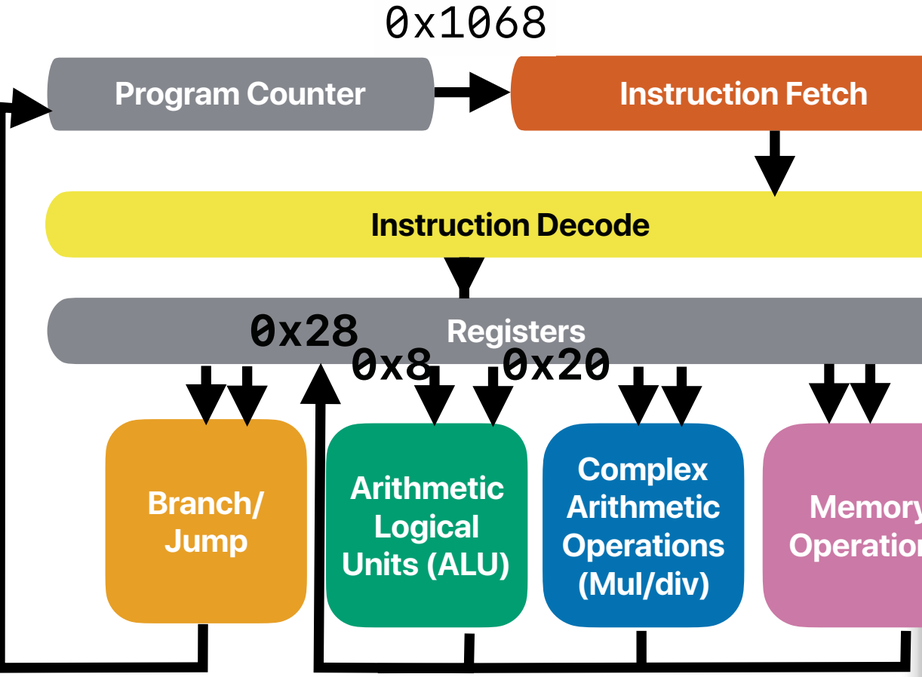
Final Review

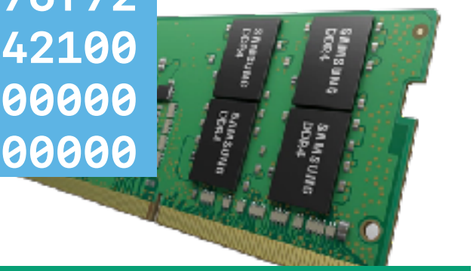
Hung-Wei Tseng

von Neumann model



Processor

Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000



Memory

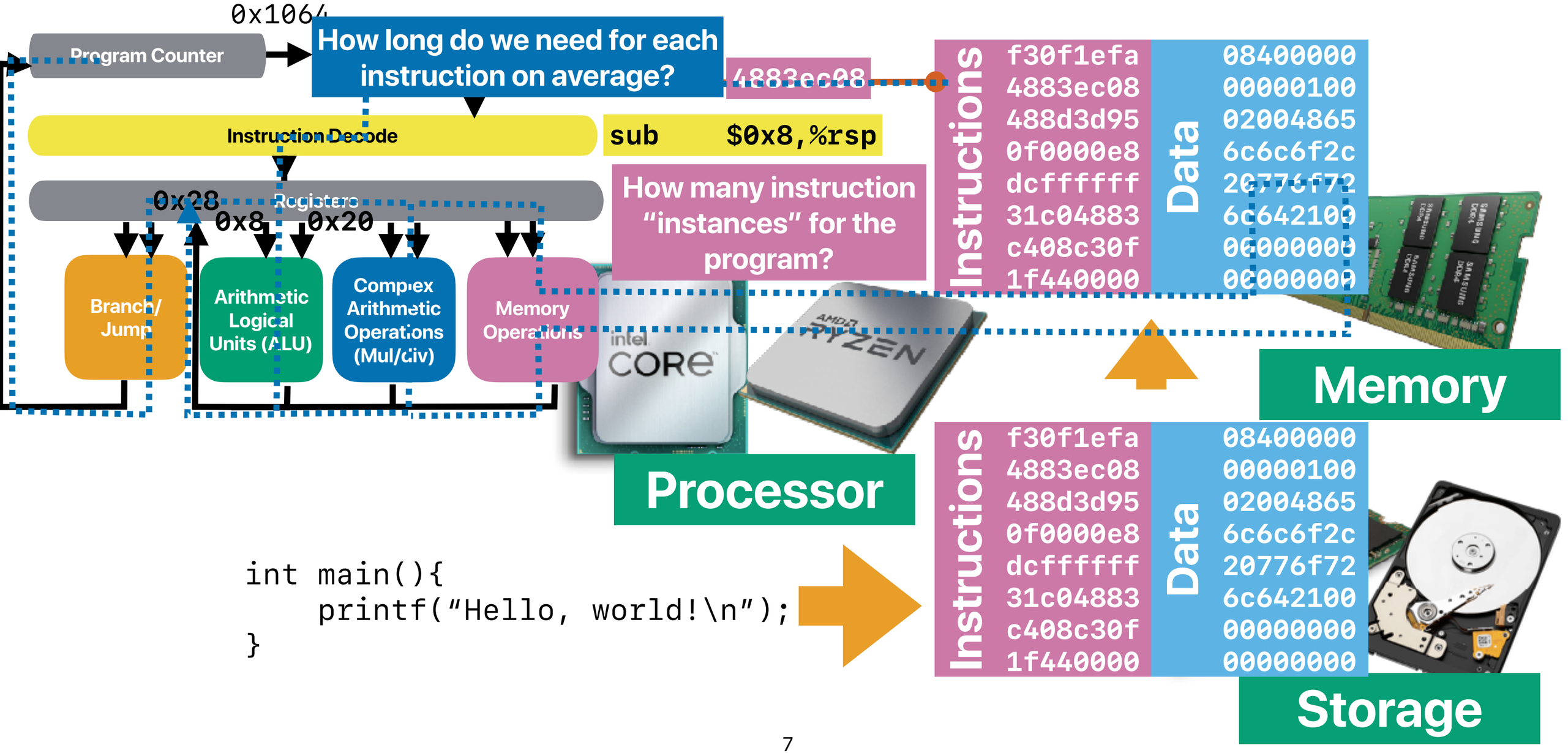
```
int main(){
    printf("Hello, world!\n");
}
```

Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000

Storage

By loading different programs into memory, your computer can perform different functions

Execution time of a program in the von Neumann model




```
int main(){
    printf("Hello, world!\n");
}
```

Classic CPU Performance Equation (ET of a program)

How many instruction "instances" for the program? × How long do we need for each instruction on average?

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

C Code	x86 instructions

`int init_data(int64_t *data, int data_size) {`
`  register unsigned int i = 0;`
`  for(i = 0; i < data_size; i++)`
`{`
`  data[i] = i;`
`  }`
`  return 0;`
`}`
 	
`int main(int argc, char **argv) {`
`  int *data = malloc(8*1024);`
`  init_data(data, 1024)`
`  return 0;`
`}`
	`init_data:`
	`.LFB16:`
	`  endbr64`
	`  testl  %esi, %esi`
	`  jle  .L2`
	`  leal -1(%rsi), %ecx`
	`  xorl %eax, %eax`
	`.L3:`
	`  movq %rax, %rdx`
	`  movq %rax, (%rdi,%rax,8)`
	`  addq $1, %rax`
	`  cmpq %rcx, %rdx`
	`  jne  .L3`
	`.L2:`
	`  xorl %eax, %eax`
	`  ret`

If data memory access instructions takes 5 cycles, others take only 1 cycle, CPU freq. = 4 GHz

$$CPI_{average} = 0.2 \times 5 + 0.8 \times 1 = 1.8$$

$$ET = (1024 \times 5) \times 1.8 \times \frac{1}{4 \times 10^9} = 2.304\ us$$

1024x

How can we get good performance?

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

How to lower CT?

- Programmer
  - Ensures the system set the speed of your processor running at maximum speed
- Hardware manufacturers
  - Using better process technology

How to lower IC?

- Programmer
  - Implementing algorithms with fewer operations, iterations
  - Wise use of instructions
    - Vector instructions (through OpenMP, intrinsics)
    - Specialized instructions (e.g., popcount)
  - Composing code that facilitates compiler optimizations
- Compiler
  - inline — remove function call overhead
  - unrolling — remove loop control overhead
  - common sub-expression elimination, constant propagation, Loop invariant code motion
- Lower the IC has no guarantees on ET reduction as each instruction might take longer amount of time

How to lower CPI?

- Probably the most important thing you learned in CSE142!
- Memory operations — must relying caching, TLB
  - Programmer must carefully write “cache-friendly code”
  - Compiler — use registers as often as possible
- Instruction-level parallelism
  - Hardware side
    - Pipelining — allowing multiple instructions to make progress concurrently
    - Branch predictions — allowing the pipeline to continuous fetching instructions
    - SuperScalar — allowing instructions to start executing (issue) concurrently
    - OoO — schedule instructions purely based on the resolution of data dependency to reduce unnecessary idle waiting
  - Programmer’s side
    - Reduce data dependencies in the code
    - Making the code behavior more predictable
  - Compiler — strength reduction, reorder instructions, loop unrolling

Lessons learned from Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$

- Corollary #1: Maximum speedup
- Corollary #2: Make the common case fast
  - Common case changes all the time
- Corollary #3: Optimization is a moving target
- Corollary #4: Exploiting more parallelism from a program is the key to performance gain in modern architectures
- Corollary #5: Single-core performance still matters
- Corollary #6: Don't hurt the most common case too much

$$Speedup_{max}(f, \infty) = \frac{1}{(1 - f)}$$

$$Speedup_{max}(f_1, \infty) = \frac{1}{(1 - f_1)}$$

$$Speedup_{max}(f_2, \infty) = \frac{1}{(1 - f_2)}$$

$$Speedup_{max}(f_3, \infty) = \frac{1}{(1 - f_3)}$$

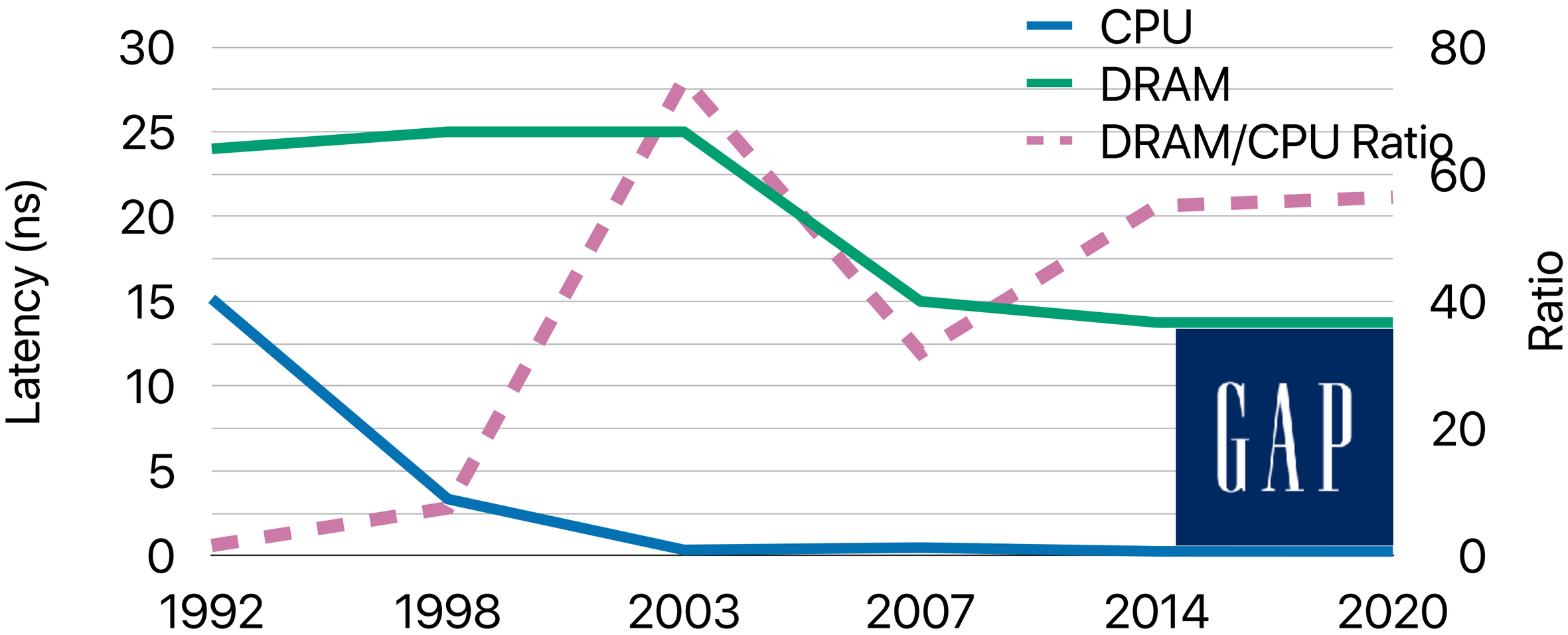
$$Speedup_{max}(f_4, \infty) = \frac{1}{(1 - f_4)}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

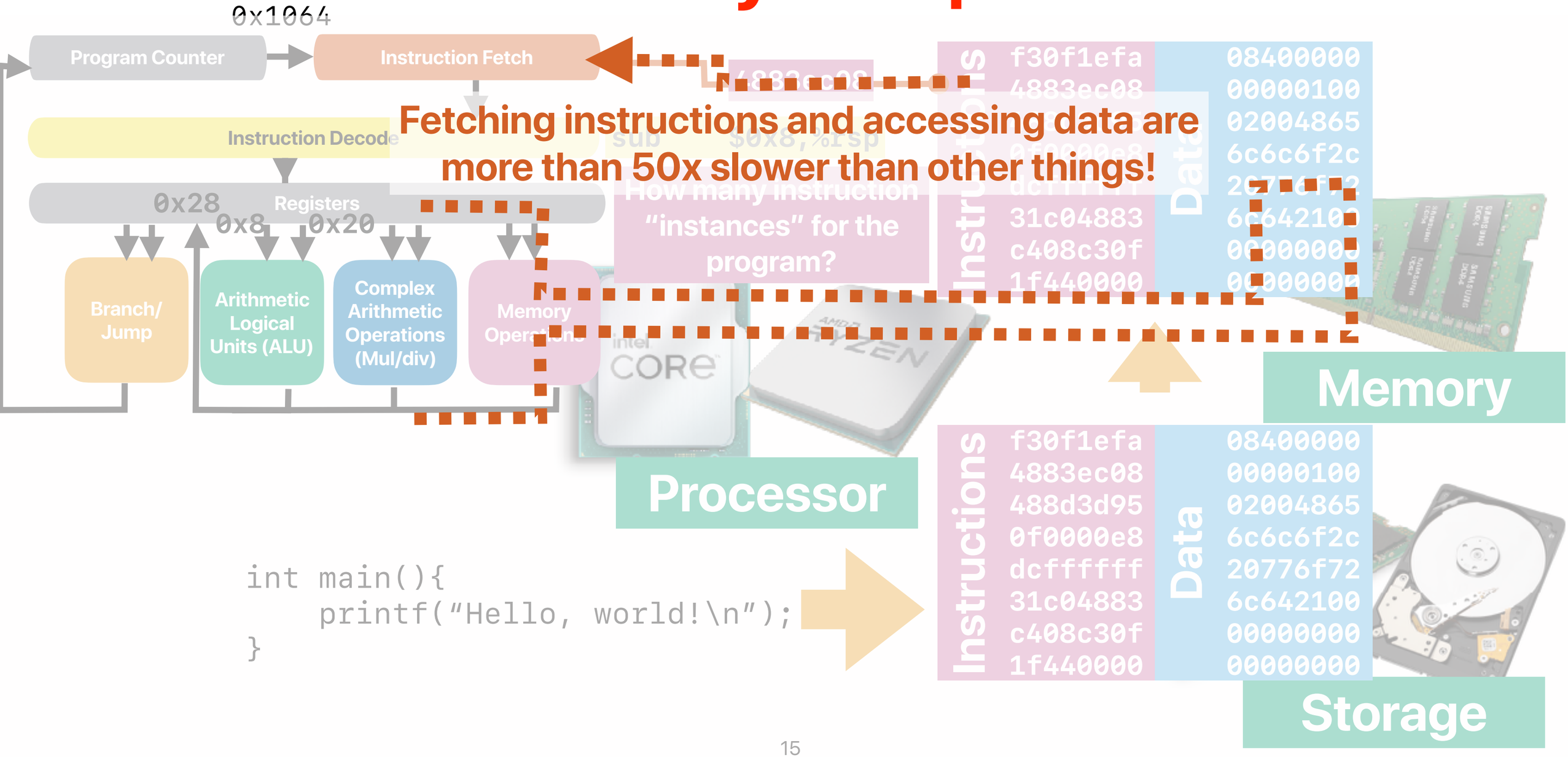
$$Speedup_{enhanced}(f, s, r) = \frac{1}{(1 - f) + perf(r) + \frac{f}{s}}$$

The “latency” gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

The memory-wall problem




```
int main(){
    printf("Hello, world!\n");
}
```

The “life” of an instruction

- **Instruction Fetch (IF) — fetch the instruction from memory**
- Instruction Decode (**ID**)
  - Decode the instruction for the desired operation and operands
  - Reading source register values
- Execution (**EX**)
  - ALU instructions: Perform ALU operations
  - Conditional Branch **(BR)**: Determine the branch outcome (taken/not taken)
- **Data Memory Access (MEM)**
  - **Memory instructions: Determine the effective address for data memory access**
  - **Read/write memory**
- Write Back (**WB**) — Present ALU result/read value in the target register
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4

Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

tens of ns

Processor

Processor Core

Registers

SRAM $

DRAM

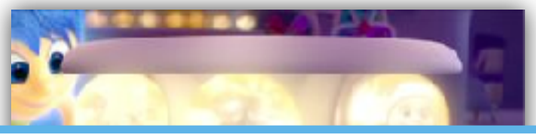
Storage

32 or

GBs

TBs

larger



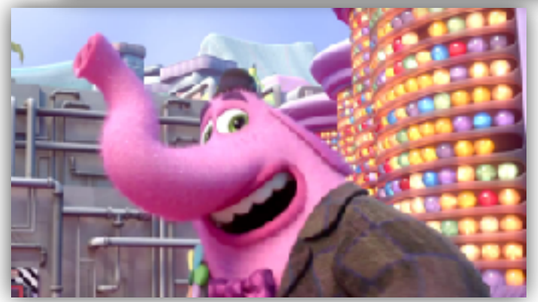
fastest

L1 $

L2 $

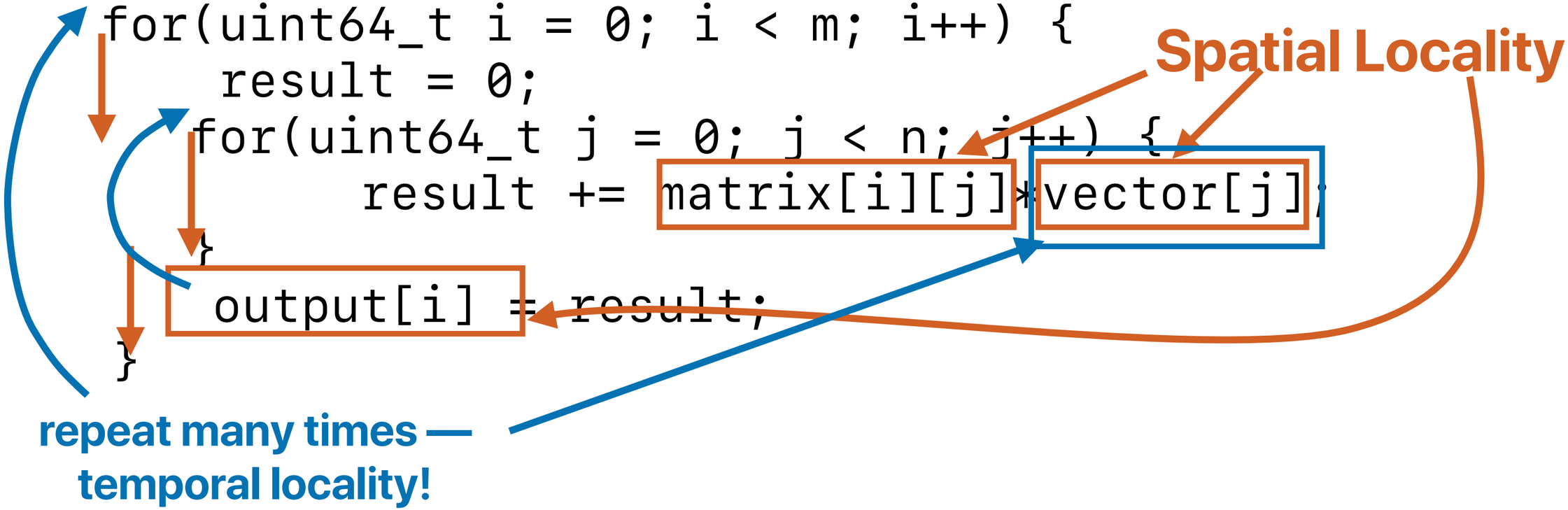
L3 $

larger

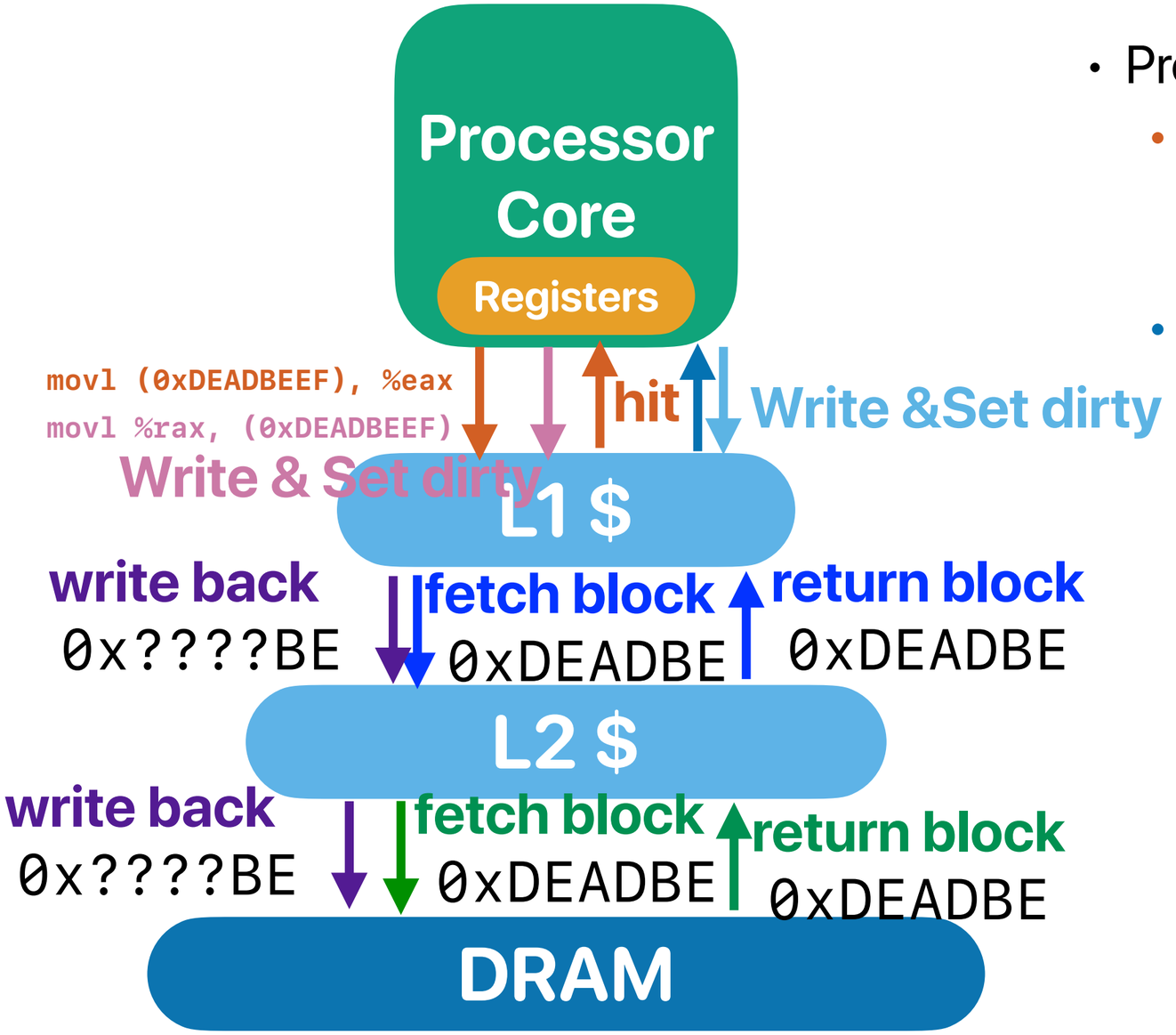


How is it possible to cache small things but hit most time?

- Theory of Locality — our program only visit a small amount of instructions/ data most of time



Processor/cache interaction — block-in-block-out



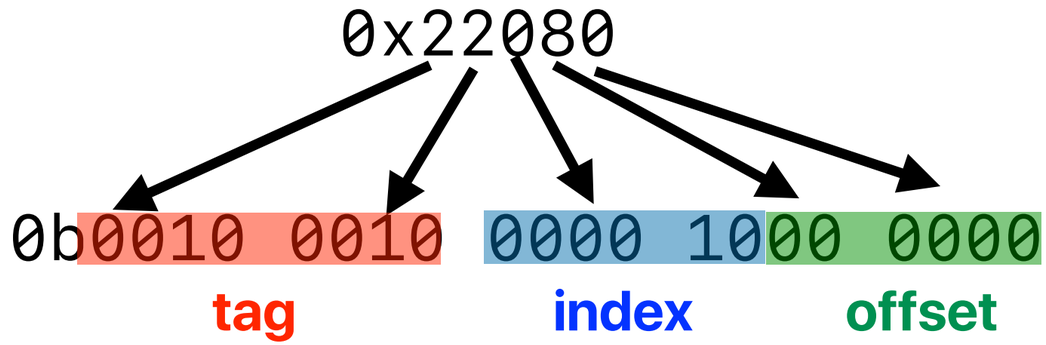
- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

How to write cache-friendly code?

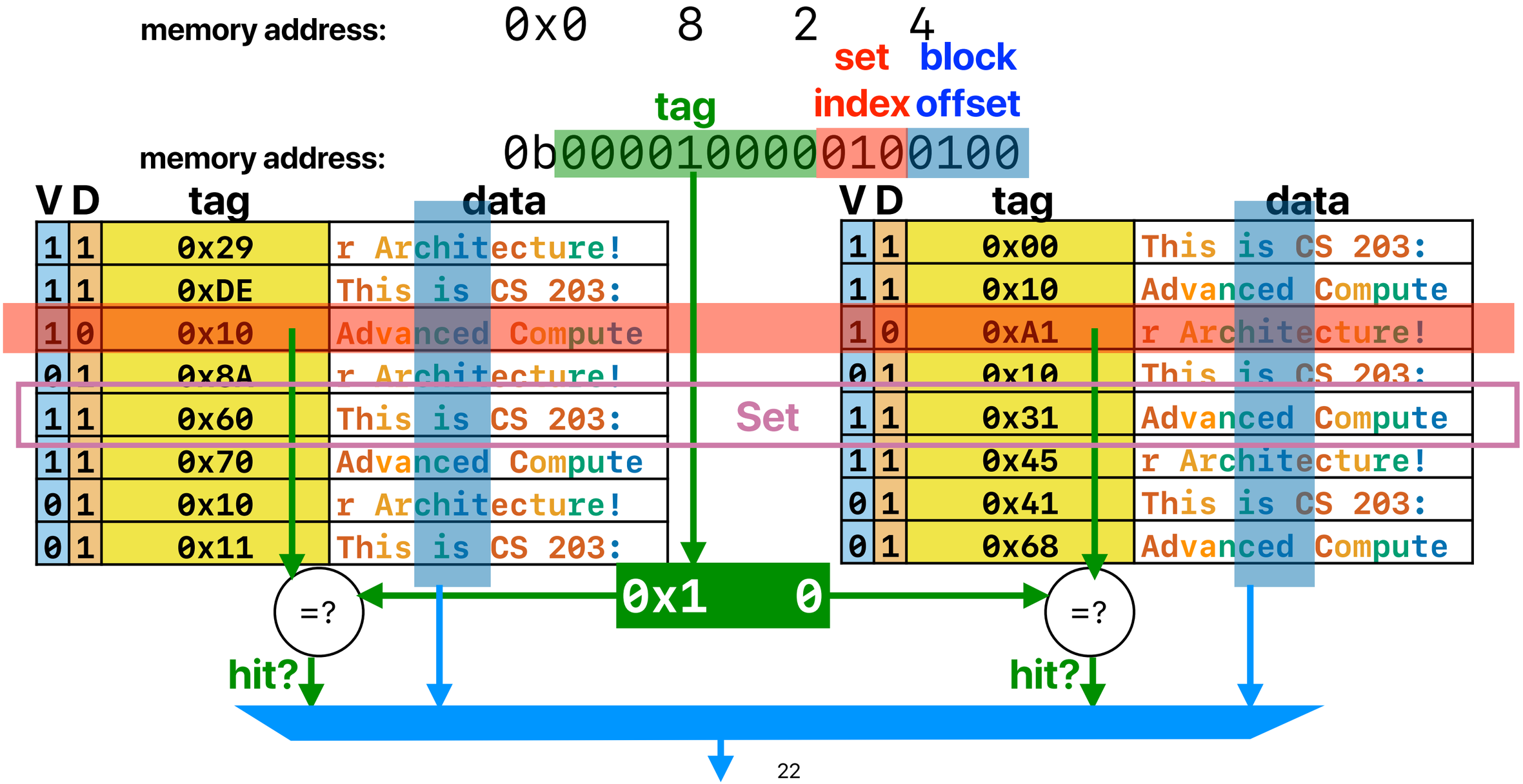
- Instruction cache is typically not a big deal. So we only discuss data accesses in most cases
- Evaluating the cache miss rate of the code
  - Understanding the geometry (i.e., C = A B S) of your cache
  - List a sequence of memory addresses of data accesses
  - Partition each address based on the geometry of the cache
  - Simulate the cache to see if it's a hit or a miss, also, what kind of misses
- Modify your implementation for each specific cause of misses in your code
  - Compulsory misses — the miss due to first time access of a **block**
  - Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - Capacity misses — the miss due to the working set size surpasses the **capacity**

Recap: C = ABS

- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Cacheline)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
- number of bits in **b**lock **offset** — lg(**B**)
- number of bits in **s**et **index**: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index



Way-associative cache



NVIDIA Tegra X1

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Compulsory Miss	a[0–7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	**Conflict Miss**	b[0–7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	**Conflict Miss**	c[0–7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	**Conflict Miss**	d[0–7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	**Conflict Miss**	e[0–7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	**Conflict Miss**	a[0–7]

When this miss occurs, we just visited 4 other blocks before the last visit to the same block — lower than the total cache capacity!

NVIDIA Tegra X1 (cont.)

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

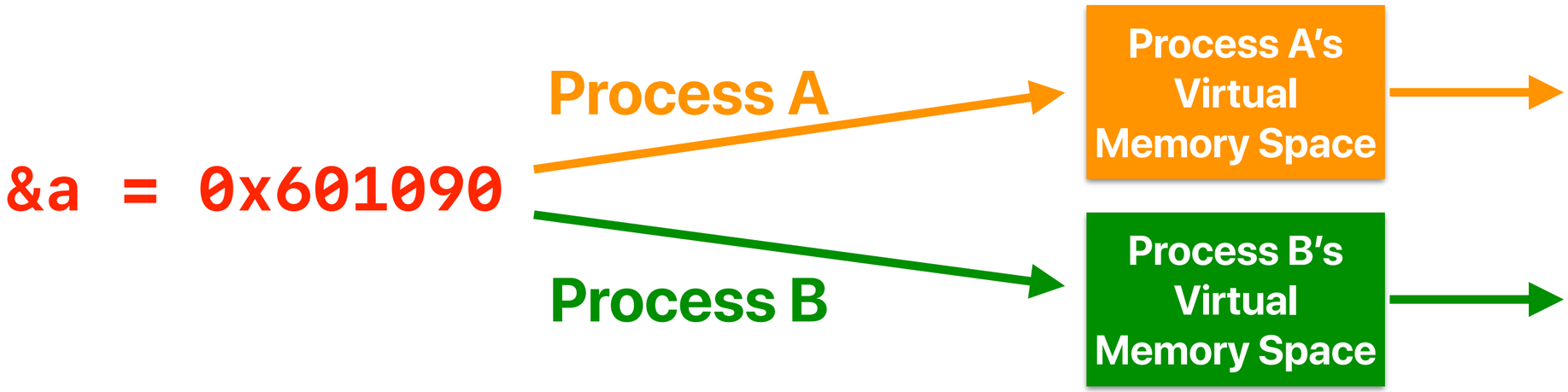
	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x8	0x0	**Conflict Miss**	
b[7]	0x20038	0b0010000000000111000	0x10	0x0	**Conflict Miss**	
c[7]	0x30038	0b0011000000000111000	0x18	0x0	**Conflict Miss**	
d[7]	0x40038	0b0100000000000111000	0x20	0x0	**Conflict Miss**	
e[7]	0x50038	0b0101000000000111000	0x28	0x0	**Conflict Miss**	a[0–7]
a[8]	0x10040	0b0001000000001000000	0x8	0x1	Compulsory Miss	
b[8]	0x20040	0b0010000000001000000	0x10	0x1	Compulsory Miss	
c[8]	0x30040	0b0011000000001000000	0x18	0x1	Compulsory Miss	
d[8]	0x40040	0b0100000000001000000	0x20	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	Compulsory Miss	a[8–15]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

100% miss rate!

Remedies of cache misses

- Compulsory misses — the miss due to first time access of a **block**
  - Can we carry more useful data within the same block? — condense data layout, change the orientation of data (row/column)
  - Can we fetch the block earlier? — prefetch
- Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - We need more associativity — change the processor
  - Can we avoid the access pattern that maps data to the same set?
    - Only work on limited blocks during a period of time — tiling
    - Change the stride (row/col size) of data accesses — padding
    - Only work on limited amount of data structures during a period of time — split the loop (loop fission)
    - Traversal data to reduce the number of blocks touched during a period of time — loop interchange
- Capacity misses — the miss due to the working set size surpasses the **capacity**
  - Reduce the working set size! — tiling
  - Make data size smaller — condense data layout

Demo revisited

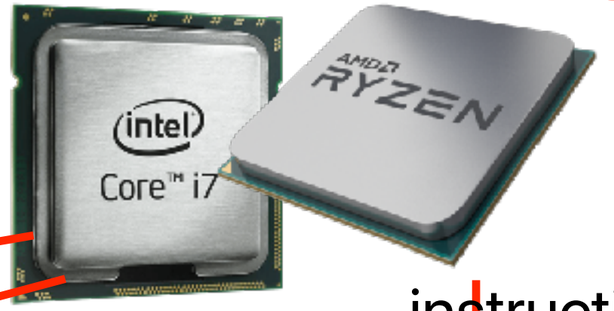



```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    return 0;
}
```

Virtual memory

- An **abstraction** of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into "**pages**"

Virtual memory



instruction
0x0

data
0x80000000

0x0

$2^{64}-1$

Virtual Memory Space

instruction
0x0

data
0x80008000

$2^{64}-1$

Virtual Memory Space

Program

Instructions		Data	
0f00bb27		00c2e800	
509cbd23		00000008	
00005d24		00c2f000	
0000bd24		00000008	
2ca422a0		00c2f800	
130020e4		00000008	
00003d24		00c30000	
2ca4e2b3		00000008	

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008

Memory

Program

Instructions		Data	
0f00bb27		00c2e800	
509cbd23		00000008	
00005d24		00c2f000	
0000bd24		00000008	
2ca422a0		00c2f800	
130020e4		00000008	
00003d24		00c30000	
2ca4e2b3		00000008	

Virtual memory

This approach is called demand paging + swapping

Program A

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

(Data)

Processor

Program B

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

(Data)

PC for A

PC for A

load

0x0

0f00bb27
509cbd23
00005d24
0000bd24

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

00c2f800
00000008
00c30000
00000008

Virtual Memory

$2^{64}-1$

page

PC for B

PC for B

load

load

0x0

0f00bb27
509cbd23
00005d24
0000bd24

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

00c2f800
00000008
00c30000
00000008

Virtual Memory

$2^{64}-1$

00c2f800
00000008
00c30000
00000008

00c2e800
00000008
00c2f000
00000008

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

0f00bb27
509cbd23
00005d24
0000bd24

2ca422a0
130020e4
00003d24
2ca4e2b3

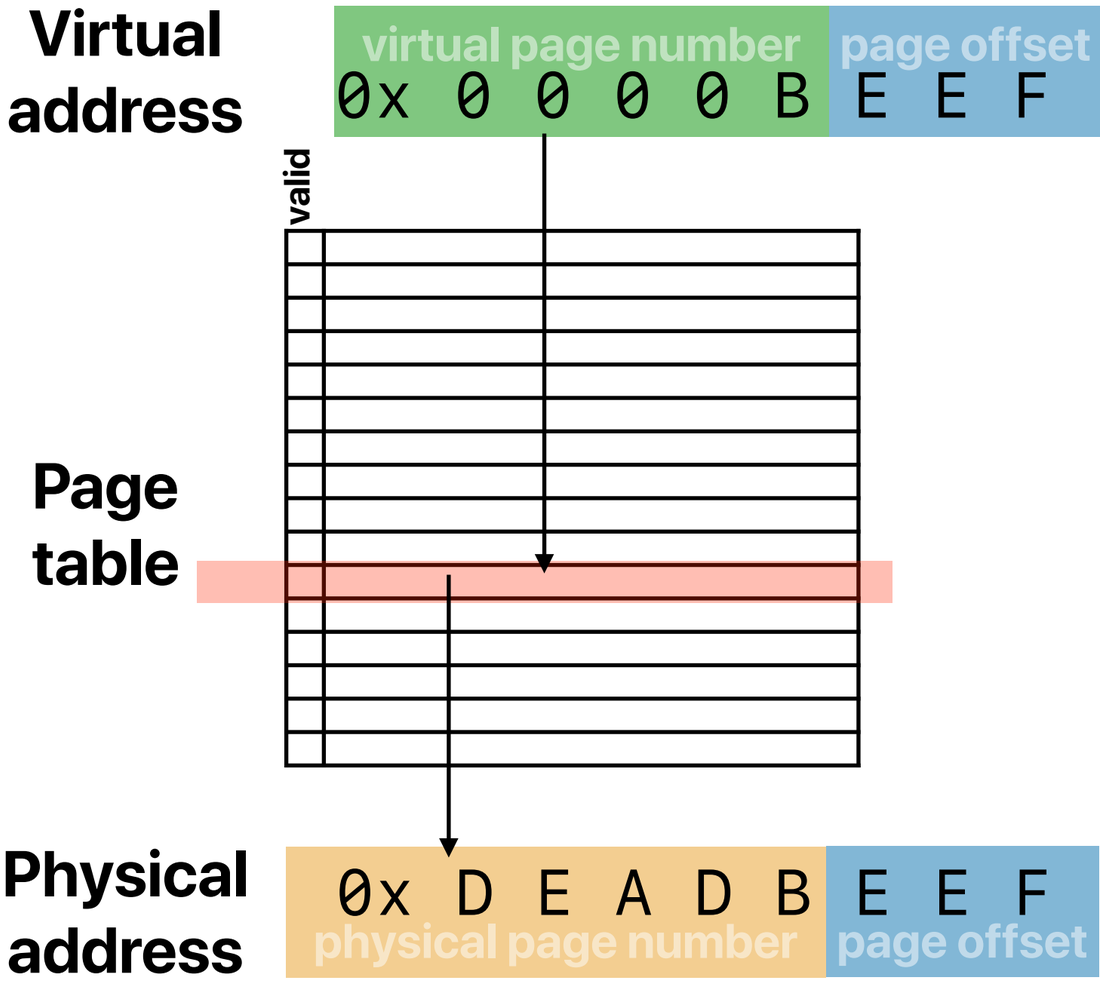
Physical Memory

0f00bb27
509cbd23
00005d24
0000bd24

Swap

Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into “pages”
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS



