
Core C++ 2026
20-21Oct. 2026 :: Tel-Aviv
Thinking Low Level, Writing High Level
Ignas Bagdonas

Thinking Low Level, Writing High Level

Thinking Low Level, Writing High Level

Exploring the internals of ABIs and platforms

Setting the context

- Performance matters
- Performance is influenced by a multitude of factors

Setting the context

- Performance matters
- Performance is influenced by a multitude of factors
- Algorithm efficiency
- The choice of data structures
- Language expressiveness
- Software and hardware interworking

Setting the context

- Performance matters
- Performance is influenced by a multitude of factors
- Algorithm efficiency
- The choice of data structures
- Language expressiveness
- **Software and hardware interworking**

Part 1 - Calling a function

How do you call a function?

Calling a function

```
function();
```

Calling a function

```
function();
int ret = function(a0, a1);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
function(const T&);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
function(const T&);
struct S0 ret = function(struct S1 s1, struct S2 *s2, struct S3 &s3);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
function(const T&);
struct S0 ret = function(struct S1 s1, struct S2 *s2, struct S3 &s3);
function(a0, ...);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
function(const T&);
struct S0 ret = function(struct S1 s1, struct S2 *s2, struct S3 &s3);
function(a0, ...);
auto ret = [](int a0, int a1);
```

Calling a function

```
function();
int ret = function(a0, a1);
char ret = function(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a1
function(char, int, float, double, uint32_t, __m256i);
std::pair<T1, T2> function(std::array<T3>);
function(void *);
function(const T&);
struct S0 ret = function(struct S1 s1, struct S2 *s2, struct S3 &s3);
function(a0, ...);
auto ret = [](int a0, int a1);
std::function<int(long, double)> function;
```

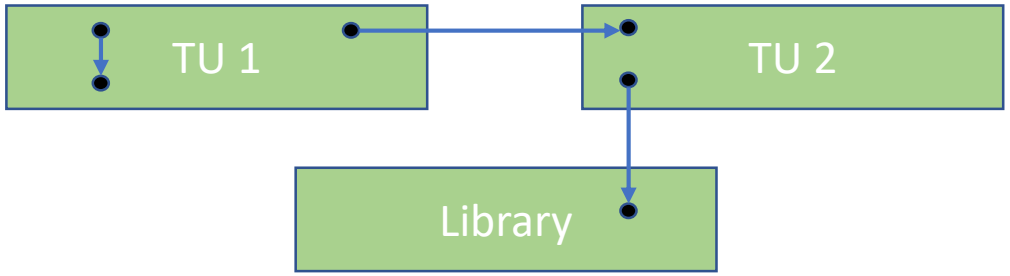
Call domains



- Same TU
- Different TU

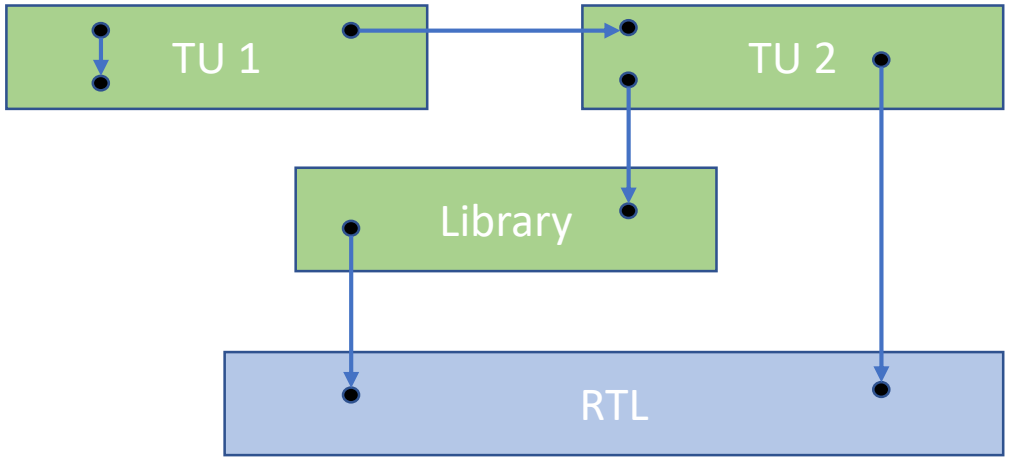
- Inlining, stack unwind

Call domains



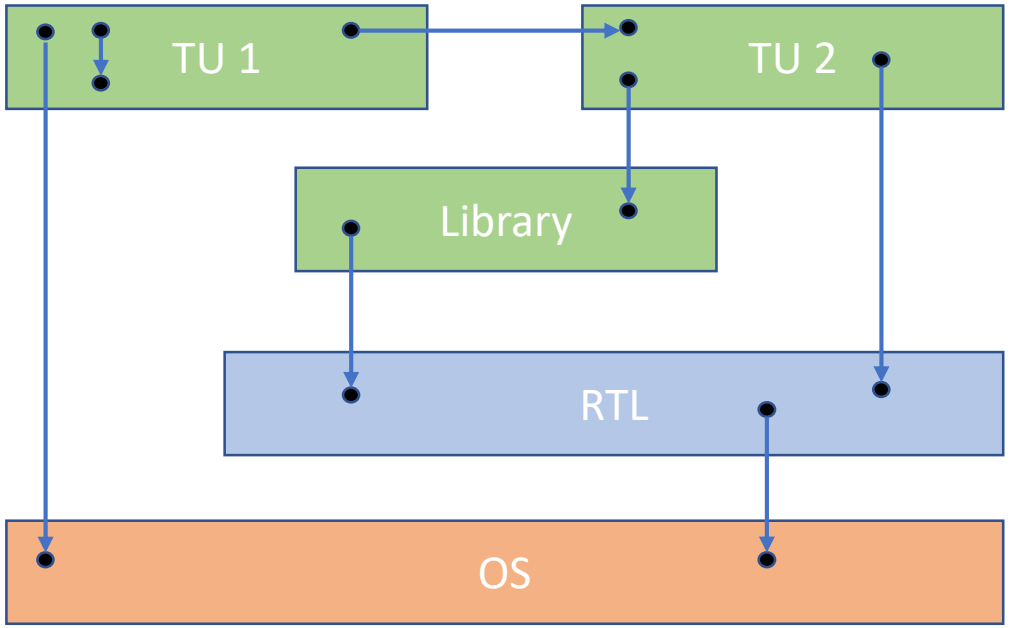
- Application dynamic library
- Static linking to libraries

Call domains



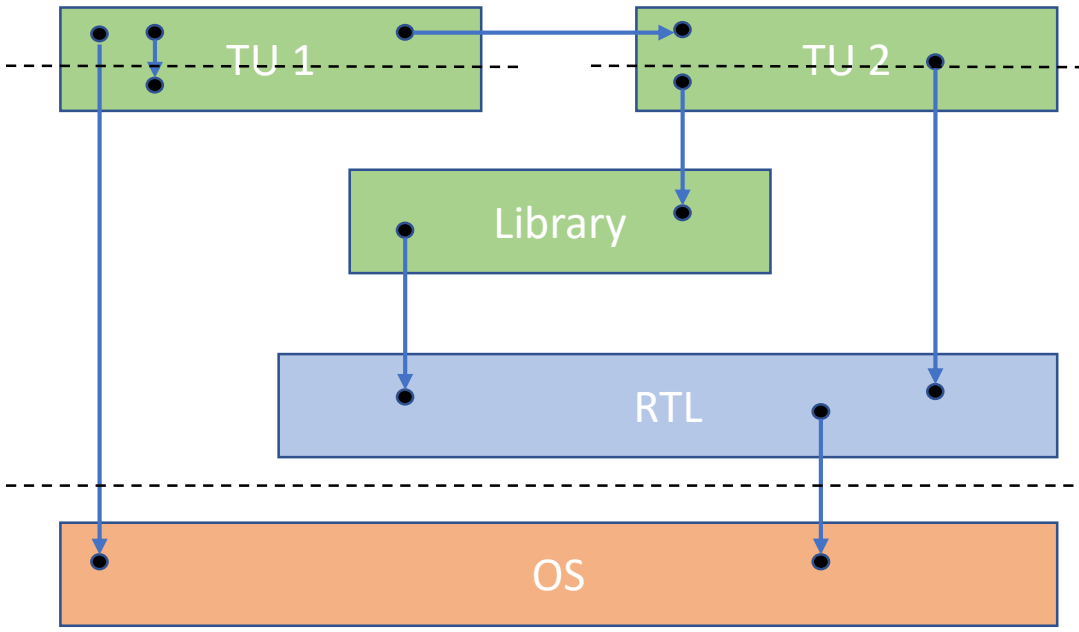
- Runtime dynamic library

Call domains



- OS service
- NULL pointer implementation

ABIs



Language ABI (gABI: C++ Itanium)

Platform ABI (psABI: SysV, Intel)

OS ABI

Hardware ABI (ISA)

ABI domains of influence

Language (= gABI)
- Storage and layout of data members
- Inheritance constructs
- Type information tracking
- Argument passing

- constness
- references, pointers
- copying, moving

Platform (= psABI)
- Registers, stack
- Values and pointers (= values)

- argument passing efficiency

Hardware (= ISA)
- Instruction sequences
- Alignment

- Performance within constraints

Call mechanics

- Registers
- Stack
- Frame linking
- Spilling

Call mechanics

```
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15
R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8, R9, R10, R11, R12, R13, R14, R15
```

Call mechanics

```
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15
R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8, R9, R10, R11, R12, R13, R14, R15
```

Call mechanics

```
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15
R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8, R9, R10, R11, R12, R13, R14, R15
R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31
```

x86 gets 32 general purpose registers via APX

x86-64-v5

Call mechanics

```
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15
R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8, R9, R10, R11, R12, R13, R14, R15
R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31
```

x86 gets 32, no, 31 general purpose registers via APX

x86-64-v5

Call mechanics

```
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15
R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8, R9, R10, R11, R12, R13, R14, R15
R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31
```

x86 gets 32, no, 31, no, 30 general purpose registers via APX

x86-64-v5

Call mechanics

- 6 forward arguments (values) in registers
- 2 return arguments in registers
- Floating point via SSE registers, 8 forward, 1 return

- Anything beyond that goes via stack

```
RDI,  RSI,  RDX,  RCX,  R8,  R9
RAX,  RDX
```

