Performance Engineering

Being Friendly to Your Hardware

IGNAS BAGDONAS

Cppcon
The C++ Conference

2024
September 15 - 20

Being Friendly to Your Hardware

Performance Engineering

A gentle introduction to hardware for software engineers

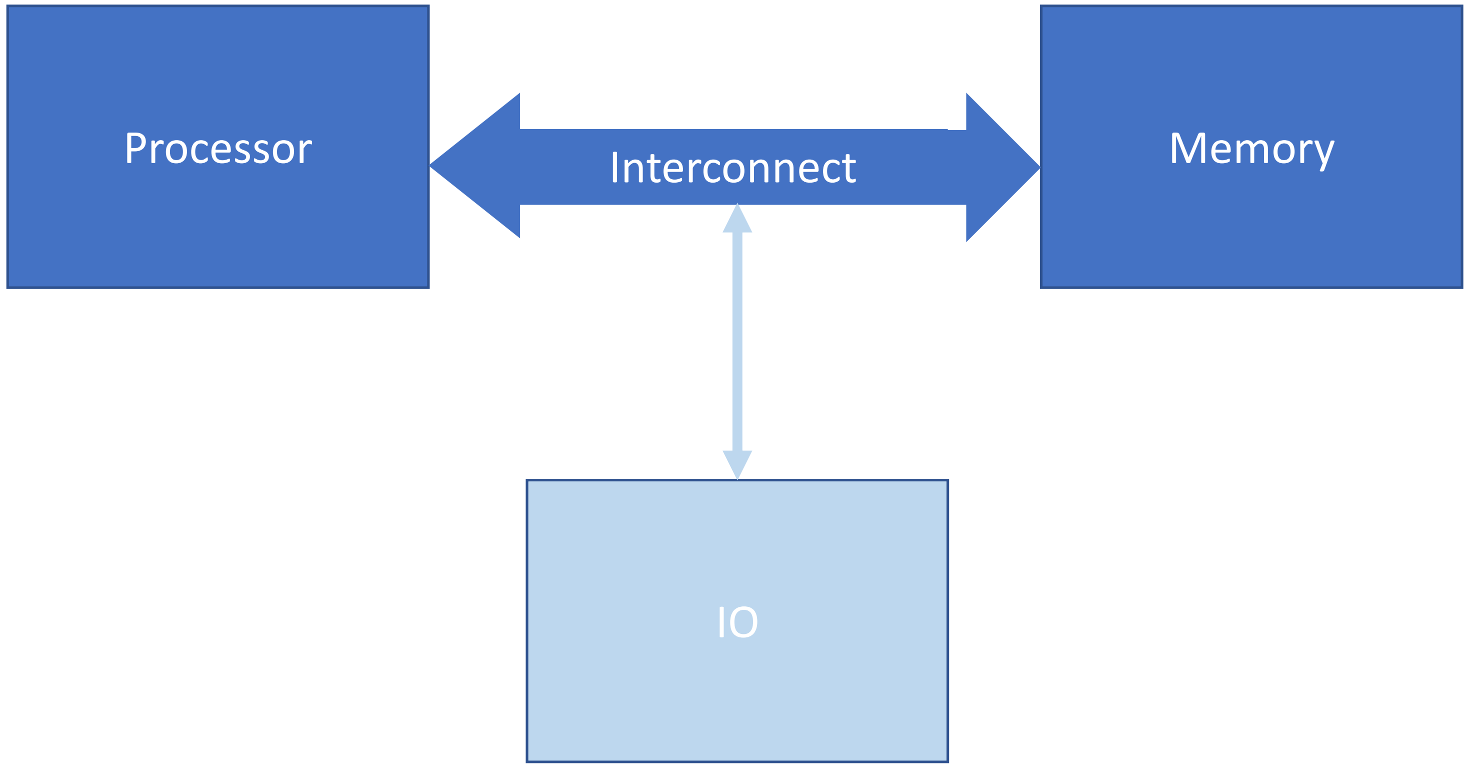
Where does C++ run?

On an abstract C++ machine

On an abstract C++ machine?

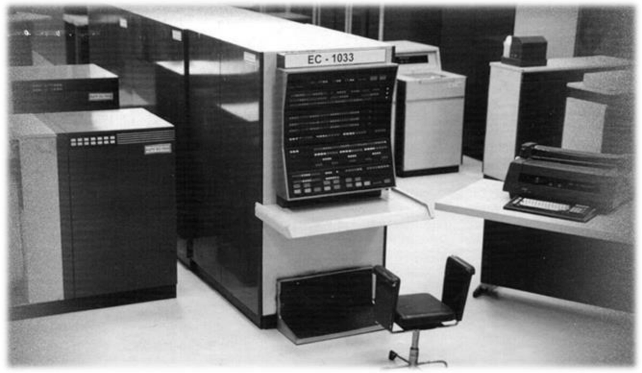
C++ runs on a computer platform

Computer platform



Computer platform

Processor + Interconnect + Memory

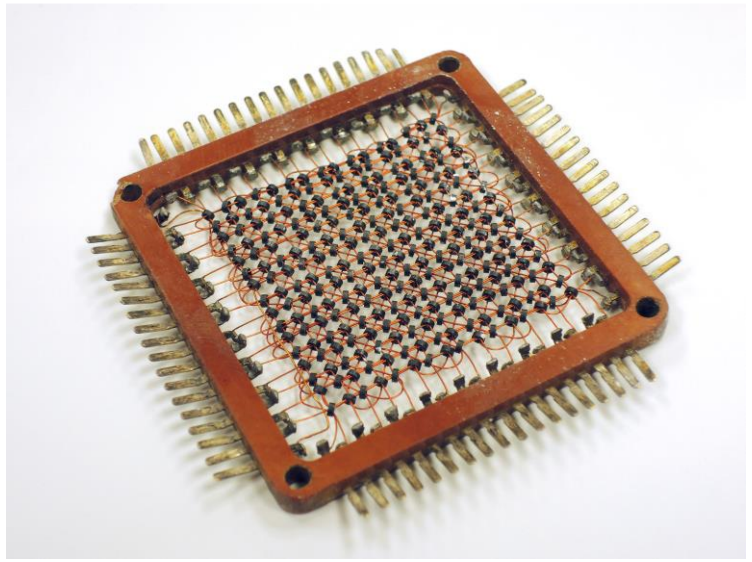


Computer platform

Processor + Interconnect + Memory

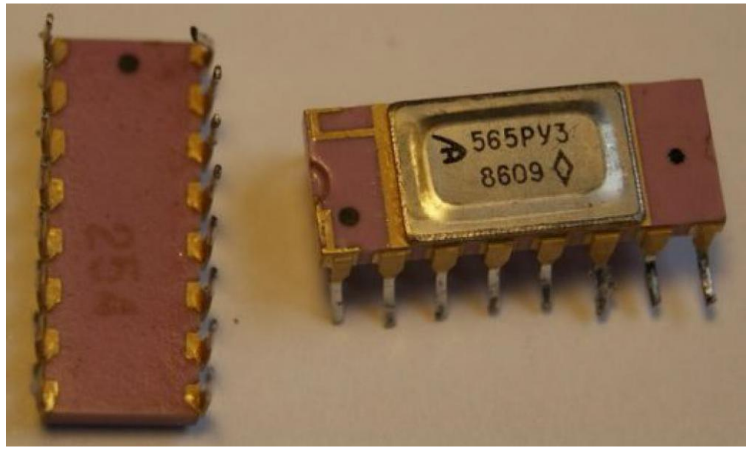
Computer platform

Processor + Interconnect + Memory



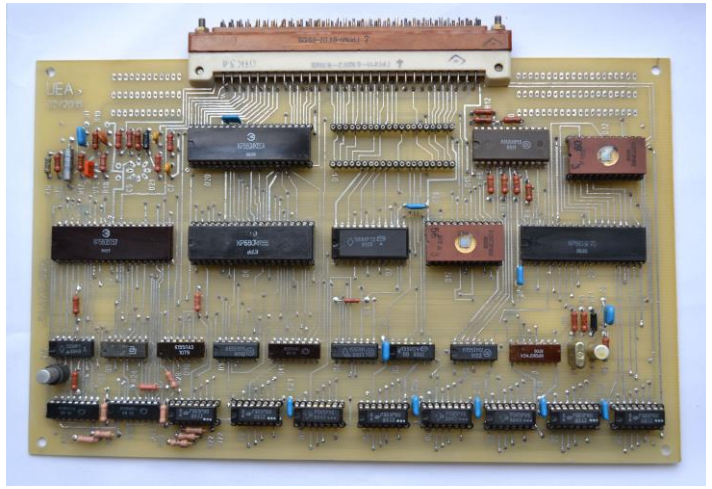
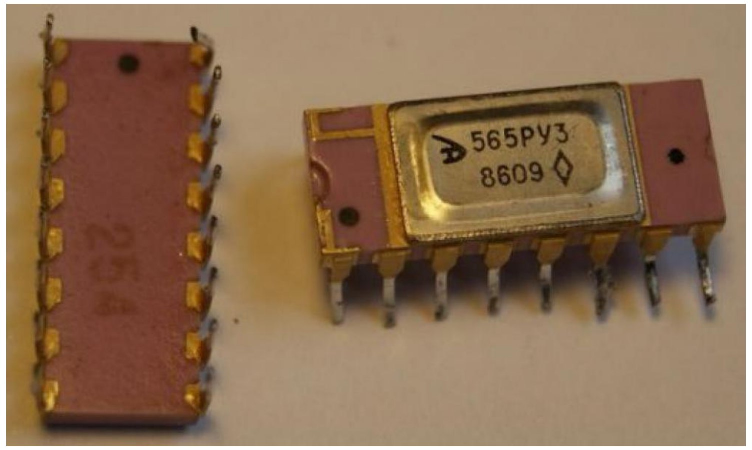
Computer platform

Processor + Interconnect + Memory



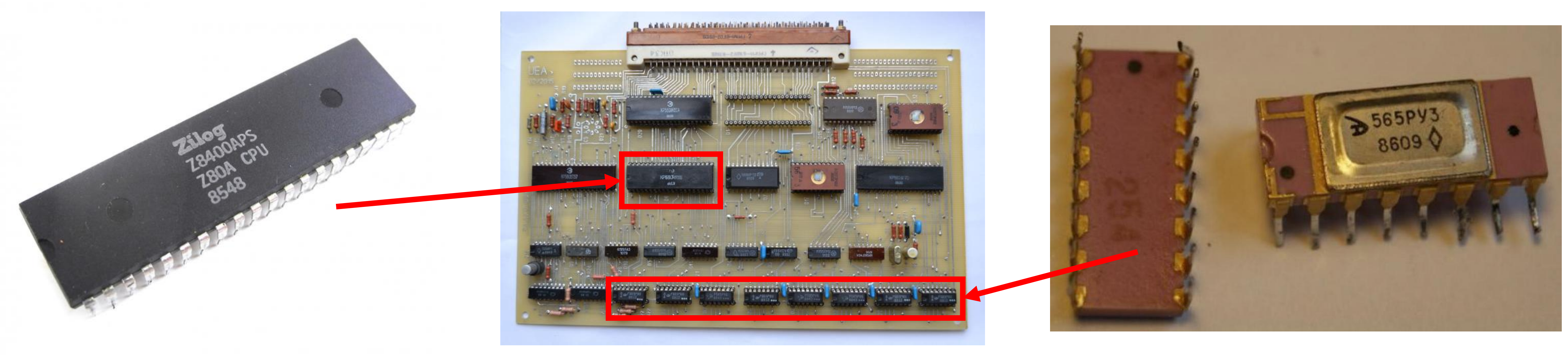
Computer platform

Processor + Interconnect + Memory

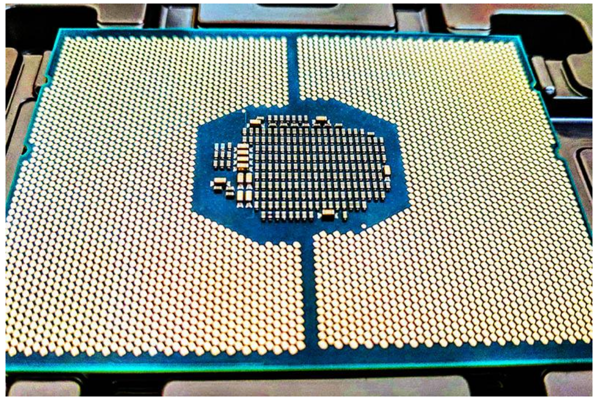
Computer platform

Processor + Interconnect + Memory



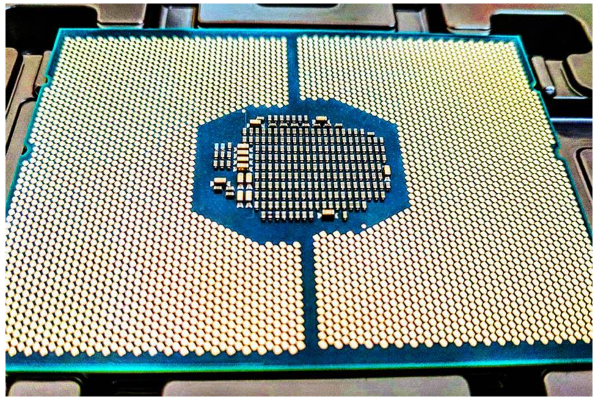
Computer system

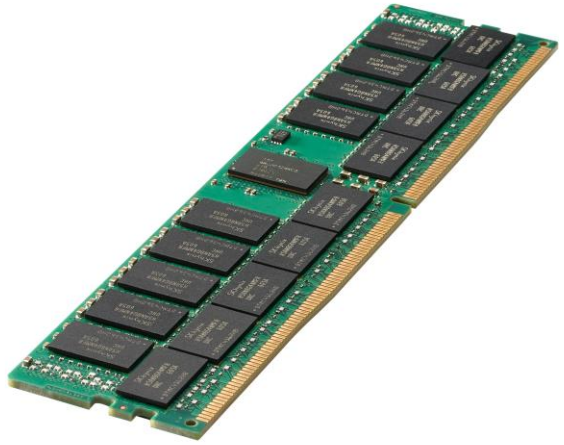
Processor cores + Interconnect + Memory



Computer system

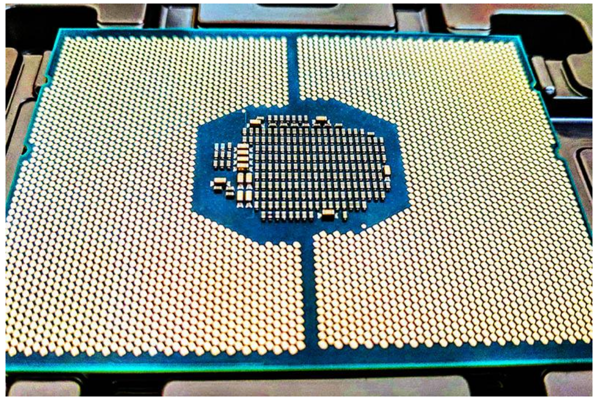
Processor cores + Interconnect + Memory





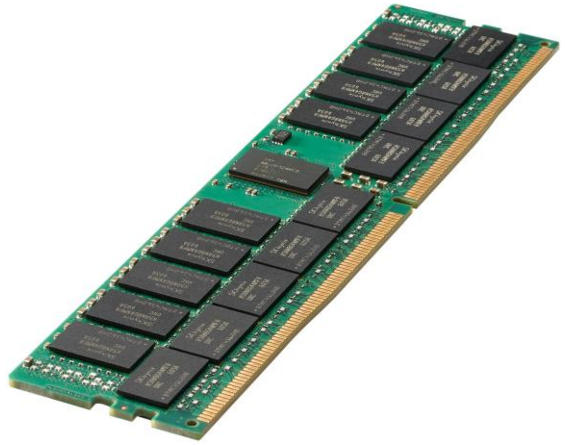
Computer system

Processor cores + Interconnect + Memory



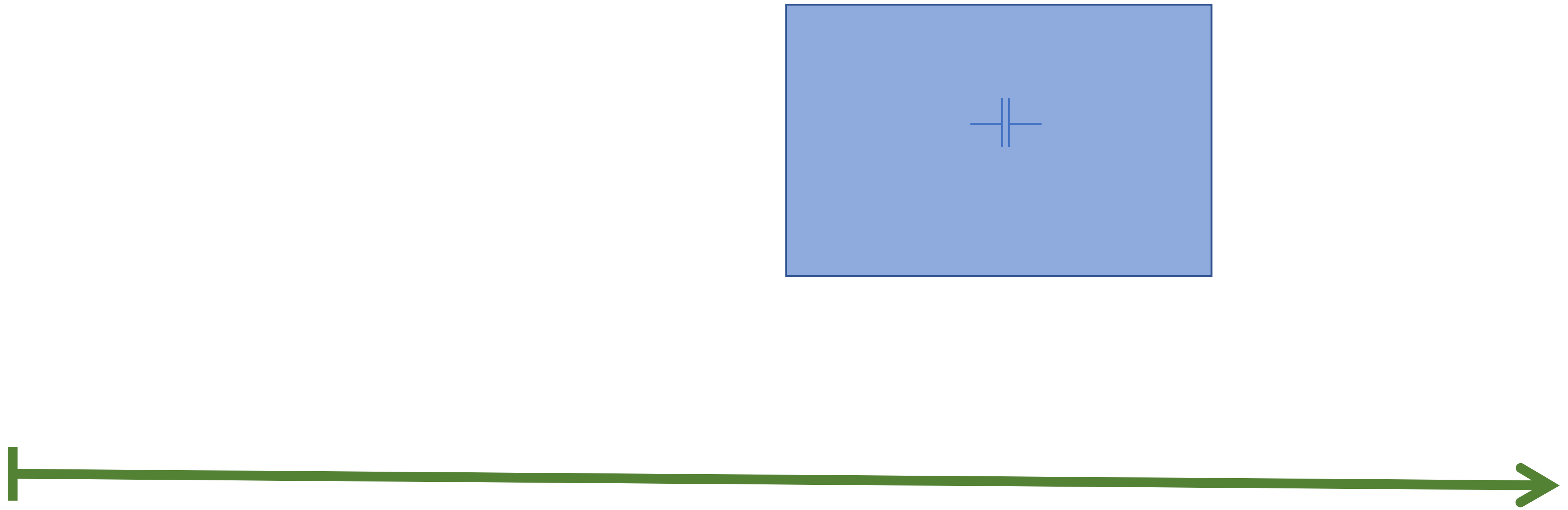
Complex and scary

And mostly out of scope of our discussion anyway



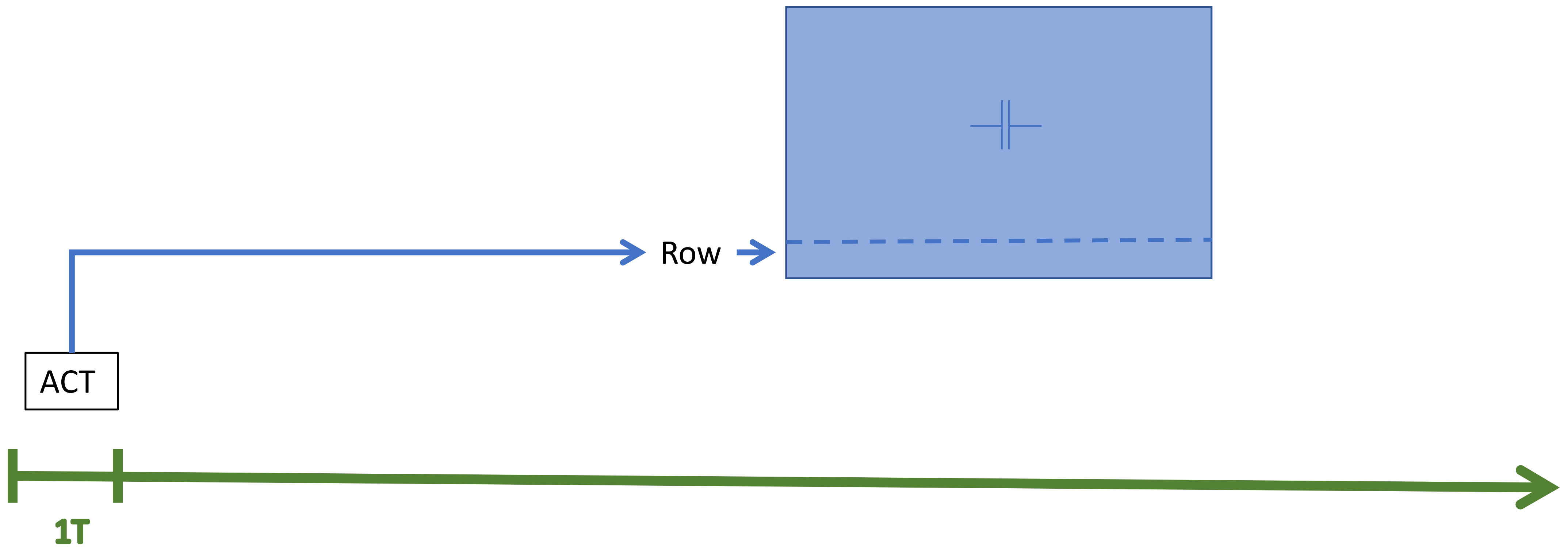
Memory

- Addressable capacitor array
- Short term nonvolatile
- Speeds – capacitor array vs bus interface



