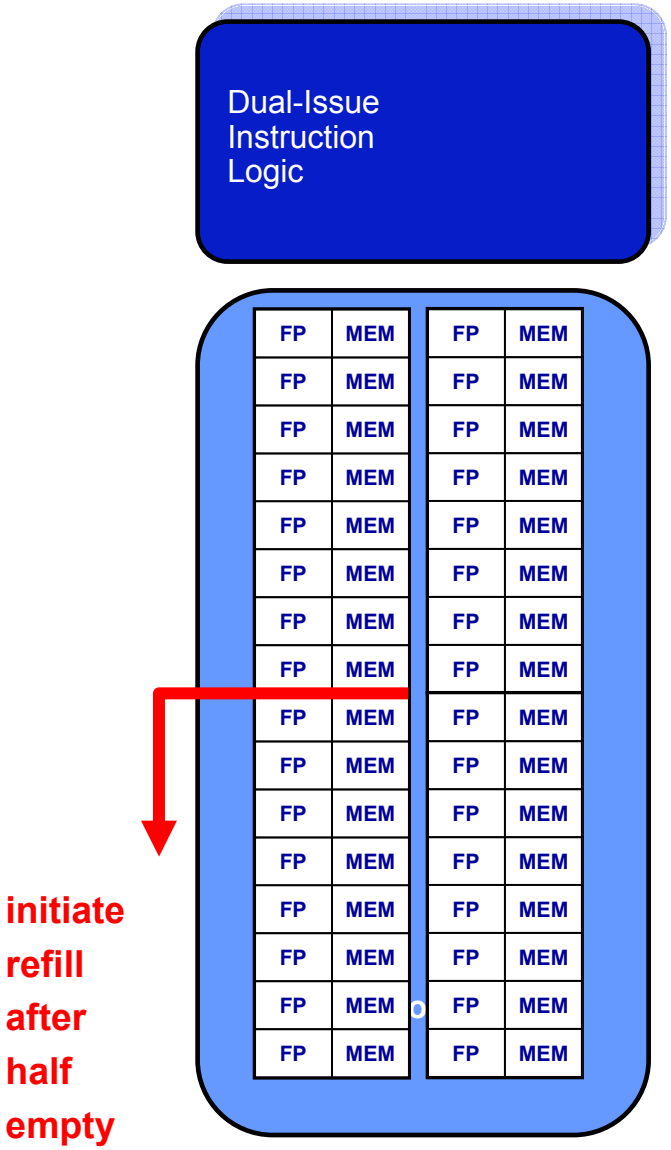
Instruction Scheduling

- Choose intrinsics/instructions to maximize dual issue rates or reduce latencies (fine tuning)

Pipe 0 Instructions	length	stall
Single precision floating-point ops	6	0
Integer multiplies, convert between float/int, interpolate	7	0
Immediate loads, logical ops, integer add/subtract, sign extend, count leading zeros, select bits, carry/borrow generate	2	0
Double precision floating-point ops	7	6
Element rotates and shift	4	0
Byte ops (count ones, abs difference, averge, sum)	4	0
Pipe 1 Instructions	**length**	**stall**
Shuffle bytes, quadword rotates	4	0
Load/store, branch hints	6	0
Branch	4	0
Channel, move to/from spr	6	0

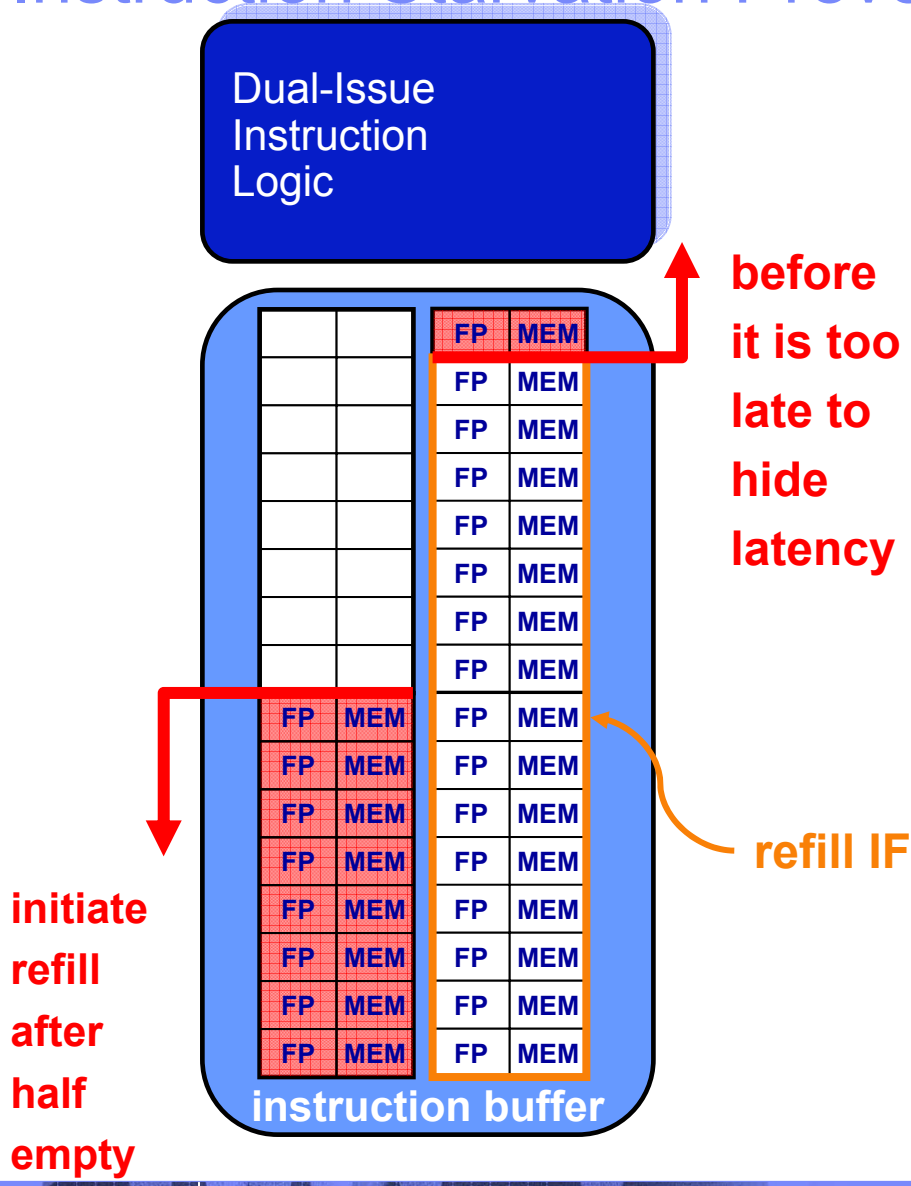
- Dual issue will occur if:
  - pipe 0 instruction even addressed, pipe 1 instruction odd address
  - no dependencies (operands are available)
- Code generators use nops (nop, lnop) to align instructions for dual issue

Instruction Starvation Situation



- **There are 2 instruction buffers**
  - up to 64 ops along the fall-through path
- **First buffer is half-empty**
  - can initiate refill
- **When MEM port is continuously used**
  - starvation occurs (no ops left in buffers)

Instruction Starvation Prevention



- **SPE has an explicit IFETCH op**
  - which initiates an instruction fetch
- **Scheduler monitors starvation situation**
  - when MEM port is continuously used
  - insert IFETCH op within the (red) window
- **Compiler design**
  - scheduler must keep track of code layout

Design for Limited Local Store

- **The Local Store holds up to 256 KB for**
  - the program, stack, local data structures, and DMA buffers.
- **Most performance optimizations put pressure on local store (e.g. multiple DMA buffers)**
- **Use plug-ins (runtime download program kernels) to build complex function servers in the LS.**