System V psABI

```
RDI,  RSI,  RDX,  R10,  R8
RAX
```

OS kernel ABI

- Interpretation of a value and a pointer

Call mechanics

- Large number of arguments might benefit of being wrapped into a structure.
- Microbenchmarks may and do lie.

```
func_direct(int&, int&, int&, int&, int&, int&, int&, int&):
        mov     eax, DWORD PTR [rsi]
        add     eax, DWORD PTR [rdi]
        add     eax, DWORD PTR [rdx]
        add     eax, DWORD PTR [rcx]
        add     eax, DWORD PTR [r8]
        add     eax, DWORD PTR [r9]
        mov     rdx, QWORD PTR [rsp+8]
        add     eax, DWORD PTR [rdx]
        mov     rdx, QWORD PTR [rsp+16]
        add     eax, DWORD PTR [rdx]
        ret
```

```
struct S1 {
    int a1;
    int a2;
    int a3;
    int a4;
    int a5;
    int a6;
    int a7;
    int a8;
};
```

```
func_ref(S1&):
        mov     eax, DWORD PTR [rdi]
        add     eax, DWORD PTR [rdi+4]
        add     eax, DWORD PTR [rdi+8]
        add     eax, DWORD PTR [rdi+12]
        add     eax, DWORD PTR [rdi+16]
        add     eax, DWORD PTR [rdi+20]
        add     eax, DWORD PTR [rdi+24]
        add     eax, DWORD PTR [rdi+28]
        ret
```

```
func_ptr(S1*):
        mov     eax, DWORD PTR [rdi]
        add     eax, DWORD PTR [rdi+4]
        add     eax, DWORD PTR [rdi+8]
        add     eax, DWORD PTR [rdi+12]
        add     eax, DWORD PTR [rdi+16]
        add     eax, DWORD PTR [rdi+20]
        add     eax, DWORD PTR [rdi+24]
        add     eax, DWORD PTR [rdi+28]
        ret
```

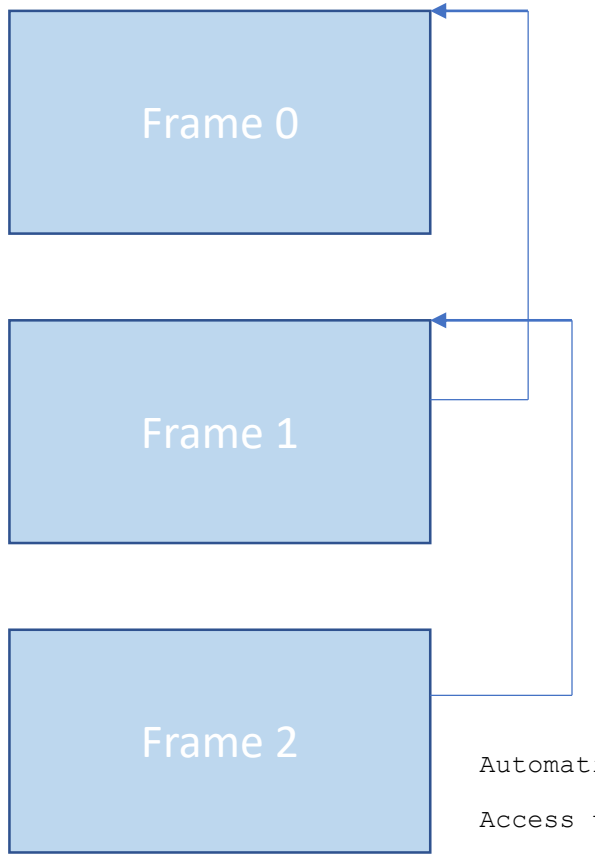
Stack

- A region of address space
- Backed by underlying memory hierarchy
- Treated as data
- Alignment required
- Can be abused
- Microarchitectural optimizations

Instruction density
Fusion vs explicit intent

```
push rax; push rdx
mov [rsp], rax; sub rsp, 8;
mov [rsp], rdx, sub rsp, 8
```

```
push2 eax, edx                pop2 edx, eax
```




```
Automatic storage
Access to incoming arguments
Access to return arguments
Spill region
Link to previous frame
```

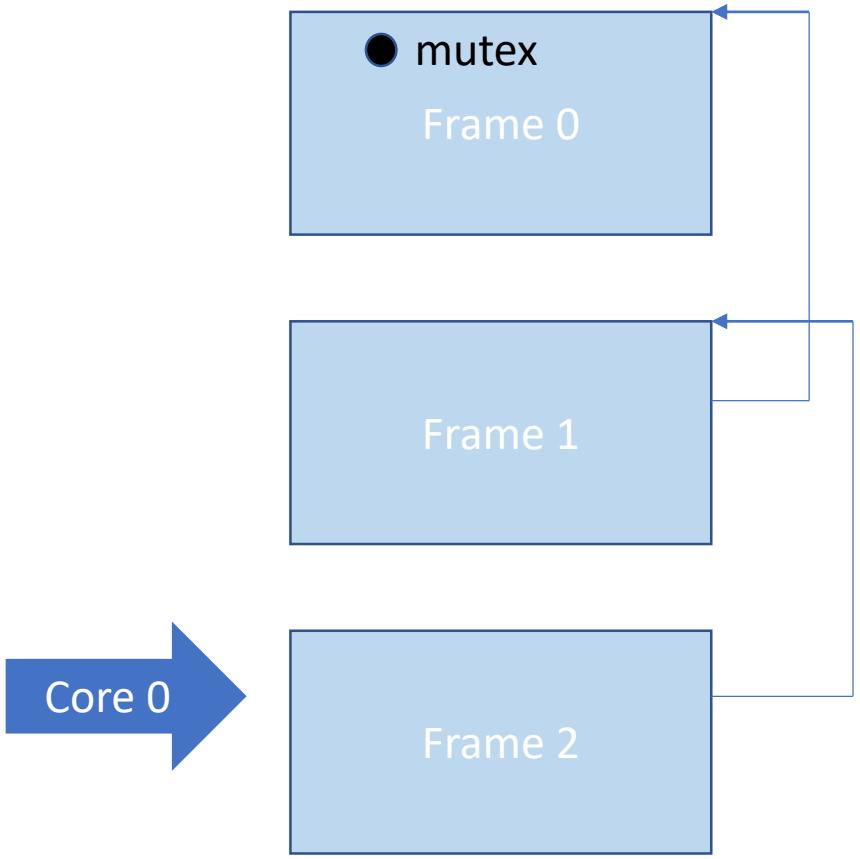
Detour – transactional memory

- Working data set is in cache, per-core
- Isolate some cache lines away
- Define a notion of transaction

- Start a transaction
- Do the work as if nothing has changed
- Commit the transaction – by bringing isolated part back into coherency domain
- If there is a conflict – rollback, otherwise you have avoided locking

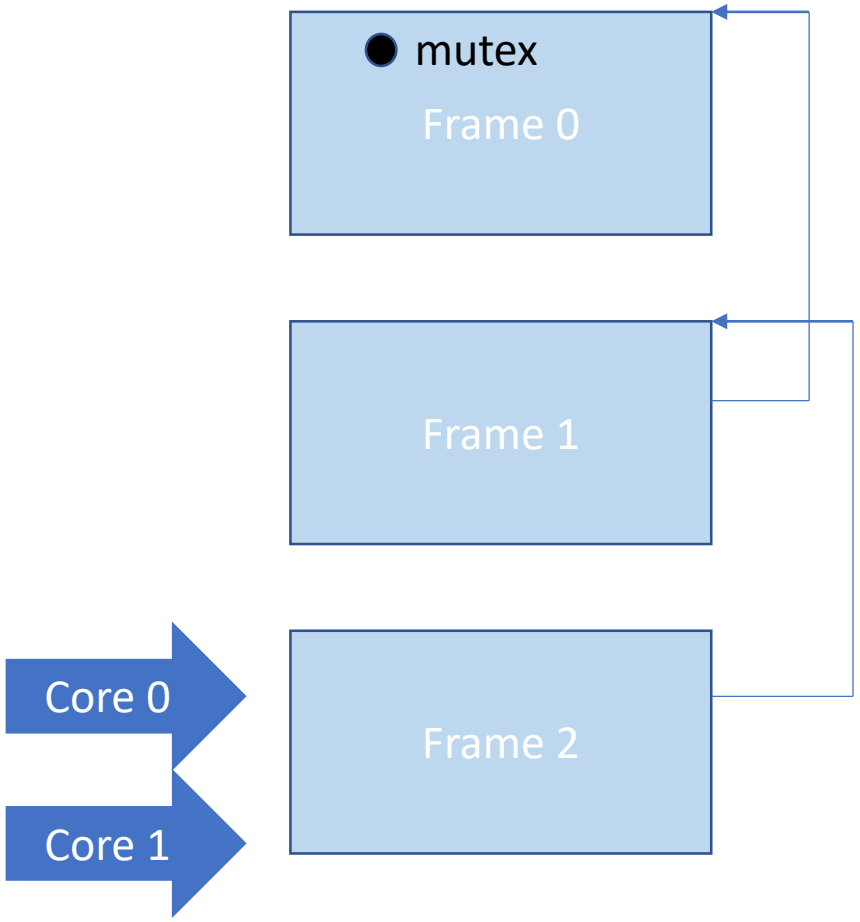
Abusing the stack

- Mutex with an auto storage duration a few frames above
- Stack protocol not violated



Abusing the stack

- Another thread tries to deal with mutex
- Thread == core from the perspective of coherency system
- Flush the stack to memory hierarchy