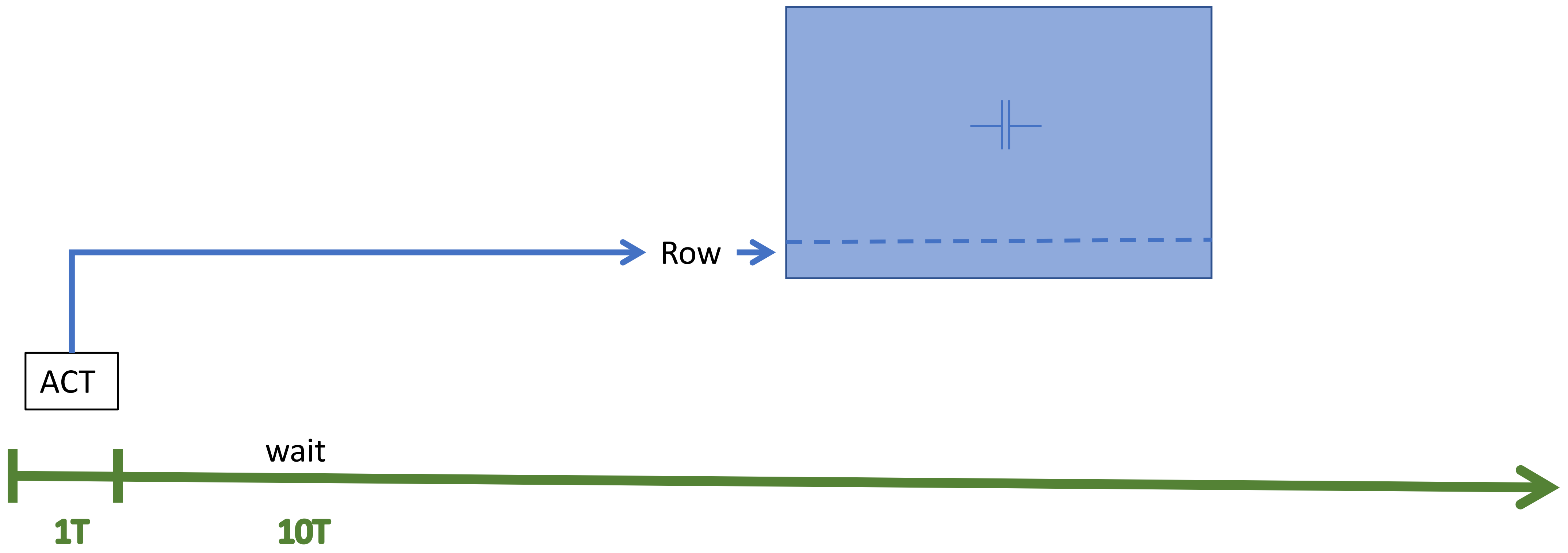
Memory

- Open a row
- Send part of address - one command



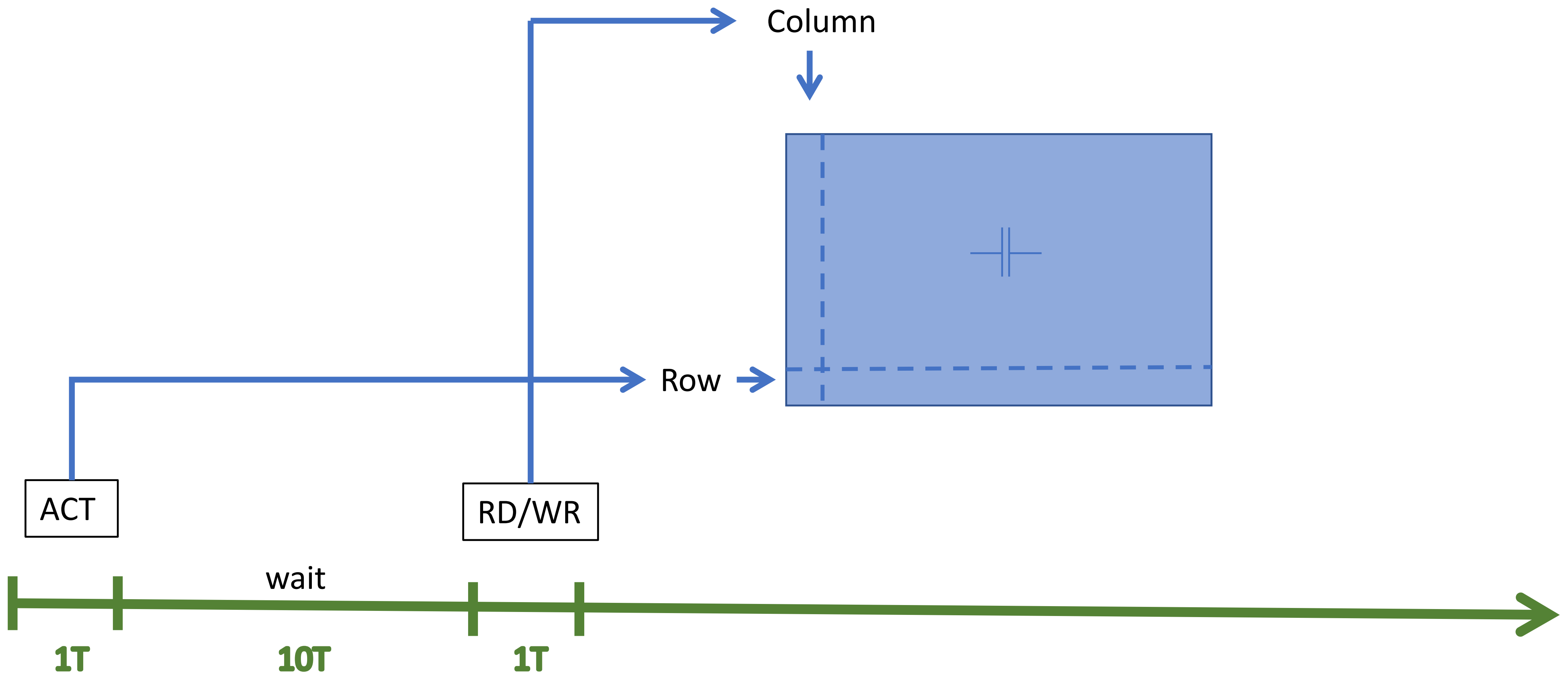
Memory

- Wait for a portion of array to get selected.



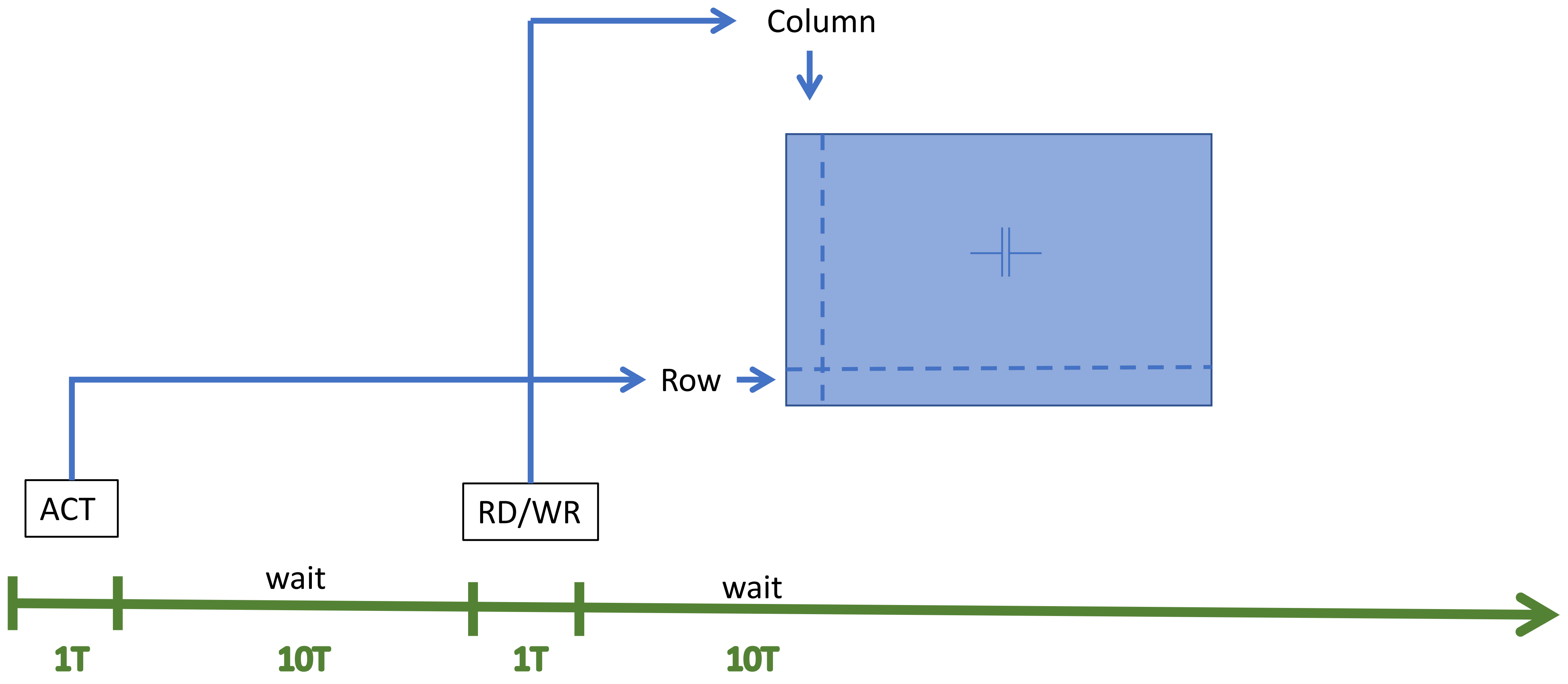
Memory

- Send column address and actual memory operation – another command



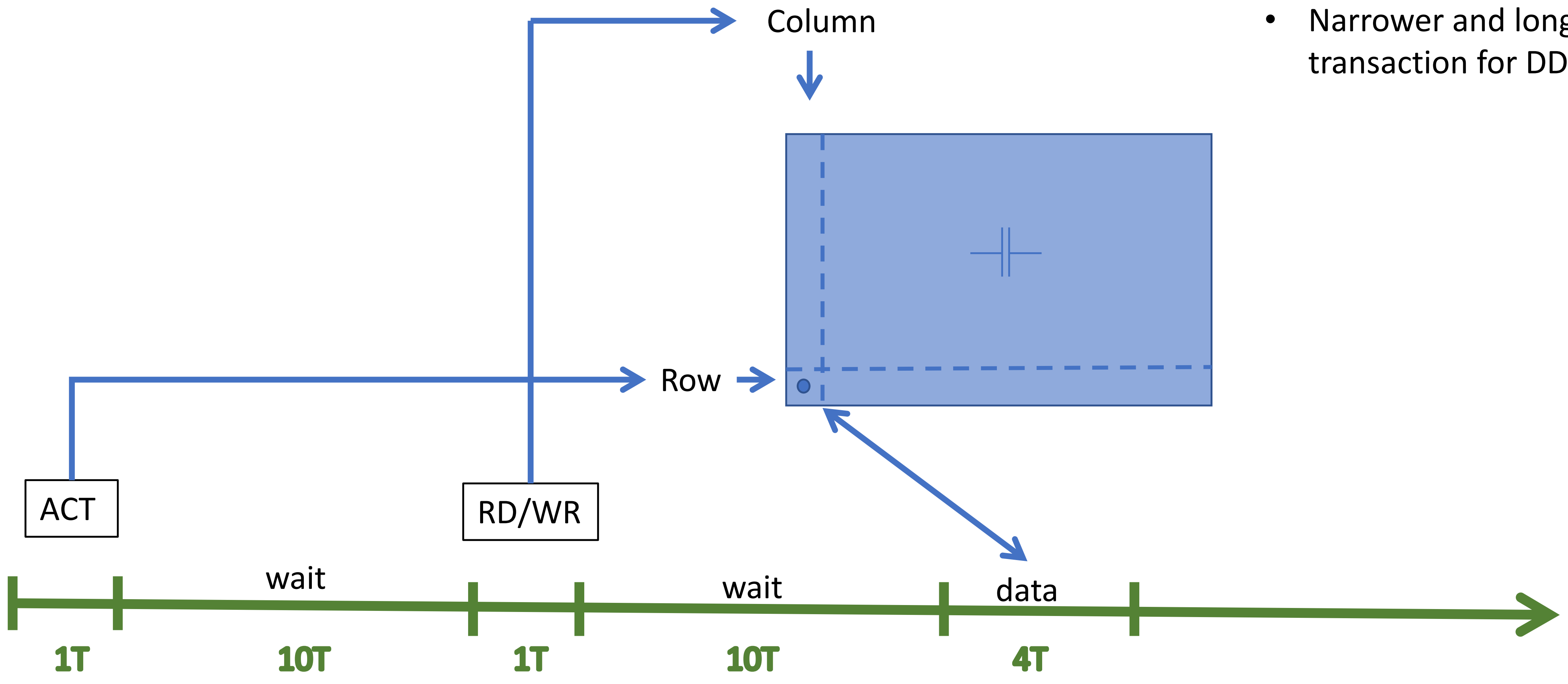
Memory

- Wait for a portion of row to get selected and capacitors sensed.



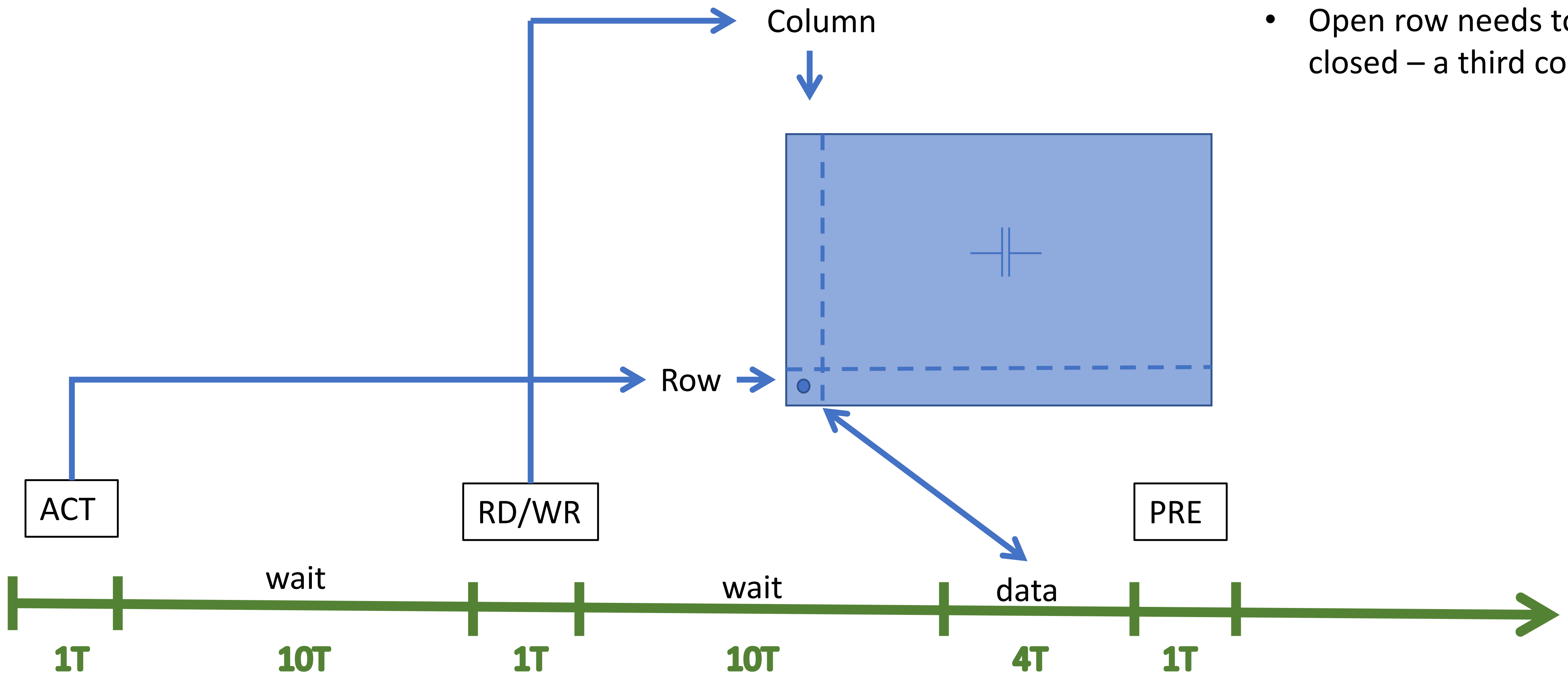
Memory

- Actual data transfer.
- Wider and shorter transaction for DDR4
- Narrower and longer transaction for DDR5



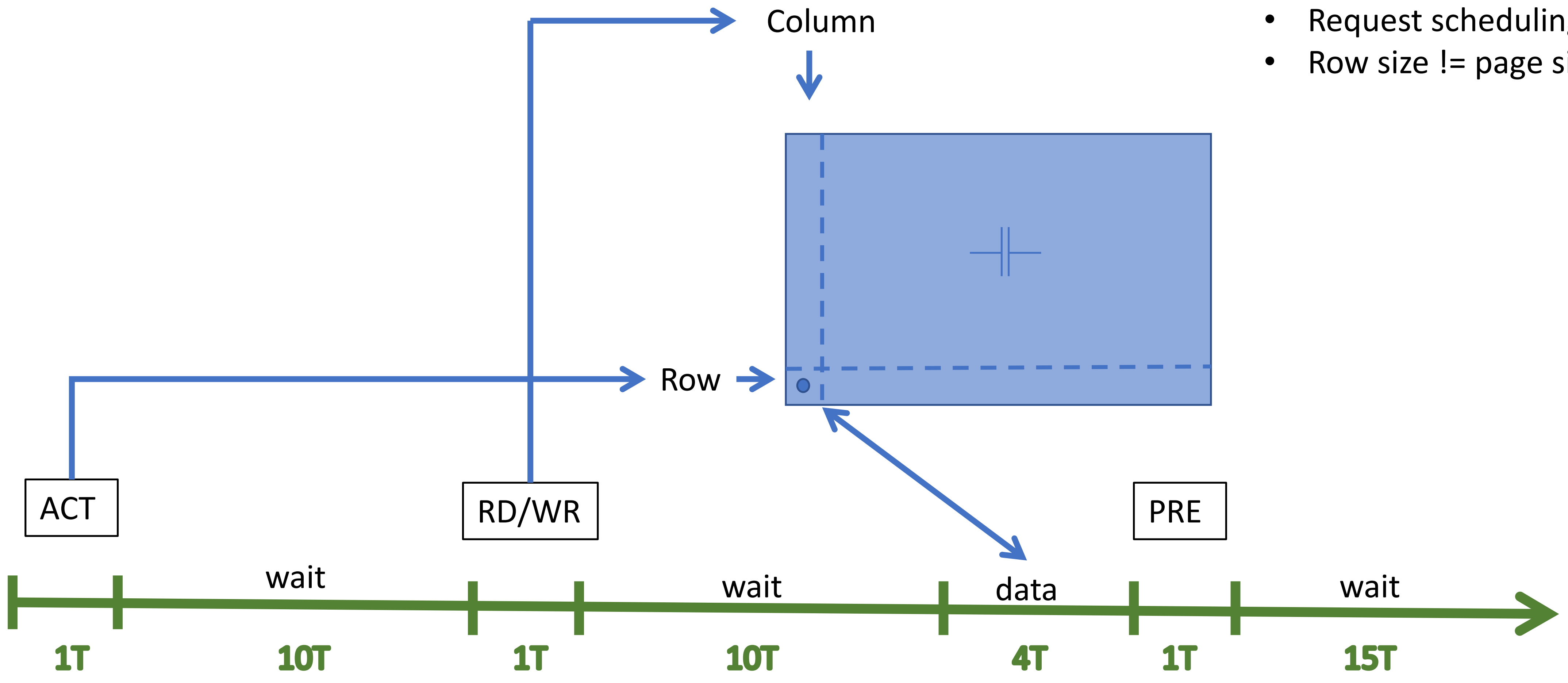
Memory

- Capacitor array access is destructive
- Even for a read
- Open row needs to be closed – a third command