Branch Optimizations

- **SPE**
  - Heavily pipelined → high penalty for branch misses (18 cycles)
  - Hardware policy: assume all branches are not taken
- **Advantage**
  - Reduced hardware complexity
  - Faster clock cycles
  - Increased predictability
- **Solution approaches**
  - If-conversions: compare and select operations
  - Predications/code re-org: compiler analysis, user directives
  - Branch hint instruction (hbr, 11 cycles before branch)

Branches

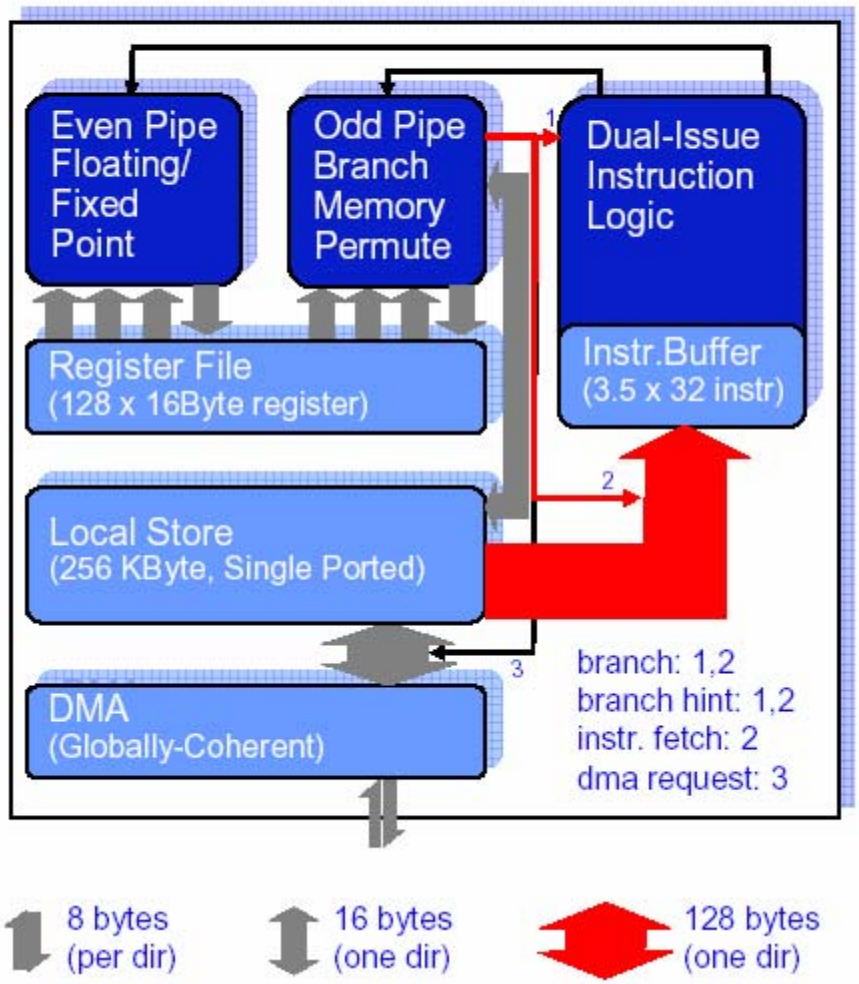
- Eliminate non-predicted branches
  - use feedback directed optimization
  - use __builtin_expect when programmer can explicitly direct branch prediction
    - ex:

```
if (a > b)                          c += 1
else                        c = a+b

if (__builtin_expect(a>b, 0))   c += 1        // predict a is not > b
else                            c = a+b
```

  - utilitze the select bits (spu_sel) instruction.
    - ex:

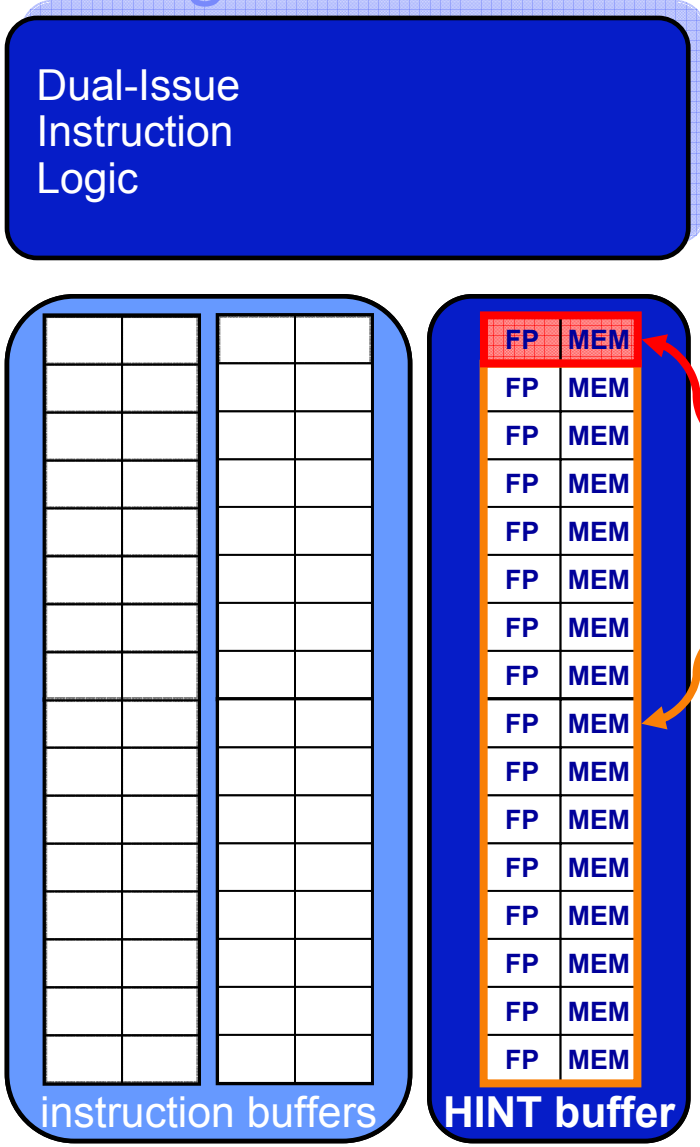
```
if (a > b)                          c += 1
else                        c = a+b

select = spu_cmpgt(a, b);
c1 = spu_add(c, 1);
ab = spu_add(a, b);
c   = spu_sel(ab, c1, select);
```

Feature #2: Software-Assisted Branch Architecture

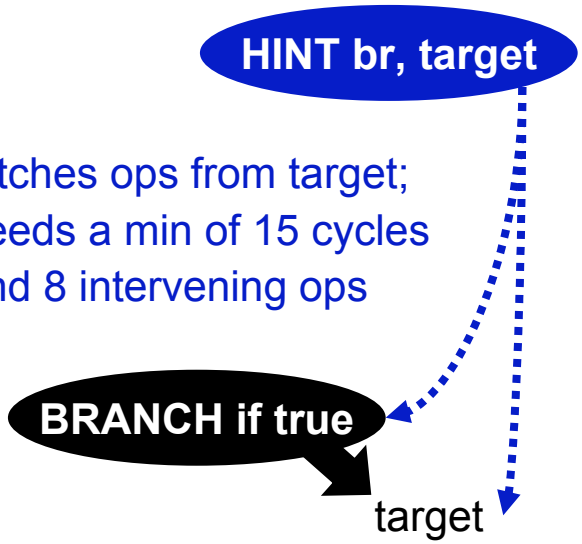


- **Branch architecture**
  - no hardware branch-predictor, but
  - compare/select ops for predication
  - software-managed branch-hint
  - one hint active at a time
- **Lowering overhead by**
  - predicating small if-then-else
  - hinting predictably taken branches

Hinting Branches & Instruction Starvation Prevention



- **SPE provides a HINT operation**
  - fetches the branch target into HINT buffer
  - no penalty for correctly predicted branches



  - compiler inserts hints when beneficial
- **Impact on instruction starvation**
  - after a correctly hinted branch, IFETCH window is smaller

Loop Unrolling

- Unroll loops
  - to reduce dependencies
  - increase dual-issue rates
- This exploits the large SPU register file.
- Compiler auto-unrolling is not perfect, but pretty good.

