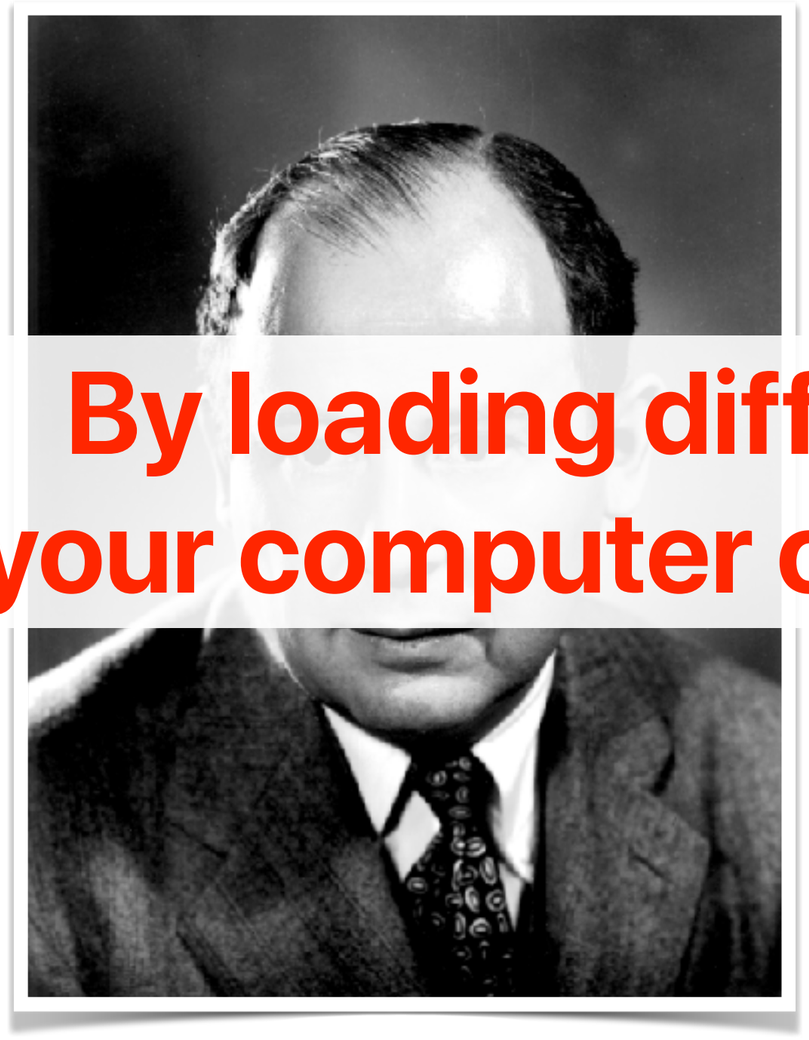
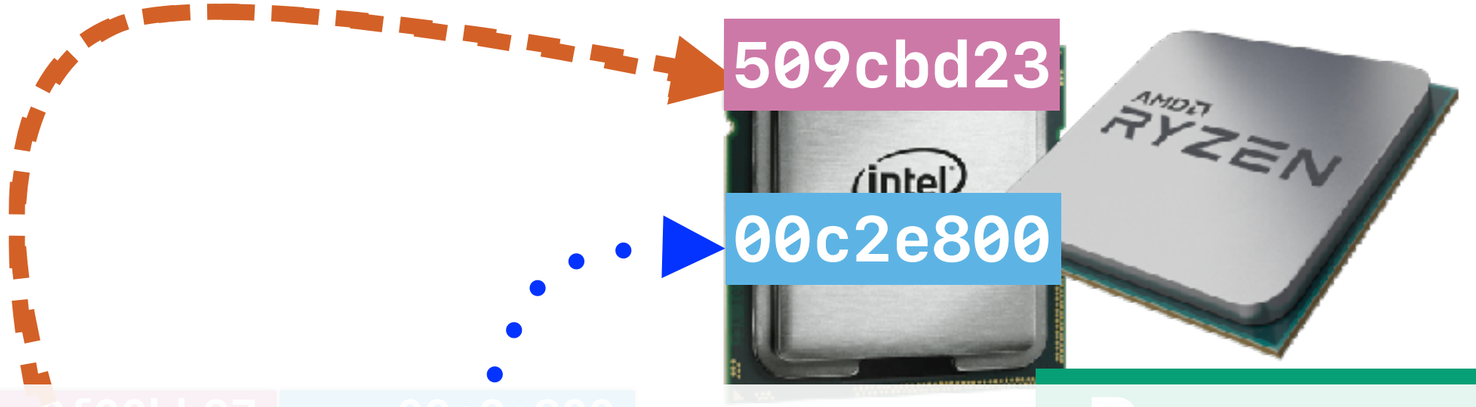
Final Review

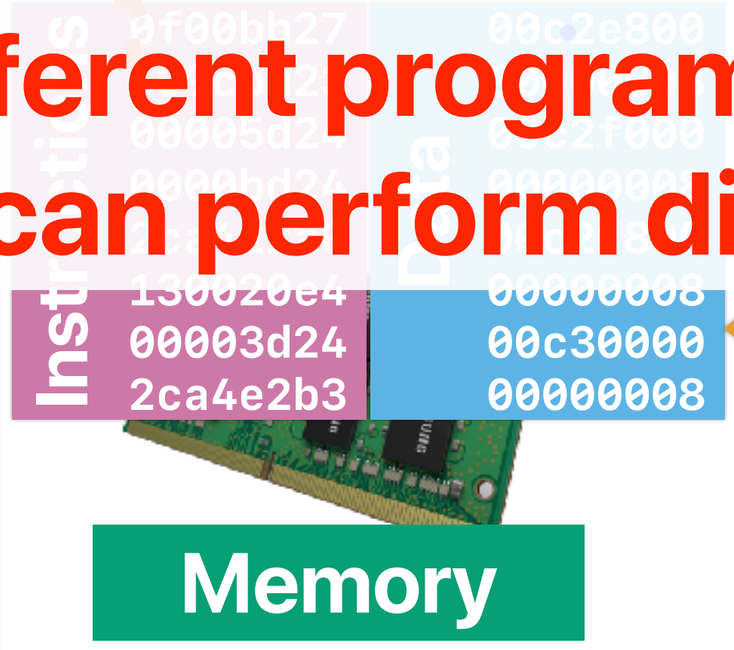
Hung-Wei Tseng

von Neumman Architecture





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

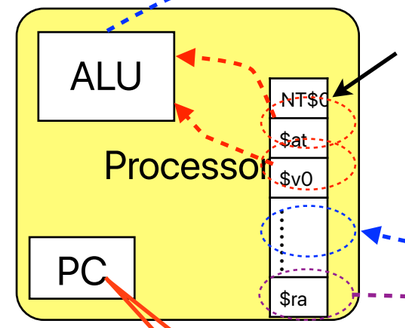


The "life" of a dynamic instruction?

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

How many of these?

- Instruction Fetch (**IF**)
  - Fetch the **instruction** pointed by **PC** 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: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (**MEM**) — Read/write **data 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



How long is it take to execution each of these?

