

TUTORIAL



Core-Level Performance Engineering

Jan Laukemann, Georg Hager

Erlangen National High Performance Computing Center (NHR@FAU)



<https://go-nhr.de/CLPE>



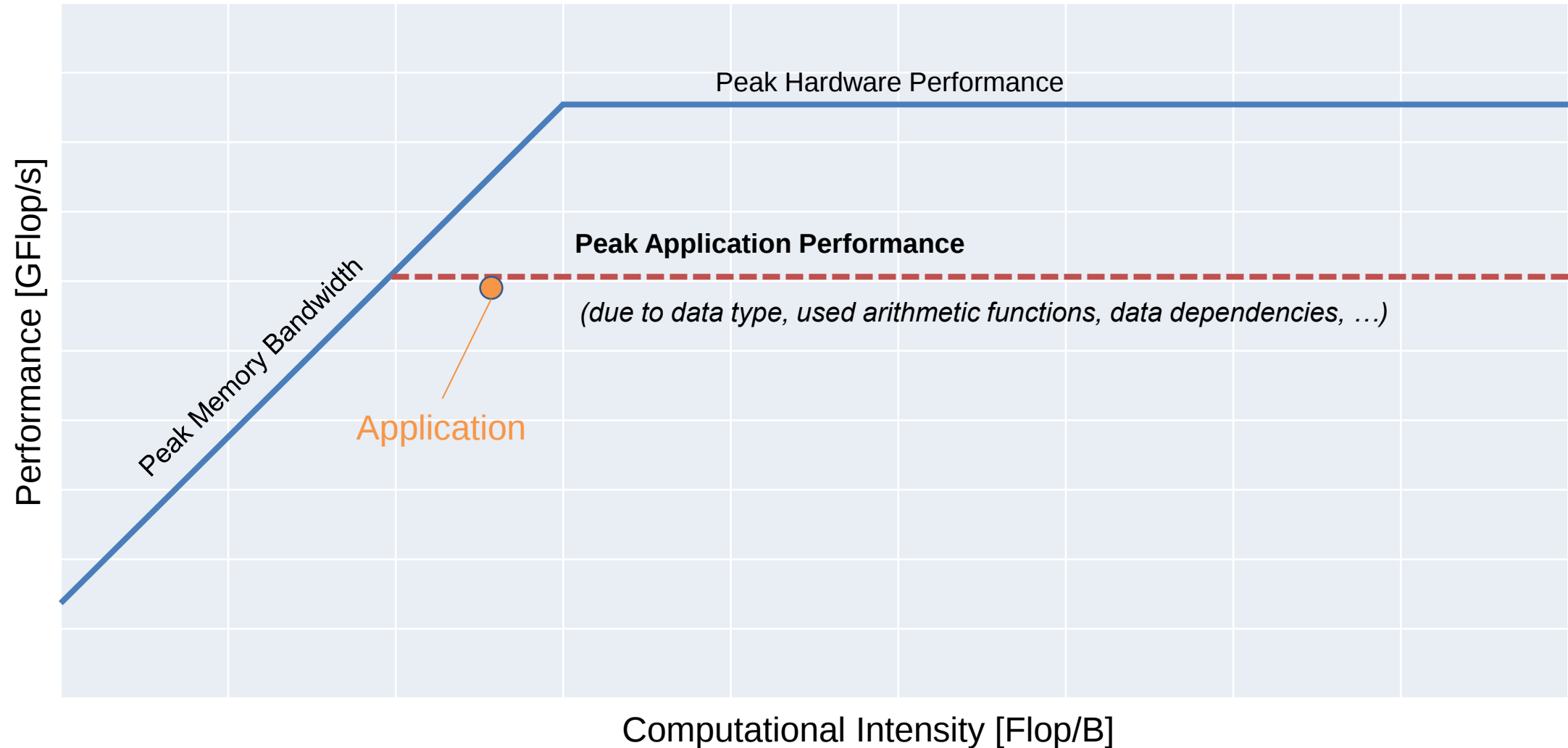
Outline

- Analytical performance modeling
- Basic x86 processor and core architecture
- Code execution on Out-of-order processor cores
- x86 instruction set intro

- Analysis of simple kernels – demo and hands-on
- Introduction to OSACA
- Arm ISA intro
- More complex case studies – demo and hands-on

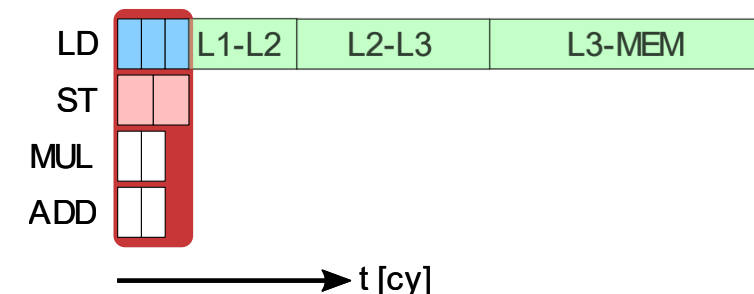
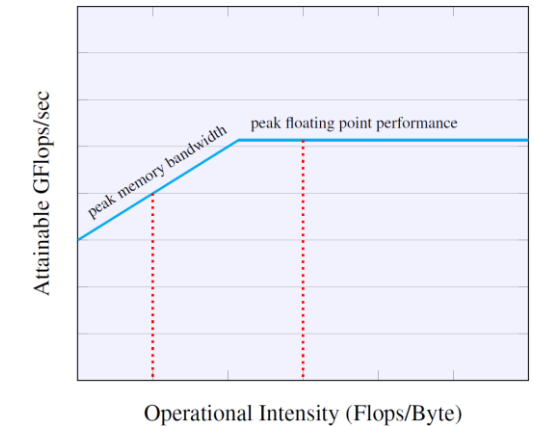
- Summary, caveats, and take-aways

Analytical Performance Modeling



Analytical Performance Modeling

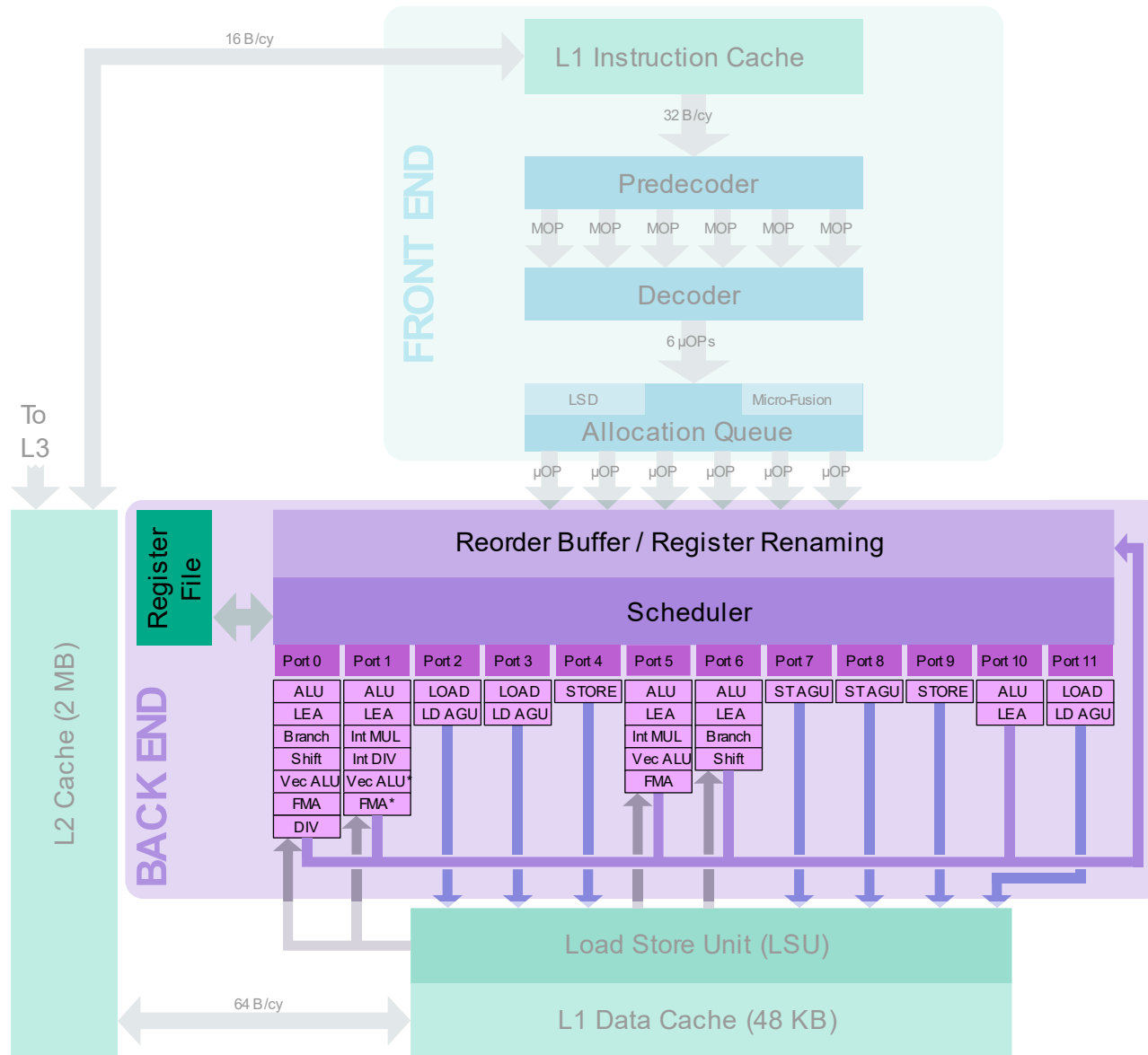
- What is the best performance my code can achieve?
- What are the relevant hardware bottlenecks?
- Apply simplified model of underlying hardware, consisting of
 - In-core execution
 - Data transfer



Basic x86 out-of-order core architecture

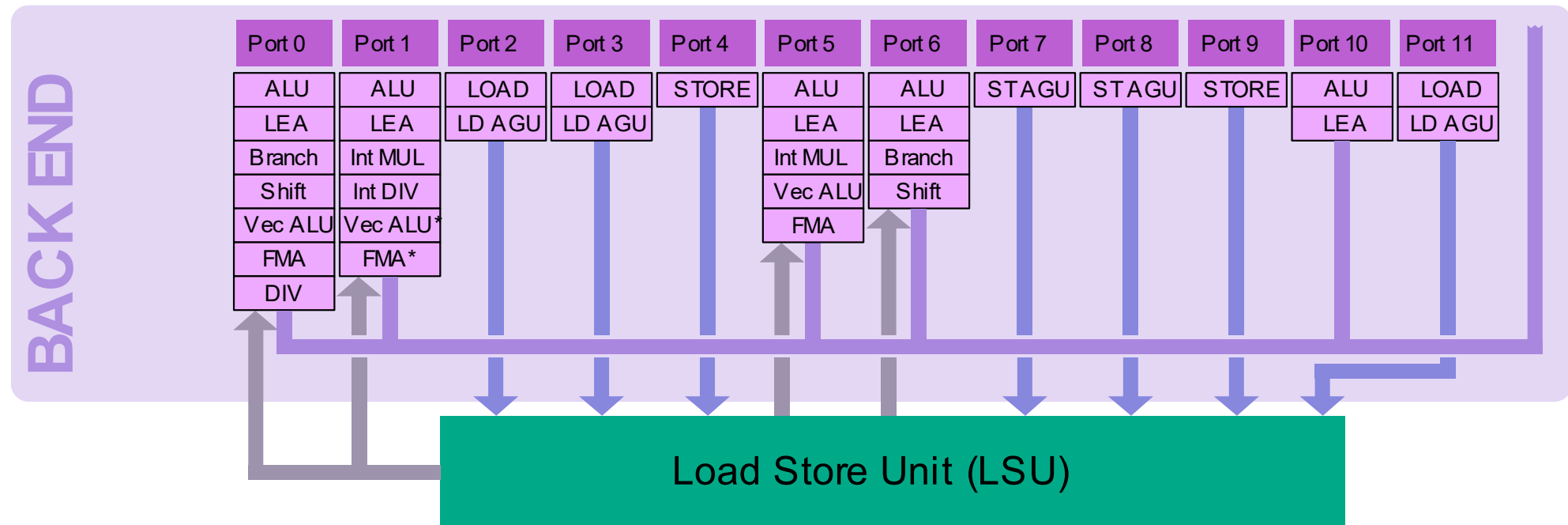
On the example of a Sapphire Rapids chip

Basic processor and core architecture – SPR



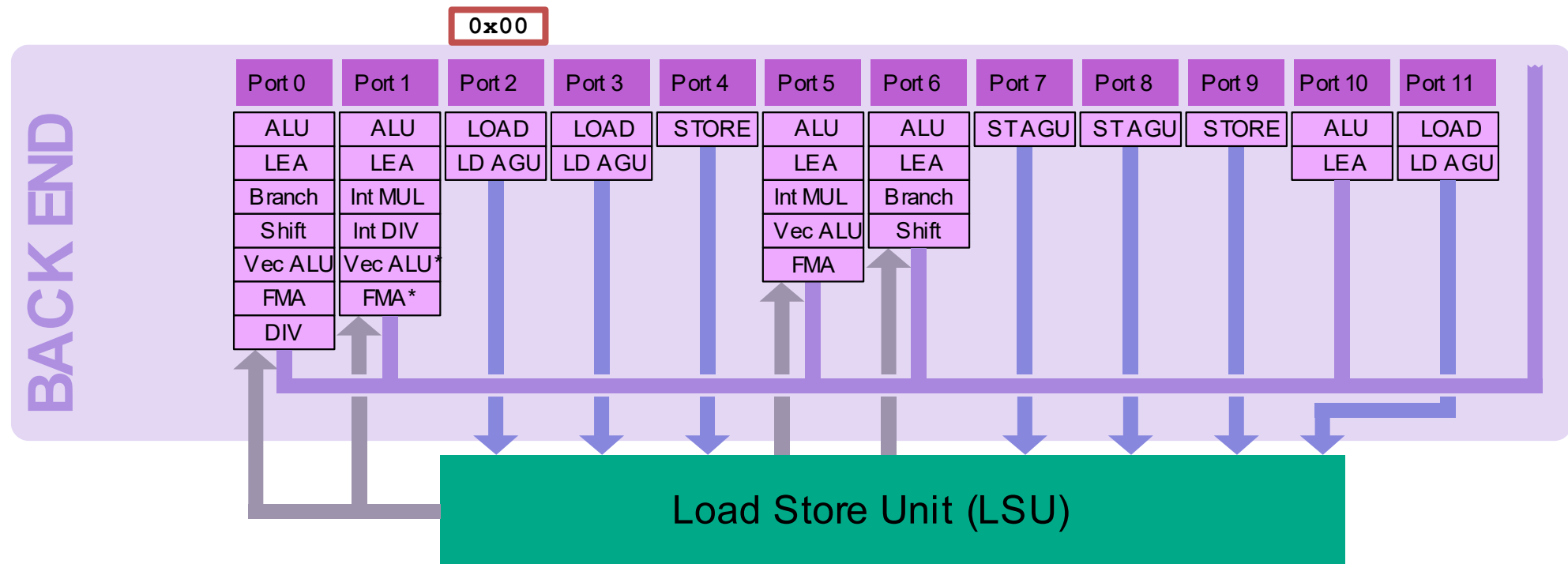
Basic processor and core architecture

0x00	LOAD from address 0x1f8223de to reg1	1cy on 2 3 11
0x04	LOAD from address 0x1f8244de to reg2	1cy on 2 3 11
0x08	ADD reg1 and reg2 and save in reg3	1cy on 0 1 5 6 10
0x0C	STORE reg3 to address 0x2010ff08	1cy on 4 9, 1cy on 7 8



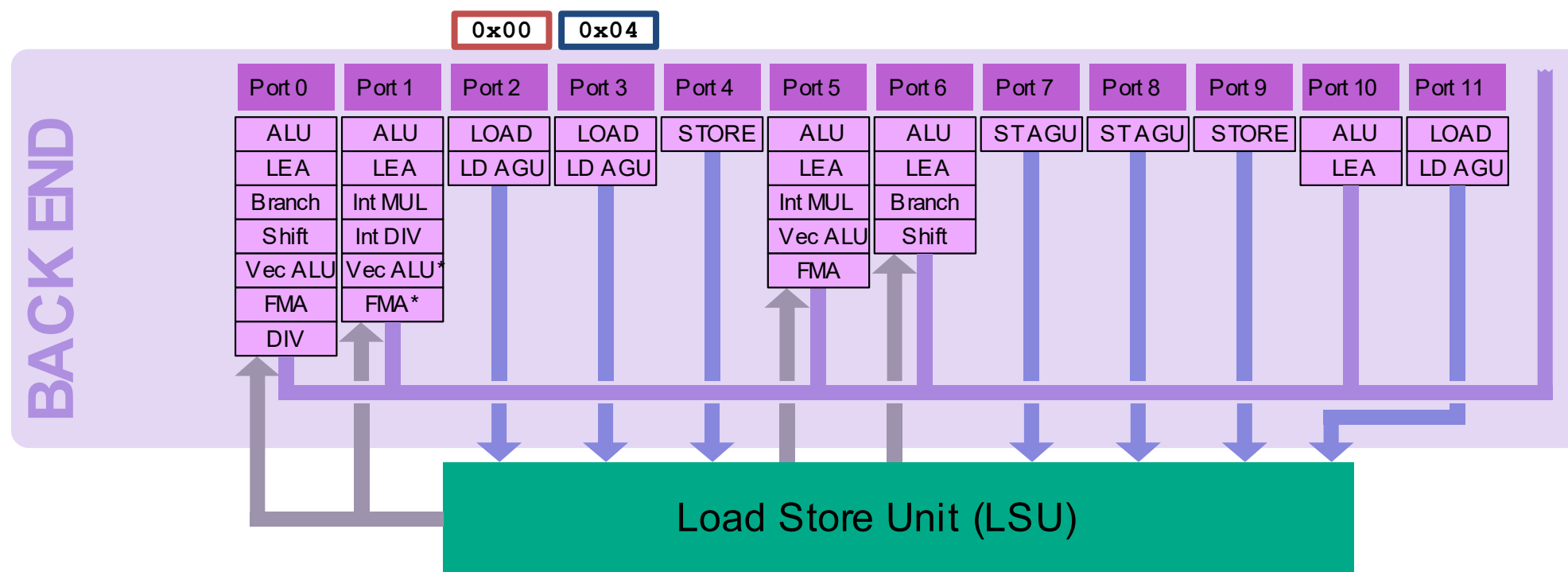
Basic processor and core architecture

0x00	LOAD from address 0x1f8223de to reg1	1cy on 2 3 11
0x04	LOAD from address 0x1f8244de to reg2	1cy on 2 3 11
0x08	ADD reg1 and reg2 and save in reg3	1cy on 0 1 5 6 10
0x0C	STORE reg3 to address 0x2010ff08	1cy on 4 9, 1cy on 7 8



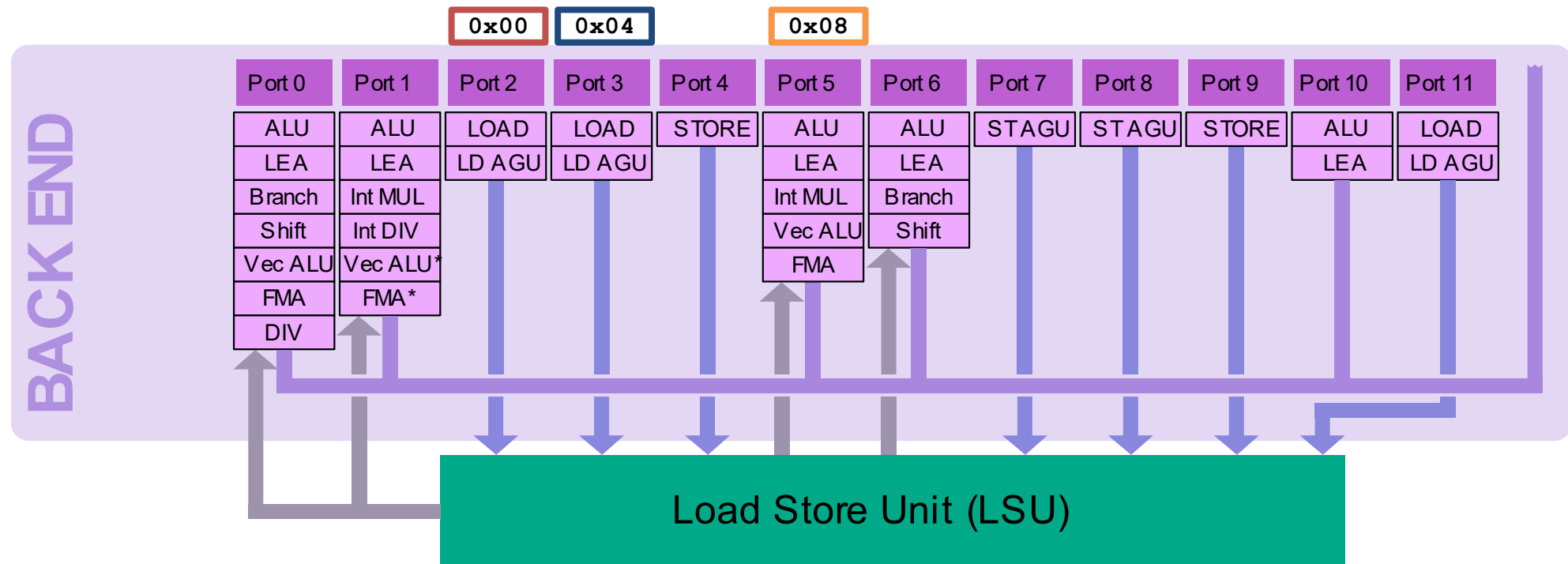
Basic processor and core architecture

0x00	LOAD from address 0x1f8223de to reg1	1cy on 2 3 11
0x04	LOAD from address 0x1f8244de to reg2	1cy on 2 3 11
0x08	ADD reg1 and reg2 and save in reg3	1cy on 0 1 5 6 10
0x0C	STORE reg3 to address 0x2010ff08	1cy on 4 9, 1cy on 7 8



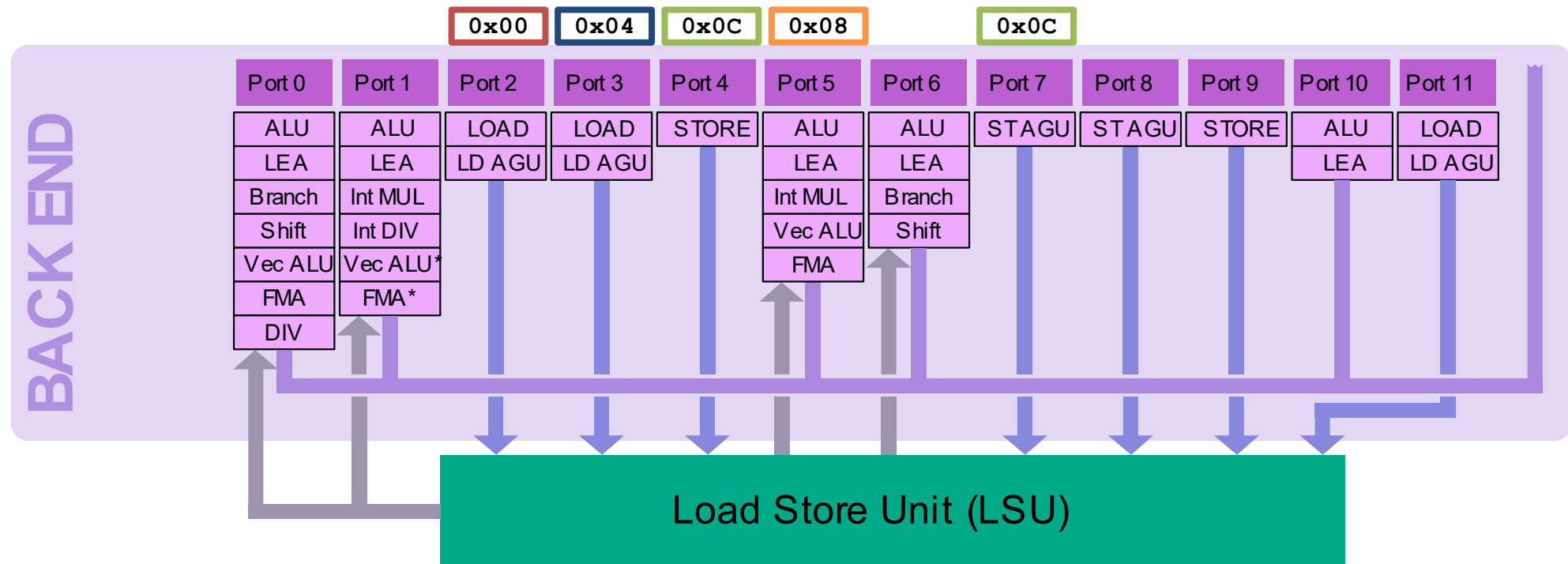
Basic processor and core architecture

0x00	LOAD from address 0x1f8223de to reg1	1cy on 2 3 11
0x04	LOAD from address 0x1f8244de to reg2	1cy on 2 3 11
0x08	ADD reg1 and reg2 and save in reg3	1cy on 0 1 5 6 10
0x0C	STORE reg3 to address 0x2010ff08	1cy on 4 9, 1cy on 7 8



Basic processor and core architecture

0x00	LOAD from address 0x1f8223de to reg1	1cy on 2 3 11
0x04	LOAD from address 0x1f8244de to reg2	1cy on 2 3 11
0x08	ADD reg1 and reg2 and save in reg3	1cy on 0 1 5 6 10
0x0C	STORE reg3 to address 0x2010ff08	1cy on 4 9, 1cy on 7 8