Loop Unrolling - Examples

```
j=N;
For(i=1, i<N, i++) {
   a[i] = (b[i] + b[j]) / 2;
   j = i;
}
```

```
a[1] = (b[1] + b[N]) / 2;
For(i=2, i<N, i++) {
   a[i] = (b[i] + b[i-1]) / 2;
}
```

```
For(i=1, i<100, i++) {
   a[i] = b[i+2] * c[i-1];
}
```

```
For(i=1, i<99, i+=2) {
   a[i] = b[i+2] * c[i-1];
   a[i+1] = b[i+3] * c[i];
}
```

SPU

- Unroll loops to reduce dependencies and increase dual issue rates.
  - exploits large SPU register file
  - Example - xformlight workload (each loop iteration processes 4 vertices)

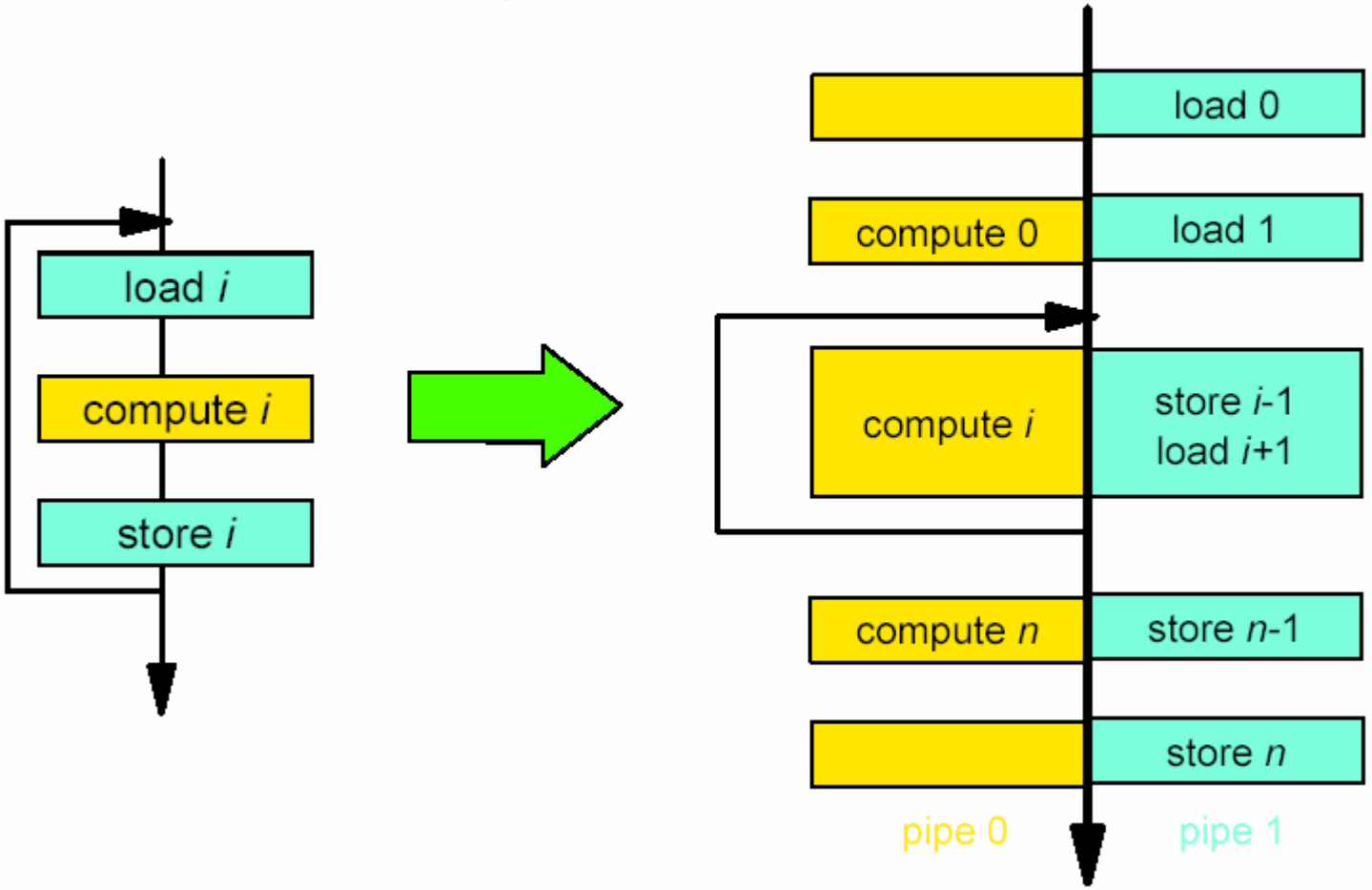
SW unroll factor	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1 (none)	1.00	1.35	3.3%	27.2%	78	768
2	1.52	0.91	19.8%	5.9%	103	1344
4	1.73	0.76	34.3%	0.9%	128	3076
8	1.66	0.67	35.8%	1.5%	128	5252

  - compiler auto-unrolling is not perfect, but doing pretty good.
  - Results using spuxlc unrolling (unroll_large):

SW unroll factor	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1 (none)	1.57	0.87	21.6%	1.9%	94	1472
2	1.52	0.91	18.4%	5.9%	103	1344
4	1.73	0.76	34.3%	0.9%	128	3076
8	1.66	0.67	35.8%	1.5%	128	5252

SPU – Software Pipeline

- Software pipeline loops to improve dual issues rates.
- spuxlc does some pipelining.



Integer Multiplies

- **Avoid integer multiplies on operands greater than 16 bits**
  - SPU supports only a 16-bit x16-bit multiply
  - 32-bit multiply requires five instructions (three 16-bit multiplies and two adds)
- **Keep array elements sized to a power-of-2 to avoid multiplies when indexing.**
- **Cast operands to *unsigned short* prior to multiplying. Constants are of type *int* and also require casting.**
- **Use a macro to explicitly perform 16-bit multiplies. This can avoid inadvertent introduction of signed extends and masks due to casting.**

```
#define MULTIPLY(a, b)\
   (spu_extract(spu_mulo((vector unsigned short)spu_promote(a,0),\
   (vector unsigned short)spu_promote(b, 0)),0))
```

Avoid Scalar Code

- Scalar load/store are slow with long latency
  - SPU only supports quadword loads and stores
  - Ex:

```
void add1(int *p) {
        *p += 1;
}

add1:   lqd     $4, 0(p)        # load the qword pointed to by p
        rotqby  $5, $4, p       # move *p to element 0 of reg 5
        ai      $5, $5, 1       # add 1
        cwd     $6, 0(ptr)      # generate a shuffle pattern to insert *p+1 into qword
        shufb   $4, $5, $3      # insert scalar into qword
        stqd    $4, 0(p)        # save qword with updated scalar pointed to by p
```

  - Strategies:
    - consider making scalars qword integer vectors
    - load or store scalar arrays as quadwords and perform your own extraction and insertion to eliminate load/store instructions.
  - SDK example is RC4 encryption - scalar, non-parallelizable algorithm