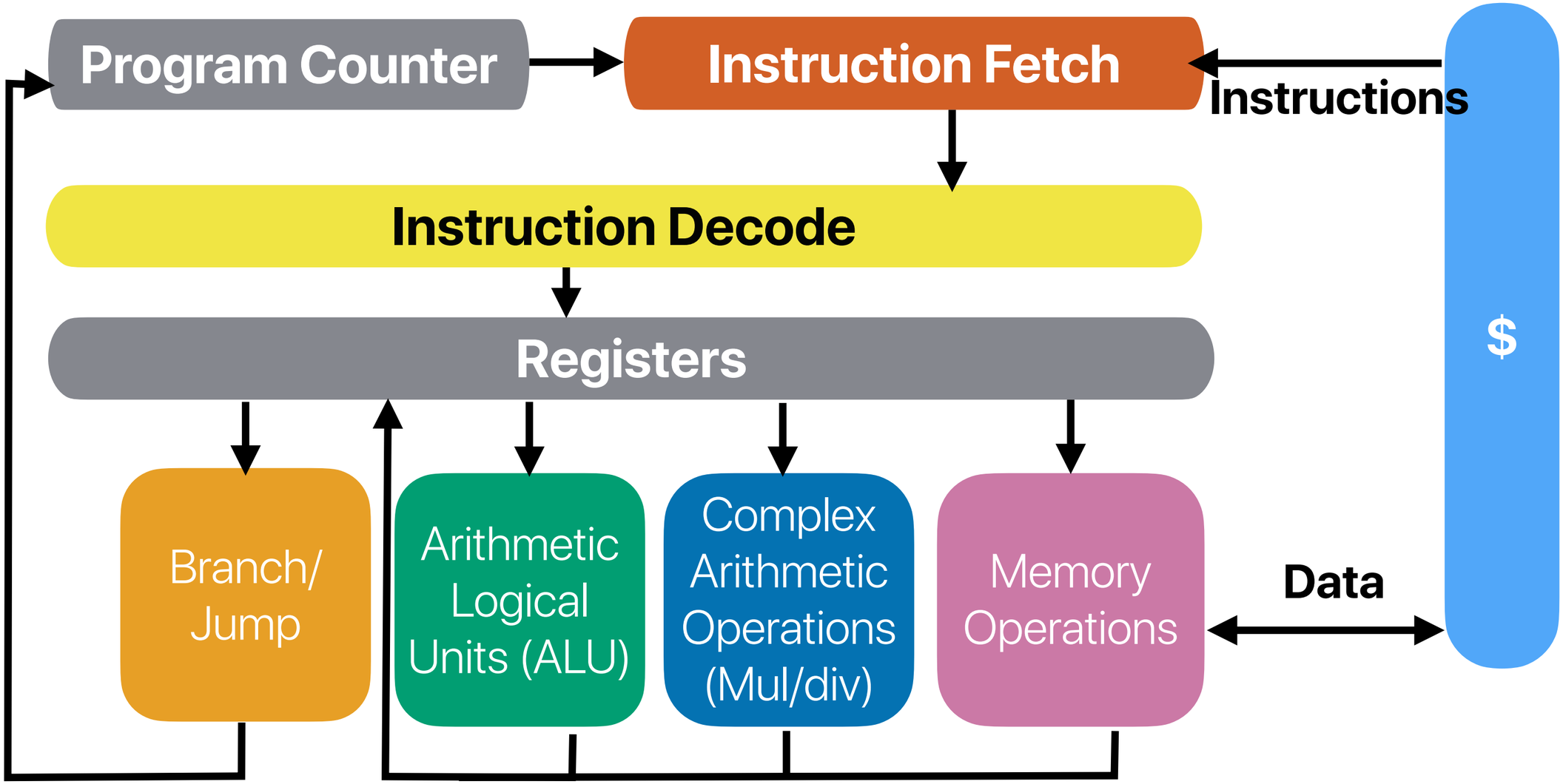
instruction memory

```
120007a30:  0f00bb27  ldah   gp,15(t12)
120007a34:  509cbd23  lda    gp,-25520(gp)
120007a38:  00005d24  ldah   t1,0(gp)
120007a3c:  0000bd24  ldah   t4,0(gp)
120007a40:  2ca422a0  ldl    t0,-23508(t1)
120007a44:  130020e4  beq    t0,120007a94
120007a48:  00003d24  ldah   t0,0(gp)
120007a4c:  2ca4e2b3  stl    zero,-23508(t1)
```

data memory

```
800bf9000:  00c2e800  12773376
800bf9004:  00000008  8
800bf9008:  00c2f000  12775424
800bf900c:  00000008  8
800bf9010:  00c2f800  12777472
800bf9014:  00000008  8
800bf9018:  00c30000  12779520
800bf901c:  00000008  8
```

"Basic" idea of the processor



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

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

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

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

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

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

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

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

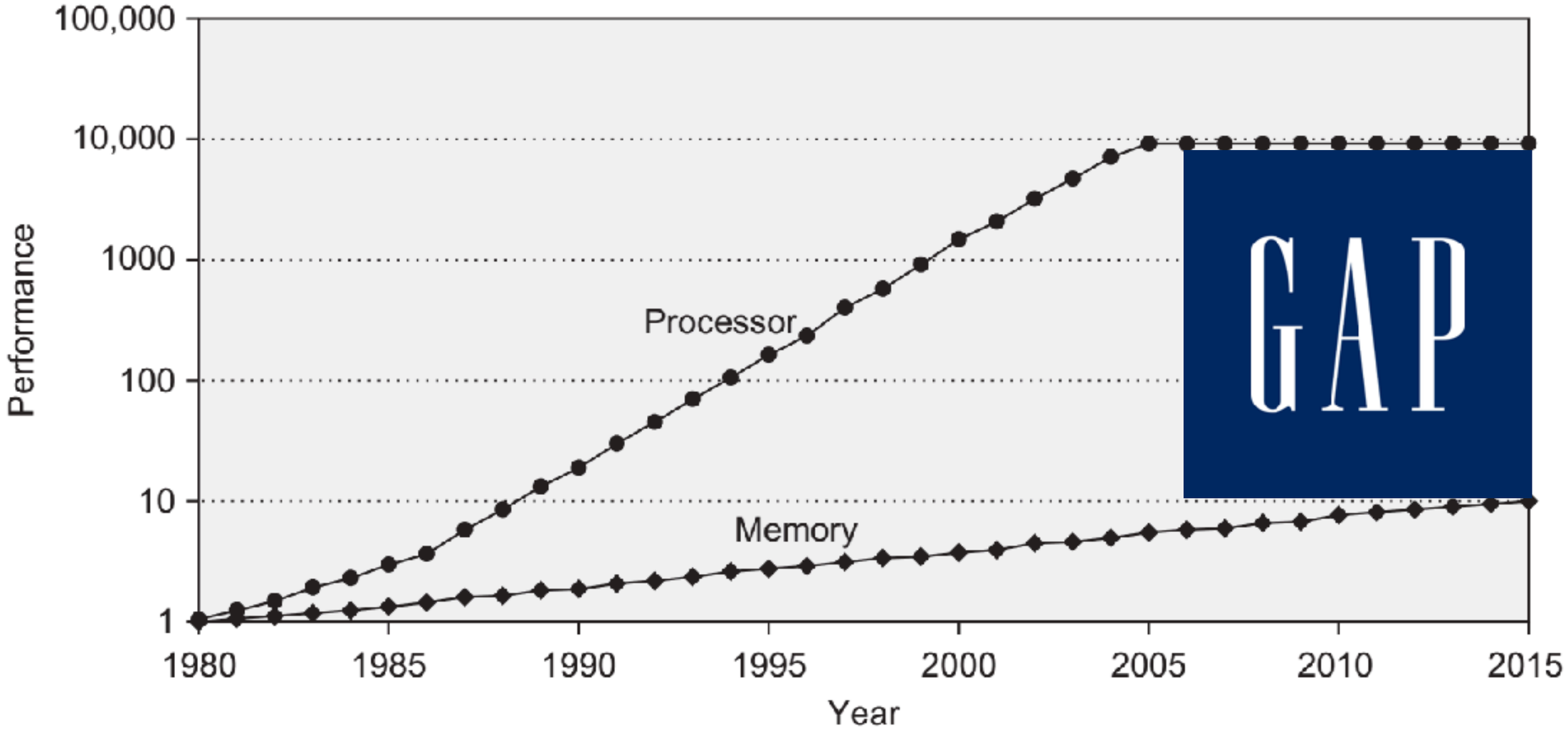
$$Speedup_{enhanced}(f, s, r) = \frac{1}{(1 - f) + perf(r) + \frac{f}{s}}$$