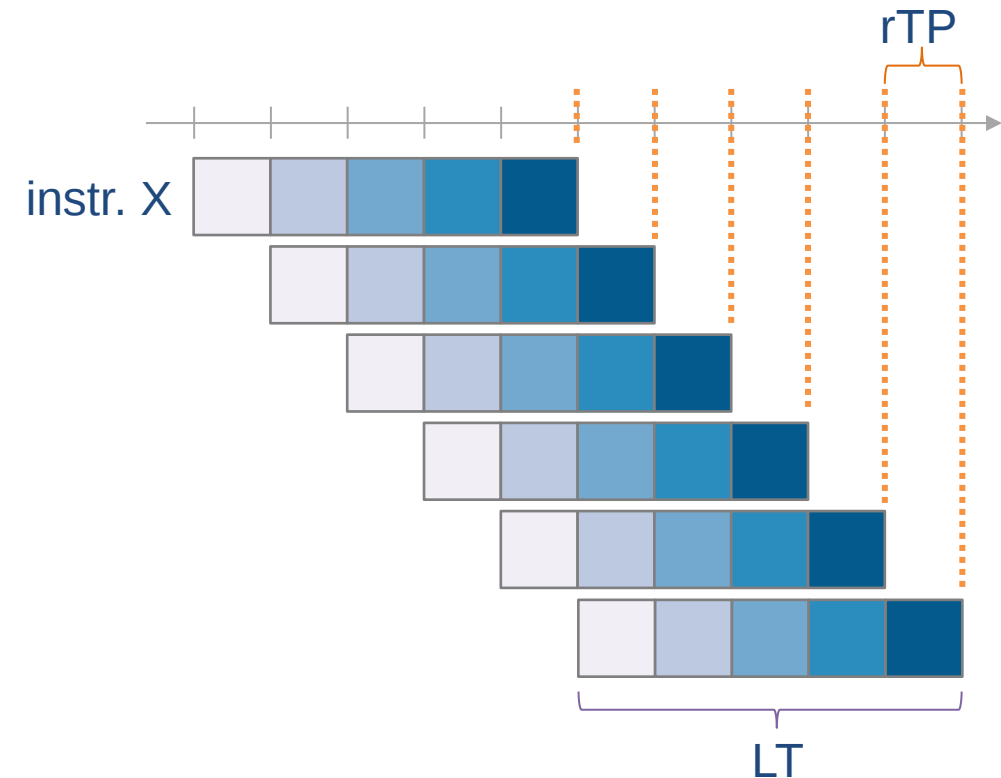
Code execution on out-of-order CPUs

Terminology and explanation

Code execution on OoO processor cores

Three metrics to estimate the in-core runtime:

- **Reciprocal Throughput (rTP)**
- **Latency (LT) and Critical Path (CP)**
- **Loop-carried dependencies (LCD)**



Simplified runtime estimation: $t_c = \max(t_{rTP}, t_{LCD})$

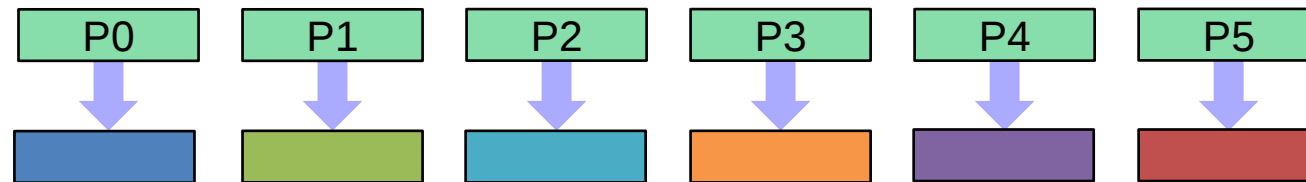
Code execution on OoO processor cores

- Simple HW model:
 - Six types of functional units (i.e., types of instructions), each functional unit (FU) assigned to one port:



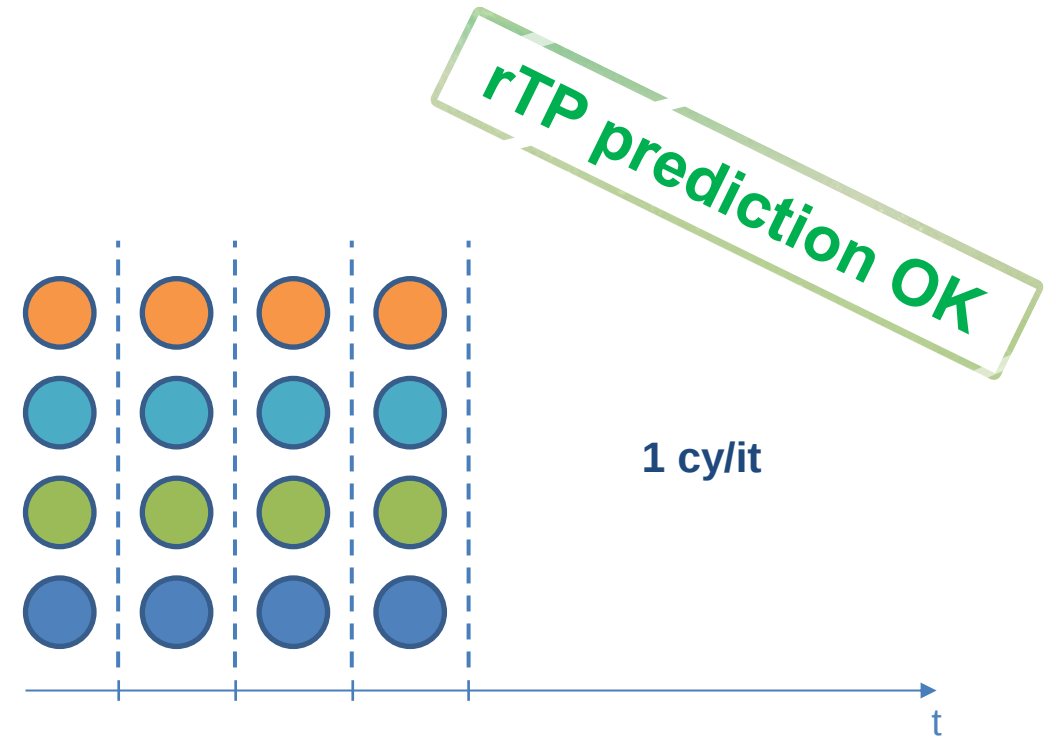
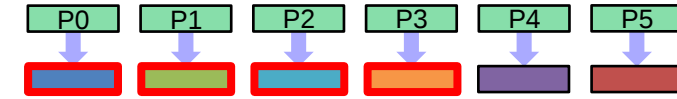
- Reciprocal throughput for each instruction: 1cy
- Latency for each instruction: 1cy

- Port model:



Code execution on OoO processor cores

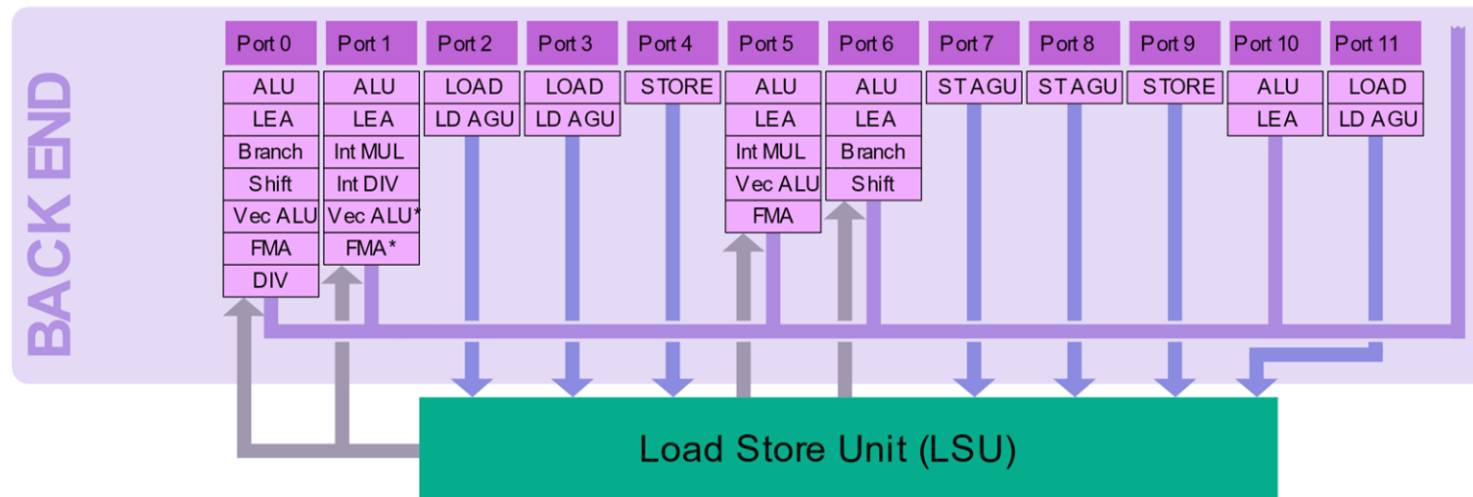
- Loop 1: 
 - No dependencies within loop
 - No intra-loop dependencies
- rTP prediction: 1 cy
- CP prediction: 1 cy
- LCD prediction: -



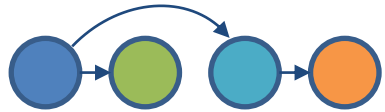
Remember slides 7-11?

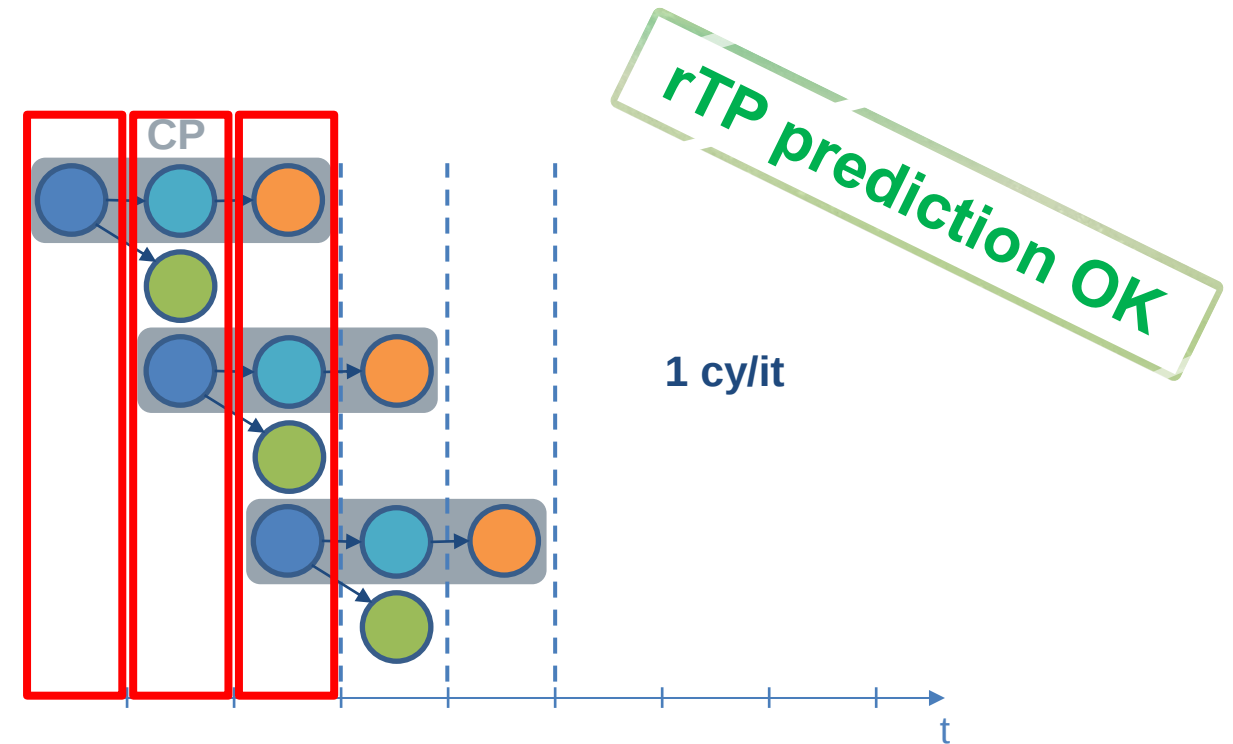
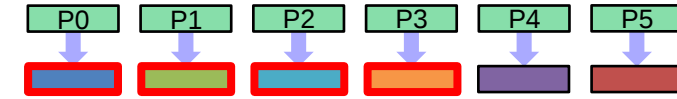
Basic processor and core architecture

0x00 LOAD from address 0x1f8223de to reg1 1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2 1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3 1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08 1cy on 4|9, 1cy on 7|8



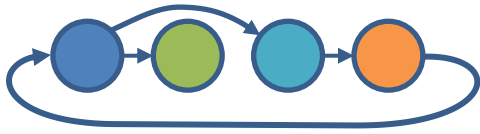
Code execution on OoO processor cores

- Loop 2: 
 - Dependencies within loop body
 - No loop-carried dependencies
- rTP prediction: 1 cy
- CP prediction: 3 cy
- LCD prediction: -

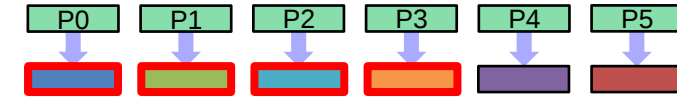


Code execution on OoO processor cores

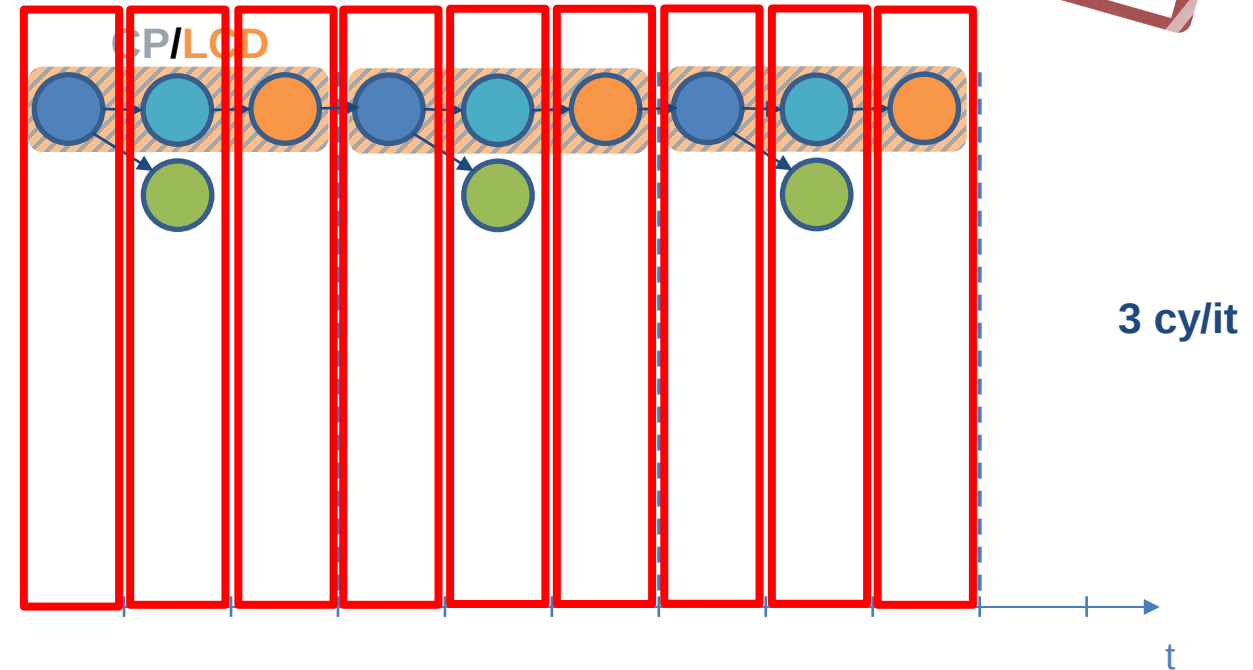
- Loop 3:



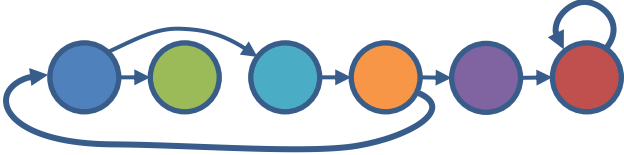
- Dependencies within loop body
- Loop-carried dependency
- rTP prediction: 1 cy
- CP prediction: 3 cy
- LCD prediction: 3 cy



*rTP prediction
not sufficient*

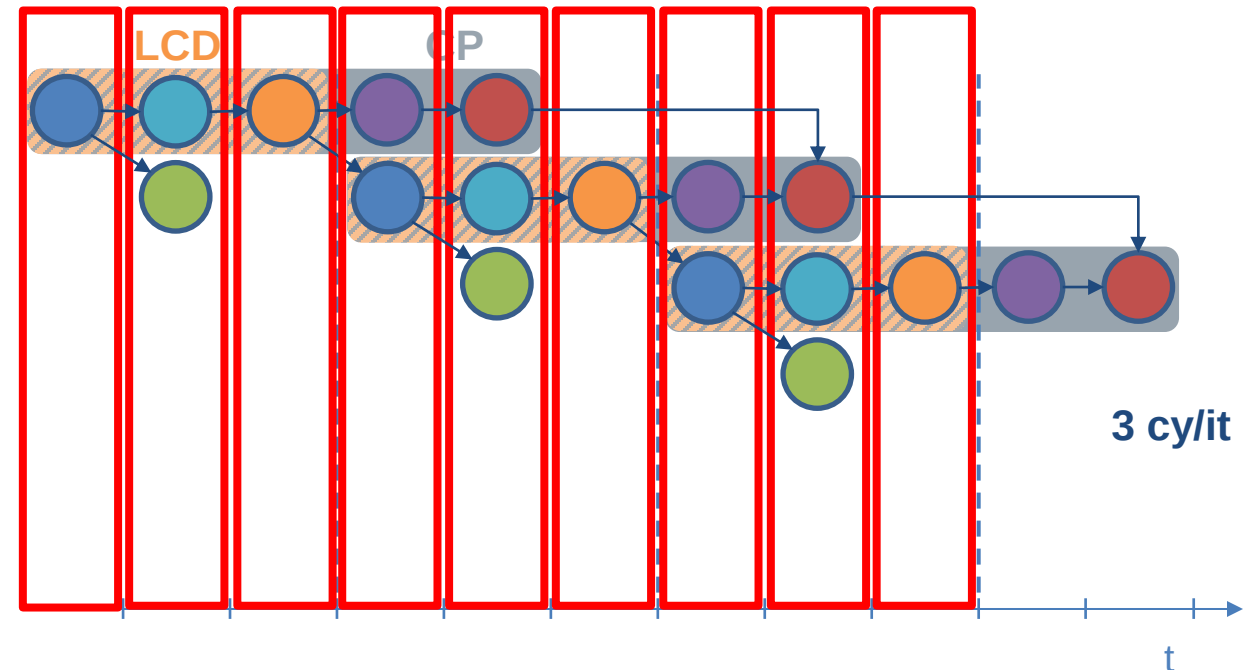
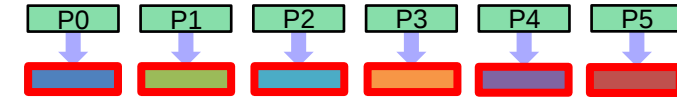


Code execution on OoO processor cores

- Loop 4: 
- Dependencies within loop body
- Loop-carried dependency

- rTP prediction: 1 cy
 - CP prediction: 5 cy
 - LCD prediction: 3 cy
- ≠

- Other limitations:
 - Reorder buffer
 - Loop length
 - Resources (not enough ports, ...)
 - Decoder
 - Data
 - ...



Introduction to the x86 ISA (Instruction Set Architecture)

Basics of the x86-64 ISA

- Operands can be registers (%), memory references ((...)) or immediates (\$)
- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing mode:
 - Intel: `[ BASE + INDEX * SCALE + OFFSET ]`
 - AT&T: `OFFSET ( BASE, INDEX, SCALE )`
- C: `A[i]` equivalent to `*(A+i)` (a pointer has the type: `A+i*8`)
- Suffixes: AT&T often uses (optional) suffixes based on the operand size
 - **b** (byte): 8 bits, **w** (word): 16 bits, **l** (long): 32 bits, **q** (quad): 64 bits

Intel syntax

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

AT&T syntax

```
movaps    %xmm3, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

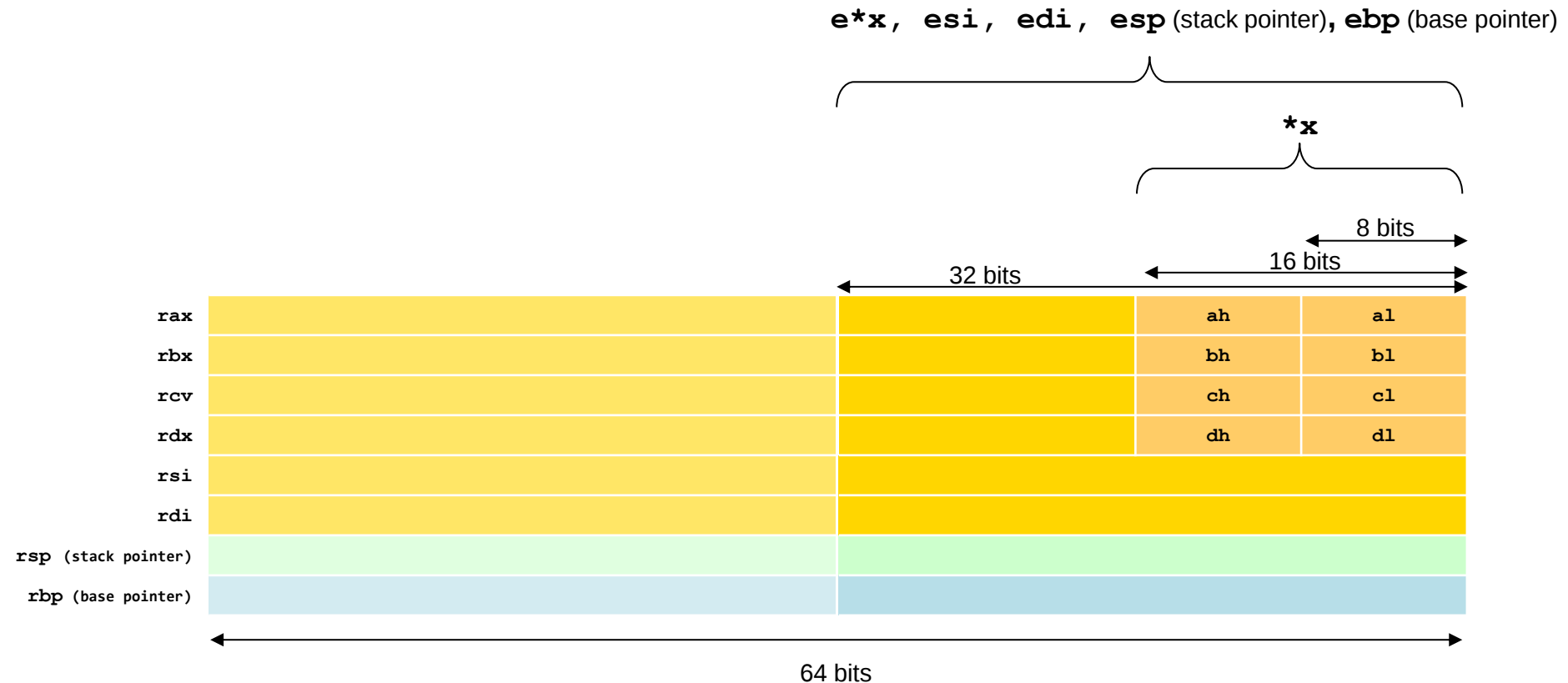
Basics of the x86-64 ISA with extensions

16 general purpose registers (64bit):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32-bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`



Basics of the x86-64 ISA with extensions

SIMD vector registers (aliased):

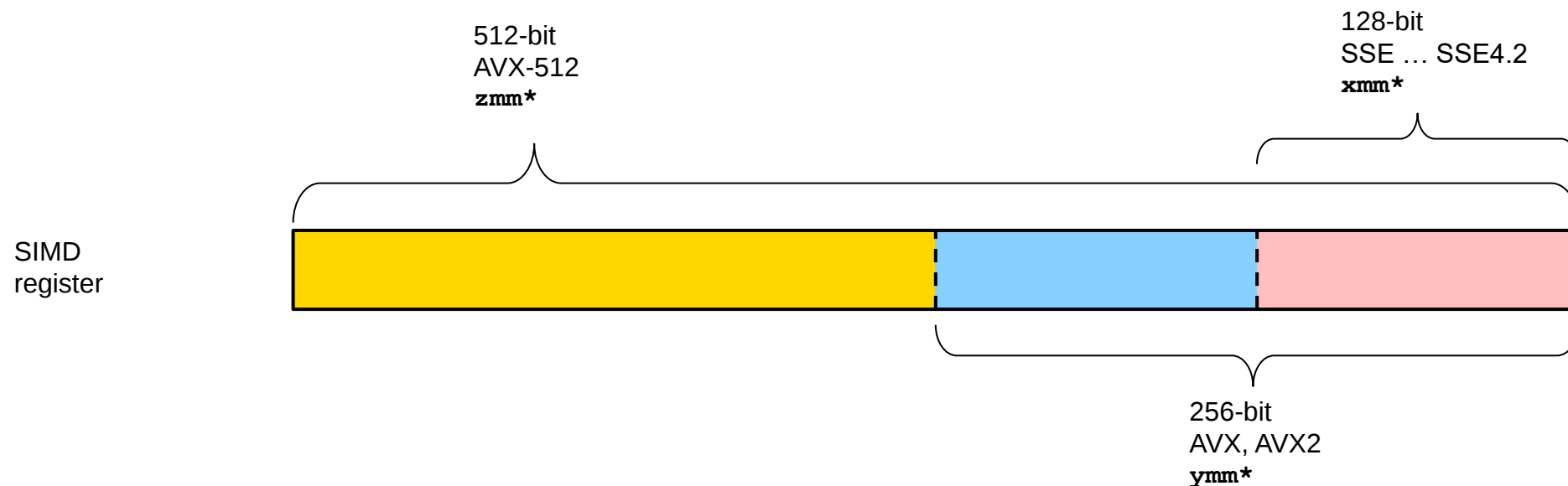
xmm0–xmm15 (. . . **xmm31**) SSE (128bit)

ymm0–ymm15 (. . . **ymm31**) AVX (256bit)

zmm0–zmm31 AVX-512 (512bit)

8 opmask registers (64 bit, AVX512 only):

k0–k7



Basics of the x86-64 ISA with extensions

SIMD instructions are distinguished by:

VEX/EVEX prefix: **v**
Operation: **mul, add, fmadd, mov**
Modifier: nontemporal (**nt**), unaligned (**u**), aligned (**a**), high (**h**), low (**l**)
Width: scalar (**s**), packed (**p**)
Data type: single (**s**), double (**d**)
... and many more

Examples:

vmulpd	→	Multiply Packed Double-Precision Floating-Point Values
vfmadd213ps	→	Fused Multiply-Add of Packed Single-Precision Floating-Point Values
addsd	→	Add Scalar Double-Precision Floating-Point Values
vmovntdq	→	Store Packed Integers Using Non-Temporal Hint

Case Study: Sum reduction (DP)