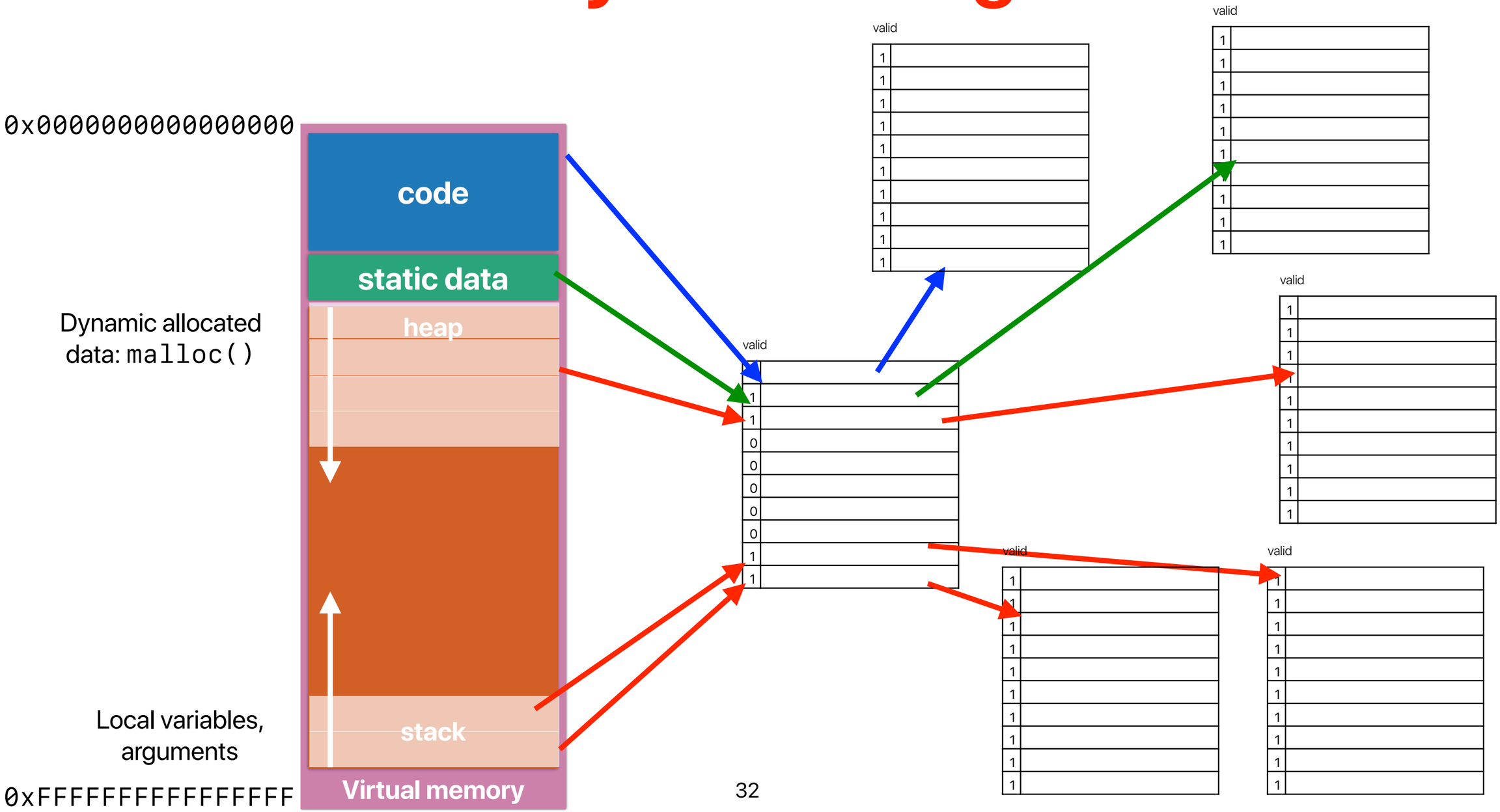
Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — 2^{20} Bytes
  B. GB — 2^{30} Bytes
  C. TB — 2^{40} Bytes
  D. PB — 2^{50} Bytes
  E. EB — 2^{60} Bytes

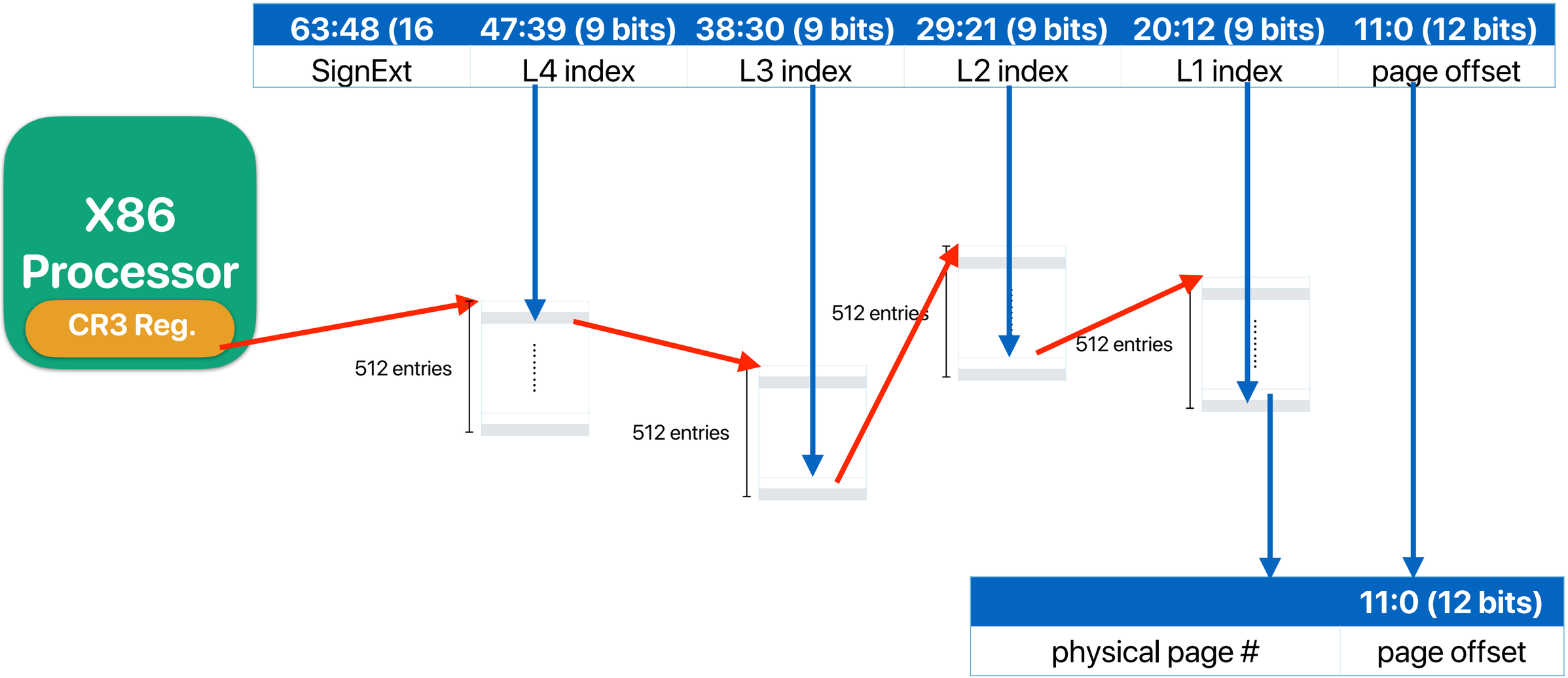
$$\frac{2^{64}\ Bytes}{4\ KB} \times 8\ Bytes = 2^{55}\ Bytes = 32\ PB$$

If you still don't know why — you need to take CSE120

Do we really need a large table?

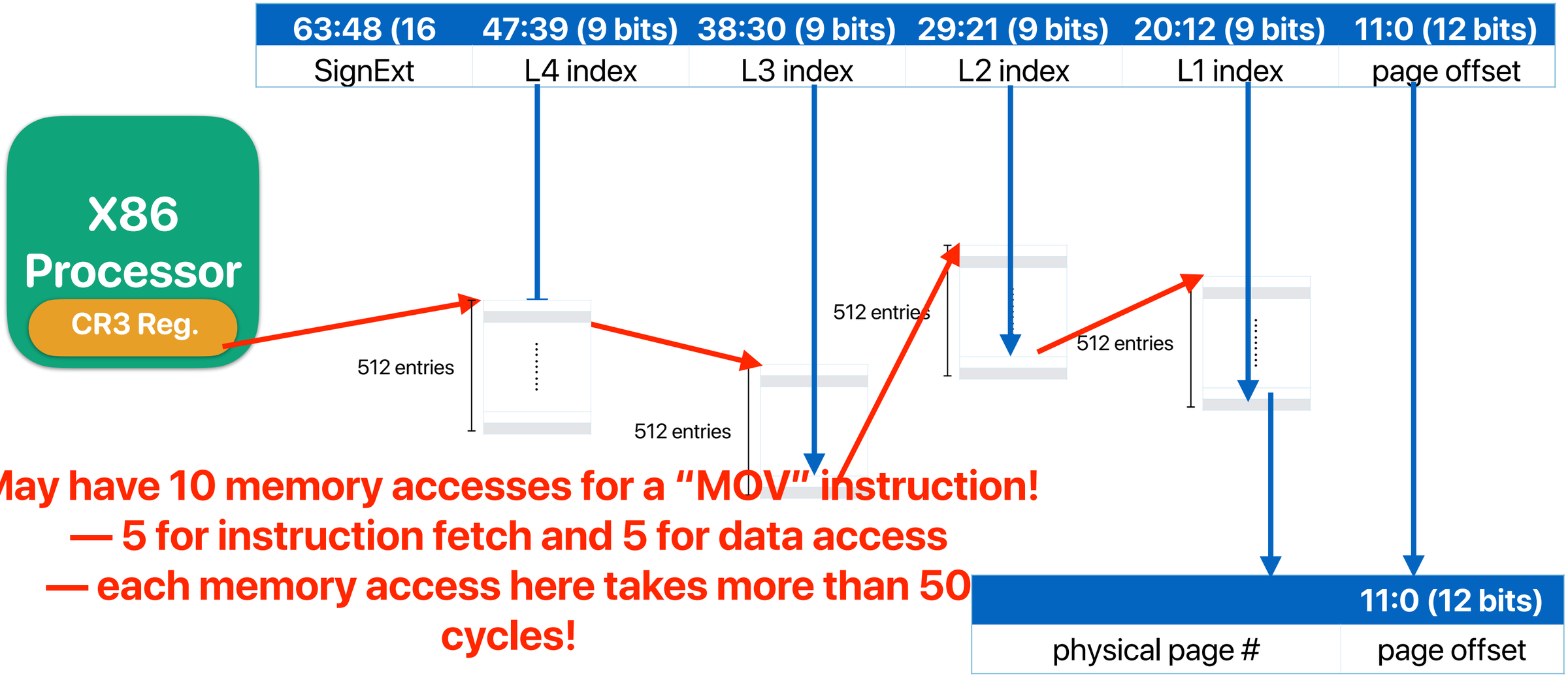


Address translation in x86-64



Translation Caching: Skip, Don't Walk (the Page Table)
Thomas W. Barr, Alan L. Cox, Scott Rixner

Address translation in x86-64

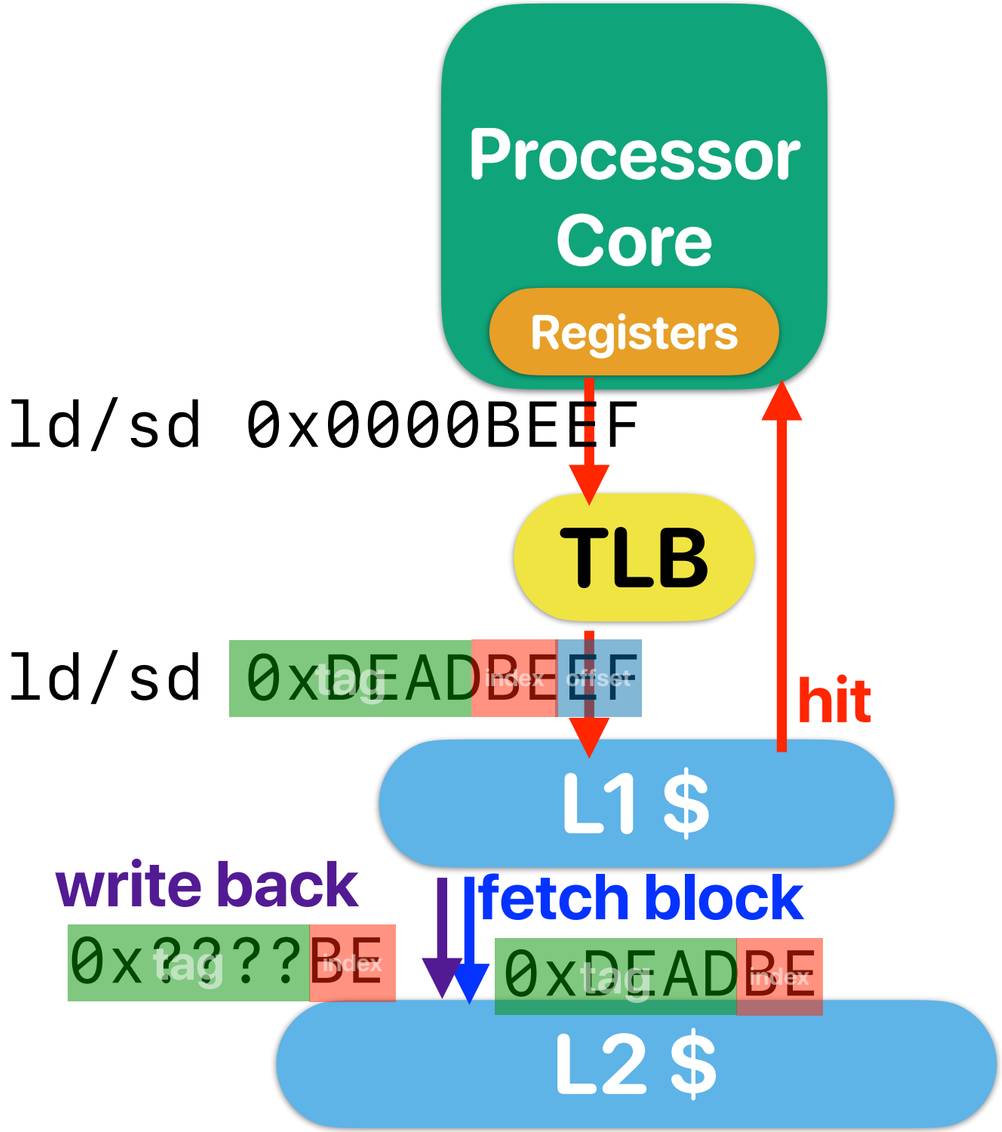


May have 10 memory accesses for a "MOV" instruction!

— 5 for instruction fetch and 5 for data access

— each memory access here takes more than 50 cycles!

TLB: Translation Look-aside Buffer



- TLB — a small SRAM stores frequently used page table entries
- Good — A lot faster than having everything going to the DRAM
- Bad — Still on the critical path

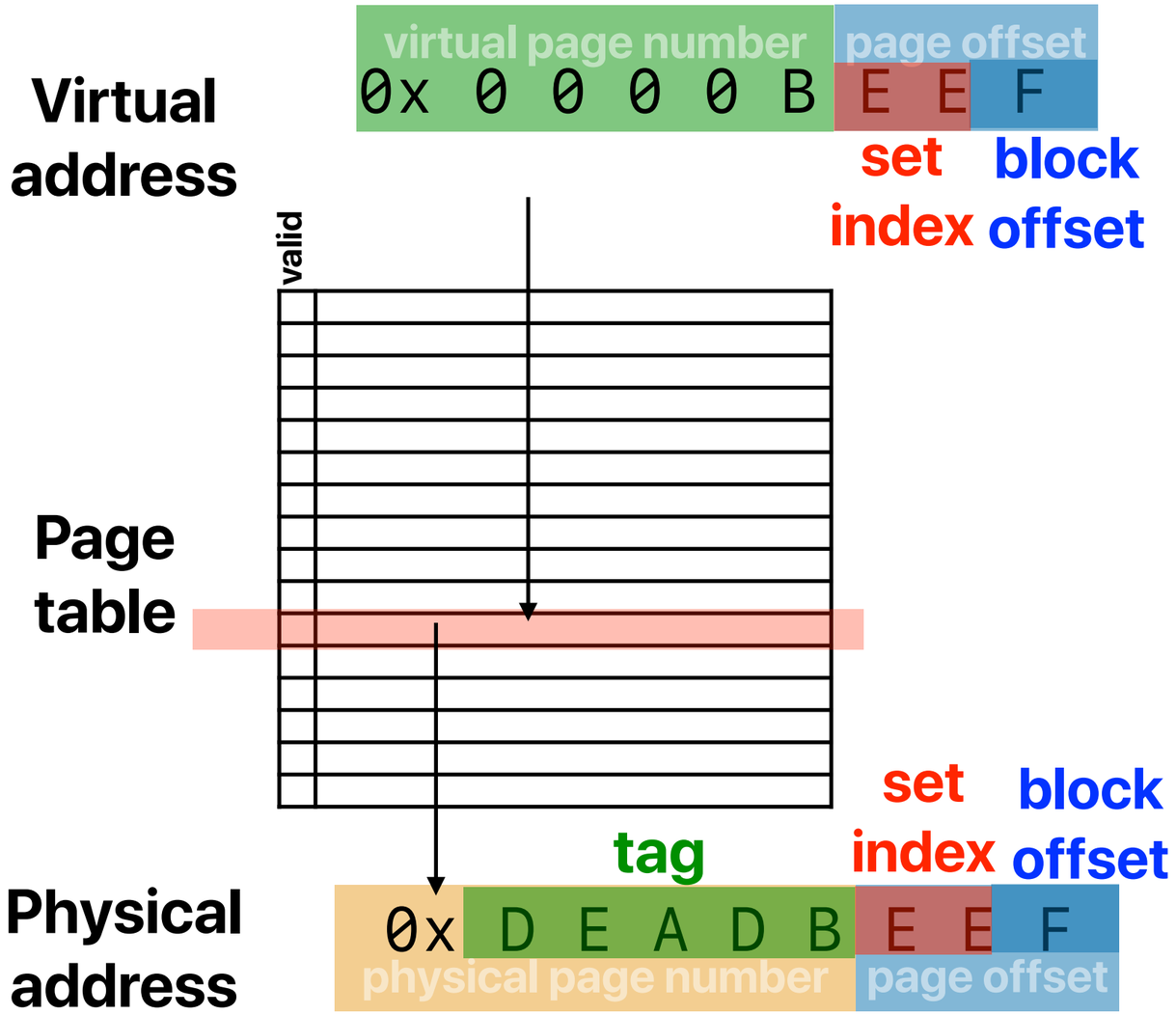
Virtually indexed, physically tagged cache

- Can we find physical address directly in the virtual address

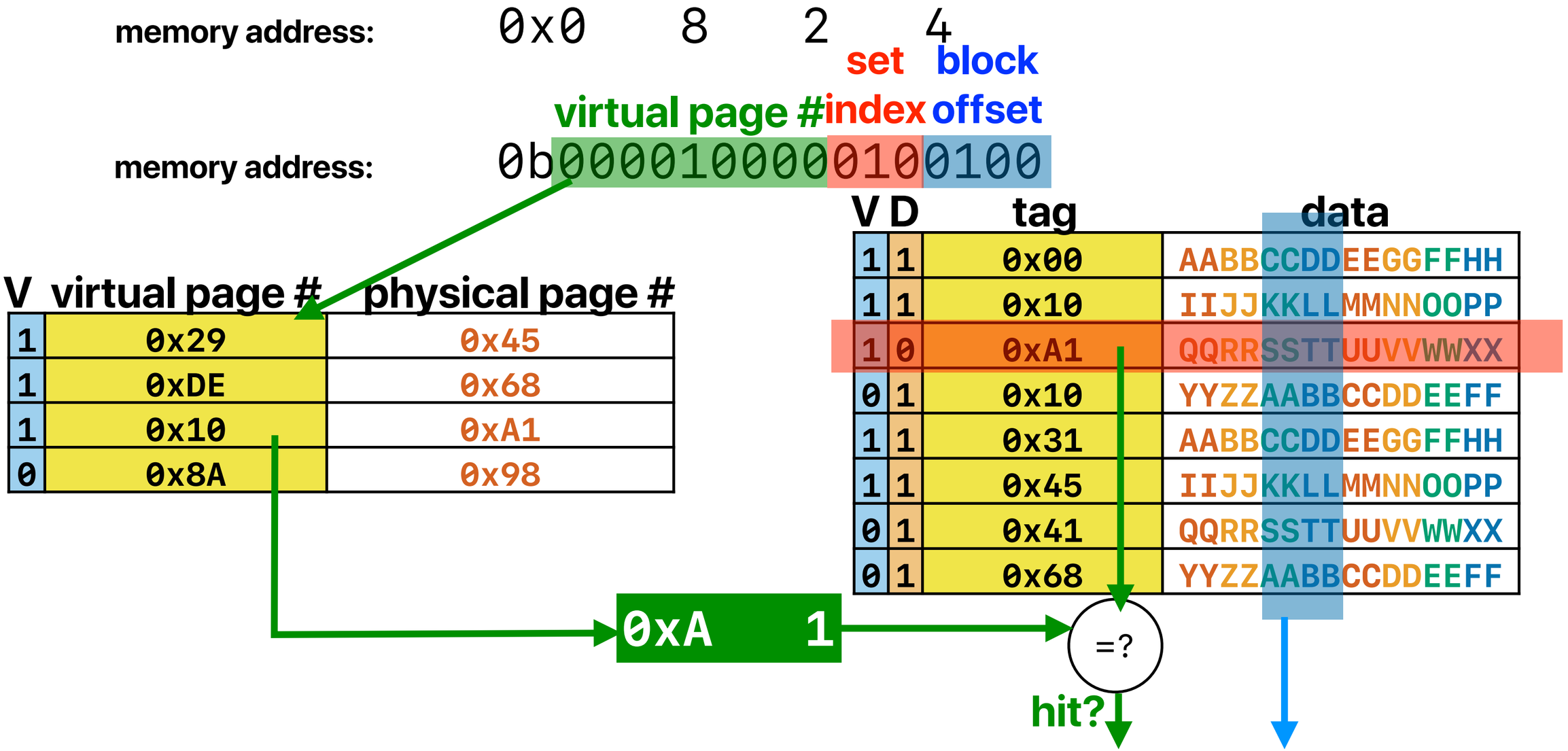
  — Not everything — but the page offset isn't changing!

- Can we indexing the cache using the "partial physical address"?

  — Yes — Just make set index + block set to be exactly the page offset



Virtually indexed, physically tagged cache



Virtually indexed, physically tagged cache

- If page size is 4KB —

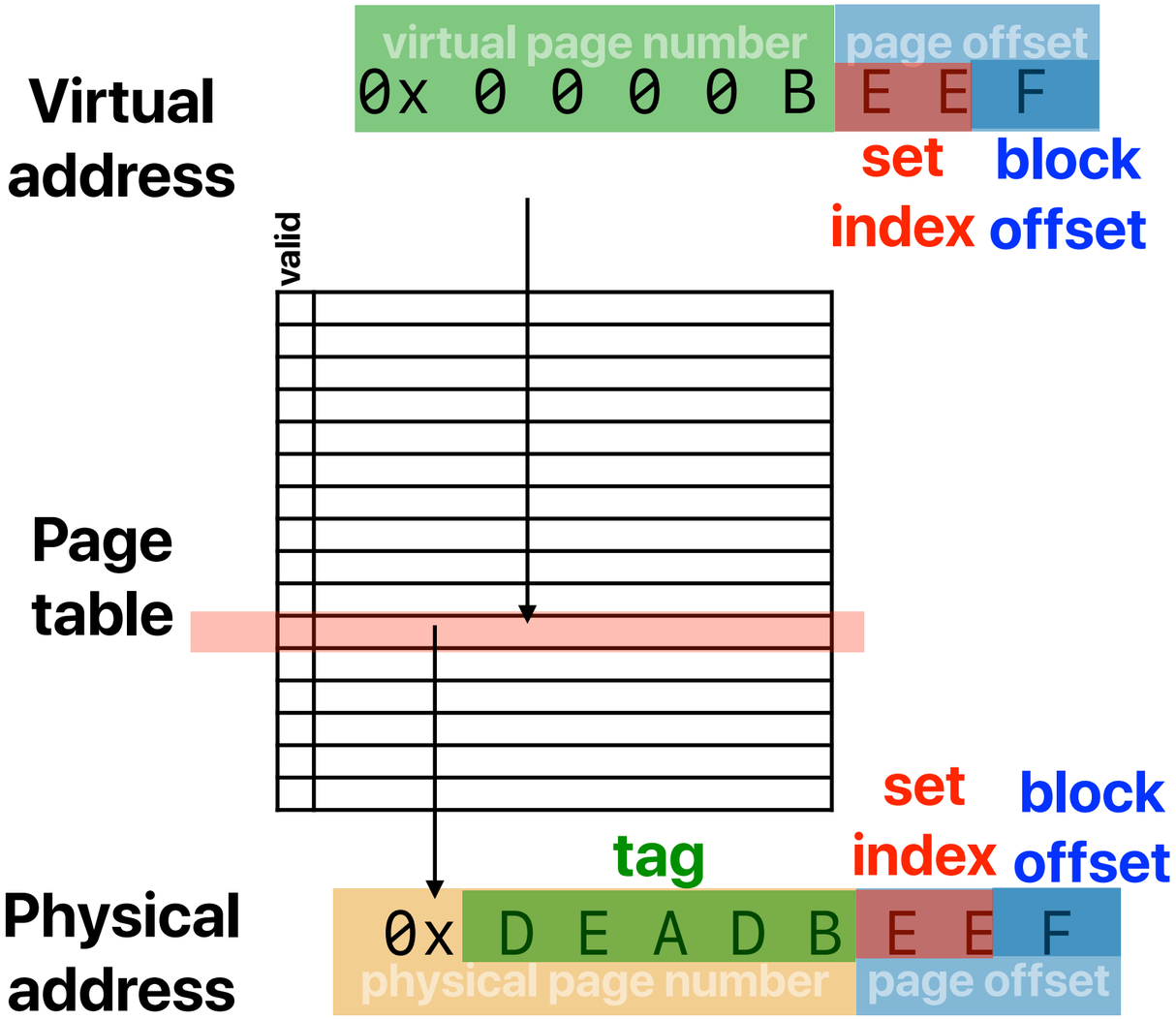
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

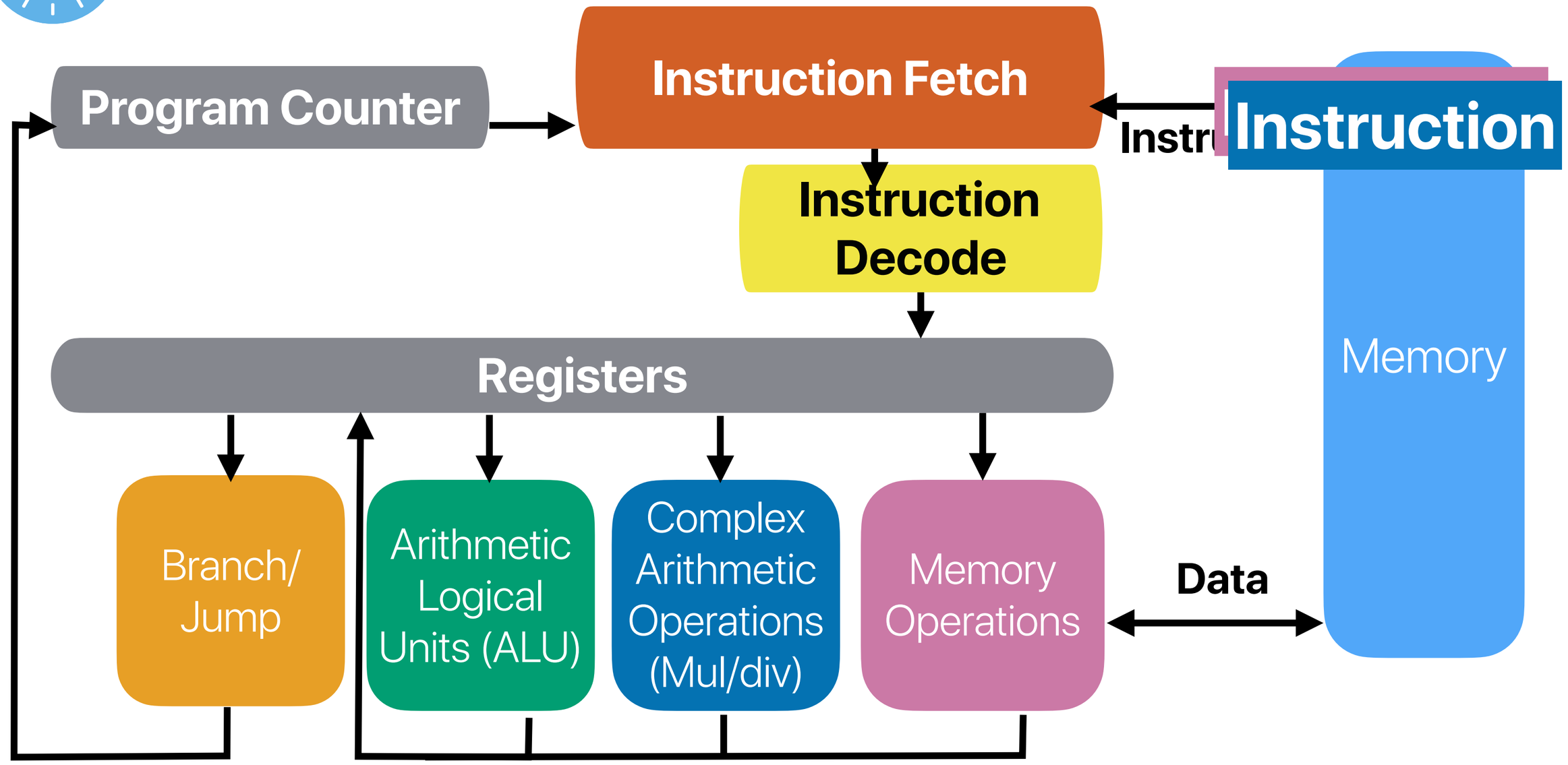
$$C = A \times 2^{12}$$

$$if\ A = 1$$

$$C = 4KB$$



Within a cycle…

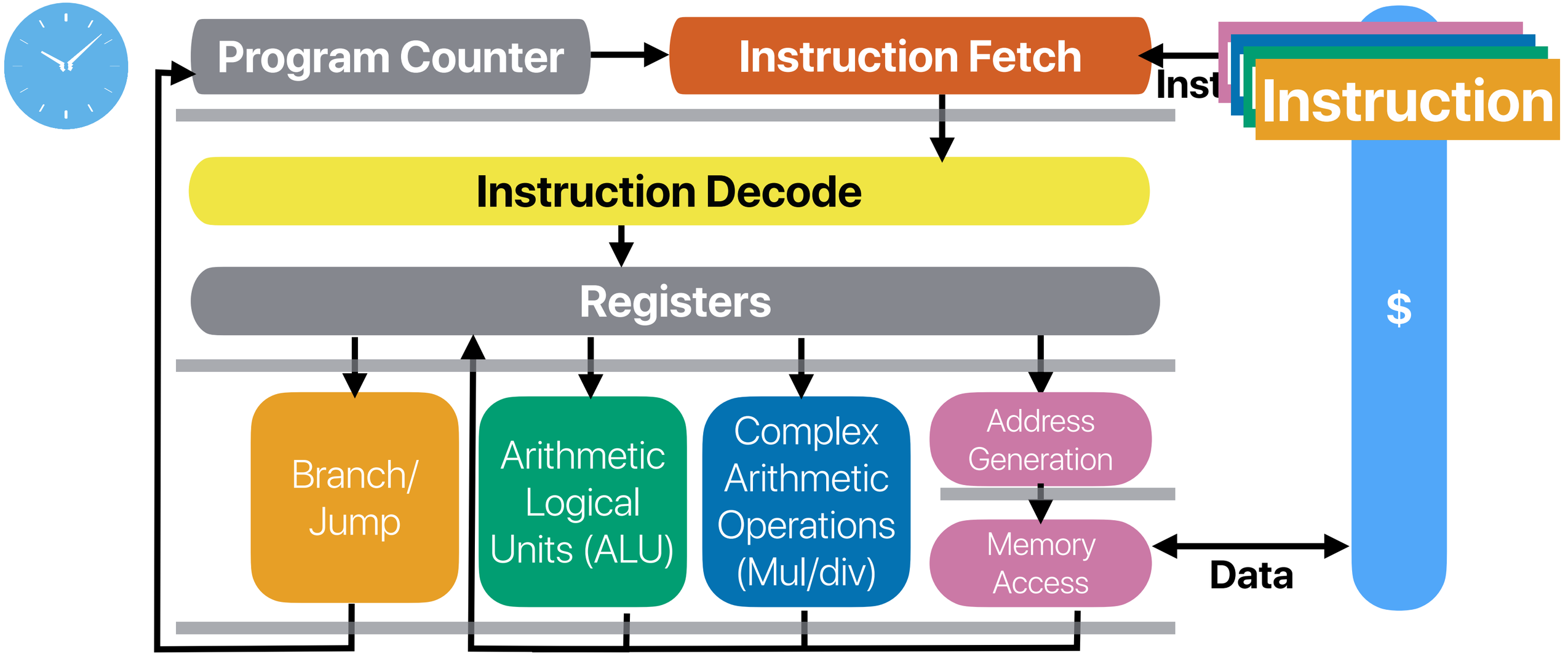


One instruction at time in a processor...

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl %esi, %esi
movl   $10, %edx
```

Cycle time is long, lots of functional units sitting idle!

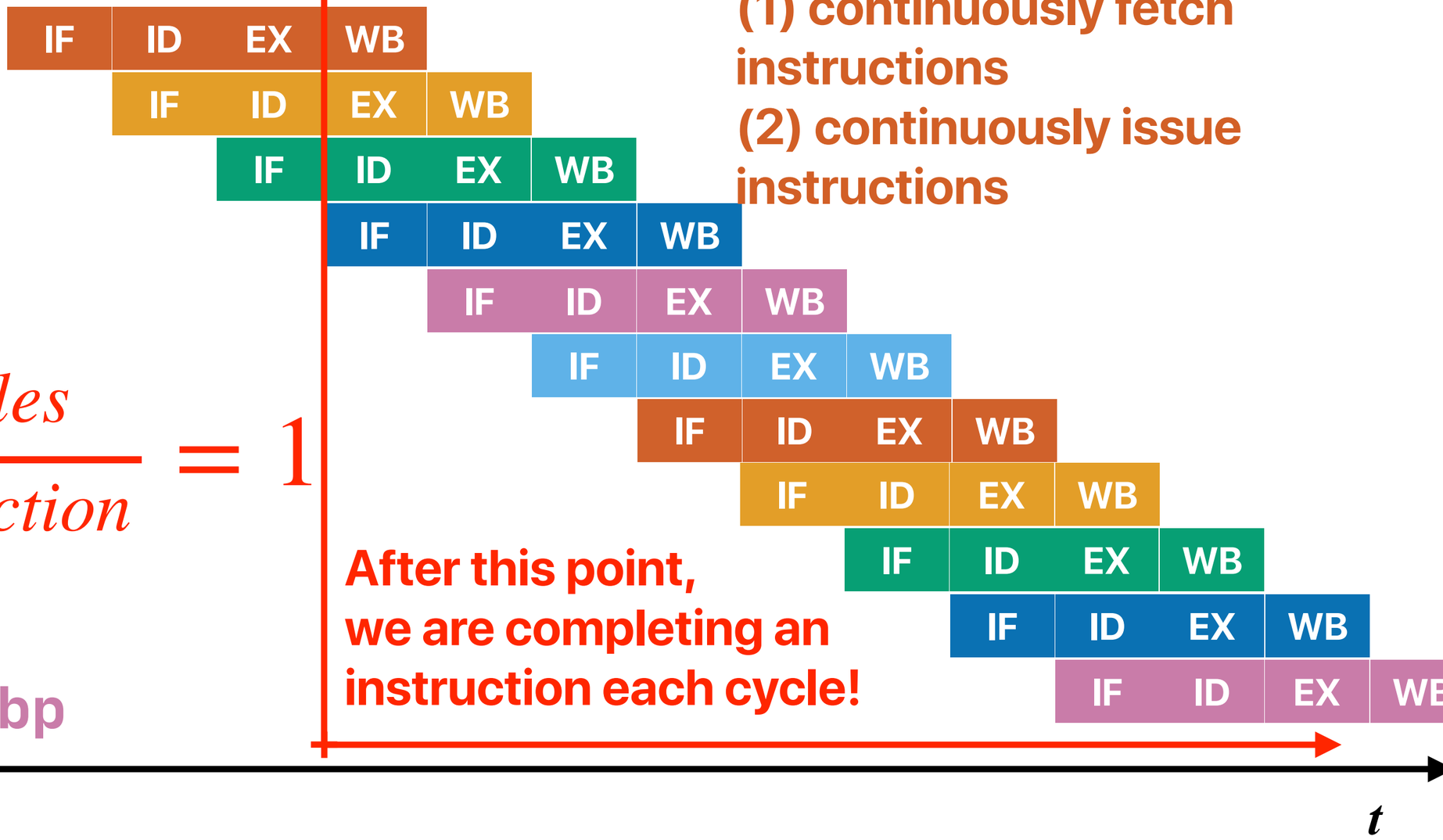
Pipelined execution



Pipelining

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl %esi, %esi
movl   $10, %edx
pushq %r12
pushq %rbp
pushq %rbx
subq   $8, %rsp
addl   %rsi, %rdi
movslq  %eax, %rbp
```

$$\frac{Cycles}{Instruction} = 1$$



However, if we want to keep generate results every cycle, we must
(1) continuously fetch instructions
(2) continuously issue instructions

After this point, we are completing an instruction each cycle!