Modern DRAM performance

SDRAM					DDR				
Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year	Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year
100	0.80	3	**24.00**	**1992**	400	3.20	5	**25.00**	**1998**
133	1.07	3	22.50		667	5.33	5	15.00	
					800	6.40	6	15.00	
DDR 2					DDR 3				
400	3.20	5	**25.00**	**2003**	800	6.40	6	**15.00**	**2007**
667	5.33	5	15.00		1066	8.53	8	15.00	
800	6.40	6	15.00		1333	10.67	9	13.50	
					1600	12.80	11	13.75	
					1866	14.93	13	13.93	
					2133	17.07	14	13.13	
DDR 4					DDR 5				
1600	12.80	11	**13.75**	**2014**	3200	25.60	22	**13.75**	**2020**
1866	14.93	13	13.92		3600	28.80	26	14.44	
2133	17.07	15	14.06		4000	32.00	28	14.00	
2400	19.20	17	14.17		4400	35.20	32	14.55	
2666	21.33	19	14.25		4800	38.40	34	14.17	
2933	23.46	21	14.32		5200	41.60	38	14.62	
3200	25.20	22	13.75		5600	44.80	40	14.29	
					6000	48.00	42	14.00	
					6400	51.20	46	14.38	

https://www.anandtech.com/show/16143/insights-into-ddr5-subtimings-and-latencies

Performance gap between Processor/Memory



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 64 words

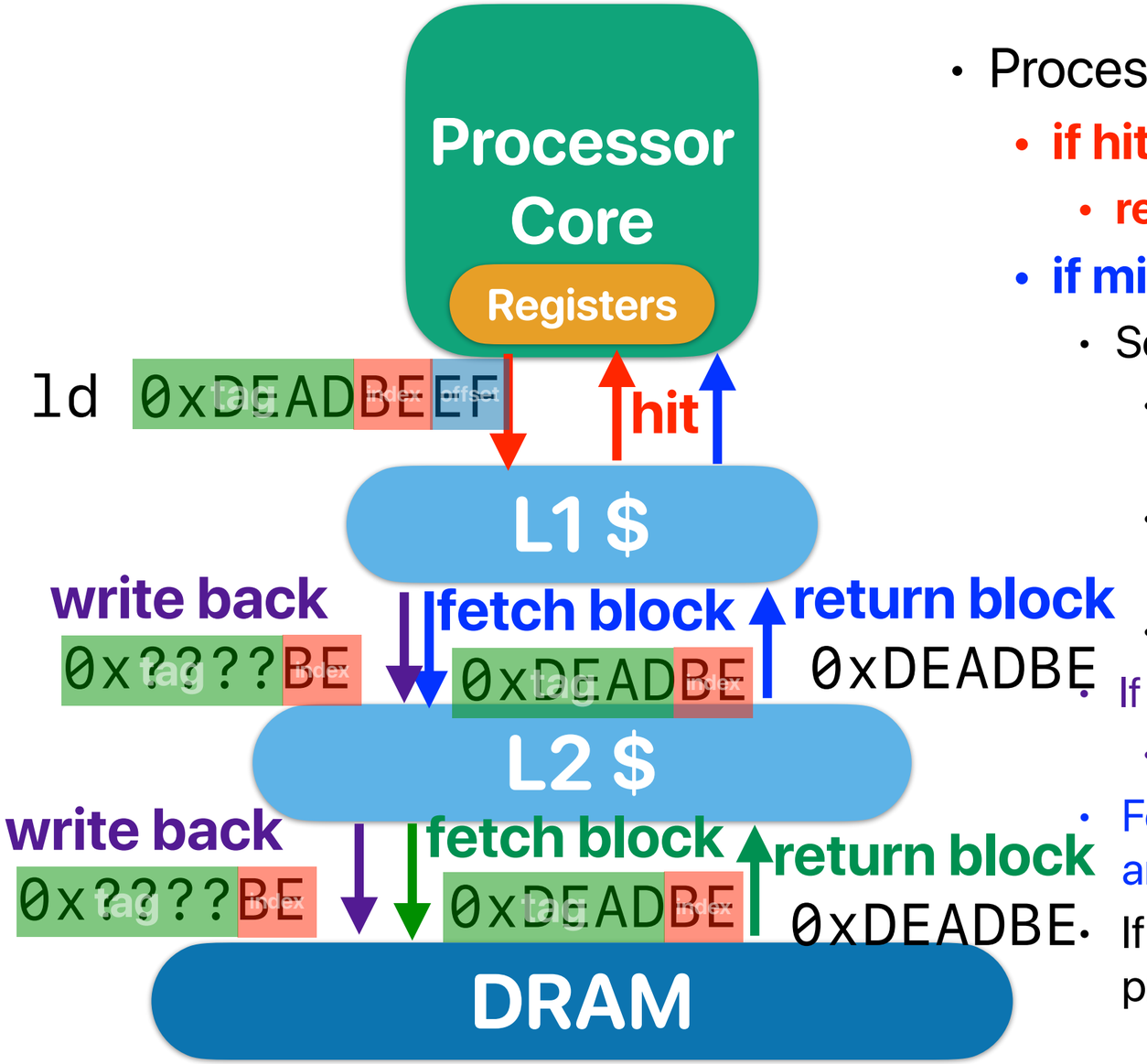
KBs ~ MBs

GBs

TBs

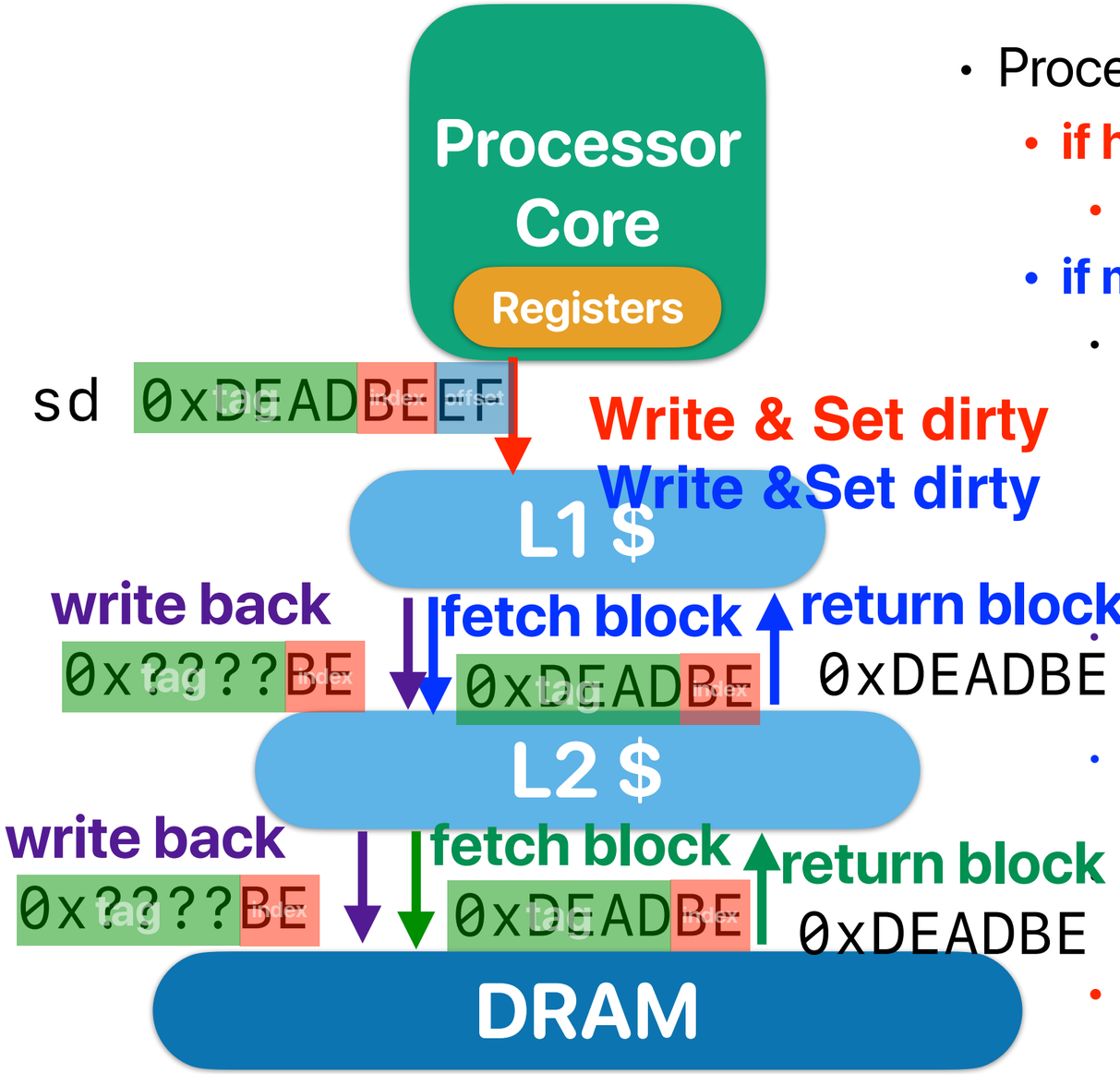
larger

What happens when we read data



- Processor sends load request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process