```
double sum = 0.0;

for (int i=0; i<size; i++) {
    sum += data[i];
}
```

To get object code use
`objdump -d` on object file or
executable or compile with `-S`

Assembly code w/ `-O1` (AT&T syntax, Intel compiler):

```
.label:
    addsd    0(%rdi, %rax, 8), %xmm0
    inc      %rax
    cmp      %rsi, %rax
    jl       .label
```

Intel syntax:

`addsd xmm0, [rdi + rax * 8]`

Sum reduction (DP) – AVX512

Assembly code w/ `-O3 -xCORE-AVX512 -qopt-zmm-usage=high` :

`..B3.28:`

```
vaddpd    (%r13,%rcx,8), %zmm5, %zmm5
vaddpd    64(%r13,%rcx,8), %zmm4, %zmm4
vaddpd    128(%r13,%rcx,8), %zmm3, %zmm3
vaddpd    192(%r13,%rcx,8), %zmm2, %zmm2
addq      $32, %rcx
cmpq      %rbx, %rcx
jb        ..B3.28
```

Bulk loop code
(8x4-way unrolled)

`..B3.29:`

```
vaddpd    %zmm4, %zmm5, %zmm4
vaddpd    %zmm2, %zmm3, %zmm2
vaddpd    %zmm2, %zmm4, %zmm5
```

`# [... SNIP ...]`

← Remainder omitted

`..B3.34:`

```
vshuff32x4 $238, %zmm5, %zmm5, %zmm2
vaddpd    %zmm5, %zmm2, %zmm3
vpermpd   $78, %zmm3, %zmm4
vaddpd    %zmm4, %zmm3, %zmm5
vpermpd   $177, %zmm5, %zmm6
vaddpd    %zmm6, %zmm5, %zmm7
vaddsd    %xmm1, %xmm7, %xmm1
```

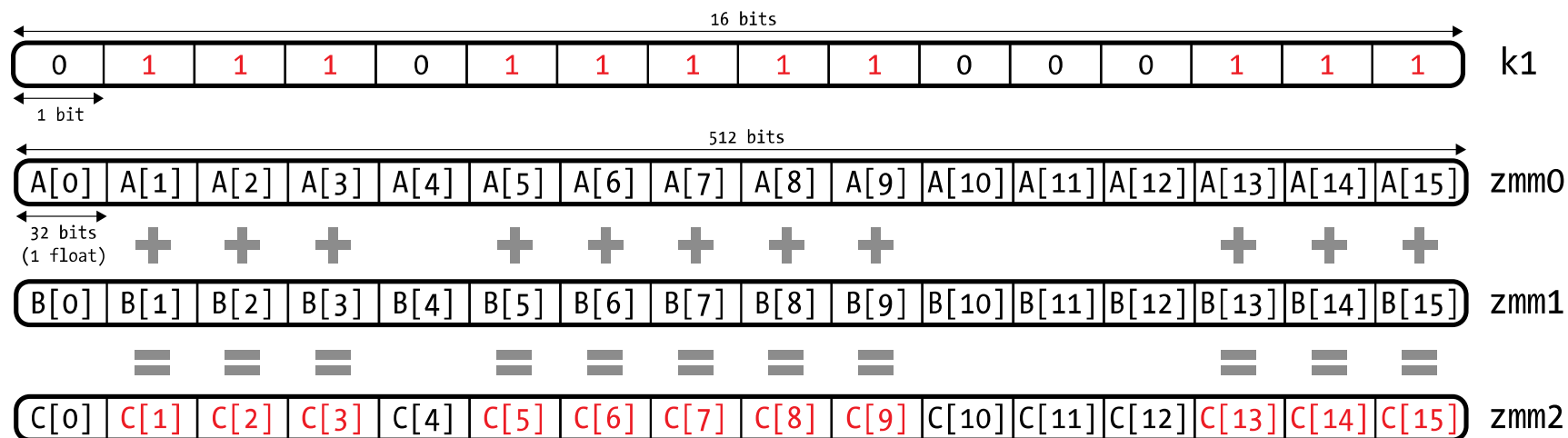
Sum up 32
partial sums into
`xmm1`

Example for masked execution

Masking is very helpful in cases such as, e.g., remainder loop handling or conditionals

Available on x86 starting with AVX-512

Example: `vaddps %zmm0, %zmm1, %zmm2 {%k1}`



STREAM Triad

A pen & paper in-core analysis

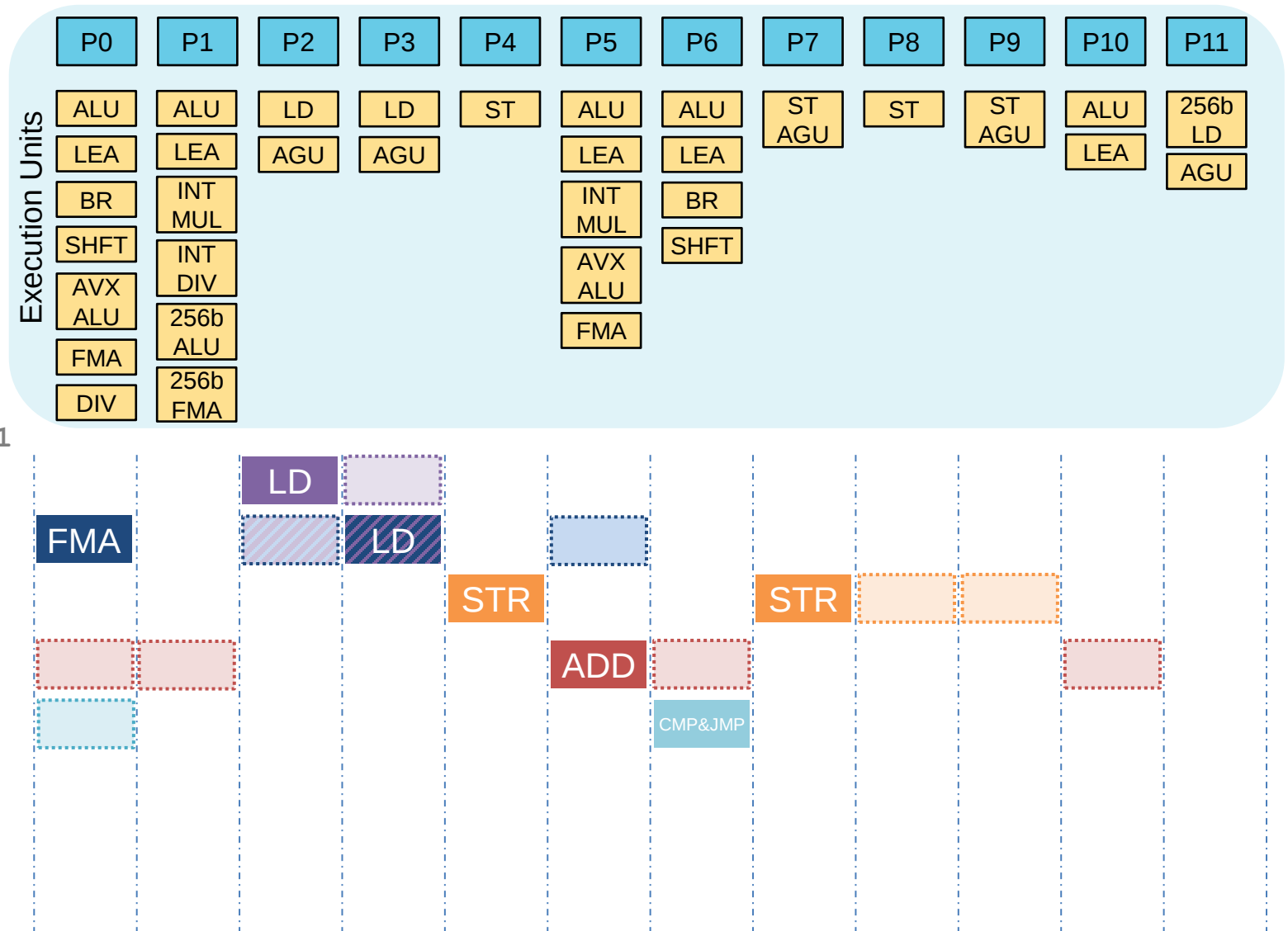
STREAM TRIAD on Intel Sapphire Rapids

STREAM TRIAD

$a[i] = b[i] + s * c[i]$

..B2.42:

```
vmovups    (%r14,%rdx,8), %zmm1
vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
vmovupd    %zmm1, (%r12,%rdx,8)
addq       $8, %rdx
cmpq       %rsi, %rdx
jb         ..B2.42
```



STREAM TRIAD on Intel Sapphire Rapids

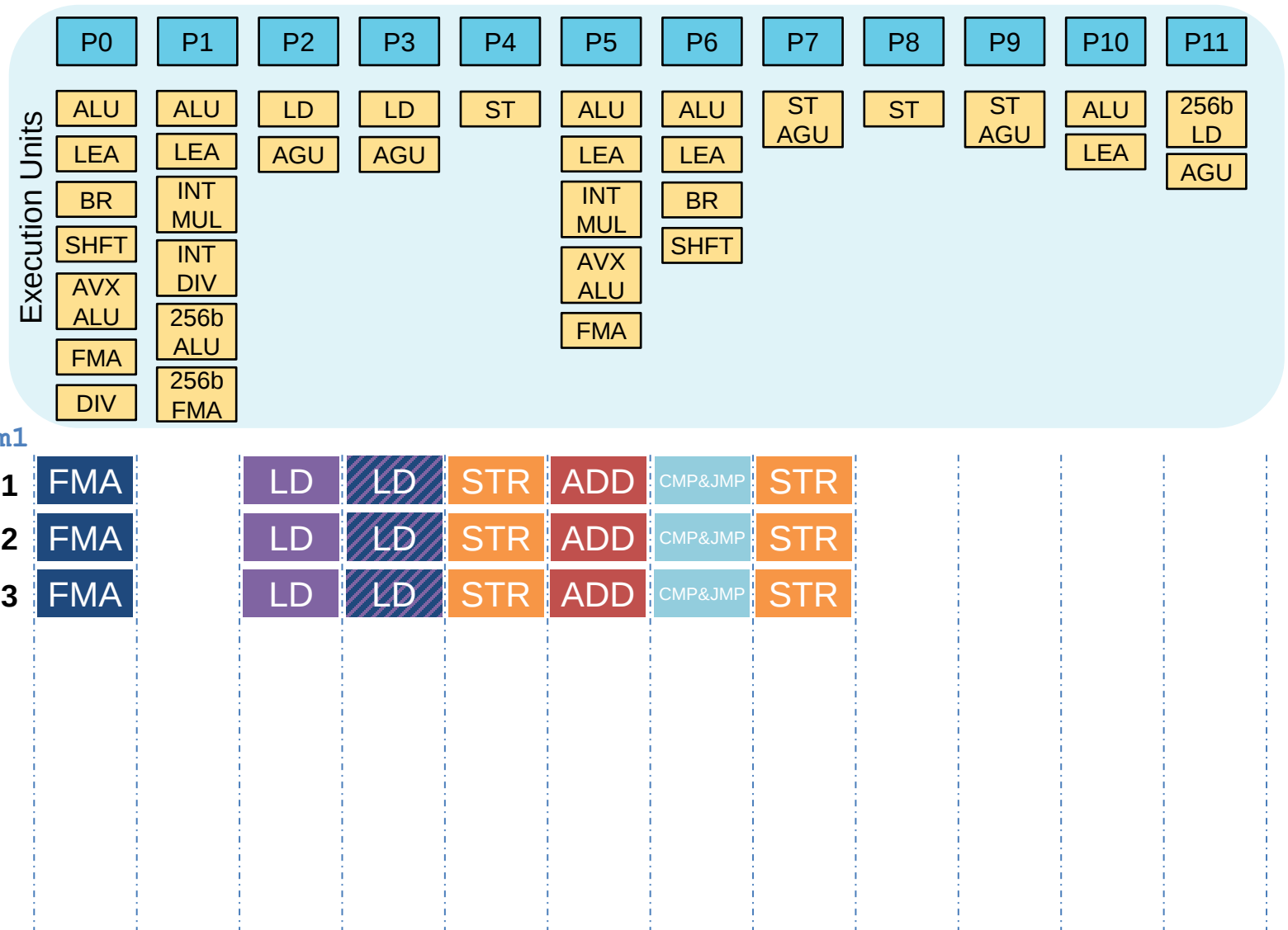
STREAM TRIAD

$a[i] = b[i] + s * c[i]$

..B2.42:

```
vmovups    (%r14,%rdx,8), %zmm1
vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
vmovupd    %zmm1, (%r12,%rdx,8)
addq       $8, %rdx
cmpq       %rsi, %rdx
jb         ..B2.42
```

It x+1
It x+2
It x+3



1 cy / 8 it

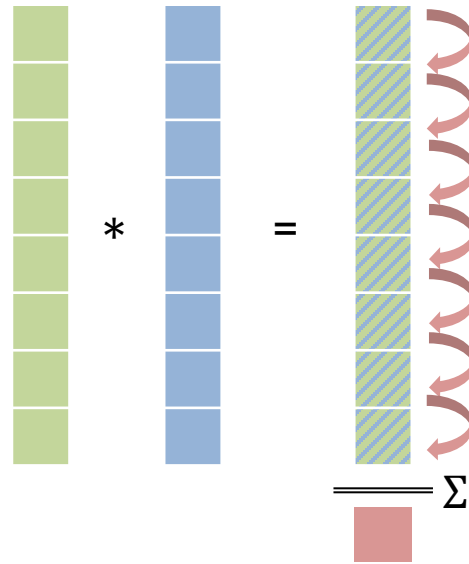
Hands-On #1: Dot product

→ <https://go-nhr.de/CLPE-ex1>

Hands-On: Benchmarking Dot Product

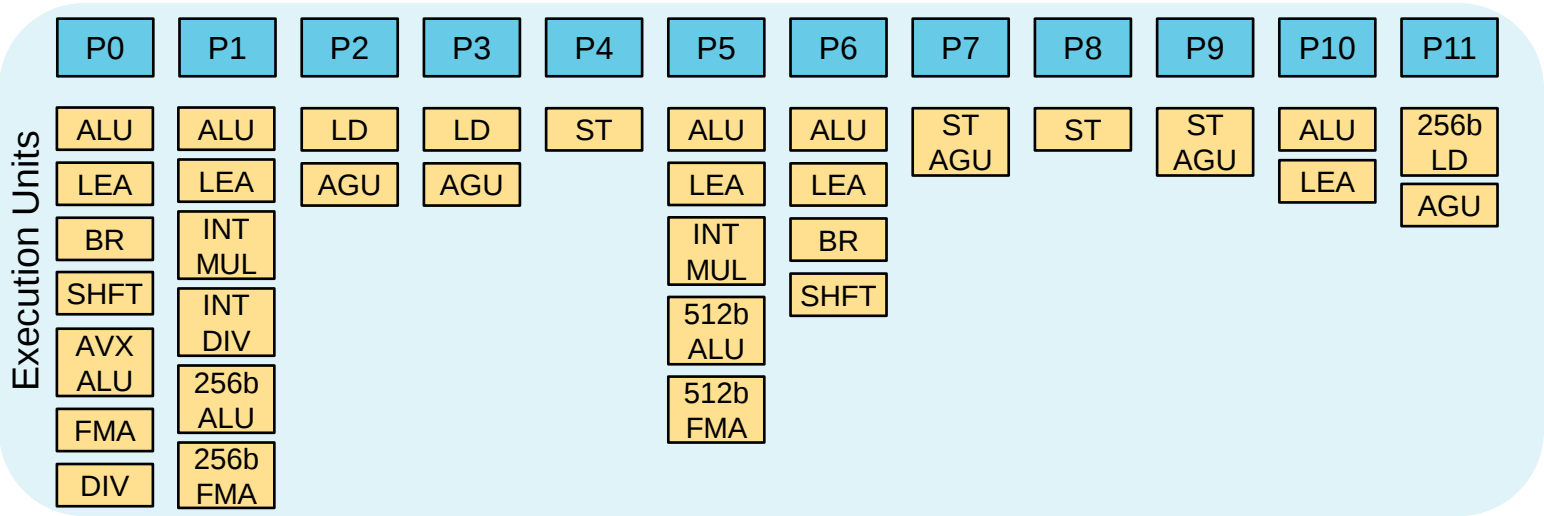
Dot product

`s = s + a[i] * b[i]`



→ Moodle, hands-on #1

Dot Product on SPR



0												
1												
2												
3												
4												
5												
6												
7												

Hands-On #2: Dot product (with Compiler Explorer)

→ <https://go-nhr.de/CLPE-ex2>

Dot Product on SPR – CE view

→ Moodle, hands-on #2

Add new compiler

Right click and “Reveal linked code” can help you find your region of interest

Set **executer** compiler and flags (separately from ASM compiler)

Set runtime parameters

Set **ASM** compiler compiler and flags

Add new **analysis** tool

Add new **executor**

Click to see your **compiler log** (warnings and errors)

Set **OSACA** runtime parameters

The screenshot displays the COMPILE EXPLORER application interface, which is used for configuring and running code analysis. The main window shows a C++ source file with a benchmark loop. A right-click context menu is open over the code, with the 'Reveal linked code' option highlighted. The bottom panel shows the 'Executer' configuration for 'x86-64 icpc 2021.6.0' with various flags. The 'Arguments' tab is selected, showing the command line. The 'Compiler output' tab shows the program's execution results, including size, cycles per iteration, and total walltime. The 'Analysis' tab shows a 'Combined Analysis Report' with a table of port pressure in cycles.

Compiler Output:

```
Program returned: 0
Program stdout
Size: 39.98 kB, 1706 elements
Cycles per high-level iteration: 0.161946
Total walltime: 1.983860, NITER: 8388608
```

Combined Analysis Report:

	0	- 0DV	1	- 1DV	2	3	4	5	6	7	8	9	10	11	CP	LCD
288																
289					0.50	0.50							0.500			
290					0.50	0.50							0.500			
291					0.32	0.33							0.850			
292					0.00	0.00							1.500			
293	0.50				0.50	0.50		0.50							9.0	
294	0.50				0.50	0.50		0.50								
295	0.50				0.50	0.50		0.50								
296	0.50				0.50	0.50		0.50								4.0
297	0.00		0.50					0.00	0.50				-0.01			
298	0.00		0.50					0.00	0.50				-0.01			

The Open-Source Architecture Code Analyzer (OSACA)

An introduction

OSACA

- Open Source Architecture Code Analyzer
- Static in-core code analysis

Assumptions

- Steady-state execution (no warm-up/cool-down)
- All data in L1
- Perfect out-of-order scheduling
- (currently) no front-end, i.e., no limit in instruction fetching, decoding, etc...
- Architecture specific model for each μ Arch

- Python module

```
$ pip install osaca
```

OSACA – Usage

```
osaca [-h] [-V] [--arch ARCH] [--fixed] [--lines LINES]
      [--ignore-unknown] [--lcd-timeout SECONDS]
      [--db-check] [--import MICROBENCH] [--insert-marker]
      [--export-graph GRAPHNAME] [--consider-flag-deps]
      [--out OUT] [--verbose]
      FILEPATH
```

Important flags:

--arch ARCH

Currently supported: **Intel** SNB – SPR, **AMD** ZEN1, ZEN2, ZEN3, ZEN4
Arm TX2, A72, N1, A64FX, TSV110, Grace

--lines L1,L2,L3-L4,L5:L6

Specify lines to analyze (if no markers are used)

--ignore-unknown

Assume 0cy TP/LAT for unknown instructions

Marking the region of interest

Comment marker

x86

```
# OSACA-BEGIN
.L22:
    vmovapd 0(%r13,%rax),%ymm0
    vfmadd213pd (%r14,%rax),%ymm1,%ymm0
    vmovapd %ymm0, (%r12,%rax)
    addq $32,%rax
    cmpq %rax,%r15
    jne .L22
# OSACA-END
```

arm

```
// OSACA-BEGIN
.L18:
    ldr q2, [x20, x0]
    ldr q1, [x21, x0]
    fmla v1.2d, v2.2d, v0.2d
    str q1, [x19, x0]
    add x0, x0, #16
    cmp x22, x0
    bne .L18
// OSACA-END
```

Insertion tool

```
$ osaca --arch ARCH --insert-marker

Blocks found in assembly file:
.L_A
...
.L_B
...
-----
Possible blocks to be marked:
.L_A
.L_B
Choose block to be marked [.L_B]: _
```

will be marked with **byte** markers, i.e.:

```
movl $111,%ebx; .byte 100,103,144; (x86)
```

...

```
movl $222,%ebx; .byte 100,103,144;
```

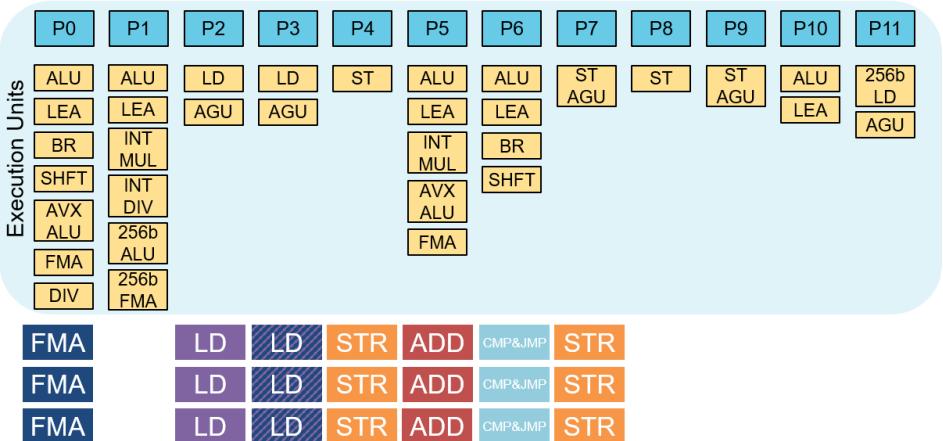
```
mov x1,#111; .byte 213,3,32,31 (aarch64)
```

...