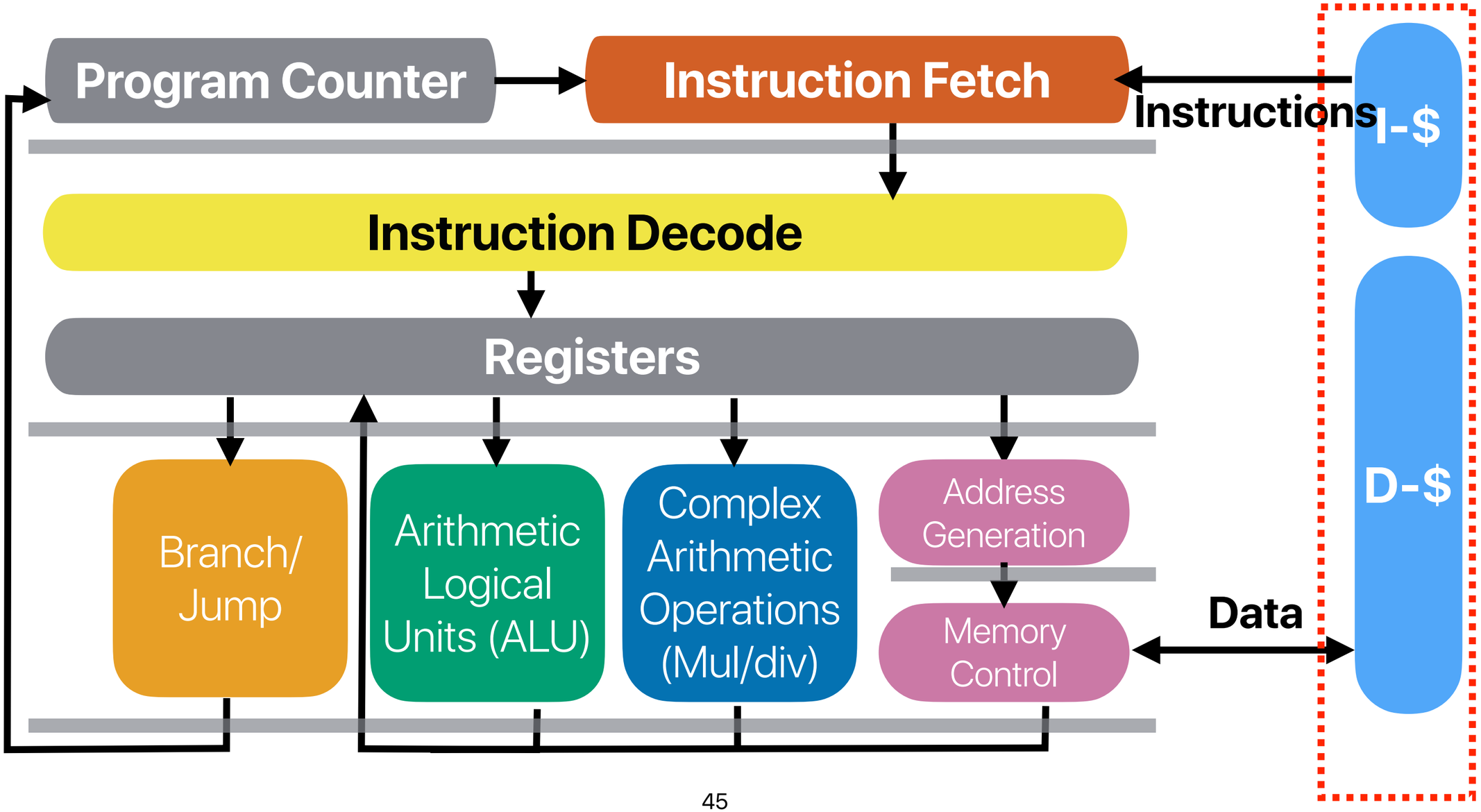
But we cannot! — three pipeline hazards

- Structural hazards — an instruction cannot be issued due to resource conflicts — two instructions in the pipeline try to use the same piece of hardware simultaneously
- Control hazards — we cannot continuously fetch instructions because the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction cannot be issued the instruction is depending on a yet-to-generate result from a prior instruction

Solutions/work-around of pipeline hazards

- Structural
  - Hardware — Stall
  - Hardware — More read/write ports
  - Hardware — Split hardware units (e.g., instruction/data caches)

Split L1-$


Program Counter
Instruction Fetch
Instructions
I-$
Instruction Decode
Registers
Branch/ Jump
Arithmetic Logical Units (ALU)
Complex Arithmetic Operations (Mul/div)
Address Generation
Memory Control
Data
D-$

Control Hazards

The frequency/cost of branches

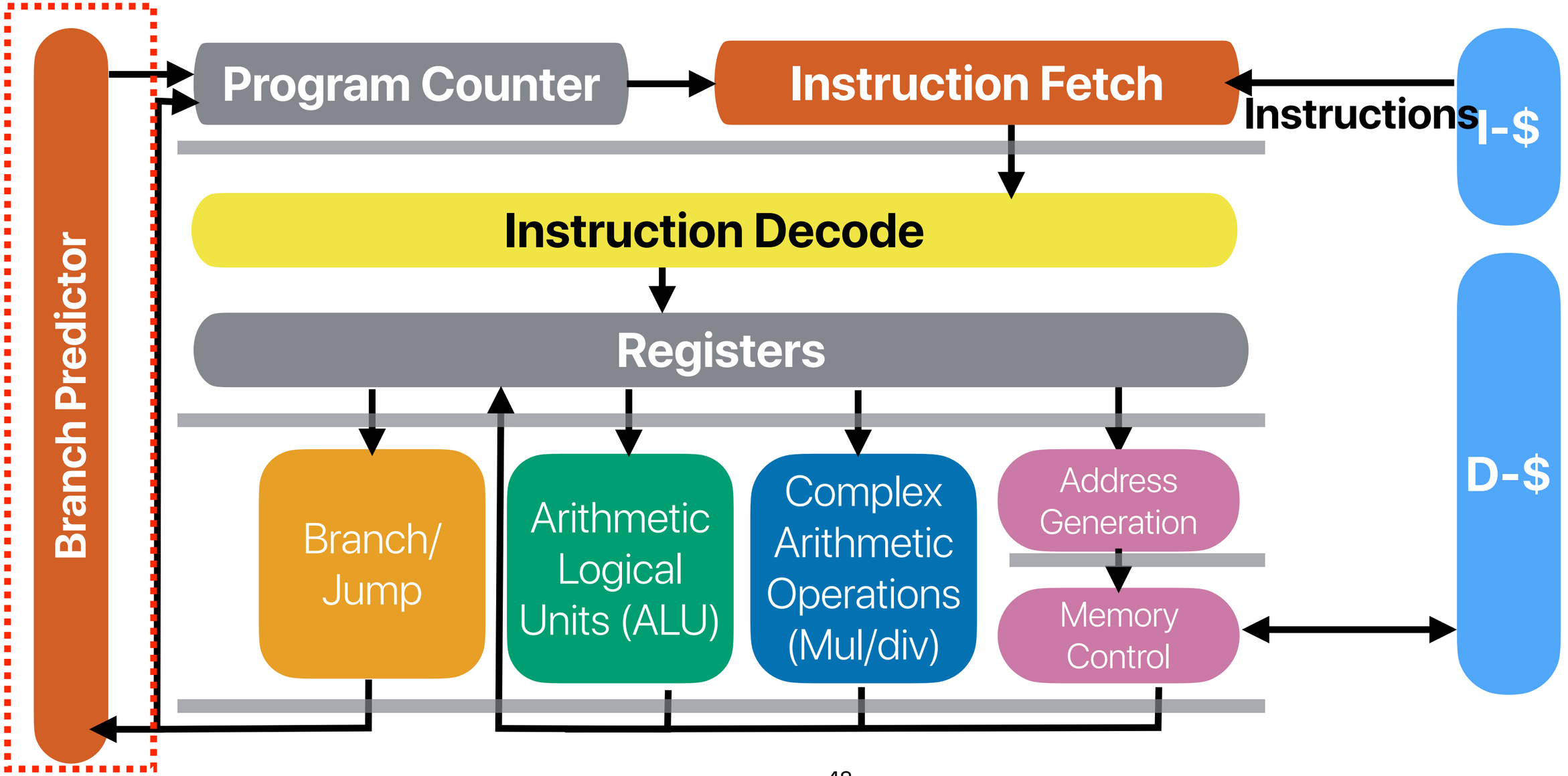
Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	5%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

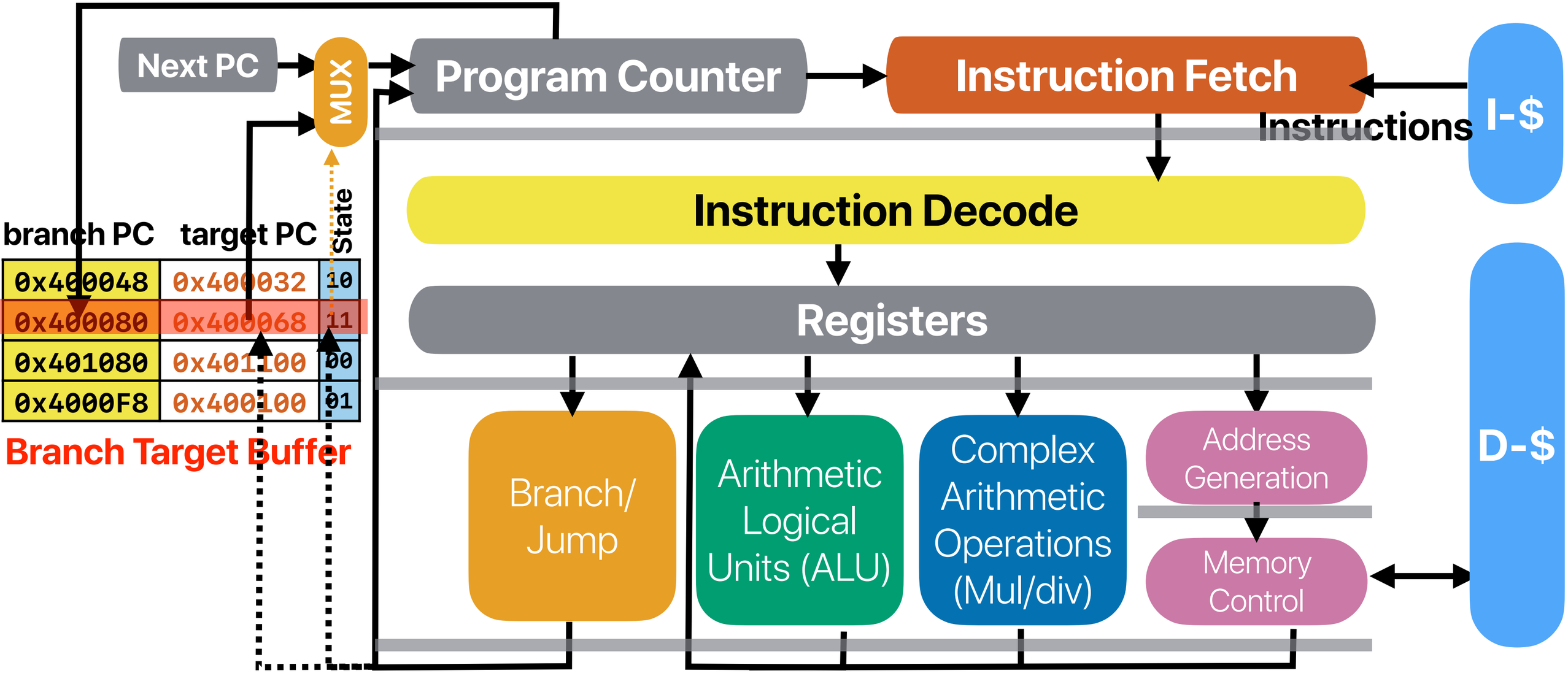
	index	size	iterations	sorted	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate	L1_dcache_misses	L1_dcache_accesses	branches	branch_misses
0	0	262144	1000	0	3017116452	2981316868	0.988135	0.194469	0.579775	0.002048	4820945	2353843792	525398457	**68000748**
1	1	262144	1000	1	3012182399	685457926	0.227562	0.195472	0.133988	0.001634	3844384	2352029090	524540744	**4945**

```
Diff_Misses = 68000748-4945
Diff_Cycles = 2981316868-685457926
print(Diff_Cycles/Diff_Misses) = 33.764715478100904
```

Microprocessor with a “branch predictor”



Detail of a basic dynamic branch predictor

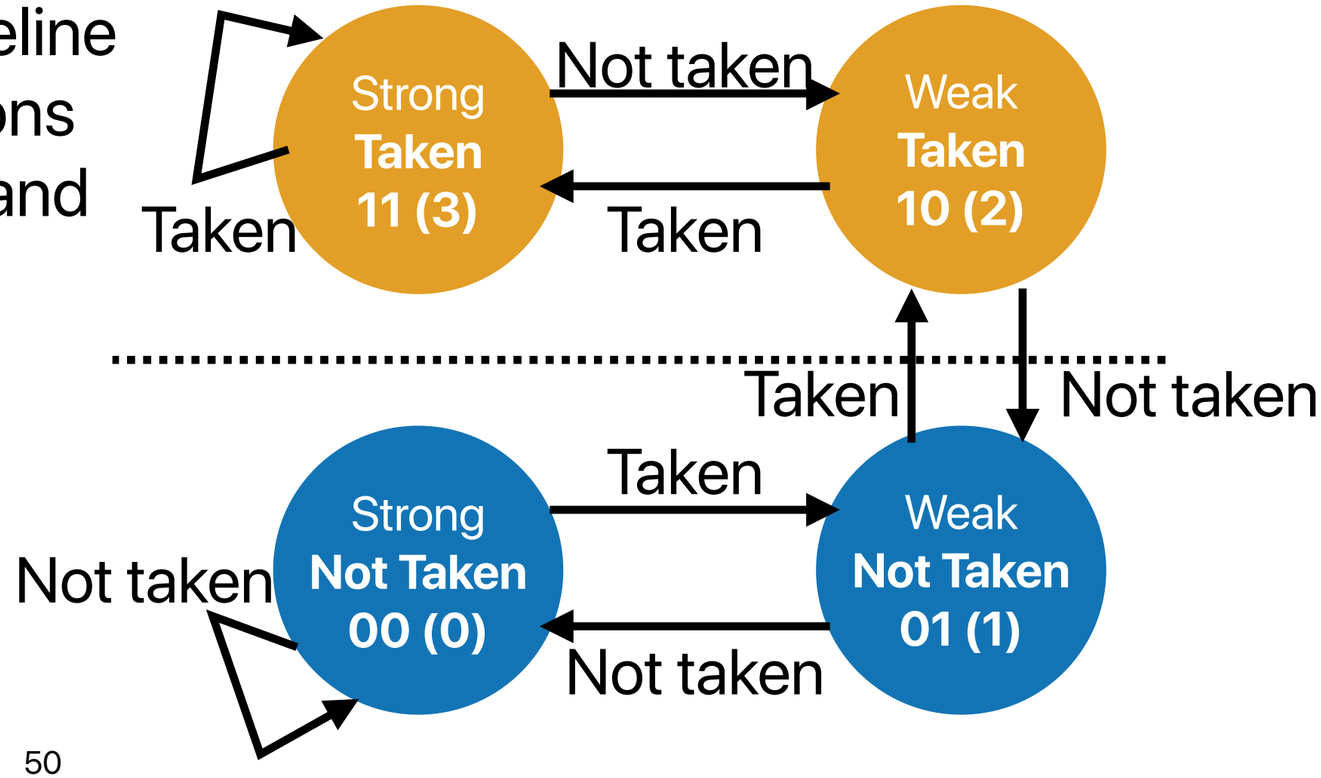


2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

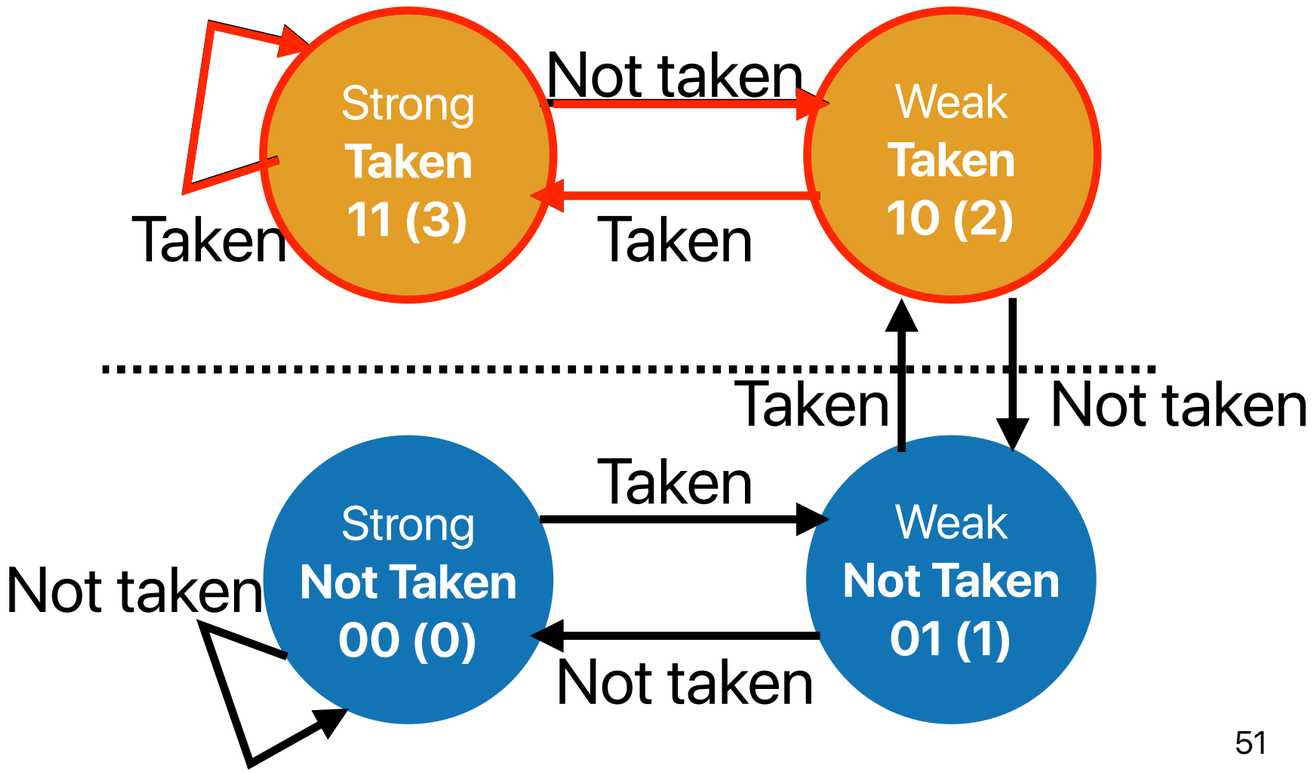
branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

Predict Taken



2-bit local predictor

```
i = 0;
do {
     sum += a[i];
} while(++i < 10);
```



i	state	predict	actual
1	10	T	T
2	11	T	T
3	11	T	T
4-9	11	T	T
10	11	T	NT

90% accuracy!

2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

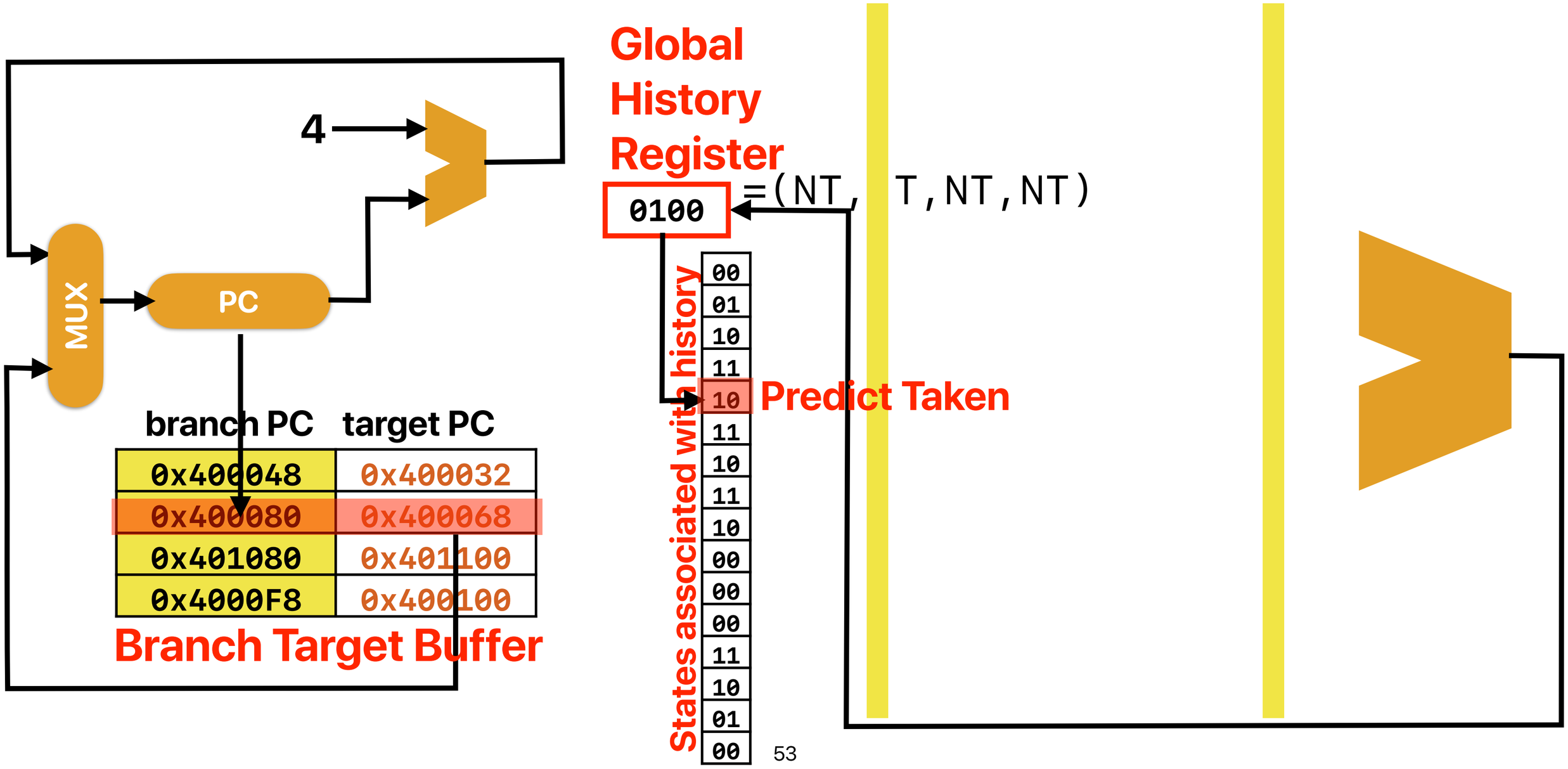
(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

For branch Y, almost 100%,
For branch X, only 50%

i	branch?	state	prediction	actual
0	X	00	NT	T
1	Y	00	NT	T
1	X	01	NT	NT
2	Y	01	NT	T
2	X	00	NT	T
3	Y	10	T	T
3	X	01	NT	NT
4	Y	11	T	T
4	X	00	NT	T
5	Y	11	T	T
5	X	01	NT	NT
6	Y	11	T	T
6	X	00	NT	T
7	Y	11	T	T

Global history (GH) predictor



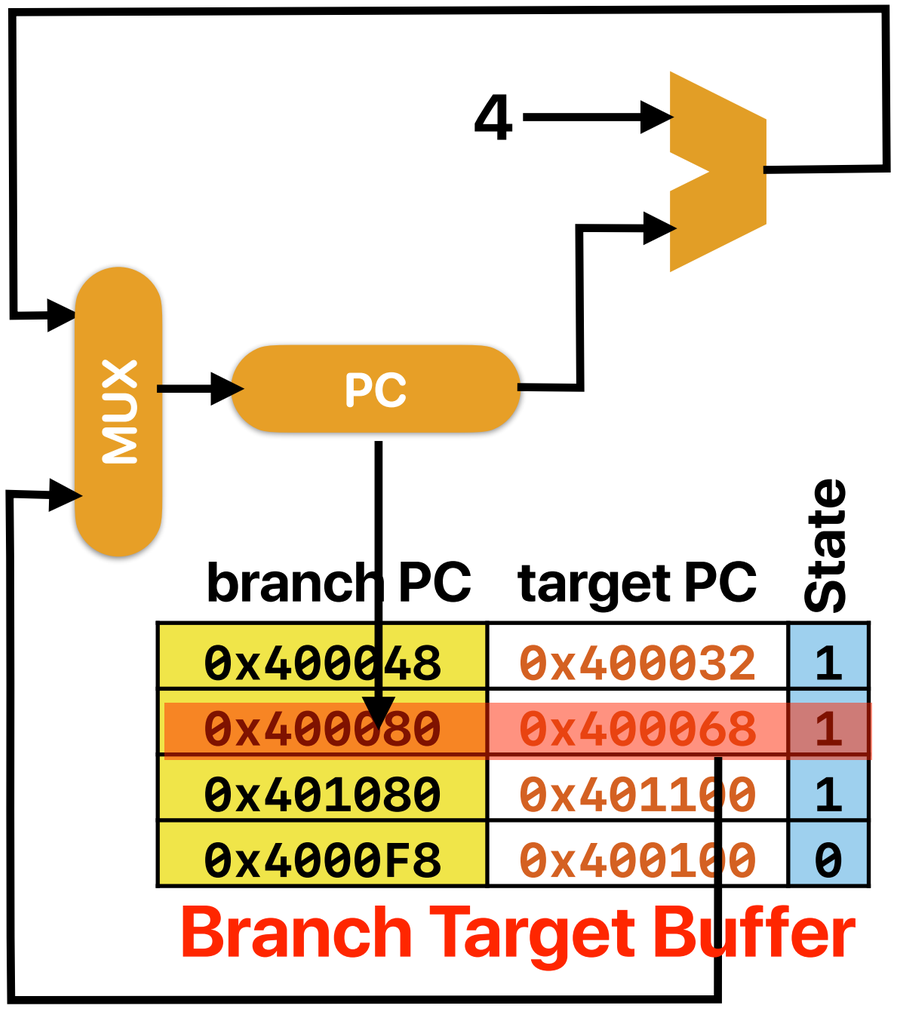
Performance of GH predictor

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

i	branch?	GHR	state	prediction	actual
0	X	000	00	NT	T
1	Y	001	00	NT	T
1	X	011	00	NT	NT
2	Y	110	00	NT	T
2	X	101	00	NT	T
3	Y	011	00	NT	T
3	X	111	00	NT	NT
4	Y	110	01	NT	T
4	X	101	01	NT	T
5	Y	011	01	NT	T
5	X	111	00	NT	NT
6	Y	110	10	T	T
6	X	101	10	T	T
7	Y	011	10	T	T
7	X	111	00	NT	NT
8	Y	110	11	T	T
8	X	101	11	T	T
9	Y	011	11	T	T
9	X	111	00	NT	NT
10	Y	110	11	T	T
10	X	101	11	T	T
11	Y	011	11	T	T

Near perfect after this

Tournament Predictor



Global History Register

0100

States associated with history

00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Local History Predictor

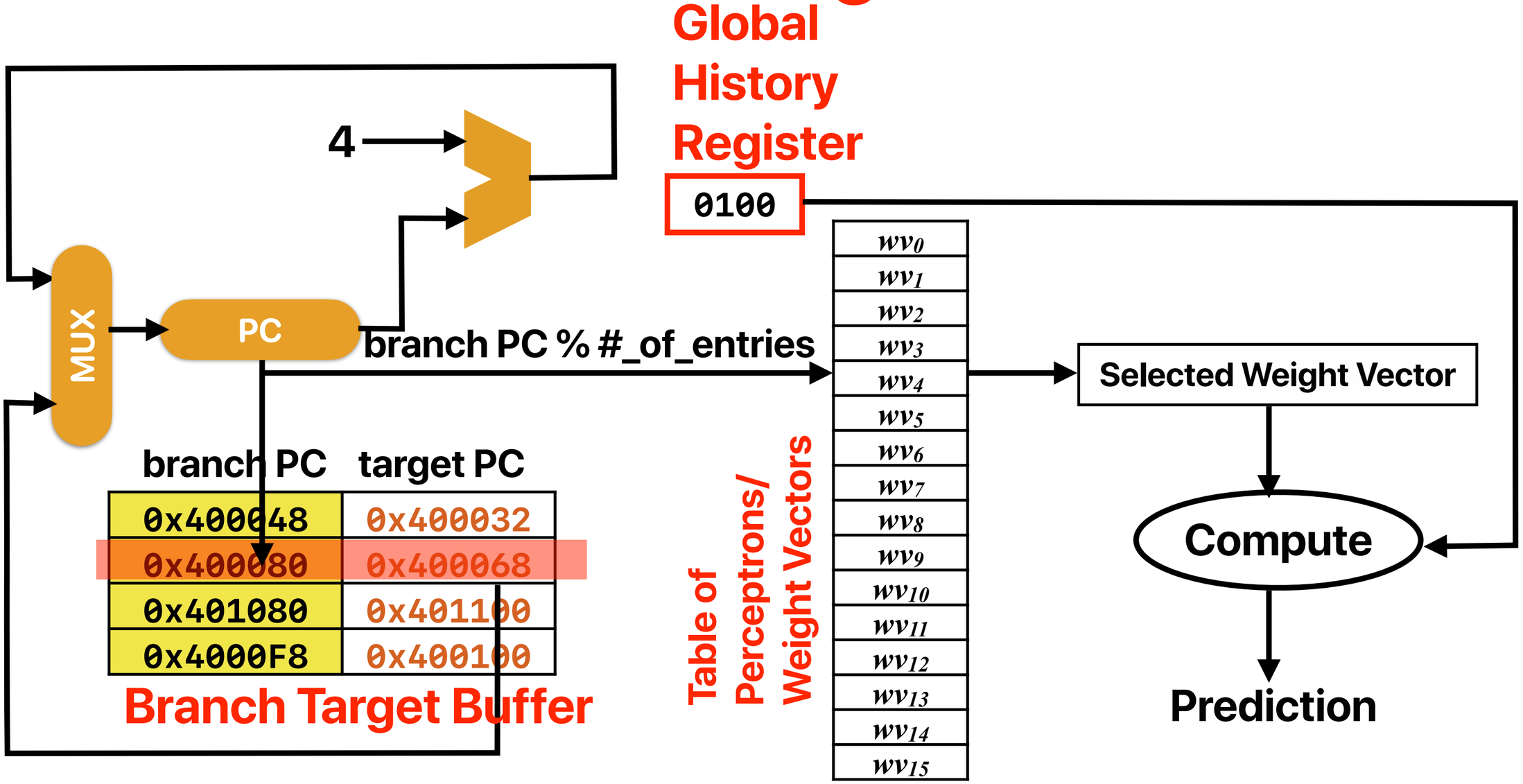
branch PC	local history
0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

Predict Taken

States associated with history

00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Predictor Organization



Don’t forget the “big idea”!

- 2-bit local
  - Each branch is considered “separately” — that’s why it’s **local**
  - 2-bit “counter” with each branch
    - The predictor will “stably” predict one direction if the pattern appears for **twice** or more **regularly**
- Global history
  - States are associate with the history of branch outcome, regardless of which branch generates the result — that’s why it’s global
  - It’s only effective if
    - The history pattern can be captured by the history registers
    - Say, if the branch outcome changes once every 10, but you have only 4-bit history, your predictor cannot learn it well
- Hybrid predictors
  - Tournament — choose either local or global based on which is more convincing for each branch
  - Perceptron — discounting the weights of irrelevant branch outcomes
  - If neither global nor local predictor works well for the code, hybrid predictors also unlikely to work well

Revisit the code

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

The **local** history

	0	1	2	3	4	5	6	7	8
X	T	NT	T	NT	T	NT	T	NT	T
Y		T	T	T	T	T	T	T	T

2-bit can at best get 50% righ

2-bit can get 100%

The **global** history

	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11
Branch	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y
Result	T	T	NT	T	T	T	NT	T	T	T	NT	T	T	T	NT	T	T	T	NT	T	T	T

4-bit history can make it work

Revisit the code (II)

```
i = 0;
do {
    if( i % 10 != 0) // Branch X, taken if i % 10 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

about the same

The **local** history

	0	1	2	3	4	5	6	7	8	9	10
X	T	NT	NT	NT	NT	NT	NT	NT	NT	NT	T
Y		T	T	T	T	T	T	T	T	T	T

2-bit can get 90%
2-bit can get 100%

The **global** history

	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11
Branch	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y
Result	T	T	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	T	T

history shorter than 18-bits cannot work for branch X when i=n*10

Branch predictor in modern processors

- Big take-aways: all modern processors have decent branch predictors
- The Intel Pentium MMX, Pentium II, and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.
- Global branch prediction is used in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors.
- Tournament predictor is used in DEC Alpha, AMD Athlon processors
- The AMD Ryzen multi-core processor's Infinity Fabric and the Samsung Exynos processor include a perceptron based neural branch predictor.

Demo revisited

- Why the sorting the array speed up the code despite the increased instruction count?

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

Hard to predict for most cases, no matter it's global or local

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
data	9	85	73	50	87	61	88	43	87	23	93	68	80	60	23	14	62	92	31	68	8	60	73	87	65

Threshold	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
57	T	NT	NT	T	NT	NT	NT	T	NT	T	NT	NT	NT	NT	T	T	NT	NT	T	NT	T	NT	NT	NT	NT
73	T	NT	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	T	T	T	NT	T	T	T	T	NT	NT	T
98	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT	NT
61	T	NT	NT	T	NT	NT	NT	T	NT	T	NT	NT	NT	T	T	T	NT	NT	T	NT	T	T	NT	NT	NT
44	T	NT	NT	NT	NT	NT	NT	T	NT	T	NT	NT	NT	NT	T	T	NT	NT	T	NT	T	NT	NT	NT	NT

Demo revisited

- Why the performance is better when option is not "0"
  - ① The amount of dynamic instructions needs to execute is a lot smaller
  - ② The amount of branch instructions to execute is smaller
  - ③ The amount of branch mis-predictions is smaller
  - ④ The amount of data accesses is smaller

A. 0
B. 1
C. 2
D. 3
E. 4

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i)
        if (data[i] >= threshold)
            sum ++;
    }
}
```

branch X

	Without sorting	With sorting
The prediction accuracy of X before threshold	50%	100%
The prediction accuracy of X after threshold	50%	100%

Solutions/work-around of pipeline hazards

- Structural
  - Hardware — Stall
  - Hardware — More read/write ports
  - Hardware — Split hardware units (e.g., instruction/data caches)
- Control
  - Hardware — Stalls
  - Hardware — Branch predictions
  - Programmer — Sorting data to facilitate branch predictions
  - Programmer — replacing branch with equivalent statements (e.g., lookup tables)

What do you need to execution an instruction?

- Whenever the instruction is fetched/decoded — put decoded instruction somewhere — control hazards resolved
- Whenever the target functional unit is available — structural hazards resolved
- Whenever the inputs are ready — **all data dependencies are resolved**

Data Hazards

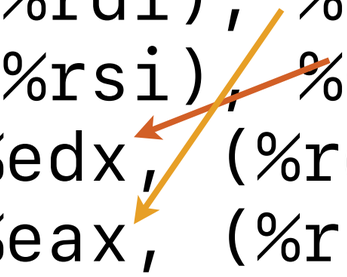
Data hazards

- An instruction currently in the pipeline cannot receive the "logically" correct value for execution
- Data dependencies
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline
- Data hazard
  - The awkward situation that the pipeline cannot make progress because of data dependency
  - A data dependency does not necessarily lead to data hazard

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



A. 1
B. 2
C. 3
D. 4
E. 5

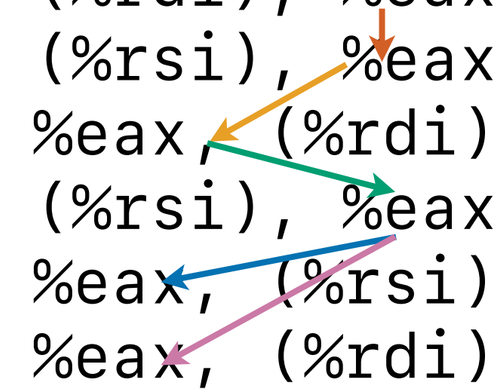
```
int temp = *a;
*a = *b;
*b = temp;
```

syntax of "movl":
movl %a, %b — %b = %a
if (%a) — memory[%a]

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

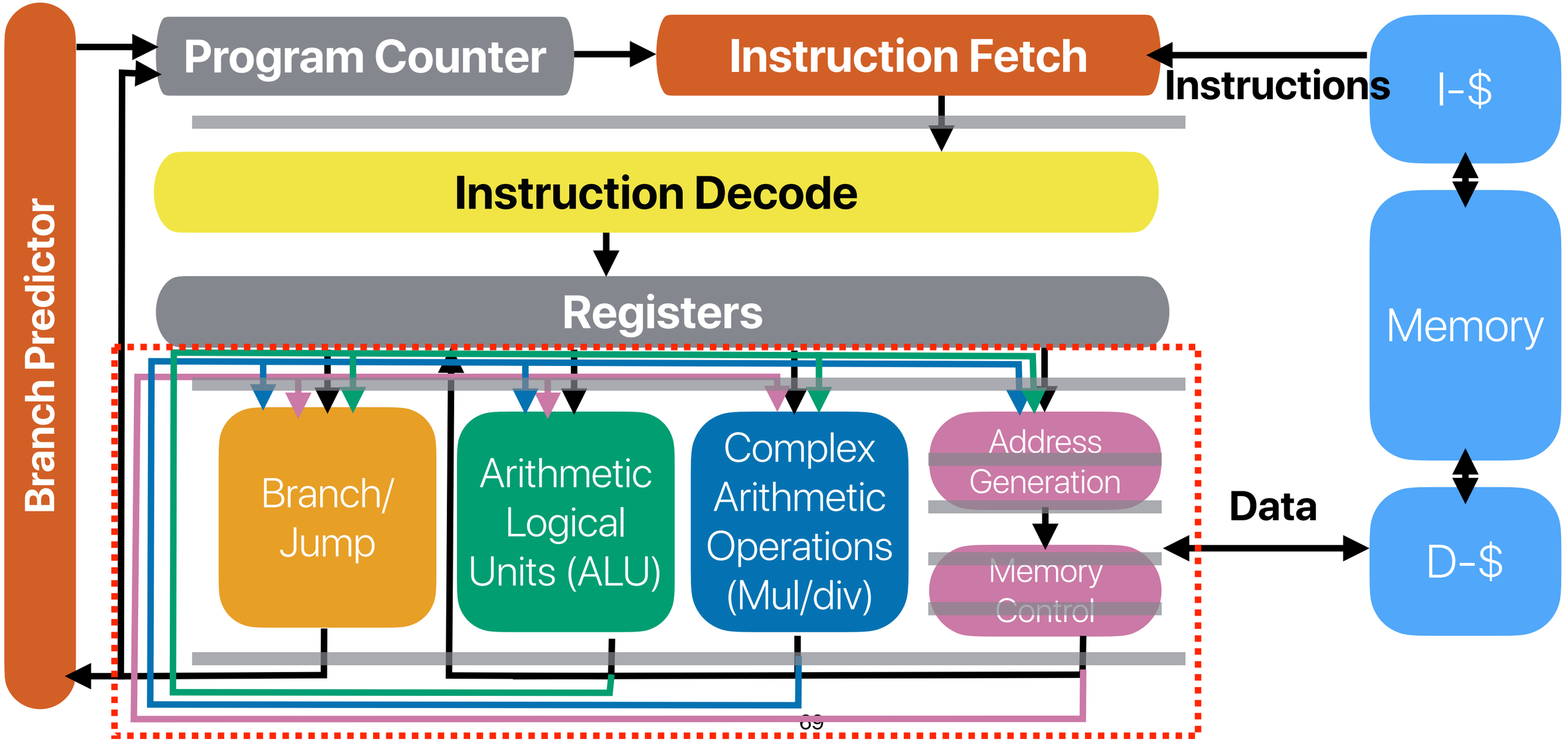


A. 1
B. 2
C. 3
D. 4
E. 5

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

syntax of "xorl":
xorl %a, %b — %b = %a ^%b

Data "forwarding"



Let's extend the example a bit…

```
for(i = 0; i < count; i++) {
    int64_t temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}
```

```
.L9:
①  movq    (%rdi,%rax), %rsi
②  movq    (%rcx,%rax), %r8
③  movq    %r8, (%rdi,%rax)
④  movq    %rsi, (%rcx,%rax)
⑤  addq    $8, %rax
⑥  cmpq    %r9, %rax
⑦  jne     .L9
⑧  movq    (%rdi,%rax), %rsi
⑨  movq    (%rcx,%rax), %r8
⑩  movq    %r8, (%rdi,%rax)
⑪  movq    %rsi, (%rcx,%rax)
⑫  addq    $8, %rax
⑬  cmpq    %r9, %rax
⑭  jne     .L9
```

	IF	ID	ALU/BR/AG	M1	M2	M3	M4/XORL	WB/Retire
1	(1)							
2	(2)	(1)						
3	(3)	(2)	(1)					
4	(4)	(3)	(2)	(1)				
5	(4)	(3)		(2)	(1)			
6	(4)	(3)			(2)	(1)		
7	(4)	(3)				(2)	(1)	
8	(4)	(3)					(2)	(1)
9	(5)	(4)	(3)					(2)
10	(6)	(5)	(4)	(3)				
11	(7)	(6)	(5)	(4)	(3)			
12	(8)	(7)	(6)		(4)	(3)		
13	(9)	(8)	(7)			(4)	(3)	
14	(10)	(9)	(8)				(4)	(3)
15	(11)	(10)	(9)	(8)				(4)
16	(11)	(10)		(9)	(8)			(5)
17	(11)	(10)			(9)	(8)		(6)
18	(11)	(10)				(9)	(8)	(7)
19	(11)	(10)					(9)	(8)
20	(12)	(11)	(10)					(9)
21	(13)	(12)	(11)	(10)				
22	(14)	(13)	(12)	(11)	(10)			
23		(14)	(13)	(12)	(11)	(10)		
24			(14)	(13)	(12)	(11)	(10)	

11 cycles for 7 instructions

CPI = 1.57

If we can do loop unrolling and reorder — we can do this!

```
for(i = 0; i < count; i++) {
    int64_t temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}
```

Compiler can only do this when it's 100% for sure count is always an even number! — loop unrolling

Compilers are limited by the number of registers available to the software!

```
movq    (%rdi,%rax), %rsi
movq    (%rcx,%rax), %r8
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
movq    (%rdi,%rax), %rsi
movq    (%rcx,%rax), %r8
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
```

```
.L9:
①  movq    (%rcx,%rax), %r8
②  movq    (%rdi,%rax), %rsi
③  addq    $8, %rax
④  movq    %r8, -8(%rdi,%rax)
⑤  movq    %rsi, -8(%rcx,%rax
⑥  movq    (%rcx,%rax), %r8
⑦  movq    (%rdi,%rax), %rsi
⑧  cmpq    %r9, %rax
⑨  jne     .L9
⑩  addq    $8, %rax
⑪  movq    %r8, -8(%rdi,%rax)
⑫  movq    %rsi, -8(%rcx,%rax
⑬  cmpq    %r9, %rax
⑭  jne     .L9
```

	IF	ID	ALU/BR/AG	M1	M2	M3	M4/XORL	WB/Retire
1	(1)							
2	(2)	(1)						
3	(3)	(2)	(1)					
4	(4)	(3)	(2)	(1)				
5	(5)	(4)	(3)	(2)	(1)			
6	(6)	(5)			(2)	(1)		
7	(7)	(6)	(4)				(2)	(1)
8	(8)	(7)	(5)	(4)				(2)
9	(9)	(8)	(6)	(5)	(4)			(3)
10	(10)	(9)	(7)		(5)	(4)		
11	(11)	(10)	(8)	(7)		(5)	(4)	
12	(12)	(11)	(9)	(8)			(5)	(4)
13	(13)	(12)	(10)	(9)	(8)			(5)
14	(14)	(13)	(11)	(10)	(9)	(8)		(6)
15		(14)				(9)	(8)	(7)
16							(9)	(8)
17			(12)	(11)				(9)
18			(13)	(12)	(11)			(10)
19			(14)		(12)	(11)		
20						(12)	(11)	
21							(12)	(11)
22								(12)
23								(13)
24								(14)
25								

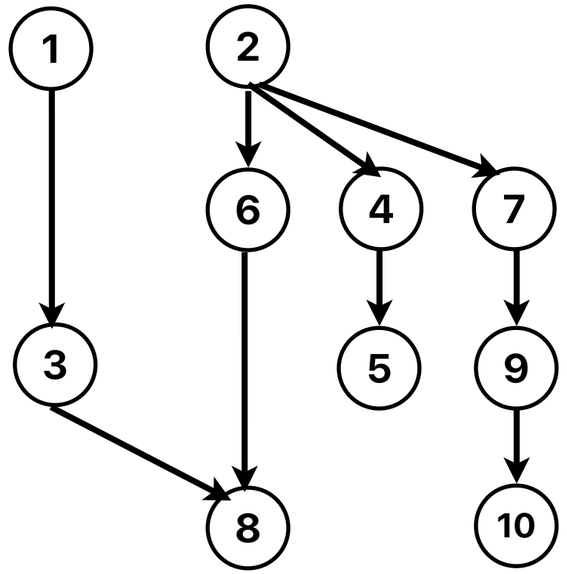
9 cycles for 7 instructions

CPI = 1.29

Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.