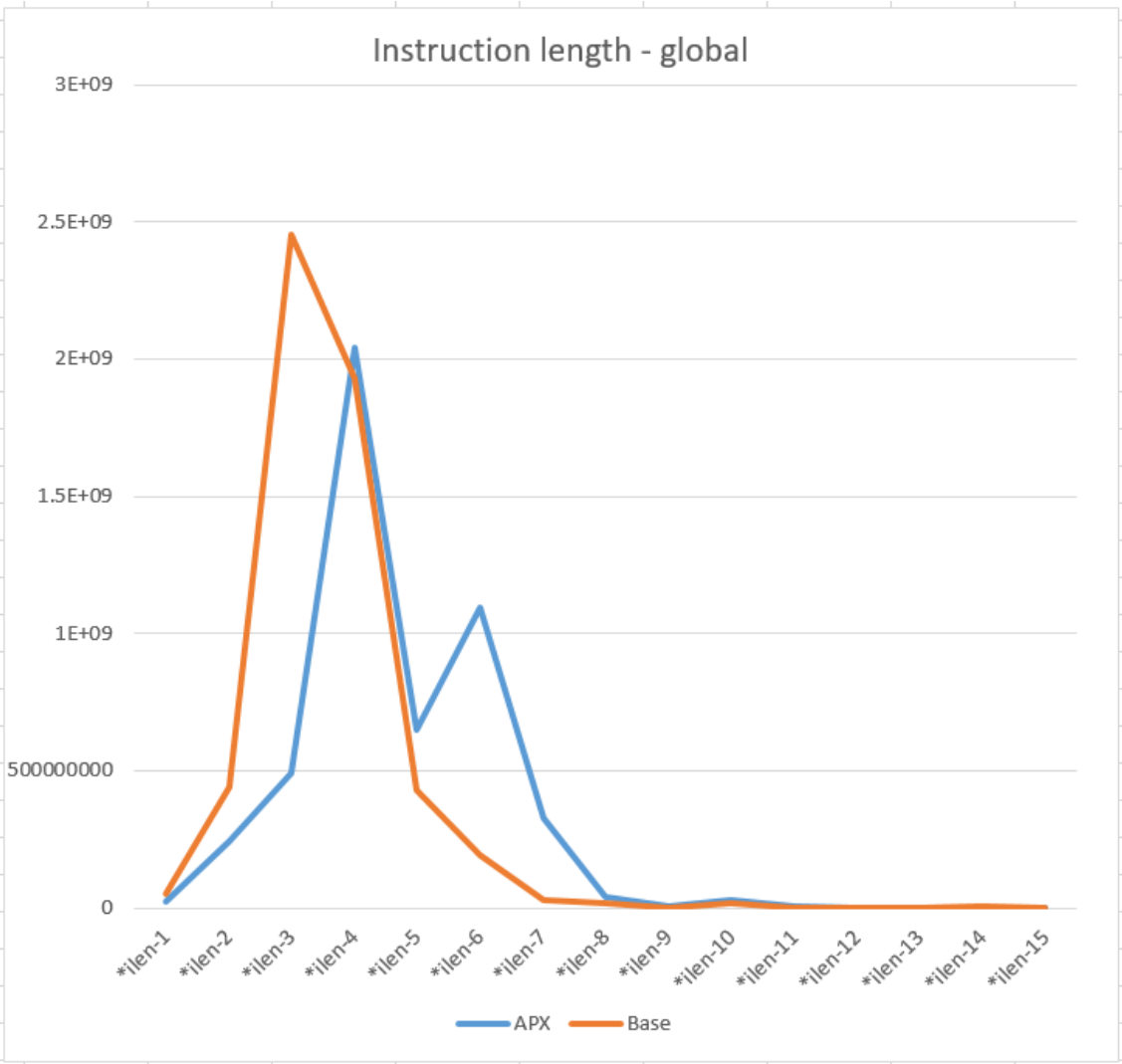


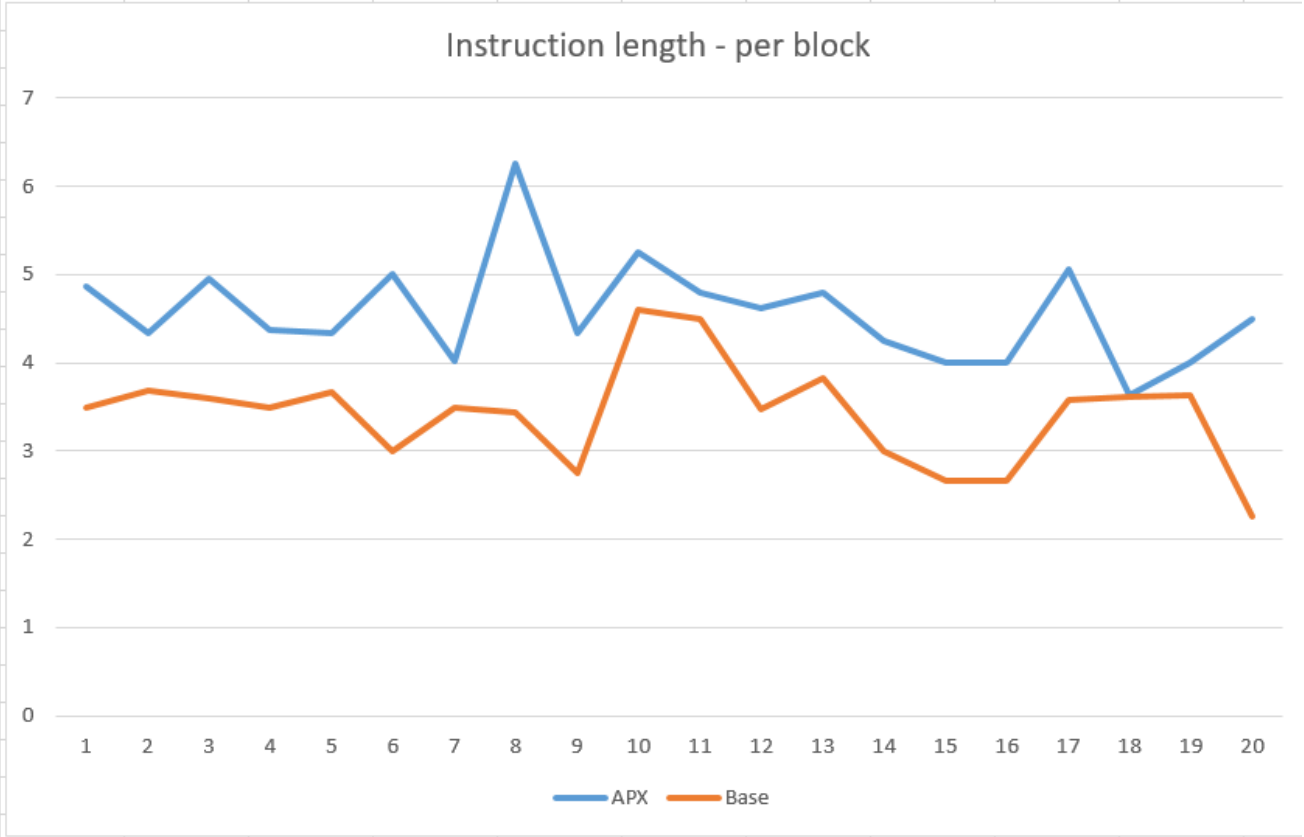
Engineering the ABI

- Not enough GPRs => spill, but not to stack. GPR – vector domain bridge.
- Tweaking the number of registers for outbound and return arguments.
- Constructing temporary data structures for argument passing.

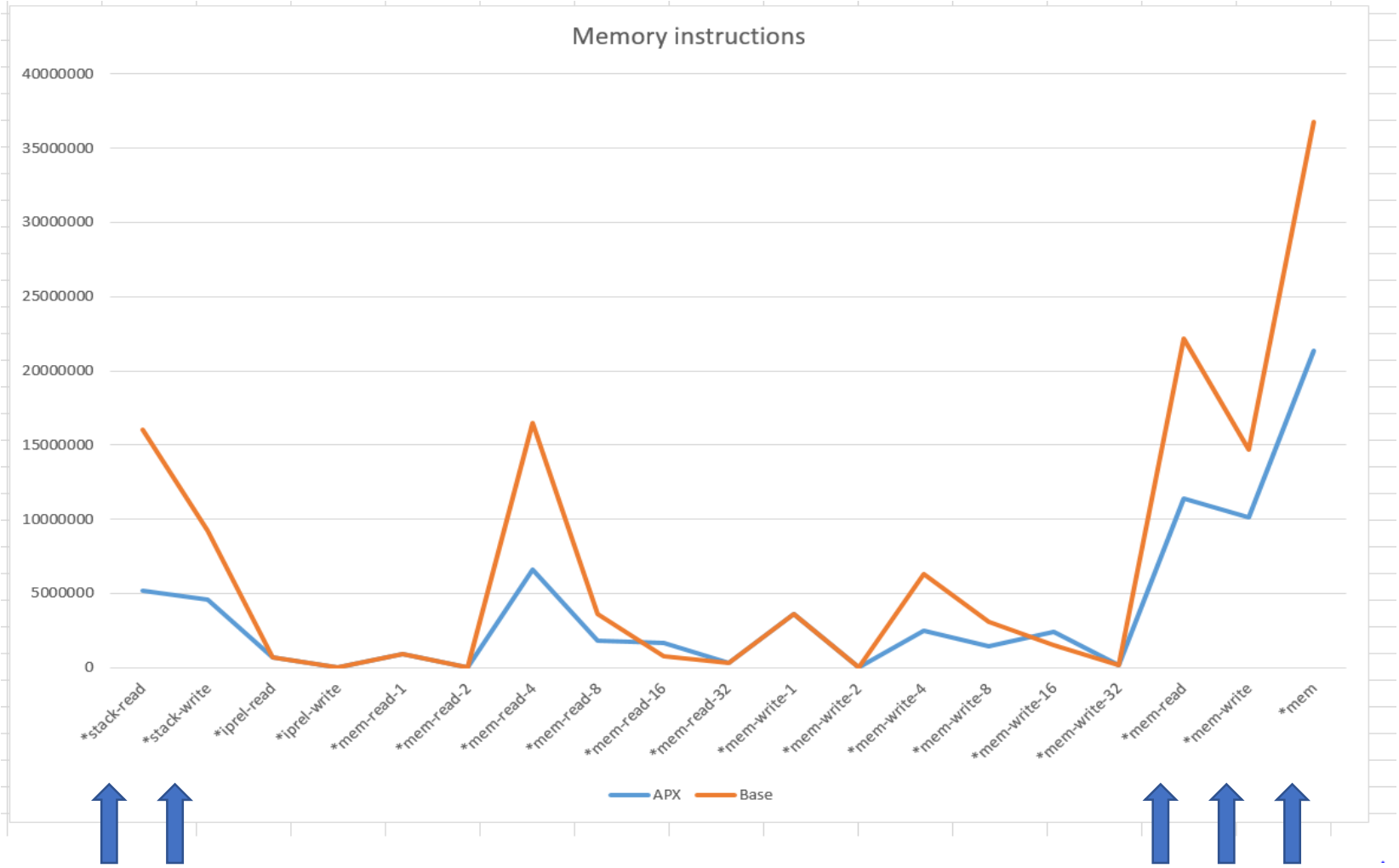
- May be fragile.
- Again, microbenchmarks may lie.