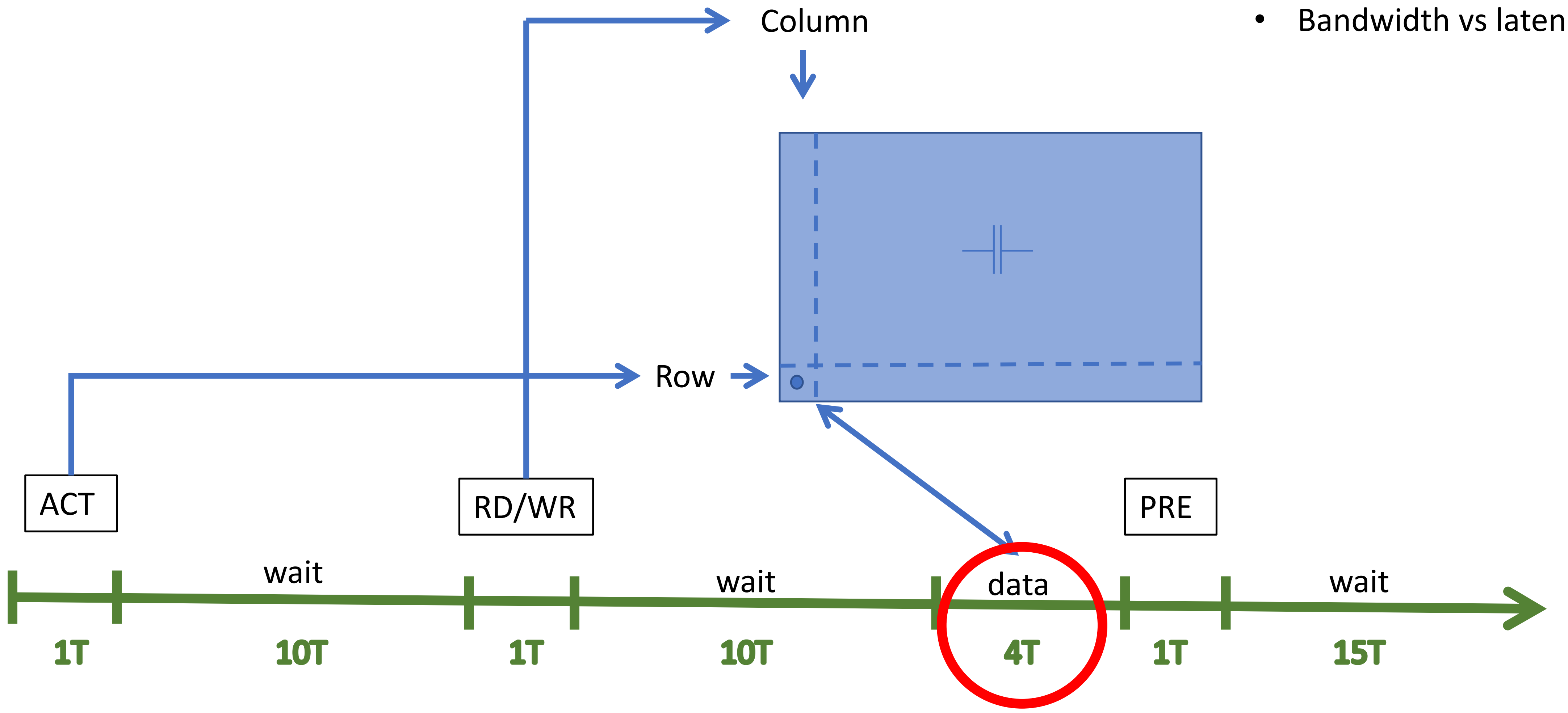
Memory

- One row active at a time
- All rows need to be refreshed periodically
- Request scheduling
- Row size != page size



Memory

- Usable utilization is quite far away from theoretical limit
- Bandwidth vs latency



Memory

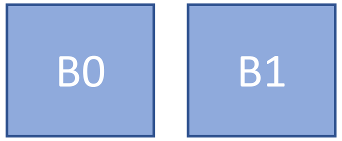
- DDR architecture – it is only a fraction of bus transaction that is in fact DDR
- Row addressing
- Column addressing
- Generation to generation capacitor array operates at mostly the same frequency, bus transfer speed in fact increases
- Multiple memory modules (sockets)
- Page size – CPU vs memory
- Address mapping

Memory

- One row open per bank
- Logical partitioning

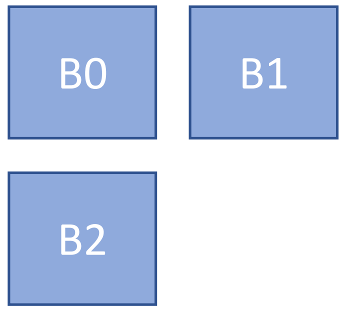
Memory

- One row open per bank
- Logical partitioning



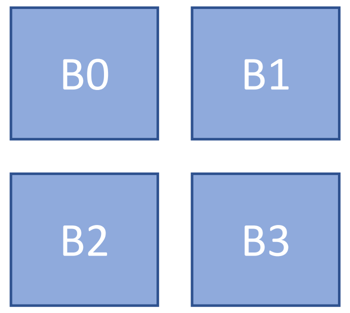
Memory

- One row open per bank
- Logical partitioning

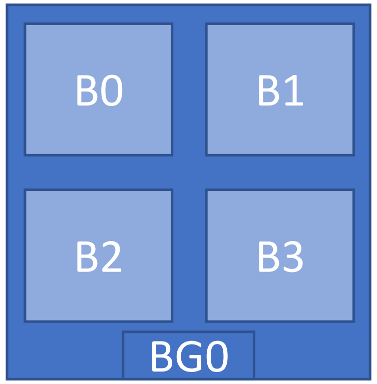


Memory

- One row open per bank
- Logical partitioning

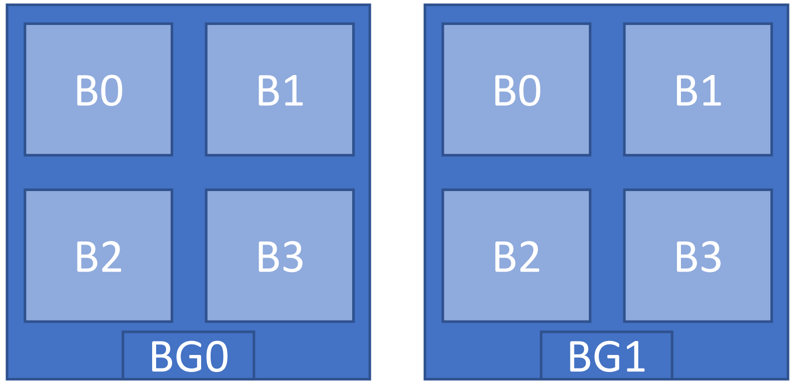


Memory



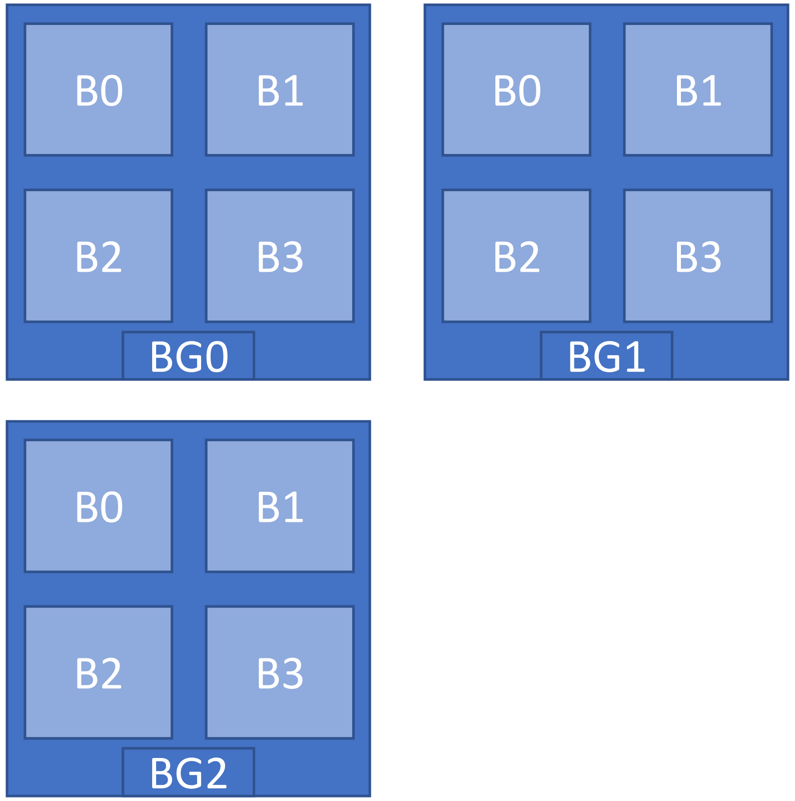
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



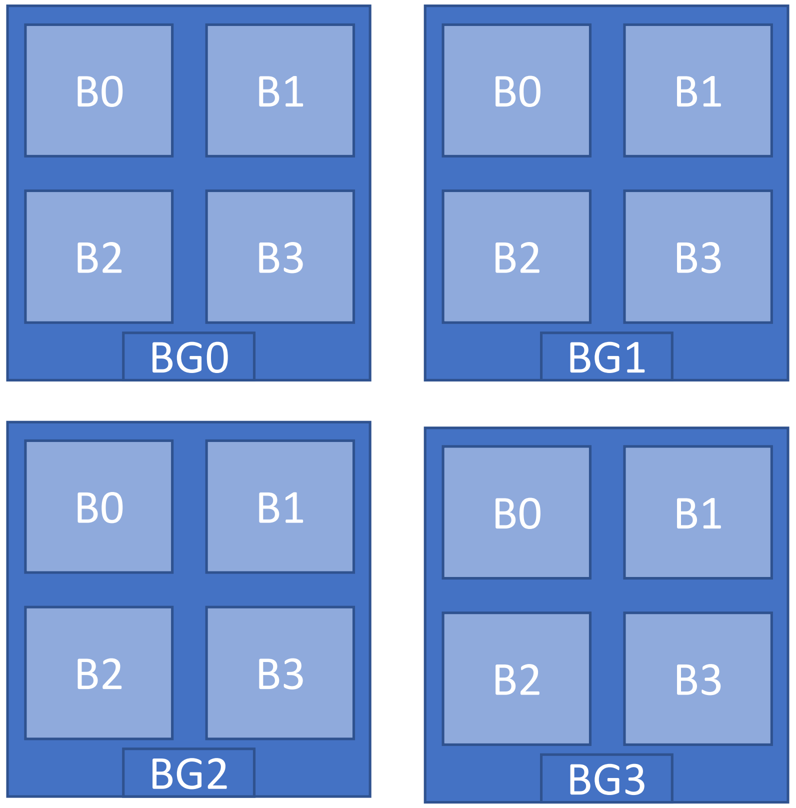
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



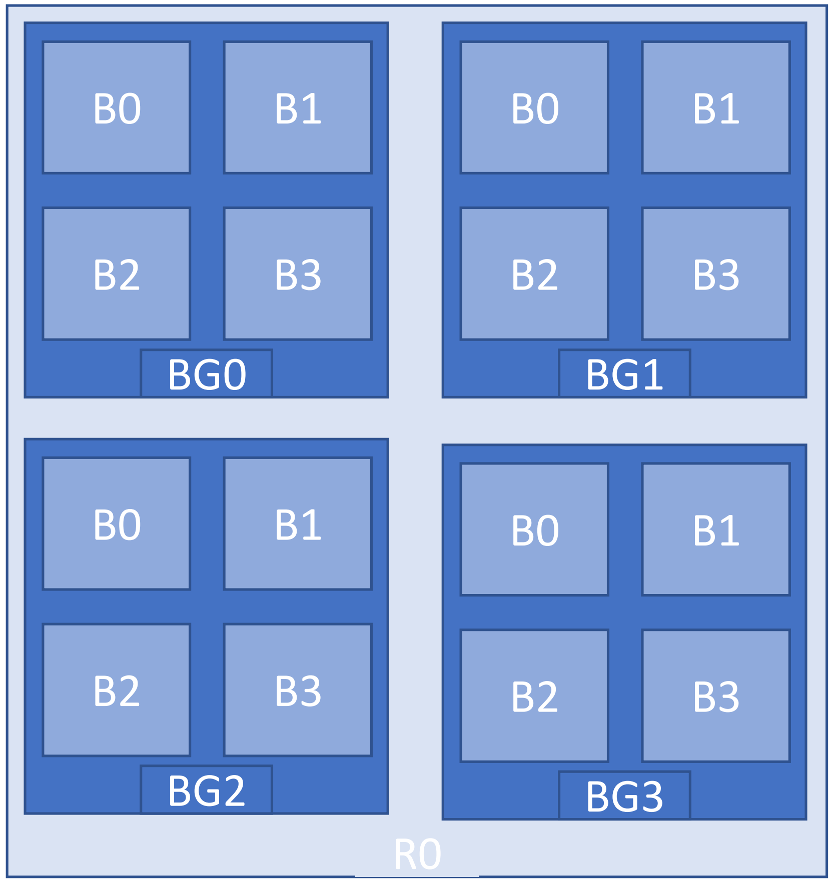
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



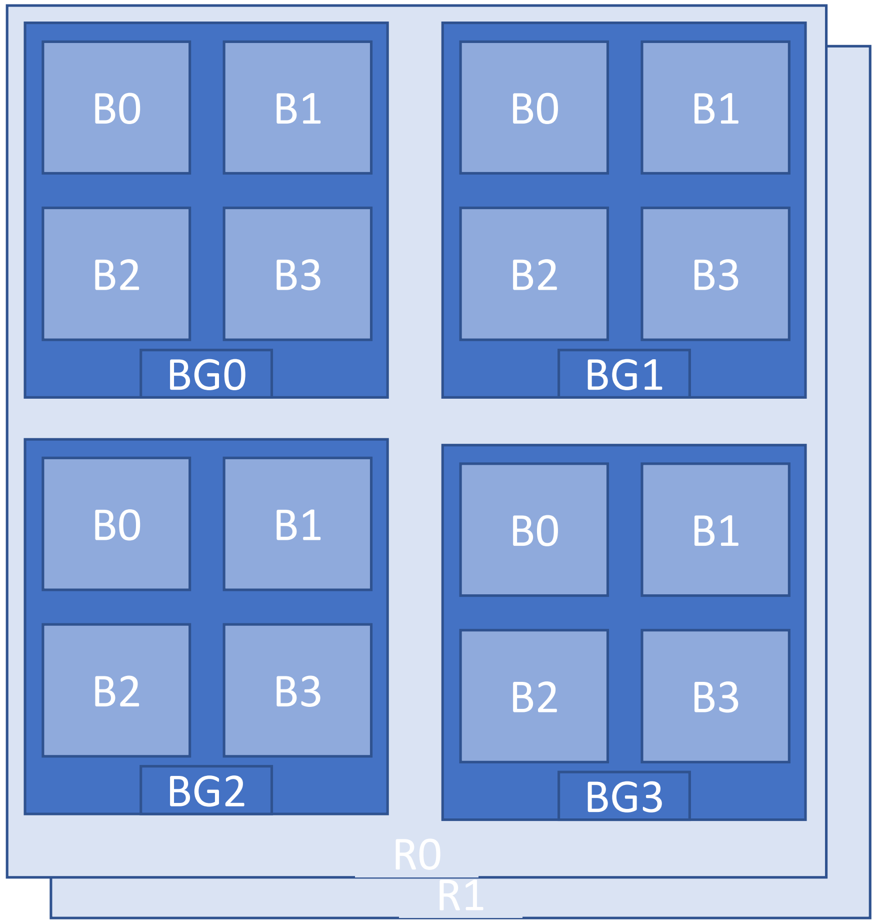
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



- Multiple bank groups for a rank
- A rank is just a CS line from the electrical domain perspective
- It is part of addressing scheme too

Memory



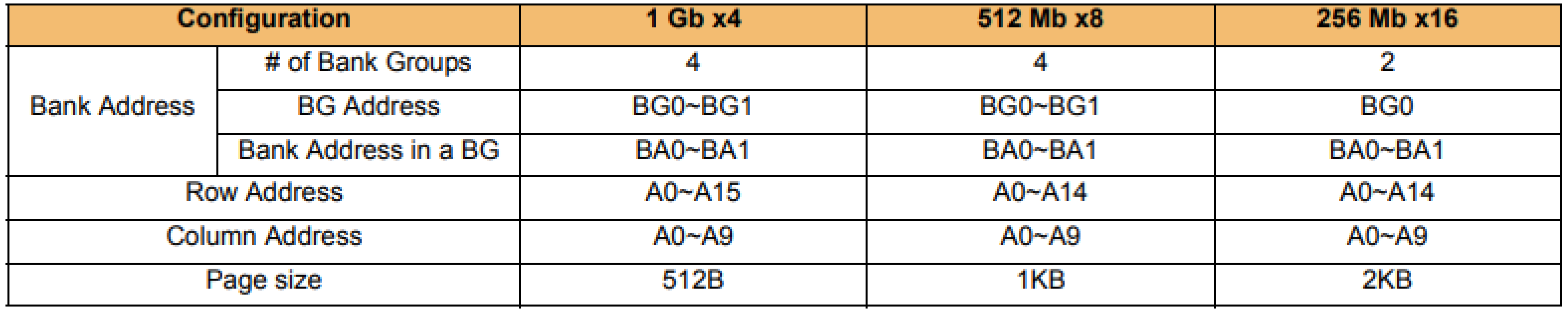
- Multiple bank groups for a rank
- A rank is just a CS line from the electrical domain perspective
- It is part of addressing scheme too
- Multiple ranks physically equal to multiple separate groups of memory components assembled on the same PCB
- MDP and vendor marketing names for multilayer components

Memory

- What about HBM?
  - Not 1024 bits, 8 x 128 instead.
  - Bandwidth will only be high when there is sufficient stream of commands inflight
- What about remote memory?
  - CXL and vendor proprietary interconnects
  - Coherency is expensive
  - Single physical address space is expensive
  - Single logical cache coherent address space is very expensive
- What about nonvolatile memory?
  - NVDIMM-* variants, vendor specifics.
  - Useful for debugging
- System address mapping design and control

Addressable memory units

- SW visible page size is (typically) 4KB
- DDR memory page (row) size depends on the actual components used
- Mapping is controllable by low level firmware
- In most practical cases at boot time only



Configuration		1 Gb x4	512 Mb x8	256 Mb x16
Bank Address	# of Bank Groups	4	4	2
	BG Address	BG0~BG1	BG0~BG1	BG0
	Bank Address in a BG	BA0~BA1	BA0~BA1	BA0~BA1
Row Address		A0~A15	A0~A14	A0~A14
Column Address		A0~A9	A0~A9	A0~A9
Page size		512B	1KB	2KB

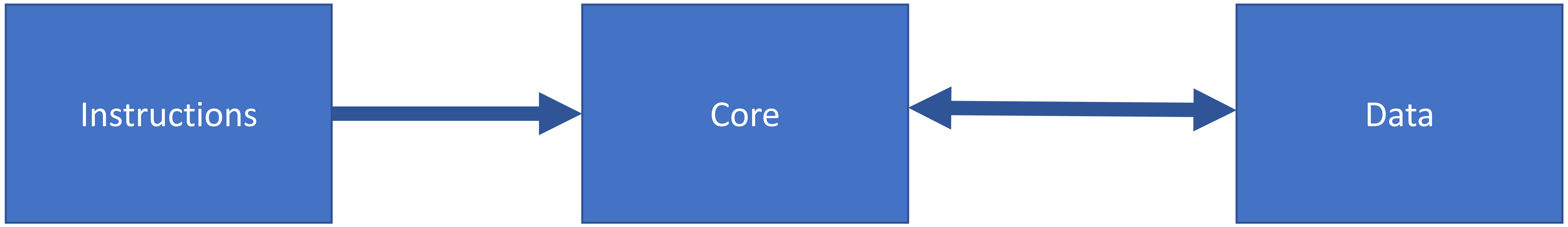
From JESD 79-4 DDR4 specification

Same capacity, different composition => different performance profile

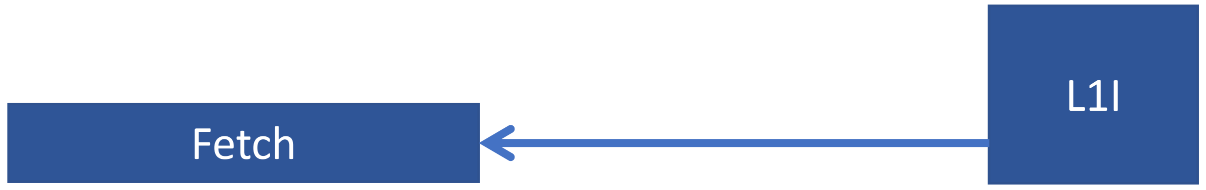
Memory

- Memory system is in the uncore
- Cores act as clients
- Remote socket cores act as clients
- Distributed cache coherence is hard and expensive
- Writing implies reading first in most of cases
- Reading has a load on cache hierarchy
- Nontemporal writes may help in some cases
- Cacheability parameters can be tuned, this is highly platform dependent though.