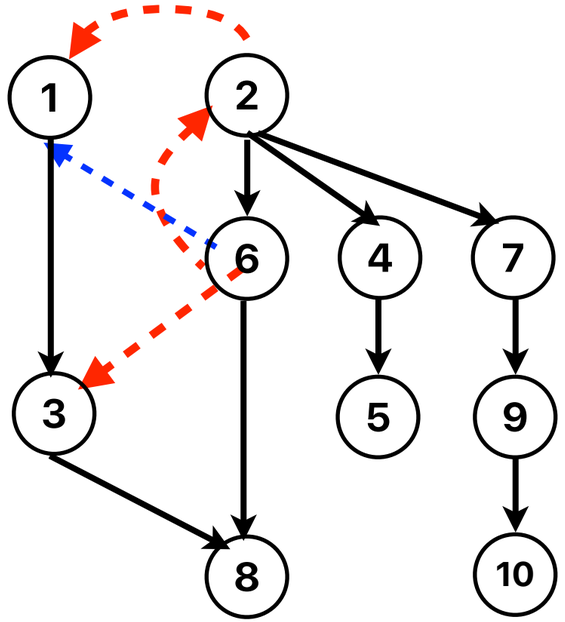
```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```



- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered

False dependencies

- We are still limited by **false dependencies**
- They are not "true" dependencies because they don't have an arrow in data dependency graph
  - WAR (Write After Read): a later instruction overwrites the source of an earlier one
    - 2 and 1, 6 and 2, 6 and 3, 9 and 5, 9 and 6, 9 and 8
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 6 and 1




```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```

Branch predictions + more registers!

```
①  movq    (%rdi,%rax), %rsi
②  movq    (%rcx,%rax), %r8
③  movq    %r8, (%rdi,%rax)
④  movq    %rsi, (%rcx,%rax)
⑤  addq    $8, %rax
⑥  cmpq    %r9, %rax
⑦  jne     .L9
⑧  movq    (%rdi,%rax), %rsi
⑨  movq    (%rcx,%rax), %r8
⑩  movq    %r8, (%rdi,%rax)
⑪  movq    %rsi, (%rcx,%rax)
⑫  addq    $8, %rax
⑬  cmpq    %r9, %rax
⑭  jne     .L9
```

```
①  movq    (%rdi,%rax), %t0
②  movq    (%rcx,%rax), %t1
③  movq    %t1, (%rdi,%rax)
④  movq    %t0, (%rcx,%rax)
⑤  addq    $8, %rax, %t2
⑥  cmpq    %r9, %t2
⑦  jne     .L9
⑧  movq    (%rdi, %t2), %t3
⑨  movq    (%rcx, %t2), %t4
⑩  movq    %t4, (%rdi,%t2)
⑪  movq    %t3, (%rcx,%t2)
⑫  addq    $8, %t2, %t5
⑬  cmpq    %r9, %t5
⑭  jne     .L9
```

Branch prediction can predict what should happen "dynamically"

All false dependencies are gone!!!

Compiler cannot do this!!!

Register renaming

- Provide a set of **physical registers** and a mapping table mapping **architectural registers** to physical registers
  - Architectural registers are virtual registers that software can see/use
- Allocate a physical register for a new output
- Eliminate **name/false dependencies** and allow instruction to be scheduled purely based on **data dependencies**
- Stages
  - Dispatch/Rename (REN) — allocate a "physical register" for the output of a decoded instruction
  - Execute (ALU, M1/M2/M3/M4, BR) — send the instruction to its corresponding pipeline if no structural hazards
  - Write Back (WB) — broadcast the result through CDB

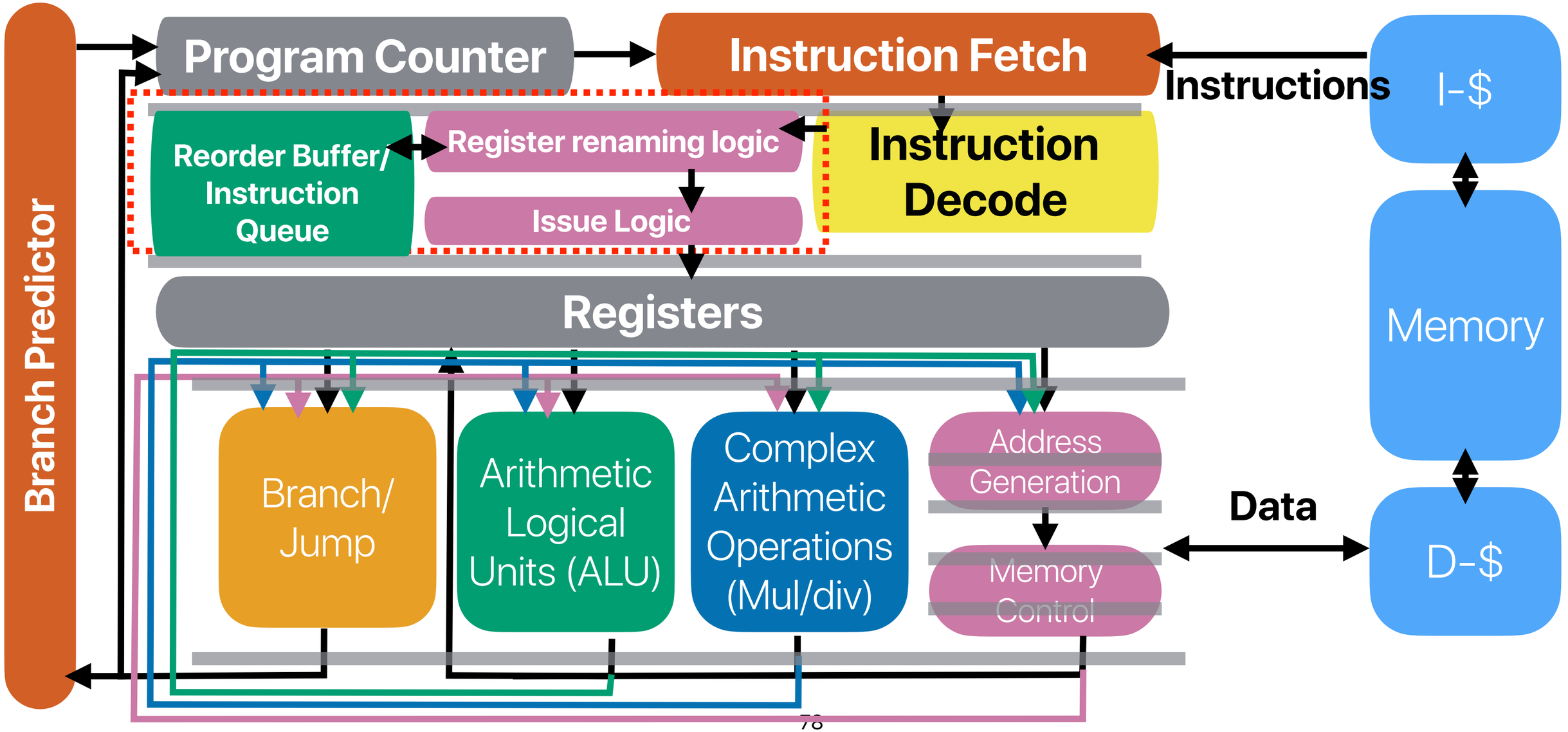
Can we really execute instructions OoO?

- Exceptions may occur anytime — divided by 0, page fault
  - A later instruction cannot write back its own result otherwise the architectural states won't be correct
  - Instructions after the one causes the exception should not be executed
- Hardware can schedule instruction across branch instructions with the help of branch prediction
  - Fetch instructions according to the branch prediction
  - However, branch predictor can never be perfect

Speculative Execution

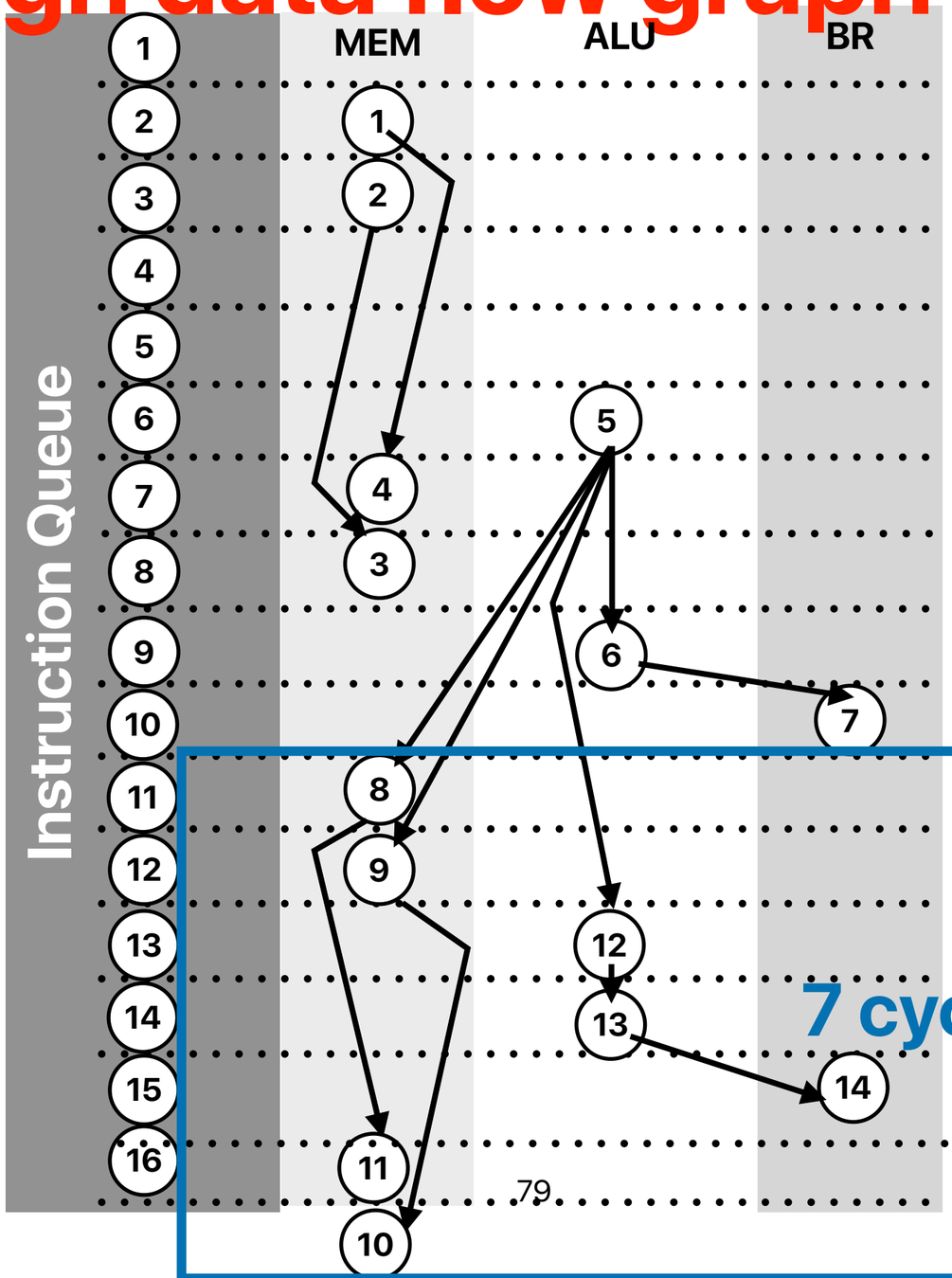
- **Speculative** execution mode: an executing instruction is considered as **speculative** before the processor hasn't determined if the instruction should be executed or not
- Reorder buffer (ROB)
  - The processor allocates an entry for each instruction in a reorder buffer
  - Store results in **reorder buffer and physical registers** when the instruction is still speculative
  - If an earlier instruction failed to commit due to an exception or mis-prediction, the physical registers and all ROB entries after the failed-to-commit instruction are flushed
- Commit/Retire
  - Present the execution result to the running program and in architectural registers when all prior instructions are non-speculative
  - Release the ROB entry

Register renaming + OoO + RoB



Through data flow graph analysis

```
①  movq (%rdi,%rax), %rsi
②  movq (%rcx,%rax), %r8
③  movq %r8, (%rdi,%rax)
④  movq %rsi, (%rcx,%rax)
⑤  addq $8, %rax
⑥  cmpq %r9, %rax
⑦  jne  .L9
⑧  movq (%rdi,%rax), %rsi
⑨  movq (%rcx,%rax), %r8
⑩  movq %r8, (%rdi,%rax)
⑪  movq %rsi, (%rcx,%rax)
⑫  addq $8, %rax
⑬  cmpq %r9, %rax
⑭  jne  .L9
⑮  movq (%rdi,%rax), %rsi
⑯  movq (%rcx,%rax), %r8
⑰  movq %r8, (%rdi,%rax)
⑱  movq %rsi, (%rcx,%rax)
⑲  addq $8, %rax
⑳  cmpq %r9, %rax
㉑  jne  .L9
```

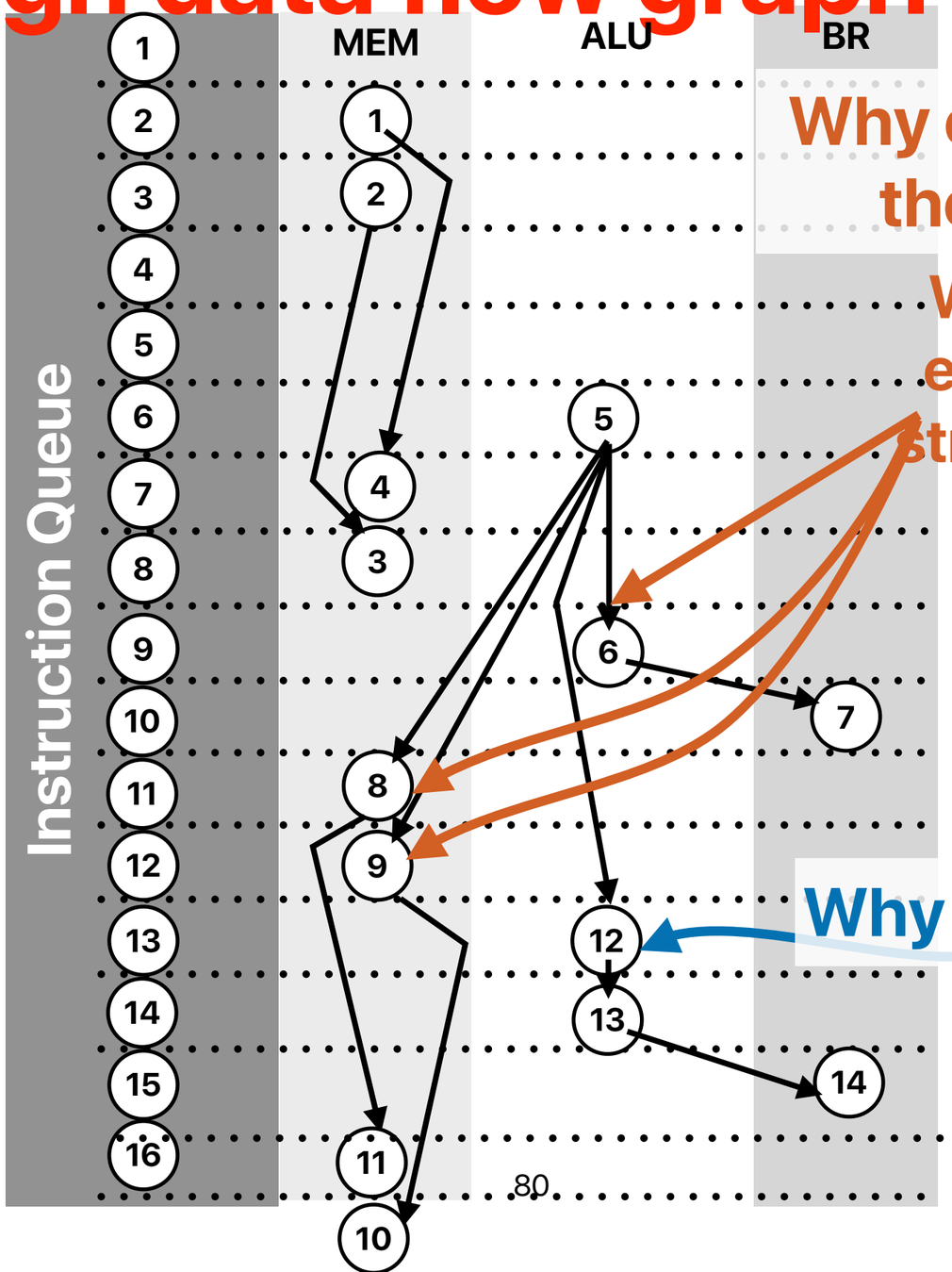


7 cycles every iteration

CPI = $\frac{7}{7} = 1$!

Through data flow graph analysis

```
①  movq (%rdi,%rax), %rsi
②  movq (%rcx,%rax), %r8
③  movq %r8, (%rdi,%rax)
④  movq %rsi, (%rcx,%rax)
⑤  addq $8, %rax
⑥  cmpq %r9, %rax
⑦  jne  .L9
⑧  movq (%rdi,%rax), %rsi
⑨  movq (%rcx,%rax), %r8
⑩  movq %r8, (%rdi,%rax)
⑪  movq %rsi, (%rcx,%rax)
⑫  addq $8, %rax
⑬  cmpq %r9, %rax
⑭  jne  .L9
⑮  movq (%rdi,%rax), %rsi
⑯  movq (%rcx,%rax), %r8
⑰  movq %r8, (%rdi,%rax)
⑱  movq %rsi, (%rcx,%rax)
⑲  addq $8, %rax
⑳  cmpq %r9, %rax
㉑  jne  .L9
```



Why can't we expand the issue logic?

We cannot issue them earlier simply because structural hazards of the issue logic!

We could have this executed earlier if it's in the queue earlier

Why can't we fetch more?

Lessons learned from Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$

- Corollary #1: Maximum speedup
- Corollary #2: Make the common case fast
  - Common case changes all the time
- Corollary #3: Optimization is a moving target
- Corollary #4: Exploiting more parallelism from a program is the key to performance gain in modern architectures
- Corollary #5: Single-core performance still matters
- Corollary #6: Don't hurt the most common case too much

$$Speedup_{max}(f, \infty) = \frac{1}{(1 - f)}$$

$$Speedup_{max}(f_1, \infty) = \frac{1}{(1 - f_1)}$$

$$Speedup_{max}(f_2, \infty) = \frac{1}{(1 - f_2)}$$

$$Speedup_{max}(f_3, \infty) = \frac{1}{(1 - f_3)}$$

$$Speedup_{max}(f_4, \infty) = \frac{1}{(1 - f_4)}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

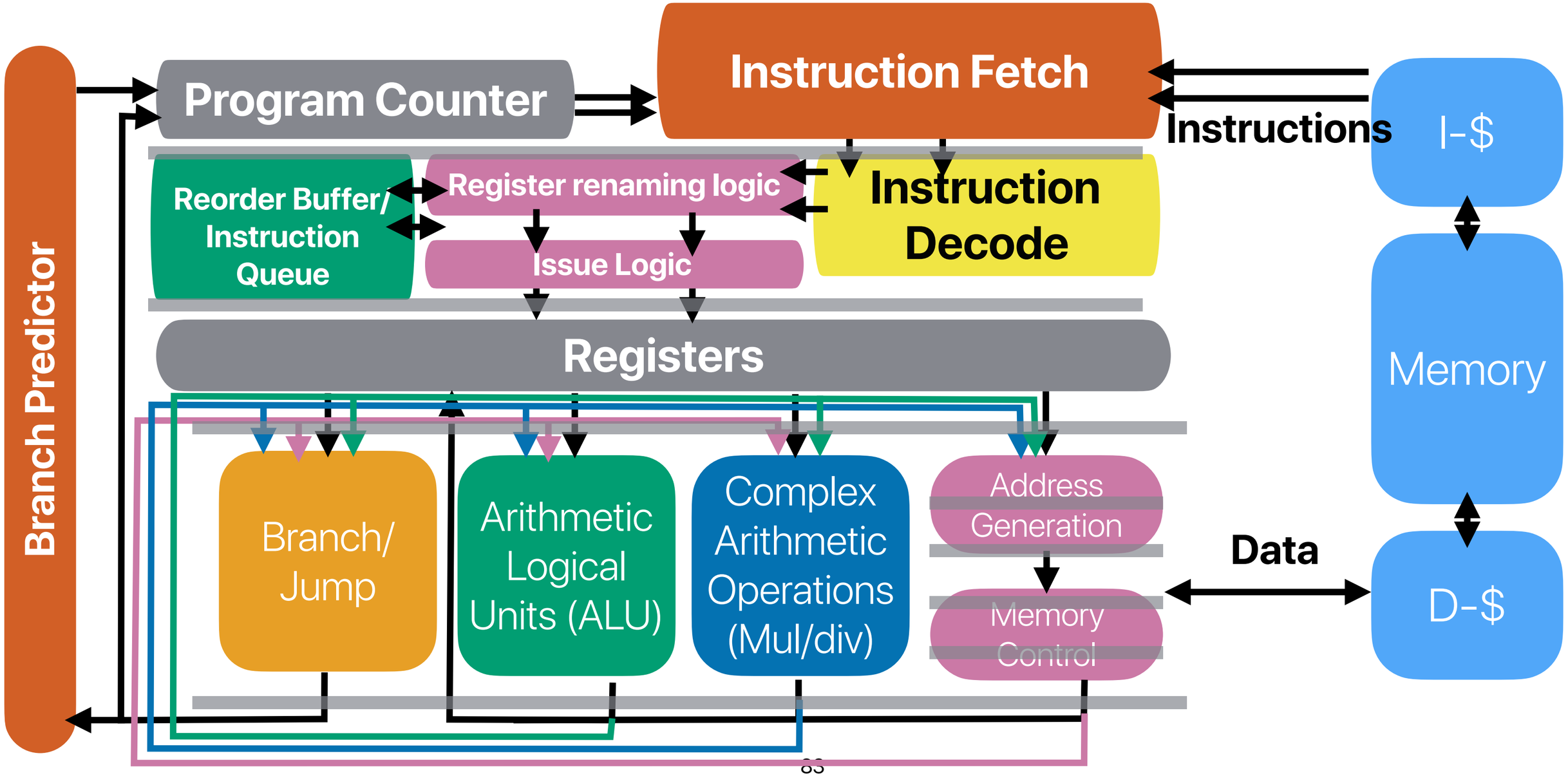
$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

$$Speedup_{enhanced}(f, s, r) = \frac{1}{(1 - f) + perf(r) + \frac{f}{s}}$$

Parallelisms

- Instruction-level parallelism — perform various, independent instructions simultaneously
  - Pipeline
  - OoO/Superscalar

Register renaming +OoO + SuperScalar



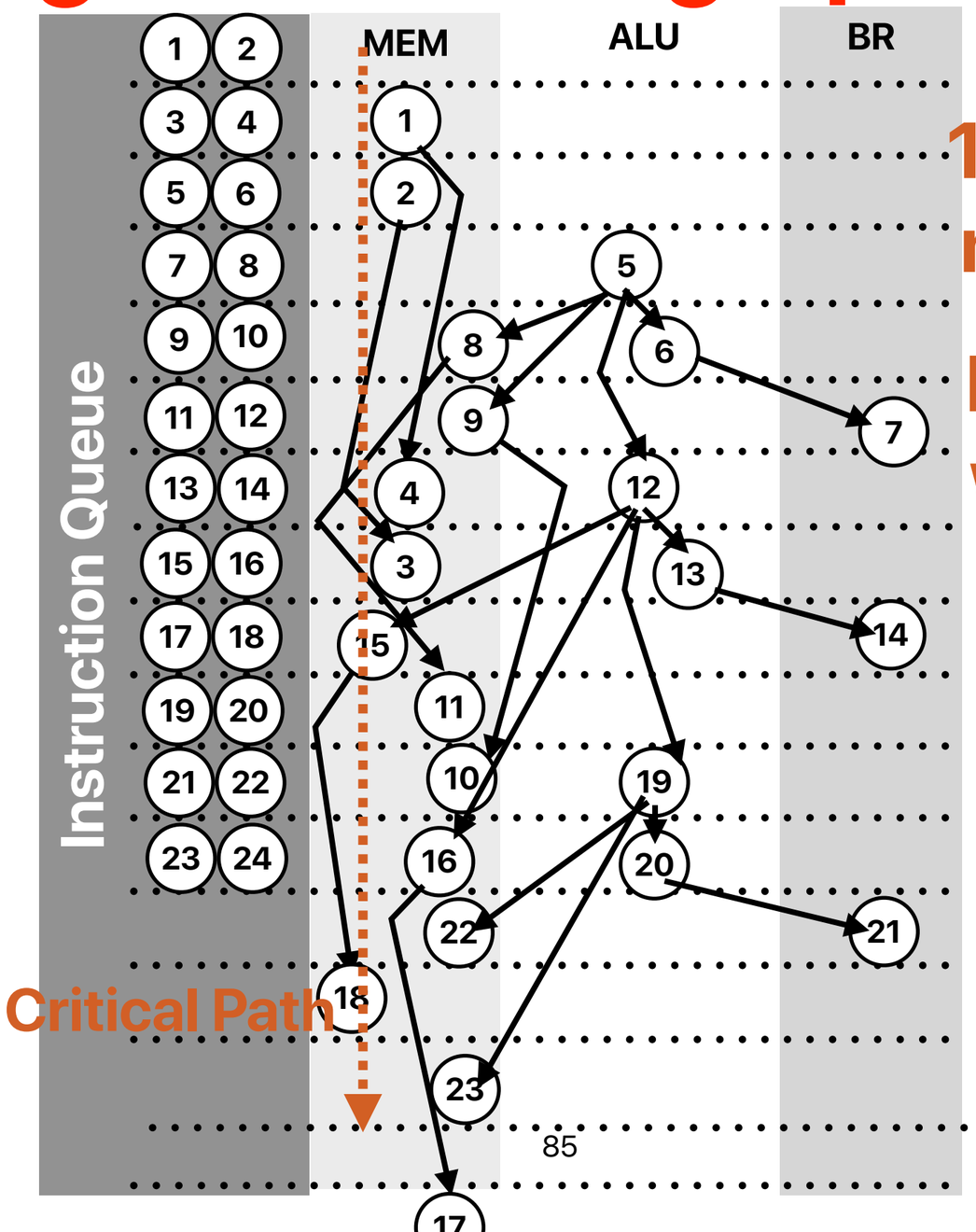
Superscalar

- Since we have many functional units now, we should fetch/decode more instructions each cycle so that we can have more instructions to issue!
- Super-scalar: fetch/decode/issue more than one instruction each cycle
  - **Fetch width:** how many instructions can the processor fetch/decode each cycle
  - **Issue width**: how many instructions can the processor issue each cycle
- The theoretical CPI should now be

$$\frac{1}{min(issue\ \ width, fetch\ \ width, decode\ \ width)}$$

Through data flow graph analysis

```
①  movq (%rdi,%rax), %rsi
②  movq (%rcx,%rax), %r8
③  movq %r8, (%rdi,%rax)
④  movq %rsi, (%rcx,%rax)
⑤  addq $8, %rax
⑥  cmpq %r9, %rax
⑦  jne  .L9
⑧  movq (%rdi,%rax), %rsi
⑨  movq (%rcx,%rax), %r8
⑩  movq %r8, (%rdi,%rax)
⑪  movq %rsi, (%rcx,%rax)
⑫  addq $8, %rax
⑬  cmpq %r9, %rax
⑭  jne  .L9
⑮  movq (%rdi,%rax), %rsi
⑯  movq (%rcx,%rax), %r8
⑰  movq %r8, (%rdi,%rax)
⑱  movq %rsi, (%rcx,%rax)
⑲  addq $8, %rax
⑳  cmpq %r9, %rax
㉑  jne  .L9
㉒  movq (%rdi,%rax), %rsi
㉓  movq (%rcx,%rax), %r8
㉔  movq %r8, (%rdi,%rax)
㉕  movq %rsi, (%rcx,%rax)
㉖  addq $8, %rax
㉗  cmpq %r9, %rax
```



12 cycles for every 11 memory instructions

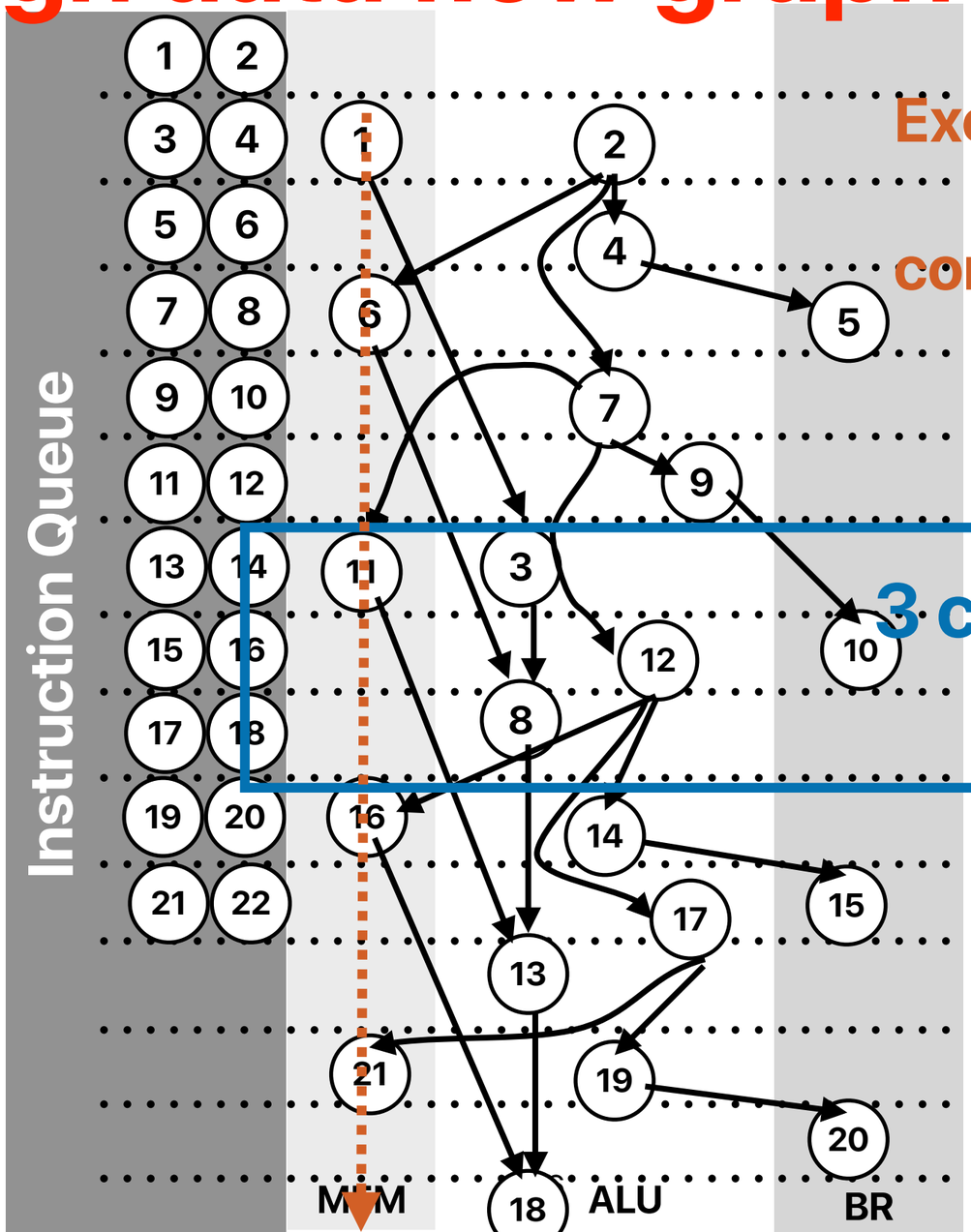
If we have 11 loops, it will have 44 memory instructions, 77 instructions in total and take 48 cycles

CPI:

$$\frac{48}{77} = 0.62$$

Through data flow graph analysis

```
①  movl (%rdi), %ecx
②  addq $4, %rdi
③  addl %ecx, %eax
④  cmpq %rdx, %rdi
⑤  jne  .L3
⑥  movl (%rdi), %ecx
⑦  addq $4, %rdi
⑧  addl %ecx, %eax
⑨  cmpq %rdx, %rdi
⑩  jne  .L3
⑪  movl (%rdi), %ecx
⑫  addq $4, %rdi
⑬  addl %ecx, %eax
⑭  cmpq %rdx, %rdi
⑮  jne  .L3
⑯  movl (%rdi), %ecx
⑰  addq $4, %rdi
⑱  addl %ecx, %eax
⑲  cmpq %rdx, %rdi
⑳  jne  .L3
㉑  movl (%rdi), %ecx
```



Execution time is determined by the "critical path" composed by 1, 6, 11, ..., 1+5n

3 cycles every iteration

CPI = $\frac{3}{5} = 0.6$!