APX impact – instruction count and density





APX impact – memory operations



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
```



```
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

Universal ISA

- Comparing and contrasting x86, ARM, RISC-V and their related ABIs
- RISC vs CISC debate, and VLIW as well
- Register count, 32 –> 31 -> even less GP registers
- Fusion
- L/S vs Op/M
- Flags, predicates
- Operand count
- Destructive destination, dependencies
- Spilling strategy
- Code density impact on caching system
- ISA as a contract

Part 2 – Accessing an array

array[i];

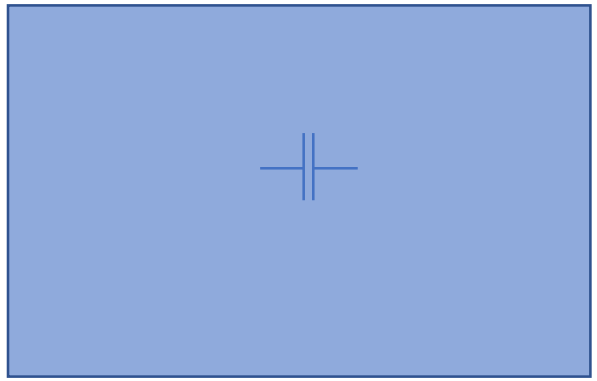
Part 2 – Accessing an array

array[i];

- Uncached, or use-once, array of [uniform] structures.
- Performance matters

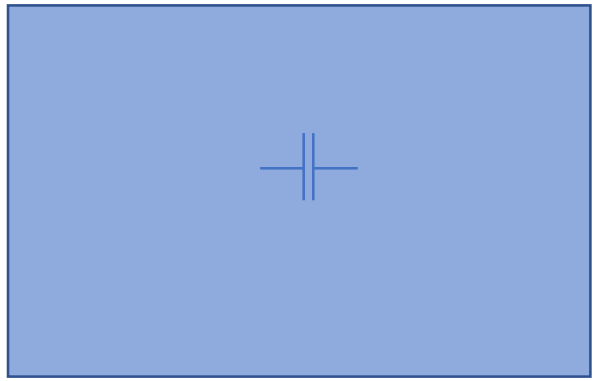
Memory

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



Memory

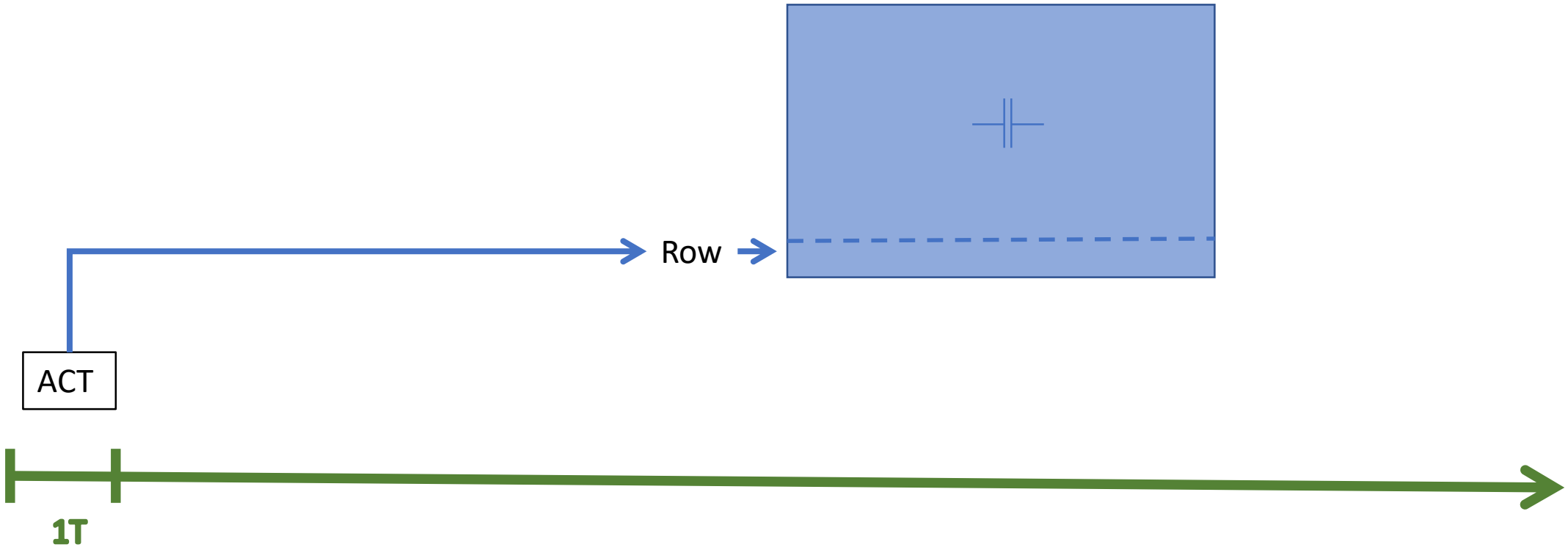
- Addressable capacitor array
- Short term nonvolatile
- Speeds – capacitor array vs bus interface





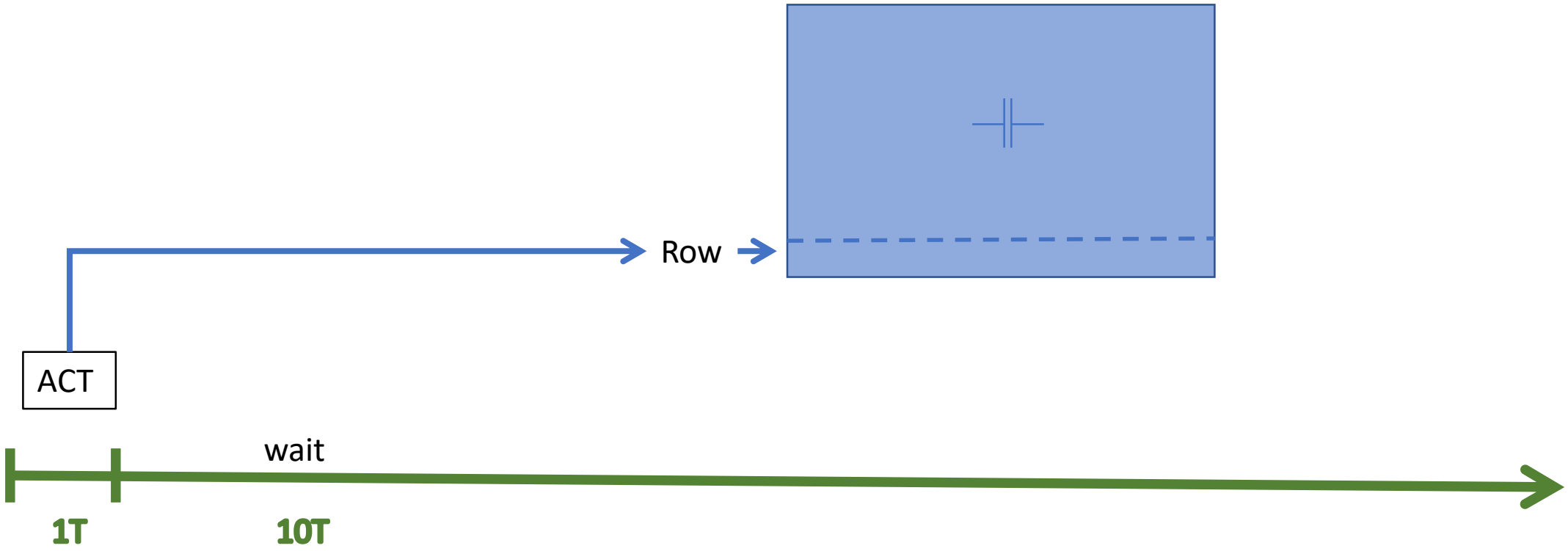
Memory

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



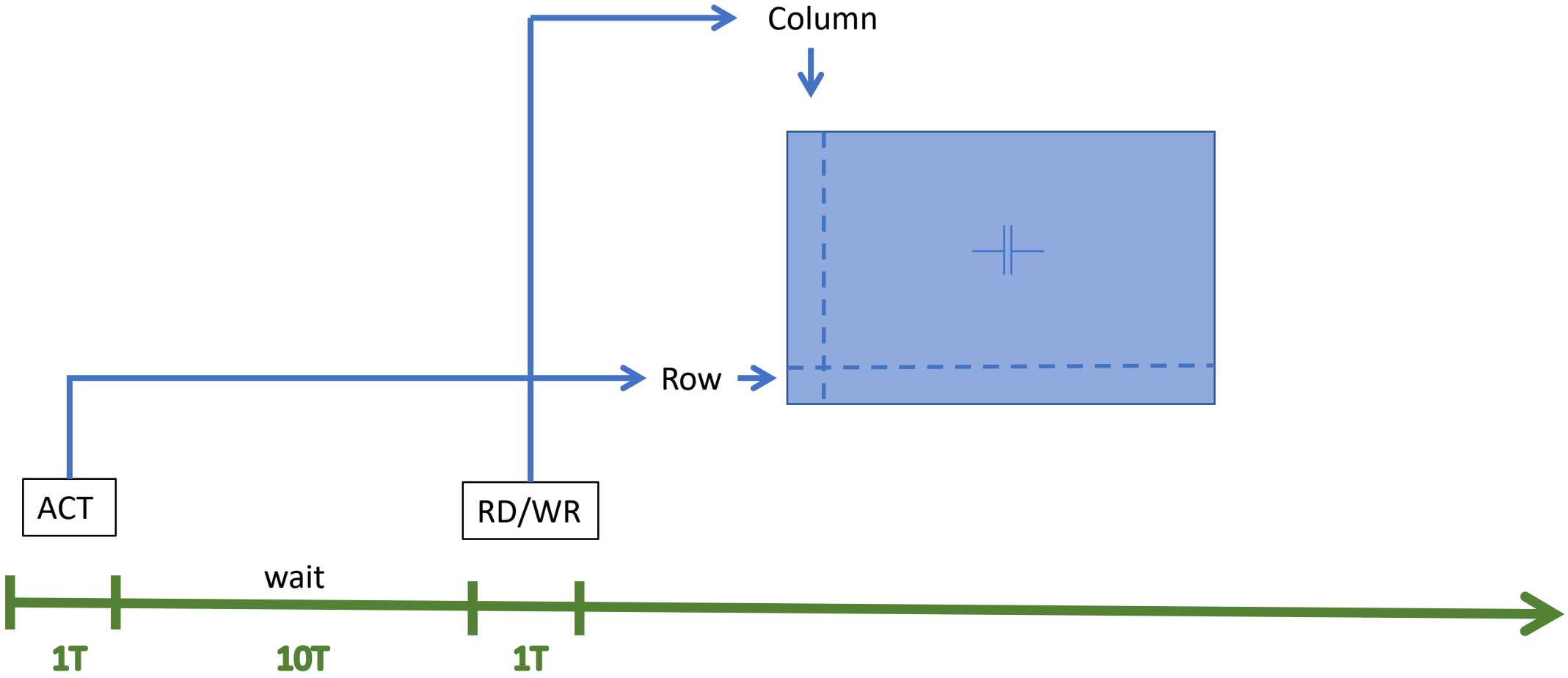
Memory

- Wait for a portion of array to get selected.