Processor (core)

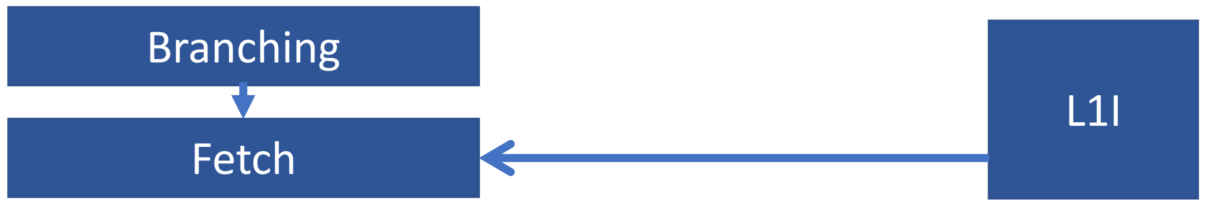


Instruction fetch



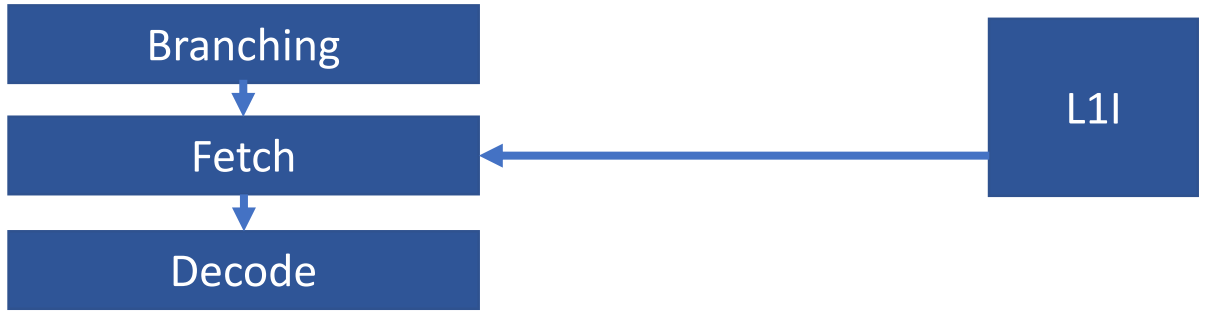
- Gets next block of (partial) instructions
- Linear fetch
- Incoming branch
- Instruction alignment
- Instruction fusing

Branch prediction



- Governs fetching of next instruction blocks
- A set of tables
- Branch history
- Branch site
- Branch target
- Repetitive patterns
- Really complex
- High cost of error

Instruction decoding

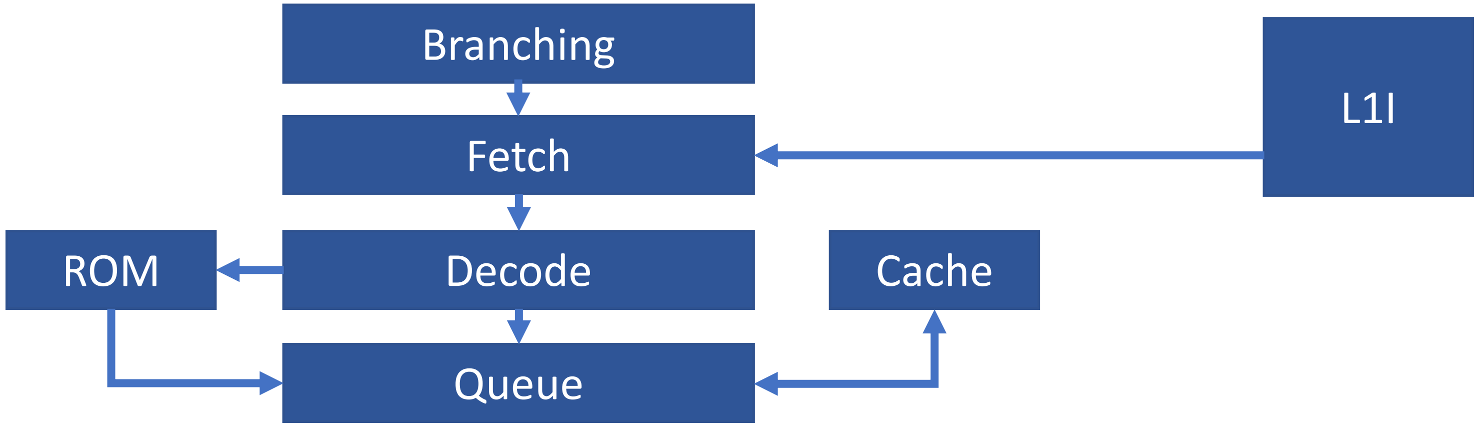


- Multiple instructions get decoded in parallel
- Even for variable length encoding
- Not that complex in HW domain
- Fusion

Instruction input

- Has to be logically sequential
- Can be physically parallel
- HW is inherently parallel
- Myth: decoding variable length instructions is:
    - Complex
    - Serial
    - Slow
- Fetch block size
- Linear fetch vs incoming branch

Instruction decoding



- Decoded operations may get cached
- There is a library of operations for complex instructions and events
- Decoded operations may get fused

Code density

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

Code density

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

```
MIPS
li      $2, 38141952
ori     $2, $2, 0x8acf
dsll    $2, $2, 17
daddiu  $2, $2, 9903
dsll    $2, $2, 18
ori     $2, $2, 0xdef0
46 02 02 3c cf 8a 42 34 78 14 02 00 af 26 42 64 b8 14 02 00 f0 de 42 34
```

Code density

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

```
MIPS
li      $2, 38141952
ori     $2, $2, 0x8acf
dsll    $2, $2, 17
daddiu  $2, $2, 9903
dsll    $2, $2, 18
ori     $2, $2, 0xdef0
46 02 02 3c cf 8a 42 34 78 14 02 00 af 26 42 64 b8 14 02 00 f0 de 42 34
```

```
RISC-V
li    a5, 305418240
addi  a5, a5, 1657
li    a0, -1698897920
slli  a5, a5, 32
addi  a0, a0, -272
add   a0, a5, a0
12 34 57 b7 67 97 87 93 9a bc e5 37 02 07 97 93 ef 05 05 13 00 a7 85 33
```

Code density

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

```
MIPS
li      $2, 38141952
ori     $2, $2, 0x8acf
dsll    $2, $2, 17
daddiu  $2, $2, 9903
dsll    $2, $2, 18
ori     $2, $2, 0xdef0
46 02 02 3c cf 8a 42 34 78 14 02 00 af 26 42 64 b8 14 02 00 f0 de 42 34
```

```
RISC-V
li    a5, 305418240
addi  a5, a5, 1657
li    a0, -1698897920
slli  a5, a5, 32
addi  a0, a0, -272
add   a0, a5, a0
12 34 57 b7 67 97 87 93 9a bc e5 37 02 07 97 93 ef 05 05 13 00 a7 85 33
```

```
SPARC
sethi  %hi(0x12345400), %g1
sethi  %hi(0x9ABCDC00), %o0
or     %g1, 0x278, %g1
or     %o0, 0x2F0, %o0
sllx   %g1, 32, %g1
add    %g1, %o0, %o0
15 8d 04 03 37 af 26 11 78 62 10 82 f0 22 12 90 20 70 28 83 08 40 00 90
```

Code density

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

```
MIPS
li      $2, 38141952
ori     $2, $2, 0x8acf
dsll    $2, $2, 17
daddiu  $2, $2, 9903
dsll    $2, $2, 18
ori     $2, $2, 0xdef0
46 02 02 3c cf 8a 42 34 78 14 02 00 af 26 42 64 b8 14 02 00 f0 de 42 34
```

```
RISC-V
li    a5, 305418240
addi  a5, a5, 1657
li    a0, -1698897920
slli  a5, a5, 32
addi  a0, a0, -272
add   a0, a5, a0
12 34 57 b7 67 97 87 93 9a bc e5 37 02 07 97 93 ef 05 05 13 00 a7 85 33
```

```
ARM
mov     x0, 0xdef0
movk    x0, 0x9abc, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
00 de 9b d2 80 57 b3 f2 00 cf ca f2 80 46 e2 f2
```

```
SPARC
sethi  %hi(0x12345400), %g1
sethi  %hi(0x9ABCDC00), %o0
or     %g1, 0x278, %g1
or     %o0, 0x2F0, %o0
sllx   %g1, 32, %g1
add    %g1, %o0, %o0
15 8d 04 03 37 af 26 11 78 62 10 82 f0 22 12 90 20 70 28 83 08 40 00 90
```

Code density - fusion

```
uint64_t v = 0x123456789abcdef0;
```

```
x86
movabs r10, 0x123456789abcdef0
49 ba f0 de bc 9a 78 56 34 12
```

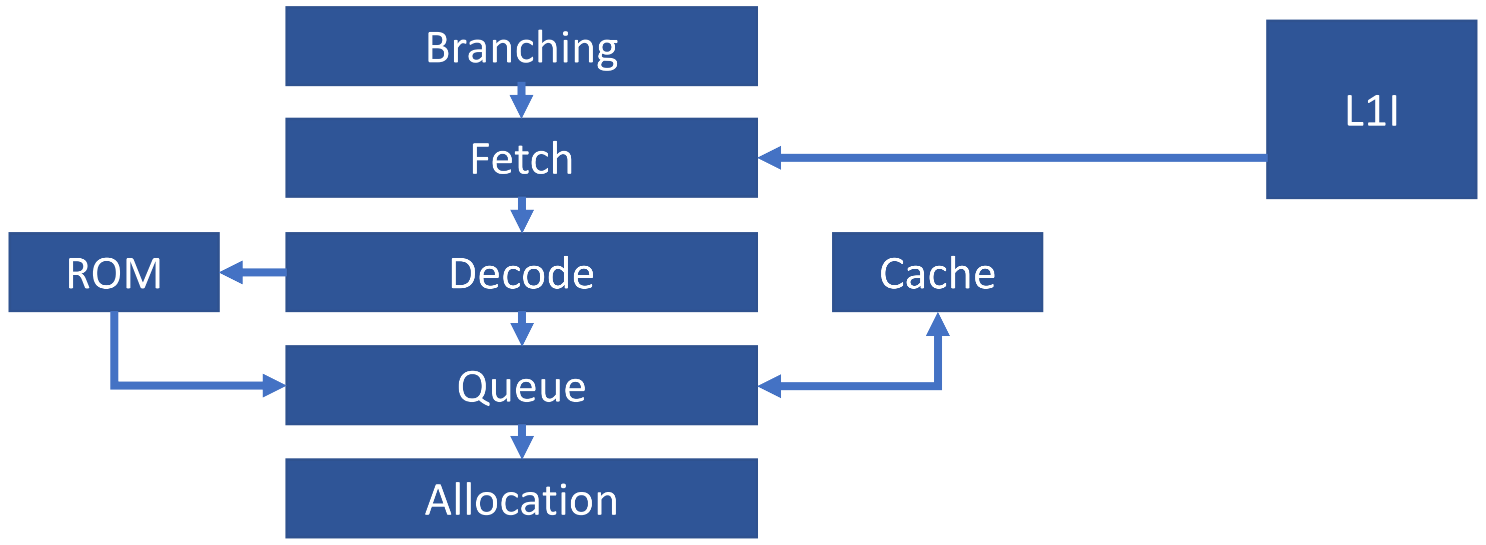
```
ARM
mov     x0, 0xdef0
movk    x0, 0x9abc, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
00 de 9b d2 80 57 b3 f2 00 cf ca f2 80 46 e2 f2
```

$\Downarrow$

```
Imaginary ARM
mov r20, 0x123456789abcdef0
```

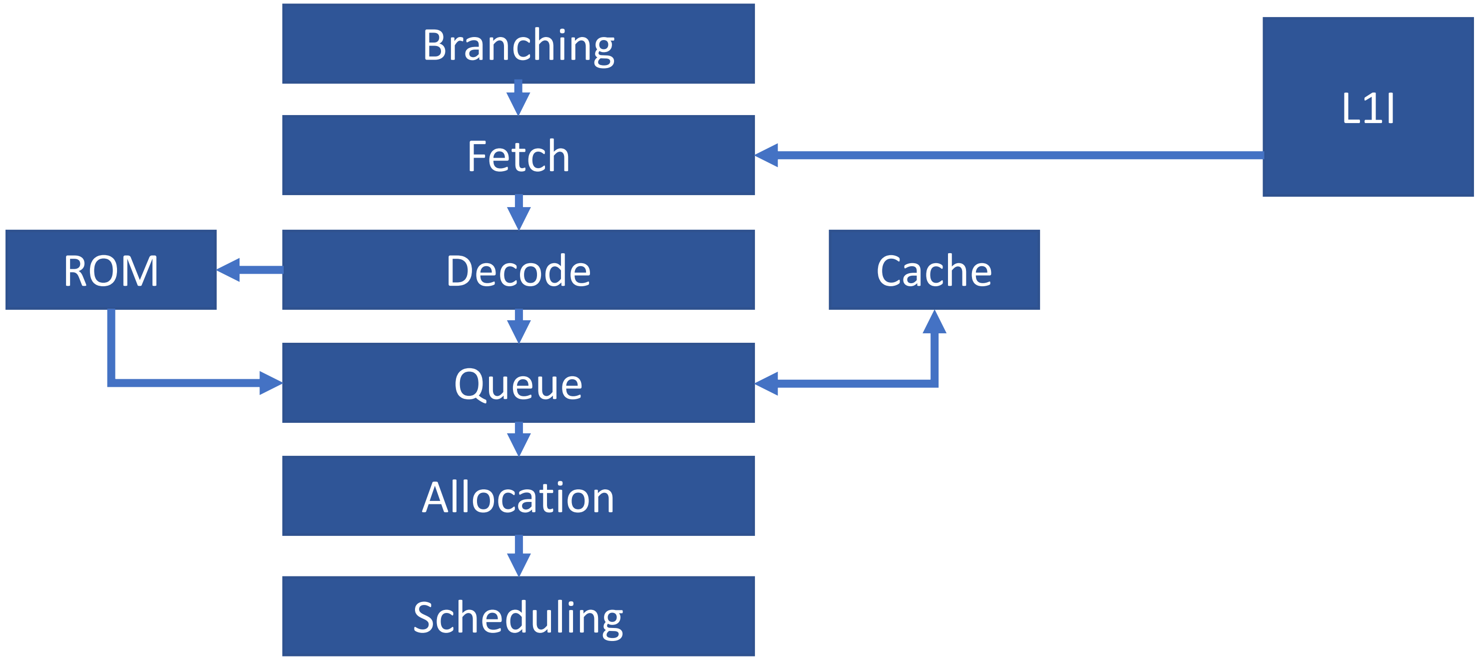
- Implementation may detect a sequence of instructions as a pattern
- And logically combine them
- Multiple instructions resulting in fewer operations
- ISA restrictions may have impact to performance

Register renaming



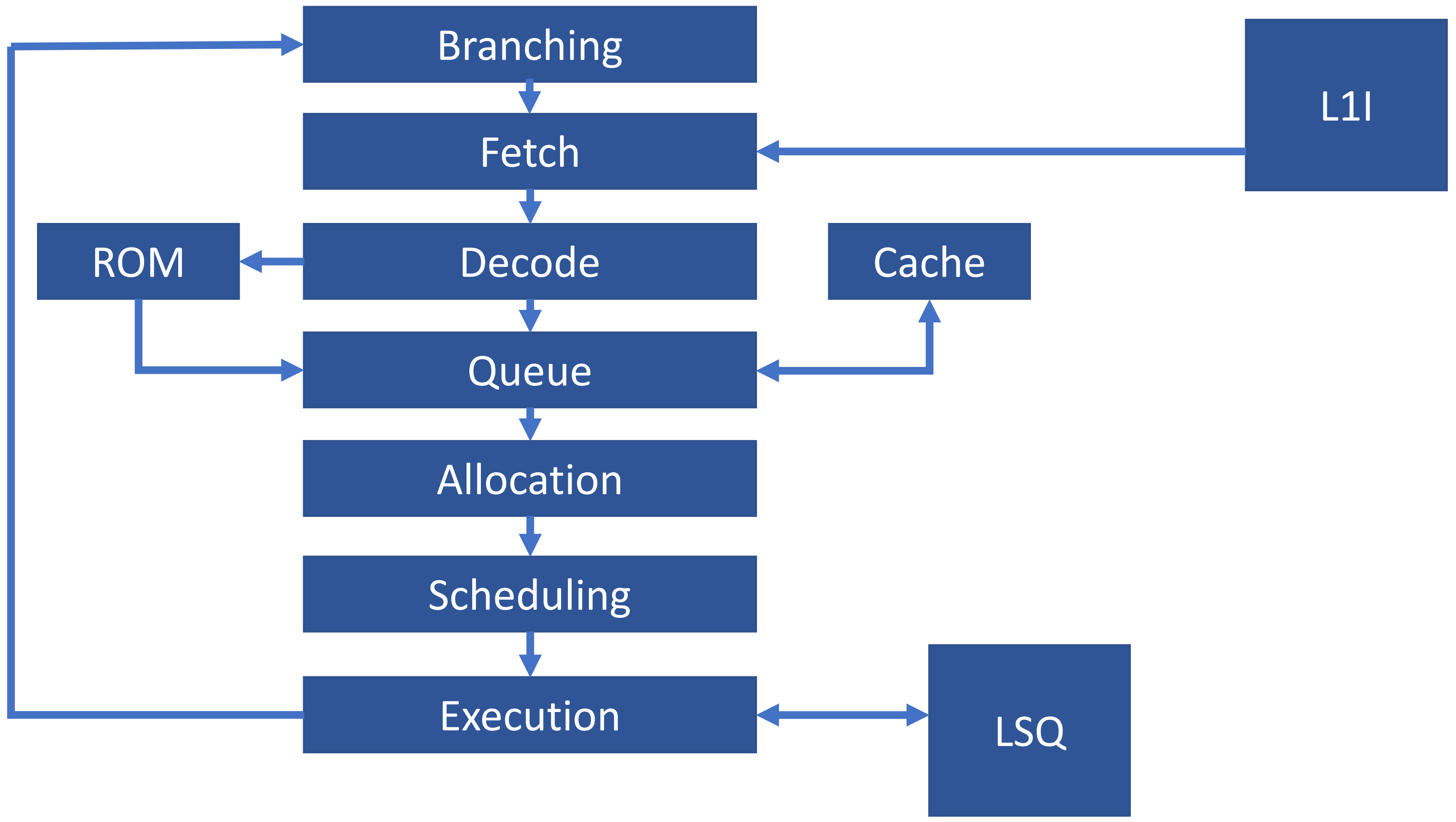
- ABI registers implement a SW contract
- They do not correspond to the actual execution
- Conversion of control flow to a variant of data flow
- Really complex
- Some operations end here
- Can a smart compiler help here?
- WLIV history