What about “linked list”

Performance determined by the critical path

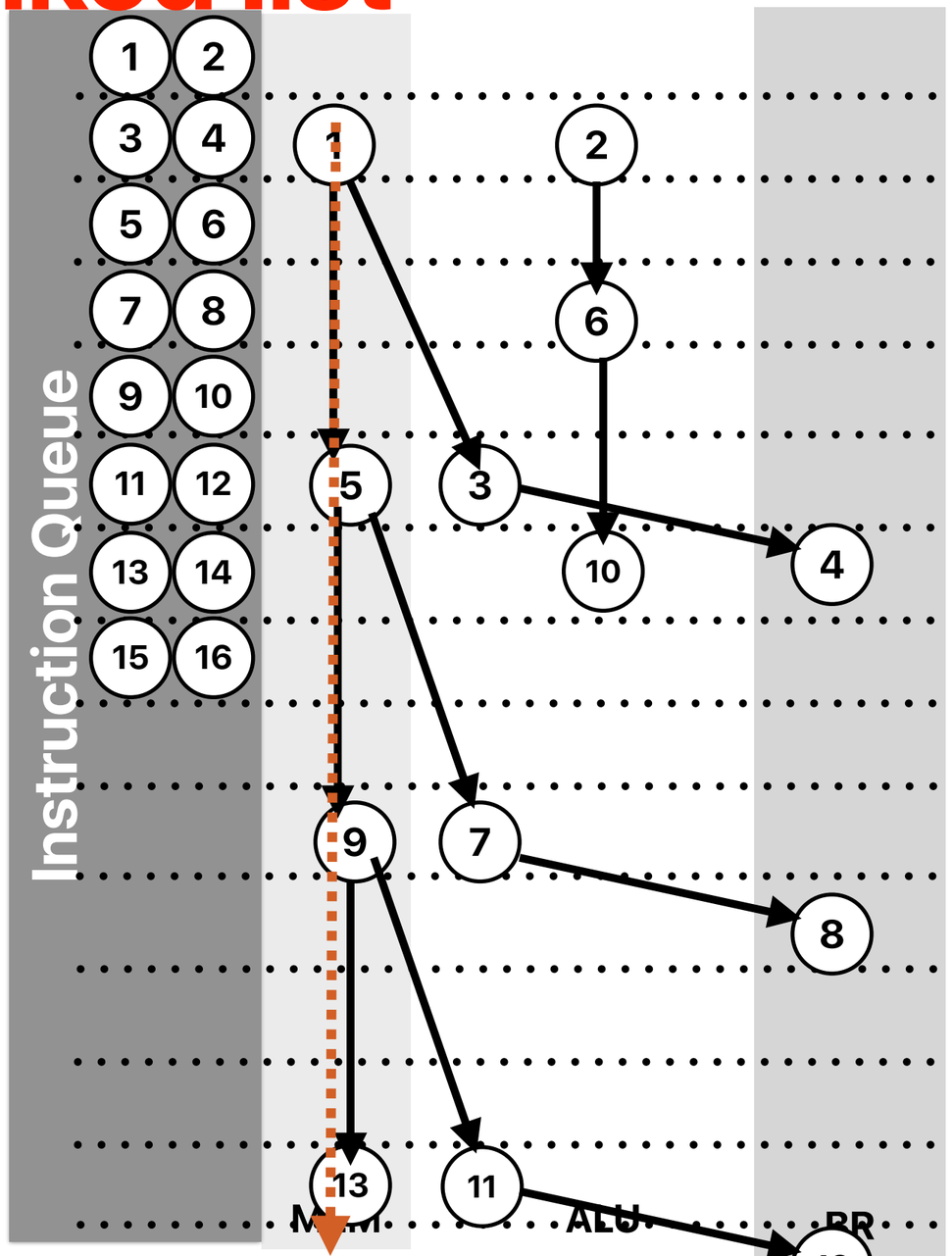
4 cycles each iteration

4 instructions per iteration

$$CPI = \frac{4}{4} = 1$$

```
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL );
```

```
① .L3:    movq    8(%rdi), %rdi
②         addl    $1, %eax
③         testq   %rdi, %rdi
④         jne     .L3
```



Solutions/work-around of pipeline hazards

- Structural
  - Hardware — Stall
  - Hardware — More read/write ports
  - Hardware — Split hardware units (e.g., instruction/data caches)
- Control
  - Hardware — Stalls
  - Hardware — Branch predictions
  - Programmer — Sorting data to facilitate branch predictions
  - Programmer — replacing branch with equivalent statements (e.g., lookup tables)
  - Programmer/compiler — Loop unrolling to remove branches
- Data
  - Hardware — Stalls
  - Hardware — Data forwarding
  - Hardware — Dynamic scheduling
  - Programmer/Compiler — reorder instructions/use equivalent statements with fewer data dependencies

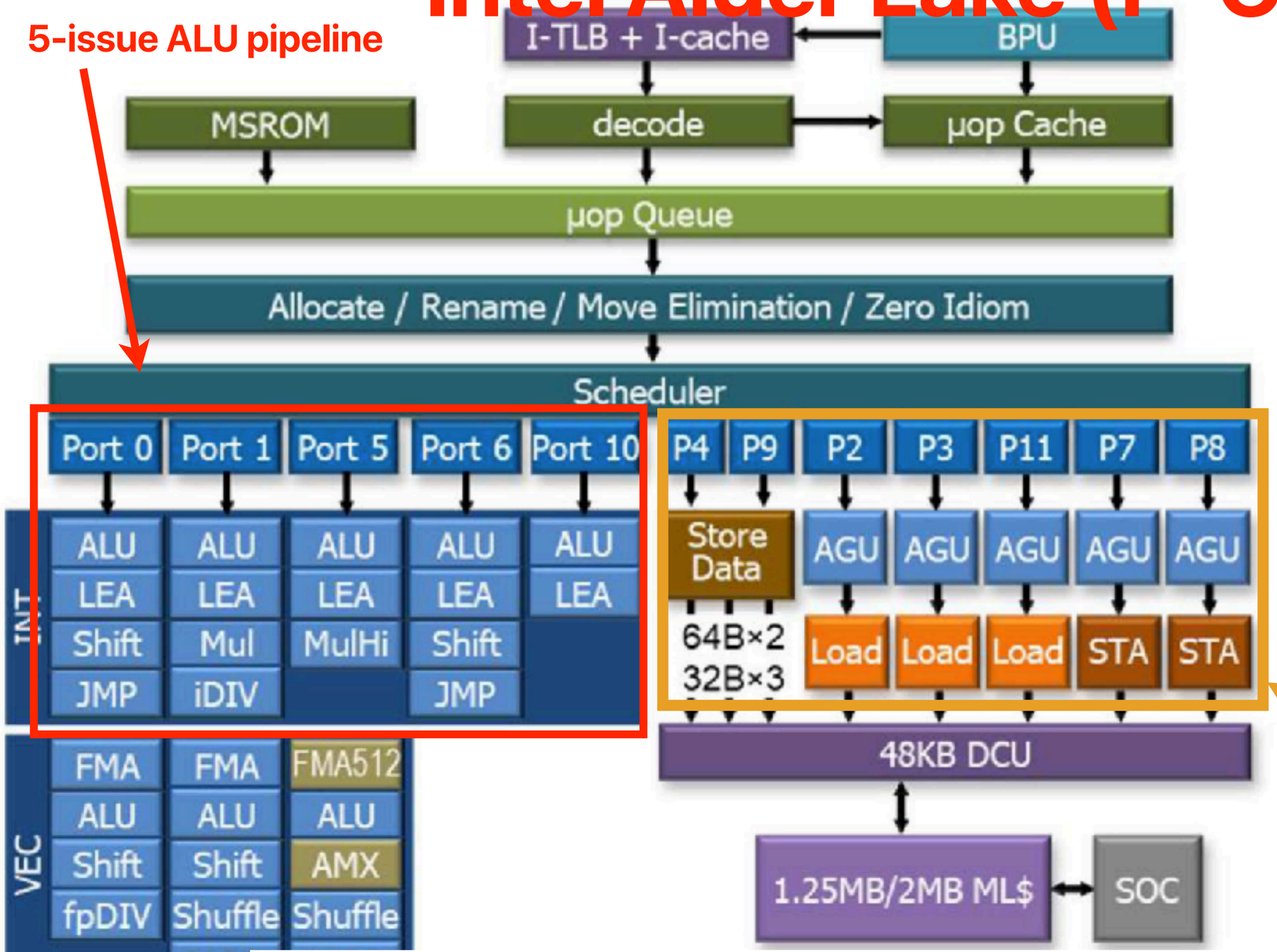
Intel Alder Lake (P-Core)

$$MinCPI = \frac{1}{12}$$

$$MinINTInst.CPI = \frac{1}{5}$$

$$MinMEMInst.CPI = \frac{1}{7}$$

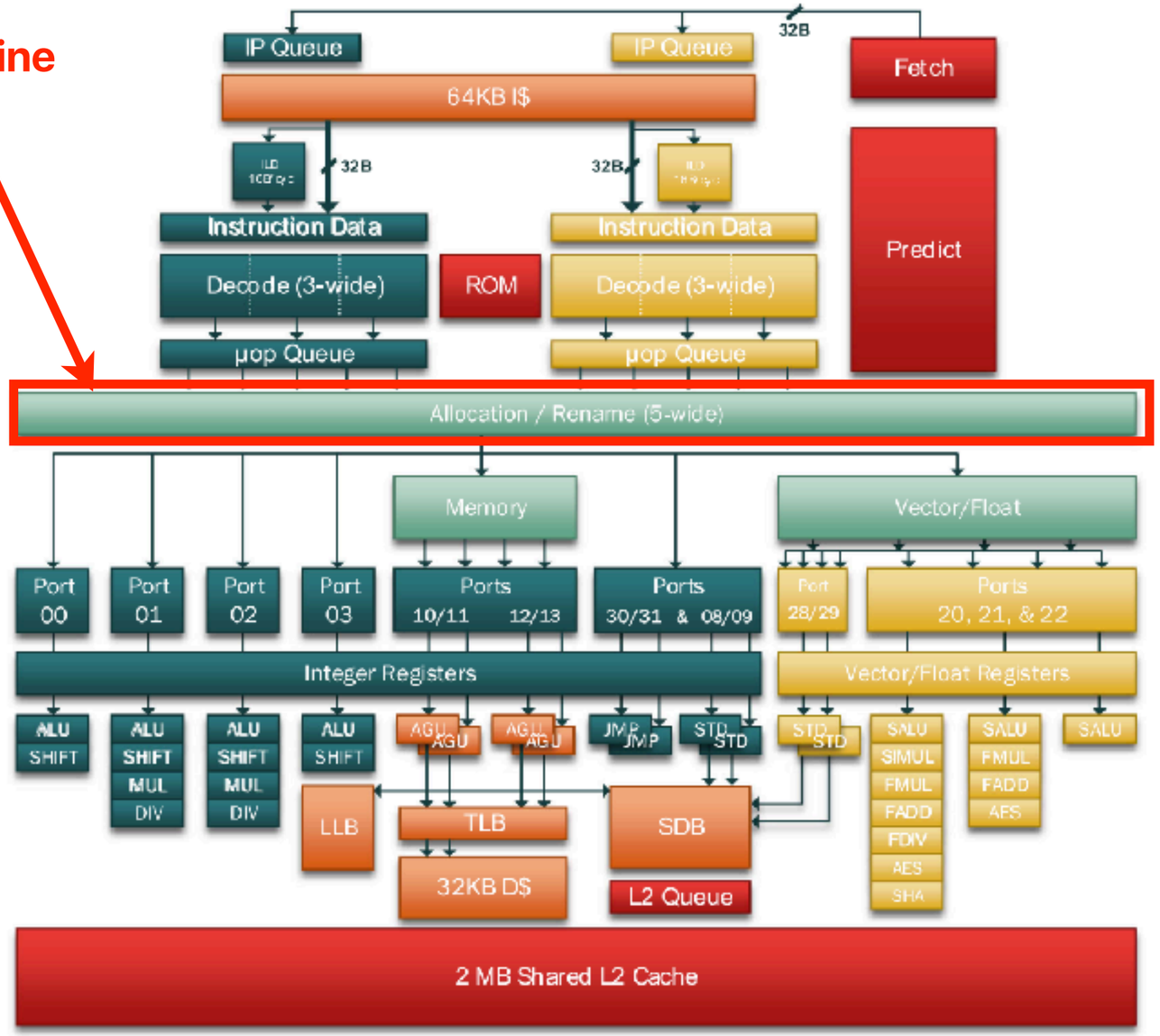
$$MinBRInst.CPI = \frac{1}{2}$$

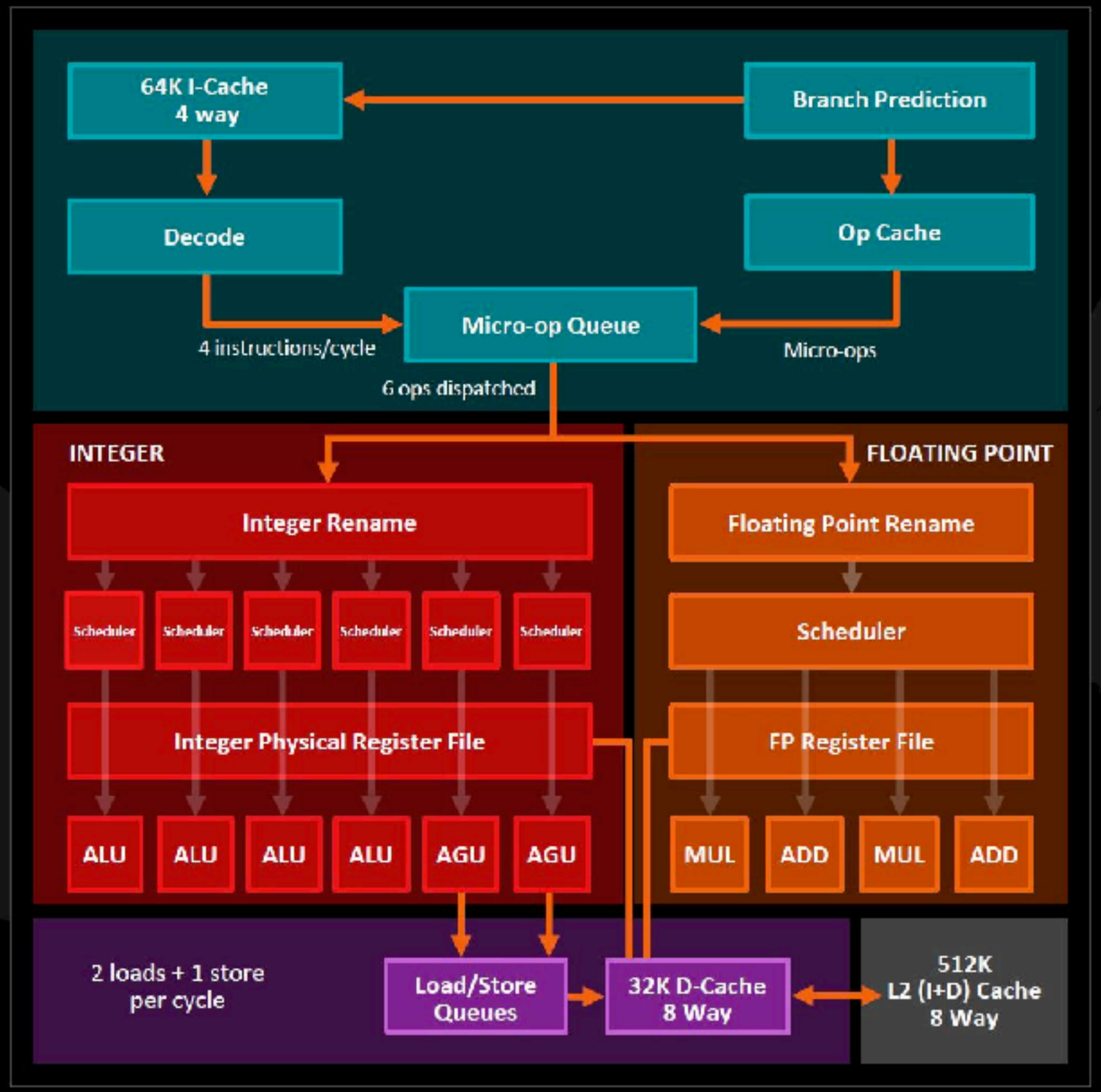


https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

Intel Alder Lake (E-Core)

5-issue pipeline





ZEN MICROARCHITECTURE

- Fetch Four x86 instructions
- Op Cache instructions
- 4 Integer units
  - Large rename space – 168 Registers
  - 192 instructions in flight/8 wide retire
- 2 Load/Store units
  - 72 Out-of-Order Loads supported
- 2 Floating Point units x 128 FMACs
  - built as 4 pipes, 2 Fadd, 2 Fmul
- I-Cache 64K, 4-way
- D-Cache 32K, 8-way
- L2 Cache 512K, 8-way
- Large shared L3 cache
- 2 threads per core

Linked list v.s. arrays

- We can use either a linked list or an array to store a list of data, compare the performance of the array (version A) and linked list (version B) implementations that can achieve the same outcome as below. Assume we have a processor with a reasonably good branch predictor and unlimited fetch/issue width, please identify the correct statements.
  - ① If the dataset is large, the A will outperform the B
  - ② If the dataset is small, there is very little performance difference between A and B
  - ③ If the dataset is small, B will outperform A as A has more branch instructions
  - ④ If the dataset is small, A will outperform B as A has fewer data dependencies

A. 0
B. 1
C. 2
D. 3
E. 4

A

```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

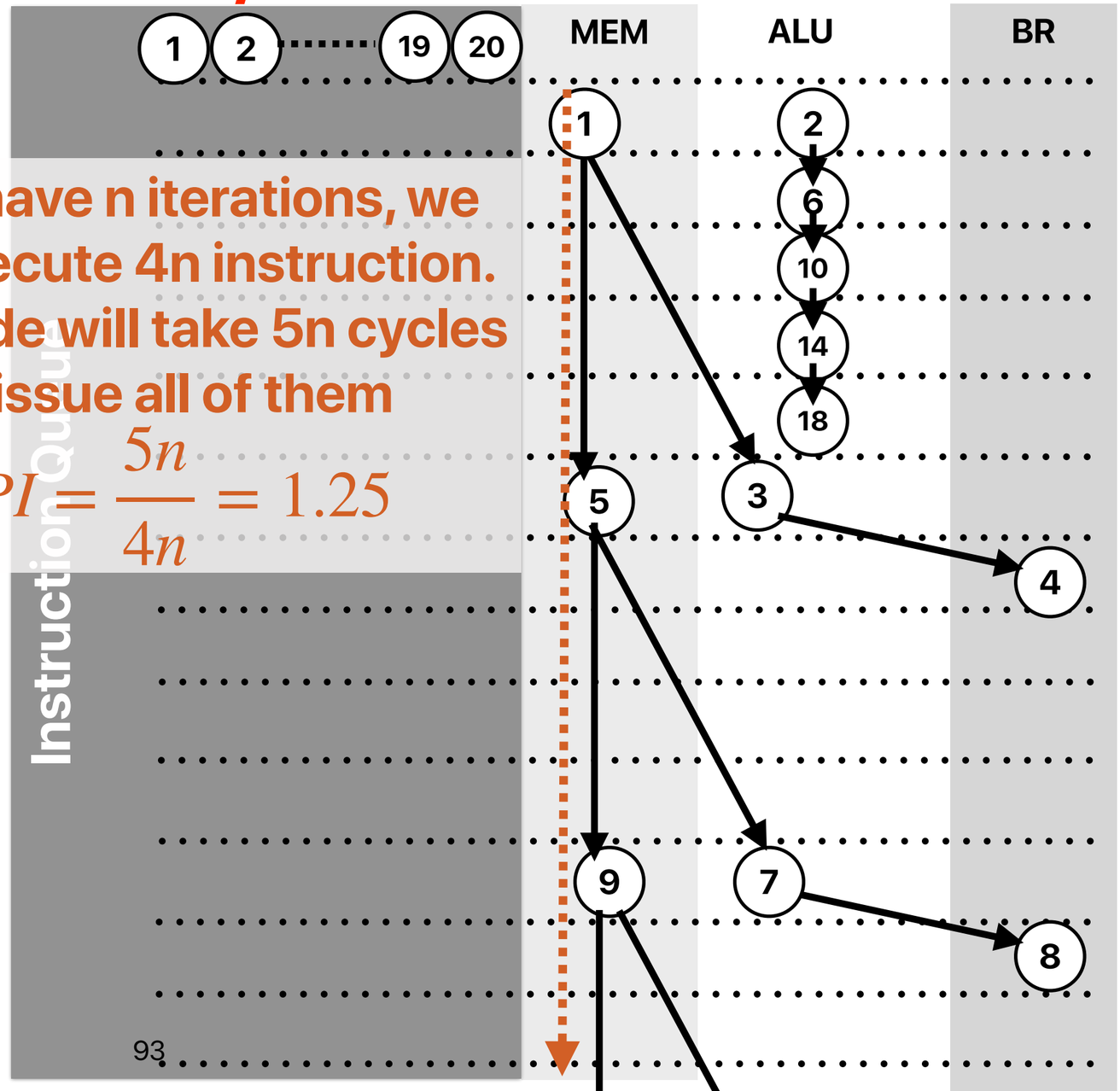
```
while(node)
{
    node = node->next;
    number_of_nodes++;
}
```

What if we have "unlimited" fetch/issue width — "linked list"

```
① .L3:    movq   8(%rdi), %rdi
②         addl   $1, %eax
③         testq  %rdi, %rdi
④         jne    .L3
⑤ .L3:    movq   8(%rdi), %rdi
⑥         addl   $1, %eax
⑦         testq  %rdi, %rdi
⑧         jne    .L3
⑨ .L3:    movq   8(%rdi), %rdi
⑩         addl   $1, %eax
⑪         testq  %rdi, %rdi
⑫         jne    .L3
⑬ .L3:    movq   8(%rdi), %rdi
⑭         addl   $1, %eax
⑮         testq  %rdi, %rdi
⑯         jne    .L3
⑰ .L3:    movq   8(%rdi), %rdi
⑱         addl   $1, %eax
⑲         testq  %rdi, %rdi
⑳         jne    .L3
```

If we have n iterations, we will execute 4n instruction. The code will take 5n cycles to issue all of them

$$CPI = \frac{5n}{4n} = 1.25$$



Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

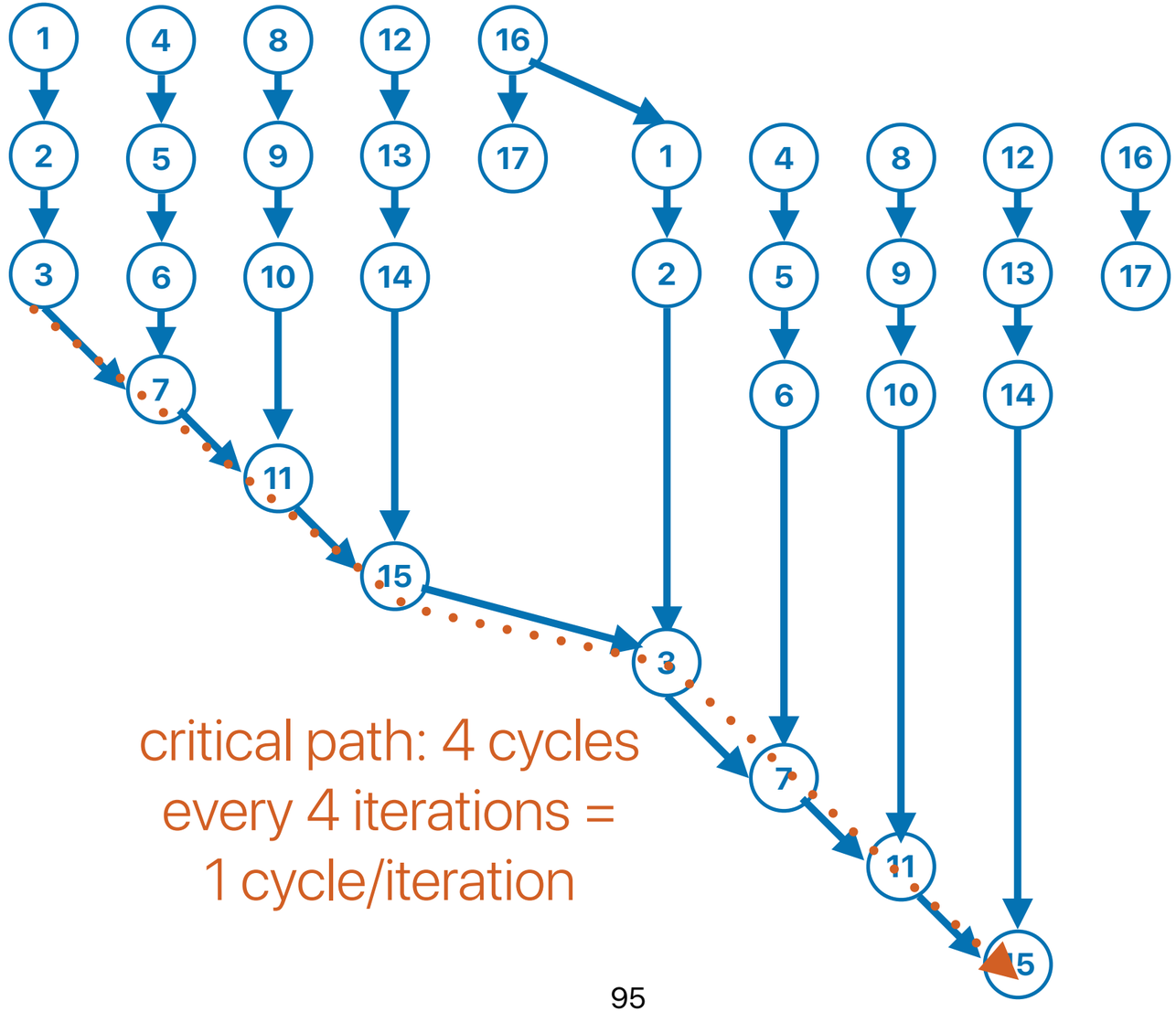
E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Why is B better than A?

B

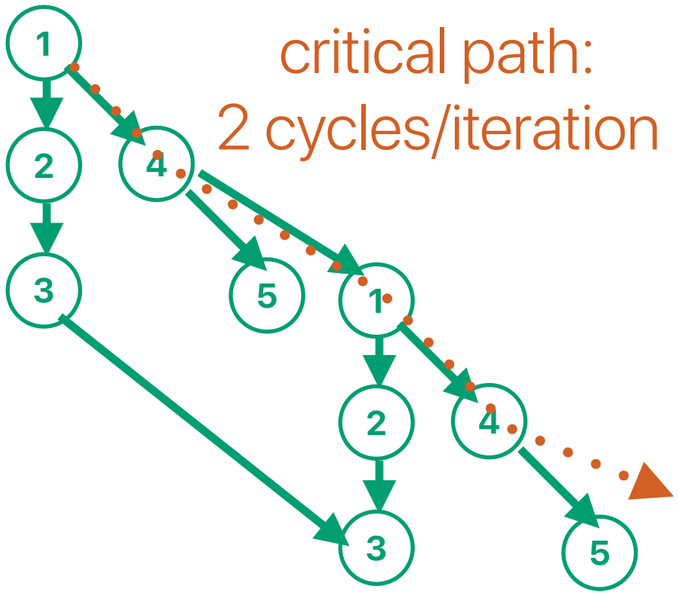
```
① movl   %ecx, %eax
② andl   $1, %eax
③ addl   %edx, %eax
④ movq   %rcx, %rdx
⑤ shrq   %rdx
⑥ andl   $1, %edx
⑦ addl   %eax, %edx
⑧ movq   %rcx, %rax
⑨ shrq   $2, %rax
⑩ andl   $1, %eax
⑪ addl   %edx, %eax
⑫ movq   %rcx, %rdx
⑬ shrq   $3, %rdx
⑭ andl   $1, %edx
⑮ addl   %eax, %edx
⑯ shrq   $4, %rcx
⑰ jne    .L6
```



critical path: 4 cycles
every 4 iterations =
1 cycle/iteration

A

```
① movl   %eax, %ecx
② andl   $1, %ecx
③ addl   %ecx, %edx
④ shrq   %rax
⑤ jne    .L6
```



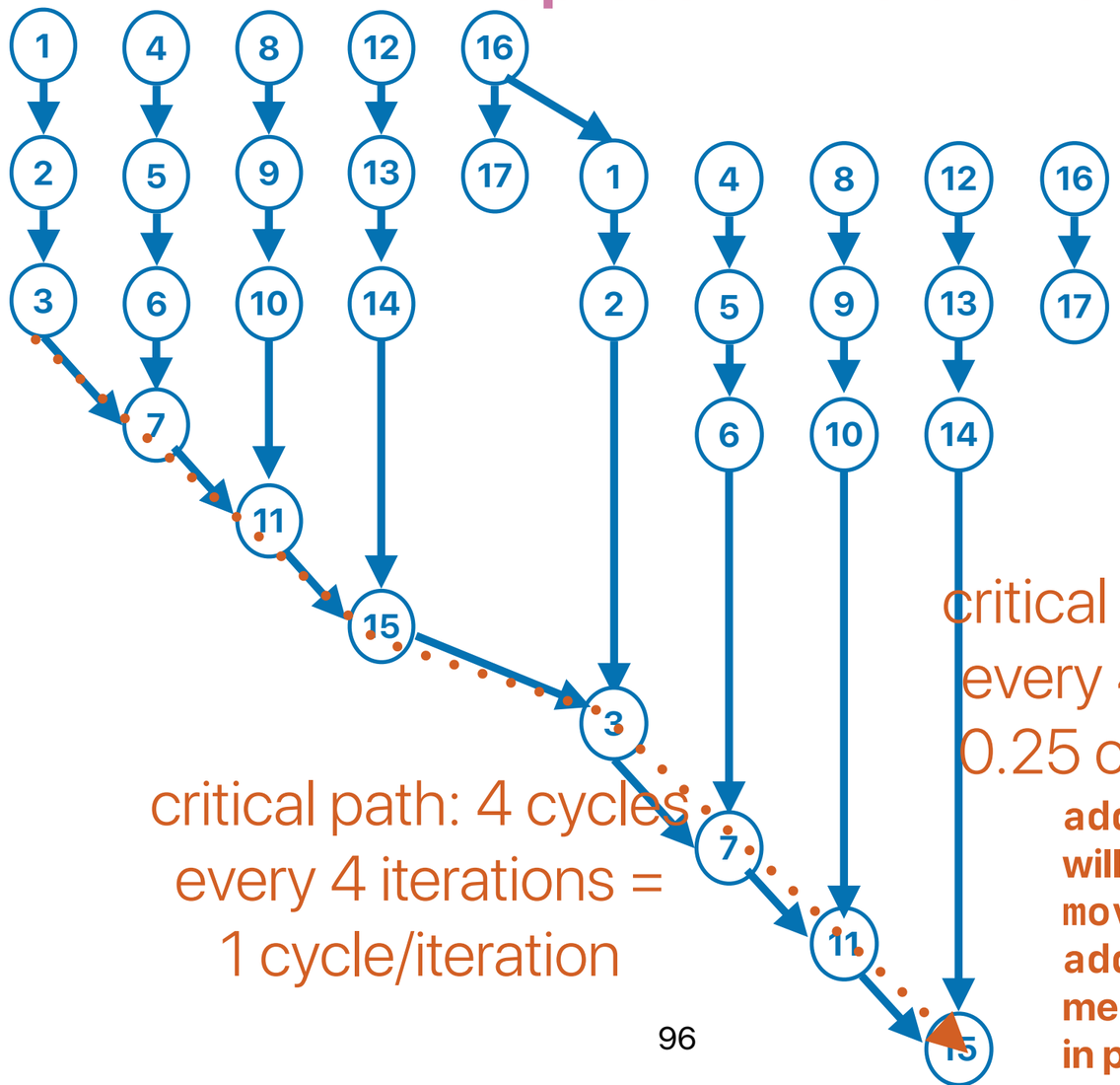
critical path:
2 cycles/iteration

Why is C better than B?

B

```
① movl    %ecx, %eax
② andl    $1, %eax
③ addl    %edx, %eax
④ movq    %rcx, %rdx
⑤ shrq    %rdx
⑥ andl    $1, %edx
⑦ addl    %eax, %edx
⑧ movq    %rcx, %rax
⑨ shrq    $2, %rax
⑩ andl    $1, %eax
⑪ addl    %edx, %eax
⑫ movq    %rcx, %rdx
⑬ shrq    $3, %rdx
⑭ andl    $1, %edx
⑮ addl    %eax, %edx
⑯ shrq    $4, %rcx
⑰ jne     .L6
```

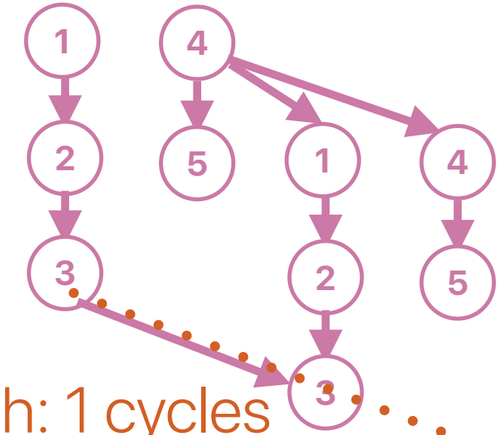
These 5 instructions represent 4 iterations!!!



critical path: 4 cycles
every 4 iterations =
1 cycle/iteration

C

```
.L6:
① movq  %rcx, %rdi
② andl  $15, %edi
③ addl  (%rsp,%rdi,4), %eax
④ shrq  $4, %rcx
⑤ jne   .L6
```



critical path: 1 cycles
every 4 iterations =
0.25 cycle/iteration

`addl (%rsp,%rdi,4), %eax`
will be translated into uOPs of
`mov (%rsp,%rdi,4), something` and
`addl something, %eax` –
memory operations can be executed in parallel

All branches are gone with loop unrolling

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
     return c;
}
```

Compiler cannot create this lookup table!

Without knowing “i<16” in the for-loop, this is not possible

Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

Eliminate some branches with manual unrolling

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

Using a LUT to reduce instructions (LUT must fit in $)

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

LUT and eliminate all branches!

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Worst b/c lots of hard-to-predict branches

Hardware acceleration

- Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a
- In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number
  - You need to compile the program with -m64 -msse4.2 flags to enable these new features

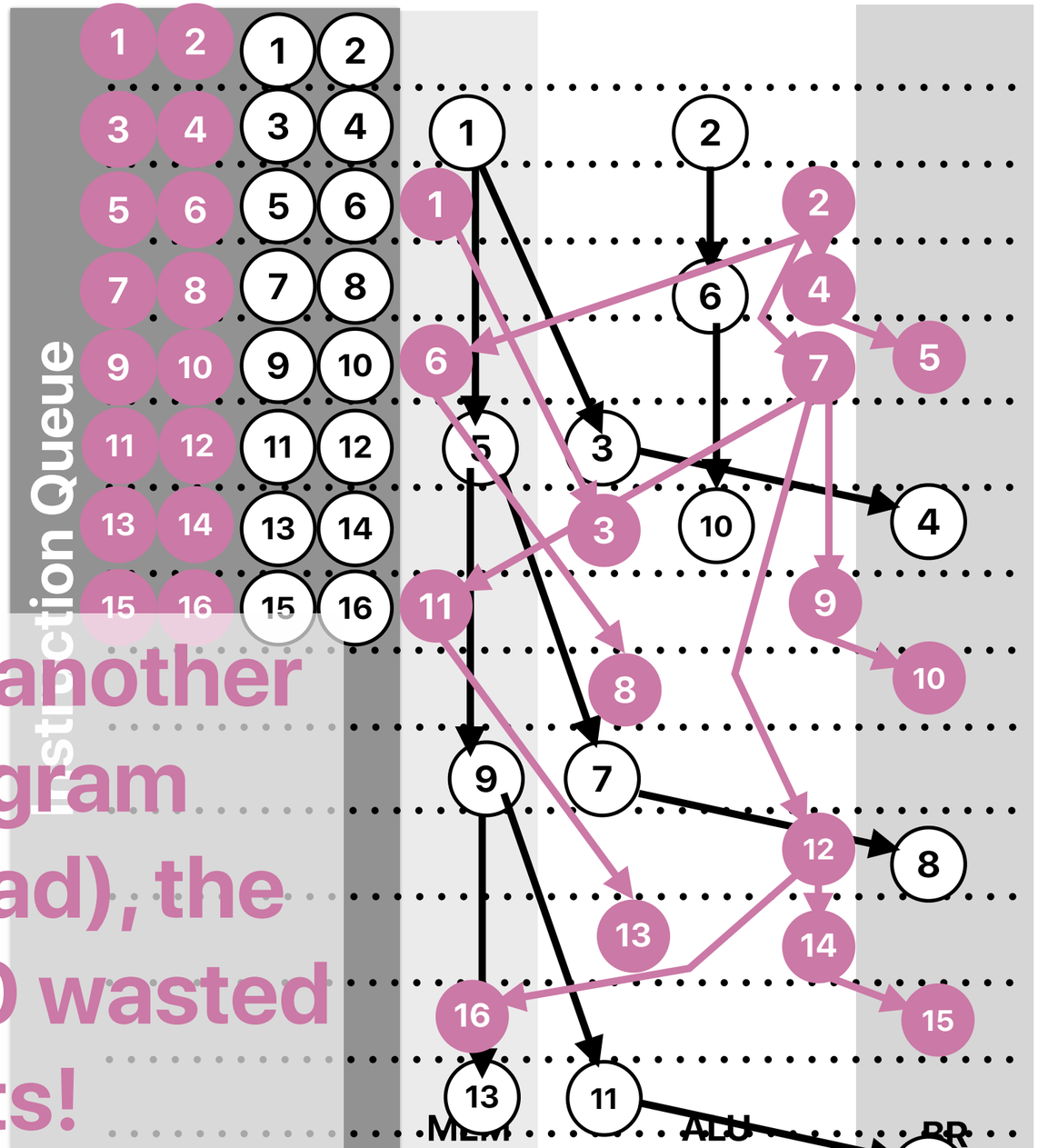
```
#include <smmintrin.h>
inline int popcount(uint64_t x) {
      int c = _mm_popcnt_u64(x);
      return c;
}
```

Tips of programming on modern processors

- Minimize the critical path operations
  - Don’t forget about optimizing cache/memory locality first!
    - Memory latencies are still way longer than any arithmetic instruction
    - Can we use arrays/hash tables instead of lists?
  - Branch can be expensive as pipeline get deeper
    - Sorting
    - Loop unrolling
    - Can we just not branch?
  - Still need to carefully avoid long latency operations (e.g., mod)
- Since processors have multiple functional units — code must be able to exploit instruction-level parallelism
  - Hide as many instructions as possible under the “critical path”
  - Try to use as many different functional units simultaneously as possible
- Modern processors also have accelerated instructions

Concept: Simultaneous Multithreading (SMT)

```
①  movq    8(%rdi), %rdi
②  addl    $1, %eax
③  testq   %rdi, %rdi
④  jne     .L3
⑤  movq    8(%rdi), %rdi
⑥  addl    $1, %eax
⑦  testq   %rdi, %rdi
⑧  jne     .L3
⑨  movq    8(%rdi), %rdi
⑩  addl    $1, %eax
⑪  testq   %rdi, %rdi
⑫  jne     .L3
```



By scheduling another running program instance (thread), the processor has 0 wasted issue slots!