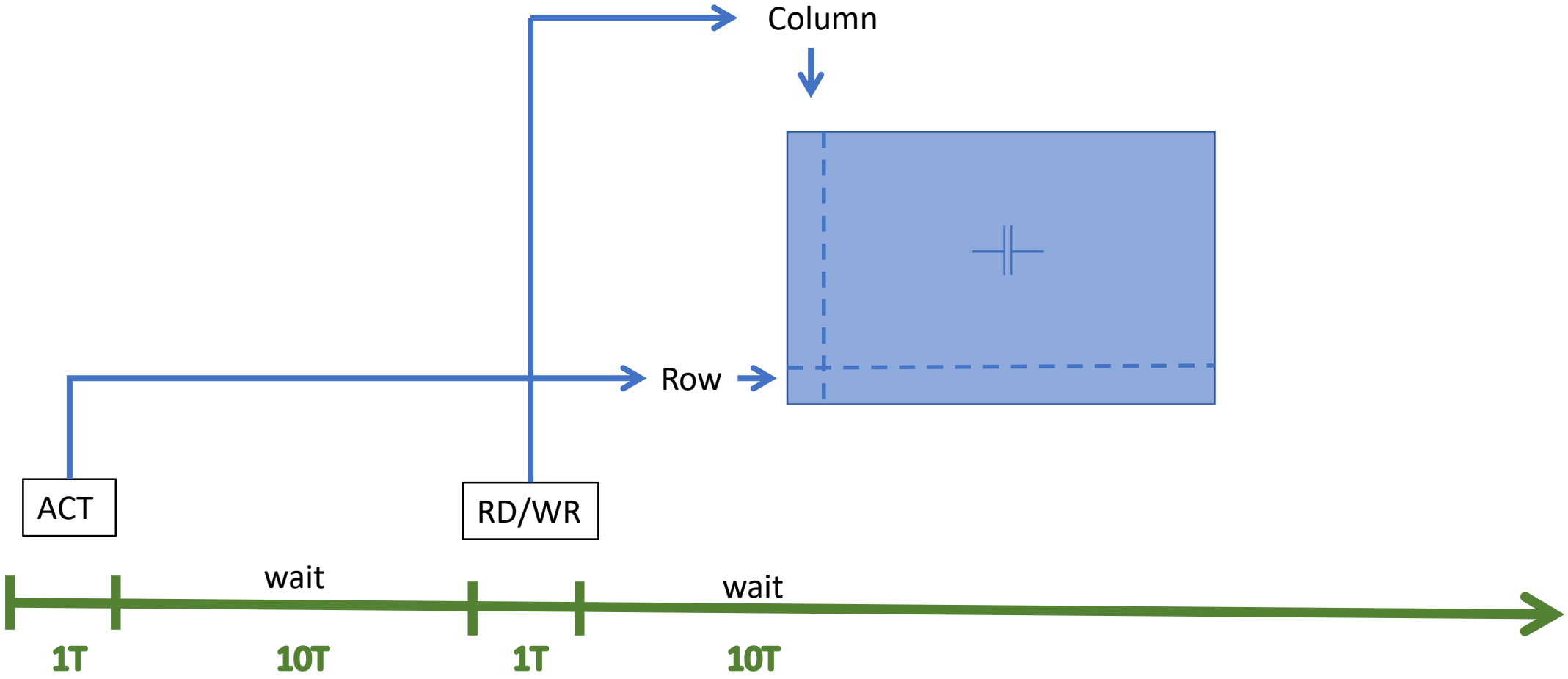
Memory

- Send column address and actual memory operation – another command



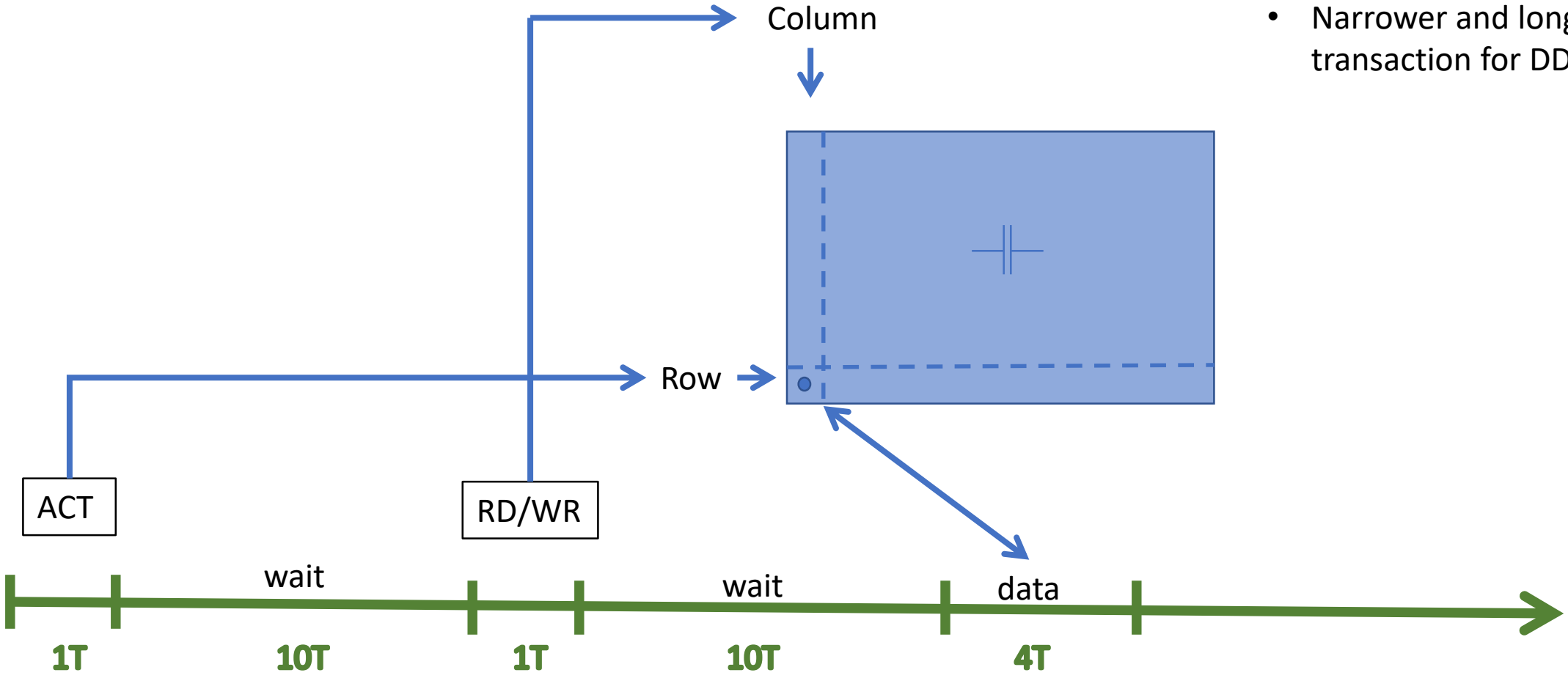
Memory

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



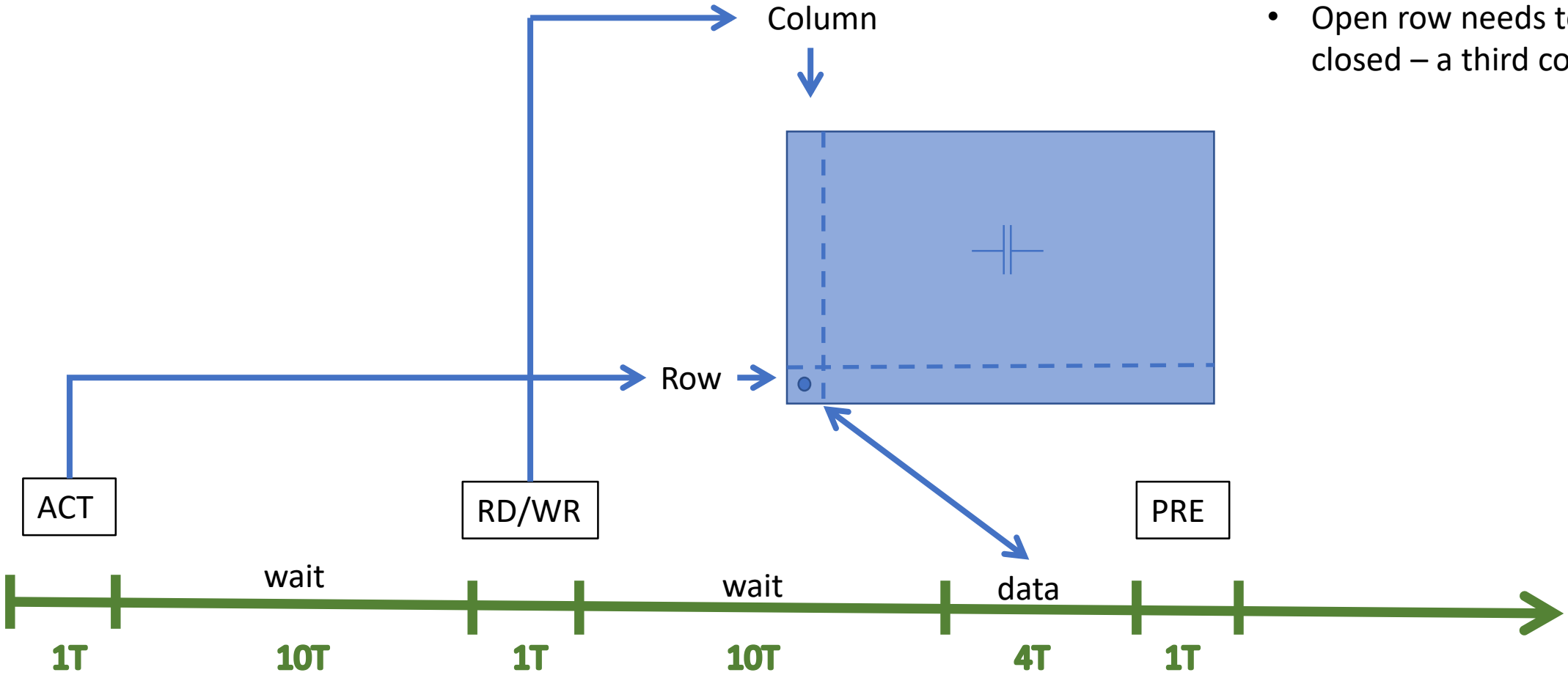
Memory

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



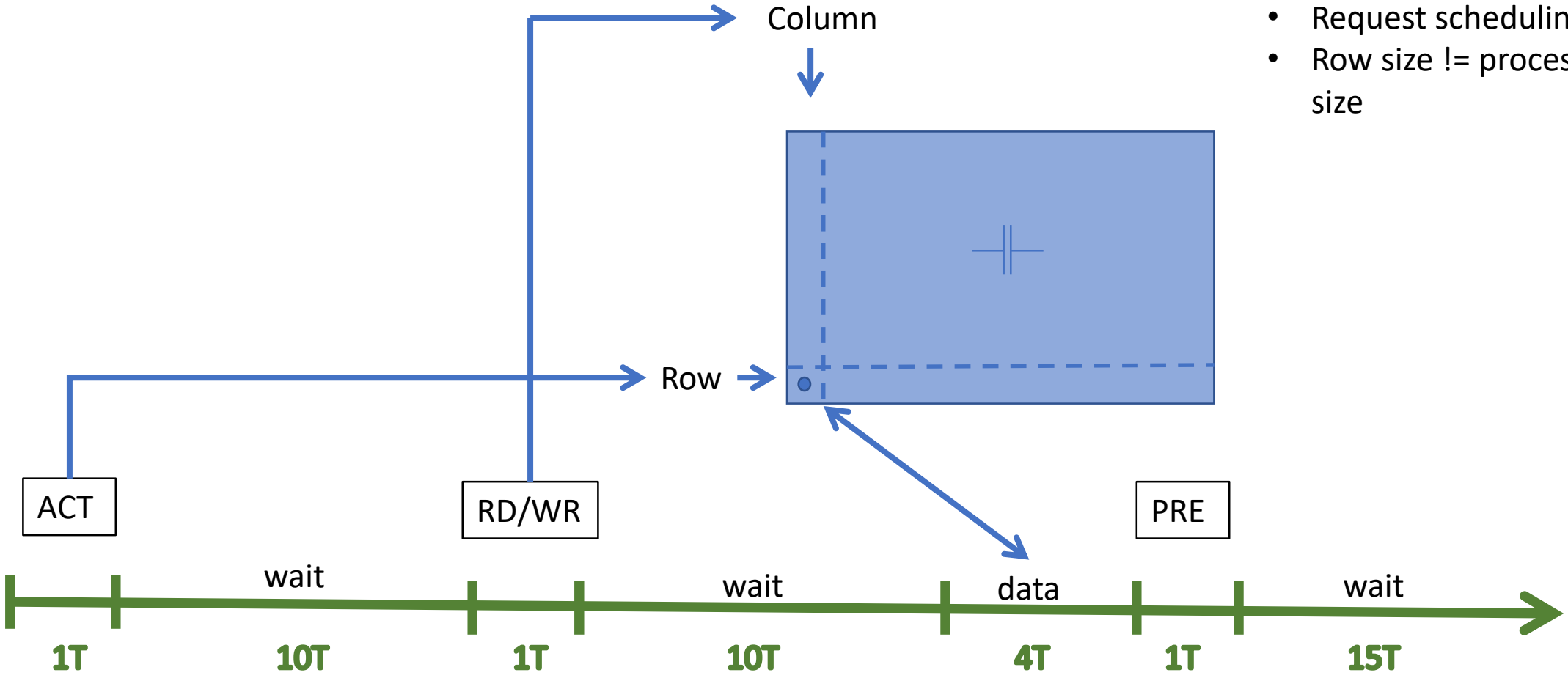
Memory

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



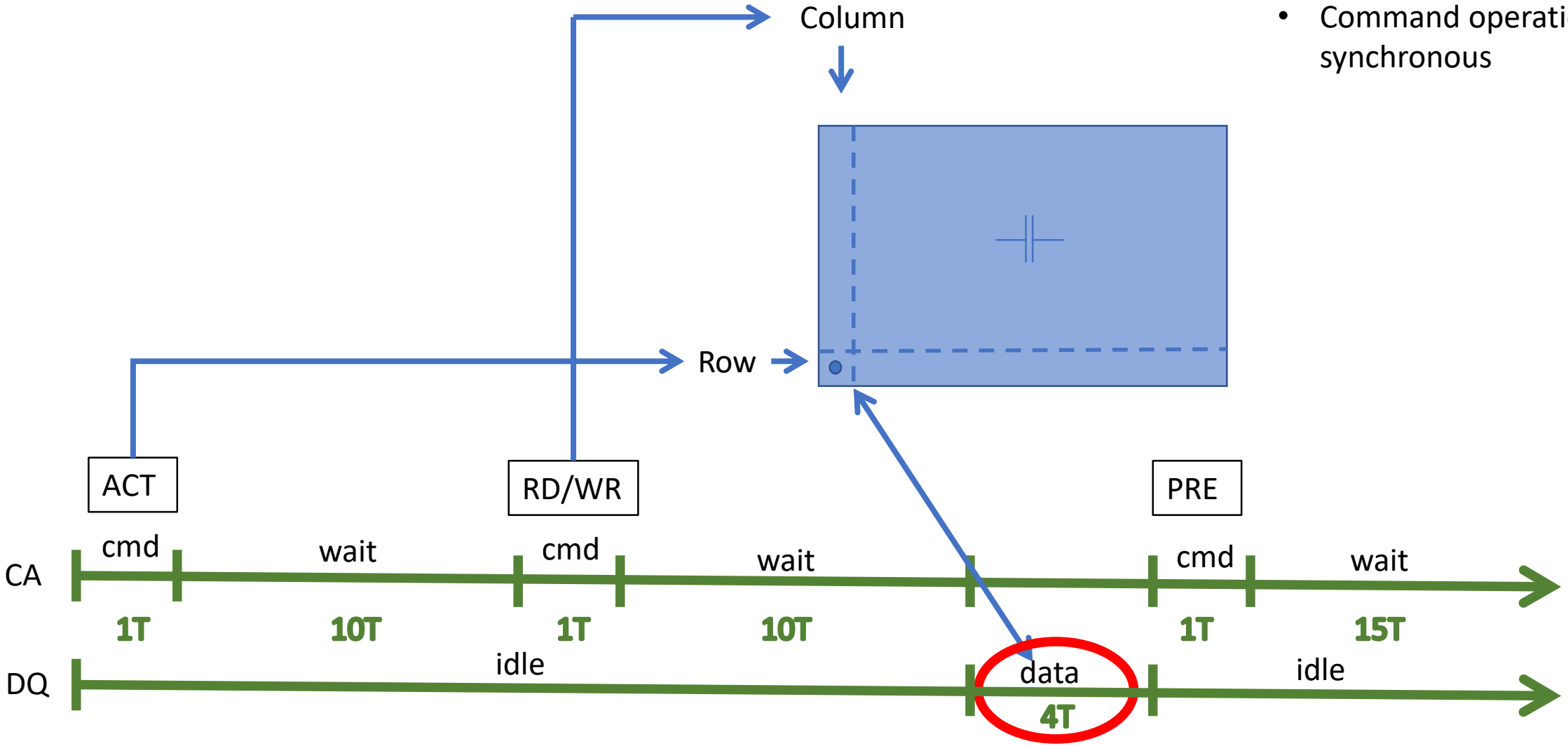
Memory

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



Memory

- Commands and data use separate buses
- Data bus is bidirectional
- Command operation is synchronous



Memory

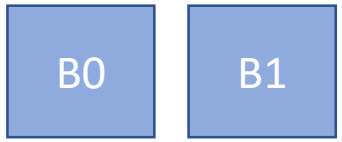
- It is only a fraction of bus transaction that is in fact DDR