Scheduling



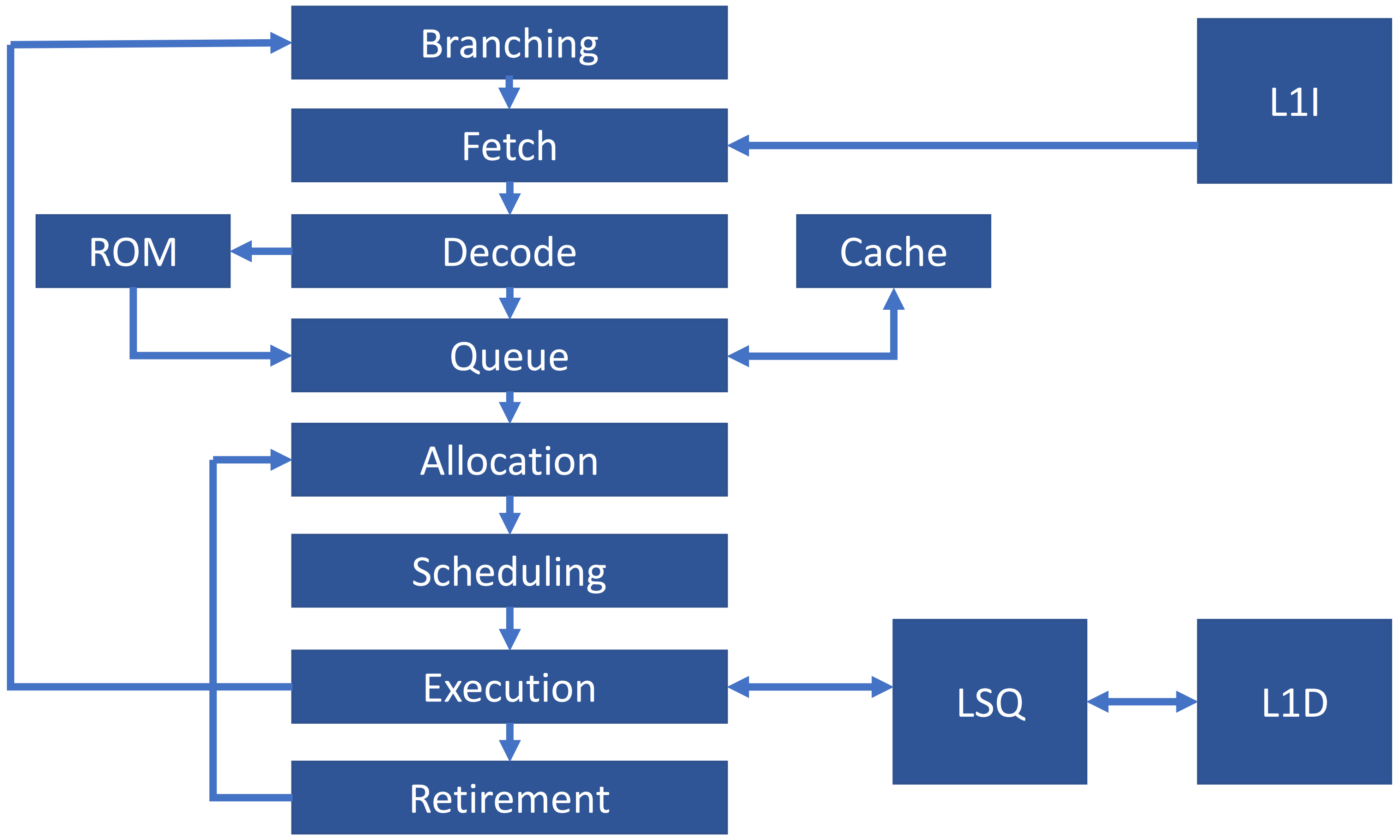
- Not all operations are equal
- Not all combinations of operations are equal

Execution



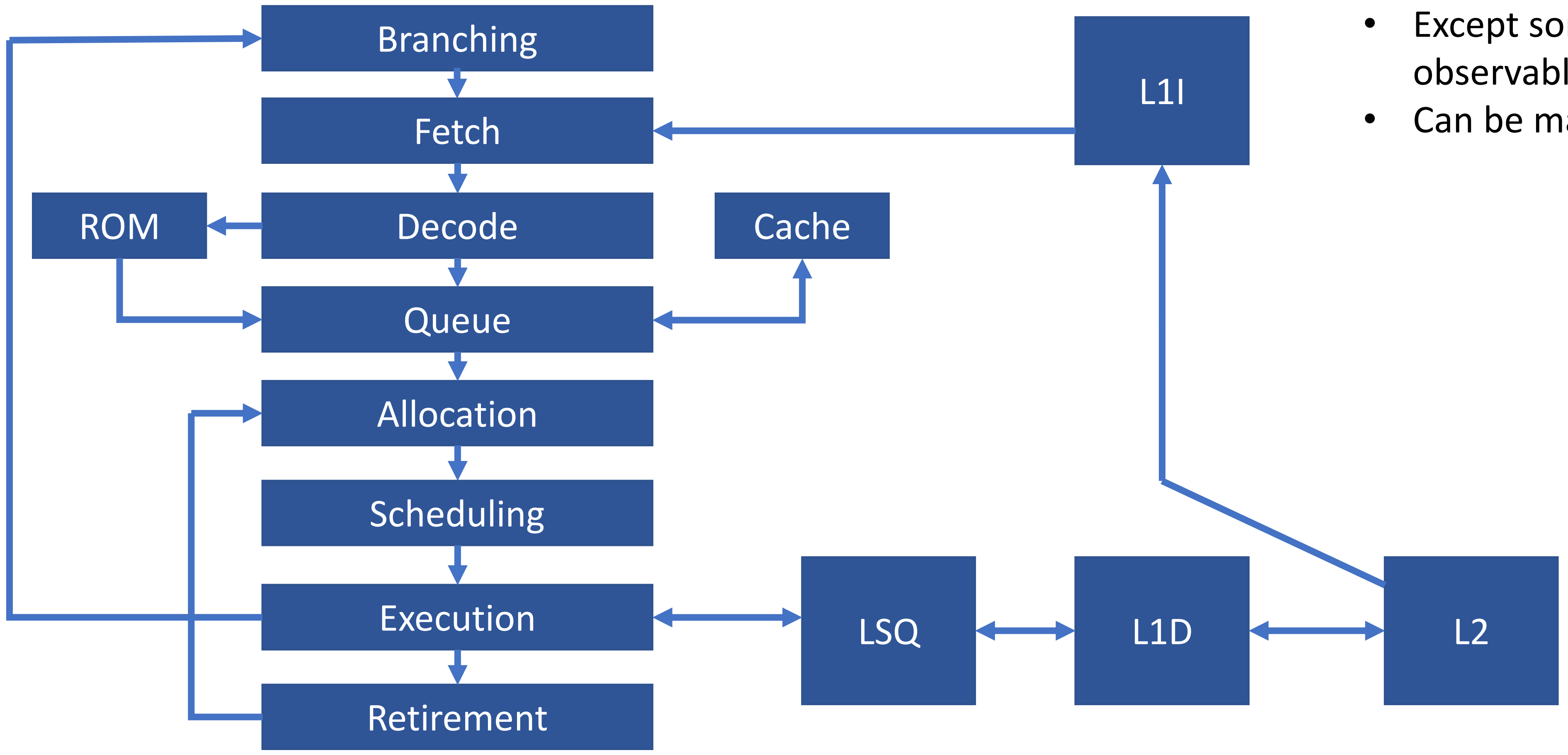
- Multiple specialized functional units
- Performs the actual (eventually) externally visible work
- Cycles
- Latency vs throughput

Retirement



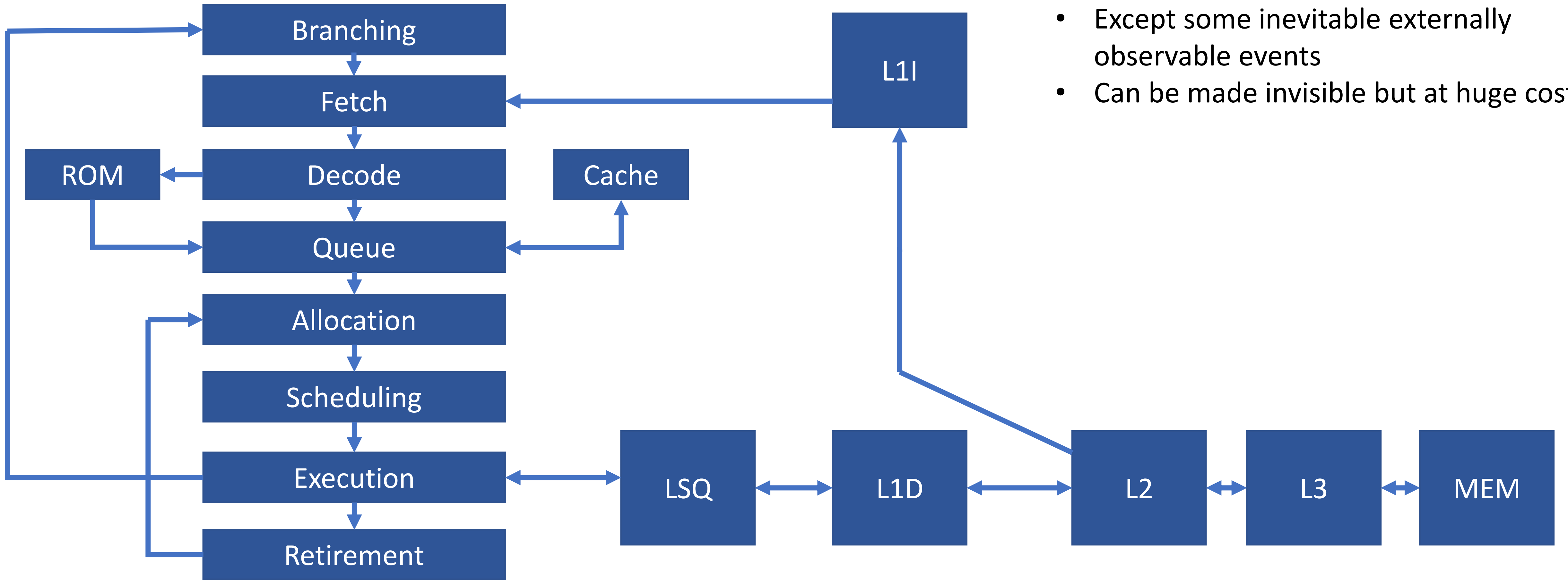
- All operations until now are speculative
- Except some inevitable externally observable events
- Can be made invisible but at huge cost

Retirement



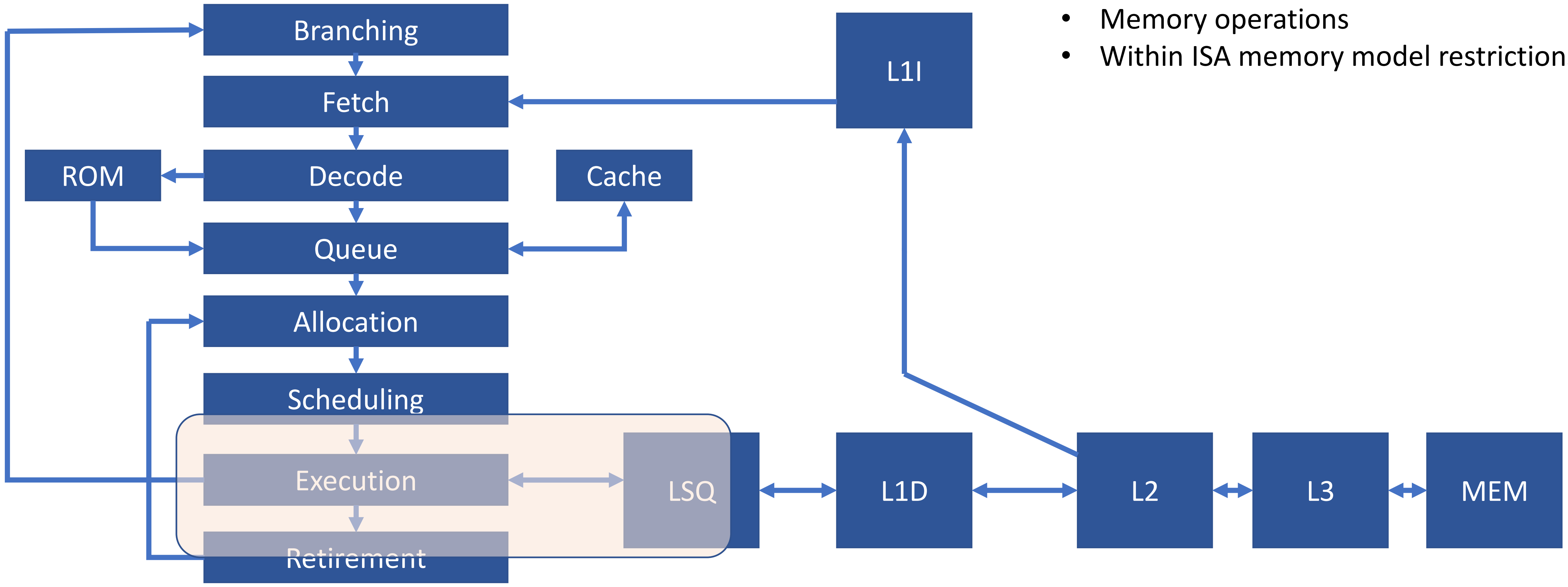
- All operations until now are speculative
- Except some inevitable externally observable events
- Can be made invisible but at huge cost

Retirement



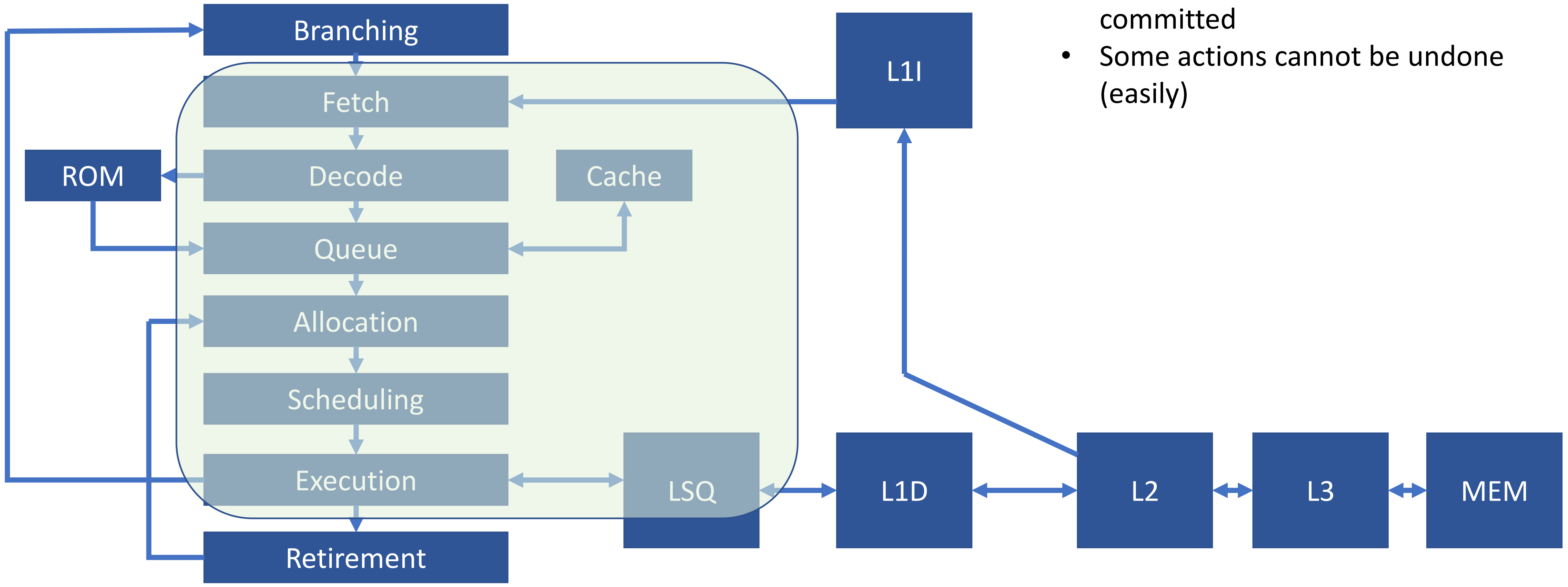
- All operations until now are speculative
- Except some inevitable externally observable events
- Can be made invisible but at huge cost

What is out of order?