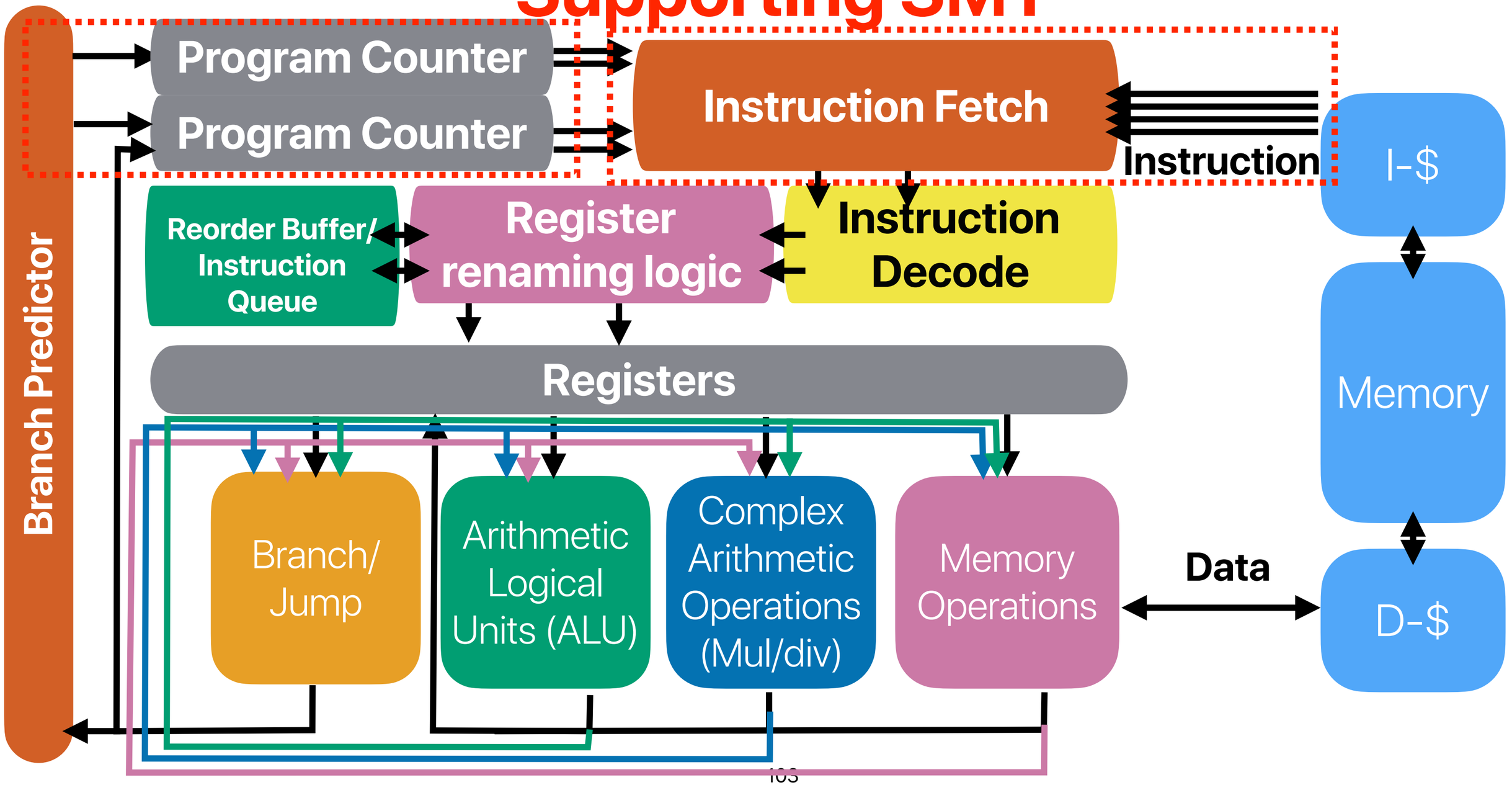
```
①  movl (%rdi), %ecx
②  addq $4, %rdi
③  addl %ecx, %eax
④  cmpq %rdx, %rdi
⑤  jne  .L3
⑥  movl (%rdi), %ecx
⑦  addq $4, %rdi
⑧  addl %ecx, %eax
⑨  cmpq %rdx, %rdi
⑩  jne  .L3
⑪  movl (%rdi), %ecx
⑫  addq $4, %rdi
⑬  addl %ecx, %eax
⑭  cmpq %rdx, %rdi
⑮  jne  .L3
⑯  movl (%rdi), %ecx
```

Parallelisms

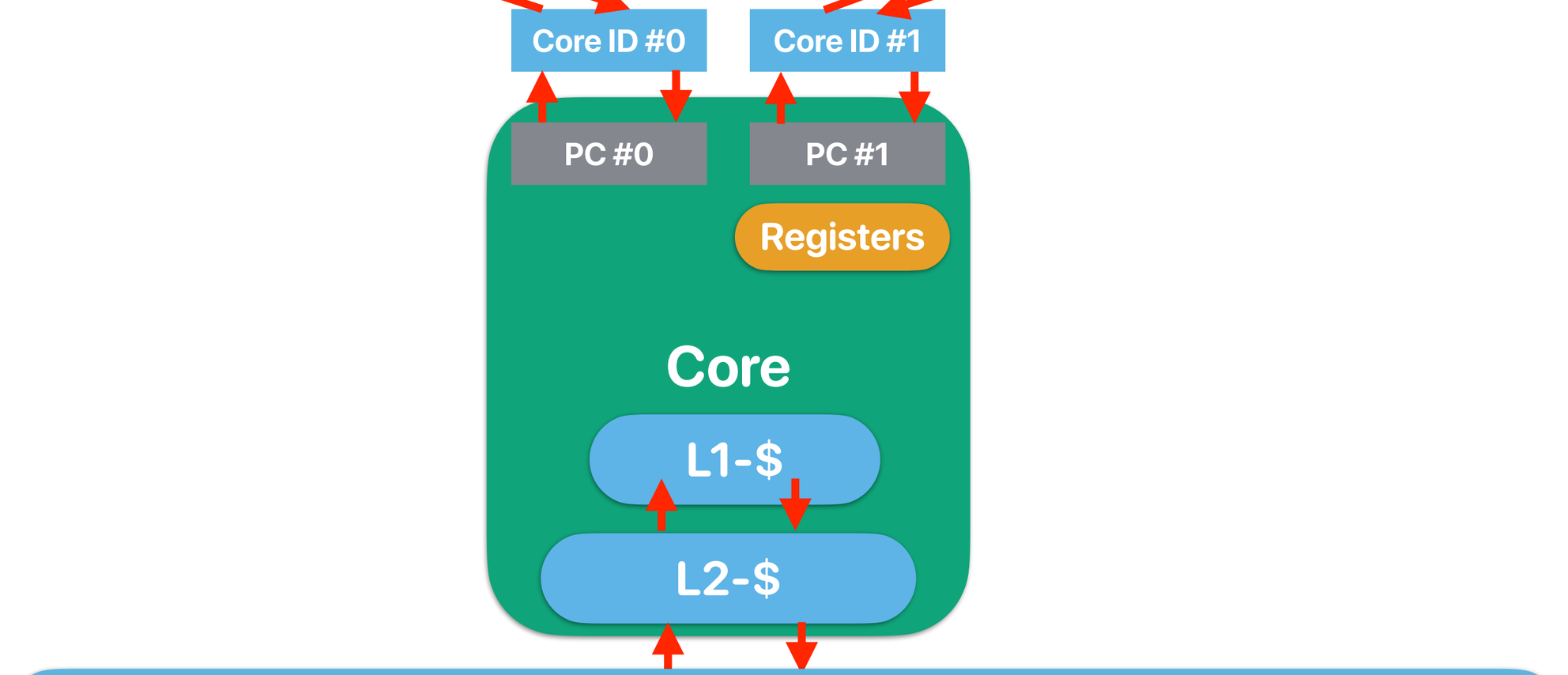
- Instruction-level parallelism — perform various, independent instructions simultaneously
  - Pipeline
  - OoO/Superscalar

- Thread-level parallelism — perform independent computation streams (composed of many instructions or SIMD instructions)
  - Multicore/SMT processors

Supporting SMT



SMT from the user/OS' perspective



SMT

- How many of the following about SMT are correct?
  - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  - ② SMT can ~~improve~~ the throughput of a single-threaded application
  - ③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width
  - ④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

We can execute from other threads/contexts instead of the current one

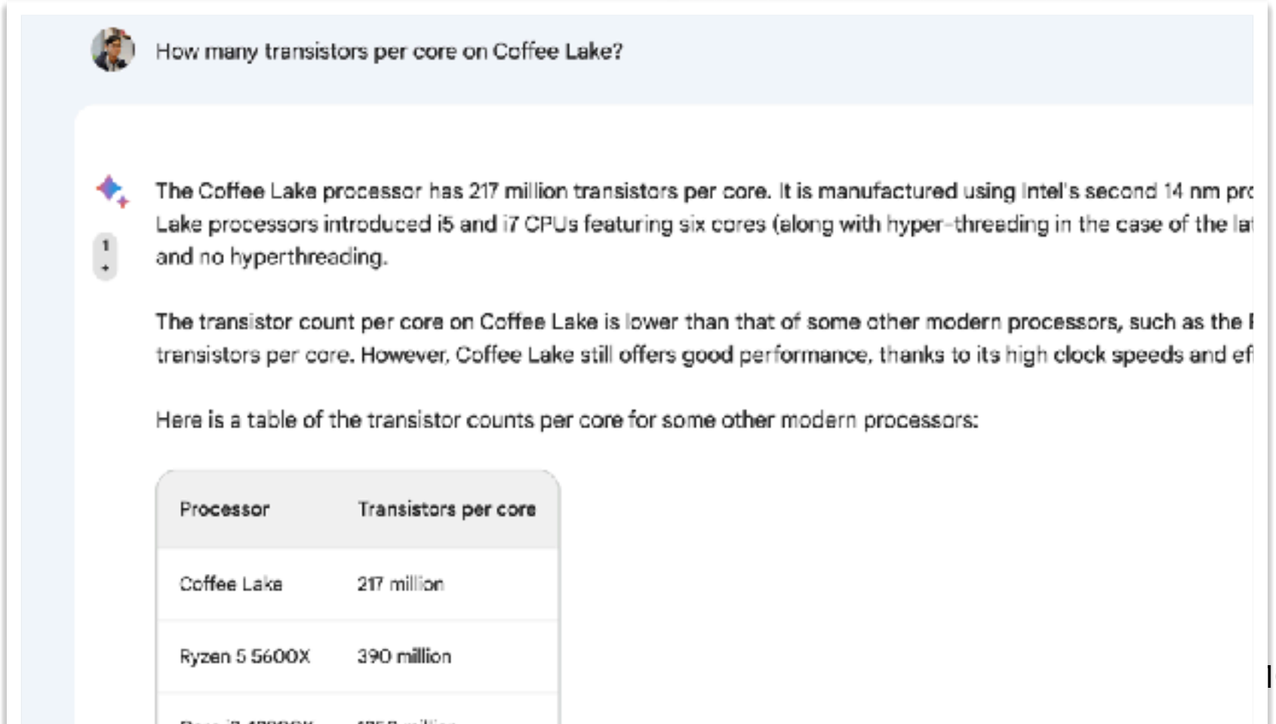
hurt, b/c you are sharing resource with other threads.

We can execute from other threads/ contexts instead of the current one

b/c we're sharing the cache

Transistor counts

Microarchitecture	Transistor Count	Issue-width	Year
Alder Lake	325 M	5x ALU, 7x Memory	2021
Coffee Lake	217 M	4x ALU, 4x Memory	2017
Sandy Bridge	290 M	3x ALU, 3x Memory	2011
Nehalem	182.75 M	3x ALU, 3x Memory	2008



How many transistors per core on Coffee Lake?

The Coffee Lake processor has 217 million transistors per core. It is manufactured using Intel's second 14 nm pro
Lake processors introduced i5 and i7 CPUs featuring six cores (along with hyper-threading in the case of the lat
and no hyperthreading.

The transistor count per core on Coffee Lake is lower than that of some other modern processors, such as the I
transistors per core. However, Coffee Lake still offers good performance, thanks to its high clock speeds and ef

Here is a table of the transistor counts per core for some other modern processors:

Processor	Transistors per core
Coffee Lake	217 million
Ryzen 5 5600X	390 million

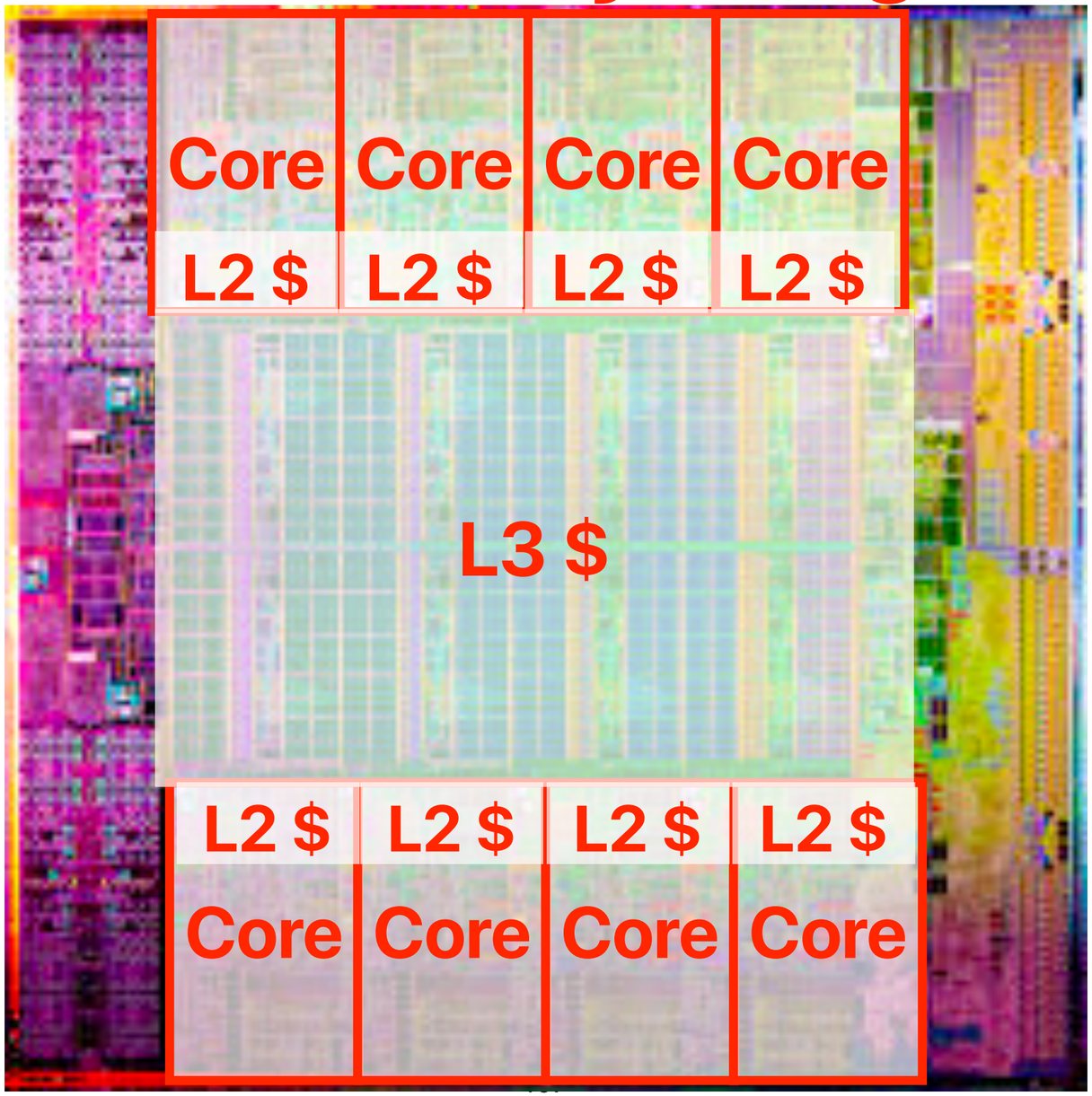
2x 3-issue ALUs Nehalem

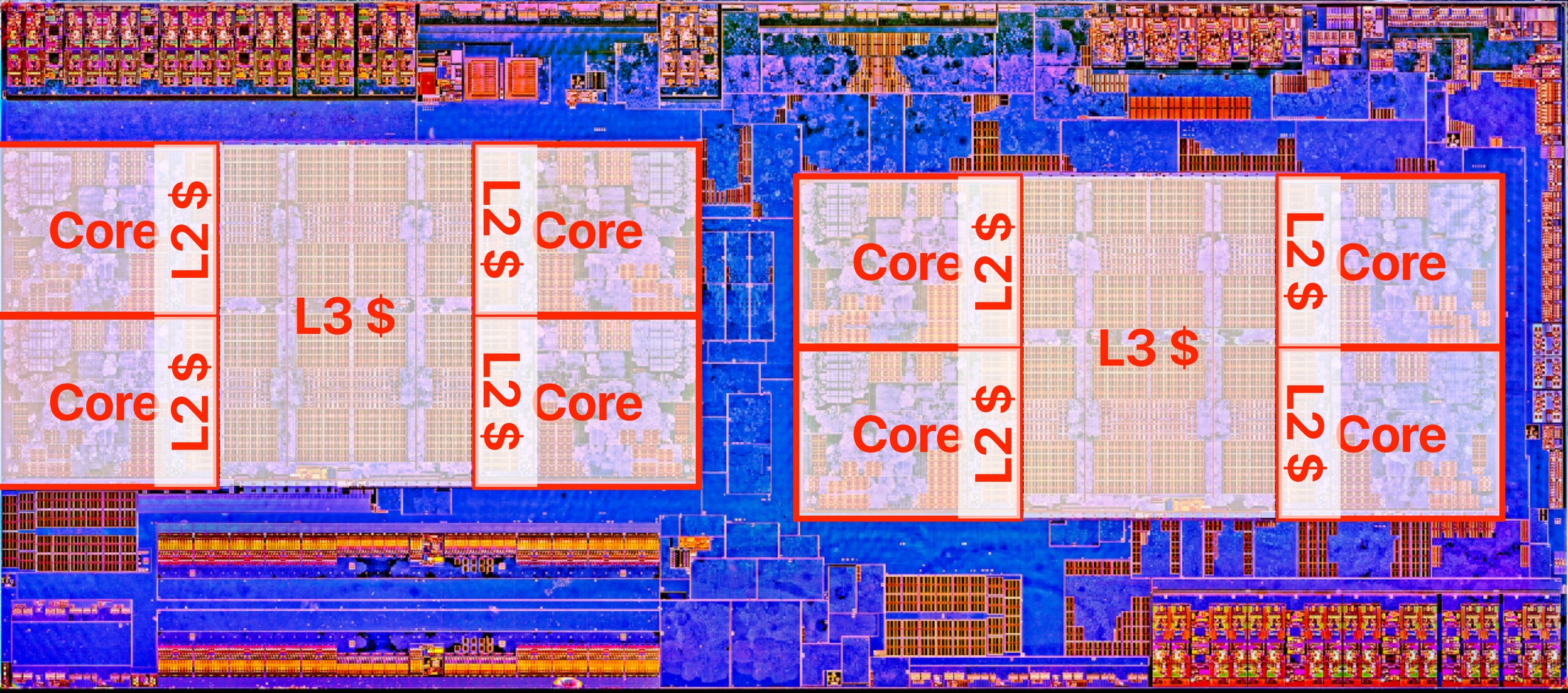
Nehalem 6-issue | Alder Lake 12-issue | Nehalem 6-issue

1x 5-issue ALUs Alder Lake

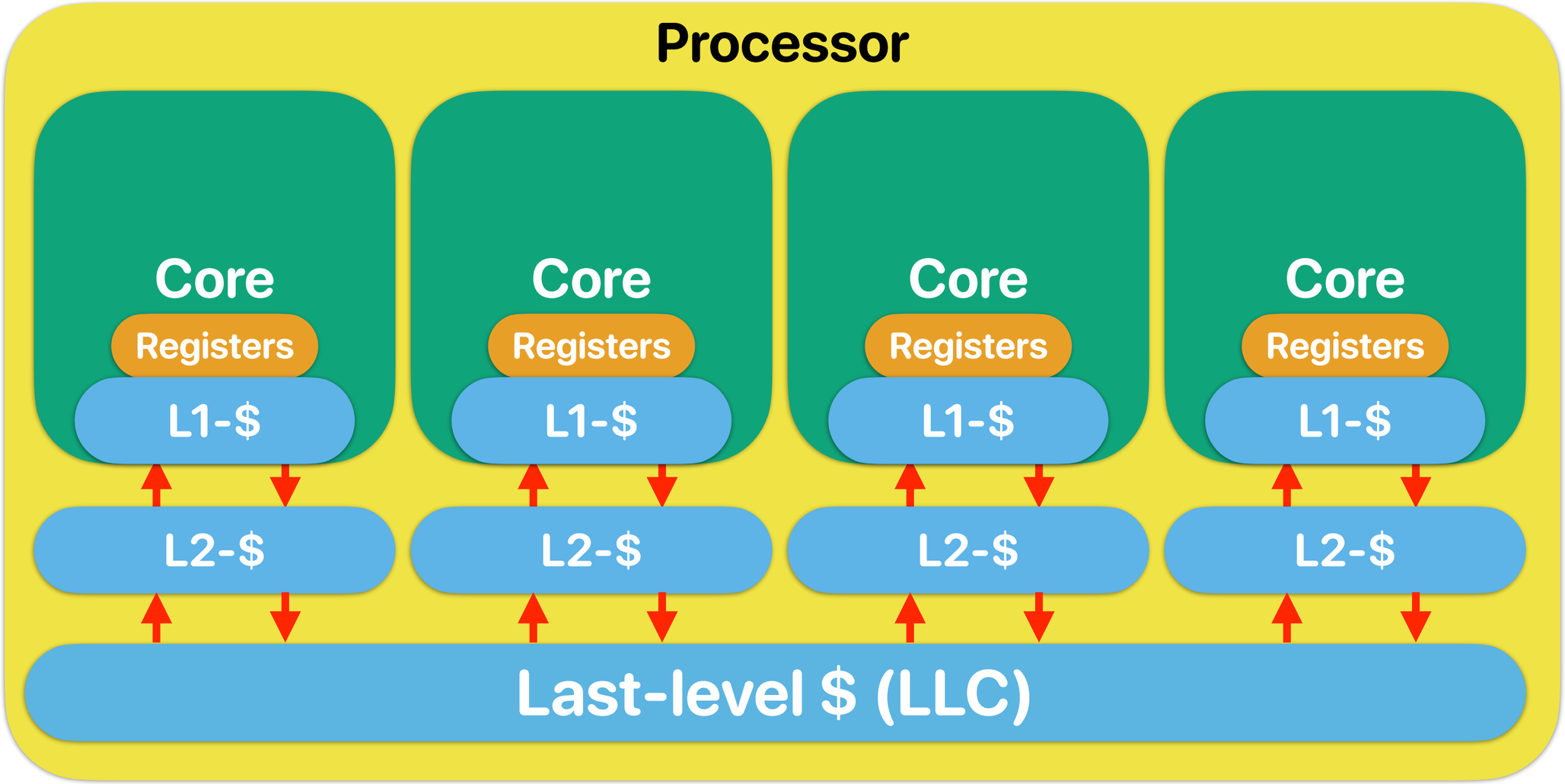
Based on https://en.wikipedia.org/wiki/Transistor_count

Intel Sandy Bridge




Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
AMD
RYZEN

Concept of CMP



SMT v.s. CMP

- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many of the following statements are/is correct when running programs on these processors.
  - ① If we are just running one program in the system, the program will perform better on an SMT processor **— you have more resources for the program**
  - ② If we are running 4 applications simultaneously, the cache miss rates will be higher in the SMT processor **— it depends!**
  - ③ If we are running 4 applications simultaneously, the branch mis-prediction will be higher in the SMT processor **— it depends!**
  - ④ If we are running one program with 4 parallel threads, the cache miss rates will be higher in the SMT processor **— it depends!**
  - ⑤ If we are running one program with 4 parallel threads simultaneously, the branch mis-prediction will be longer in the SMT processor **— it depends!**

A. 1
B. 2
C. 3
D. 4
E. 5

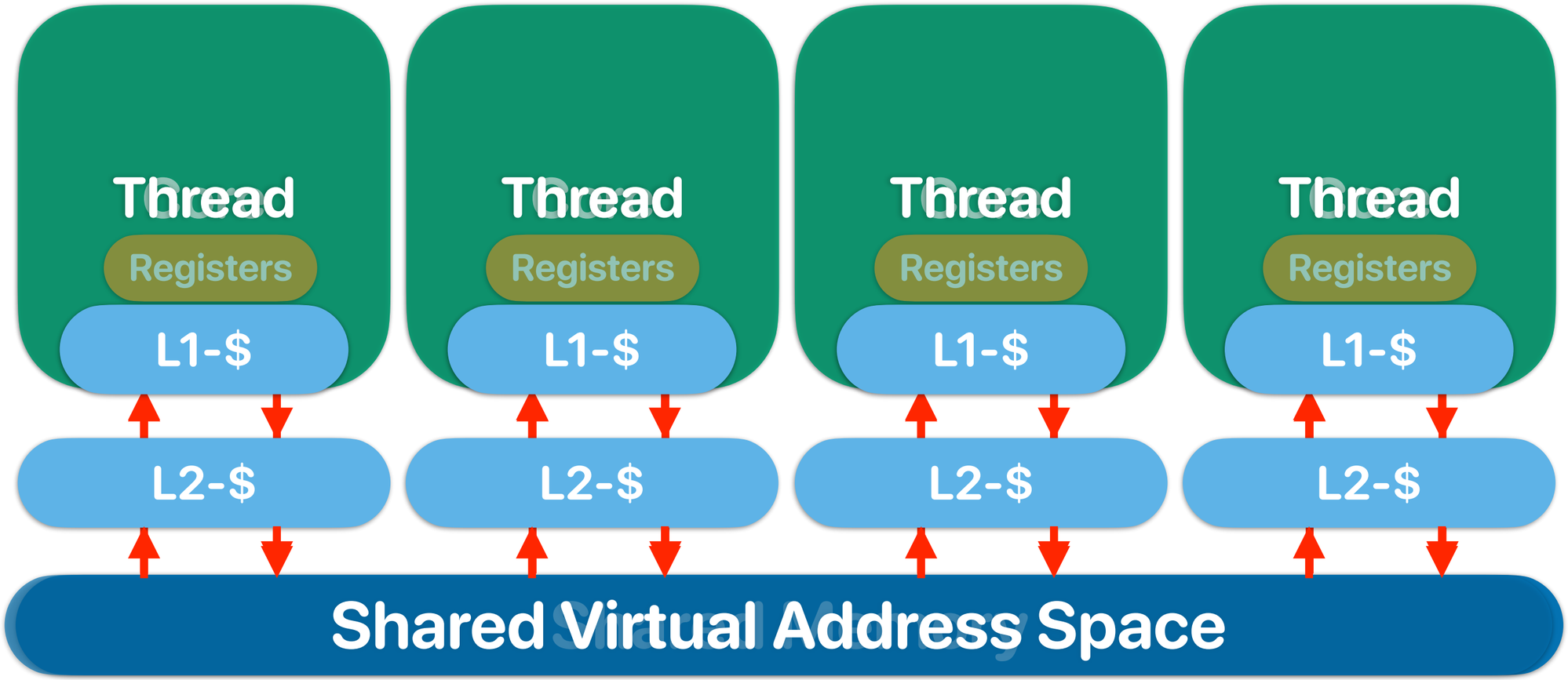
There is no clear win on each — why not having both?

The only thing we know for sure — if we don't parallel the program, it won't get any faster on CMP

Modern processors have both CMP/SMT

Architectural Support for Parallel Programming

What software thinks about “multiprogramming” hardware



Coherency & Consistency

- Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
  - What value should be seen
- Consistency — All threads see the change of data in the same order
  - When the memory operation should be done

Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
`while(1)` `  printf("%d ",a);`	`while(1)` `  a++;`

① 0 1 2 3 4 5 6 7 8 9

② 1 2 5 9 3 6 8 10 12 13

③ 1 1 1 1 1 1 1 64 100

④ 1 1 1 1 1 1 1 1 1 100

A. 0

B. 1

C. 2

D. 3

E. 4

4Cs of cache misses

- 3Cs:
  - Compulsory, Conflict, Capacity
- Coherency miss:
  - A “block” invalidated because of the sharing among processors.

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
  tids[i] = i;
  pthread_create(&thread[i], NULL, threaded_vadd, &tids
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
  pthread_join(thread[i], NULL);
```

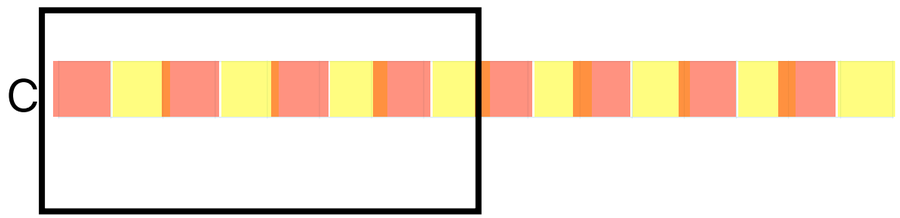
L v.s. R

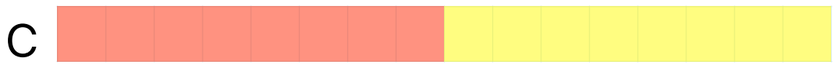
Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```





Even without coherence misses,
you're also only using a small fraction of the block — you need more blocks to capture the same amount of data — more likely to result in capacity or conflict misses

Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

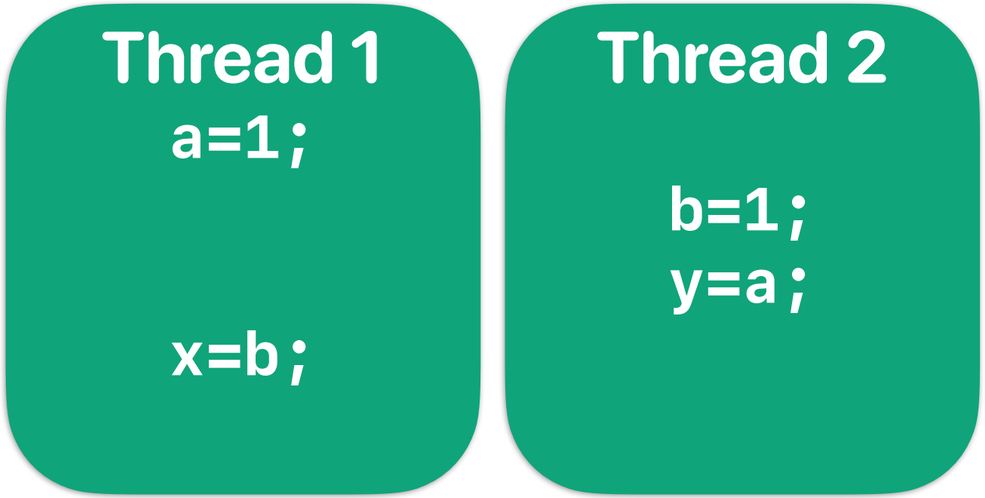
① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

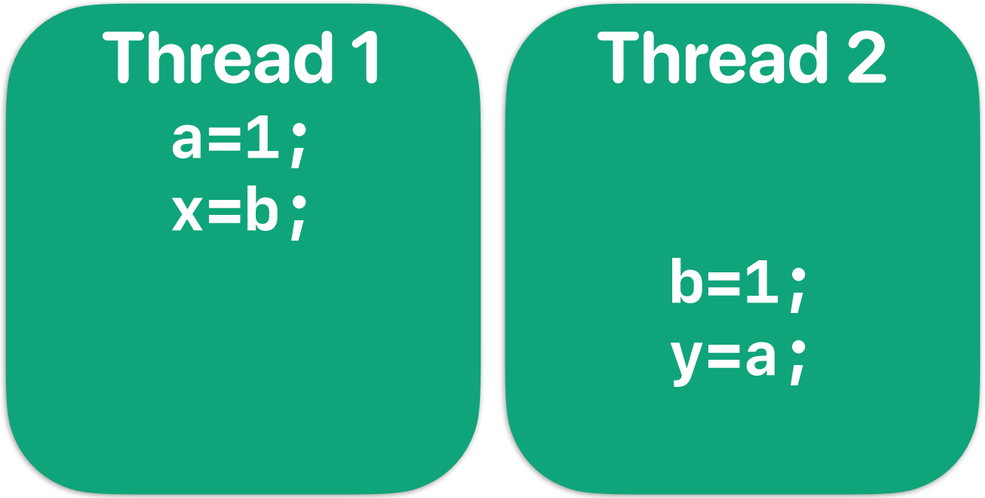
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
  a=1;
  x=b;
  return NULL;
}
void* modifyb(void *z) {
  b=1;
  y=a;
  return NULL;
}
```

```c
int main() {
  int i;
  pthread_t thread[2];
  pthread_create(&thread[0], NULL, modifya, NULL);
  pthread_create(&thread[1], NULL, modifyb, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  fprintf(stderr,"(%d, %d)\n",x,y);
  return 0;
}
```

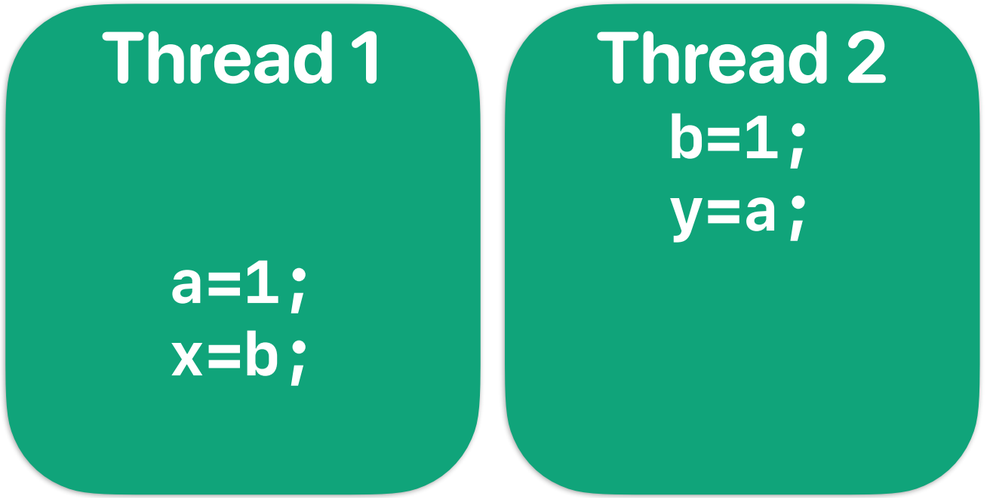
Possible scenarios



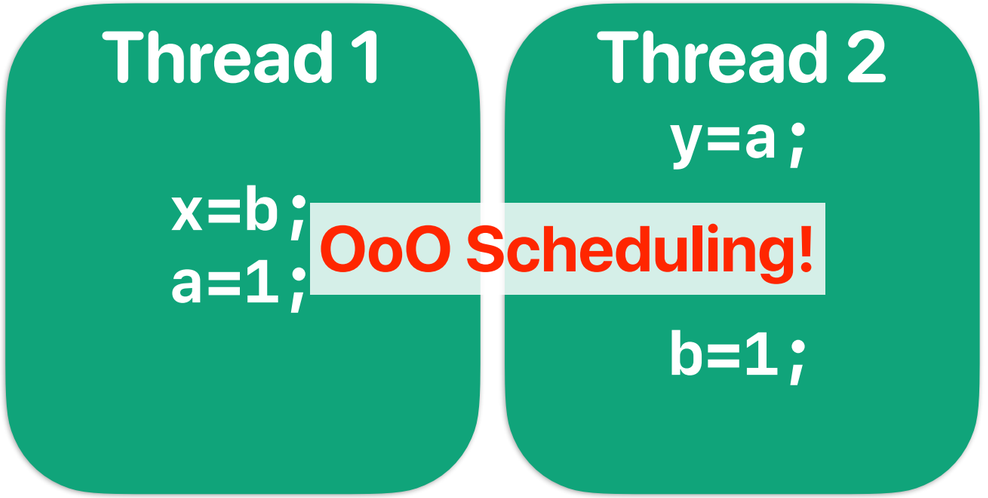
(1,1)



(0,1)



(1,0)



(0,0)

fence instructions

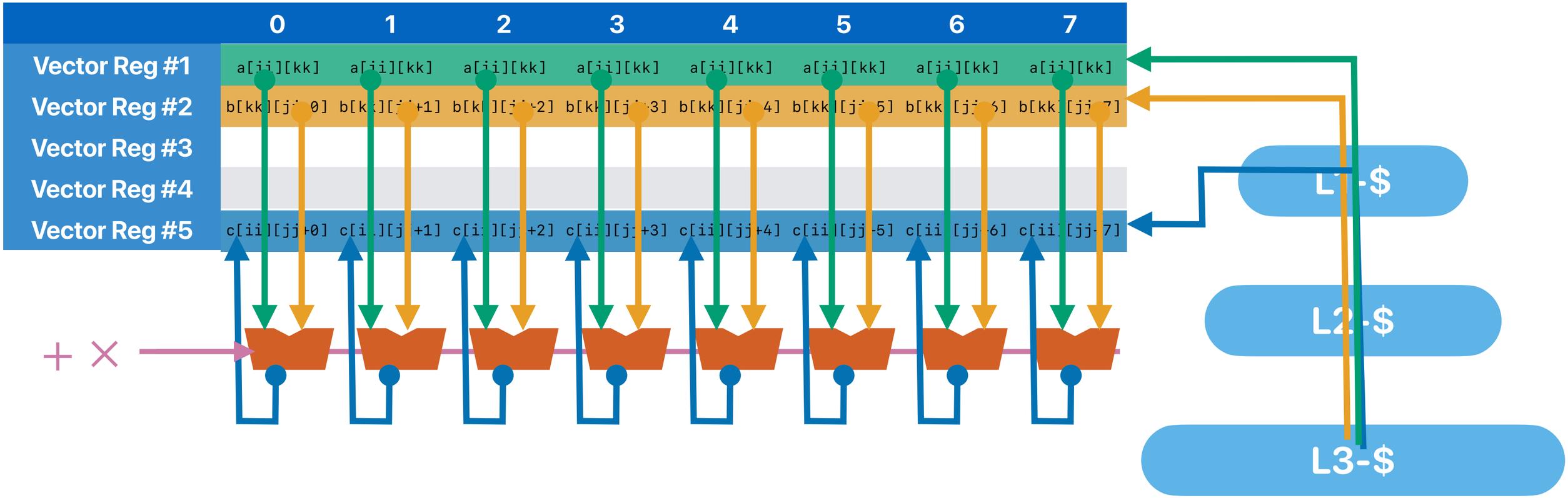
- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

thread 1	thread 2

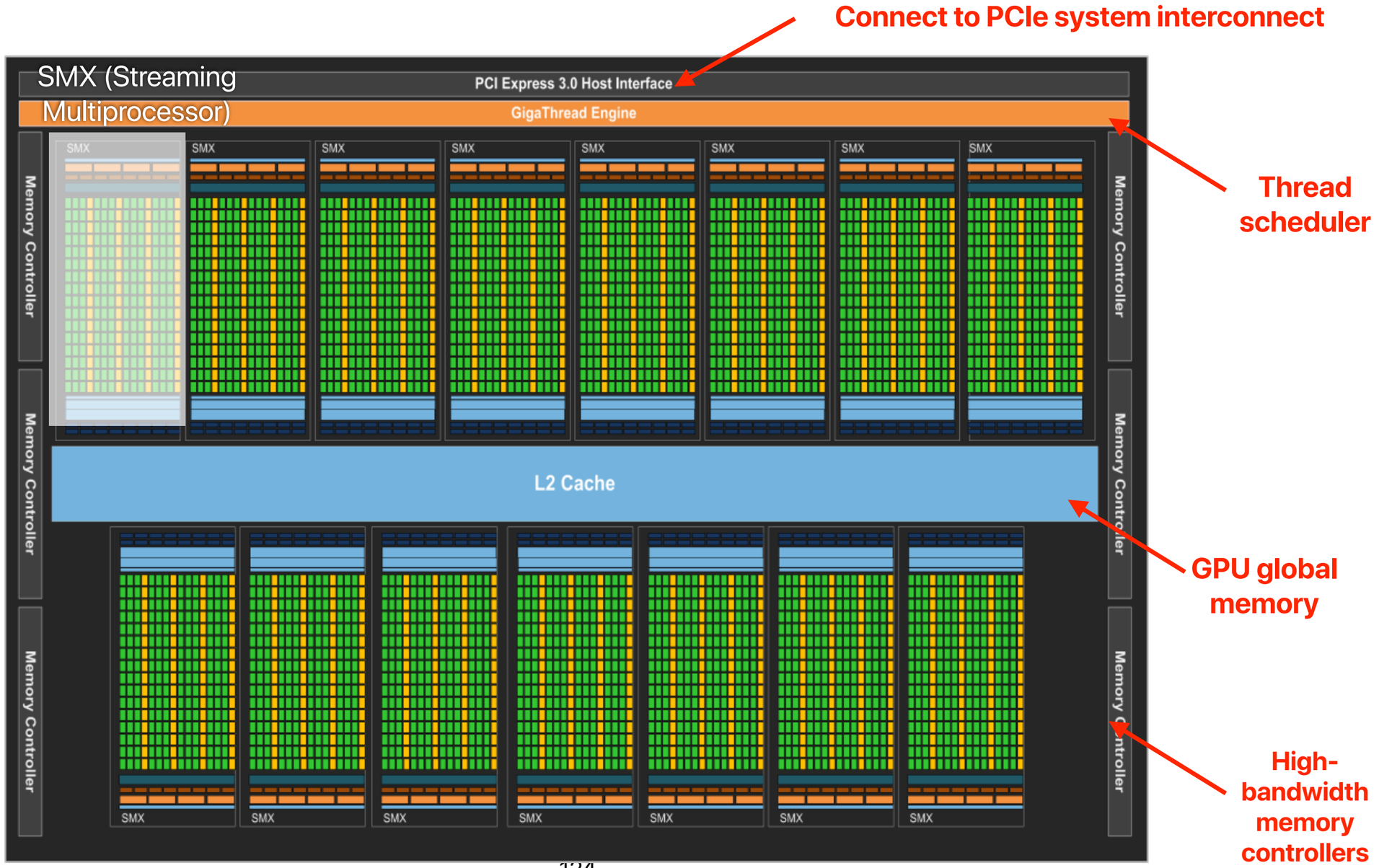
`a=1;`
`mfence` **a=1 must occur/update before mfence**
`x=b;`
	`b=1;`
	`mfence` **b=1 must occur/update before mfence**
	`y=a;`

Alternative Parallel Architectures

Vector processing architecture



Nvidia GPU architecture



What do you want from a GPU?

- Given the basic idea of shading algorithms, how many of the following statements would fit the agenda of designing a GPU?
  1. Many ALUs to process multiple pixels simultaneously
  2. Low latency memory bus to supply pixels, vectors and textures
  3. ~~High performance branch predictors~~
  4. ~~Powerful ALUs to process many different kinds of operators~~

Each frame contains 1920*1080 pixels!

Actually, high bandwidth since each pixel requires different L, N, R, V and we need to feed thousands of pixels simultaneously

not really, the behavior is uniform across all pixels

not really, we only need vector add, vector mul, vector div. Low frequency is OK since we have many threads

In terms of latency, even for 120 frames, you still have 8ms latency to get everything done!