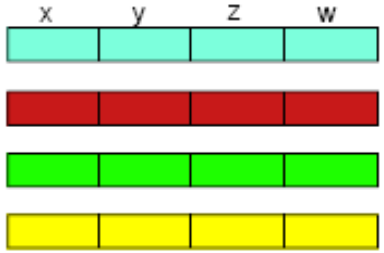
	instructions	cycles	CPI	speedup
scalar	540120	723245	1.34	-
optimized to eliminate scalar overhead	265794	388457	1.46	1.86

Choose an SIMD strategy appropriate for your algorithm

- Evaluate array-of-structure (AOS) organization
  - **For graphics vertices, this organization (also called or vector-across) can have more-efficient code size and simpler DMA needs,**
  - **but less-efficient computation unless the code is unrolled.**
- Evaluate structure-of-arrays (SOA) organization.
  - **For graphics vertices, this organization (also called parallel-array) can be easier to SIMDize,**
  - **but the data must be maintained in separate arrays or the SPU must shuffle AOS data into an SOA form.**

Choose SIMD strategy appropriate for algorithm

- **vec-across**
  - More efficient code size
  - Typically less efficient code/computation unless code is unrolled
  - Typically simpler DMA needs



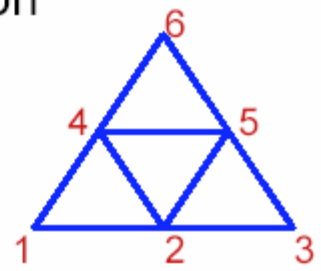
4 independent 4-component vectors (cyan, red, green and yellow)

- **parallel-array**

  - Easy to SIMD – program as if scalar, operating on 4 independent objects at a time
  - Data must be maintained in separate arrays or SPU must shuffle vec-across data into a parallel array form
- **Consider unrolling affects when picking SIMD strategy**

SIMD Example

- SIMD example - point-normal triangle subdivision
  - problem: compute subdivided vertices for *n* independent triangles



  - solution 1: vec-across
    - evaluate vertices one at a time, unroll in improve performance
  - solution 2: parallel array
    - evaluate 4 subdivision vertices at a time for a single triangle
  - solution 3: parallel array
    - evaluate 1 subdivision point at a time on 4 independent triangles

solution	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1	1.0	1.10	9.1%	14.1%	72	1472
1 (unrolled by 2)	1.26	1.04	12.6%	13.6%	112	2496
1 (unrolled by 4)	1.54	0.90	18.5%	5.8%	127	4480
2	1.34	0.96	11.9%	2.6%	107	1856
3	1.70	0.99	13.6%	4.7%	113	512

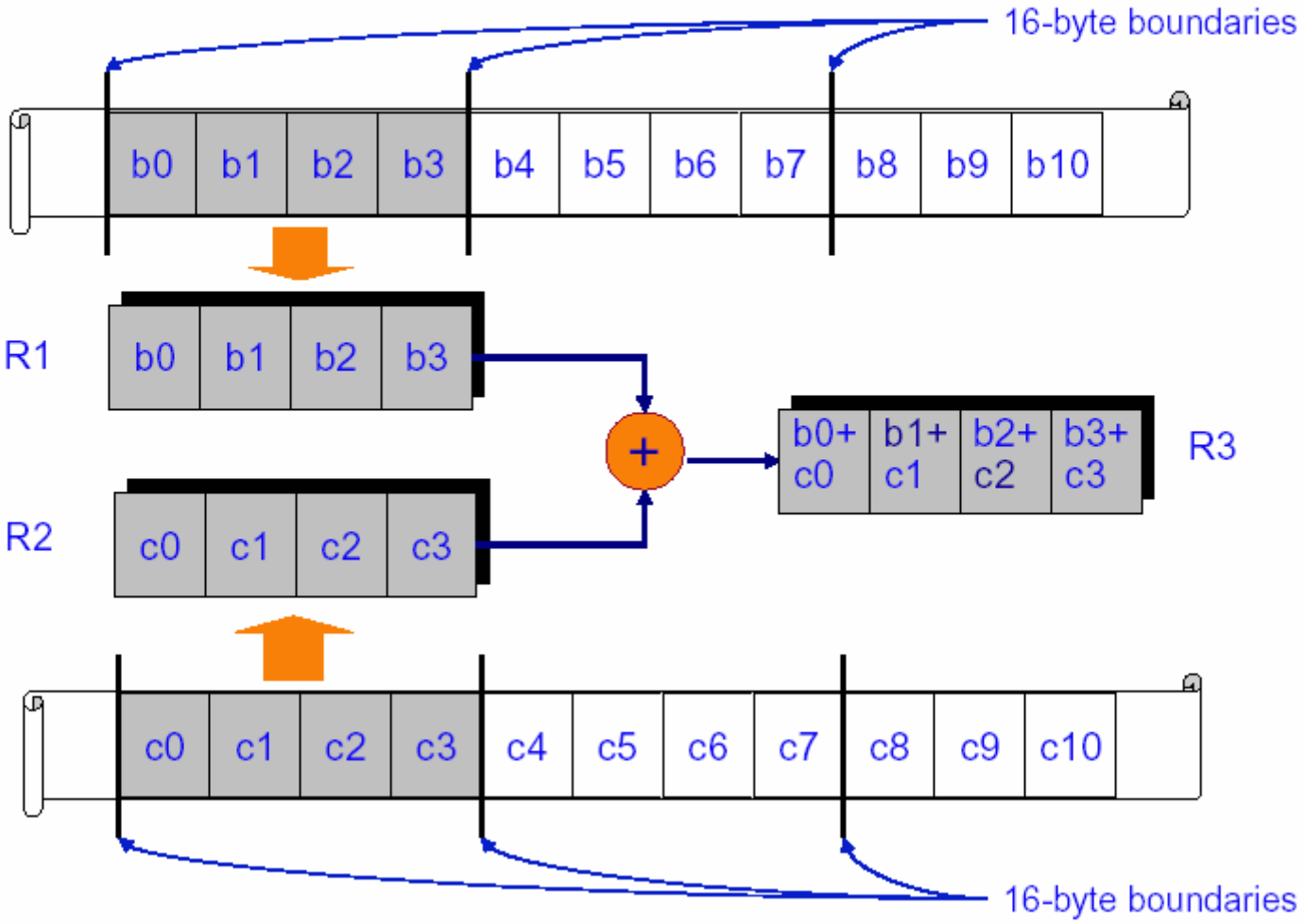
Load / Store by Quadword

- **Scalar loads and stores are slow, with long latency.**
- **SPUs only support quadword loads and stores.**
- **Consider making scalars into quadword integer vectors.**
- **Load or store scalar arrays as quadwords, and perform your own extraction and insertion to eliminate load and store instructions.**