What happens when we write data



- Processor sends load request to L1-$
  - **if hit**
    - **return data — set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
  - **Present the write "ONLY" in L1 and set DIRTY**

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

NVIDIA Tegra X1

100% miss rate!

- 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[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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

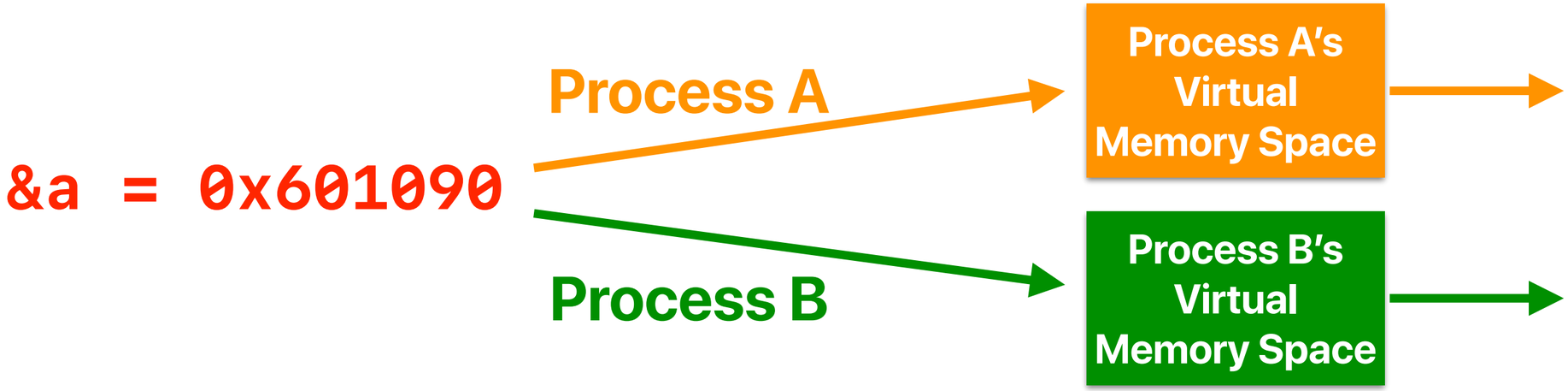
Summary of Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
  - Matrix transpose
  - Column-store vs. row-store
- Blocking/tiling — capacity miss, conflict miss
  - Matrix multiplications
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Loop interchange — conflict/capacity miss