```
mov x1,#222; .byte 213,3,32,31
```

Triad on SPR with OSACA

- Recap: Manual analysis resulted in 1 cy/8 it



```
$ osaca --arch SPR triad.s
Open Source Architecture Code Analyzer (OSACA) - 0.6.0
Architecture: SPR

* - Instruction micro-ops not bound to a port
X - No throughput/latency information for this instruction in data file
```

Port pressure in cycles														CP	LCD
0	1	2	3	4	5	6	7	8	9	10	11				
2															
3		0.50	0.50								0.50	5			
4	0.50		0.50	0.50	0.50						0.50	4			
5				1.00			1.00	1.00	1.00			0			
6	0.10	0.26			0.10	0.27				0.27		1			
7	0.00	0.34			0.00	0.33				0.33					
8															
0.60 0.60 1.00 1.00 1.00 0.60 0.60 1.00 1.00 1.00 0.60 1.00														9	1

```
..B2.42:
vmovups (%r14,%rdx,8), %zmm1
vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
vmovupd %zmm1, (%r12,%rdx,8)
addq $8, %rdx
cmpq %rsi, %rdx
jb ..B2.42
```

Loop-Carried Dependencies Analysis Report

6	1.0	addq	\$8, %rdx	[6]
---	-----	------	-----------	-----

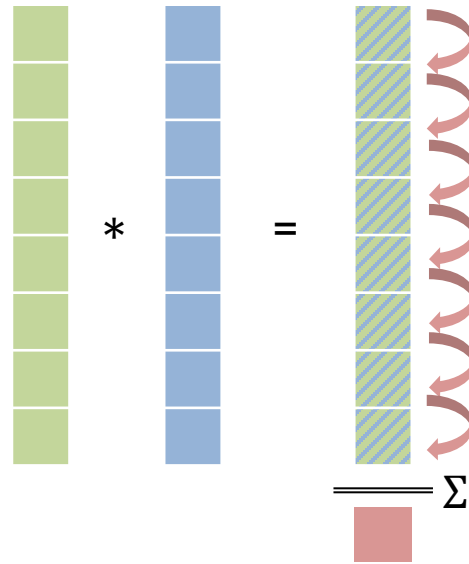
Hands-On #3: Dot Product with OSACA

→ <https://go-nhr.de/CLPE-ex3>

Hands-On: Benchmarking Dot Product (DP)

Dot Product

$s = s + a[i] * b[i]$



→ Moodle, hands-on #3

Hands-On #4: PI by integration

→ <https://go-nhr.de/CLPE-ex4>

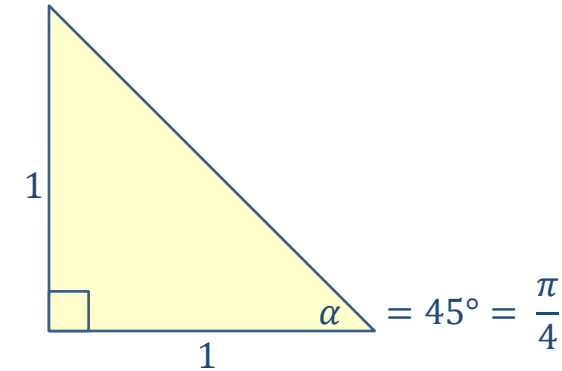
Hands-On: PI by integration

PI

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

```
double delta_x = 1./n;  
double sum      = 0.0;  
  
for (int i=0; i<n; i++)  
{  
    x      = (i + 0.5) * delta_x;  
    sum += (4.0 / (1.0 + x * x));  
}
```

→ Moodle, hands-on #4



$$\tan(\alpha) = \frac{1}{1} = 1 \Rightarrow \arctan(1) = \frac{\pi}{4}$$

$$\Rightarrow \pi = 4 \cdot \arctan(1)$$

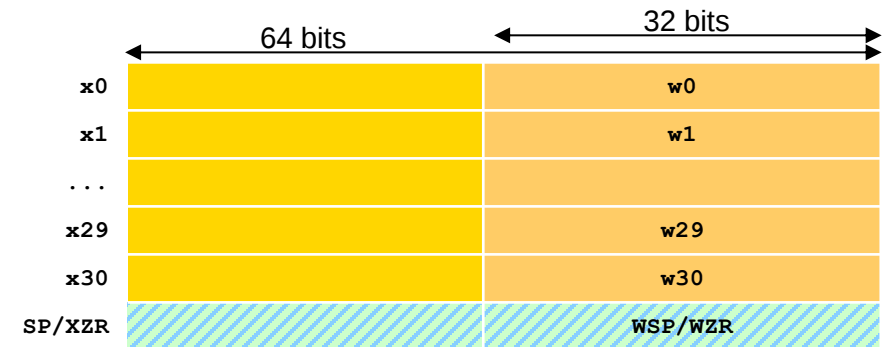
$$\frac{d}{dx} \arctan(x) = \frac{1}{1+x^2} \Rightarrow \pi = \int_0^1 \frac{4}{1+x^2} dx$$

Break

AArch64 Arm ISA

AArch64 ISA – differences to x86

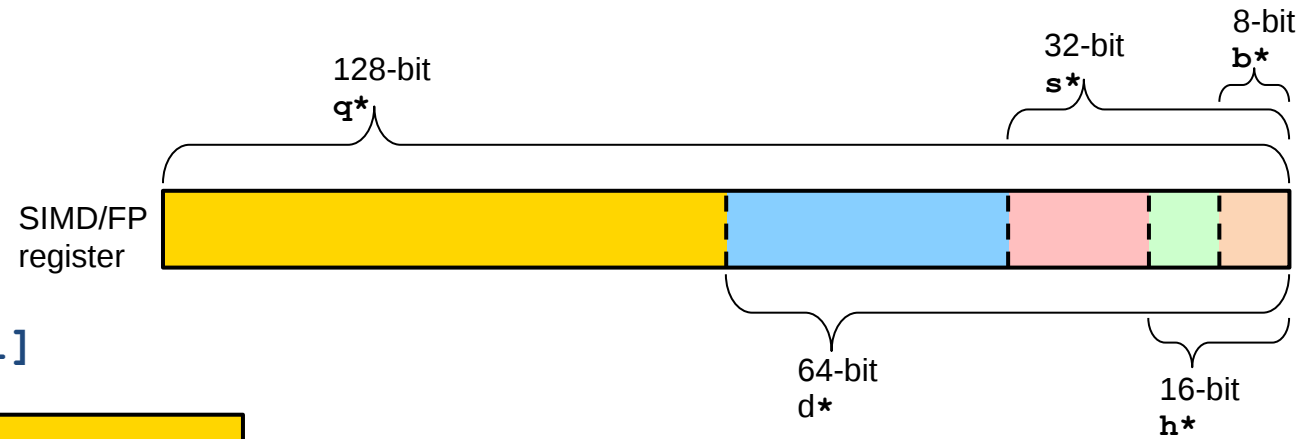
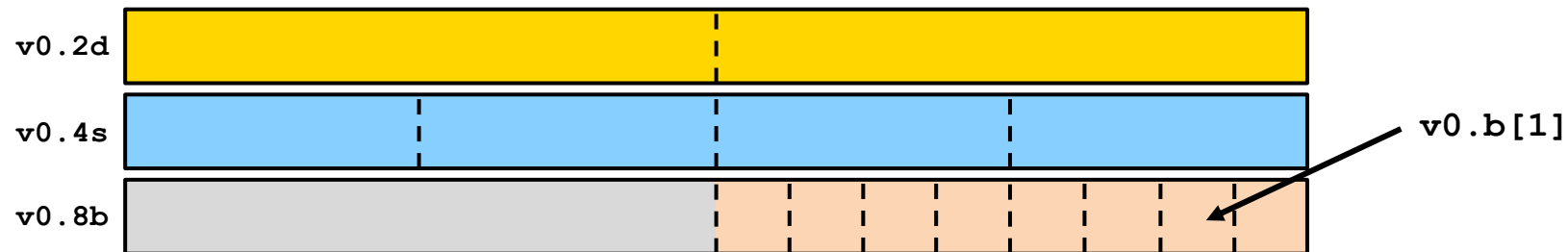
- Addressing mode: [BASE, INDEX, SCALE]
- Similar to Intel (left) syntax with STORE (**STR/STP**) as exception
 - `add x1, x1, 8` # $x1 \leftarrow x1 + 8$
 - `ldr x0, [x1]` # $x0 \leftarrow \text{mem at } x1$
 - `ldp x0, x1, [x2]` # $x0, x1 \leftarrow \text{mem at } x2$
 - `str x0, [x1]` # $\text{mem at } x1 \leftarrow x0$
 - `stp x0, x1, [x2]` # $\text{mem at } x2 \leftarrow x0, x1$
- 31 general purpose registers (64 bits):
 - `x0–x30` (aliases with 32-bit GPRs `w0–w30`)
 - 32nd register is stack pointer and zero register



AArch64 ISA – differences to x86

- 32 SIMD and FP registers (NEON, 128 bits)

- $v0-v31$
- can be optionally specified with **shapes** and **lanes** $vn.<LANES><SHAPE>$
- a single element can be indexed via brackets $[i]$

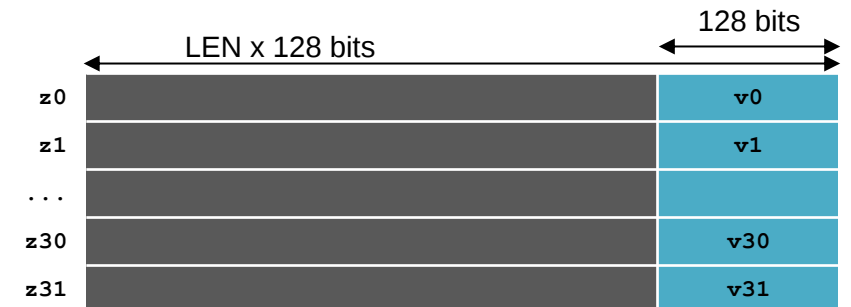


- 32 scalable vector registers (128–2048 bits):

- $z0-z31$, extending v registers, multiples of 128 bits
- size defined in OS

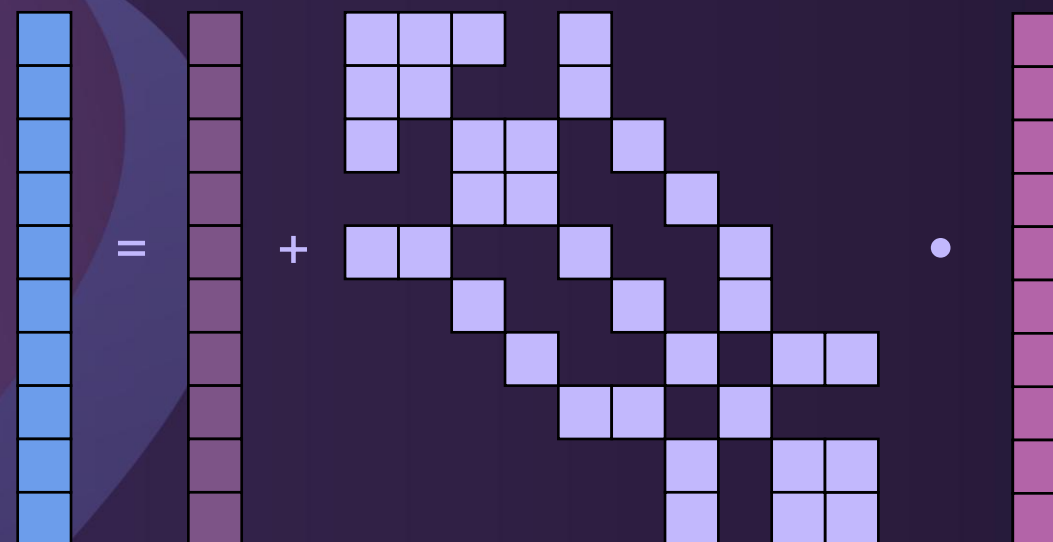
- 16 predicate registers (16–256 bits)

- $p0-p15$, multiples of 16 bits
- optional with predication operation $/z$, $/m$, $/x$ (zeroing, merging, don't care)



Case Study: SpMV on A64FX

Sparse Matrix-Vector Multiplication



Based on:

C. Alappat, N. Meyer, J. Laukemann, T. Gruber,

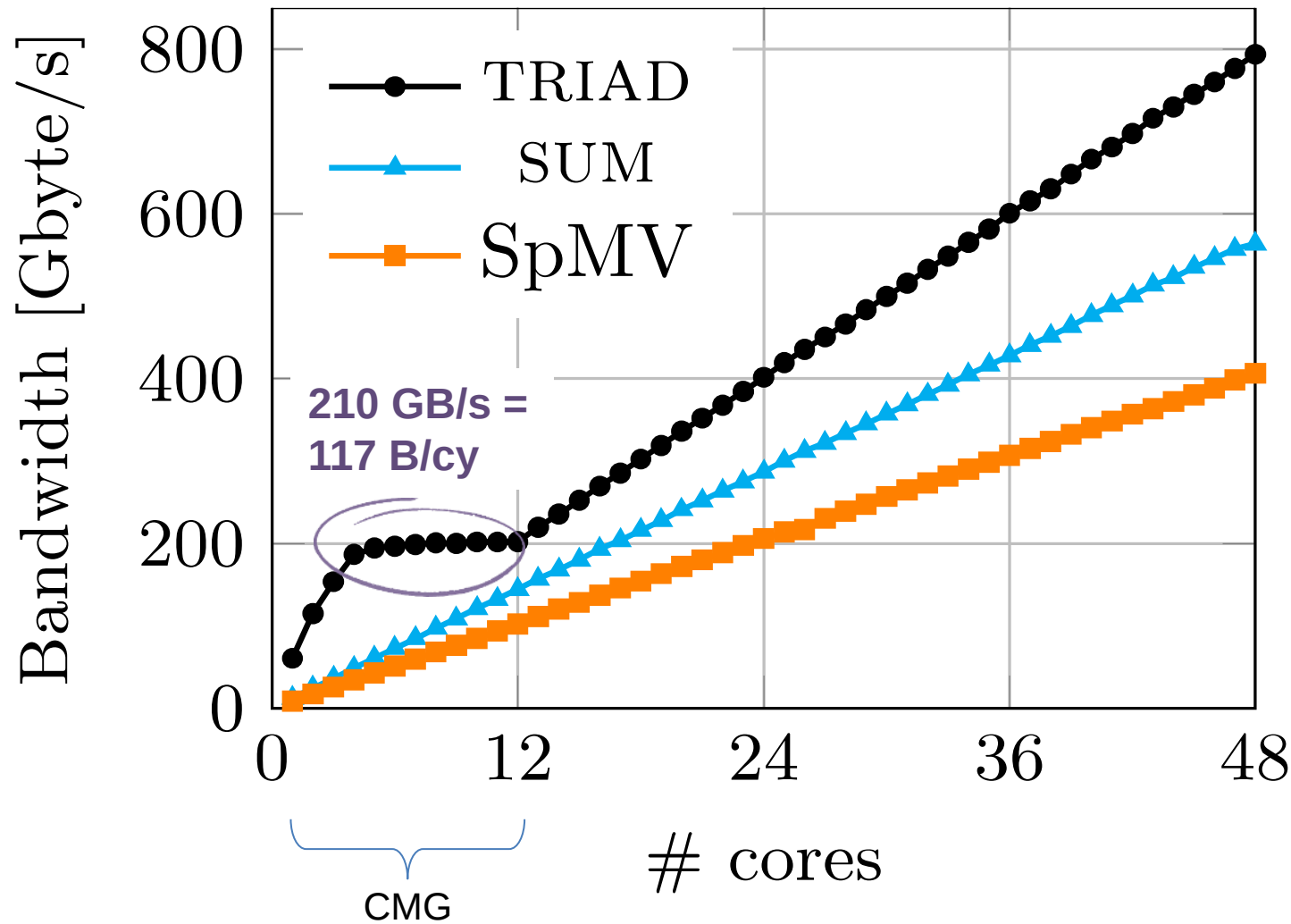
G. Hager, G. Wellein, and T. Wettig:

ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.

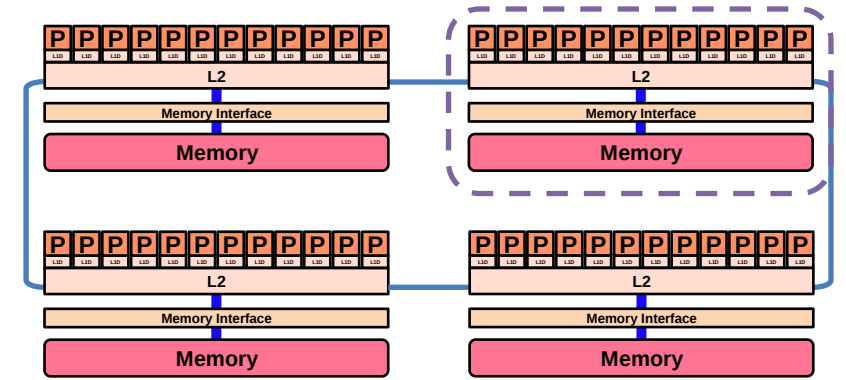
Concurrency and Computation: Practice and Experience, e6512 (2021).

DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

Motivation



Thread pinning : Compact

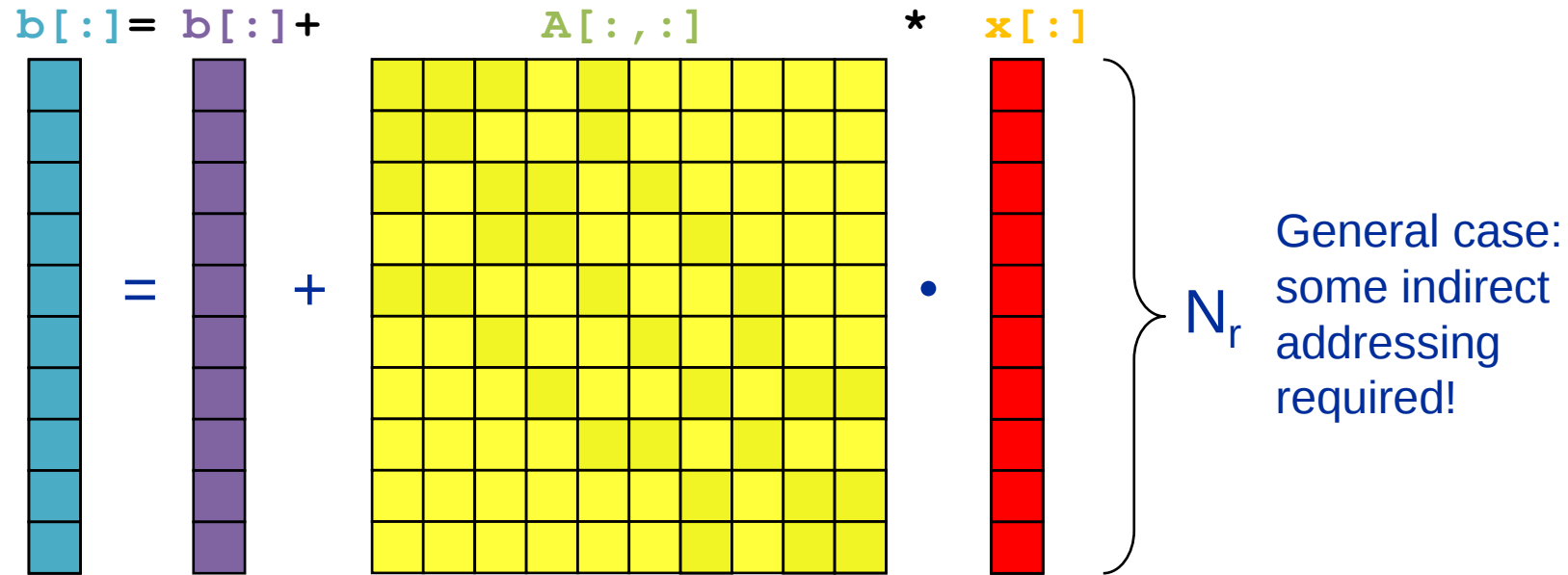


Clear memory bandwidth saturation for STREAM TRIAD ($a[i] = b[i] + s*c[i]$)

But why not for SUM ($s += a[i]$) and SpMV ($b[:] = A[:, :] * x[i[:]]$)?

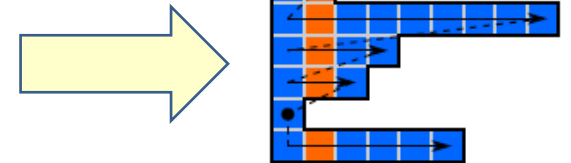
SpMV

Sparse Matrix-Vector Multiplication (SpMV) : $b = Ax$



In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
         $b[i] = b[i] + A[j] * x[col\_idx[j]]$ 
```



Assembly of the short inner-loop

<pre> .L6: ld1sw z0.d, p0/z, [x17, x20, lsl 2] ld1d z2.d, p0/z, [x18, x20, lsl 3] ld1d z3.d, p0/z, [x30, z0.d, lsl 3] add x20, x20, 8 fmla z1.d, p0/m, z3.d, z2.d whilelo p0.d, x20, x14 b.any .L6 </pre>	load col_idx[j] load A[j] gather from x[] increase loop counter j “the work” loop mechanics
<pre> faddv d4, p1, z1.d </pre>	reduction across SIMD register

In Compressed Row Storage (CRS) format

```

for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
    
```

Assembly of the short inner-loop

.L6:

```
ld1sw    z0.d, p0/z, [x17, x20, lsl 2]
ld1d     z2.d, p0/z, [x18, x20, lsl 3]
ld1d     z3.d, p0/z, [x30, z0.d, lsl 3]
add       x20, x20, 8
fmla     z1.d, p0/m, z3.d, z2.d
whilelo p0.d, x20, x14
b.any    .L6
faddv    d4, p1, z1.d
```

FMA: Update **z1.d**

Latency: 9 cycles

Loop length : 27
HPCG matrix

Horizontal add of
512-bit register

latency = 49 cycles

85 cy per inner loop traversal
→ 100 GB/s per CMG

→ No saturation



In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

```
$ osaca --arch a64fx spmv-inner-loop.s
[...]
```

Combined Analysis Report

Port pressure in cycles													
0	- 0DV	1	2	3	4	5	- 5D	6	- 6D	7	CP	LCD	
1													.L6:
2						0.50	0.50	0.50	0.50		8.0		ld1sw z0.d, p0/z, [x17,x20,ls1 2]
3						0.50	0.50	0.50	0.50				ld1d z2.d, p0/z, [x18,x20,ls1 3]
4 1.00				1.00		2.00	2.00	2.00	2.00		11.0		ld1d z3.d, p0/z, [x30,z0.d,ls1 3]
5 0.00			0.00	0.00	1.00								add x20, x20, 8
6 0.00			1.00								9.0	9.0	fmla z1.d, p0/m, z3.d, z2.d
7		1.00		1.00									whilelo p0.d, x20, x14
8										1.00			b.any .L6
1.00 1.00 1.00 2.00 1.00 3.00 3.00 3.00 3.00 1.00 28.0 9.0													

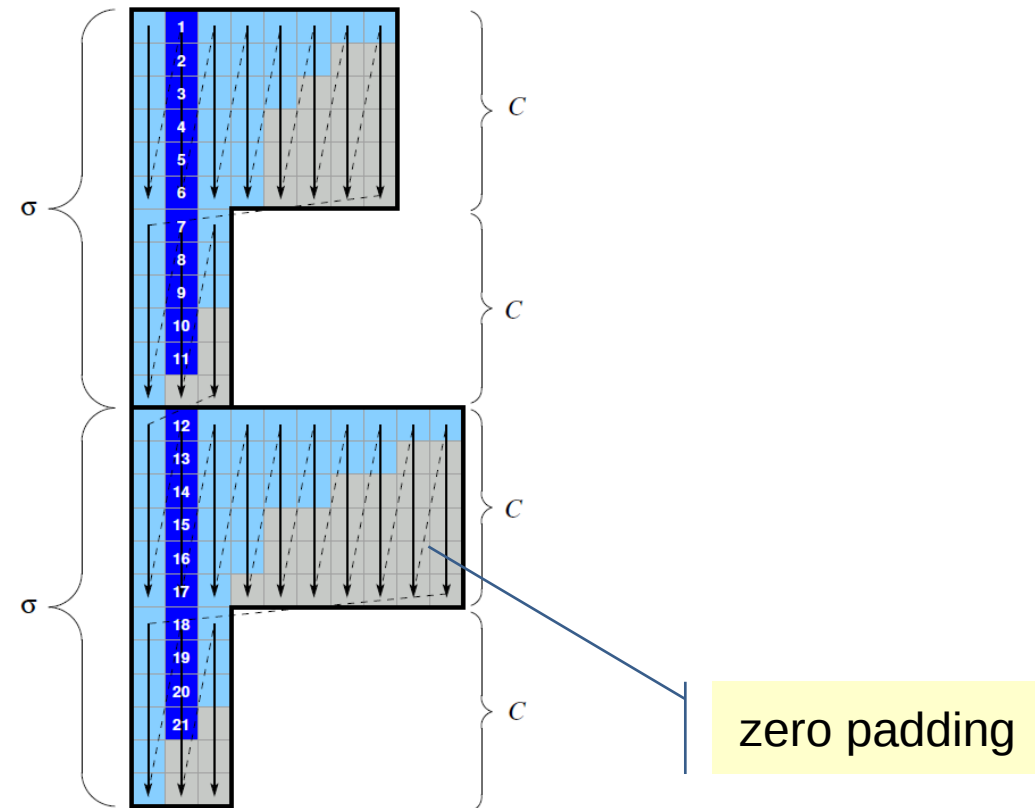
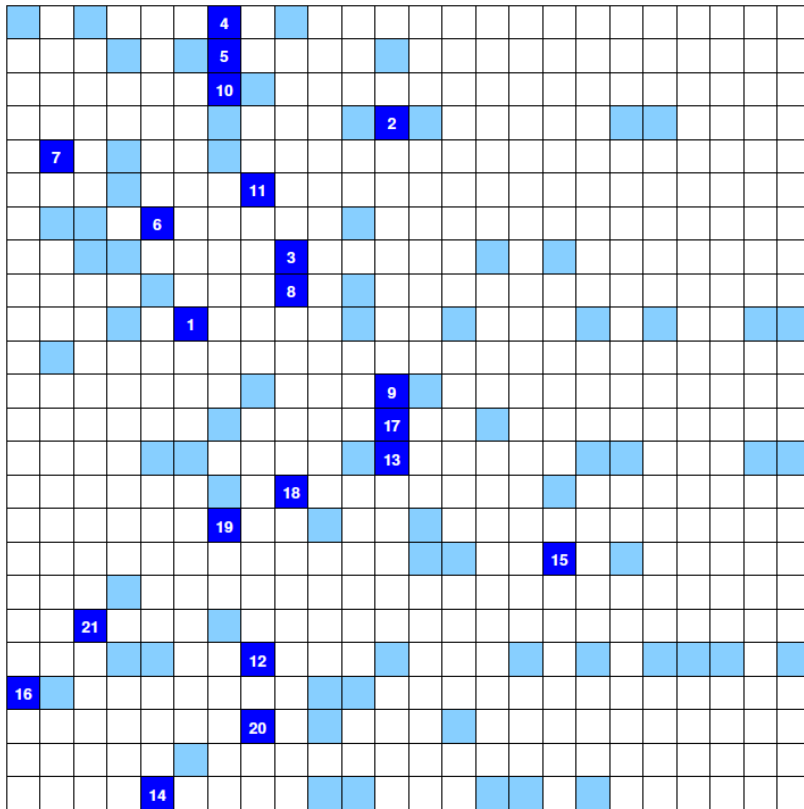
Loop-Carried Dependencies Analysis Report

6	9.0	fmla	z1.d, p0/m, z3.d, z2.d	[6]
5	1.0	add	x20, x20, 8	[5]

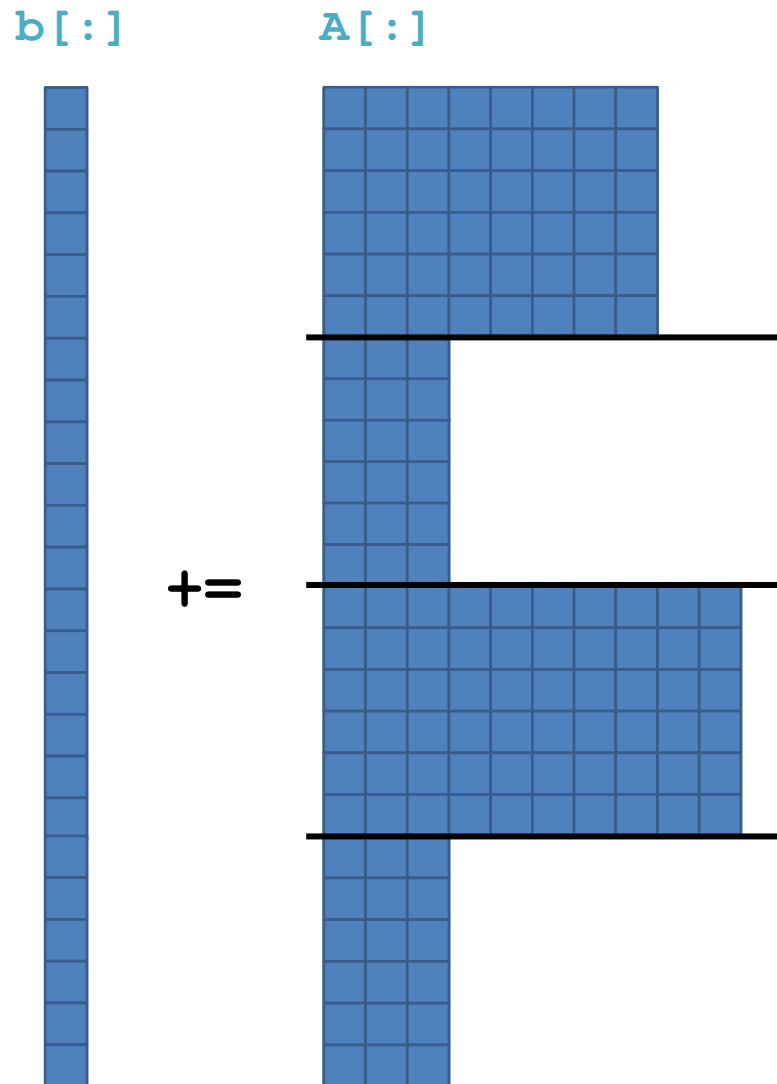
SELL-C- σ

Idea

- Sort rows according to length within **sorting scope σ**
- Store nonzeros column-major in zero-padded **blocks of height C**