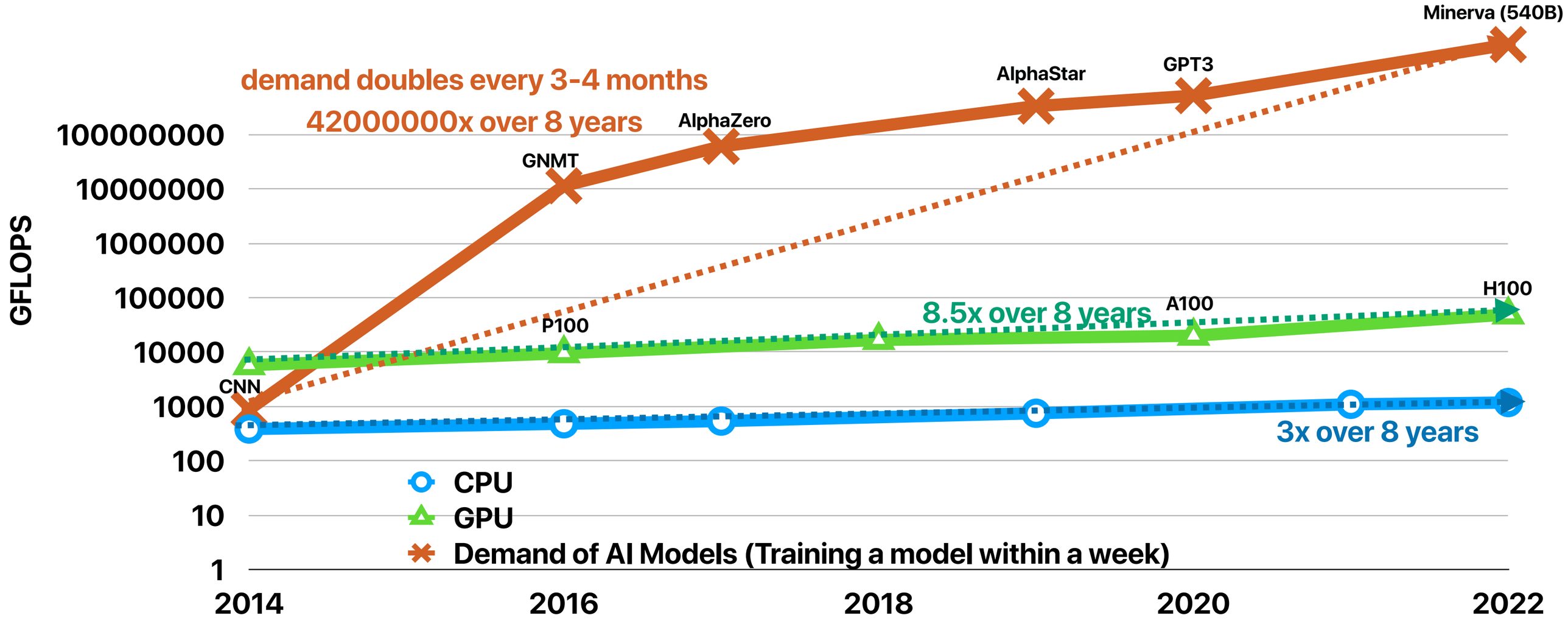
A. 0
B. 1
C. 2
D. 3
E. 4

Parallelisms

- Instruction-level parallelism — perform various, independent instructions simultaneously
  - Pipeline
  - OoO/Superscalar
- Data-level parallelism — perform the same operation on multiple data elements in parallel
  - SIMD instructions
  - Compute within an SM in GPUs
- Thread-level parallelism — perform independent computation streams (composed of many instructions or SIMD instructions)
  - Multicore/SMT processors
  - Compute using multiple SMs in GPUs

Power/Energy/Dark Silicon

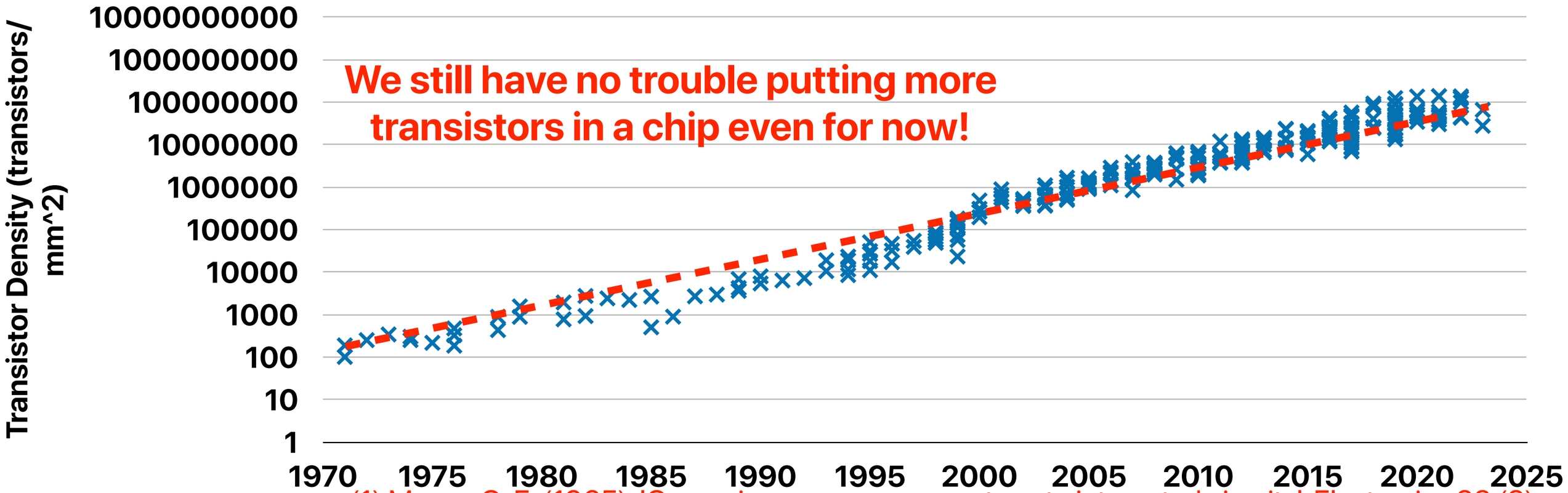
Mis-matching AI/ML demand and general-purpose processing



https://ourworldindata.org/grapher/artificial-intelligence-training-computation

Moore's Law[1]

- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains

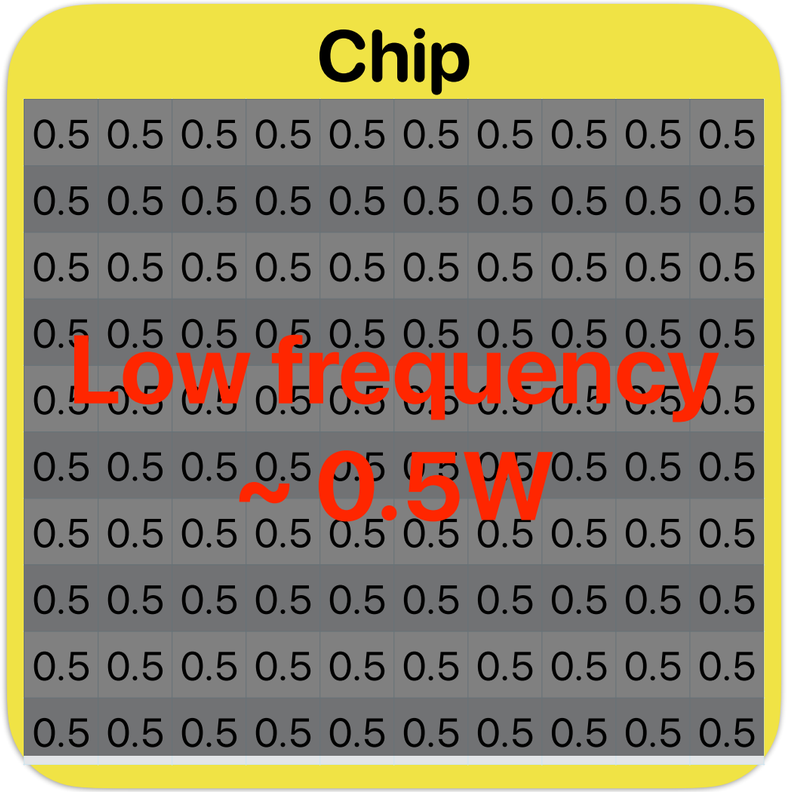


(1) Moore, G. E. (1965), 'Cramming more components onto integrated circuits', Electronics 38 (8) .

Plot based on https://en.wikipedia.org/wiki/Transistor_count by Hung-Wei Tseng

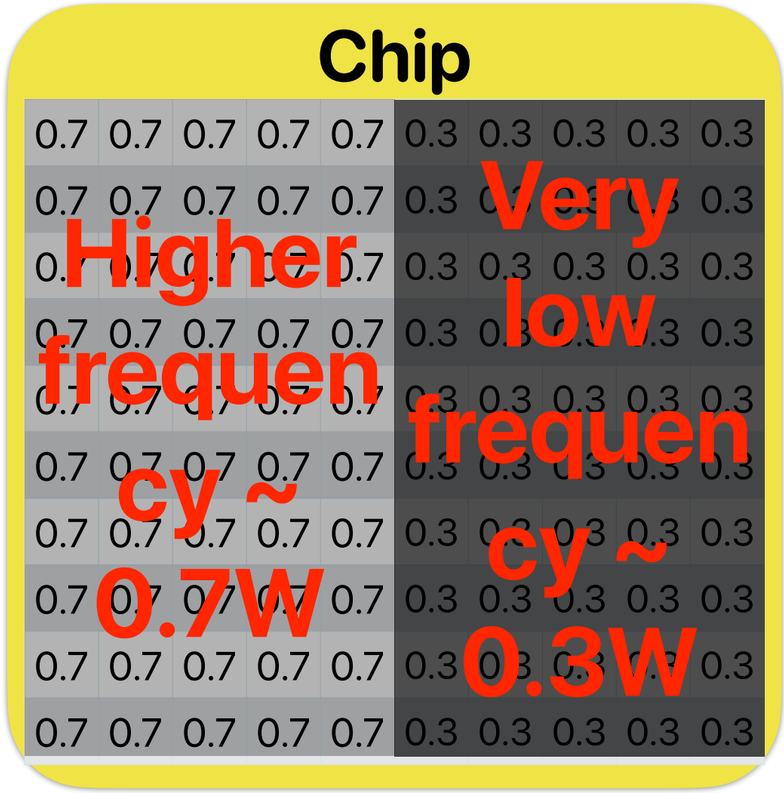
Given the same power budget, maximize the efficiency per chip

Slowdown all of them



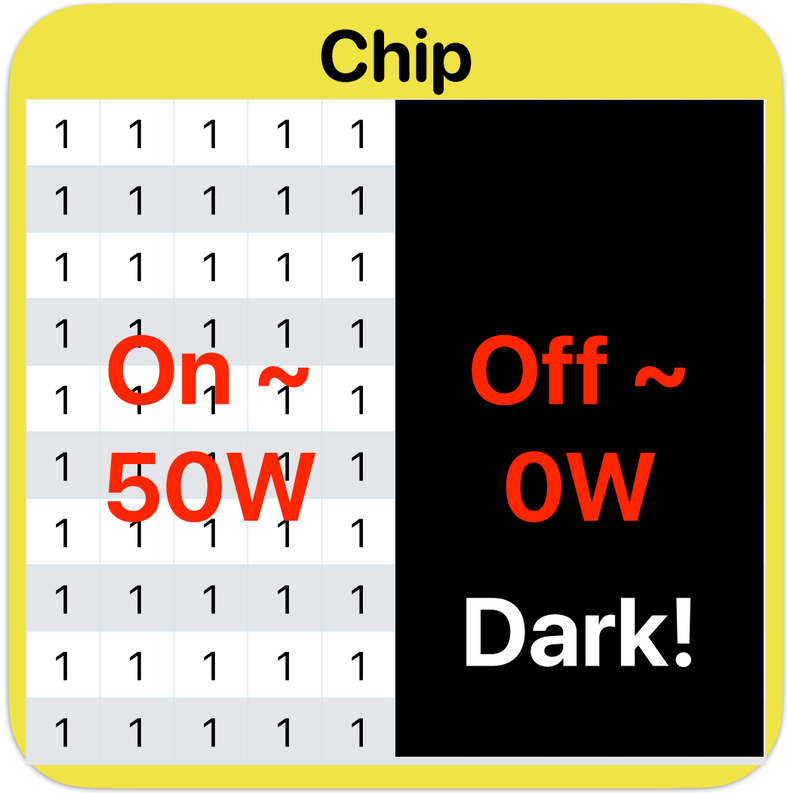
=50W!

Some faster, some slower



=50W!

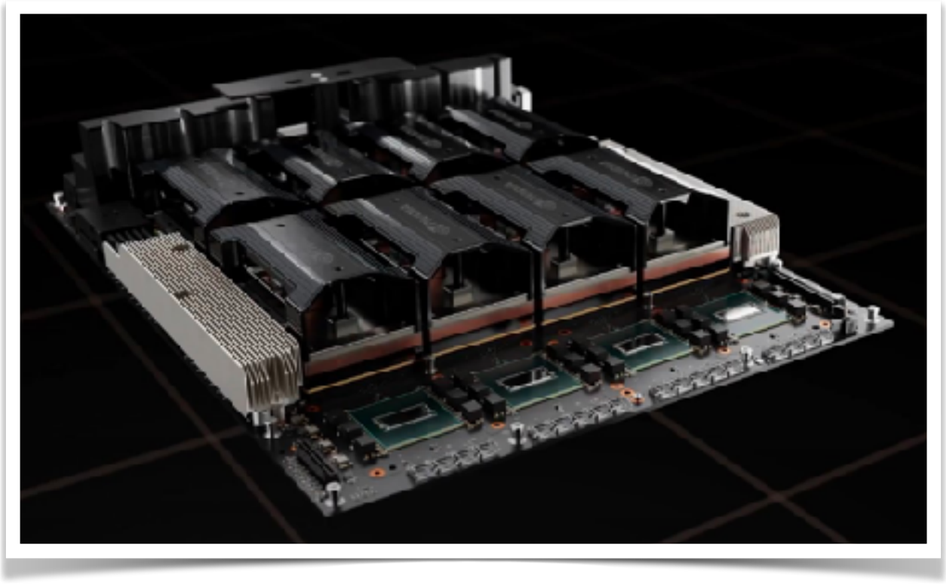
Some at top speed, some are not functioning



If you can add power budget…

NVIDIA Accelerator Specification Comparison			
	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)
GPU	GH100 (814mm2)	GA100 (826mm2)	GV100 (815mm2)
Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FFN
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta

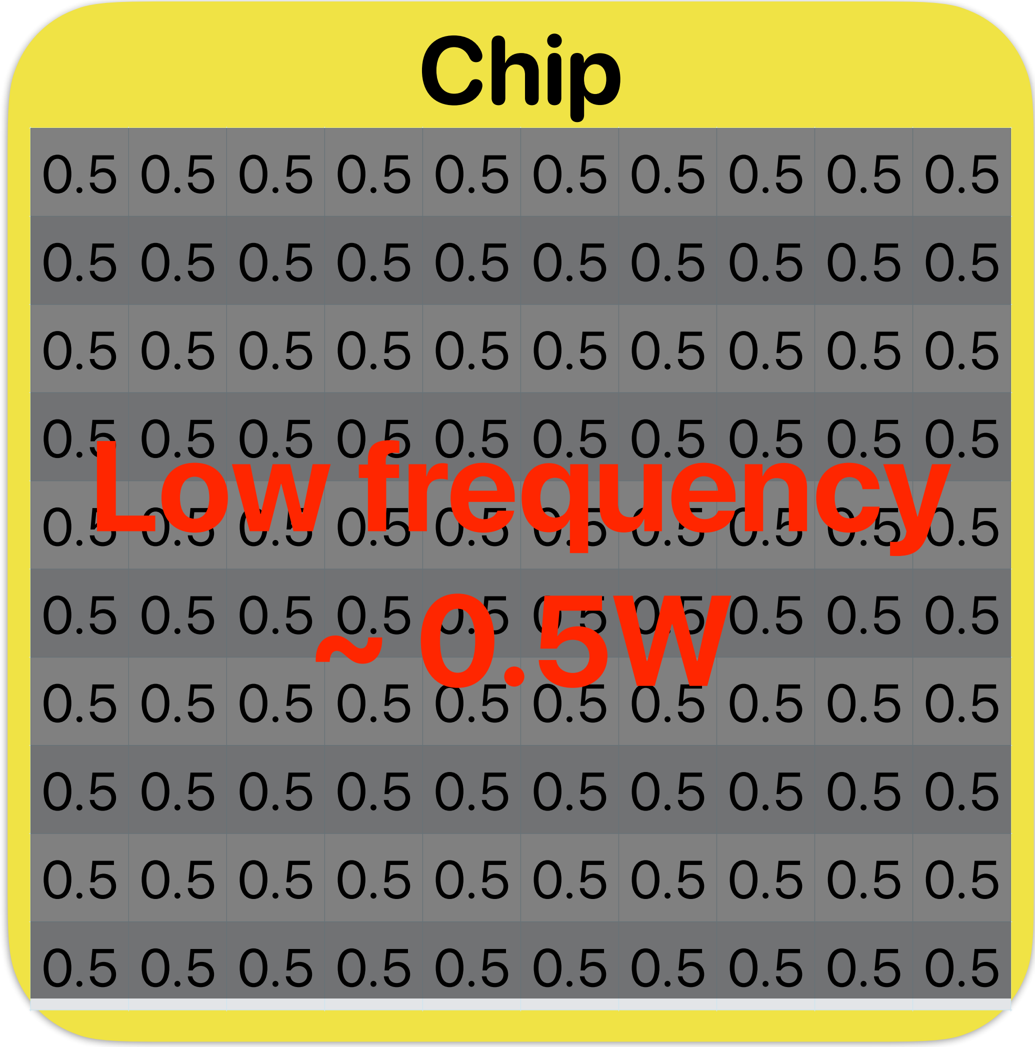
https://www.workstationspecialist.com/product/nvidia-tesla-a100/



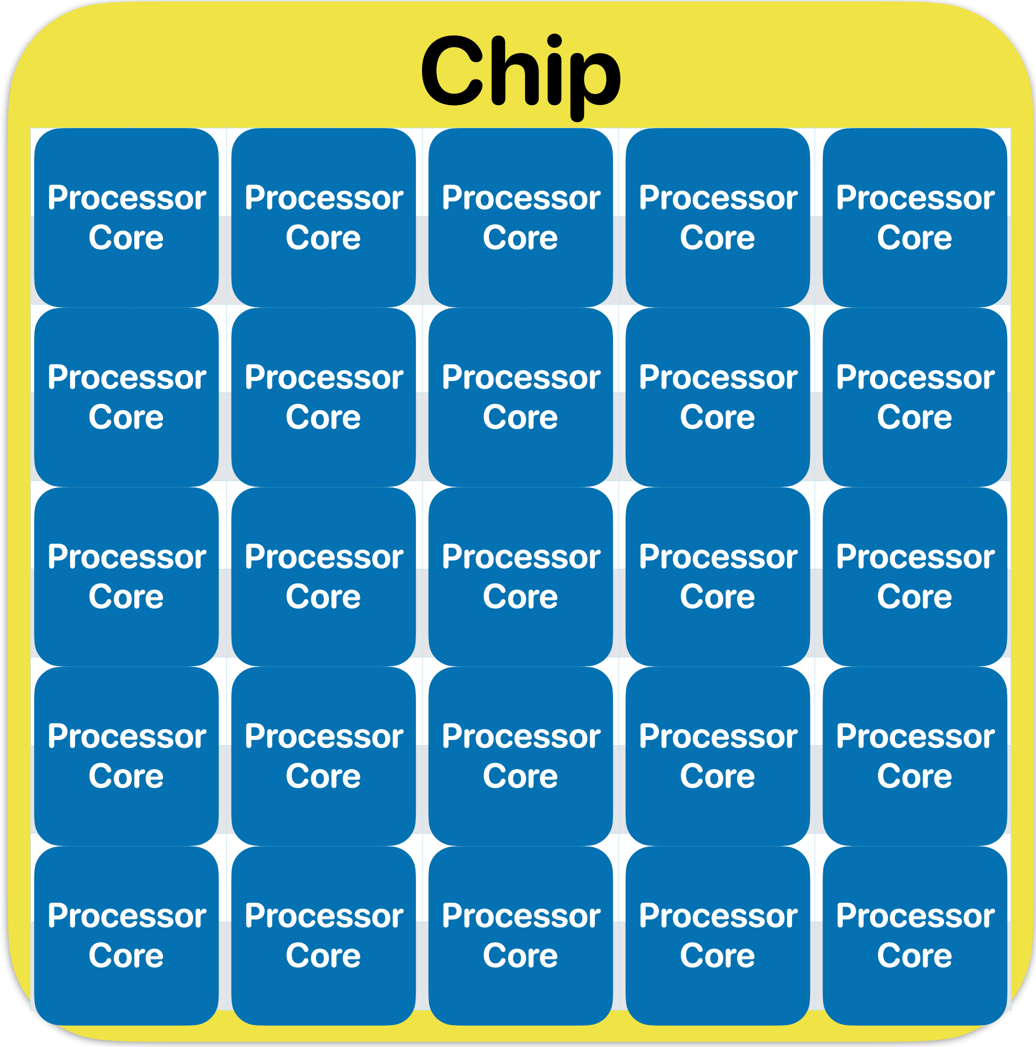
https://www.servethehome.com/wp-content/uploads/2022/03/NVIDIA-GTC-2022-H100-in-HGX-H100.jpg

Given the same power budget, maximize the efficiency per chip

Slowdown all of them



=50W!



All of them are smaller, simpler, lower-frequency, lower-power cores

What's the "appropriate" GPU architecture

- Lots of ALUs to process pixels in parallel — 2M pixels in HD resolution, very regular workloads
  - Vector processing model
- Simple operations
  - The ALUs only supports very few instructions
  - Almost no branches
- Deadline driven and throughput-oriented rather than latency oriented
  - High-bandwidth but also "higher-latency" memory
  - ALUs can be slower

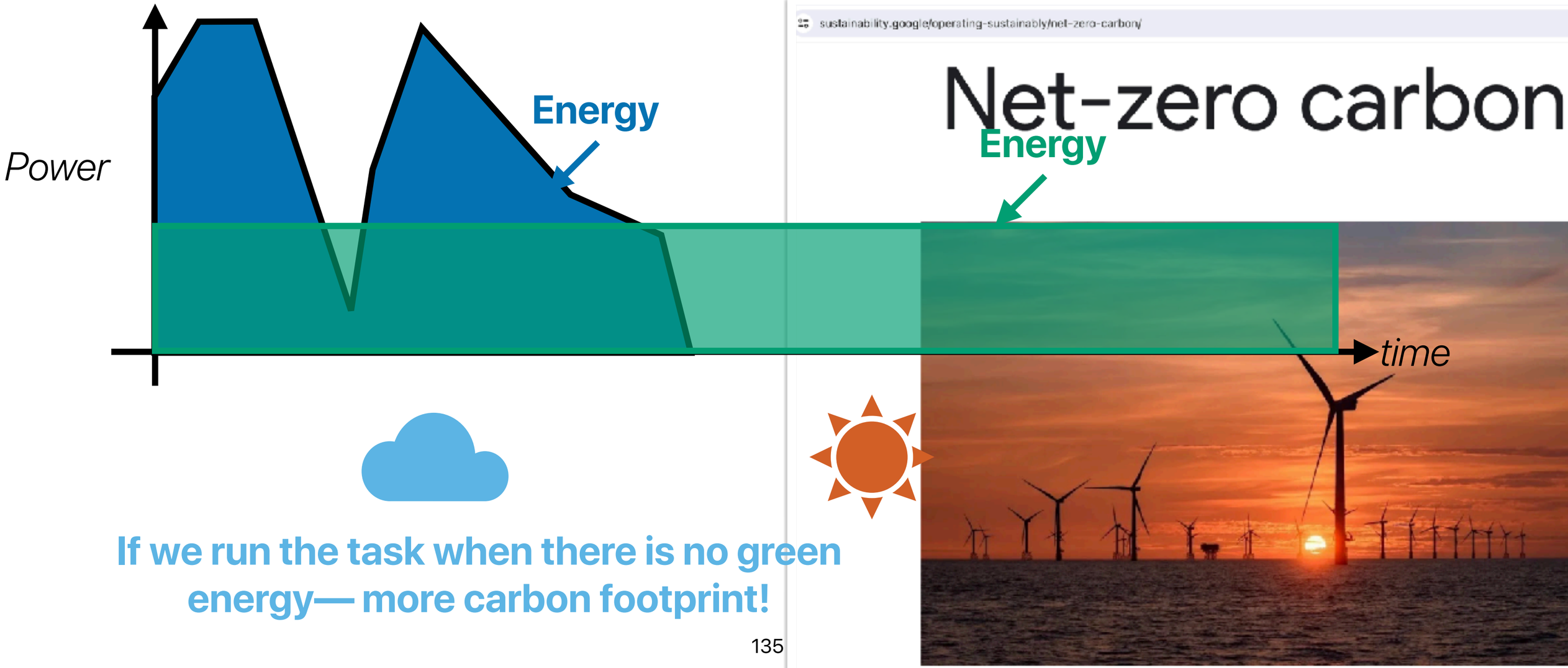
GPU also follows the idea of slower, but more!

Power v.s. Energy

- Power is the direct contributor of “heat”
  - Packaging of the chip
  - Heat dissipation cost
  - Dynamic power
  - Leakage power
- Energy = $Power \times Execution\_Time$
  - The electricity bill and battery life is related to energy!
  - Lower power does not necessary means better battery life if the processor slow down the application too much

Power/Energy/Carbon footprint

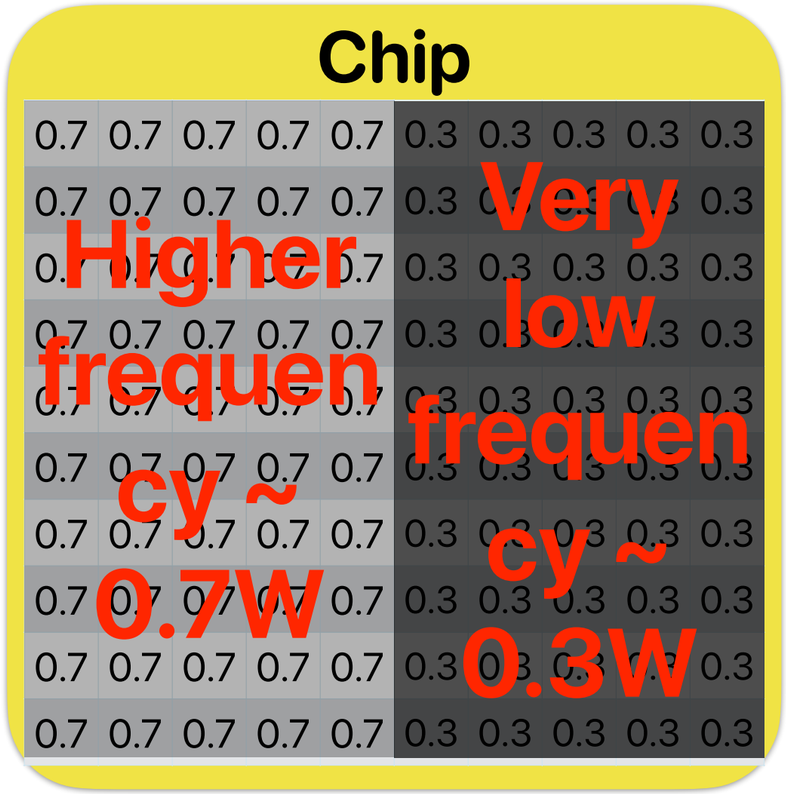
The Green can be more if power is not low enough

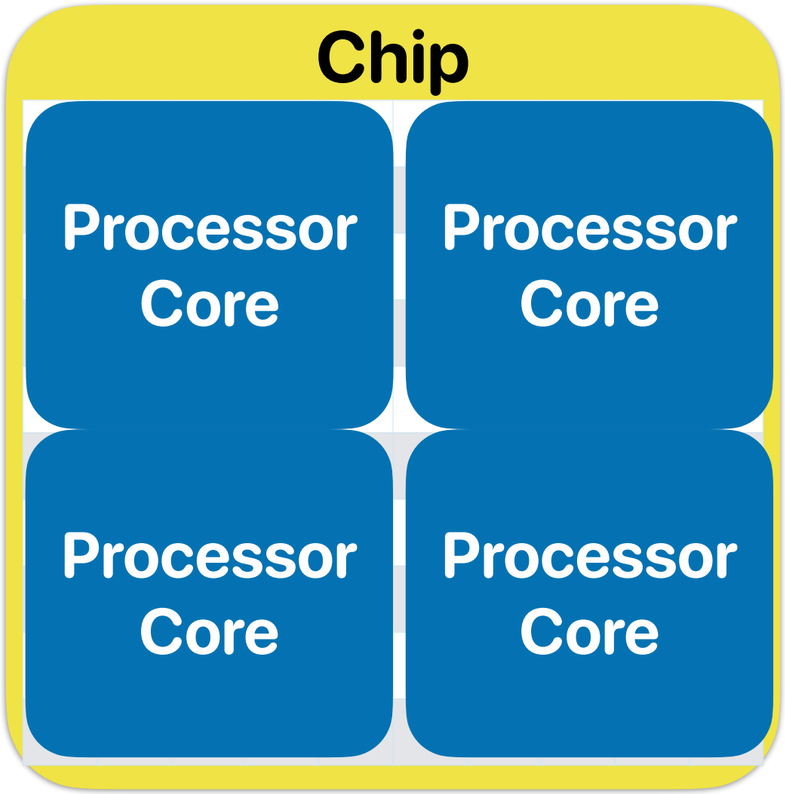


If we run the task when there is no green energy— more carbon footprint!

Given the same power budget, maximize the efficiency per chip

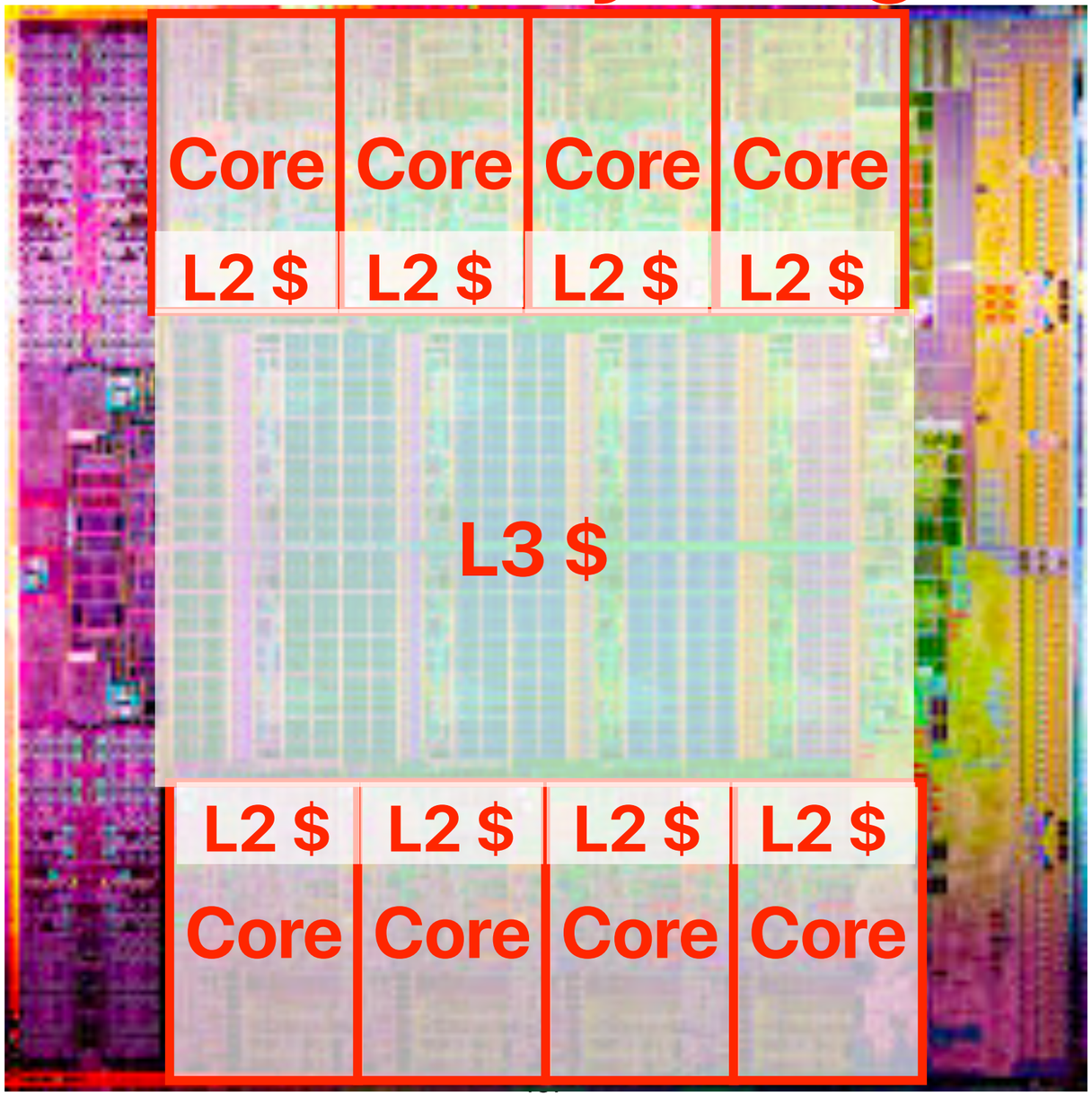
Some faster, some slower

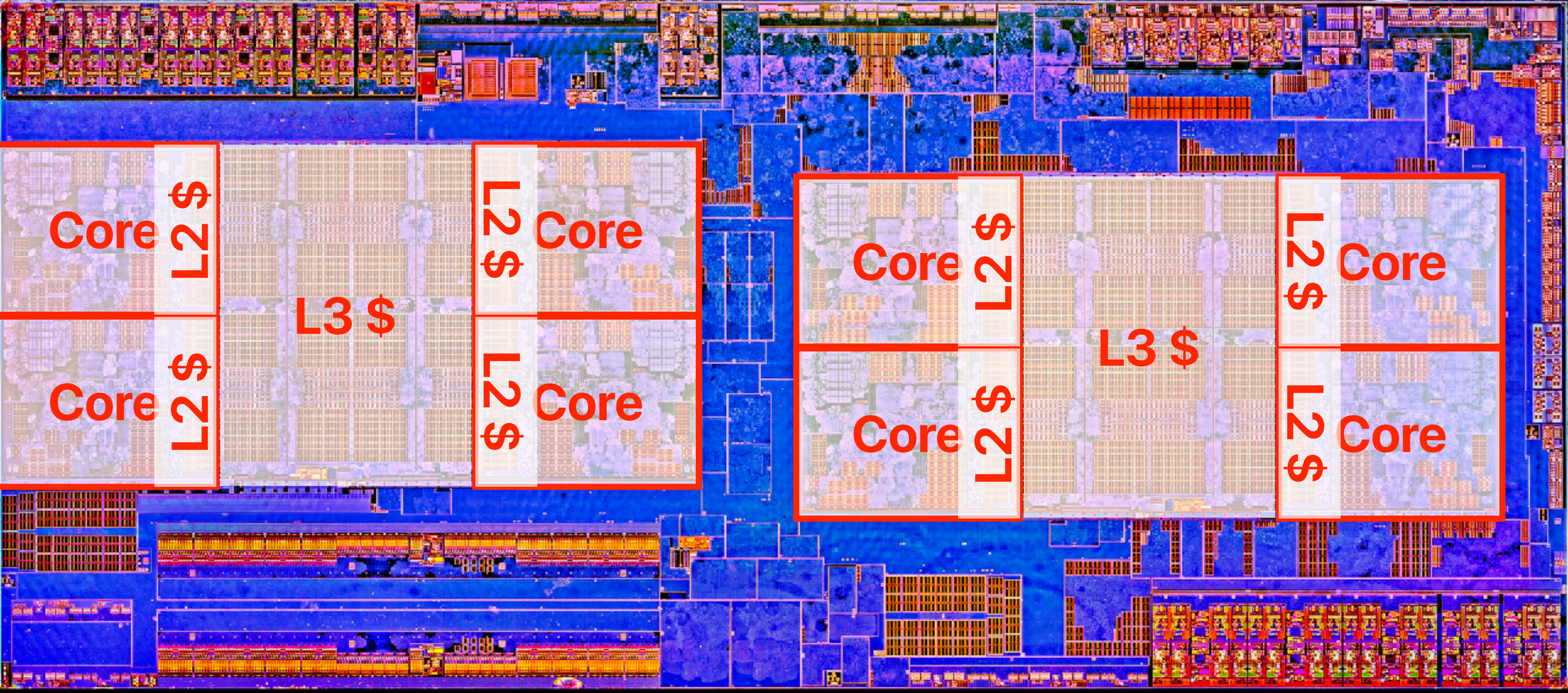




Aggressively adjust the frequency of processor cores — If only one program is compute-intensive, having it running at full speed, others at lower frequency or turning off

Intel Sandy Bridge



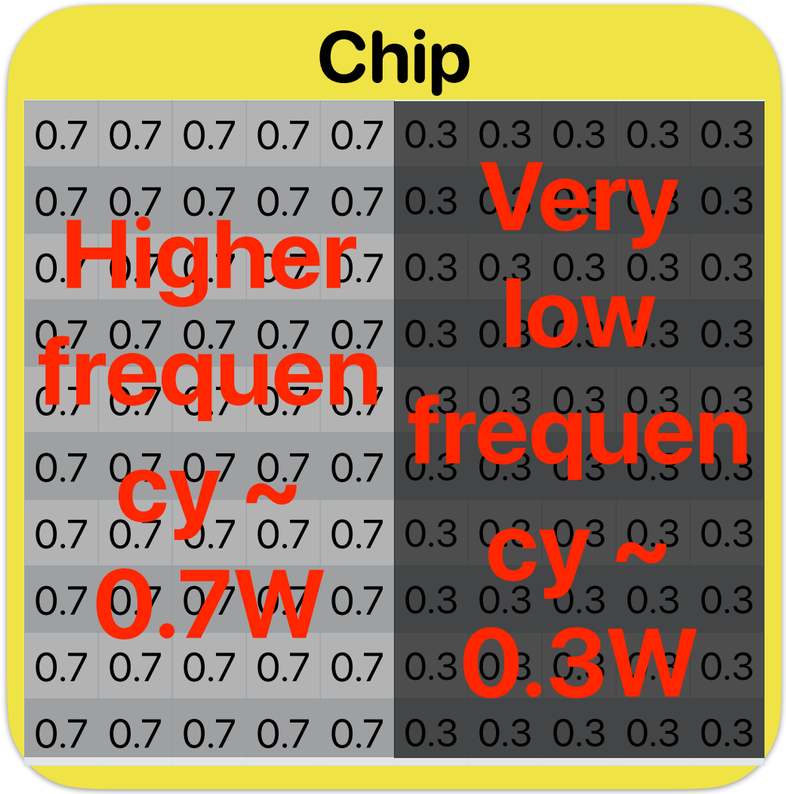

Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
AMD
RYZEN

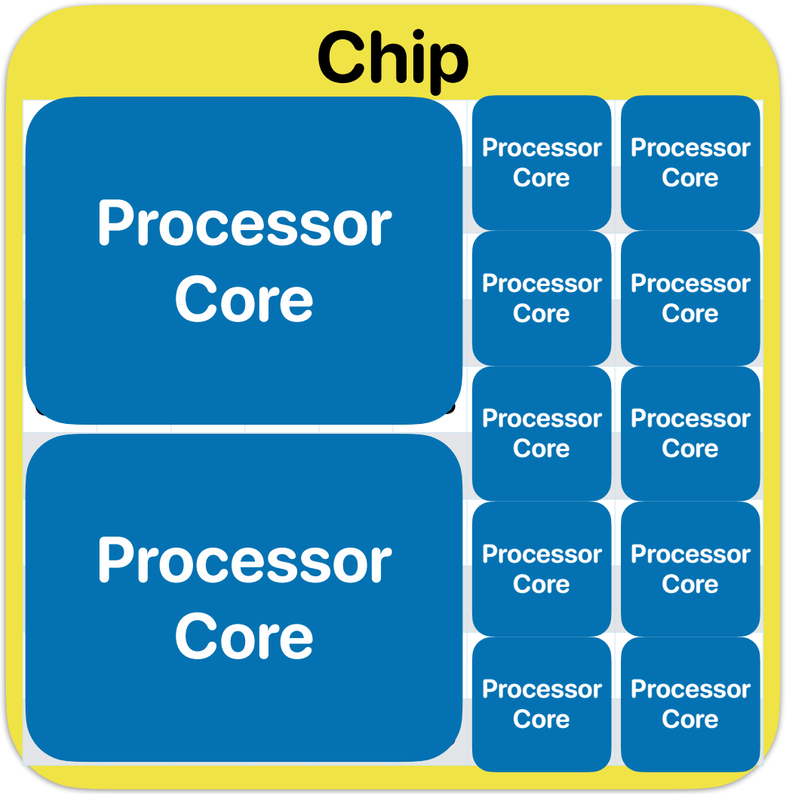
Take-aways: the new golden age of computer architectures