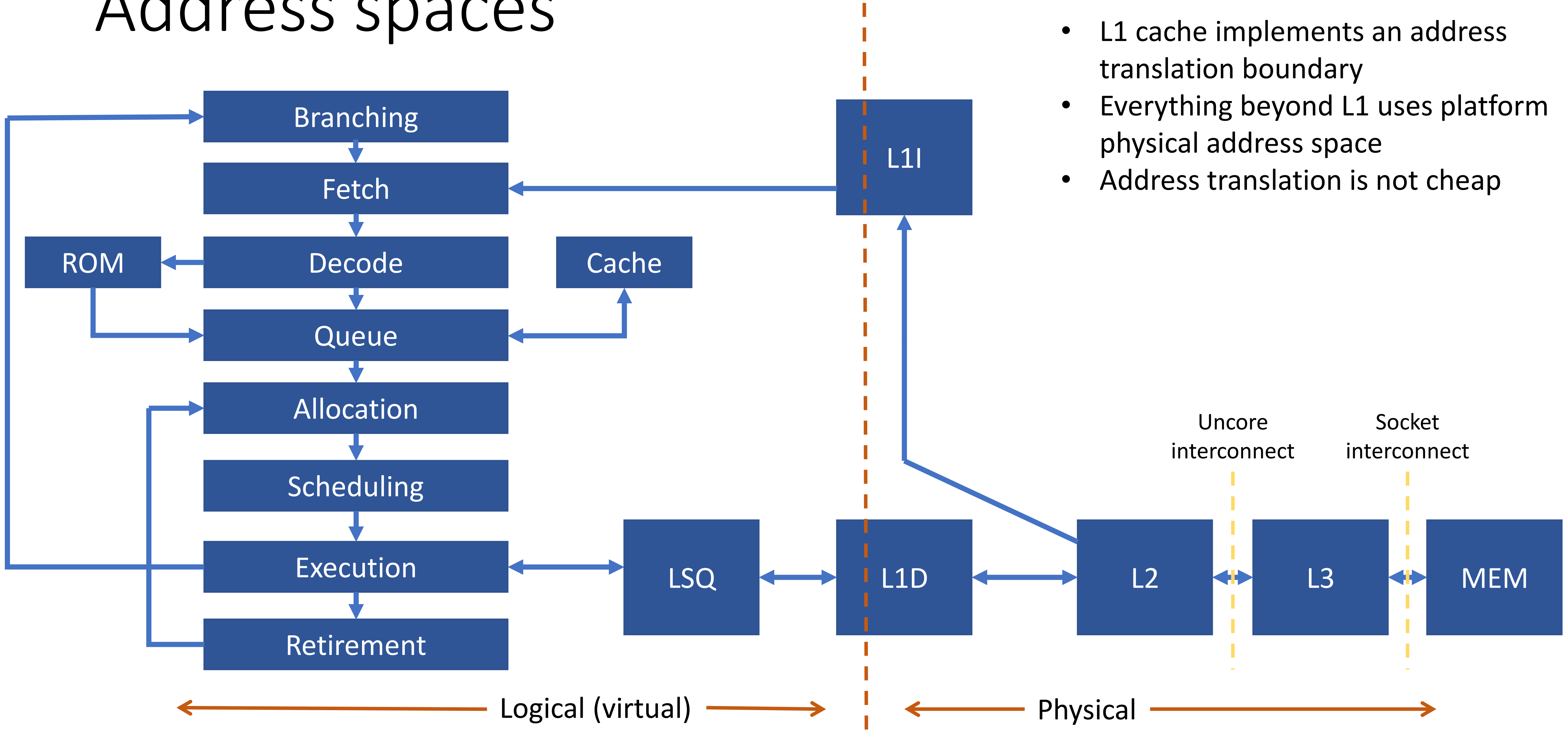
- Execution of operations
- With some exceptions
- Memory operations
- Within ISA memory model restrictions

What is speculative?



- Mostly everything
- Until the results can be safely committed
- Some actions cannot be undone (easily)

Address spaces



- L1 cache implements an address translation boundary
- Everything beyond L1 uses platform physical address space
- Address translation is not cheap

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B|75 09|89 F8|C1 E0 04|01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B|75 09|89 F8|C1 E0 04|01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
cmp rdx, 27
jne +0x9
mov eax, edi
shl eax, 4
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B|75 09|89 F8|C1 E0 04|01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne +0x9
mov eax, edi
shl eax, 4
```

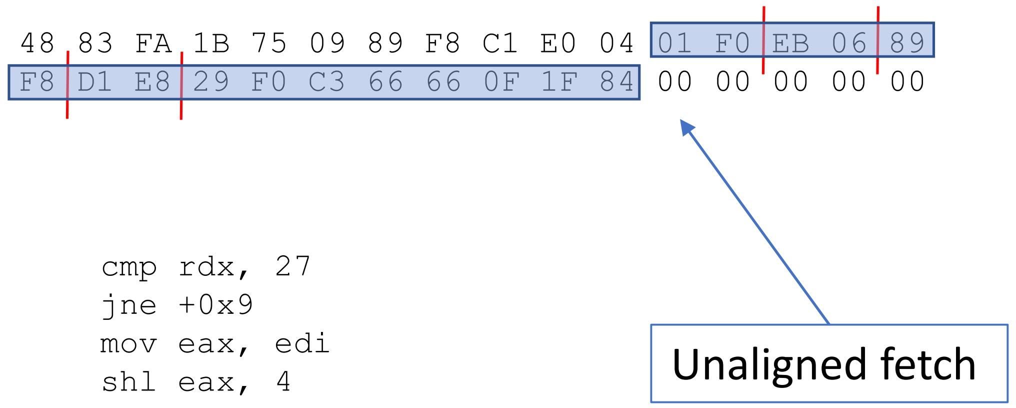
No history entry, forward direction => not taken

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 42);
```



```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne +0x9
mov eax, edi
shl eax, 4

add eax, esi
jmp +0x6
mov eax, edi
shr eax, 1
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
cmp rdx, 27
jne +0x9
mov eax, edi
shl eax, 4

add eax, esi
jmp +0x6
mov eax, edi
shr eax, 1
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
cmp rdx, 27
jne 0x0f
mov eax, edi
shl eax, 4

add eax, esi
jmp 0x15
mov eax, edi
shr eax, 1
```

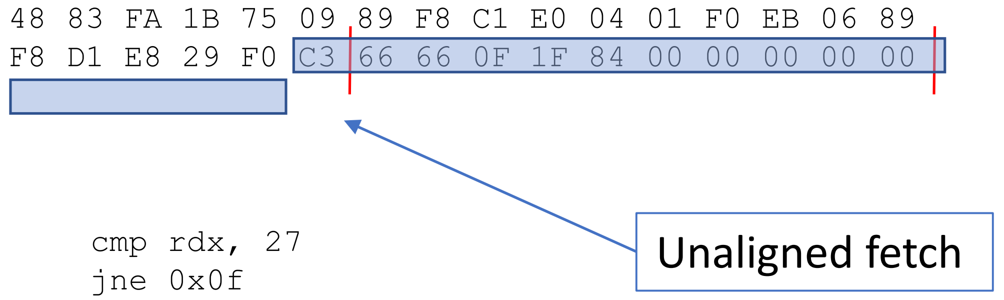
Unconditional branch
=> flush subsequent
decoded instructions

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 42);
```



```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne 0x0f
mov eax, edi
shl eax, 4

add eax, esi
jmp 0x15
mov eax, edi
shr eax, 1

ret
nop 10
...
```

Example - branching

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 42);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne 0x0f
mov eax, edi
shl eax, 4

add eax, esi
jmp 0x15
mov eax, edi
shr eax, 1

ret
nop 10
...
```

```
cmp rdx, 27
jne 0x0f
mov eax, edi
shl eax, 4
add eax, esi
ret
...
```

Flush, update branch history, refetch

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B|75 09|89 F8|C1 E0 04|01 F0 EB 06 89
F8 D1 E8 29|F0 C3|66 66|0F 1F 84|00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B | 75 09 | 89 F8 | C1 E0 04 | 01 F0 EB 06 89
F8 D1 E8 29 | F0 C3 | 66 66 | 0F 1F 84 | 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

Example – branching, 2nd attempt

```c
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```c
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B|75 09|89 F8|C1 E0 04|01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

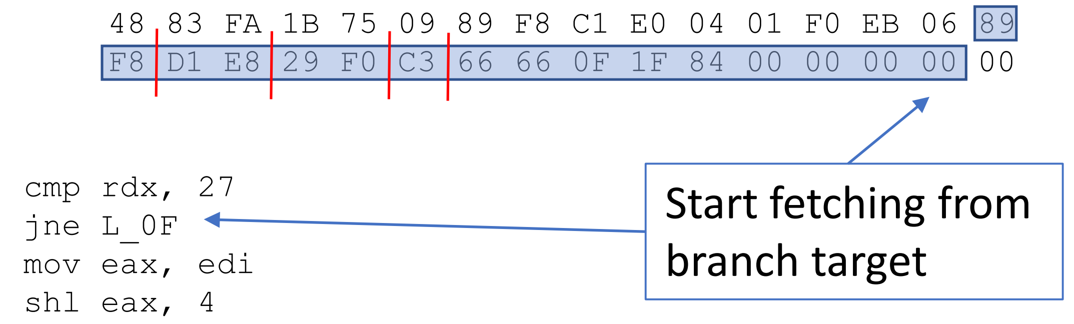
Branch history is present from last run => taken

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```



```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4

mov eax, edi
shr eax, 1
sub eax, esi
ret
```

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4

mov eax, edi
shr eax, 1
sub eax, esi
ret
```

Example – branching, 2nd attempt

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4

mov eax, edi
shr eax, 1
sub eax, esi
ret
```

Flush, update branch history, and restart

Example – branching, two-way

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

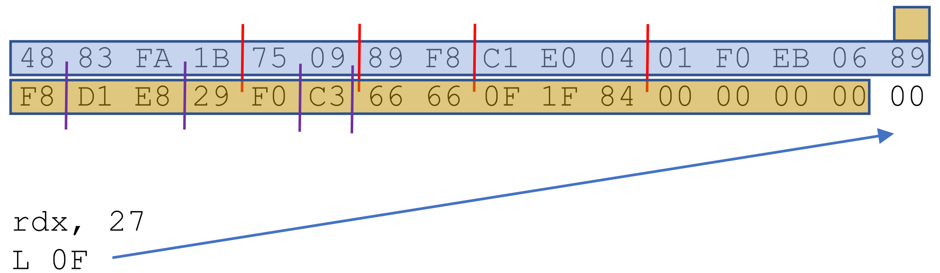
```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

Example – branching, two-way

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```

```
uint32_t res = fn1(x, y, 27);
```



```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

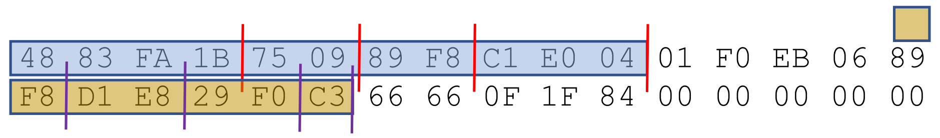
```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

Example – branching, two-way

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 27);
```

```
  cmp rdx, 27
  jne L_0F

  mov eax, edi
  shl eax, 4
  add eax, esi
  jmp L_15

L_0F:
  mov eax, edi
  shr eax, 1
  sub eax, esi

L_15:
  ret
  nop 10
```



```
48 83 FA 1B 75 09 89 F8 C1 E0 04 01 F0 EB 06 89
F8 D1 E8 29 F0 C3 66 66 0F 1F 84 00 00 00 00 00
```

```
cmp rdx, 27
jne L_0F
mov eax, edi
shl eax, 4
```

```
mov eax, edi
shr eax, 1
sub eax, esi
ret
```

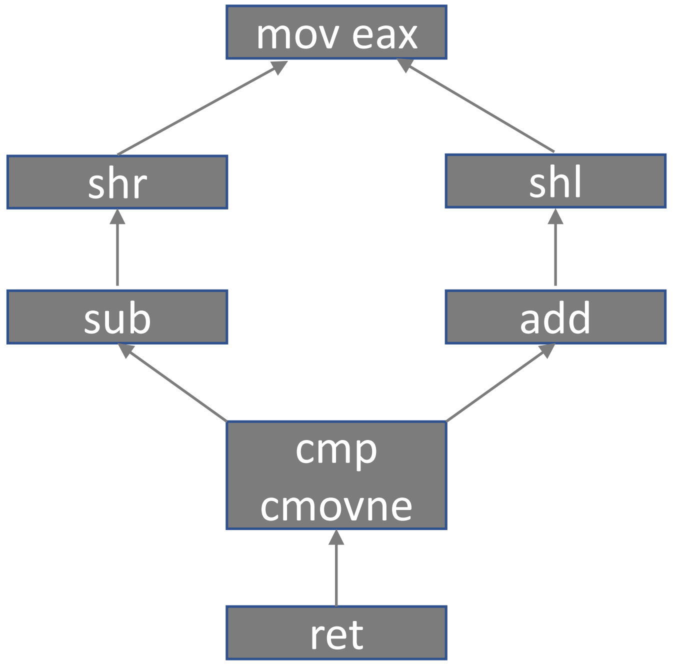
Example – branching, factual

```
uint32_t fn(uint32_t x, uint32_t y, size_t cond) {

    if (cond == 27)
        return (x << 4) + y;
    else
        return (x >> 1) - y;
}
```

```
uint32_t res = fn1(x, y, 0);
```

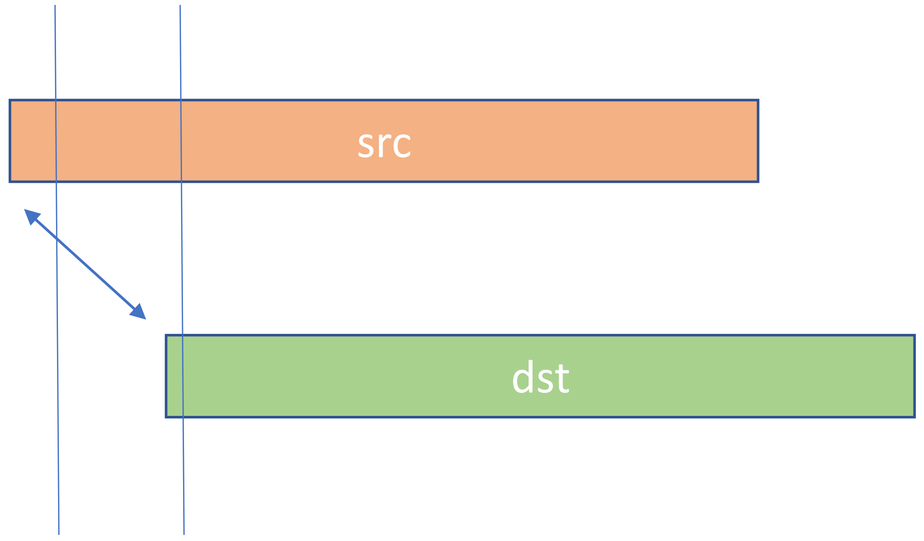
```
89 f8 c1 e0 04 01 f0 d1 ef 29 f7 48 83 fa 1b 0f
45 c7 c3 66 66 66 66 2e 0f 1f 84 00 00 00 00 00
```

```
mov eax, edi
shl eax, 4
add eax, esi
shr edi, 1
sub edi, esi
cmp rdx, 27
cmovne eax, edi
ret
nop 13
```



Example - memcpy

- Problem space
- Performance requirements
- Scalar, various vectors, specialty instructions, on-core and off-core accelerators
- Data layout: both software and hardware characteristics



- Alignment: source and destination
- Size
- Direction
- Linearity

Example – memcpy: scalar naive

```
void *memcpy_scalar(char *dst, const char *src, size_t n) {

    if(n) {
        while (n--) {
            *dst++ = *src++;
        }
    }

    return dst;

}
```

```
0000000000001270 <_Z13memcpy_scalarPcPKcm>:
    1270:       48 89 f8                mov    rax,rdi
    1273:       48 85 d2                test   rdx,rdx
    1276:       74 1b                   je     1293 <_Z13memcpy_scalarPcPKcm+0x23>
    1278:       31 c9                   xor    ecx,ecx
    127a:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]
    1280:       0f b6 3c 0e             movzx  edi,BYTE PTR [rsi+rcx*1]
    1284:       40 88 3c 08             mov    BYTE PTR [rax+rcx*1],dil
    1288:       48 ff c1                inc    rcx
    128b:       48 39 ca                cmp    rdx,rcx
    128e:       75 f0                   jne    1280 <_Z13memcpy_scalarPcPKcm+0x10>
    1290:       48 01 c8                add    rax,rcx
    1293:       c3                      ret
    1294:       66 66 66 2e 0f 1f 84    data16 data16 nop WORD PTR cs:[rax+rax*1+0x0]
    129b:       00 00 00 00 00
```

Scalar base ISA only, no vectorization

Memory operations noticeably suboptimal
Caching system will step in, with some impact

Example – memcpy: scalar++

```
void *memcpy_scalar(char *dst, const char *src, size_t n) {

    if(n) {
        while (n--) {
            *dst++ = *src++;
        }
    }

    return dst;

}
```

Autovectorized, loops not unrolled

```
0000000000001280 <_Z13memcpy_scalarPcPKcm>:
    1280:       48 85 d2                test   rdx,rdx
    1283:       74 26                   je     12ab <_Z13memcpy_scalarPcPKcm+0x2b>
    1285:       48 83 fa 20             cmp    rdx,0x20
    1289:       0f 92 c0                setb   al
    128c:       48 89 f9                mov    rcx,rdi
    128f:       48 29 f1                sub    rcx,rsi
    1292:       48 81 f9 80 00 00 00    cmp    rcx,0x80
    1299:       0f 92 c1                setb   cl
    129c:       08 c1                   or     cl,al
    129e:       74 0f                   je     12af <_Z13memcpy_scalarPcPKcm+0x2f>
    12a0:       48 89 d1                mov    rcx,rdx
    12a3:       49 89 f0                mov    r8,rsi
    12a6:       48 89 f8                mov    rax,rdi
    12a9:       eb 3f                   jmp    12ea <_Z13memcpy_scalarPcPKcm+0x6a>
    12ab:       48 89 f8                mov    rax,rdi
    12ae:       c3                      ret
    12af:       49 89 d1                mov    r9,rdx
    12b2:       49 83 e1 e0             and    r9,0xffffffffffffffe0
    12b6:       89 d1                   mov    ecx,edx
    12b8:       83 e1 1f                and    ecx,0x1f
    12bb:       4e 8d 04 0e             lea    r8,[rsi+r9*1]
    12bf:       4a 8d 04 0f             lea    rax,[rdi+r9*1]
    12c3:       45 31 d2                xor    r10d,r10d
    12c6:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
    12cd:       00 00 00
    12d0:       c4 a1 7c 10 04 16       vmovups ymm0,YMMWORD PTR [rsi+r10*1]
    12d6:       c4 a1 7c 11 04 17       vmovups YMMWORD PTR [rdi+r10*1],ymm0
    12dc:       49 83 c2 20             add    r10,0x20
    12e0:       4d 39 d1                cmp    r9,r10
    12e3:       75 eb                   jne    12d0 <_Z13memcpy_scalarPcPKcm+0x50>
    12e5:       49 39 d1                cmp    r9,rdx
    12e8:       74 1a                   je     1304 <_Z13memcpy_scalarPcPKcm+0x84>
    12ea:       31 d2                   xor    edx,edx
    12ec:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
    12f0:       41 0f b6 34 10          movzx  esi,BYTE PTR [r8+rdx*1]
    12f5:       40 88 34 10             mov    BYTE PTR [rax+rdx*1],sil
    12f9:       48 ff c2                inc    rdx
    12fc:       48 39 d1                cmp    rcx,rdx
    12ff:       75 ef                   jne    12f0 <_Z13memcpy_scalarPcPKcm+0x70>
    1301:       48 01 d0                add    rax,rdx
    1304:       c5 f8 77                vzeroupper
    1307:       c3                      ret
    1308:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
    130f:       00
```

Example – memcpy: handcrafted

```
void *memcpy_ch(void * __restrict dst_, const void * __restrict src_, size_t size) {

    char * __restrict dst = reinterpret_cast<char * __restrict>(dst_);
    const char * __restrict src = reinterpret_cast<const char * __restrict>(src_);

    void * ret = dst;

tail:

    if (size <= 16)
    {
        if (size >= 8)
        {
            __builtin_memcpy(dst + size - 8, src + size - 8, 8);
            __builtin_memcpy(dst, src, 8);
        }
        else if (size >= 4)
        {
            __builtin_memcpy(dst + size - 4, src + size - 4, 4);
            __builtin_memcpy(dst, src, 4);
        }
        else if (size >= 2)
        {
            __builtin_memcpy(dst + size - 2, src + size - 2, 2);
            __builtin_memcpy(dst, src, 2);
        }
        else if (size >= 1)
        {
            *dst = *src;
        }
    }
    else
    {
        if (size <= 128)
        {
            _mm_storeu_si128(reinterpret_cast<__m128i *>(dst + size - 16), _mm_loadu_si128(reinterpret_cast<const __m128i *>(src + size - 16)));
```

Factual Clickhouse implementation as an example

Example – memcpy: handcrafted

```
void *memcpy_ch(void * __restrict dst_, const void * __restrict src_, size_t size) {

    char * __restrict dst = reinterpret_cast<char * __restrict>(dst_);
    const char * __restrict src = reinterpret_cast<const char * __restrict>(src_);

    v

tail:

    i
    {





















    }
    e
    {
```

```
    while (size > 16)
            {
                _mm_storeu_si128(reinterpret_cast<__m128i *>(dst), _mm_loadu_si128(reinterpret_cast<const __m128i *>(src)));
                dst += 16;
                src += 16;
                size -= 16;
            }
        }
        else
        {
            size_t padding = (16 - (reinterpret_cast<size_t>(dst) & 15)) & 15;

            if (padding > 0)
            {
                __m128i head = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src));
                _mm_storeu_si128(reinterpret_cast<__m128i*>(dst), head);
                dst += padding;
                src += padding;
                size -= padding;
            }