SIMD-friendly execution of SpMV with SELL-C- σ

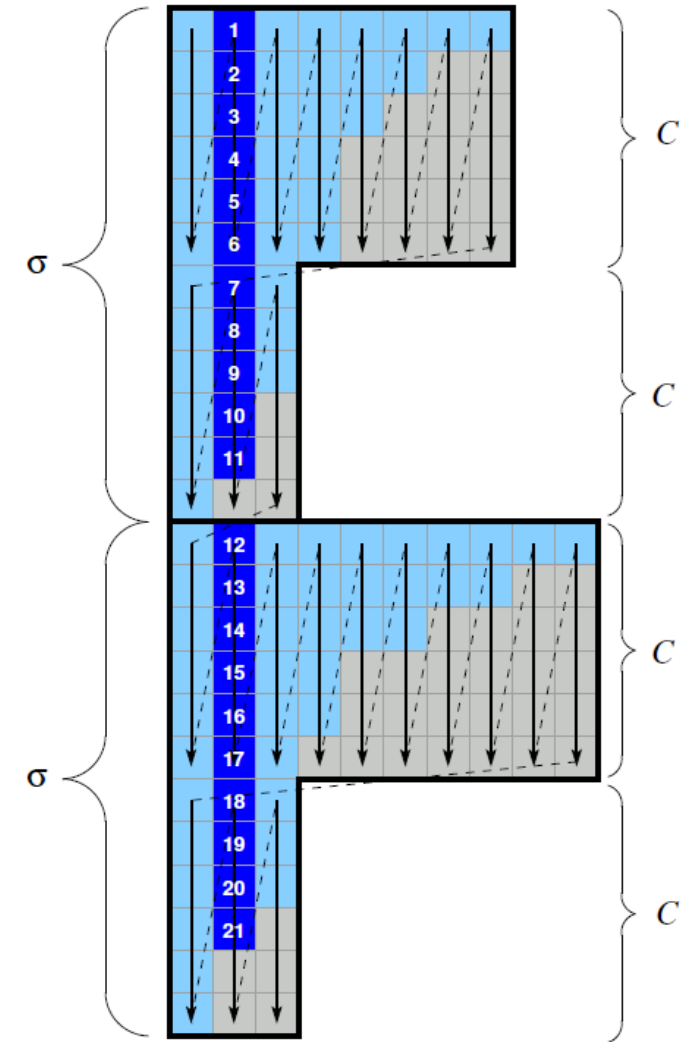


- Inner loop goes down one block column
- Column-major storage within block
→ consecutive access of matrix
- Enables SIMD FMA instructions for column traversal in block and LHS update
- No reductions across SIMD register slots
- Longer inner loop (in assembly) than CRS
- RHS access still indirect (gather)

How to choose the parameters?

- C
 - $n \times$ SIMD width to allow good utilization of SIMD units
 - $n > 1$ useful for hiding ADD pipeline latency
- σ
 - As small as possible, as large as necessary
 - Large σ reduces zero padding
 - Sorting alters RHS access pattern

M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). [DOI: 10.1137/130930352](https://doi.org/10.1137/130930352),



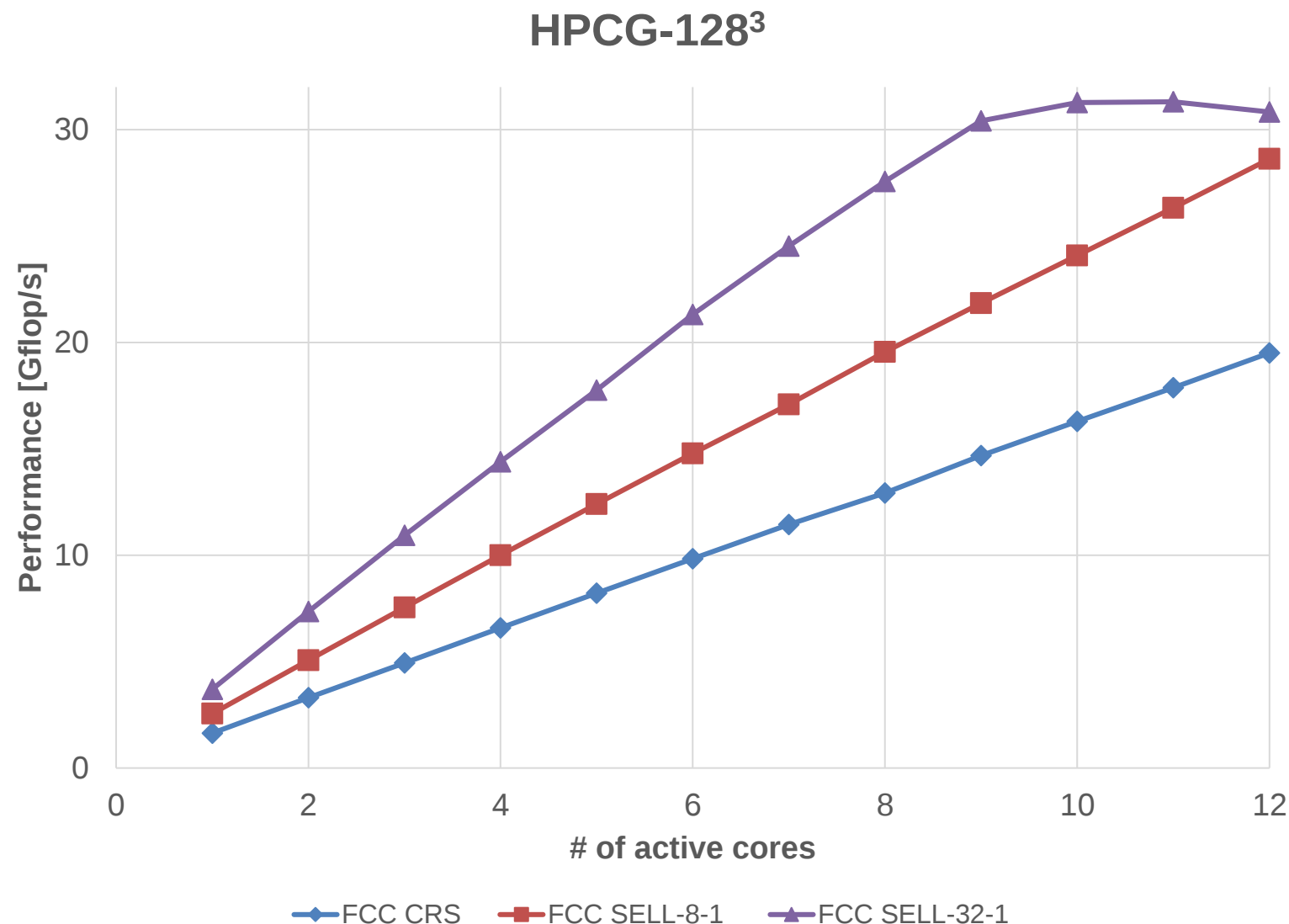
SELL-32- σ kernel OSACA analysis for A64FX

	0	- 0DV	1	2	3	4	5	- 5D	6	- 6D	7	CP	LCD	
92														.L4:
93							0.50	0.50	0.50	0.50				ldlsw z16.d, p0/z, [x11]
94							0.50	0.50	0.50	0.50				ldlsw z17.d, p0/z, [x11, #1, mul vl]
95							0.50	0.50	0.50	0.50				ldlsw z20.d, p0/z, [x11, #2, mul vl]
96							0.50	0.50	0.50	0.50		8.0		ldlsw z21.d, p0/z, [x11, #3, mul vl]
97	0.00		0.00	0.00	1.00									add x10, x10, 32
98	0.00		0.00	0.00	1.00									add x11, x11, 128
99	0.00		0.00	0.00	1.00									add x12, x12, 256
100							0.50	0.50	0.50	0.50				ldld z19.d, p0/z, [x12, #-4, mul vl]
101							0.50	0.50	0.50	0.50				ldld z18.d, p0/z, [x12, #-3, mul vl]
102							0.50	0.50	0.50	0.50				ldld z25.d, p0/z, [x12, #-2, mul vl]
103							0.50	0.50	0.50	0.50				ldld z27.d, p0/z, [x12, #-1, mul vl]
104	1.00			1.00			2.00	2.00	2.00	2.00				ldld z22.d, p0/z, [x3, z16.d, lsl 3]
105	1.00			1.00			2.00	2.00	2.00	2.00				ldld z23.d, p0/z, [x3, z17.d, lsl 3]
106	1.00			1.00			2.00	2.00	2.00	2.00				ldld z24.d, p0/z, [x3, z20.d, lsl 3]
107	1.00			1.00			2.00	2.00	2.00	2.00		11.0		ldld z26.d, p0/z, [x3, z21.d, lsl 3]
108		1.00		1.00										whilelo p1.d, x10, x9
109	0.00		1.00											fmla z4.d, p0/m, z19.d, z22.d
110	0.00		1.00											fmla z5.d, p0/m, z18.d, z23.d
111	0.00		1.00											fmla z6.d, p0/m, z25.d, z24.d
112	0.00		1.00									9.0		fmla z7.d, p0/m, z27.d, z26.d
113	0.00		0.00	0.00	1.00									mov p0.b, p1.b
114											1.00			b.any .L4
	4.00		1.00	4.00	5.00	4.00	12.0	12.0	12.0	12.0	1.00	28.0	9.0	

Shift of bottleneck

SpMV performance with SELL-C- σ (1 CMG)

- SELL-C- σ separates SIMD from sum reduction
- $C > 8$ allows for reduction of `fm1a` latency impact

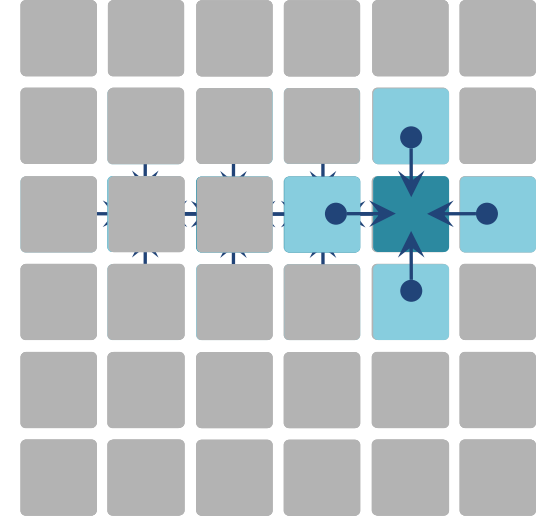


Hands-On #5: 2D Gauss-Seidel analysis

→ <https://go-nhr.de/CLPE-ex5>

Hands-On: Gauss-Seidel Method

- Limited by loop-carried dependency
- Create code with `-Ofast`, `-funroll-loops`
- Analyze for SPR

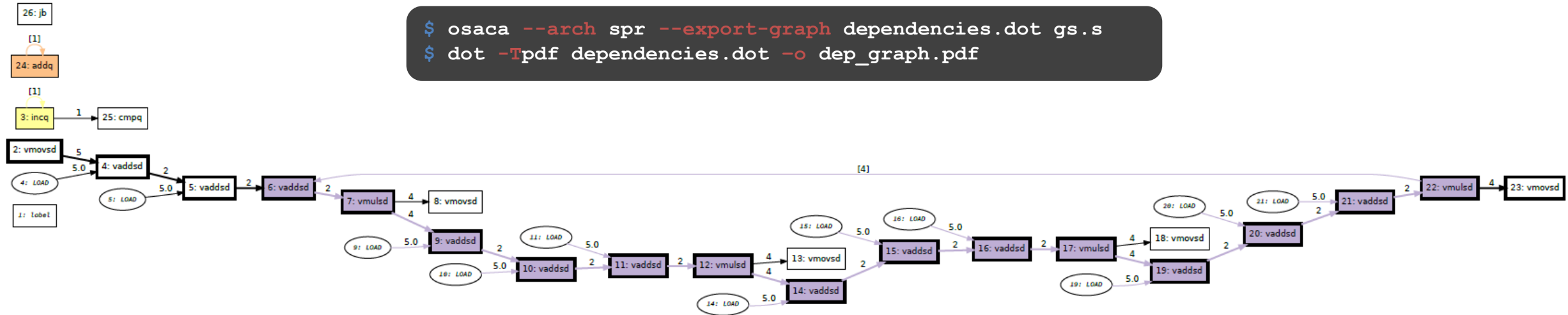


```
for(int it=0; it<NITER; ++it) {  
    for (int i=1; i<NI-1; ++i) {  
        for (int k=1; k<NK-1; ++k) {  
            phi[i][k] = 0.25 * (  
                phi[i][k-1] + phi[i+1][k] +  
                phi[i][k+1] + phi[i-1][k]  
            );  
        }  
    }  
}
```

→ Moodle, hands-on #5

Hands-On: Gauss-Seidel Method – standard version

```
$ osaca --arch spr --export-graph dependencies.dot gs.s
$ dot -Tpdf dependencies.dot -o dep_graph.pdf
```



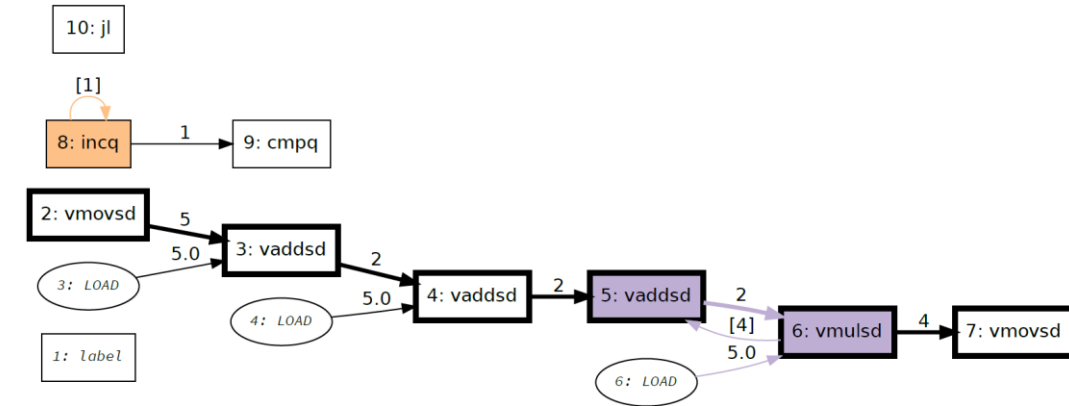
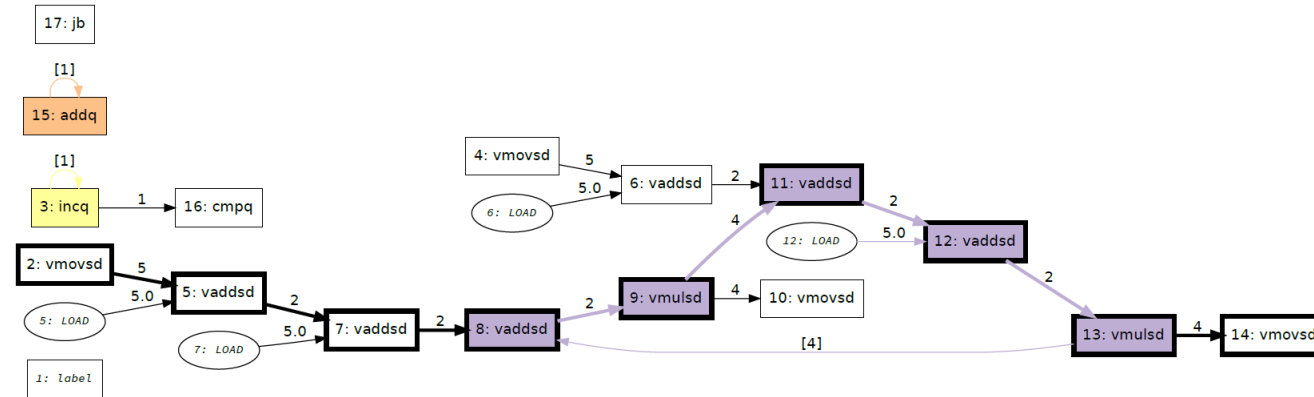
	LCD		
2		vmovsd	(%rsi,%r9), %xmm2
3		incq	%rdx
4		vaddsd	8(%rsi,%r10), %xmm2, %xmm3
5		vaddsd	16(%rsi,%r9), %xmm3, %xmm4
6	2.0	vaddsd	%xmm1, %xmm4, %xmm1
7	4.0	vmulsd	%xmm1, %xmm0, %xmm5
8		vmovsd	%xmm5, 8(%rsi,%r9)
9	2.0	vaddsd	(%rsi,%r10), %xmm5, %xmm6
10	2.0	vaddsd	8(%rsi,%r11), %xmm6, %xmm7
11	2.0	vaddsd	16(%rsi,%r10), %xmm7, %xmm8
12	4.0	vmulsd	%xmm8, %xmm0, %xmm9
13		vmovsd	%xmm9, 8(%rsi,%r10)
14	2.0	vaddsd	(%rsi,%r11), %xmm9, %xmm10
15	2.0	vaddsd	8(%rsi,%r8), %xmm10, %xmm11
16	2.0	vaddsd	16(%rsi,%r11), %xmm11, %xmm12
17	4.0	vmulsd	%xmm12, %xmm0, %xmm13
18		vmovsd	%xmm13, 8(%rsi,%r11)
19	2.0	vaddsd	(%rsi,%r8), %xmm13, %xmm14
20	2.0	vaddsd	8(%rsi,%r14), %xmm14, %xmm15
21	2.0	vaddsd	16(%rsi,%r8), %xmm15, %xmm16
22	4.0	vmulsd	%xmm16, %xmm0, %xmm1
23		vmovsd	%xmm1, 8(%rsi,%r8)
24		addq	%r13, %rsi
25	36.0	cmpq	%r12, %rdx

dep chain of
36 cy out of 45 of CP
→ 20% can overlap

Hands-On: Gauss-Seidel Method – opt. version (SPR)

-Ofast / -O3

-O1



	LCD	
1		..B1.72:
2		vmovsd 8(%r10,%r11), %xmm2
3		incq %rdx
4		vmovsd 16(%r10,%r11), %xmm5
5		vaddsd 16(%r10,%rsi), %xmm2, %xmm3
6		vaddsd 24(%r10,%rsi), %xmm5, %xmm7
7		vaddsd 8(%r10,%r13), %xmm3, %xmm4
8	2.0	vaddsd %xmm1, %xmm4, %xmm1
9	4.0	vmulsd %xmm0, %xmm1, %xmm6
10		vmovsd %xmm6, 8(%r10,%rsi)
11	2.0	vaddsd %xmm7, %xmm6, %xmm8
12	2.0	vaddsd 16(%r10,%r13), %xmm8, %xmm9
13	4.0	vmulsd %xmm0, %xmm9, %xmm1
14		vmovsd %xmm1, 16(%r10,%rsi)
15		addq \$16, %r10
16		cmpq %r15, %rdx
17		* jb ..B1.72

dep chain of
14 cy / 23 cy CP → 39% overlap

	LCD	
1		..B1.34:
2		vmovsd (%rsi,%rdi,8), %xmm0
3		vaddsd 8(%rcx,%rdi,8), %xmm0, %xmm1
4		vaddsd (%rax,%rdi,8), %xmm1, %xmm2
5	2.0	vaddsd %xmm3, %xmm2, %xmm3
6	4.0	vmulsd .L 2il0floatpacket.0(%rip), %xmm3, %xmm3
7		vmovsd %xmm3, (%rcx,%rdi,8)
8		incq %rdi
9		cmpq %r13, %rdi
10		* jl ..B1.34

dep chain of
6 cy / 15 cy CP → 60% overlap

Summary & Caveats

- A code analyzer helps you to predict the **in-core** runtime of a basic block
 - Might be sufficient, but often a full analysis requires a **memory model** as well!
- An analysis of **(loop-carried-)dependencies** can help you find performance limitations!
- Analysis is done on **compiler-generated** code which always holds a factor of uncertainty
- There might be additional things slowing you, e.g.:
 - Cache trashing
 - Loads across cache lines
 - Front end limitations
 - Bank conflicts
 - ...

There is not just THE one code analyzer

- **OSACA:** <https://github.com/RRZE-HPC/OSACA>
- **uiCA:** <https://www.uops.info/uiCA.html>
- **LLVM-MCA:** <https://llvm.org/docs/CommandGuide/llvm-mca.html>
- **IACA (EoL):**
<https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html>

THANK YOU! QUESTIONS?



Friedrich-Alexander-Universität
Erlangen-Nürnberg



Erlangen National High Performance
Computing Center



OSACA:

<https://github.com/RRZE-HPC/osaca>

```
pip: $ pip install -u osaca
```

Compiler Explorer: <https://godbolt.org>

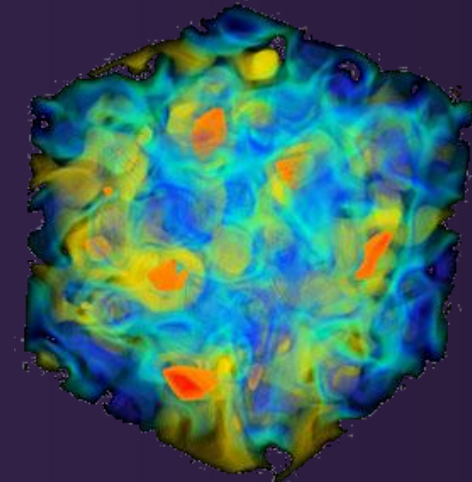


Appendix

Additional slides for self-studies

Case Study: Domain Wall (DW) Kernel

from Quantum Chromodynamics (QCD)



© Brookhaven National Lab

Based on:

C. Alappat, N. Meyer, J. Laukemann, T. Gruber,
G. Hager, G. Wellein, and T. Wettig:

ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.
Concurrency and Computation: Practice and Experience, e6512 (2021).

DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

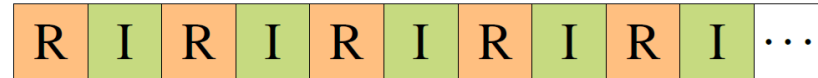
DW stencil kernel (simplified)

```
#define x_p 1 // x-plus direction
#define x_m 2 // x-minus direction
#define y_p 3 // y-plus direction
...
#pragma omp parallel for schedule(static)
for{t,z,y,x} = 1:{Lt-2,Lz-2,Ly-2,Lx-2} //collapsed loop over 4d space-time
{
    for(int s=0; s<Ls; ++s) //loop over fifth dimension
    {
        O[t][z][y][x][s] = R(x_p) · U[x_p][t][z][y][x] · P(x_p) · I[t][z][y][x+1][s] +
                           R(x_m) · U[x_m][t][z][y][x] · P(x_m) · I[t][z][y][x-1][s] +
                           R(y_p) · U[y_p][t][z][y][x] · P(y_p) · I[t][z][y+1][x][s] +
                           R(y_m) · U[y_m][t][z][y][x] · P(y_m) · I[t][z][y-1][x][s] +
                           R(z_p) · U[z_p][t][z][y][x] · P(z_p) · I[t][z+1][y][x][s] +
                           R(z_m) · U[z_m][t][z][y][x] · P(z_m) · I[t][z-1][y][x][s] +
                           R(t_p) · U[t_p][t][z][y][x] · P(t_p) · I[t+1][z][y][x][s] +
                           R(t_m) · U[t_m][t][z][y][x] · P(t_m) · I[t-1][z][y][x][s];
    }
}
```

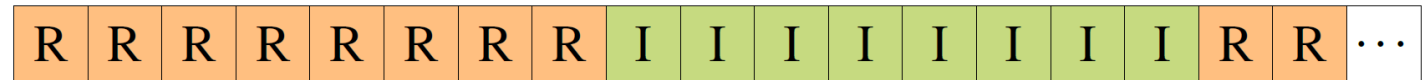
- “Grid” lattice QCD framework
- Uses SVE intrinsics
- Data type: double complex

Complex numbers data layout choice

AoS (standard)



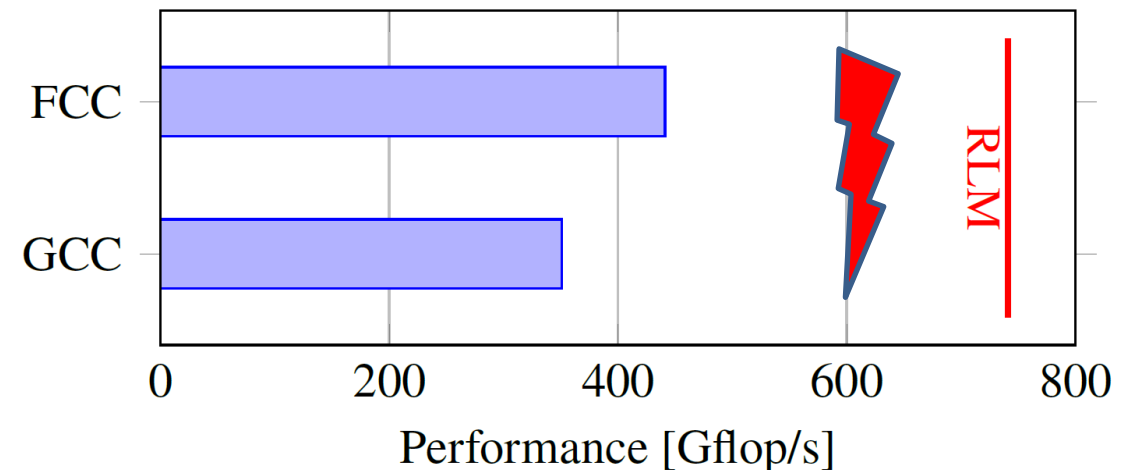
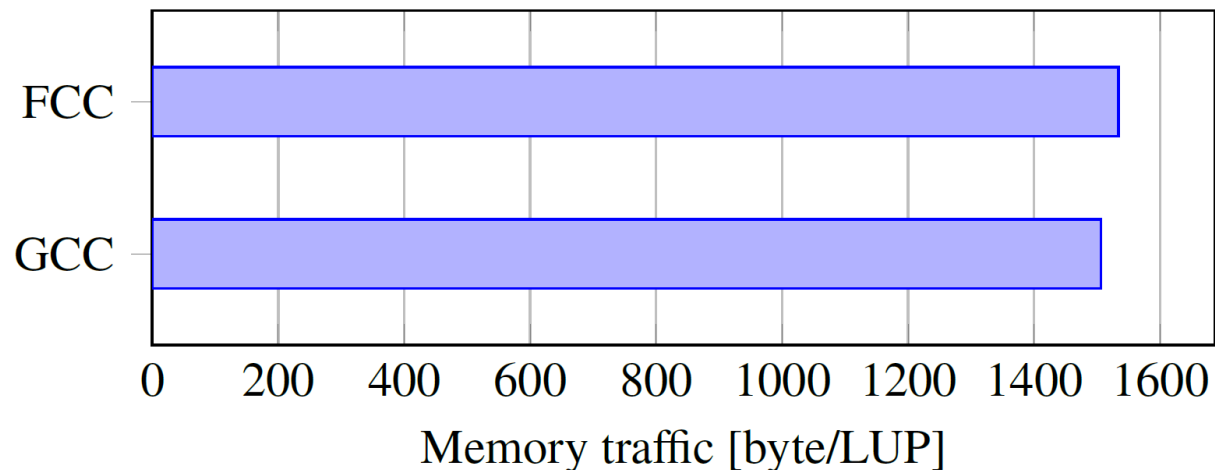
AoSoA



vector length

Observed performance

- Starting point: AoS layout, ACLE intrinsics, GCC/FCC
 - 1320 flops/LUP (theoretical)
 - Measured code balance: 1500 byte/LUP
 - A64FX (FX1000): $B_m = 0.25 \frac{\text{byte}}{\text{flop}} \rightarrow$ expect memory bound
- $B_c \approx 1.14 \frac{\text{byte}}{\text{flop}}$



In-core analysis (complex-AoS)

\$ osaca --
[...]

Combined A

| 0 -

560 |

561 | 0.00

562 |

563 | 0.00

564 |

565 |

5

1

1

1369 |

1370 |

680

500

30

30

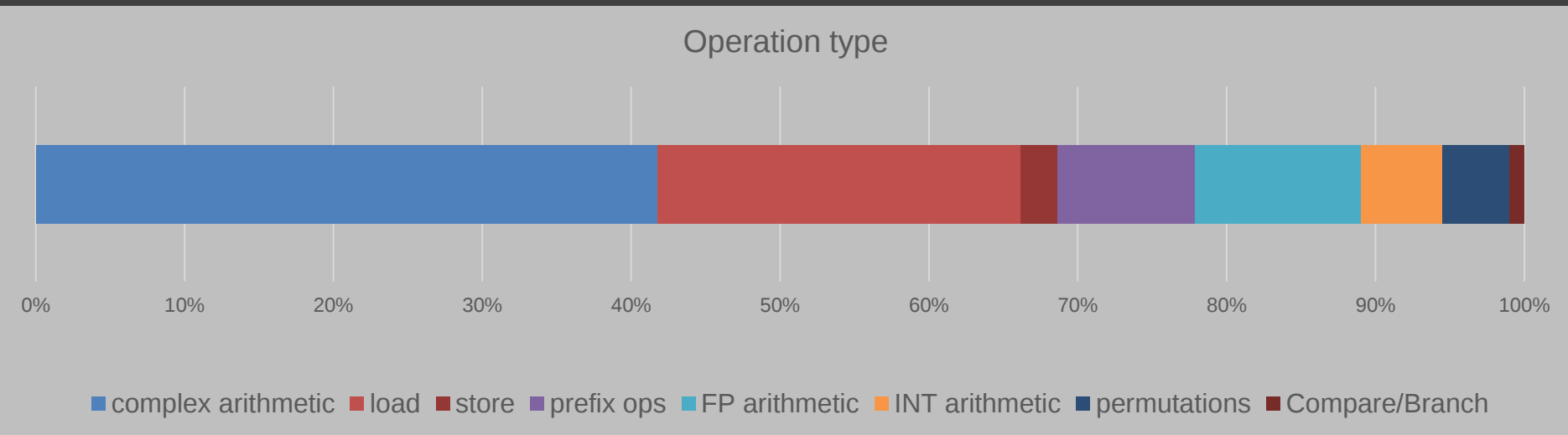
118.5

98.5

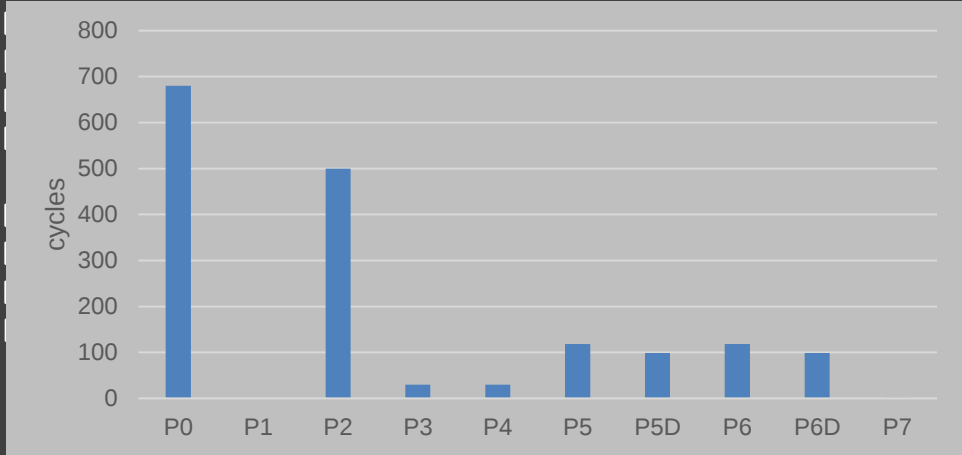
118.5

98.5

1.0



FCMLA	Zd, Pg, Zn, Zm, c	2cy on P0, 1cy on P2
FCADD	Zd, Pg, Zn, Zm, c	1cy on P0, 1cy on P2

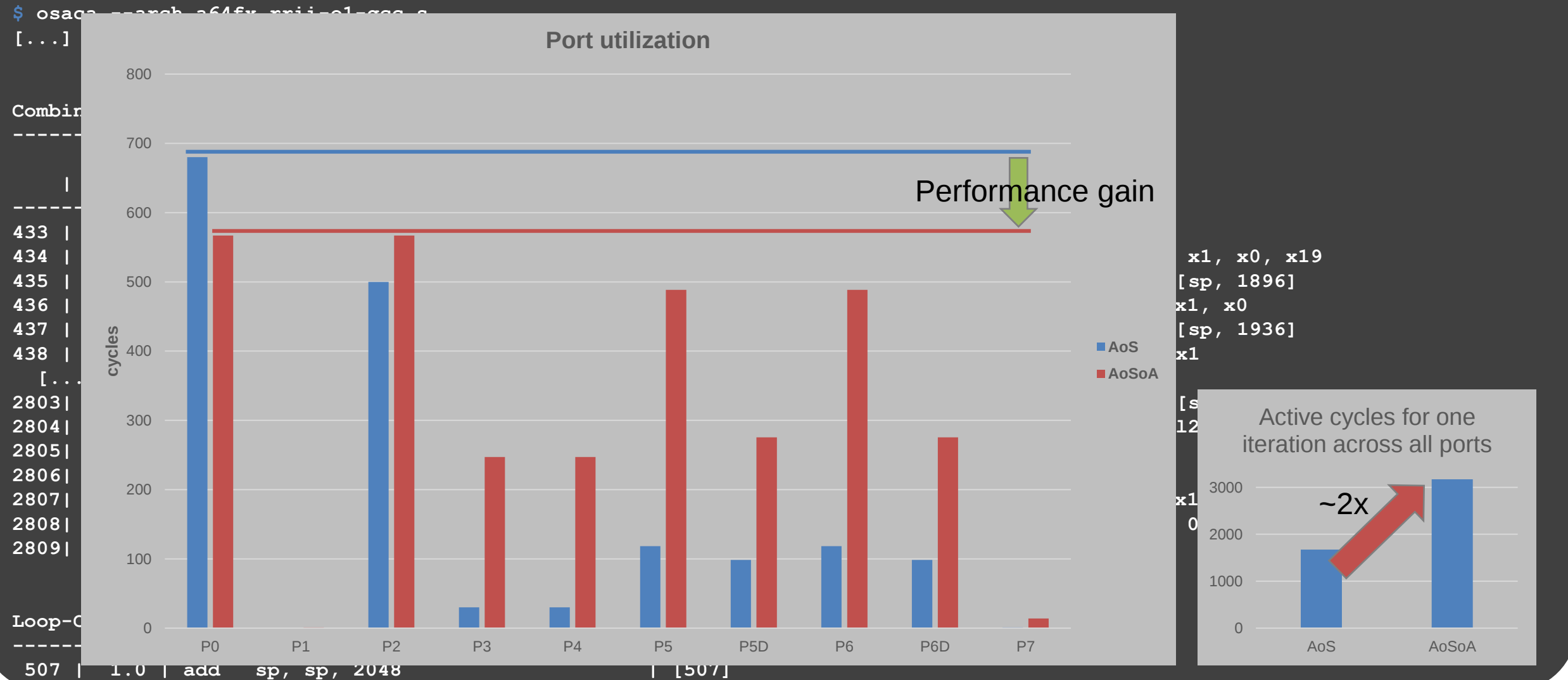


Loop-Carried Dependencies Analysis Report

1360 | 1.0 | add w13, w13, 1 | [1360]

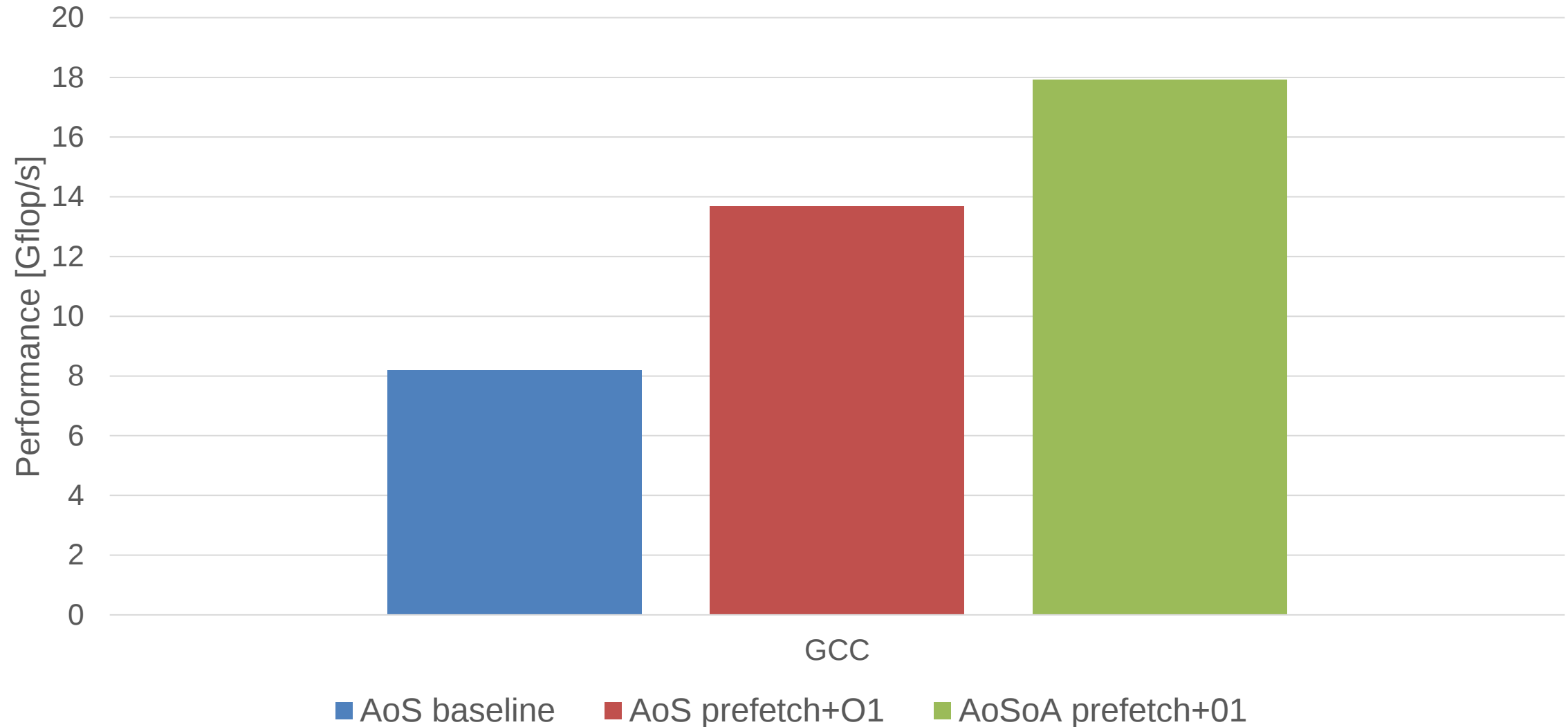
FMLA Zd, Pg, Zn, Zm 1cy on P0 OR P2
FADD Zd, Pg, Zn, Zm 1cy on P0 OR P2

In-core analysis (complex-AoSoA)



DW kernel optimizations

DW kernel



Summary of optimizations for DW

- **AoSoA** (RRII) data layout
 - Prevents use of complex arithmetic instructions **fcmla/fcadd**
 - Removes imbalance between FLA and FLB ports in the core
 - Some register spills occur, but still better than AoS (RIRI)
 - More instructions but better performance
- **Software prefetching** decreases L2 data volume
- **-O1** makes compiler obey the ordering hints in the computational kernel (more efficient OoO execution)

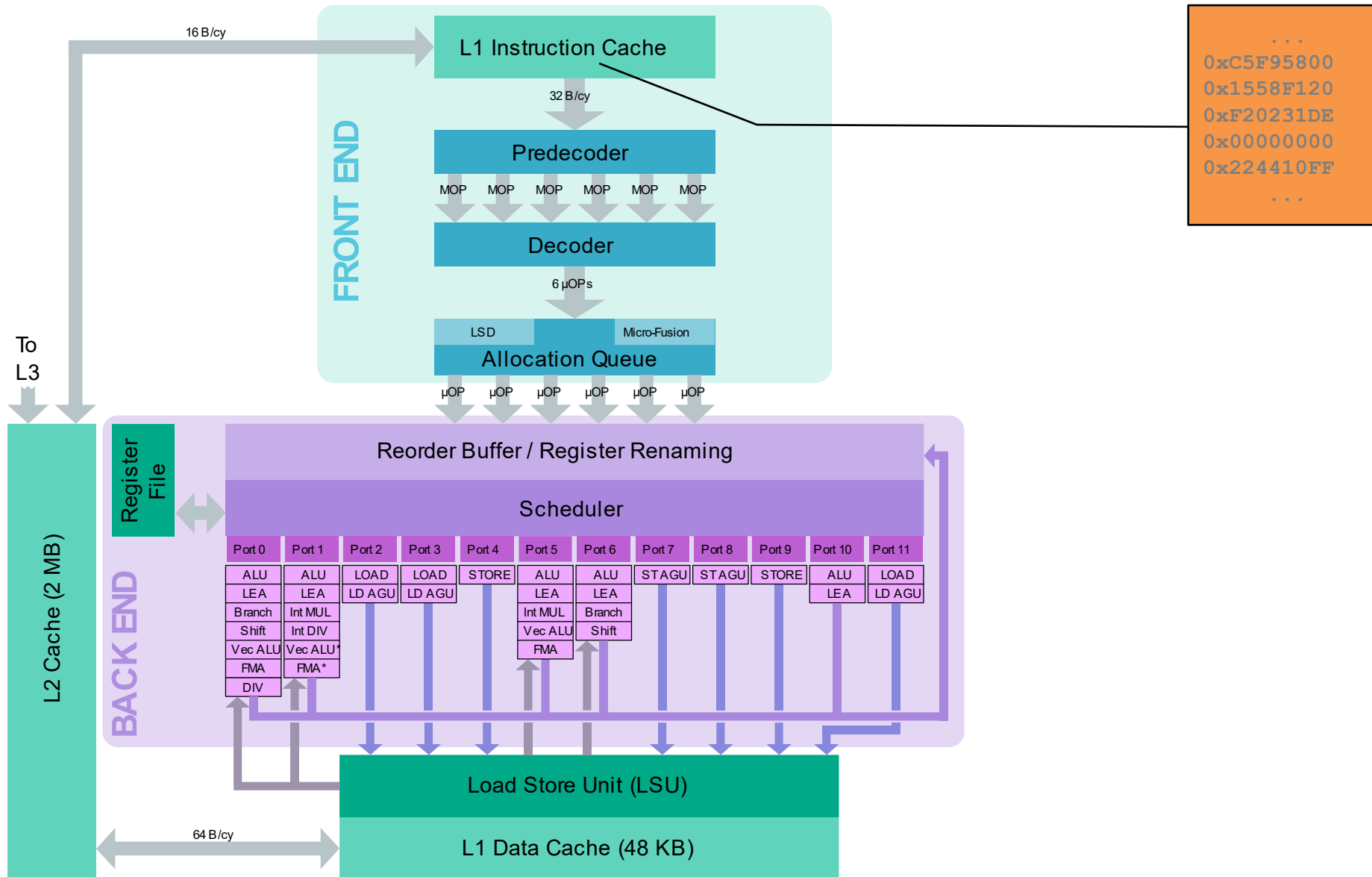
Appendix

Additional slides for self-studies

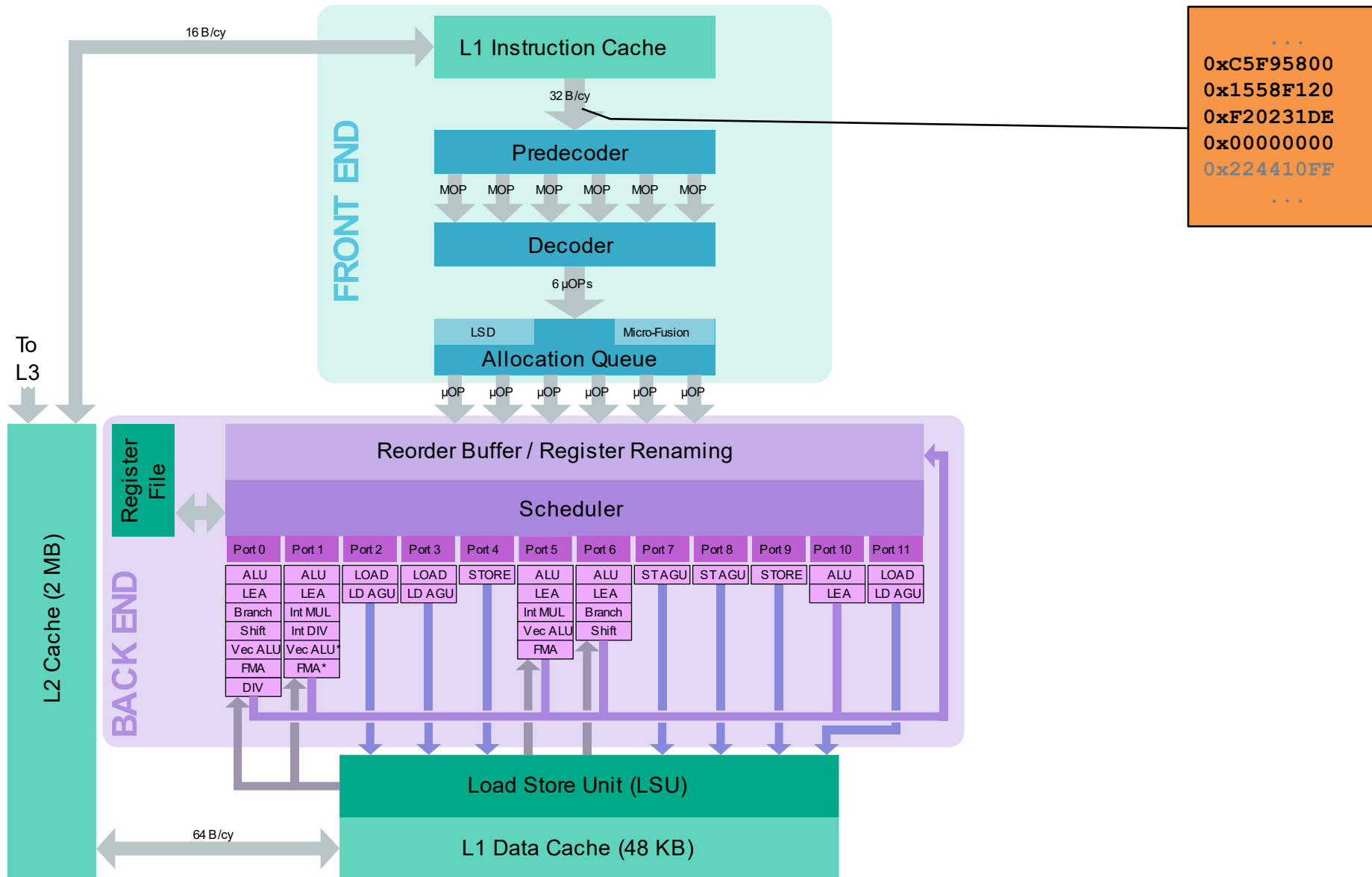
Basic x86 out-of-order core architecture

On the example of a Sapphire Rapids chip

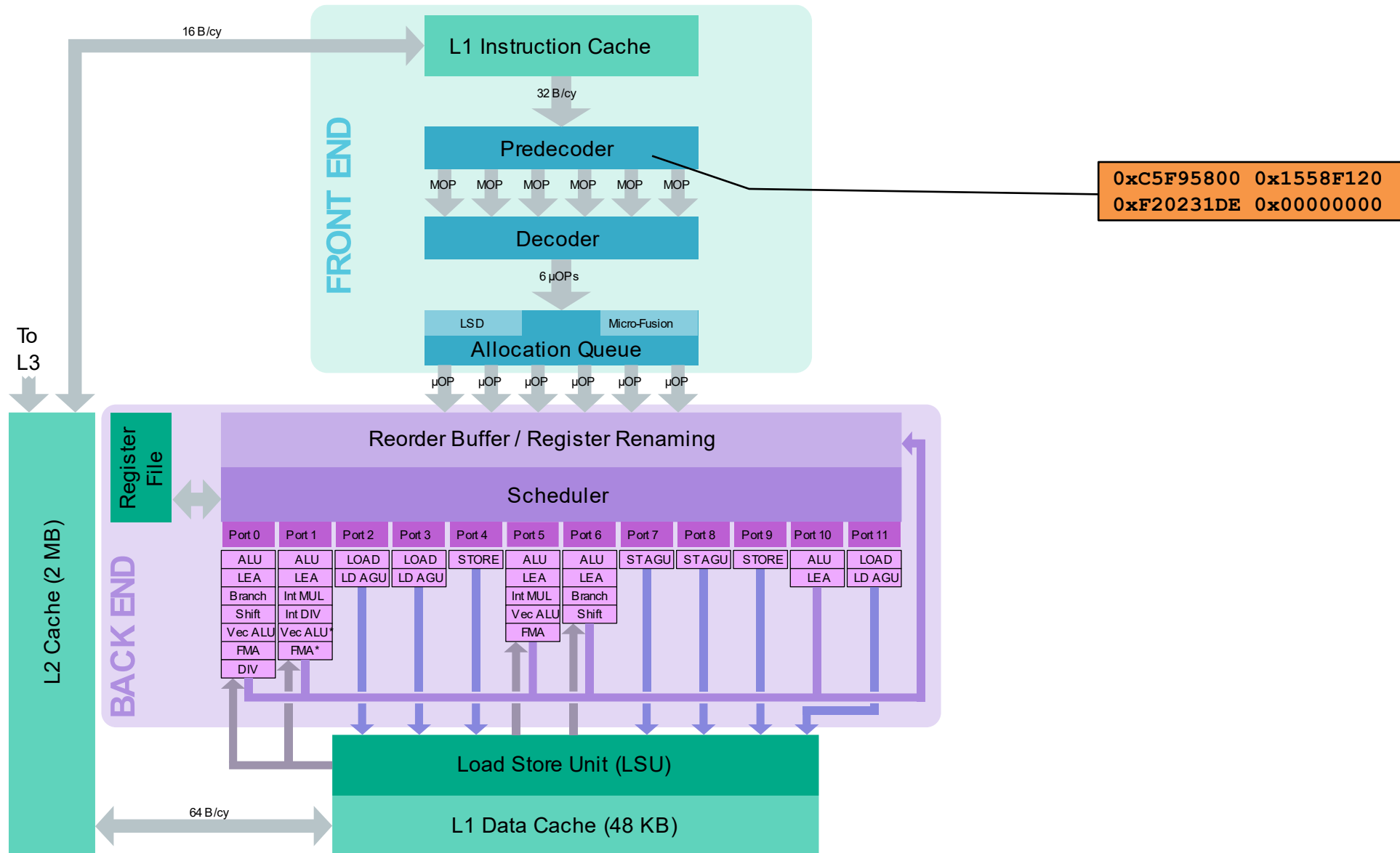
Basic processor and core architecture – SPR