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

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

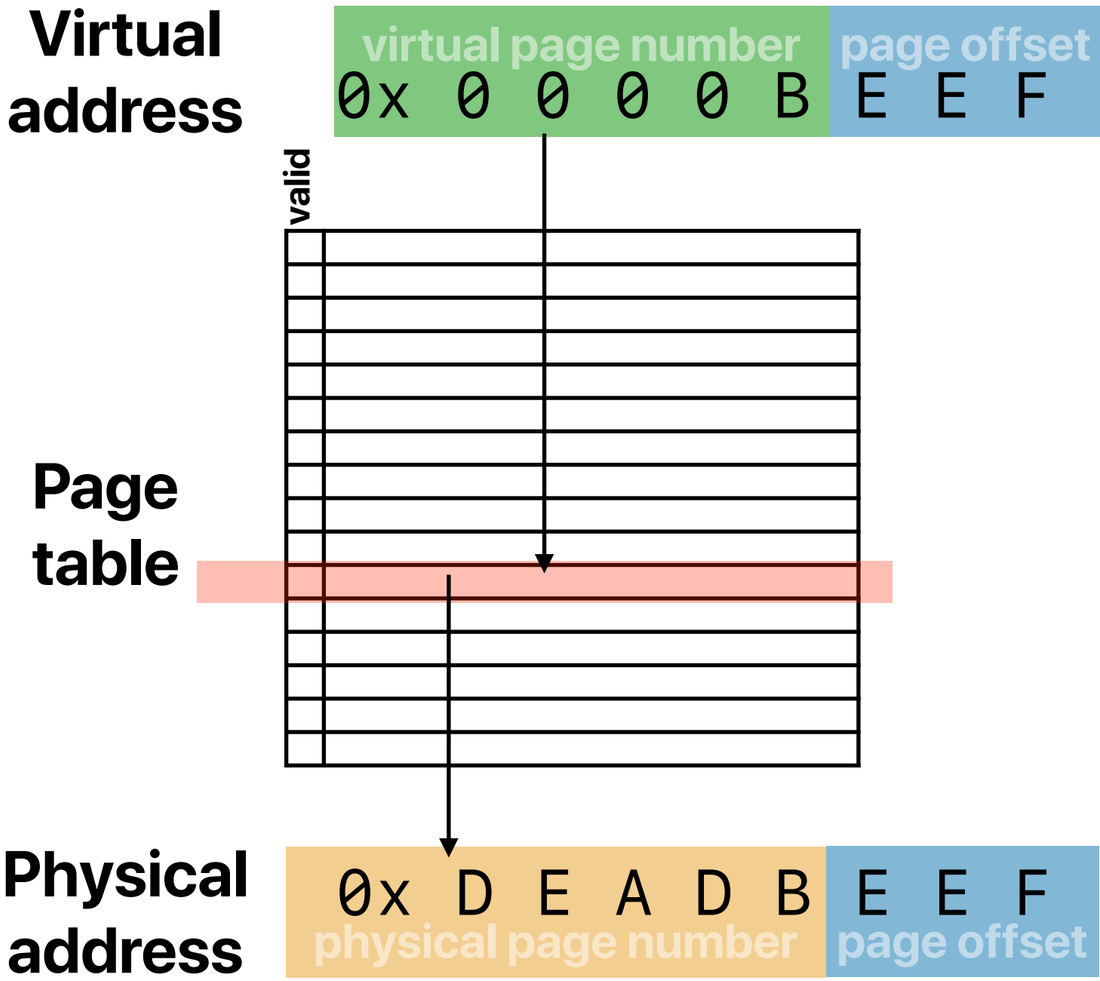
Physical Memory

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



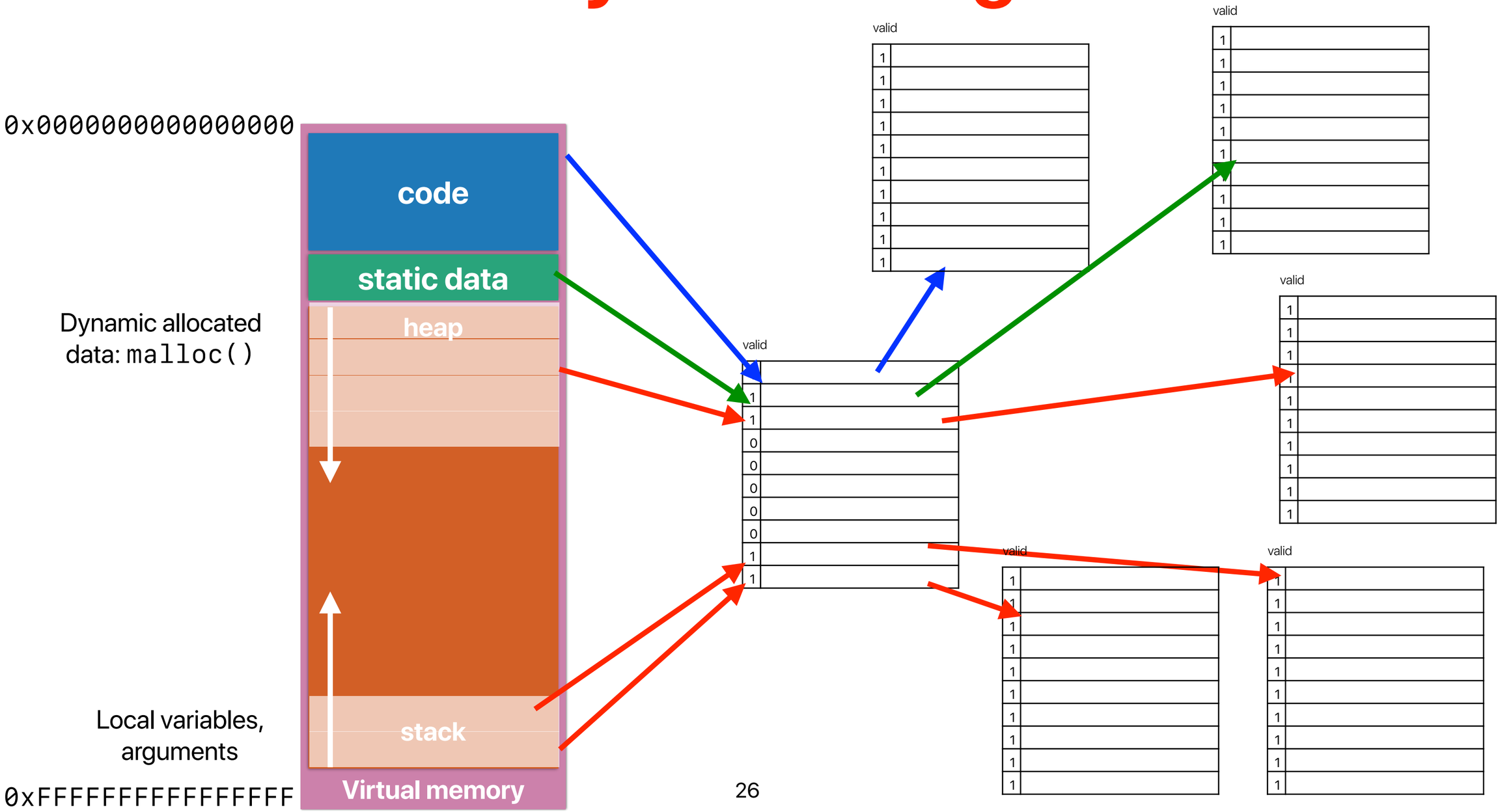
Size of page table

- 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

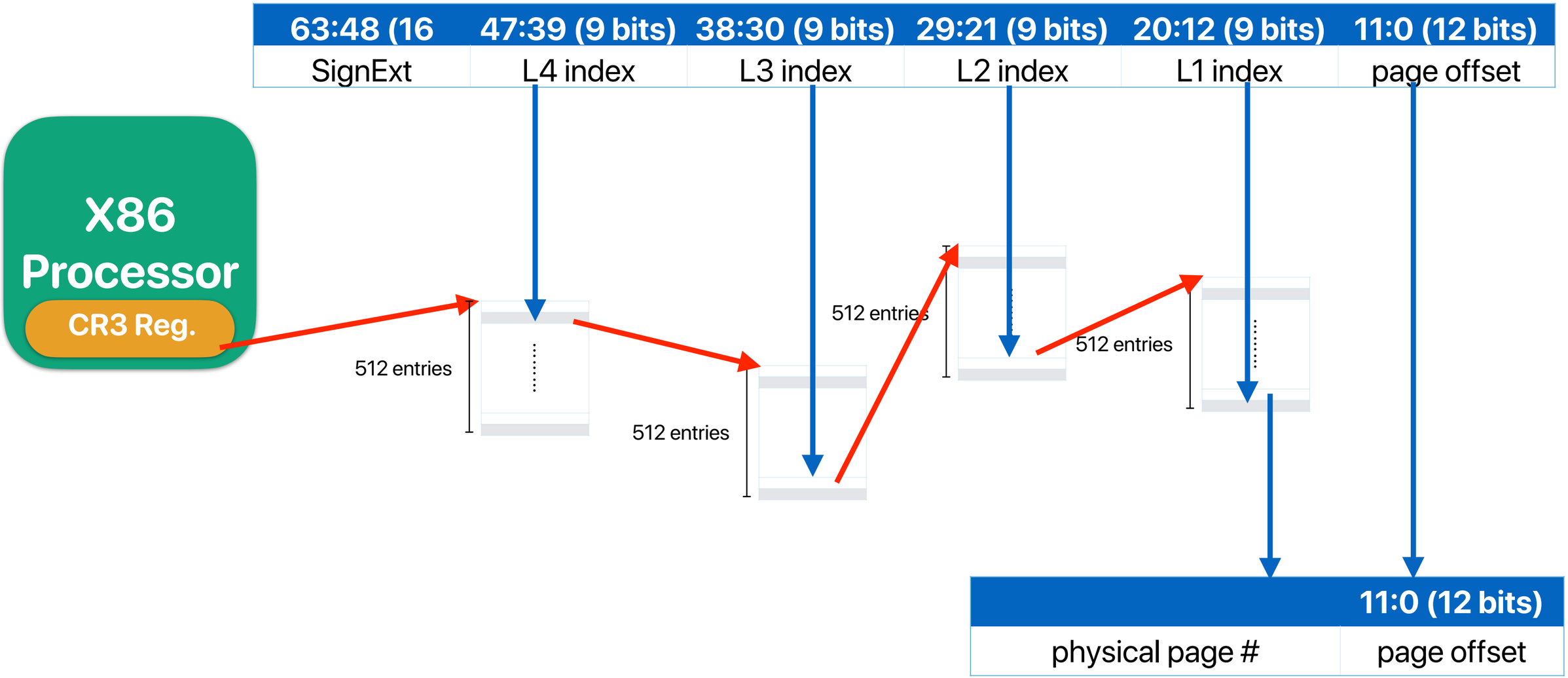
$$\frac{2^{64}\ Bytes}{4\ KB} \times 8\ Bytes = 2^{55}\ Bytes = 32\ PB$$

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

Do we really need a large table?