            __m128i c0, c1, c2, c3, c4, c5, c6, c7;

            while (size >= 128)
            {
                c0 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 0);
                c1 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 1);
                c2 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 2);
                c3 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 3);
                c4 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 4);
                c5 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 5);
                c6 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 6);
                c7 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 7);
                src += 128;
```

Factual Clickhouse implementation as an example

Example – memcpy: handcrafted

```
void *memcpy_ch(void * __restrict dst_, const void * __restrict src_, size_t size) {

    char * __restrict dst = reinterpret_cast<char * __restrict>(dst_);
    const char * __restrict src = reinterpret_cast<const char * __restrict>(src_);

    v

tail:

    i
    {
```

```
    while (size > 16)
                {
                    _mm_storeu_si128(reinterpret_cast<__m128i *>(dst), _mm_loadu_si128(reinterpret_cast<const __m128i *>(src)));
```

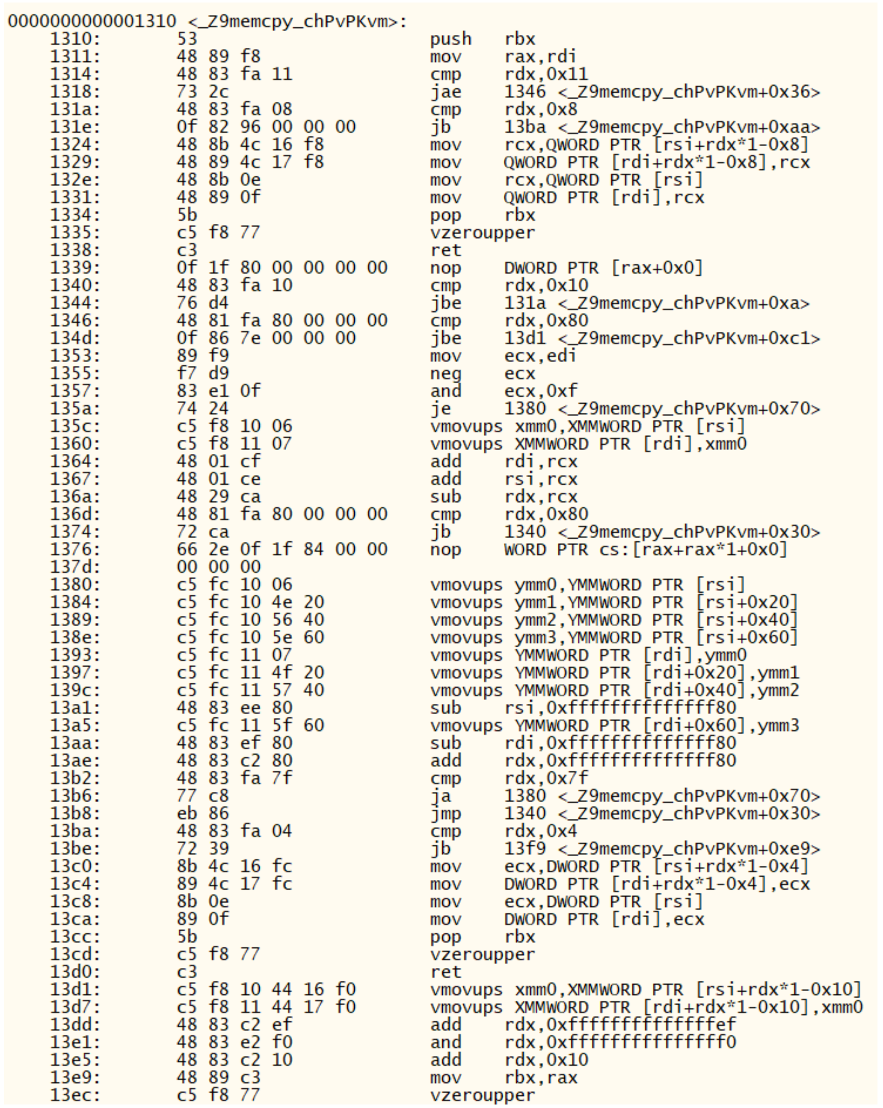
```
          }
          e
          {
```

```
                                 _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 0), c0);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 1), c1);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 2), c2);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 3), c3);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 4), c4);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 5), c5);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 6), c6);
                      _mm_store_si128((reinterpret_cast<__m128i*>(dst) + 7), c7);
                      dst += 128;

                      size -= 128;
                  }

                  goto tail;
              }
          }

          return ret;
      }
```

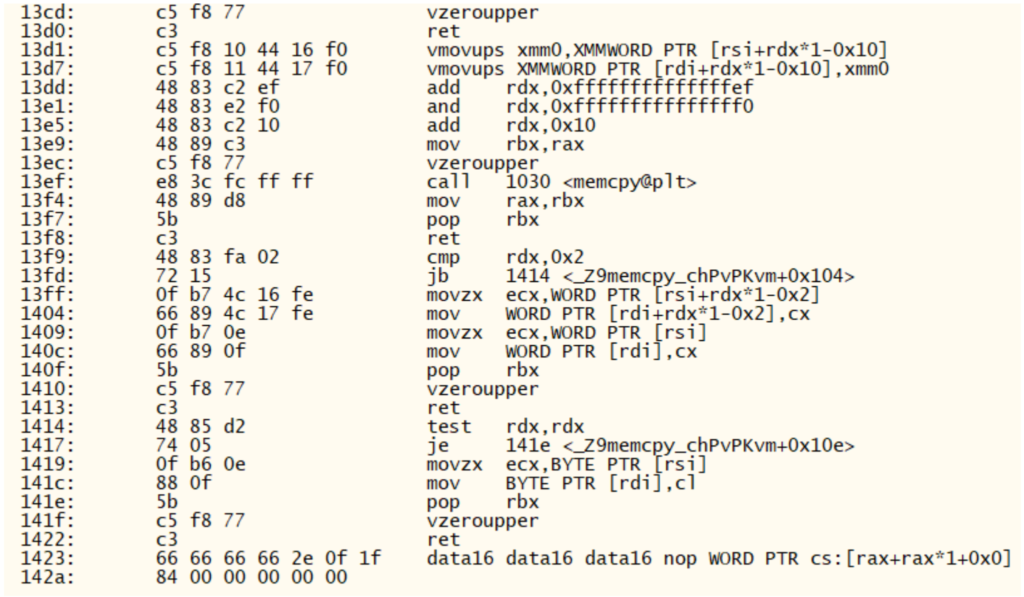
```
    }       c1 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 1);
    e       c2 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 2);
    {       c3 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 3);
            c4 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 4);
            c5 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 5);
            c6 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 6);
            c7 = _mm_loadu_si128(reinterpret_cast<const __m128i*>(src) + 7);
            src += 128;
```

Factual Clickhouse implementation as an example

Example – memcpy: handcrafted



```
0000000000001310 <_Z9memcpy_chPvPKvm>:
    1310:       53                      push   rbx
    1311:       48 89 f8                mov    rax,rdi
    1314:       48 83 fa 11             cmp    rdx,0x11
    1318:       73 2c                   jae    1346 <_Z9memcpy_chPvPKvm+0x36>
    131a:       48 83 fa 08             cmp    rdx,0x8
    131e:       0f 82 96 00 00 00       jb     13ba <_Z9memcpy_chPvPKvm+0xaa>
    1324:       48 8b 4c 16 f8          mov    rcx,QWORD PTR [rsi+rdx*1-0x8]
    1329:       48 89 4c 17 f8          mov    QWORD PTR [rdi+rdx*1-0x8],rcx
    132e:       48 8b 0e                mov    rcx,QWORD PTR [rsi]
    1331:       48 89 0f                mov    QWORD PTR [rdi],rcx
    1334:       5b                      pop    rbx
    1335:       c5 f8 77                vzeroupper
    1338:       c3                      ret
    1339:       0f 1f 80 00 00 00 00    nop    DWORD PTR [rax+0x0]
    1340:       48 83 fa 10             cmp    rdx,0x10
    1344:       76 d4                   jbe    131a <_Z9memcpy_chPvPKvm+0xa>
    1346:       48 81 fa 80 00 00 00    cmp    rdx,0x80
    134d:       0f 86 7e 00 00 00       jbe    13d1 <_Z9memcpy_chPvPKvm+0xc1>
    1353:       89 f9                   mov    ecx,edi
    1355:       f7 d9                   neg    ecx
    1357:       83 e1 0f                and    ecx,0xf
    135a:       74 24                   je     1380 <_Z9memcpy_chPvPKvm+0x70>
    135c:       c5 f8 10 06             vmovups xmm0,XMMWORD PTR [rsi]
    1360:       c5 f8 11 07             vmovups XMMWORD PTR [rdi],xmm0
    1364:       48 01 cf                add    rdi,rcx
    1367:       48 01 ce                add    rsi,rcx
    136a:       48 29 ca                sub    rdx,rcx
    136d:       48 81 fa 80 00 00 00    cmp    rdx,0x80
    1374:       72 ca                   jb     1340 <_Z9memcpy_chPvPKvm+0x30>
    1376:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
    137d:       00 00 00
    1380:       c5 fc 10 06             vmovups ymm0,YMMWORD PTR [rsi]
    1384:       c5 fc 10 4e 20          vmovups ymm1,YMMWORD PTR [rsi+0x20]
    1389:       c5 fc 10 56 40          vmovups ymm2,YMMWORD PTR [rsi+0x40]
    138e:       c5 fc 10 5e 60          vmovups ymm3,YMMWORD PTR [rsi+0x60]
    1393:       c5 fc 11 07             vmovups YMMWORD PTR [rdi],ymm0
    1397:       c5 fc 11 4f 20          vmovups YMMWORD PTR [rdi+0x20],ymm1
    139c:       c5 fc 11 57 40          vmovups YMMWORD PTR [rdi+0x40],ymm2
    13a1:       48 83 ee 80             sub    rsi,0xffffffffffffff80
    13a5:       c5 fc 11 5f 60          vmovups YMMWORD PTR [rdi+0x60],ymm3
    13aa:       48 83 ef 80             sub    rdi,0xffffffffffffff80
    13ae:       48 83 c2 80             add    rdx,0xffffffffffffff80
    13b2:       48 83 fa 7f             cmp    rdx,0x7f
    13b6:       77 c8                   ja     1380 <_Z9memcpy_chPvPKvm+0x70>
    13b8:       eb 86                   jmp    1340 <_Z9memcpy_chPvPKvm+0x30>
    13ba:       48 83 fa 04             cmp    rdx,0x4
    13be:       72 39                   jb     13f9 <_Z9memcpy_chPvPKvm+0xe9>
    13c0:       8b 4c 16 fc             mov    ecx,DWORD PTR [rsi+rdx*1-0x4]
    13c4:       89 4c 17 fc             mov    DWORD PTR [rdi+rdx*1-0x4],ecx
    13c8:       8b 0e                   mov    ecx,DWORD PTR [rsi]
    13ca:       89 0f                   mov    DWORD PTR [rdi],ecx
    13cc:       5b                      pop    rbx
    13cd:       c5 f8 77                vzeroupper
    13d0:       c3                      ret
    13d1:       c5 f8 10 44 16 f0       vmovups xmm0,XMMWORD PTR [rsi+rdx*1-0x10]
    13d7:       c5 f8 11 44 17 f0       vmovups XMMWORD PTR [rdi+rdx*1-0x10],xmm0
    13dd:       48 83 c2 ef             add    rdx,0xffffffffffffffef
    13e1:       48 83 e2 f0             and    rdx,0xfffffffffffffff0
    13e5:       48 83 c2 10             add    rdx,0x10
    13e9:       48 89 c3                mov    rbx,rax
    13ec:       c5 f8 77                vzeroupper
```



```
    13cd:       c5 f8 77                vzeroupper
    13d0:       c3                      ret
    13d1:       c5 f8 10 44 16 f0       vmovups xmm0,XMMWORD PTR [rsi+rdx*1-0x10]
    13d7:       c5 f8 11 44 17 f0       vmovups XMMWORD PTR [rdi+rdx*1-0x10],xmm0
    13dd:       48 83 c2 ef             add    rdx,0xffffffffffffffef
    13e1:       48 83 e2 f0             and    rdx,0xfffffffffffffff0
    13e5:       48 83 c2 10             add    rdx,0x10
    13e9:       48 89 c3                mov    rbx,rax
    13ec:       c5 f8 77                vzeroupper
    13ef:       e8 3c fc ff ff          call   1030 <memcpy@plt>
    13f4:       48 89 d8                mov    rax,rbx
    13f7:       5b                      pop    rbx
    13f8:       c3                      ret
    13f9:       48 83 fa 02             cmp    rdx,0x2
    13fd:       72 15                   jb     1414 <_Z9memcpy_chPvPKvm+0x104>
    13ff:       0f b7 4c 16 fe          movzx  ecx,WORD PTR [rsi+rdx*1-0x2]
    1404:       66 89 4c 17 fe          mov    WORD PTR [rdi+rdx*1-0x2],cx
    1409:       0f b7 0e                movzx  ecx,WORD PTR [rsi]
    140c:       66 89 0f                mov    WORD PTR [rdi],cx
    140f:       5b                      pop    rbx
    1410:       c5 f8 77                vzeroupper
    1413:       c3                      ret
    1414:       48 85 d2                test   rdx,rdx
    1417:       74 05                   je     141e <_Z9memcpy_chPvPKvm+0x10e>
    1419:       0f b6 0e                movzx  ecx,BYTE PTR [rsi]
    141c:       88 0f                   mov    BYTE PTR [rdi],cl
    141e:       5b                      pop    rbx
    141f:       c5 f8 77                vzeroupper
    1422:       c3                      ret
    1423:       66 66 66 66 2e 0f 1f    data16 data16 data16 nop WORD PTR cs:[rax+rax*1+0x0]
    142a:       84 00 00 00 00 00
```

Re-autovectorized, loops unrolled
Still predominantly unaligned

Factual Clickhouse implementation as an example

Example – memcpy: ERMS

```
void *memcpy_erms(void *dst, const void *src, size_t n) {

    asm volatile ("rep movsb"
                  : "=D" (dst), "=S" (src), "=c" (n)
                  : "0" (dst), "1" (src),  "2" (n)
                  : "memory");

    return dst;
}
```

```
0000000000001260 <_Z11memcpy_ermsPvPKvm>:
    1260:       48 89 d1                mov    rcx,rdx
    1263:       f3 a4                   rep movs BYTE PTR es:[rdi],BYTE PTR ds:[rsi]
    1265:       48 89 f8                mov    rax,rdi
    1268:       c3                      ret
    1269:       0f 1f 80 00 00 00 00    nop    DWORD PTR [rax+0x0]
```

- Microcoded subroutine
- Aware of platform specifics
- Has insight into internal machine state

Let the HW do the right thing

Example – memcpy: ERMS

```
void *memcpy_erms(void *dst, const void *src, size_t n) {

    asm volatile ("rep movsb"
                  : "=D" (dst), "=S" (src), "=c" (n)
                  : "0" (dst), "1" (src),  "2" (n)
                  : "memory");

    return dst;
}
```

```
0000000000001260 <_Z11memcpy_ermsPvPKvm>:
    1260:       48 89 d1                mov     rcx,rdx
    1263:       f3 a4                   rep movs BYTE PTR es:[rdi],BYTE PTR ds:[rsi]
    1265:       48 89 f8                mov     rax,rdi
    1268:       c3                      ret
    1269:       0f 1f 80 00 00 00 00    nop     DWORD PTR [rax+0x0]
```

- Microcoded subroutine
- Aware of platform specifics
- Has insight into internal machine state

Let the HW do the right thing

Will this universally be the best option?
Only representative performance testing will show

Example - hashing

```
uint32_t hash_fnv1a(const char* key, const size_t len) {

    uint32_t hash = 0x811c9dc5;

    for(size_t i = 0; i < len; ++i) {
        hash = hash ^ key[i];
        hash *= 0x1000193;;
    }

    return hash;
}
```

```
 b8 c5 9d 1c 81          mov eax, 0x811c9dc5
 48 85 f6                test rsi, rsi
 74 1a                   je L11c4
 31 c9                   xor ecx, ecx
 0f 1f 40 00             nop 4
                    L11b0:
 0f b6 14 0f             movzx edx, BYTE PTR [rdi+rcx*1]
 31 c2                   xor edx, eax
 69 c2 93 01 00 01       imul eax, edx, 0x1000193
 48 ff c1                inc rcx
 48 39 ce                cmp rsi, rcx
 75 ec                   jne L11b0
                    L11c4:
 c3                      ret
 66 66 2e 0f 1f 84 00    nop 11
 00 00 00 00
```

Example - hashing

```
uint32_t hash_fnv1a(const char* key, const size_t len) {

    uint32_t hash = 0x811c9dc5;

    for(size_t i = 0; i < len; ++i) {
        hash = hash ^ key[i];
        hash *= 0x1000193;;
    }

    return hash;
}
```

```
 b8 c5 9d 1c 81           mov eax, 0x811c9dc5
 48 85 f6                 test rsi, rsi
 74 1a                    je L11c4
 31 c9                    xor ecx, ecx
 0f 1f 40 00              nop 4
                     L11b0:
 0f b6 14 0f              movzx edx, BYTE PTR [rdi+rcx*1]
 31 c2                    xor edx, eax
 69 c2 93 01 00 01        imul eax, edx, 0x1000193
 48 ff c1                 inc rcx
 48 39 ce                 cmp rsi, rcx
 75 ec                    jne L11b0
                     L11c4:
 c3                       ret
 66 66 2e 0f 1f 84 00     nop 11
 00 00 00 00
```

Candidate for fusion

Example - hashing

```
uint32_t hash_fnv1a(const char* key, const size_t len) {

    uint32_t hash = 0x811c9dc5;

    for(size_t i = 0; i < len; ++i) {
        hash = hash ^ key[i];
        hash *= 0x1000193;;
    }

    return hash;
}
```

```
 b8 c5 9d 1c 81          mov eax, 0x811c9dc5
 48 85 f6                test rsi, rsi
 74 1a                   je L11c4
 31 c9                   xor ecx, ecx
 0f 1f 40 00             nop 4
                      L11b0:
 0f b6 14 0f             movzx edx, BYTE PTR [rdi+rcx*1]
 31 c2                   xor edx, eax
 69 c2 93 01 00 01       imul eax, edx, 0x1000193
 48 ff c1                inc rcx
 48 39 ce                cmp rsi, rcx
 75 ec                   jne L11b0
                      L11c4:
 c3                      ret
 66 66 2e 0f 1f 84 00    nop 11
 00 00 00 00
```

```
mov eax, 0x811c9dc5
test rsi, rsi
je L11c4

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

ret
```

Several iterations run ahead speculatively

Example - hashing

```
uint32_t hash_fnv1a(const char* key, const size_t len) {

    uint32_t hash = 0x811c9dc5;

    for(size_t i = 0; i < len; ++i) {
        hash = hash ^ key[i];
        hash *= 0x1000193;;
    }

    return hash;
}
```

```
b8 c5 9d 1c 81            mov eax, 0x811c9dc5
48 85 f6                  test rsi, rsi
74 1a                     je L11c4
31 c9                     xor ecx, ecx
0f 1f 40 00               nop 4
                      L11b0:
0f b6 14 0f               movzx edx, BYTE PTR [rdi+rcx*1]
31 c2                     xor edx, eax
69 c2 93 01 00 01         imul eax, edx, 0x1000193
48 ff c1                  inc rcx
48 39 ce                  cmp rsi, rcx
75 ec                     jne L11b0
                      L11c4:
c3                        ret
66 66 2e 0f 1f 84 00      nop 11
00 00 00 00
```