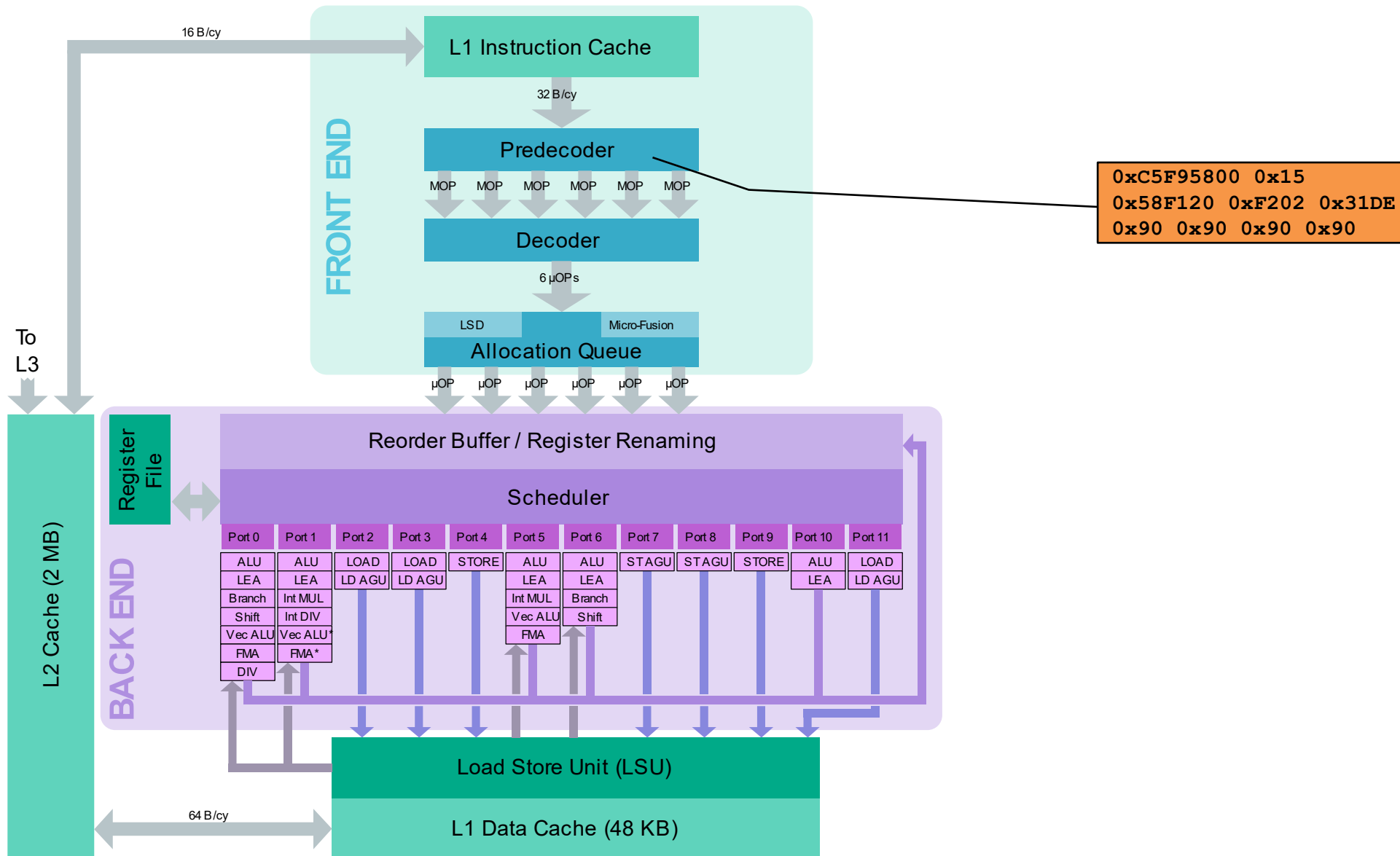
Basic processor and core architecture – SPR



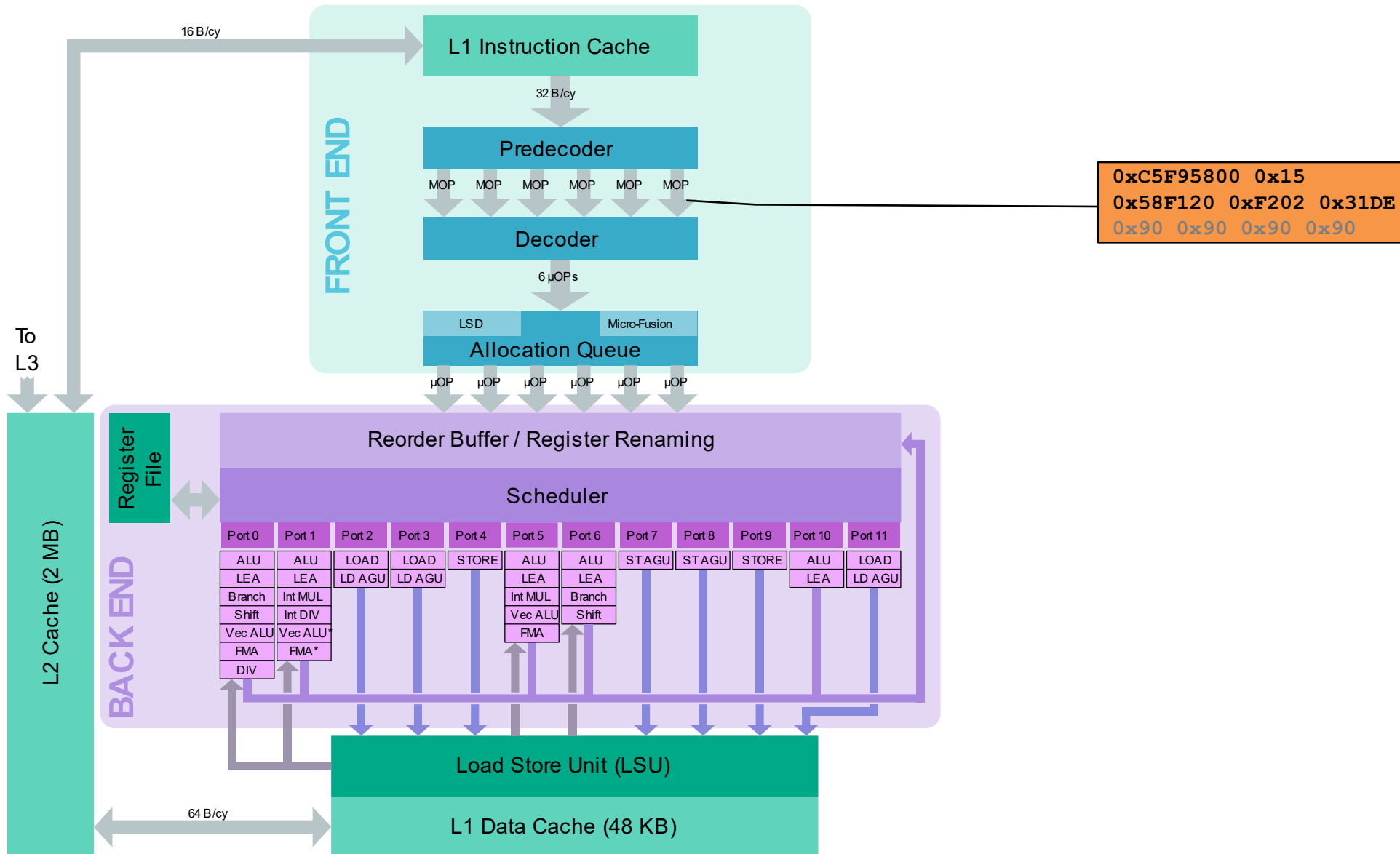
Basic processor and core architecture – SPR



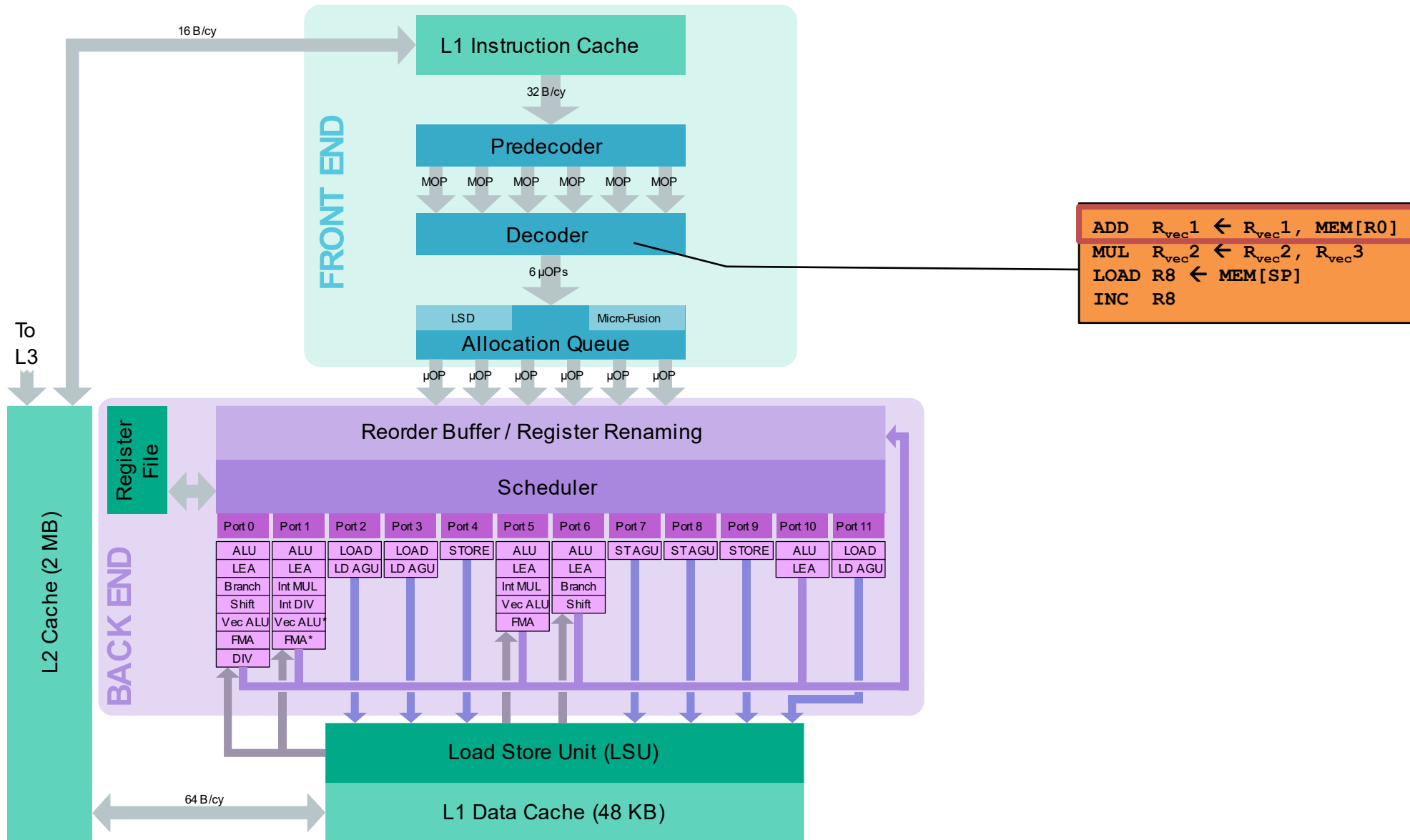
Basic processor and core architecture – SPR



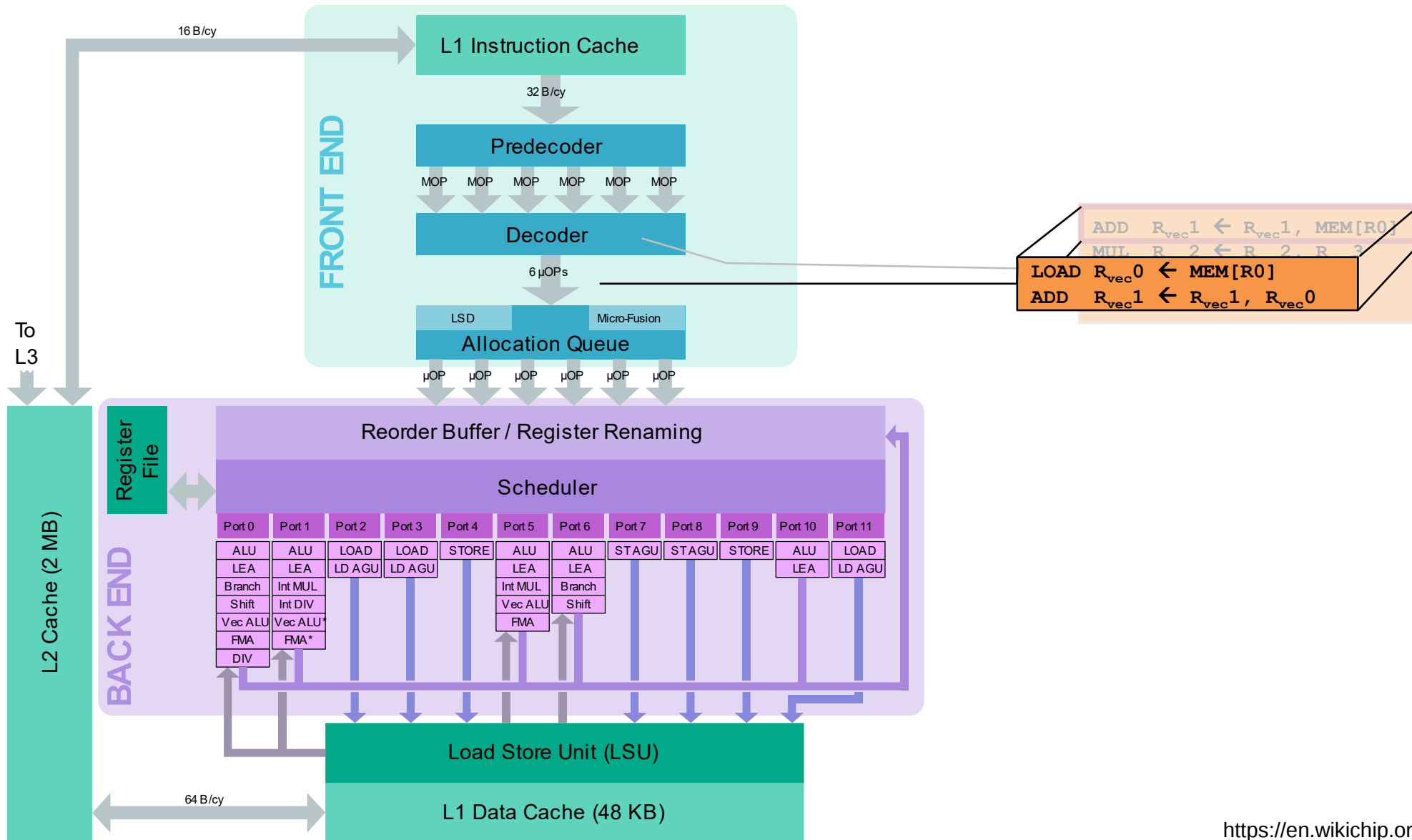
Basic processor and core architecture – SPR



Basic processor and core architecture – SPR

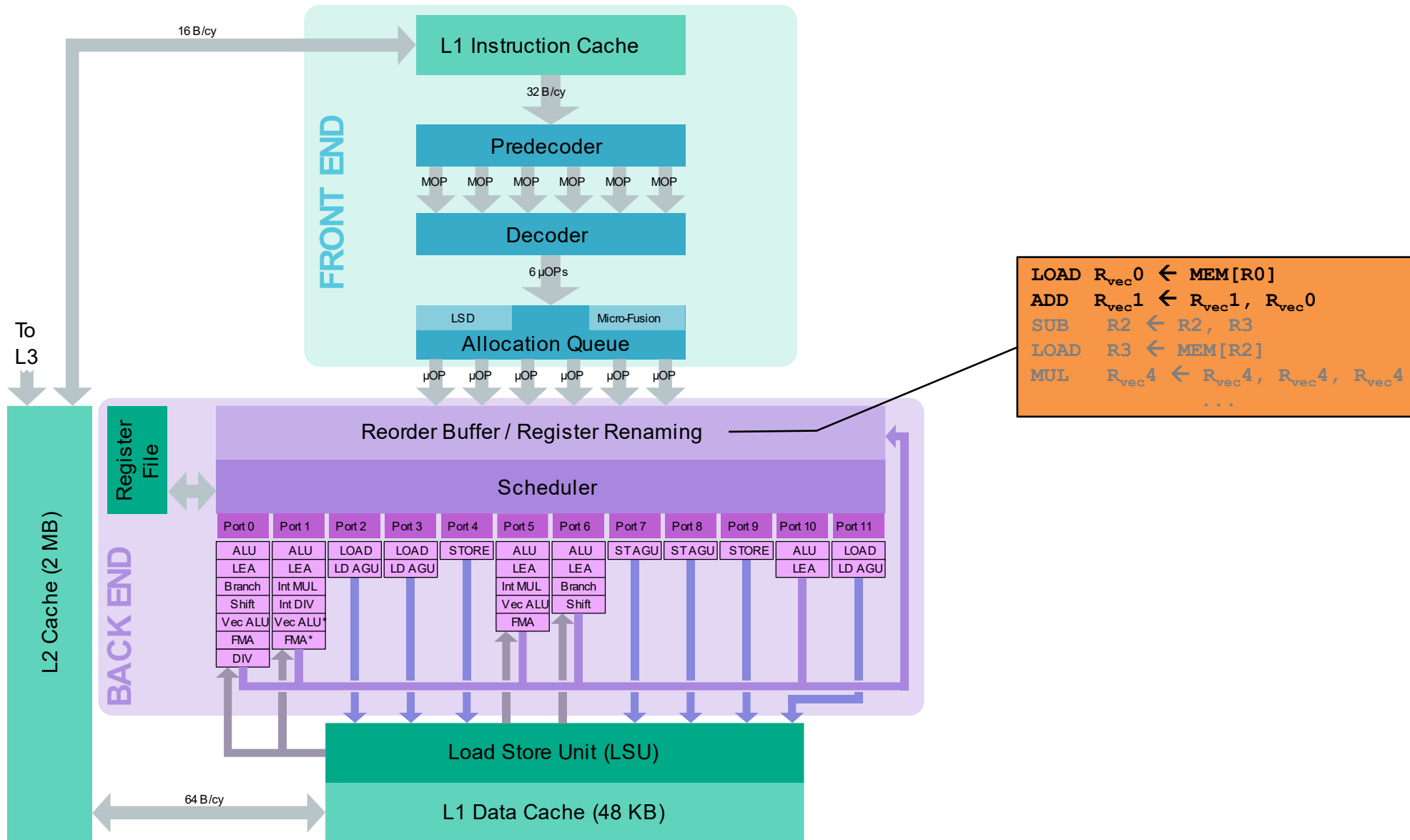


Basic processor and core architecture – SPR

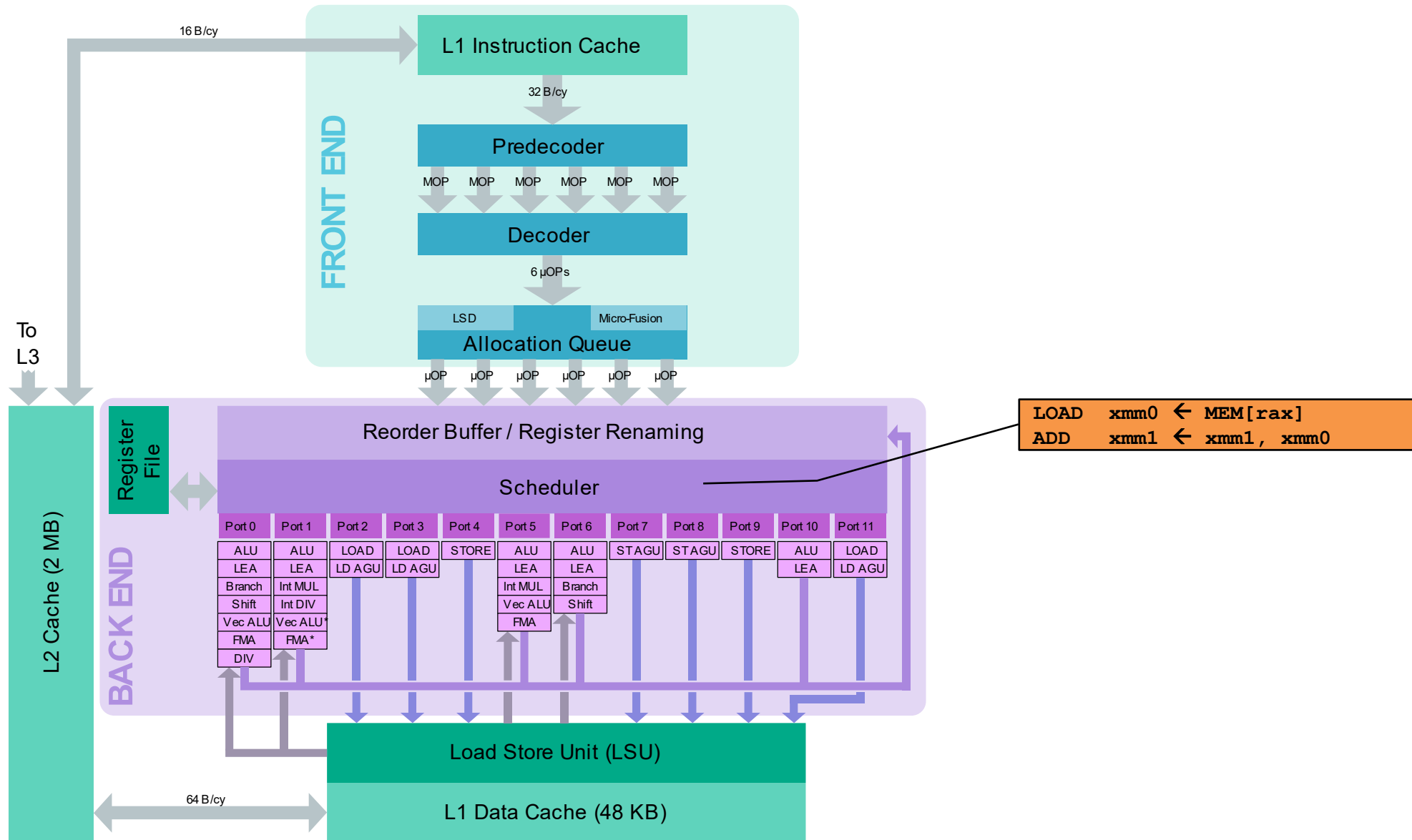


https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

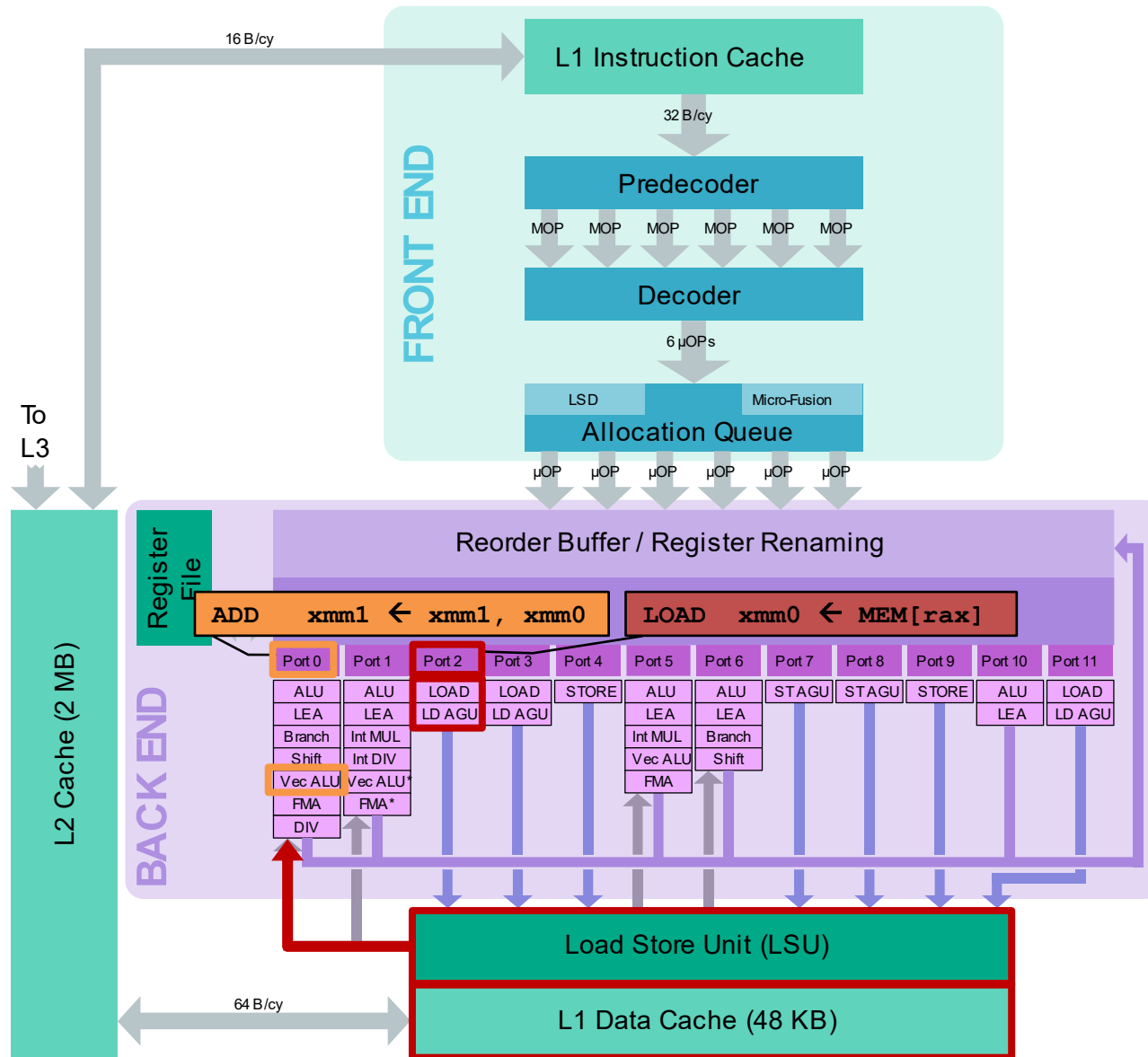
Basic processor and core architecture – SPR