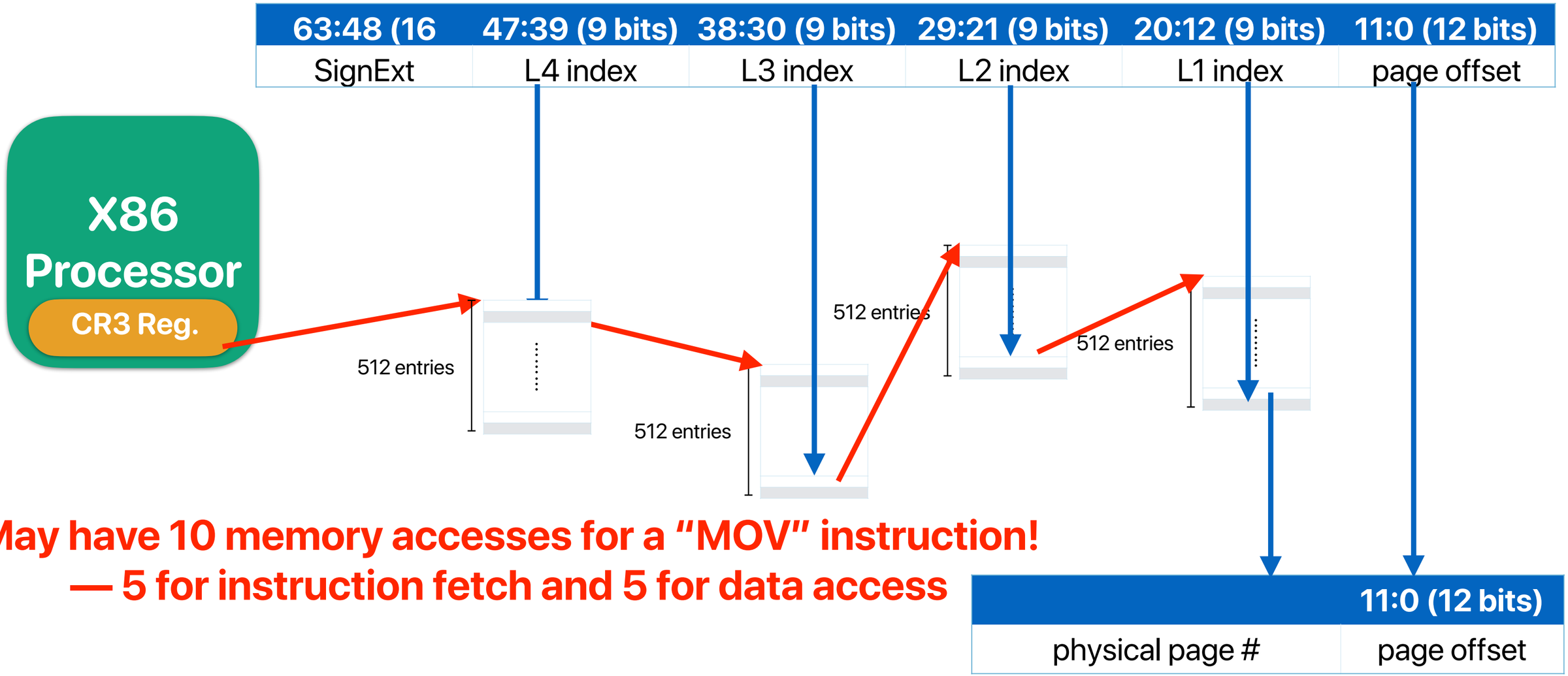


Address translation in x86-64



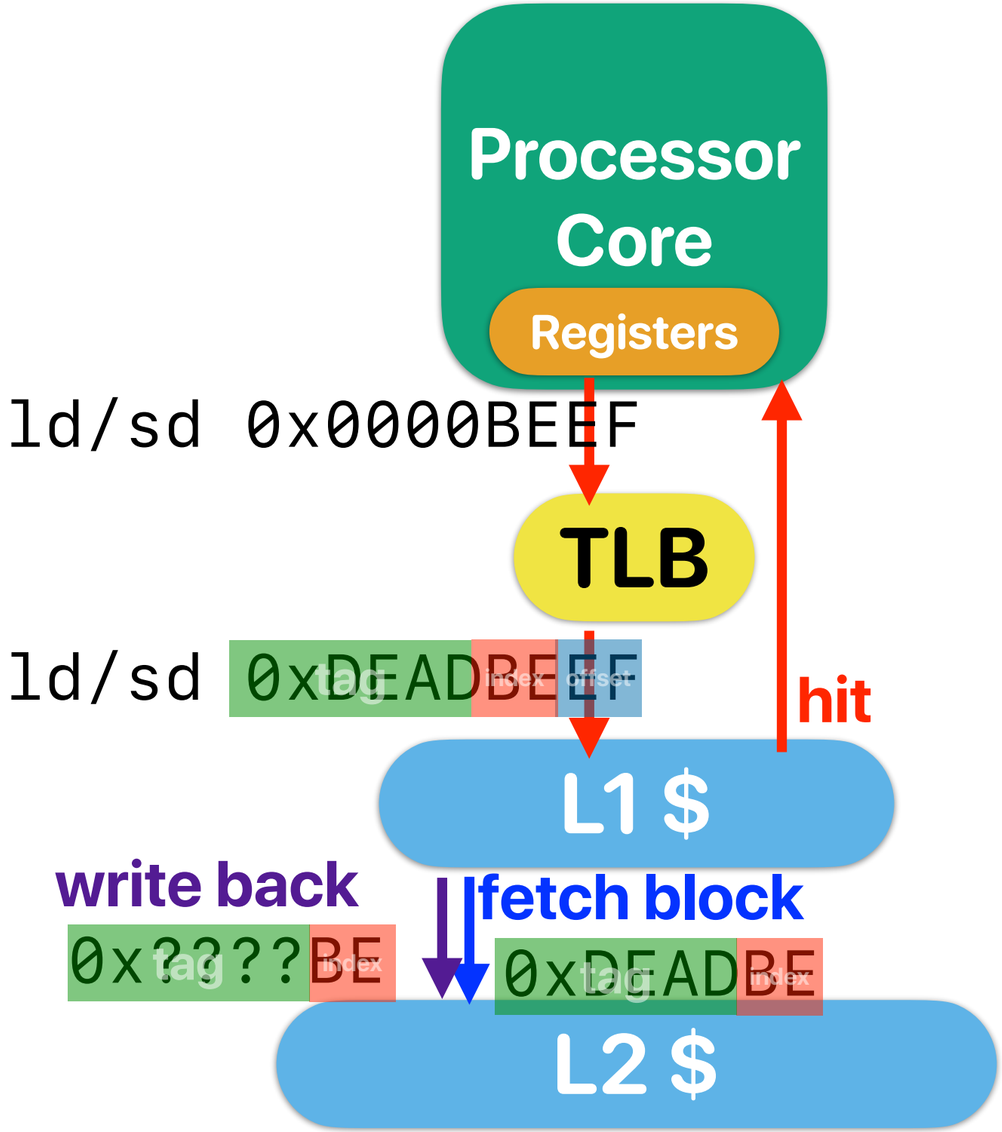
Translation Caching: Skip, Don't Walk (the Page Table)
Thomas W. Barr, Alan L. Cox, Scott Rixner

Address translation in x86-64



May have 10 memory accesses for a "MOV" instruction!
— 5 for instruction fetch and 5 for data access

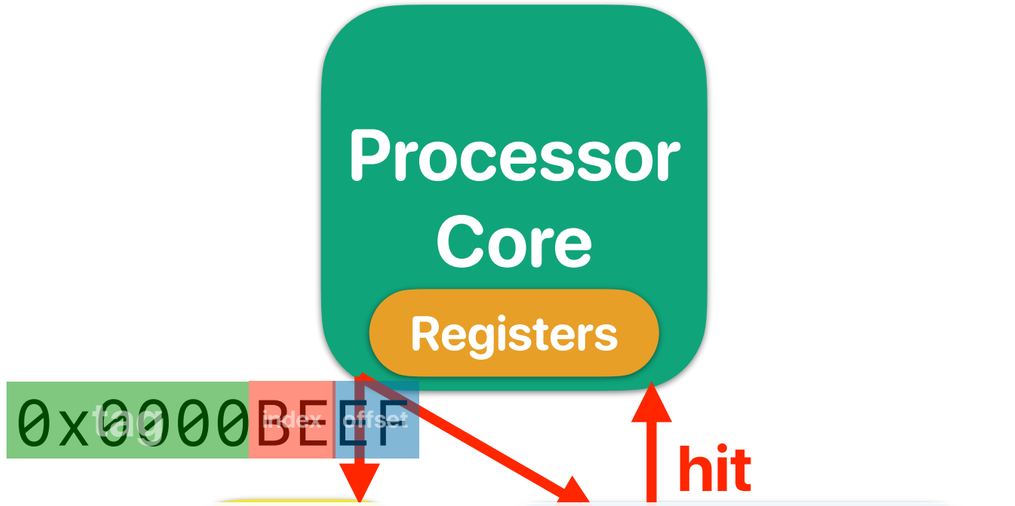
TLB: Translation Look-aside Buffer



- 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

TLB + Virtual cache

- L1 $ accepts virtual address — you don't need to translate
- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
- Bad — it doesn't work in practice
  - Many applications have the same virtual address but should be pointing different **physical addresses**
  - An application can have "aliasing virtual addresses" pointing to the same **physical address**