- Only a single row open for data operations at a time
- Generation to generation capacitor array operates at mostly the same frequency, bus transfer speed in fact increases
- The laws of physics still hold and govern the performance profile.
- Multiple higher latency transactions in flight for throughput.

Memory



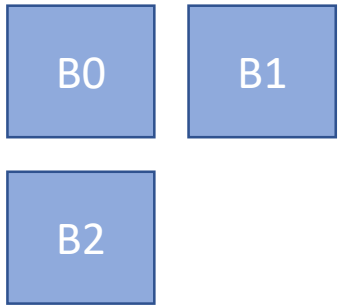
- One row open per bank
- Logical partitioning

Memory



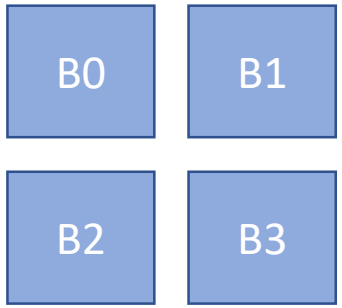
- One row open per bank
- Logical partitioning

Memory



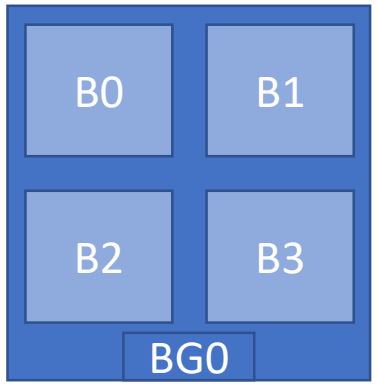
- One row open per bank
- Logical partitioning

Memory



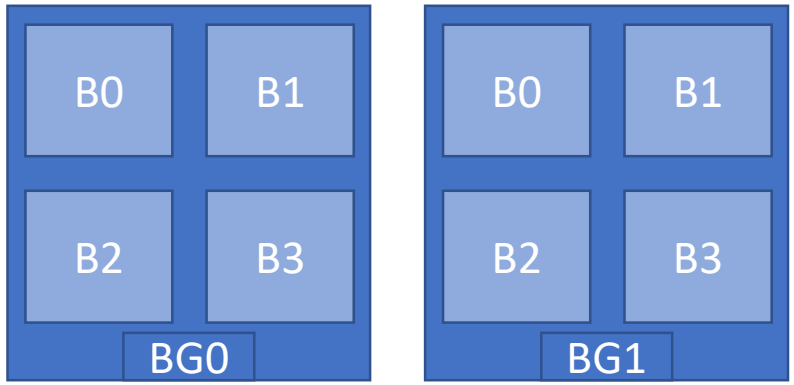
- One row open per bank
- Logical partitioning

Memory



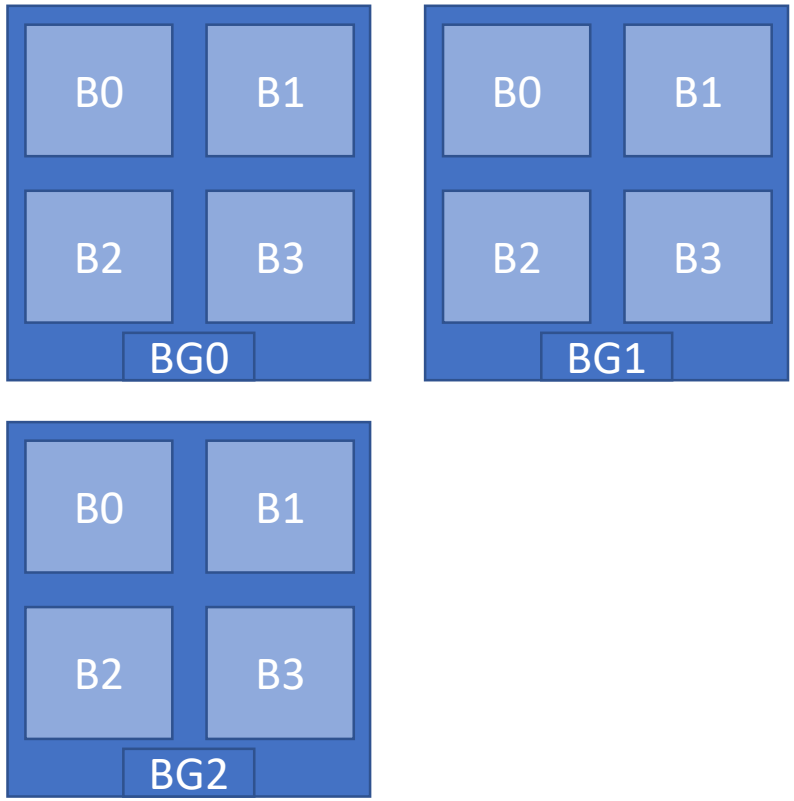
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