SIMD Programming Tips

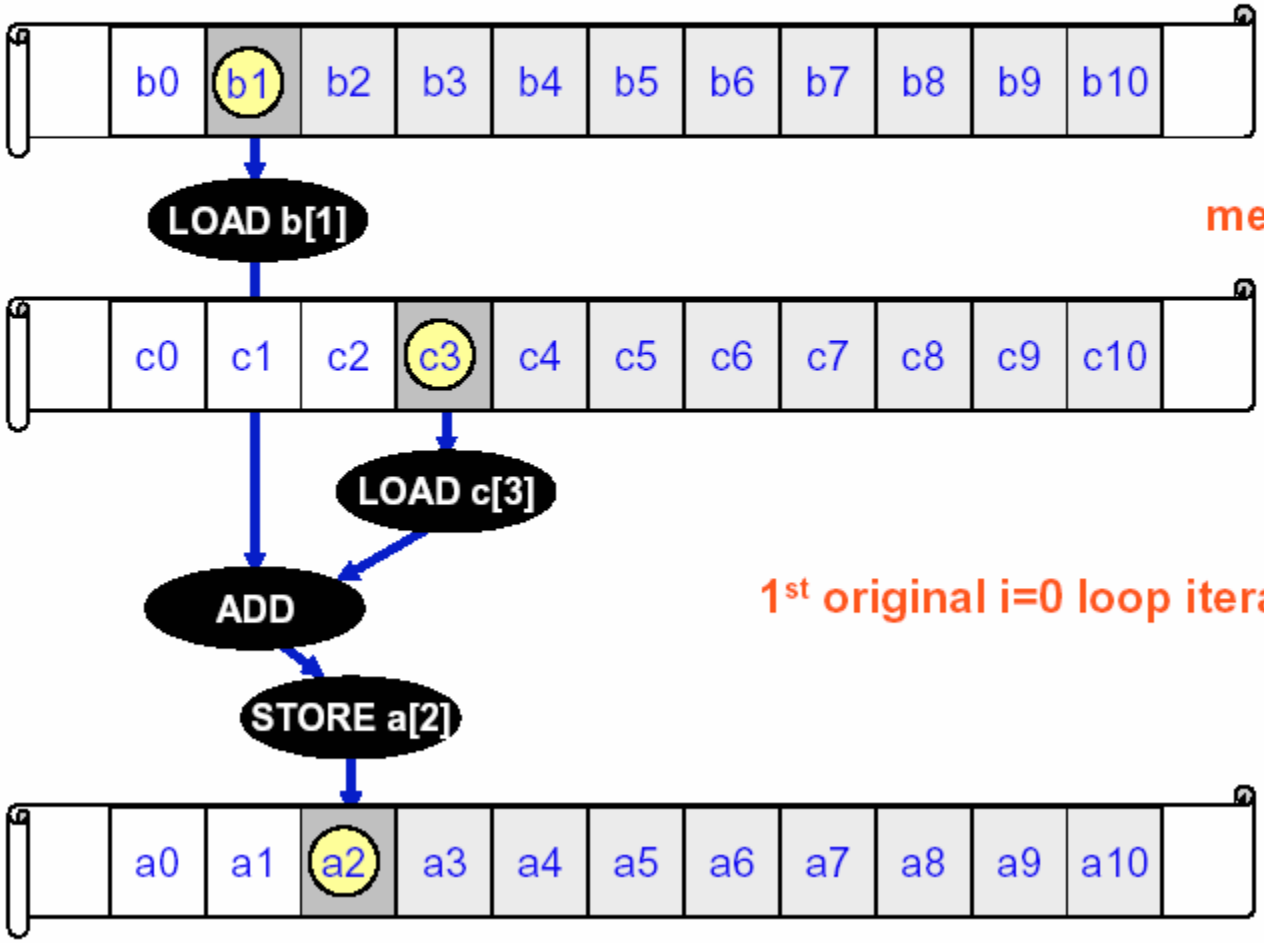
Single Instruction Multiple Data (SIMD) Computation

Process multiple "b[i]+c[i]" data per operations



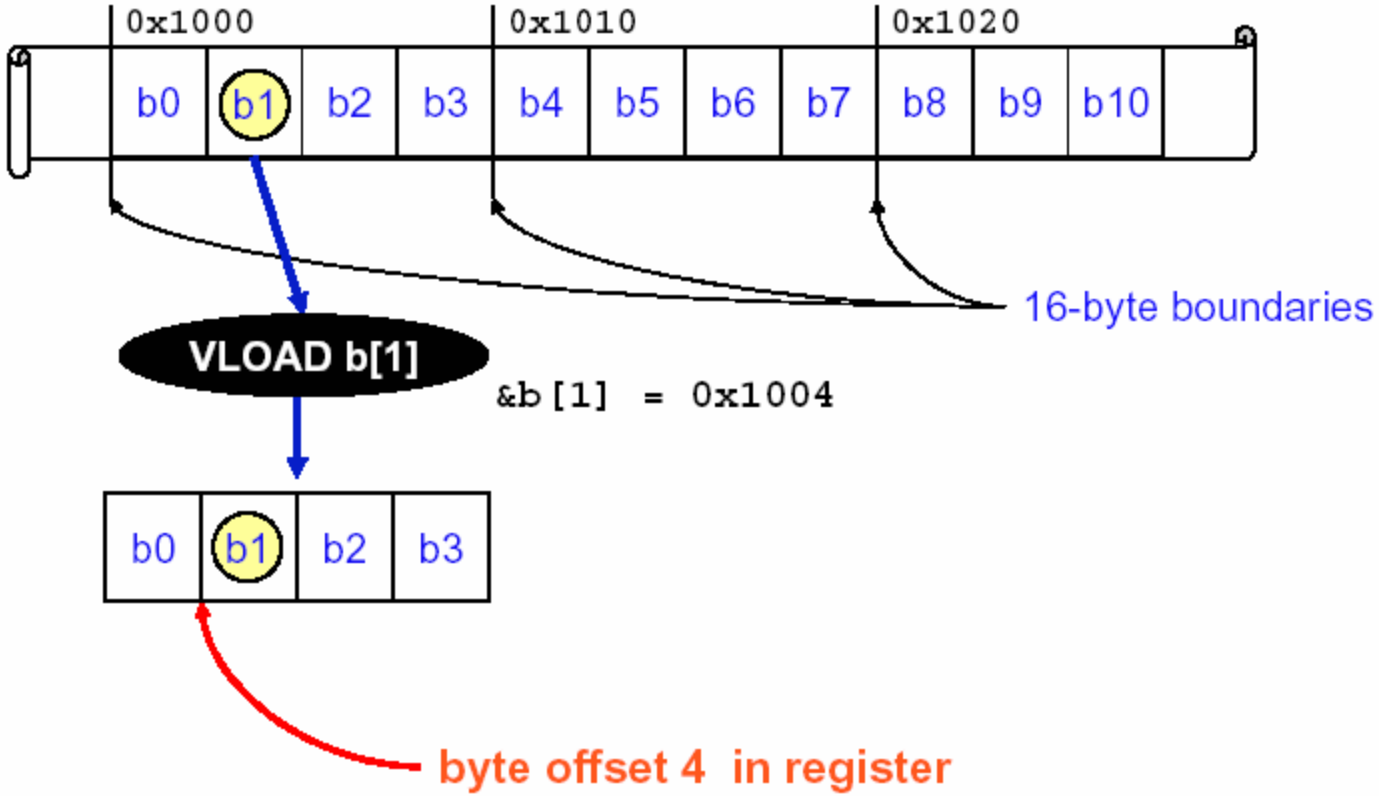
Sequential Execution of a Loop

- **for (i=0; i<100; i++)** a[i+2] = b[i+1] + c[i+3];