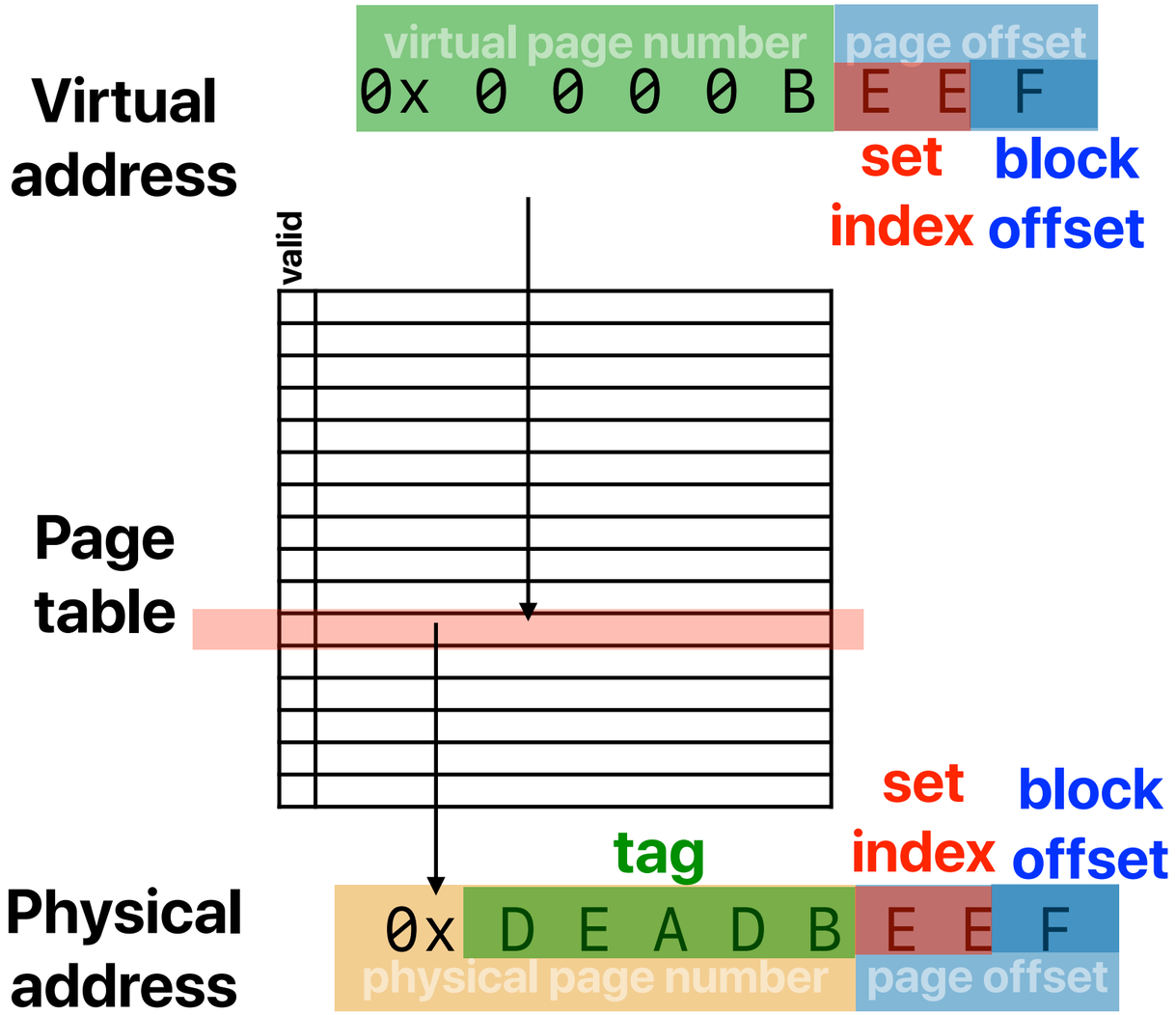
You really need "physical address" to judge if that's what you want

Virtually indexed, physically tagged cache

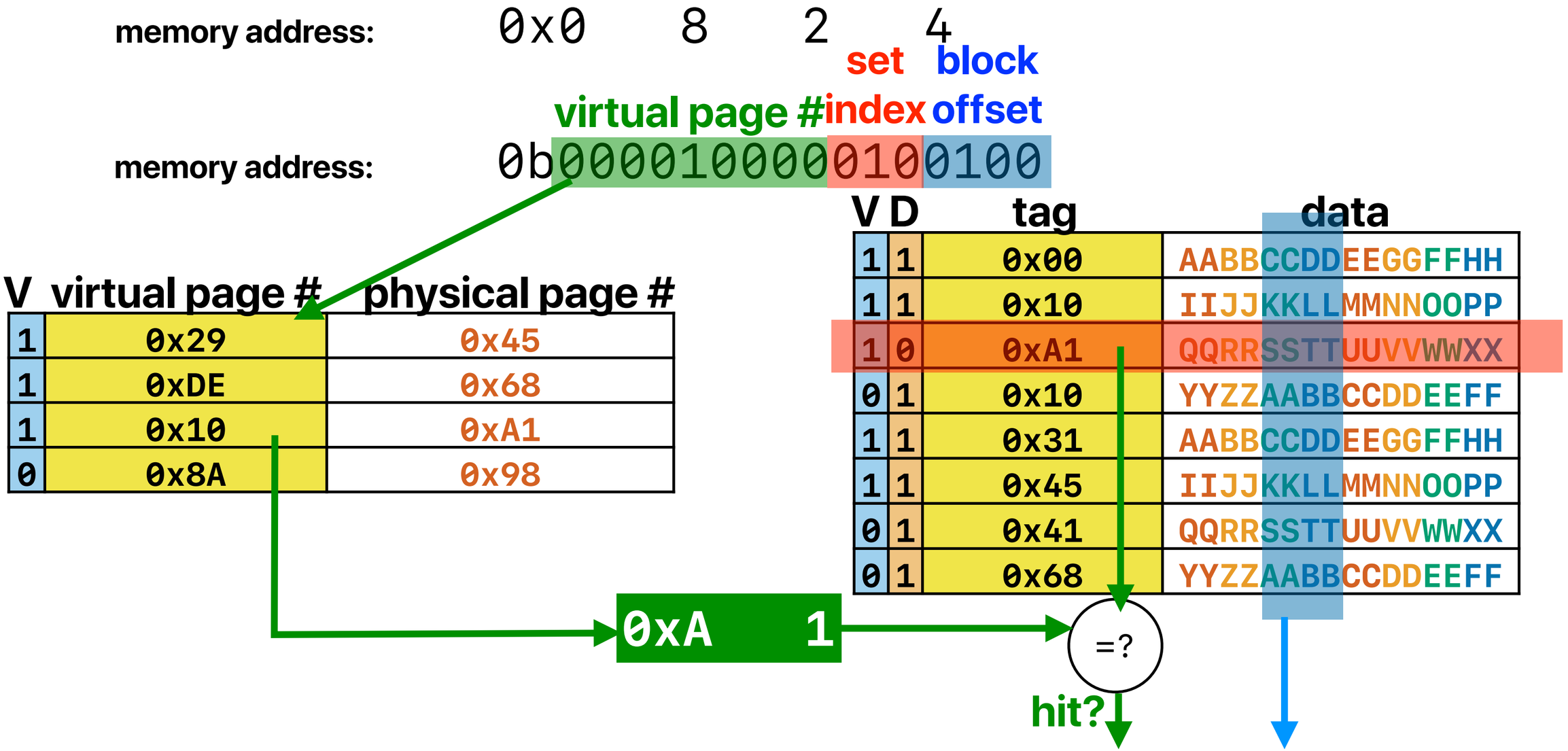
- Can we find physical address directly in the virtual address