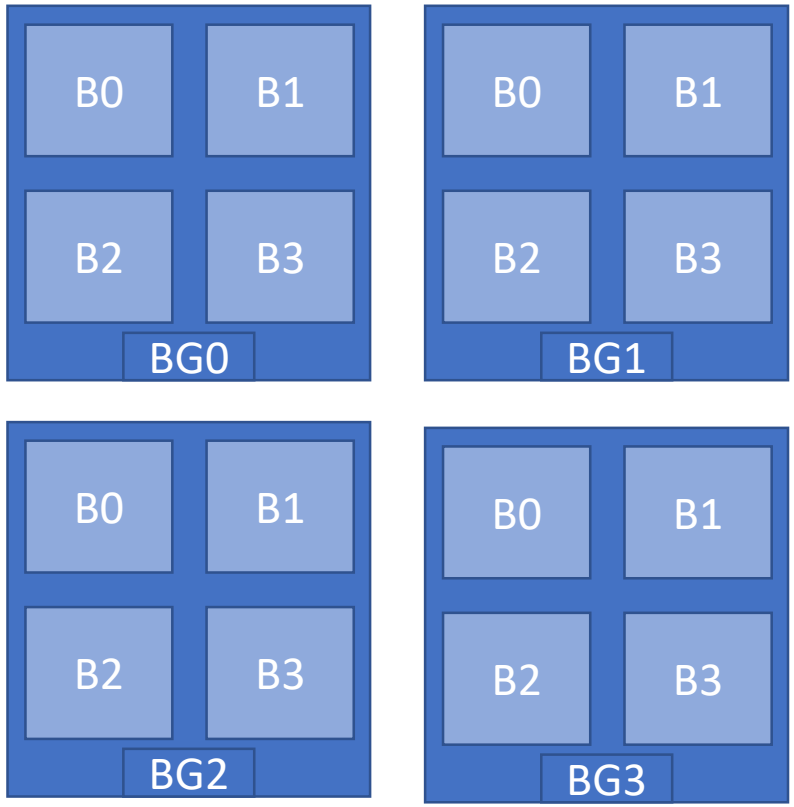
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



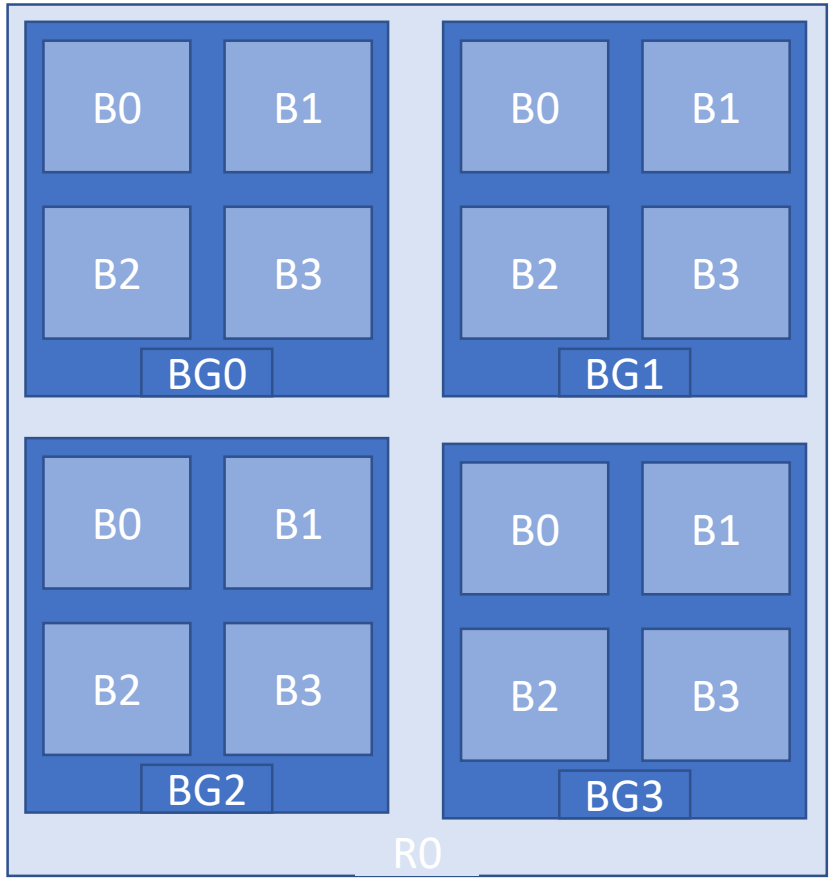
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

Memory



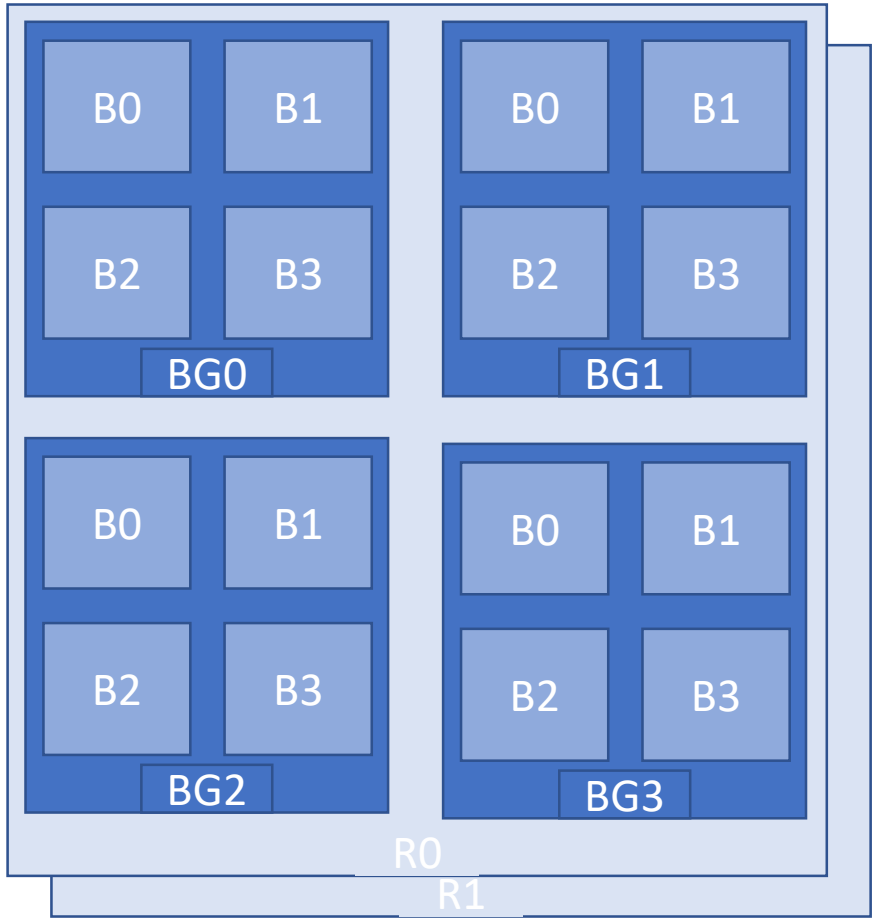
- Bank group – a grouping of logically addressable banks
- Multiple open rows per bank group
- Multiple outstanding commands in progress

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

Addressable memory units

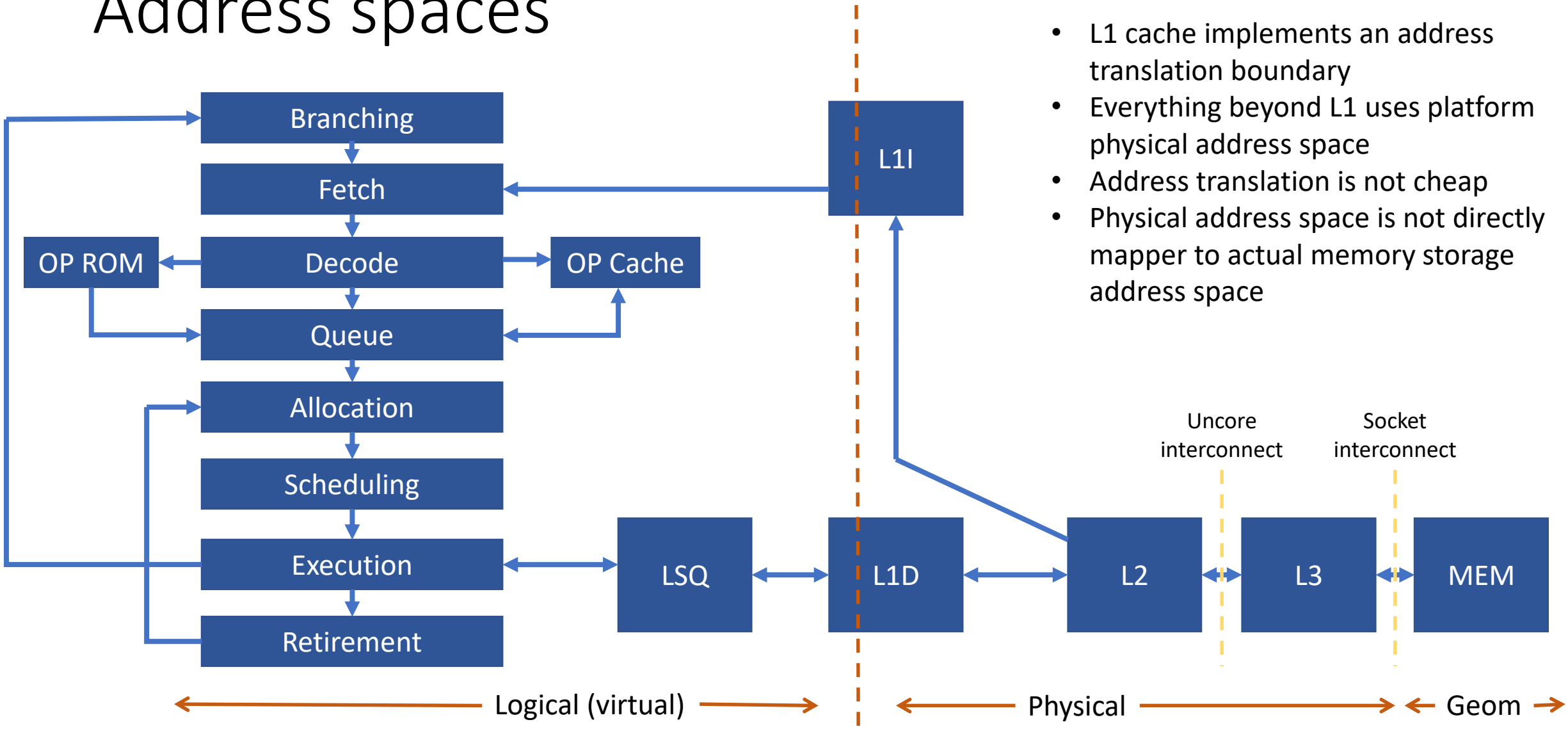
- SW visible page size is (typically) 4KB
- DDR memory page (row) size depends on the actual components used
- Mapping is controllable by low level firmware
- In most practical cases at boot time only