SIMD Load/Store Preserve Memory Alignment

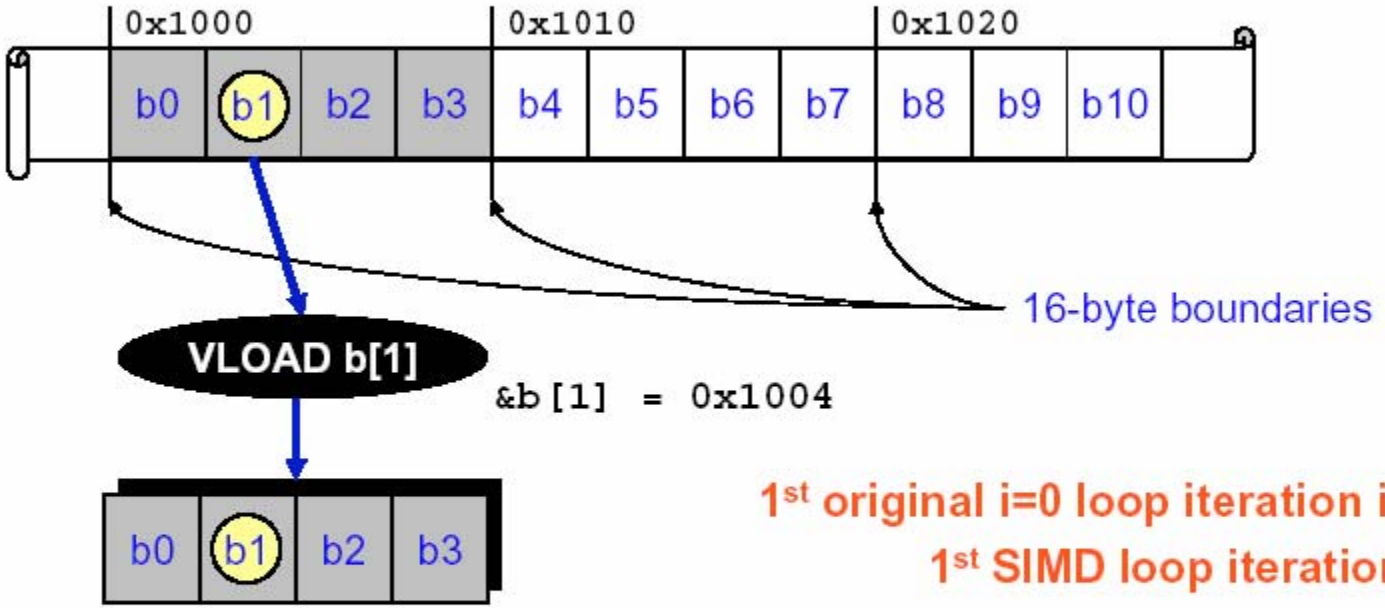
- **Access only a 16-byte chunk of 16-byte aligned data***



* AltiVec/VMX and others; SSE supports unaligned access, but less efficiently

SIMD Load/Store Preserve Memory Alignment

- **Access only a 16-byte chunk of 16-byte aligned data***



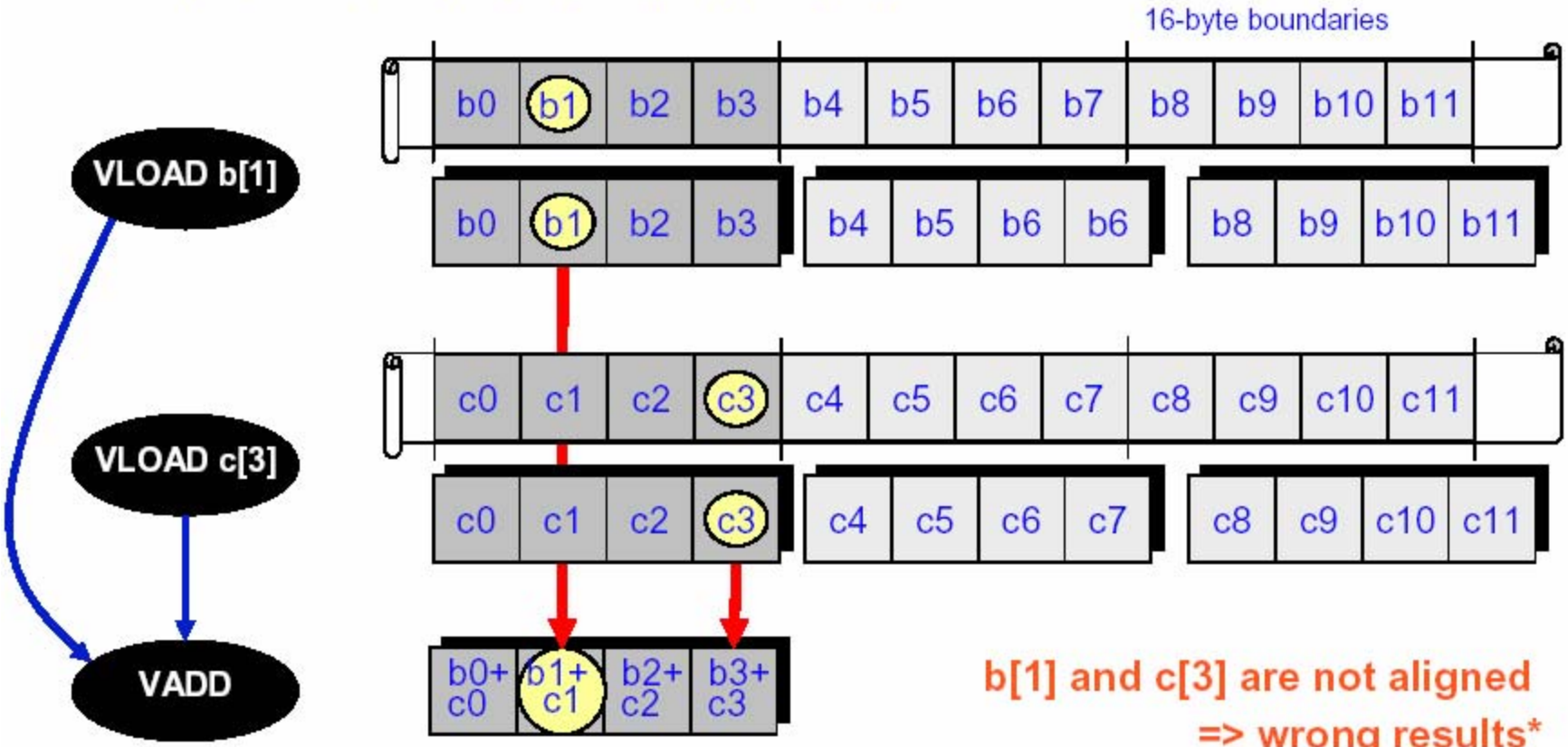
1st original i=0 loop iteration in yellow