Basic processor and core architecture – SPR



Basic processor and core architecture – SPR



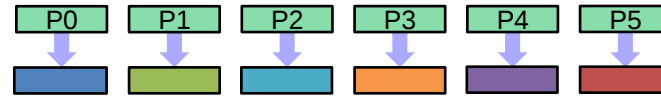
Hands-On #0: Out-of-Order Execution

→ <https://go-nhr.de/CLPE-ex0>

Hands-On: Out-of-Order Execution

Dot product

Machine model:



Instructions: ● ● ● ● ● ●
each with a reciprocal throughput
and latency of 1 cy

→ Moodle, hands-on #0 (both Multiple-Choice and Drag&Drop)



Friedrich-Alexander-Universität
Erlangen-Nürnberg



Erlangen National High Performance
Computing Center

Introduction to the x86 ISA (Instruction Set Architecture)



SIMD with masking – sum reduction with condition

```
double sum = 0.0;

for (int i=0; i<size; i++){
    if (data[i] > 0.0)
        sum += data[i];
}
```

Bulk loop code
(8x4-way unrolled)

..B1.38:

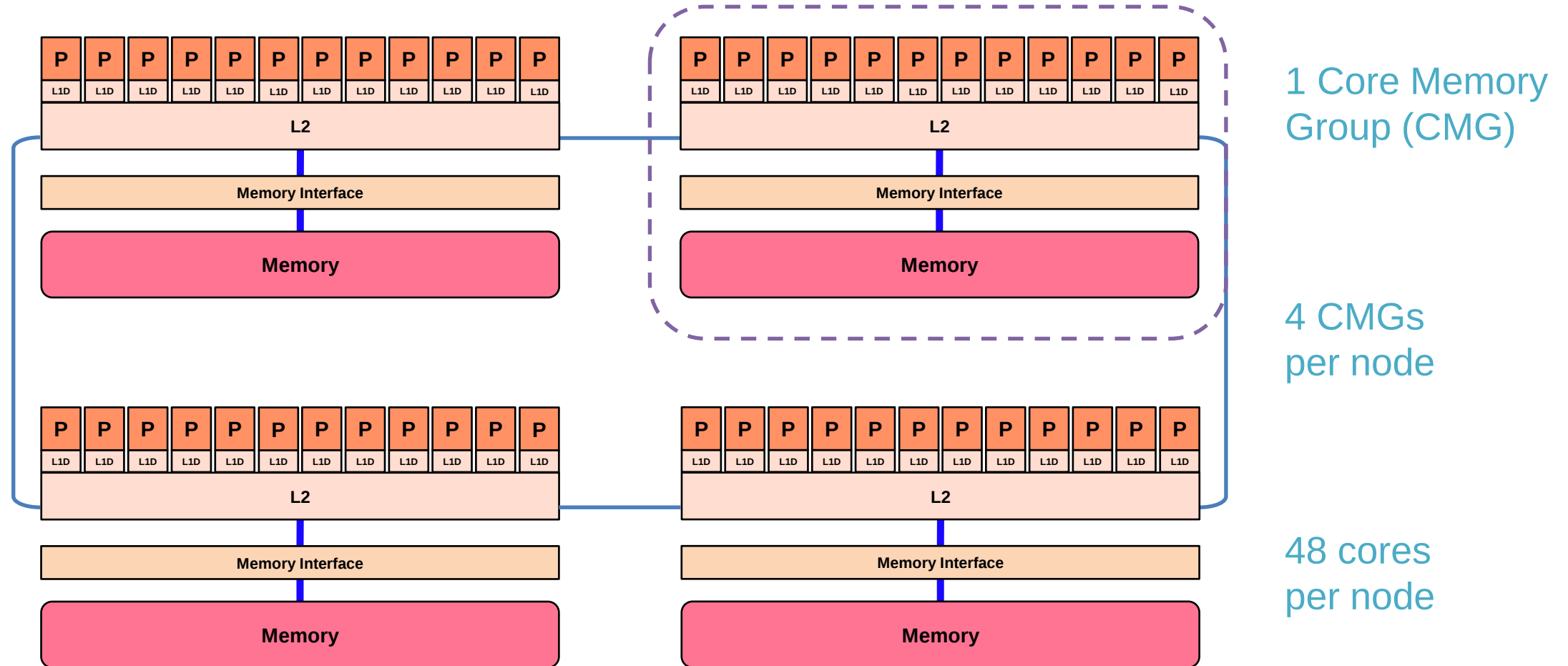
```
vmovups    (%r15,%rcx,8), %zmm6
vmovups    64(%r15,%rcx,8), %zmm7
vmovups    128(%r15,%rcx,8), %zmm8
vmovups    192(%r15,%rcx,8), %zmm9
vcmppd     $14, %zmm10, %zmm6, %k1
vcmppd     $14, %zmm10, %zmm7, %k2
vcmppd     $14, %zmm10, %zmm8, %k3
vcmppd     $14, %zmm10, %zmm9, %k4
vaddpd     %zmm6, %zmm5, %zmm5{%k1}
vaddpd     %zmm7, %zmm4, %zmm4{%k2}
vaddpd     %zmm8, %zmm3, %zmm3{%k3}
vaddpd     %zmm9, %zmm2, %zmm2{%k4}
addq       $32, %rcx
cmpq       %r14, %rcx
jb         ..B1.38
```

SIMD mask
generation

masked SIMD
ADDs
(accumulates)

A64FX core architecture and AArch64 Arm ISA

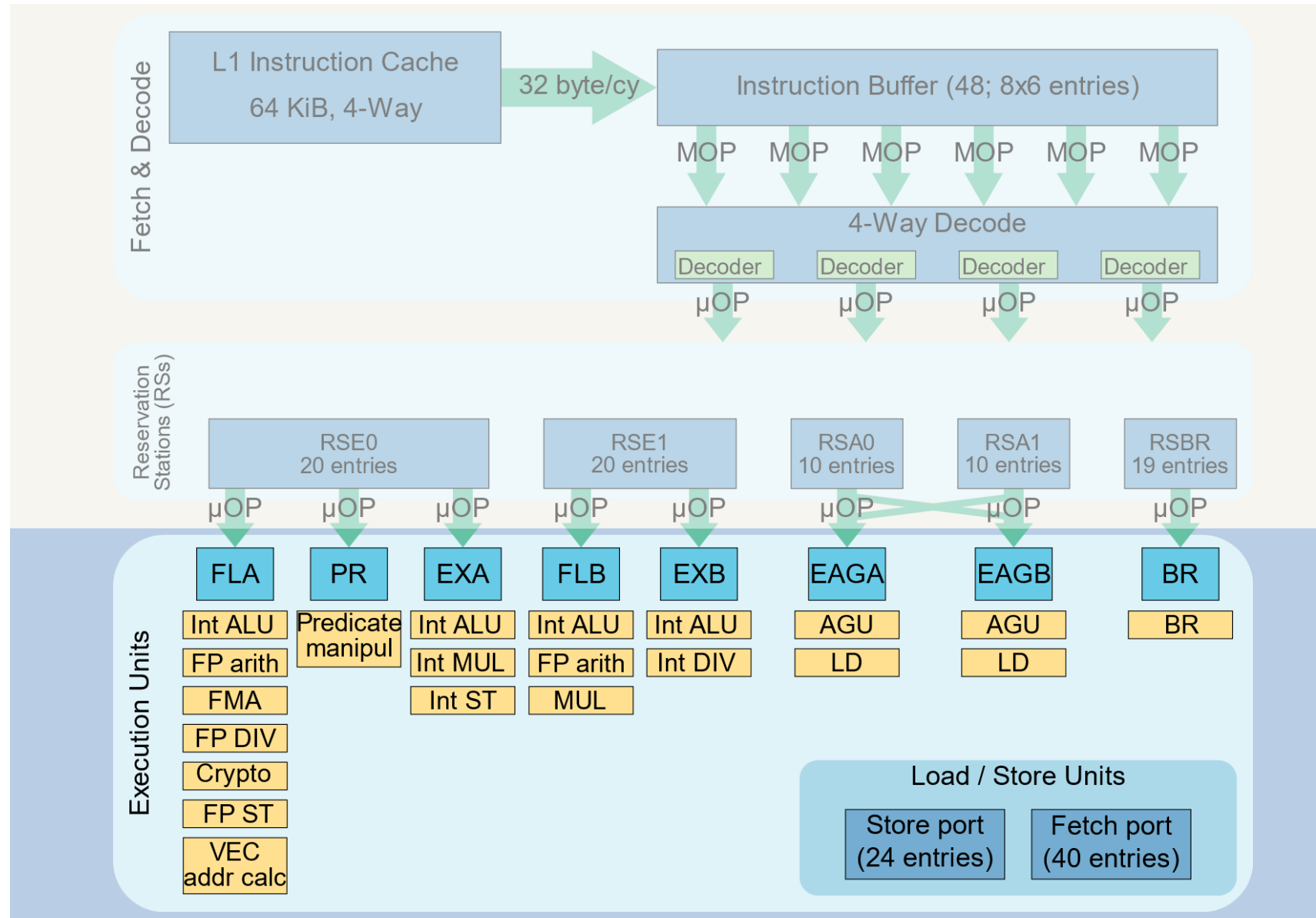
Node architecture of A64FX – FX700



Port model for the A64FX

Frontend

Backend



AArch64 ISA – differences to x86

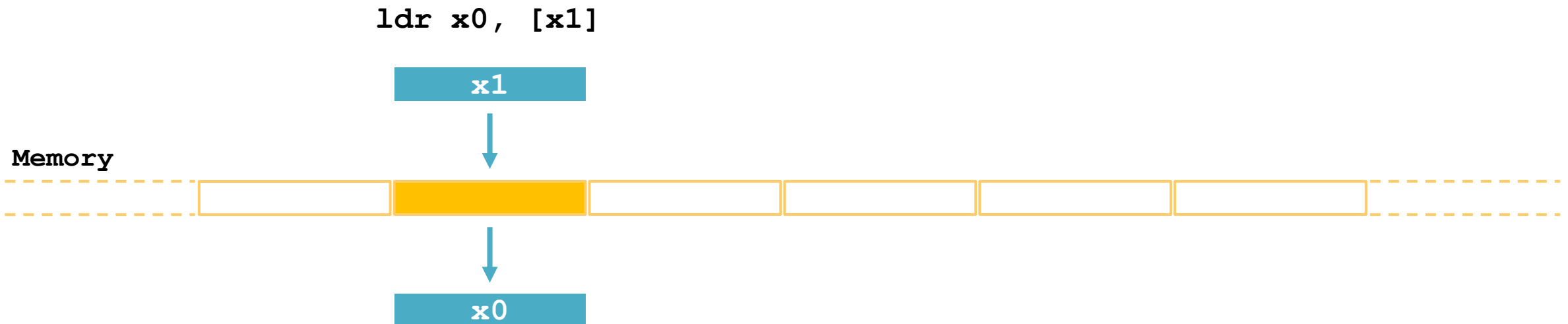
- Addressing Modes:

- Simple ([BASE])
`ldr x0, [x1]`
- Offset ([BASE, OFFSET])
`ldr x0, [x1, #64]`
- Modified Offset
`ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!)
`ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET)
`ldr x0, [x1], #64`

AArch64 ISA – differences to x86

■ Addressing Modes:

- | | |
|---------------------------------|--------------------------------------|
| ▪ Simple ([BASE]) | <code>ldr x0, [x1]</code> |
| ▪ Offset ([BASE, OFFSET]) | <code>ldr x0, [x1, #64]</code> |
| ▪ Modified Offset | <code>ldr x0, [x1, x2, lsl 3]</code> |
| ▪ Pre-indexed ([BASE, OFFSET]!) | <code>ldr x0, [x1, #64]!</code> |
| ▪ Post-indexed ([BASE], OFFSET) | <code>ldr x0, [x1], #64</code> |

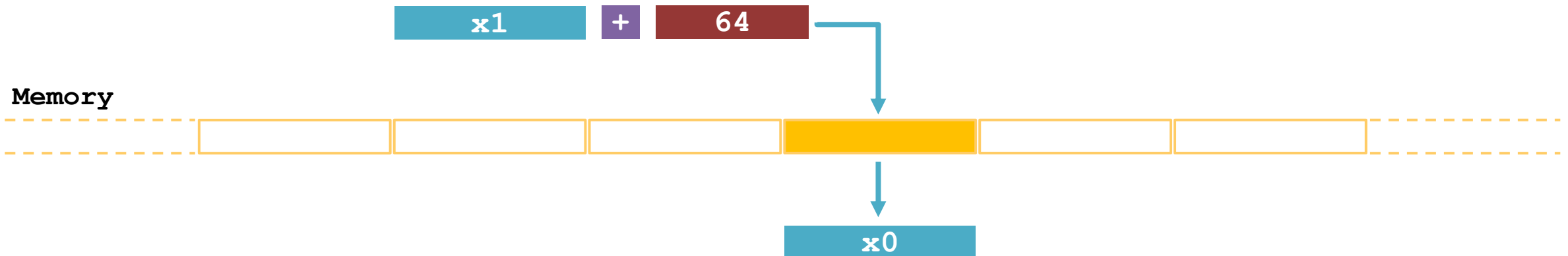


AArch64 ISA – differences to x86

■ Addressing Modes:

- Simple ([BASE]) `ldr x0, [x1]`
- Offset ([BASE, OFFSET]) `ldr x0, [x1, #64]`
- Modified Offset `ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!) `ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET) `ldr x0, [x1], #64`

`ldr x0, [x1, #64]`

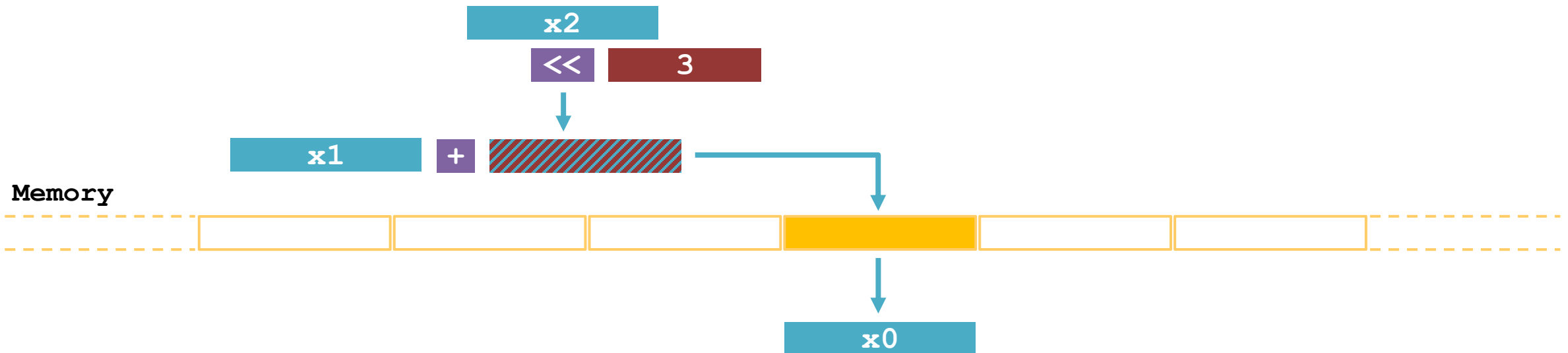


AArch64 ISA – differences to x86

■ Addressing Modes:

- Simple ([BASE]) `ldr x0, [x1]`
- Offset ([BASE, OFFSET]) `ldr x0, [x1, #64]`
- Modified Offset `ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!) `ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET) `ldr x0, [x1], #64`

`ldr x0, [x1, x2, lsl 3]`

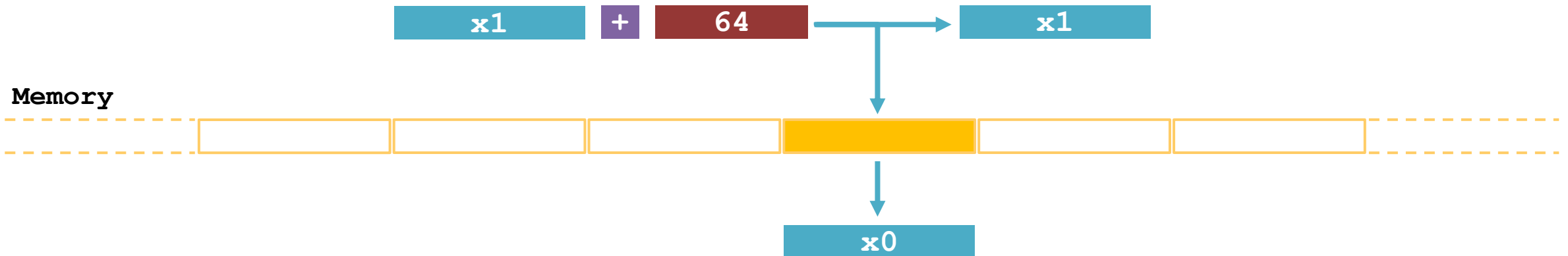


AArch64 ISA – differences to x86

■ Addressing Modes:

- Simple ([BASE]) `ldr x0, [x1]`
- Offset ([BASE, OFFSET]) `ldr x0, [x1, #64]`
- Modified Offset `ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!) `ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET) `ldr x0, [x1], #64`

`ldr x0, [x1, #64]!`

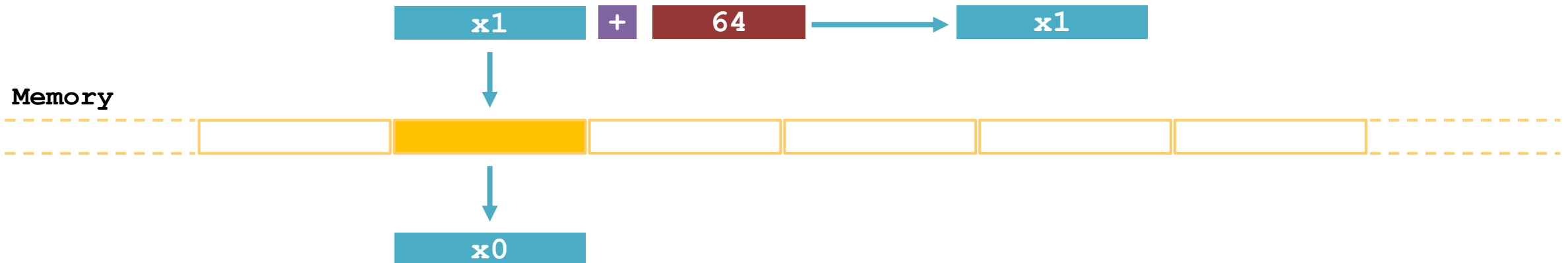


AArch64 ISA – differences to x86

■ Addressing Modes:

- Simple ([BASE]) `ldr x0, [x1]`
- Offset ([BASE, OFFSET]) `ldr x0, [x1, #64]`
- Modified Offset `ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!) `ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET) `ldr x0, [x1], #64`

`ldr x0, [x1], #64`



AArch64 ISA – differences to x86

- Addressing Modes:

- Simple ([BASE])
- Offset ([BASE, OFFSET])
- Modified Offset
- Pre-indexed ([BASE, OFFSET]!)
- Post-indexed ([BASE], OFFSET)

```
ldr x0, [x1]
```

```
ldr x0, [x1, #64]
```

```
ldr x0, [x1, x2, lsl 3]
```

```
ldr x0, [x1, #64]!
```

```
ldr x0, [x1], #64
```

- SVE provides predicate masks and more sophisticated instructions for gather/scatter, e.g.

- `ld1d z0.d, p0/z, [...]`

```
ld1d z0.d, p0/z, [x0, z1, uxtw]
```

p0

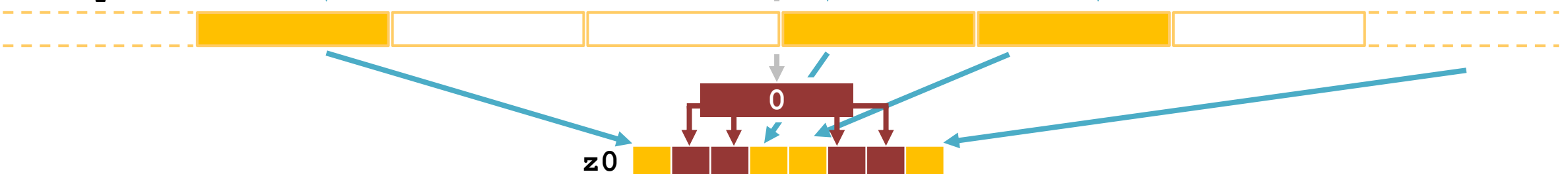
`x0 + uxtw(z1[0])`

`x0 + uxtw(z1[3])`

`x0 + uxtw(z1[4])`

...

Memory



AArch64 ISA – differences to x86

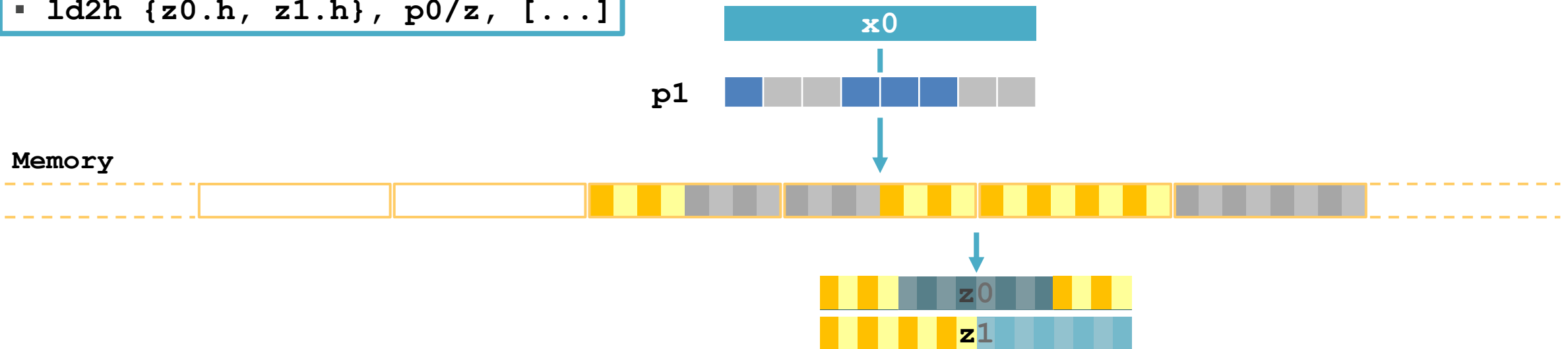
- Addressing Modes:

- Simple ([BASE])
`ldr x0, [x1]`
- Offset ([BASE, OFFSET])
`ldr x0, [x1, #64]`
- Modified Offset
`ldr x0, [x1, x2, lsl 3]`
- Pre-indexed ([BASE, OFFSET]!)
`ldr x0, [x1, #64]!`
- Post-indexed ([BASE], OFFSET)
`ldr x0, [x1], #64`

- SVE provides predicate masks and more sophisticated instructions for gather/scatter, e.g.

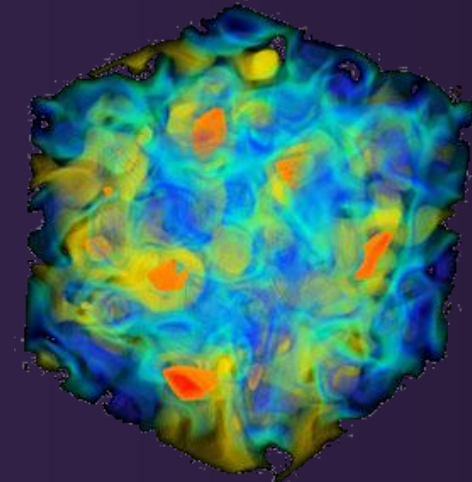
- `ld1d z0.d, p0/z, [...]`
- `ld2h {z0.h, z1.h}, p0/z, [...]`

`ld2w {z0.s, z1.s}, p1/z, [x0]`



Case Study: Domain Wall (DW) Kernel

from Quantum Chromodynamics (QCD)



© Brookhaven National Lab

Based on:

C. Alappat, N. Meyer, J. Laukemann, T. Gruber,
G. Hager, G. Wellein, and T. Wettig:

ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.
Concurrency and Computation: Practice and Experience, e6512 (2021).

DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

Domain Wall (DW) Kernel – Context

- Lattice QCD simulates the strong interaction
- Iterative multigrid techniques on regular (4D or 5D) lattices
- Core component: Apply Dirac operator D to quark-field vector Ψ
- Domain Wall (DW) formulation: quark field lives on 4D boundary of a 5D space-time volume $V_4 \times L_s$

$$(D\Psi)(n, s)_{\alpha a} =$$

$$\sum_{\mu=1} \sum_{\beta=1} \sum_{b=1} \left\{ U_{\mu}(n)_{ab} (1 + \gamma_{\mu})_{\alpha\beta} \psi(n + \hat{\mu}, s)_{\beta b} + U_{\mu}^{\dagger}(n - \hat{\mu})_{ab} (1 - \gamma_{\mu})_{\alpha\beta} \psi(n - \hat{\mu}, s)_{\beta b} \right\}$$

Hands-On #5: 2D Gauss-Seidel analysis

→ <https://go-nhr.de/CLPE-ex5>

Hands-On: Gauss-Seidel Method on SPR

Prediction [cy/it]	standard			optimized		
	-Ofast	-O3	-O1	-Ofast	-O3	-O1
icc 2021	9	9	14	7	7	6
icx 2022	9	9	8	8	8	8
icx 2024	9	11	10	8	10	10
GCC 14.2	8	10	10.5	8	4	6
Clang 18	10	10	10	10	4	4