```
mov eax, 0x811c9dc5
test rsi, rsi
je L11c4

movzx r100b, BYTE PTR [rdi+rcx*1]
xor r100d, eax
imul eax, r100d, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx r101b, BYTE PTR [rdi+rcx*1]
xor r101d, eax
imul eax, r101d, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx r102b, BYTE PTR [rdi+rcx*1]
xor r102d, eax
imul eax, r102d, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

ret
```

Several iterations run ahead speculatively

And out of program order

Example - hashing

```
uint32_t hash_fnv1a(const char* key, const size_t len) {

    uint32_t hash = 0x811c9dc5;

    for(size_t i = 0; i < len; ++i) {
        hash = hash ^ key[i];
        hash *= 0x1000193;;
    }

    return hash;
}
```

```
 b8 c5 9d 1c 81           mov eax, 0x811c9dc5
 48 85 f6                 test rsi, rsi
 74 1a                    je L11c4
 31 c9                    xor ecx, ecx
 0f 1f 40 00              nop 4
                     L11b0:
 0f b6 14 0f              movzx edx, BYTE PTR [rdi+rcx*1]
 31 c2                    xor edx, eax
 69 c2 93 01 00 01        imul eax, edx, 0x1000193
 48 ff c1                 inc rcx
 48 39 ce                 cmp rsi, rcx
 75 ec                    jne L11b0
                     L11c4:
 c3                       ret
 66 66 2e 0f 1f 84 00     nop 11
 00 00 00 00
```

```
mov eax, 0x811c9dc5
test rsi, rsi
je L11c4

movzx r100b, BYTE PTR [rdi+rcx_100*1]
inc rcx_100
movzx r101b, BYTE PTR [rdi+rcx_101*1]   <----  Possible OOB!
inc rcx_101
movzx r102b, BYTE PTR [rdi+rcx_102*1]   <----  Possible OOB!
inc rcx_102

xor r100d, eax
imul eax, r100d, 0x1000193
cmpjne rsi, rcx_100, L11b0

xor r101d, eax
imul eax, r101d, 0x1000193
cmpjne rsi, rcx_101, L11b0

xor r102d, eax
imul eax, r102d, 0x1000193
cmpjne rsi, rcx_102, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]   <----  Possible OOB!
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

movzx edx, BYTE PTR [rdi+rcx*1]   <----  Possible OOB!
xor edx, eax
imul eax, edx, 0x1000193
inc rcx
cmpjne rsi, rcx, L11b0

ret
```

Example - hashing

```
struct kv {
    uint32_t key1;
    uint16_t key2;
    uint64_t key3;
    void *value;
} kv;

uint32_t hash_ext(struct kv *kv) {

    uint32_t fnv1a_hash {};

    fnv1a_hash += hash_fnv1a_ext((const char *)&kv->key1, sizeof(kv->key1));
    fnv1a_hash += hash_fnv1a_ext((const char *)&kv->key2, sizeof(kv->key2));
    fnv1a_hash += hash_fnv1a_ext((const char *)&kv->key3, sizeof(kv->key3));

    return fnv1a_hash;

}
```

Hash function implementation in a separate translation unit.

```
00000000000012a0 <_Z4hash_extP2kv>:
    12a0:       55                      push   rbp
    12a1:       41 56                   push   r14
    12a3:       53                      push   rbx
    12a4:       48 89 fb                mov    rbx,rdi
    12a7:       be 04 00 00 00          mov    esi,0x4
    12ac:       e8 7f fe ff ff          call   1130 <_Z14hash_fnv1a_extPKcm>
    12b1:       89 c5                   mov    ebp,eax
    12b3:       48 8d 7b 04             lea    rdi,[rbx+0x4]
    12b7:       be 02 00 00 00          mov    esi,0x2
    12bc:       e8 6f fe ff ff          call   1130 <_Z14hash_fnv1a_extPKcm>
    12c1:       41 89 c6                mov    r14d,eax
    12c4:       41 01 ee                add    r14d,ebp
    12c7:       48 83 c3 08             add    rbx,0x8
    12cb:       be 08 00 00 00          mov    esi,0x8
    12d0:       48 89 df                mov    rdi,rbx
    12d3:       e8 58 fe ff ff          call   1130 <_Z14hash_fnv1a_extPKcm>
    12d8:       44 01 f0                add    eax,r14d
    12db:       5b                      pop    rbx
    12dc:       41 5e                   pop    r14
    12de:       5d                      pop    rbp
    12df:       c3                      ret
```

```
0000000000000000 <_Z14hash_fnv1a_extPKcm>:
   0:   b8 c5 9d 1c 81          mov    eax,0x811c9dc5
   5:   48 85 f6                test   rsi,rsi
   8:   74 1a                   je     24 <_Z14hash_fnv1a_extPKcm+0x24>
   a:   31 c9                   xor    ecx,ecx
   c:   0f 1f 40 00             nop    DWORD PTR [rax+0x0]
  10:   0f be 14 0f             movsx  edx,BYTE PTR [rdi+rcx*1]
  14:   31 c2                   xor    edx,eax
  16:   69 c2 93 01 00 01       imul   eax,edx,0x1000193
  1c:   48 ff c1                inc    rcx
  1f:   48 39 ce                cmp    rsi,rcx
  22:   75 ec                   jne    10 <_Z14hash_fnv1a_extPKcm+0x10>
  24:   c3                      ret
```

Example - hashing

```
struct kv {
    uint32_t key1;
    uint16_t key2;
    uint64_t key3;
    void *value;
} kv;

uint32_t hash_int(struct kv *kv) {

    uint32_t fnv1a_hash {};

    fnv1a_hash += hash_fnv1a_int((const char *)&kv->key1, sizeof(kv->key1));
    fnv1a_hash += hash_fnv1a_int((const char *)&kv->key2, sizeof(kv->key2));
    fnv1a_hash += hash_fnv1a_int((const char *)&kv->key3, sizeof(kv->key3));

    return fnv1a_hash;

}
```

```
00000000000012a0 <_Z4hash_intP2kv>:
    12a0:       b9 c5 9d 1c 81          mov    ecx,0x811c9dc5
    12a5:       31 c0                   xor    eax,eax
    12a7:       66 0f 1f 84 00 00 00    nop    WORD PTR [rax+rax*1+0x0]
    12ae:       00 00
    12b0:       0f b6 14 07             movzx  edx,BYTE PTR [rdi+rax*1]
    12b4:       31 ca                   xor    edx,ecx
    12b6:       69 ca 93 01 00 01       imul   ecx,edx,0x1000193
    12bc:       48 ff c0                inc    rax
    12bf:       48 83 f8 04             cmp    rax,0x4
    12c3:       75 eb                   jne    12b0 <_Z4hashP2kv+0x10>
    12c5:       b8 c5 9d 1c 81          mov    eax,0x811c9dc5
    12ca:       31 d2                   xor    edx,edx
    12cc:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
    12d0:       0f b6 74 17 04          movzx  esi,BYTE PTR [rdi+rdx*1+0x4]
    12d5:       31 c6                   xor    esi,eax
    12d7:       69 c6 93 01 00 01       imul   eax,esi,0x1000193
    12dd:       48 ff c2                inc    rdx
    12e0:       48 83 fa 02             cmp    rdx,0x2
    12e4:       75 ea                   jne    12d0 <_Z4hashP2kv+0x30>
    12e6:       ba c5 9d 1c 81          mov    edx,0x811c9dc5
    12eb:       31 f6                   xor    esi,esi
    12ed:       0f 1f 00                nop    DWORD PTR [rax]
    12f0:       44 0f b6 44 37 08       movzx  r8d,BYTE PTR [rdi+rsi*1+0x8]
    12f6:       41 31 d0                xor    r8d,edx
    12f9:       41 69 d0 93 01 00 01    imul   edx,r8d,0x1000193
    1300:       48 ff c6                inc    rsi
    1303:       48 83 fe 08             cmp    rsi,0x8
    1307:       75 e7                   jne    12f0 <_Z4hashP2kv+0x50>
    1309:       01 c8                   add    eax,ecx
    130b:       01 d0                   add    eax,edx
    130d:       c3                      ret
    130e:       66 90                   xchg   ax,ax
```

Hash function implementation in the same translation unit.

Example - hashing

```
#define __packed __attribute__((packed))

struct kv {
    uint32_t key1;
    uint16_t key2;
    uint64_t key3;
    void *value;
} __packed kv;



uint32_t hash_packed(struct kv *kv) {

    uint32_t fnv1a_hash {};

    fnv1a_hash += hash_fnv1a_ref((const char *)&kv,
                    sizeof(kv->key1) +
                    sizeof(kv->key2) +
                    sizeof(kv->key3));

    return fnv1a_hash;

}
```

```
0000000000001310 <_Z11hash_packedP2kv>:
    1310:       48 89 7c 24 f8          mov    QWORD PTR [rsp-0x8],rdi
    1315:       b8 c5 9d 1c 81          mov    eax,0x811c9dc5
    131a:       31 c9                   xor    ecx,ecx
    131c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
    1320:       0f b6 54 0c f8          movzx  edx,BYTE PTR [rsp+rcx*1-0x8]
    1325:       31 c2                   xor    edx,eax
    1327:       69 c2 93 01 00 01       imul   eax,edx,0x1000193
    132d:       48 ff c1                inc    rcx
    1330:       48 83 f9 0e             cmp    rcx,0xe
    1334:       75 ea                   jne    1320 <_Z11hash_packedP2kv+0x10>
    1336:       c3                      ret
    1337:       66 0f 1f 84 00 00 00    nop    WORD PTR [rax+rax*1+0x0]
    133e:       00 00
```

Rearranged data structure

Measuring performance

- All of this is very platform specific
- Core and uncore performance counters
- Granularity, resolution, value limits
- Not everything at once
- Microarchitectural microbenchmarks are fun
- Although nowhere near being easy
- And might not matter that much in a global scope
- Fleetwide benchmarks do matter though

Universal ISA

- Comparing x86, ARM, R-V, some historical others
- RISC vs CISC debate, and VLIW as well
- 32 –> 31 -> even less registers
- L/S vs R/M
- Flags
- Operand count
- Destructive destination
- Spilling strategy
- ISA as a contract

Vectorization?

- The logic of a scalar algorithm applied to a multitude of separate sets of data.
- Performance characterization in terms of latency and throughput.
- SIMD as a specific instantiation of vectorization approach.

- Practical and Everyday => relies on the compiler. But we are not there yet. Not certain whether we will be at all.

Vectorization in 2~~0~~ minutes

In order to vectorize scalar code successfully, horizontal operations are mandatory.

Control flow dependencies need to be translated to data flow dependencies.

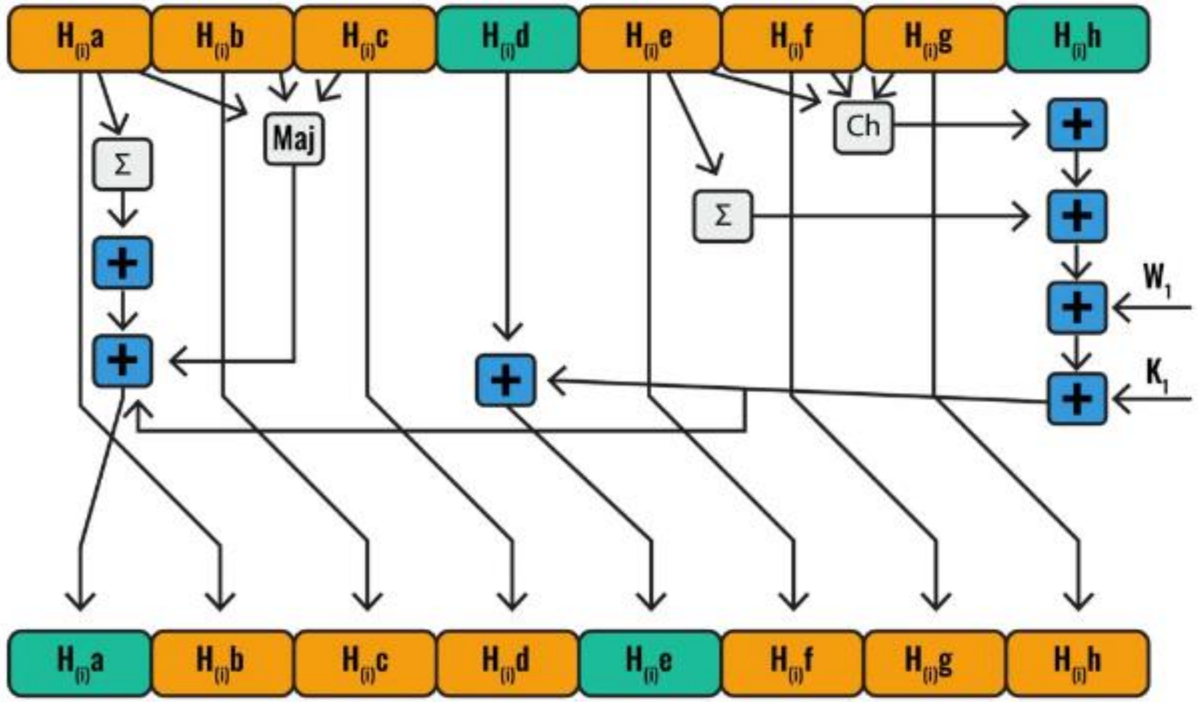
Data structure layout should not conflict with native vector width and length.

- Controllable/maskable access to memory
- Assembling a vector register from scalar values (insert/extract)
- Mixing multiple vectors into one (blend)
- Swapping blocks and elements within a vector (permute, shuffle)
- Nonlinear and indexed memory access (scatter, gather)
- Conditional execution (predicates, masking)

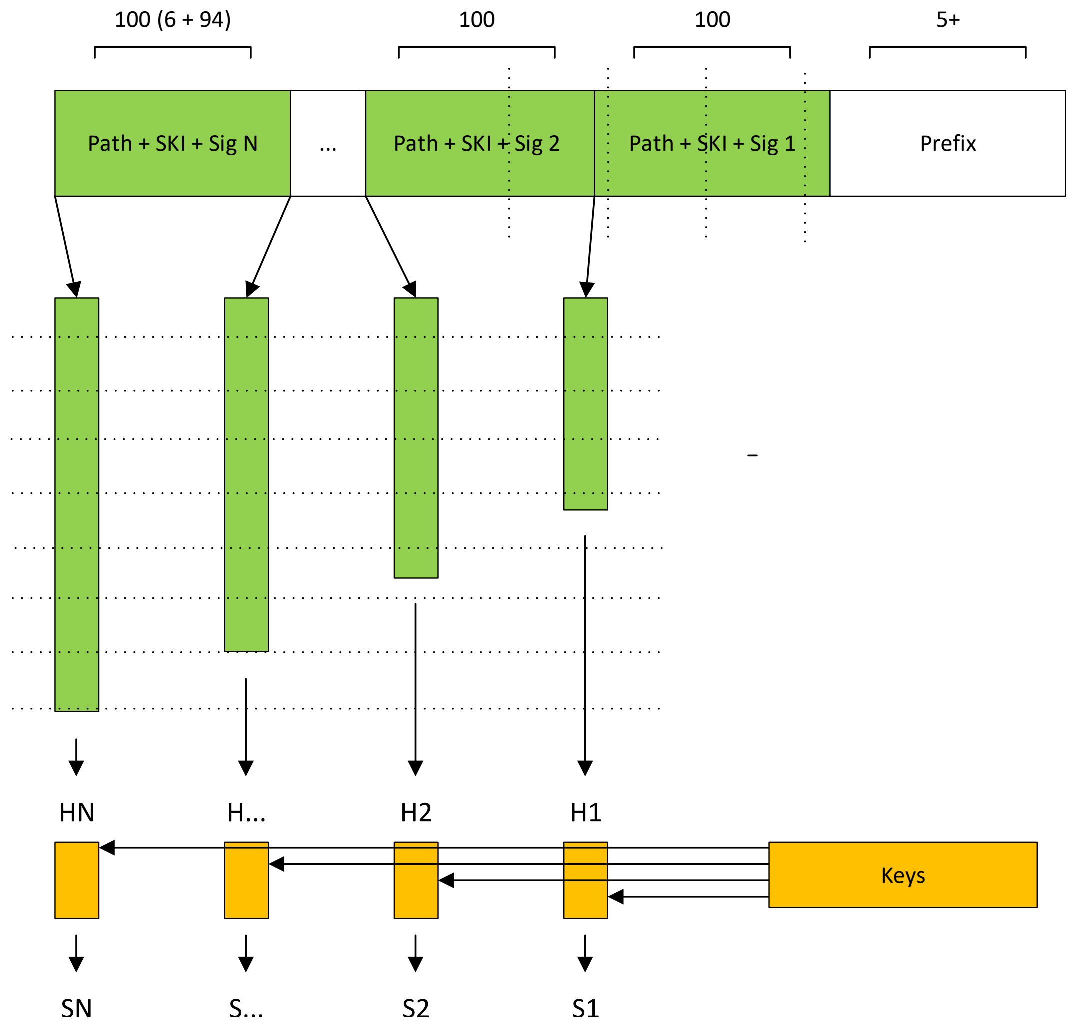
Vectorization Example – SHA2

- Block-based cryptographic hash function.
- Simple bitwise operations – many of them.
- Instruction set equivalence may not be assumed.
- Vertical vs horizontal data layout.

	Latency	Throughput
Scalar	1	1
Accelerated	4	2.2
Vectorized	0.85	14



Vectorized SHA2 and P-256 ECDSA



Linear code block operating on different data sets in parallel

Hash multiple blocks in parallel
Sign/verify multiple hashes/signatures in parallel

Vector lanes of fixed width

Gather operations place significant restrictions on data format

+20% latency results in +1500% throughput

Only if data structures allow!

Why bother?

Compiler is smart, one just needs to specify a correct command-line option, no?

```
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    for (int p = 0; p < k; p++) {
      C(i, j) += A(i, p) * B(p, j);
    }
  }
}
```

For vertical operations that is indeed not a complex task to do, and compilers perform just fine.

Why bother?

What if the level of triviality is reduced a little?

- Control flow dependencies.
- Different lane widths and parameters
- Branching
- Iterations of different length

```
missed: not vectorized: complicated access pattern.
missed: not vectorized: no vectype for stmt: sum[0] = { 0, 0, 0, 0 };
missed: couldn't vectorize loop
```

```
uint32_t CityHash32(const char *s, size_t len) {

  if (len <= 24) {
    return len <= 12 ? (len <= 4 ?
          Hash32Len0to4(s, len) :
          Hash32Len5to12(s, len)) :
          Hash32Len13to24(s, len);
  }

  h ^= a0;
  h = h * 5 + 0xe6546b64;
  h ^= a2;

  do {
    h ^= a0;
    h = Rotate32(h, 18);
    h = h * 5 + 0xe6546b64;
    f += a1;
    s += 20;
  } while (--iters != 0);

  h = h * 5 + 0xe6546b64;
  h = Rotate32(h, 17) * c1;
  h = Rotate32(h + f, 19);
  h = h * 5 + 0xe6546b64;
  h = Rotate32(h, 17) * c1;

  return h;
}
```

Historical Perspective

MMX (1996)	8 64-bit integer registers 8/16/32/64, saturation, two-operand.
SSE (1999):	8 128-bit int and fp registers, three-operand, some domain-specific accelerations.
AVX (2011):	16 256-bit fp-only registers, 128-bit lanes, 32/64-bit elements.
AVX2 (2013):	Integer AVX version, horizontal operations, gather, 8/16/32/64-bit elements.
LRBNI (2009):	32 x 512-bit data and 8 x 16-bit predicate registers, i32, f32/f64, gather and scatter, flexible bit manipulation.
IMCI (2010):	32 + 8 registers, cache management, fp focus.
AVX-512 (2015)	IMCI backport to AVX2, 32 + 8 registers, int and fp focus.

Vectorize what?

- Historically the domain of HPC
- Differential equation systems and CFD simulations are important
- TLS termination is important too
- An open question remains – what is the ratio of cores doing the former and the latter?

Discussion