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

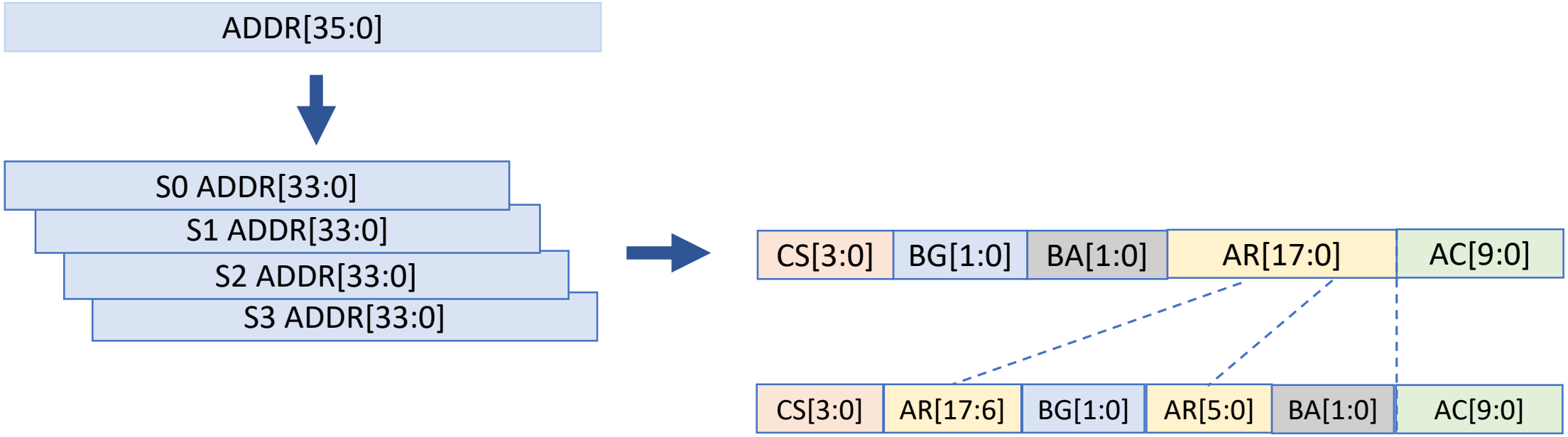
Address spaces



- L1 cache implements an address translation boundary
- Everything beyond L1 uses platform physical address space
- Address translation is not cheap
- Physical address space is not directly mapper to actual memory storage address space

Memory Address Mapping

- Sequential physical addresses can (and should) map to distributed geometrical addresses.
- Deeply platform specific, generally static, generally at boot time.



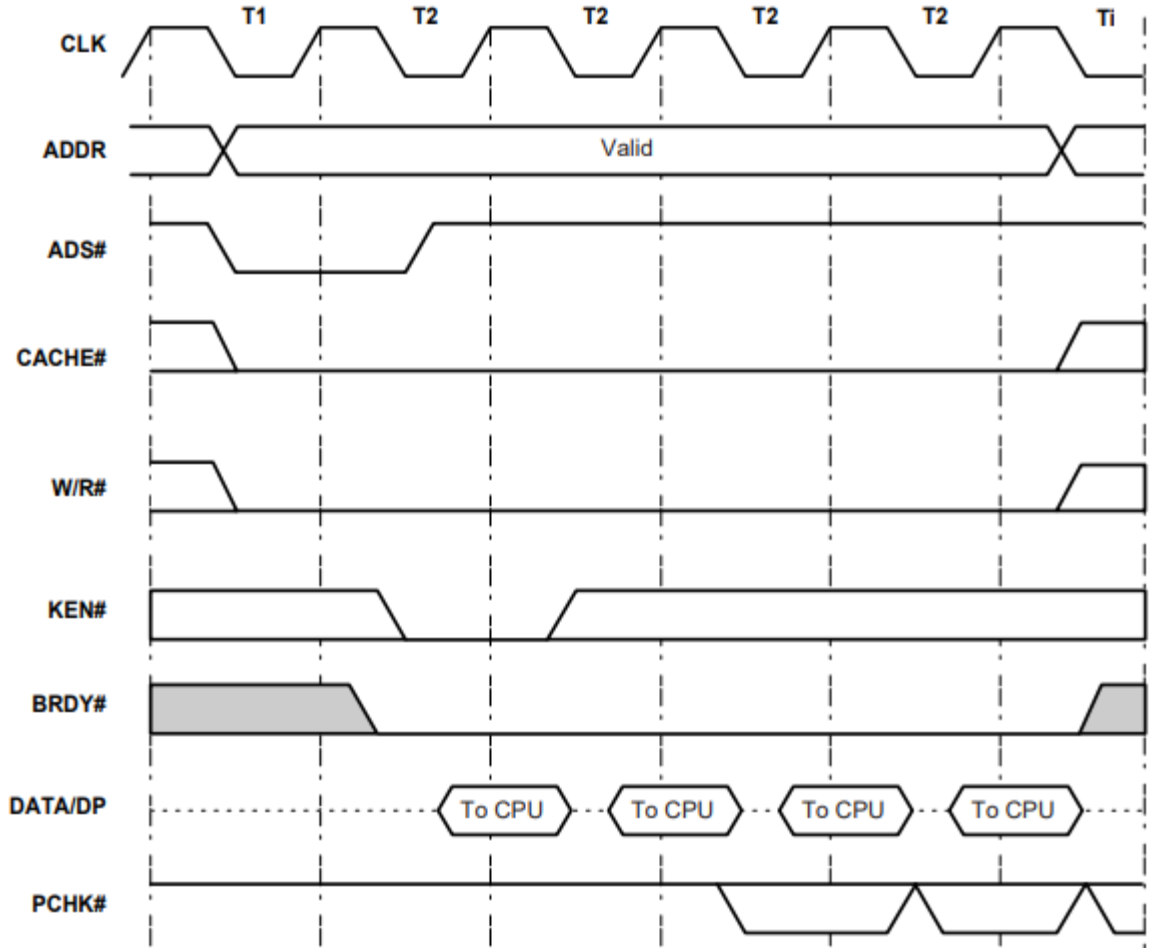
Accessing the array

- Seems simple – just unroll.
- But at what stride?
- For composite elements – in what sequence?

Detour – bus protocols

- 80586 as a simplified example
- 64 bit datapath, 32 byte cacheline

- Pentium 4 moved to 128 bit datapath and 64 byte cacheline



Detour – bus protocols

- Accesses are not necessarily sequential

Table 19-12. Processor Burst Order

1st Address	2nd Address	3rd Address	4th Address
0	8	10	18
8	0	18	10
10	18	0	8
18	10	8	0

NOTE: The addresses are represented in hexadecimal format

- Cache lines are for state tracking
- Cache blocks are for data access
- Critical word

```
struct S2 {
  uint64_t e0;
  uint64_t e1;
  uint64_t e2;
  uint64_t e3;
};
```

Detour – bus protocols

- DRAM bus width / transaction width

Table 18 — Burst Type and Burst Order

Burst Length	Read/ Write	Starting Column Address (A2,A1,A0)	burst type = Sequential (decimal) A3=0	burst type = Interleaved (decimal) A3=1	NOTE
4 Chop	READ	0 0 0	0,1,2,3,T,T,T,T	0,1,2,3,T,T,T,T	1,2,3
		0 0 1	1,2,3,0,T,T,T,T	1,0,3,2,T,T,T,T	1,2,3
		0 1 0	2,3,0,1,T,T,T,T	2,3,0,1,T,T,T,T	1,2,3
		0 1 1	3,0,1,2,T,T,T,T	3,2,1,0,T,T,T,T	1,2,3
		1 0 0	4,5,6,7,T,T,T,T	4,5,6,7,T,T,T,T	1,2,3
		1 0 1	5,6,7,4,T,T,T,T	5,4,7,6,T,T,T,T	1,2,3
		1 1 0	6,7,4,5,T,T,T,T	6,7,4,5,T,T,T,T	1,2,3
		1 1 1	7,4,5,6,T,T,T,T	7,6,5,4,T,T,T,T	1,2,3
	WRITE	0, V, V	0,1,2,3,X,X,X,X	0,1,2,3,X,X,X,X	1,2,4,5
		1, V, V	4,5,6,7,X,X,X,X	4,5,6,7,X,X,X,X	1,2,4,5
8	READ	0 0 0	0,1,2,3,4,5,6,7	0,1,2,3,4,5,6,7	2
		0 0 1	1,2,3,0,5,6,7,4	1,0,3,2,5,4,7,6	2
		0 1 0	2,3,0,1,6,7,4,5	2,3,0,1,6,7,4,5	2
		0 1 1	3,0,1,2,7,4,5,6	3,2,1,0,7,6,5,4	2
		1 0 0	4,5,6,7,0,1,2,3	4,5,6,7,0,1,2,3	2
		1 0 1	5,6,7,4,1,2,3,0	5,4,7,6,1,0,3,2	2
		1 1 0	6,7,4,5,2,3,0,1	6,7,4,5,2,3,0,1	2
		1 1 1	7,4,5,6,3,0,1,2	7,6,5,4,3,2,1,0	2
	WRITE	V, V, V	0,1,2,3,4,5,6,7	0,1,2,3,4,5,6,7	2,4

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

Part 3 – Mapping to Higher Level Constructs

- A partitioned hash table-based data structure
- Heavy on fragment iteration
- Hash function has a larger working set

- Explored tweaks to x86 psABI for spill reduction and parameter passing optimization
- Unrolling with physical memory stride awareness

Measuring Performance

- All of this is very platform specific
- Microarchitectural microbenchmarks are fun
- Although nowhere near being easy
- And might not matter that much on a global scale
- Fleetwide benchmarks and telemetry do matter though

Discussion