- Challenges and SOTA solutions in the dark silicon era
  - GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads
  - Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great

Given the same power budget, maximize the efficiency per chip

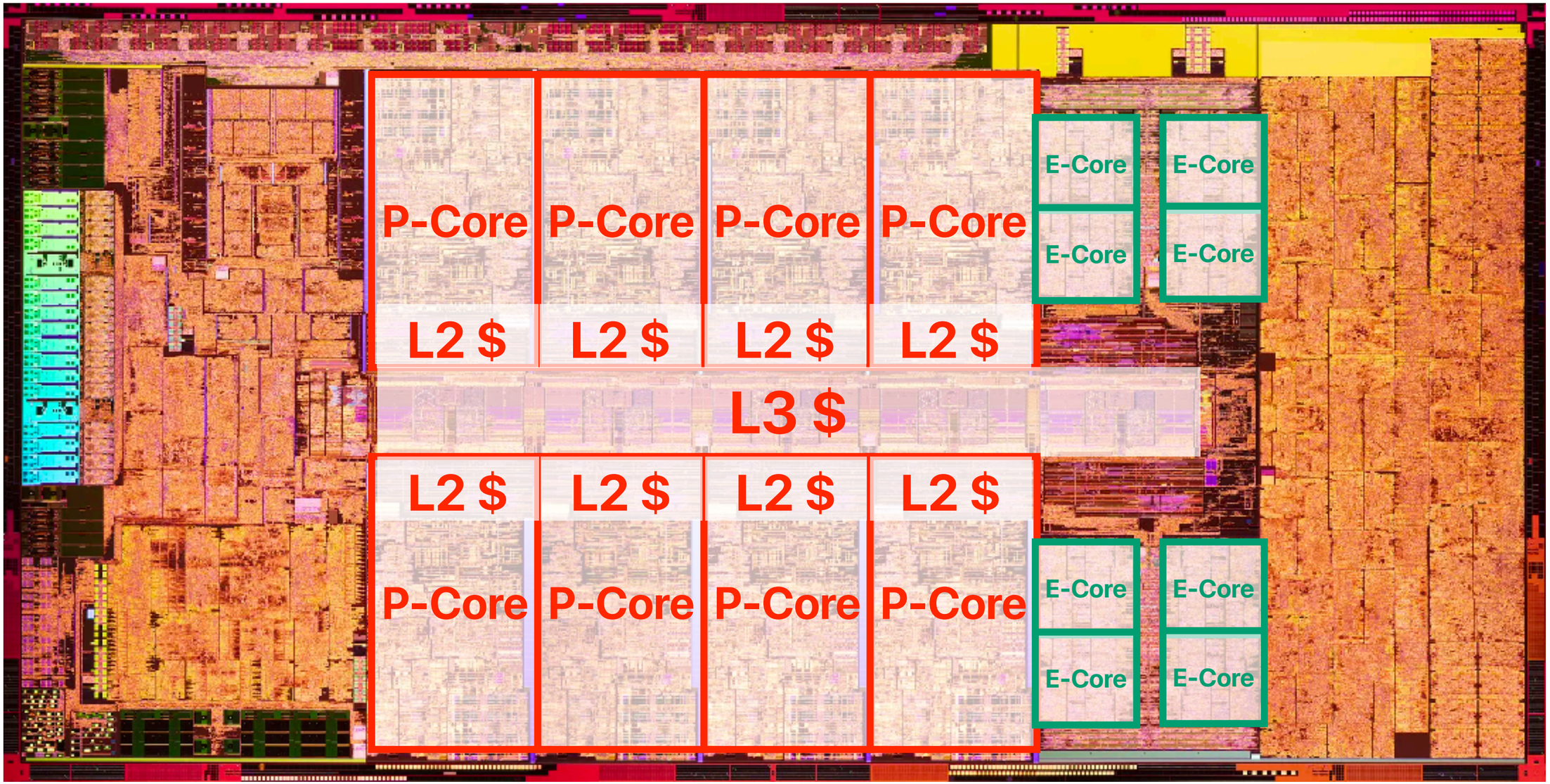
Some faster, some slower



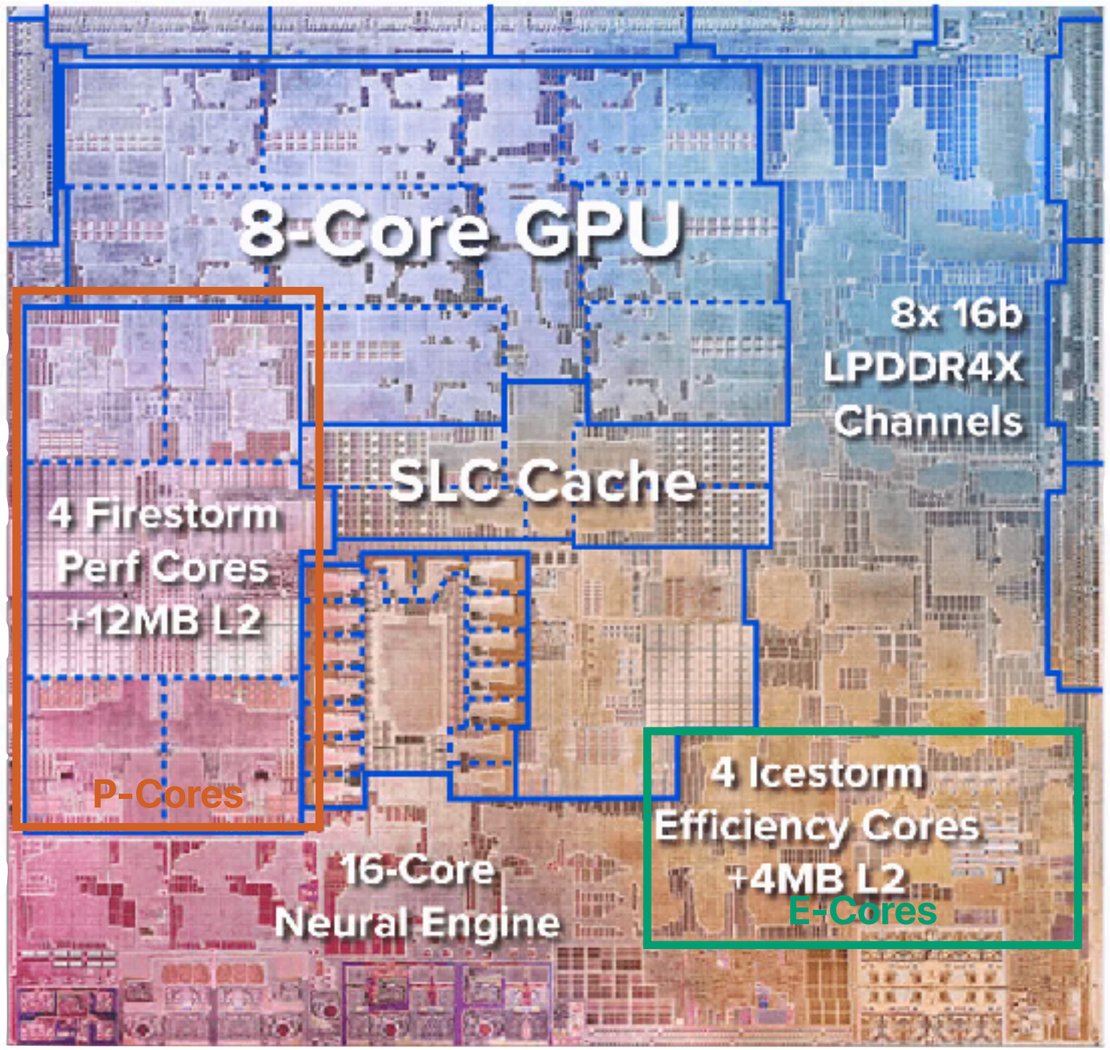


Some powerful cores for latency sensitive workloads, many small cores for highly parallelizable workloads

Intel Alder Lake



Single ISA heterogeneous CMP in Apple's M1



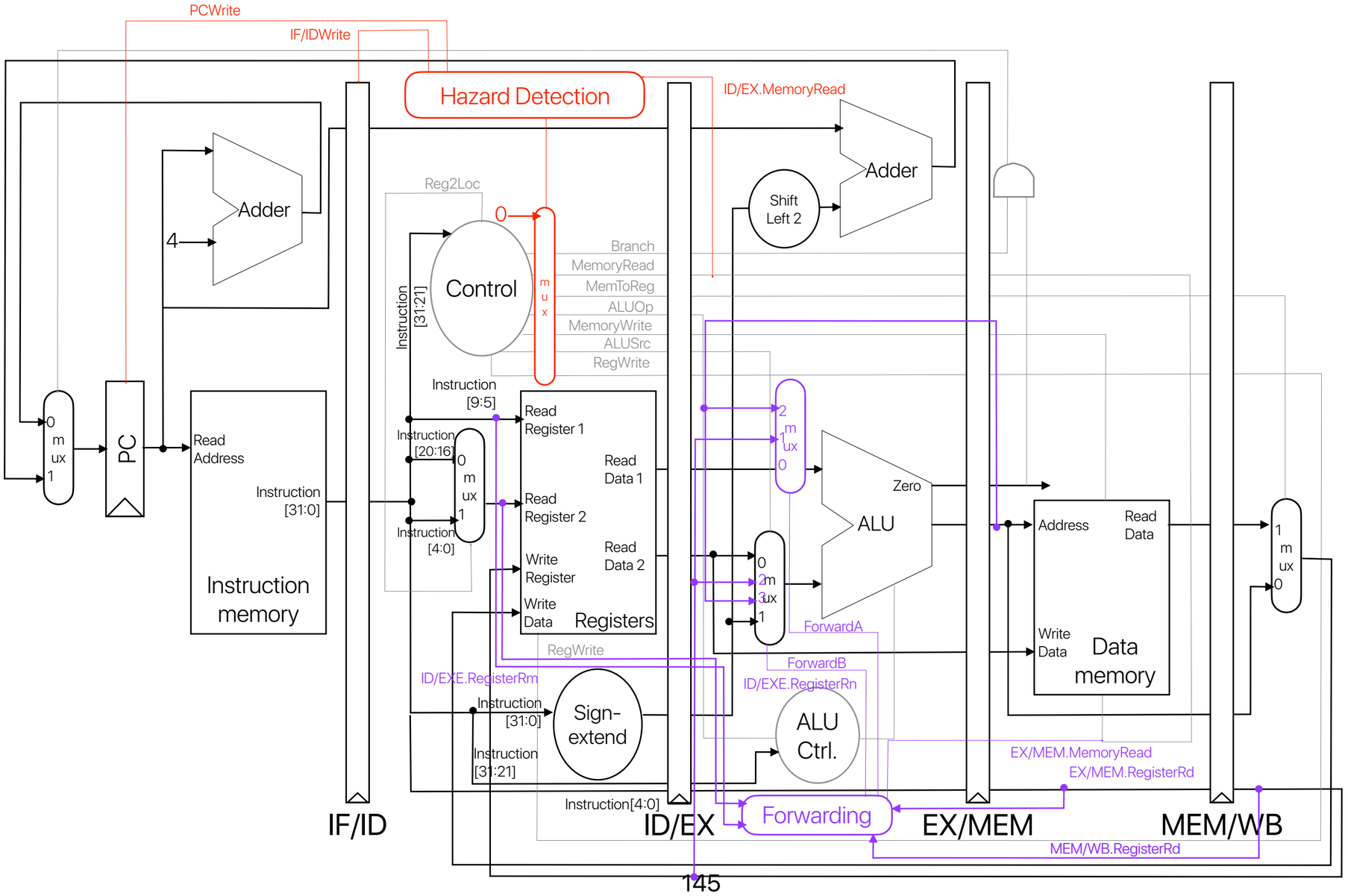
https://www.toptal.com/apple/apple-m1-processor-compatibility-overview

Take-aways: the new golden age of computer architectures

- Challenges and SOTA solutions in the dark silicon era
  - GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads
  - Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great
  - Single ISA, heterogeneous CMPs (e.g., big.Little cores, Intel/Apple's P-cores/E-cores) find a balance the trade-offs of general-purpose workloads, but won't be ideal if your applications go to either extreme of throughput (massive parallelism) or latency

Can we maximize both latency & throughput?

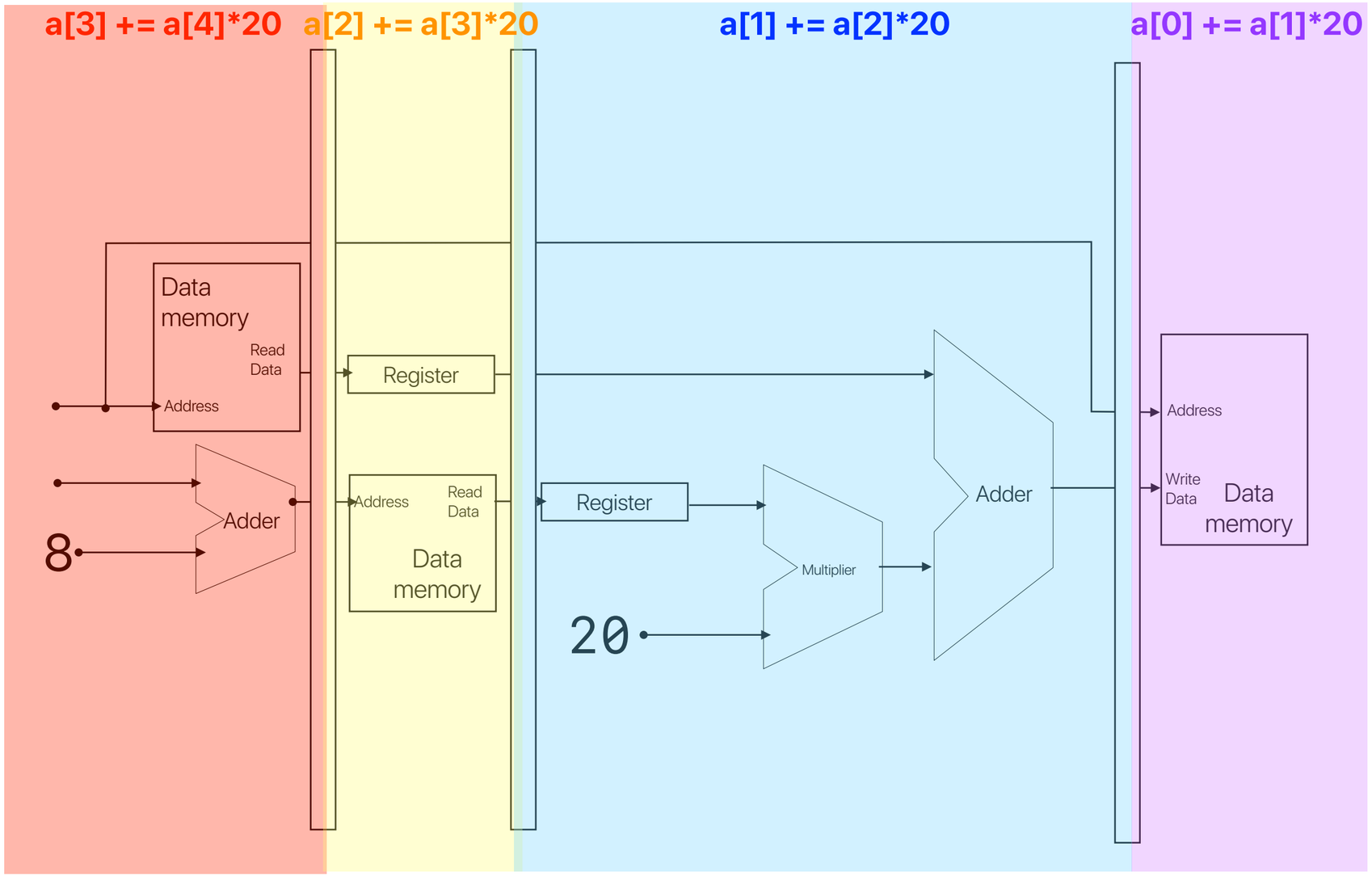
This is what you need for these instructions



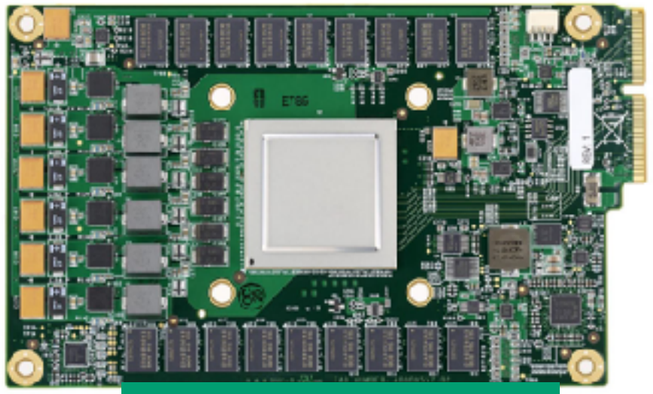
The pipeline for a[i] += a[i+1]*20

Each stage can still be as fast as the pipelined processor

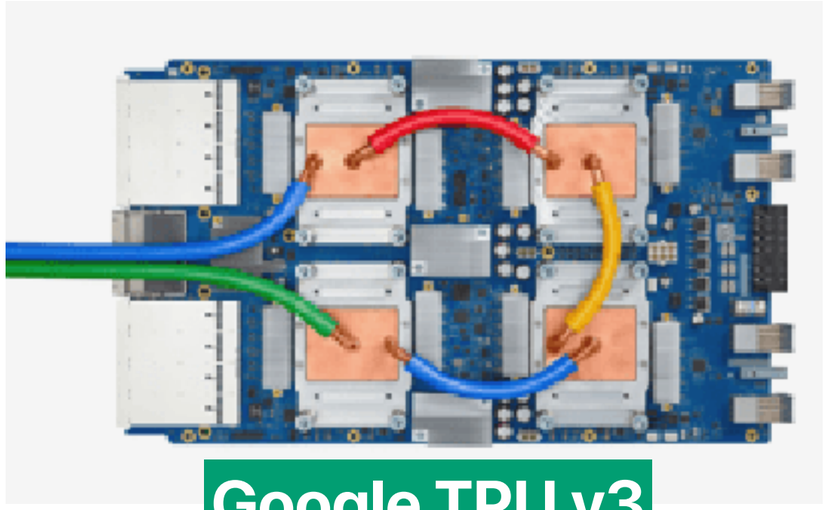
But each stage is now working on what the original 6 instructions would do



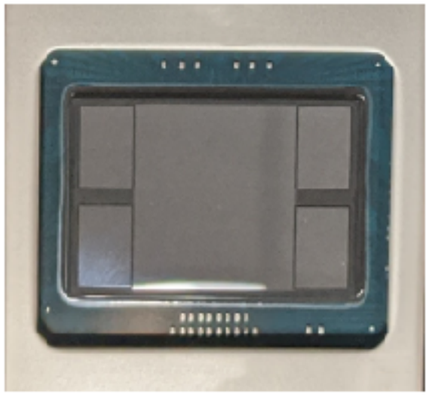
Tensor Processing Units

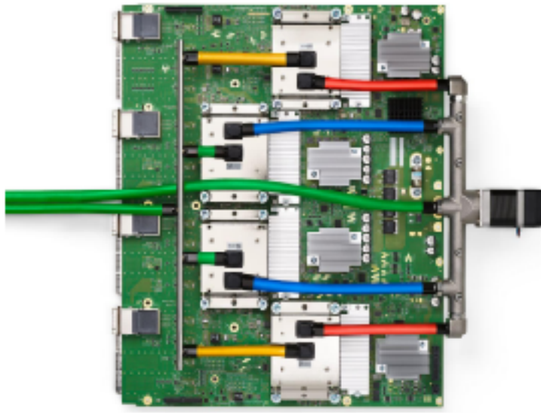


Google TPU v1

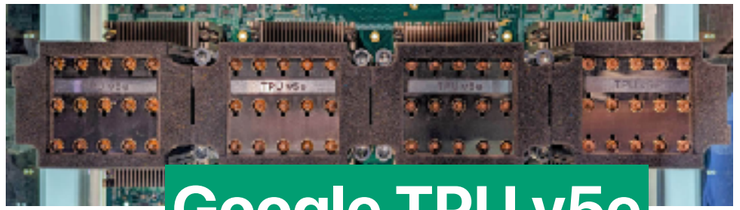


Google TPU v3





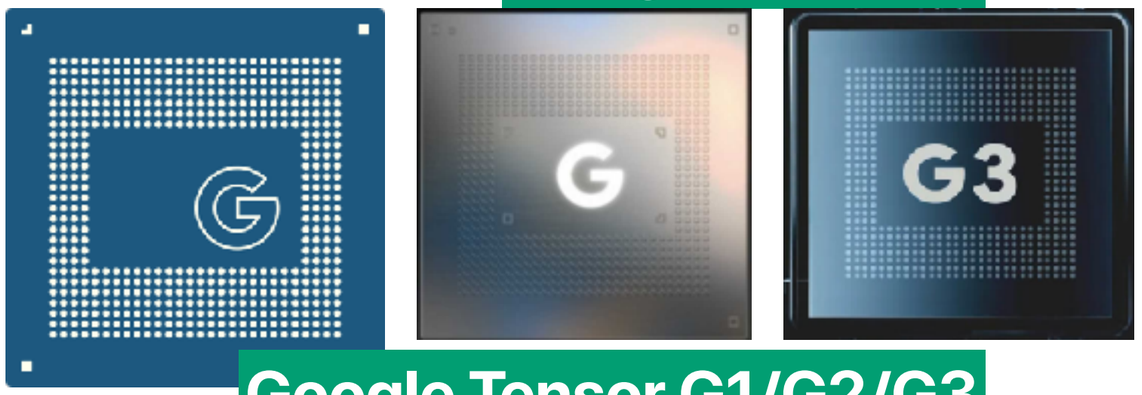
Google TPU v4



Google TPU v5e

Edge TPU

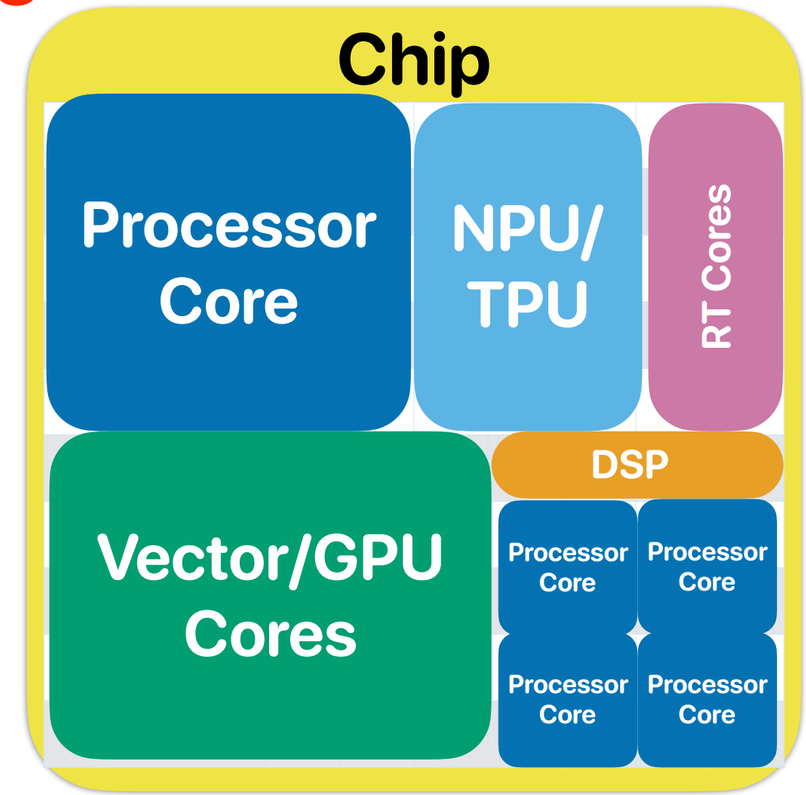
Google TPU v2



Google Tensor G1/G2/G3

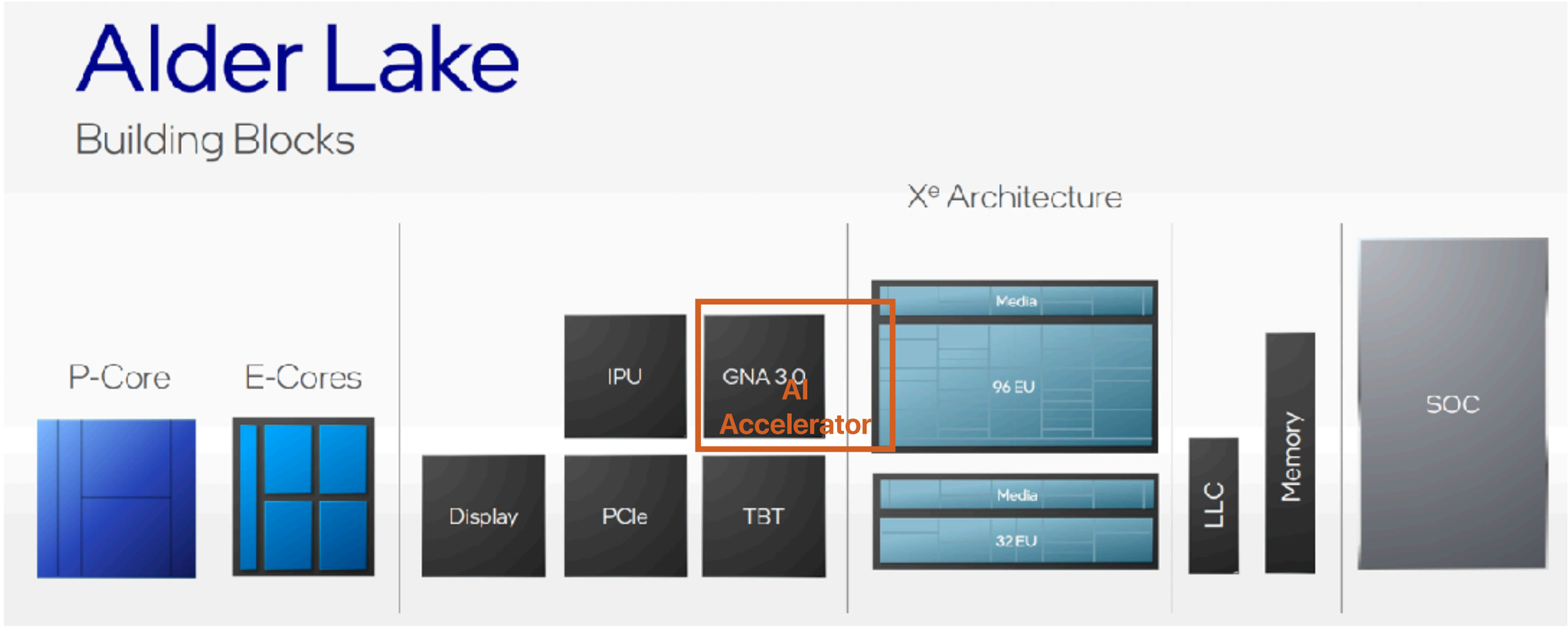
Given the same power budget, maximize the efficiency per chip

Some at top speed, some are not functioning.

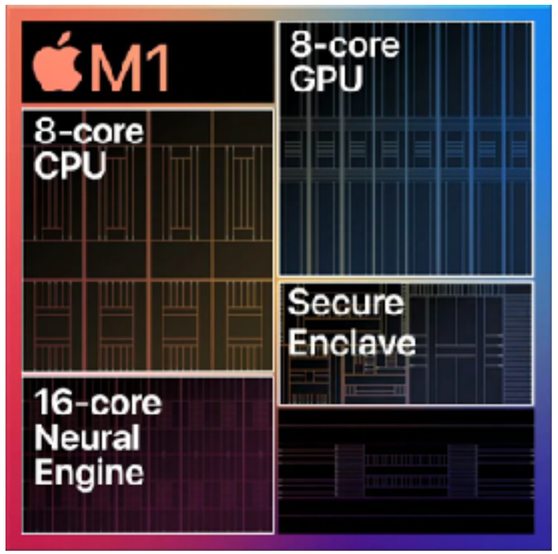


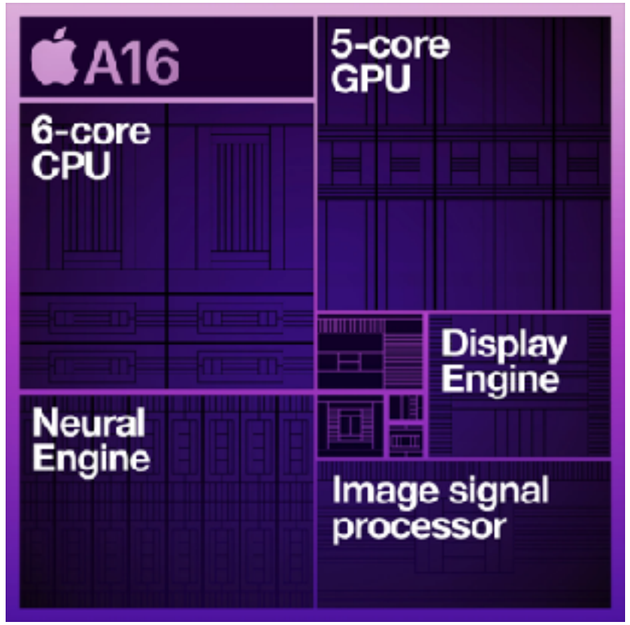
Turn off unnecessary ones when we don't need specialized functions!

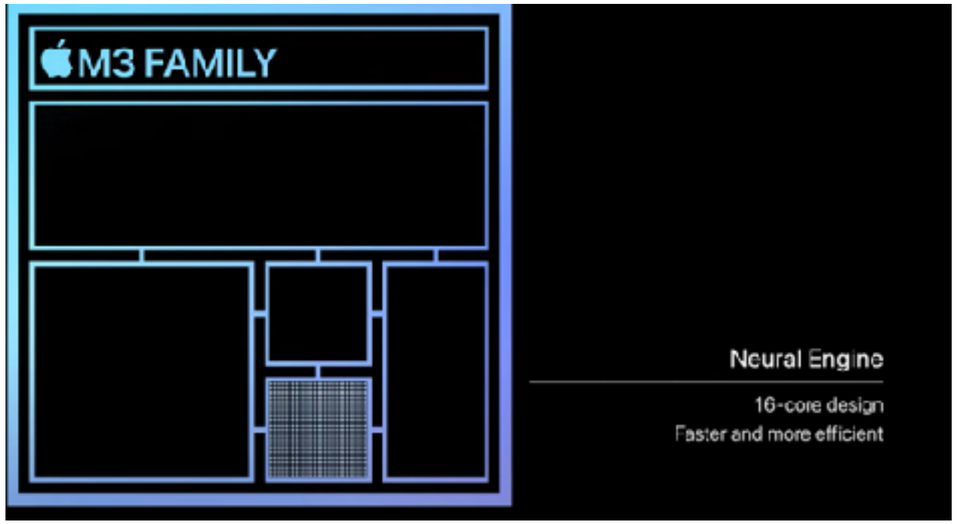
Modern processors also contain “accelerators”

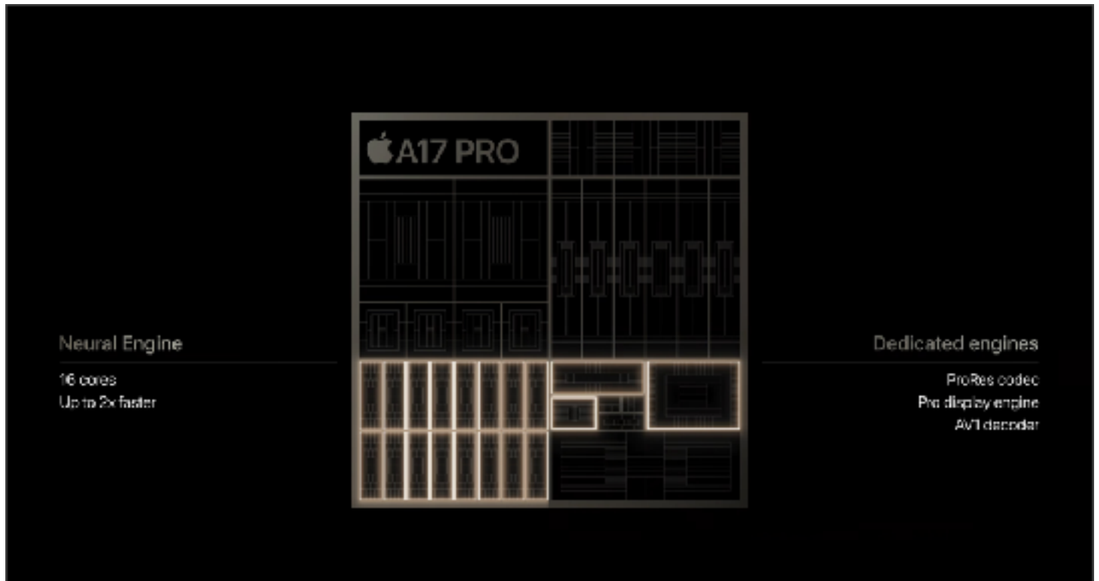


Apple's Neural Engine



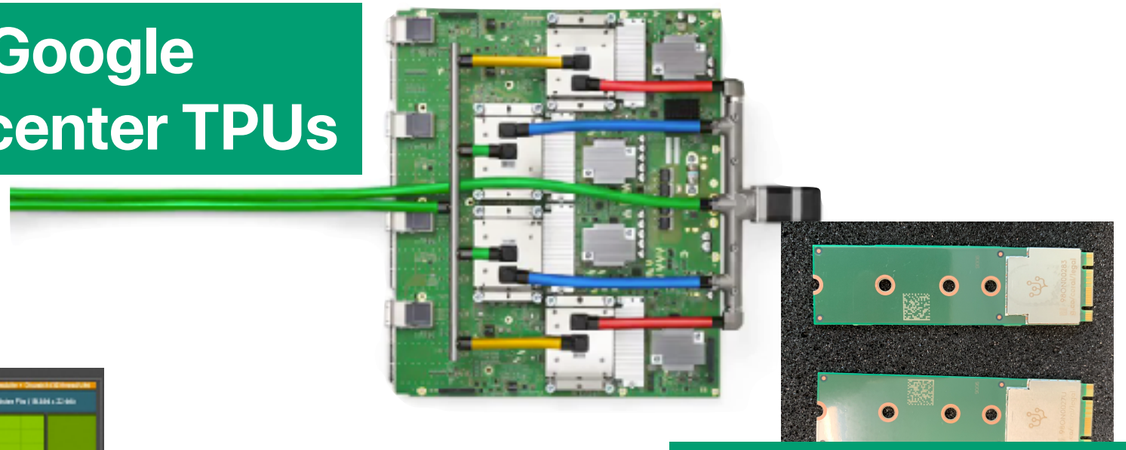






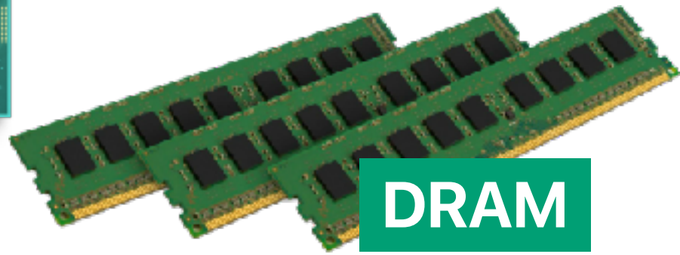
The “landscape” of modern computers

Google Datacenter TPUs



Google Edge TPUs

CPU

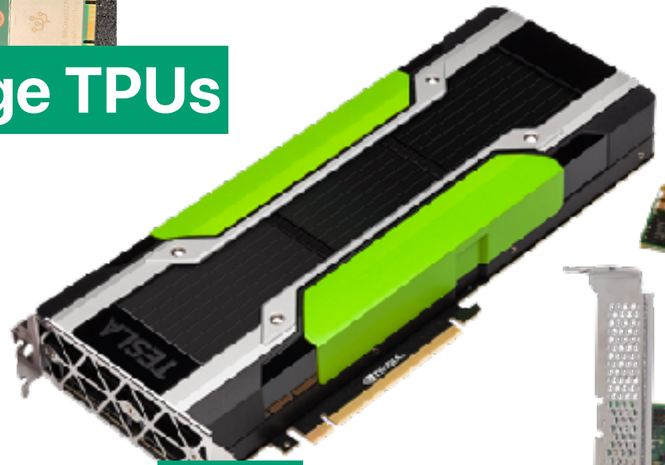


DRAM

NVIDIA Tensor Core Units

AI/ML Accelerators

Apple Neural Engines



GPU

SSD

NIC

Data movement overhead before calling GPUs

```
// Allocate memory space on the device
D_TYPE *d_a, *d_b, *d_c;
cudaMalloc((void **) &d_a, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);
cudaMalloc((void **) &d_b, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);
cudaMalloc((void **) &d_c, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);

// copy matrix A and B from host to device memory
cudaMemcpy(d_a, a, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyHostToDevice);

unsigned int grid_rows = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
unsigned int grid_cols = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

// Launch kernel
gpu_block_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, ARRAY_SIZE);

// Transefr results from device to host
cudaMemcpy(c, d_c, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyDeviceToHost);
cudaThreadSynchronize();
// time counting terminate
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

// compute time elapse on GPU computing
cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
```

Take-aways: the new golden age of computer architectures

- Challenges and SOTA solutions in the dark silicon era
  - GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads
  - Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great
  - Single ISA, heterogeneous CMPs (e.g., big.Little cores, Intel/Apple's P-cores/E-cores) find a balance the trade-offs of general-purpose workloads, but won't be ideal if your applications go to either extreme of throughput or latency
  - Domain-specific accelerators or application-specific ICs make more efficient use of chip areas to fulfill the latency or throughput demand of applications, but sacrifices flexibility and application designers are now also hardware designers

Conclusion

- Computer architecture is now more important than you could ever imagine in the dark silicon era + the "new" golden age of computer architecture
- Just being a "programmer" is easy. You need to know architecture a lot to be a "performance programmer"
  - Optimizing locality and footprint for caching
  - Make your code or data more branch prediction friendly
  - Exploiting more instruction-level parallelism via carefully revisiting the critical path of execution
- Multicore era — to get your multithreaded program correct and perform well, you need to take care of coherence and consistency
- We're now in the "dark silicon era"
  - Single-core isn't getting any faster
  - Multi-core doesn't scale anymore
  - We will see more and more ASICs
  - You need to write more "system-level" programs to use these new ASICs.
- **Can we be more creative in the New Golden Age of Computer Architecture?**

Announcements

- **Course Evaluation** started and ends on 9/05/2024
  - Submit the prove of your participation in course evaluation through Gradescope
  - It can become a full credit reading quiz (it helps to amortize the penalty of another least performing one) — we will drop a total of three now
- **Assignment 5 due tonight**
  - The due date is earlier to allow publication of solutions before the deadline
  - Will publish the non-lab part of the solution tomorrow at 8am
- Check your grades tomorrow after 8am
  - We should have everything except for the final examine at that time
  - Let us know if you see an error there
- **Final exam**
  - 9/6 3p-6p @ **PCH 121**
  - Closed book, no cheatsheet — the same rules as the midterm
  - Pizza party after the examine


Computer
Science &
Engineering
142
Thank you!
CSE142 Summer 2024!
ありがとう
Prof. Usagi