1st SIMD loop iteration in grey

* AltiVec/VMX and others; SSE supports unaligned access, but less efficiently

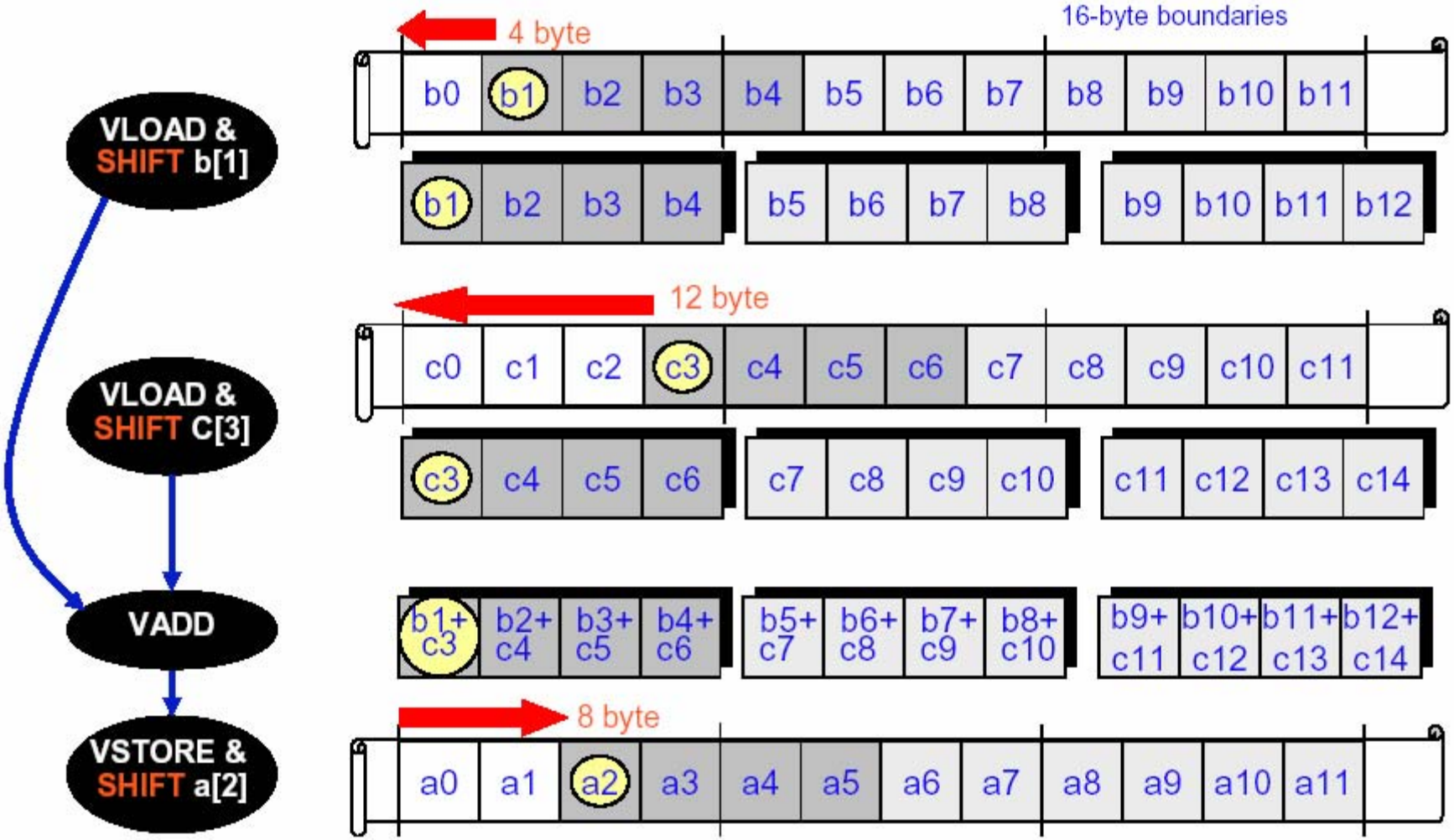
Erroneous SIMD Execution

- **for (i=0; i<100; i++)** a[i+2] = b[i+1] + c[i+3];



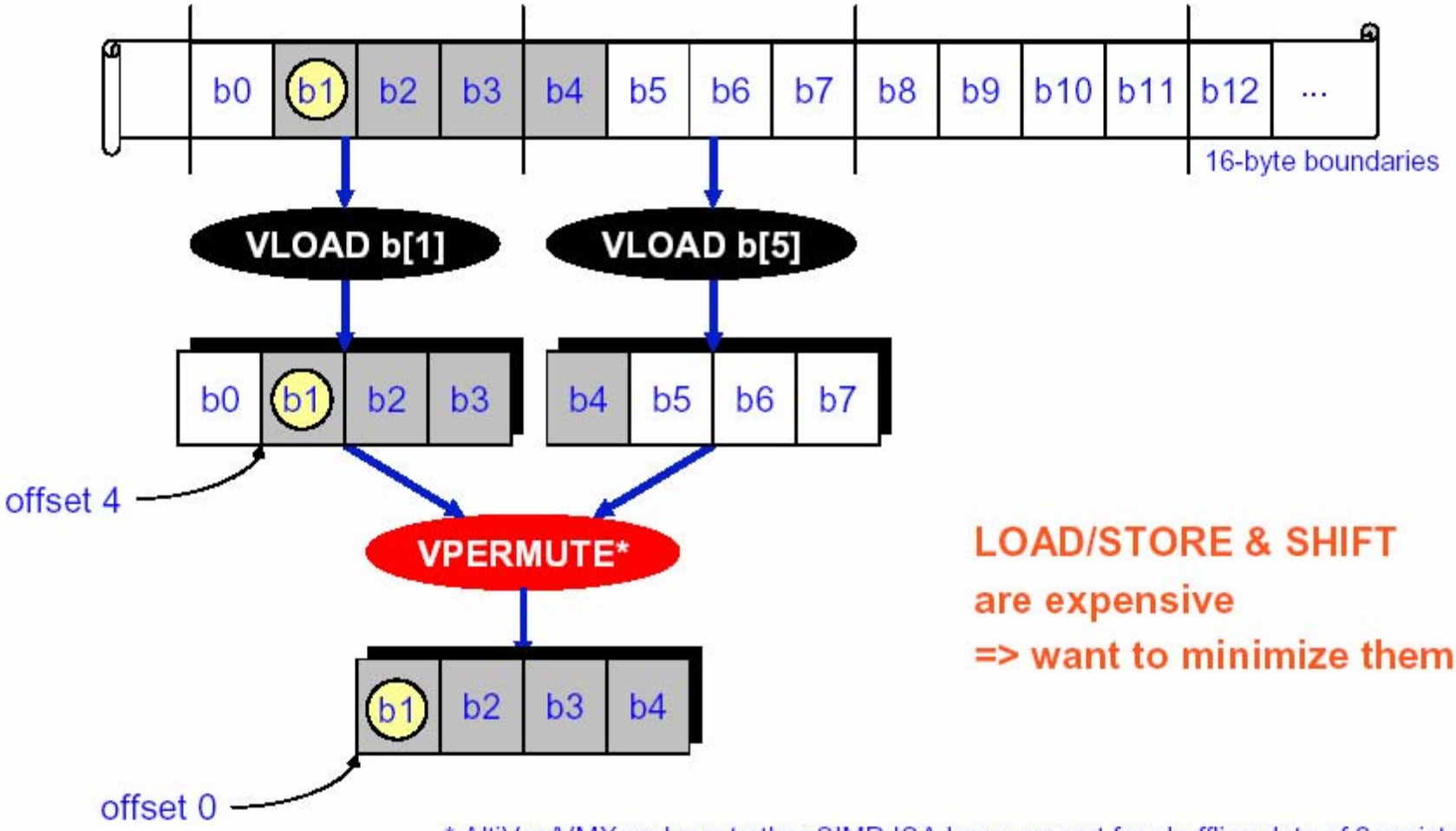
Correct SIMD Execution (Zero-Shift) [VAST 04]

- for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];



How to Align Data in Registers

- Load 4 values from misaligned address b[1]

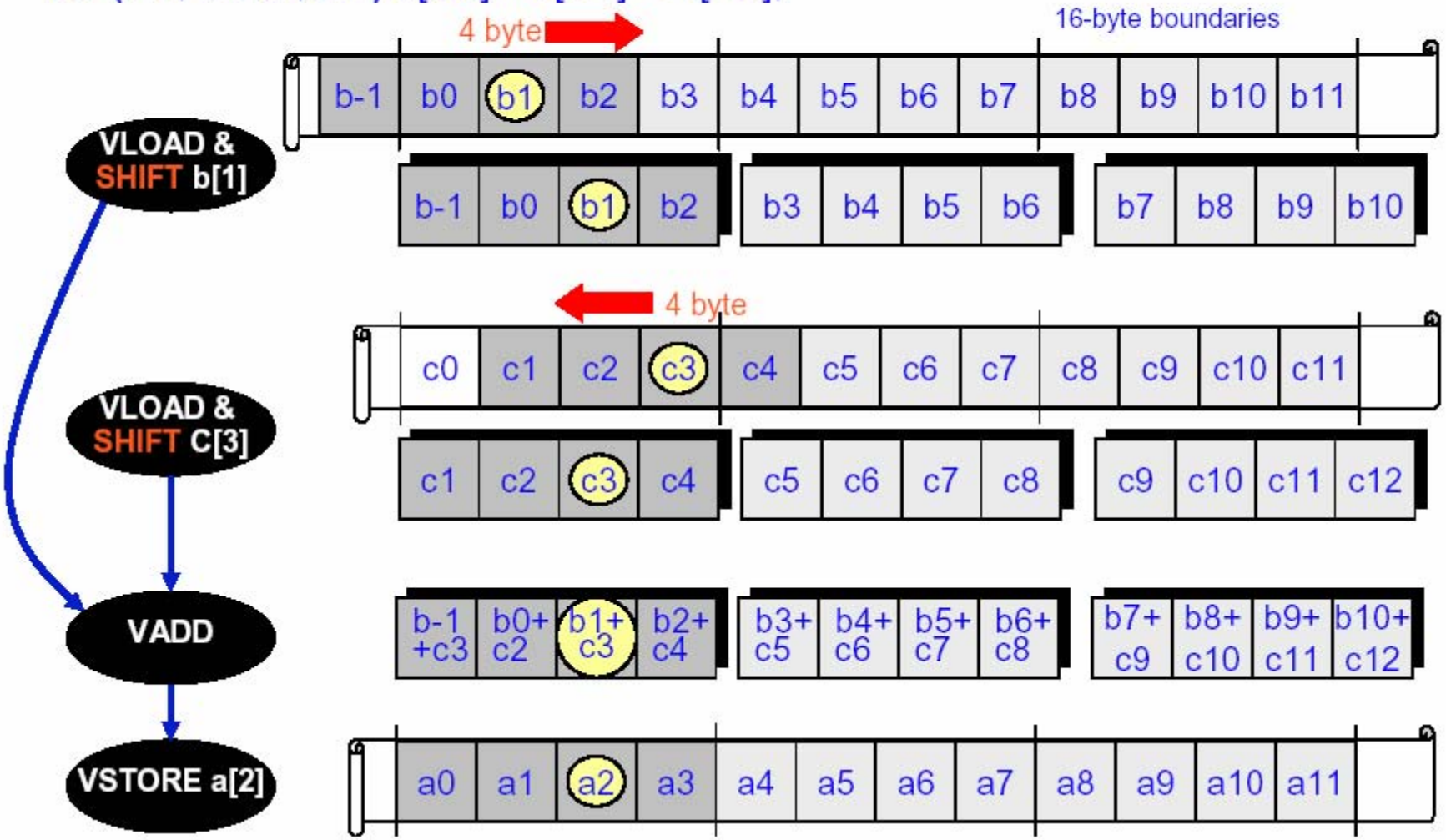


LOAD/STORE & SHIFT
are expensive
=> want to minimize them

* AltiVec/VMX and most other SIMD ISA have support for shuffling data of 2 registers

Better SIMD Execution (Lazy-Shift) [Eichen 04]

```
for (i=0; i<100; i++) a[i+2] = b[i+1]  + c[i+3];
```



Use Offset Pointer

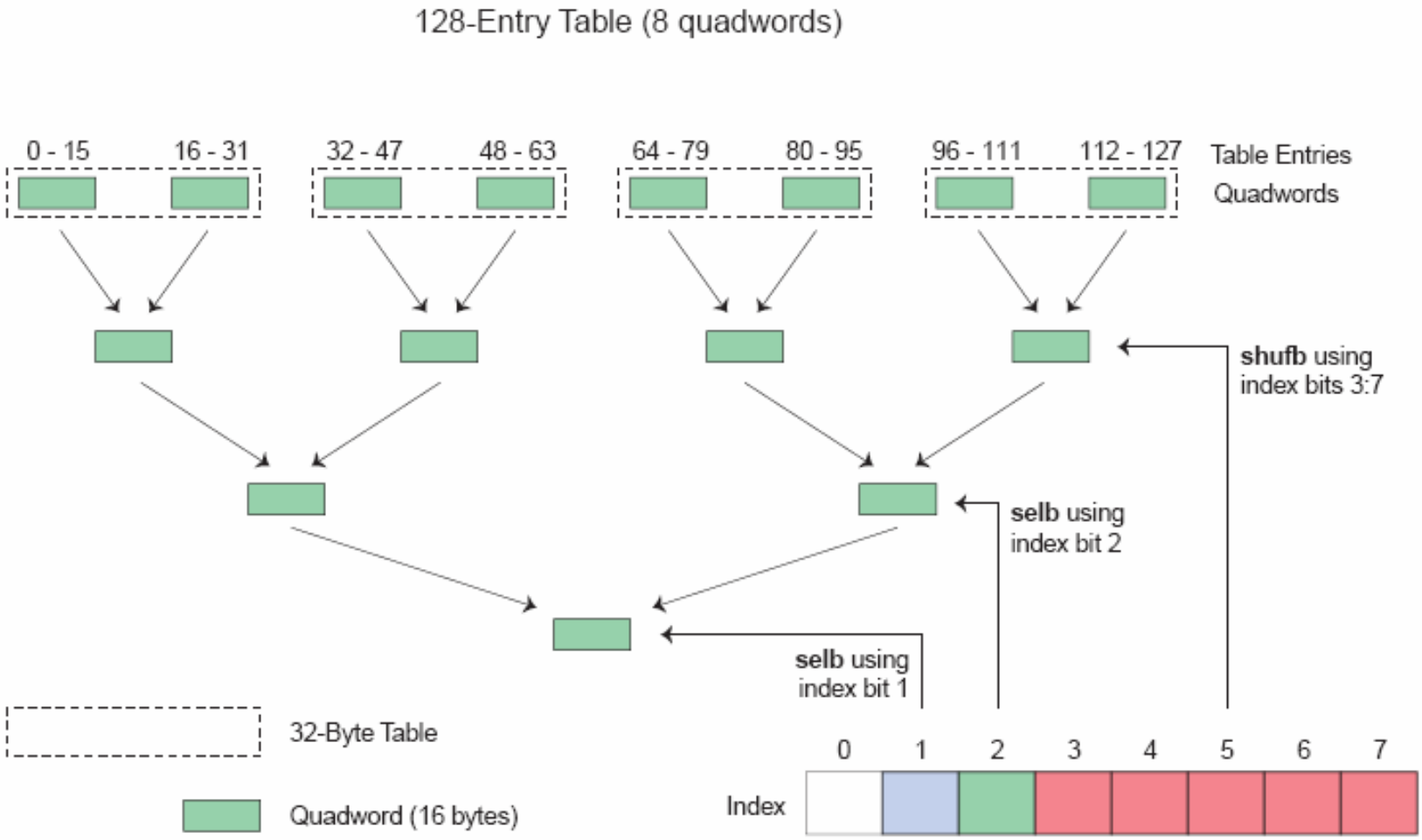
- Use the PPE's load/store with update instructions. These allow sequential indexing through an array without the need of additional instructions to increment the array pointer.
- For the SPEs (which do not support load/store with update instructions), use the d-form instructions to specify an immediate offset from a base array pointer
- For example, consider the following PPE code that exploits the PowerPC store with update instruction:

```
#define FILL_VEC_FLOAT(_q, _data) *(vector float)(_q++) = _data;
FILL_VEC_FLOAT(q, x);
FILL_VEC_FLOAT(q, y);
FILL_VEC_FLOAT(q, z);
FILL_VEC_FLOAT(q, w);
```

- The same code can be modified for SPU execution as follows:

```
#define FILL_VEC_FLOAT(_q, _offset, _data) *(vector float)(_q+(_offset)) = _data;
FILL_VEC_FLOAT(q, 0, x);
FILL_VEC_FLOAT(q, 1, y);
FILL_VEC_FLOAT(q, 2, z);
FILL_VEC_FLOAT(q, 3, w);
q += 4;
```

Shuffle byte instructions for table look-ups



IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is
http://www.chips.ibm.com