  — 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 —

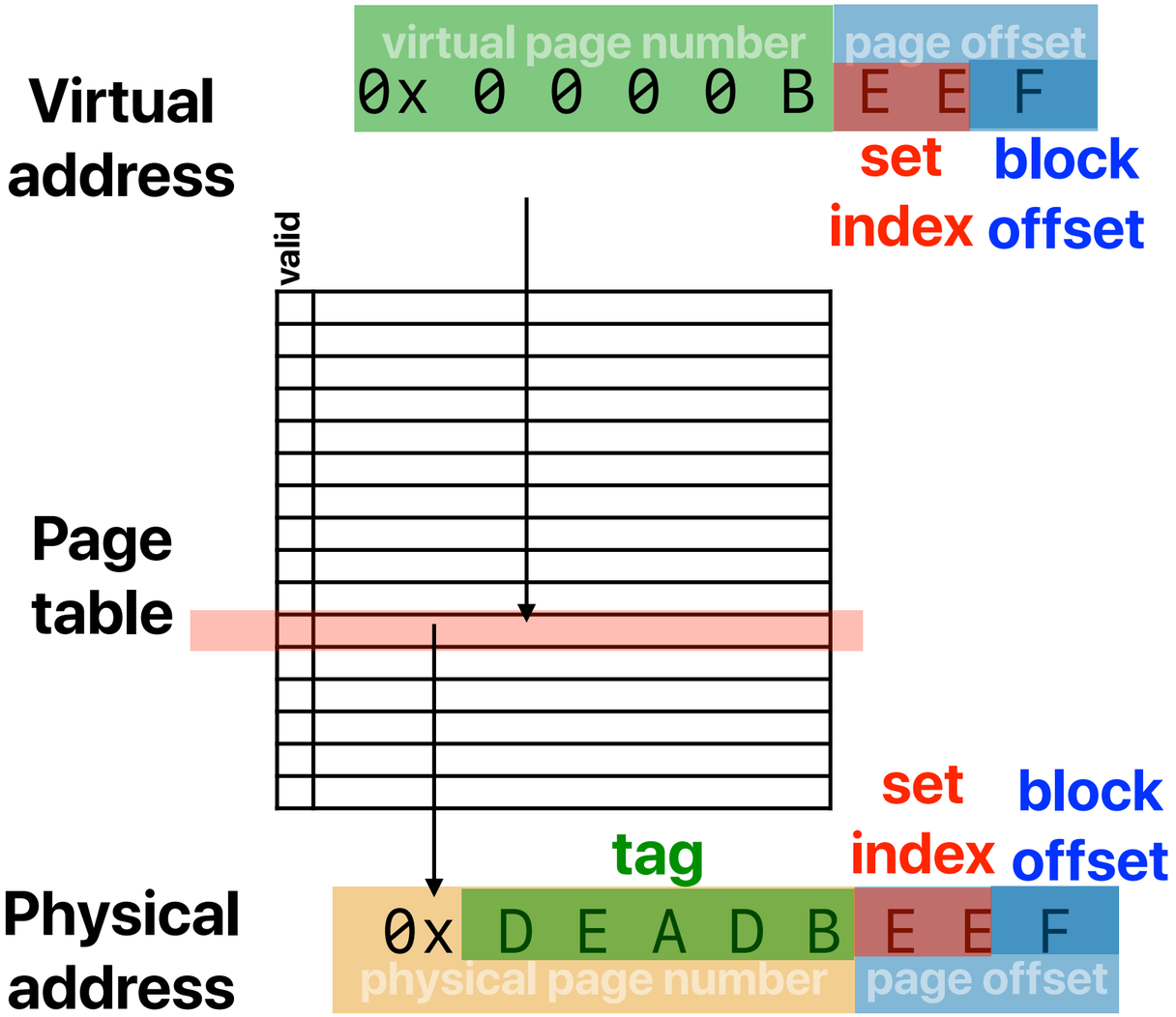
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

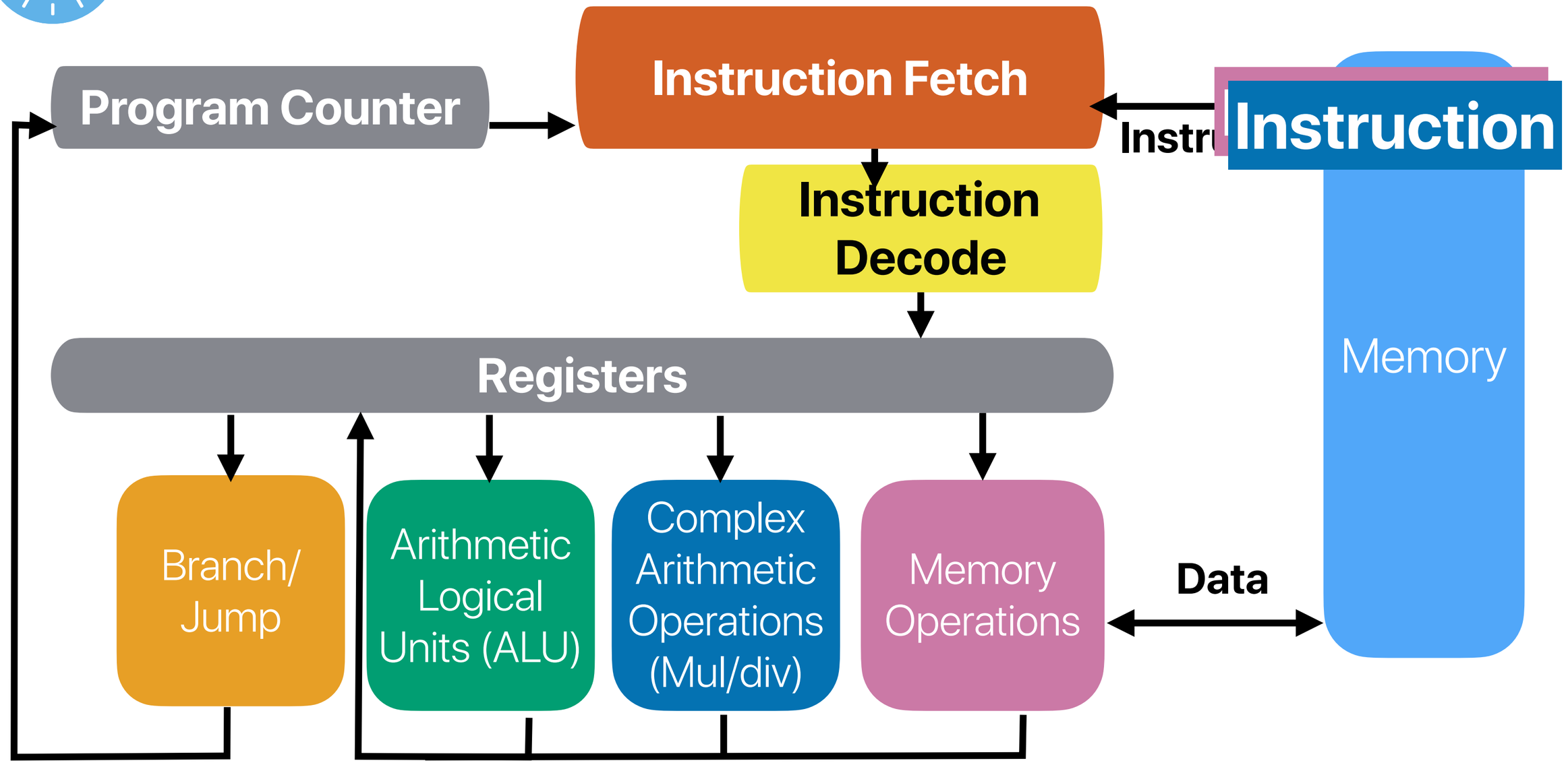
$$C = A \times 2^{12}$$

$$if\ A = 1$$

$$C = 4KB$$



Within a cycle…

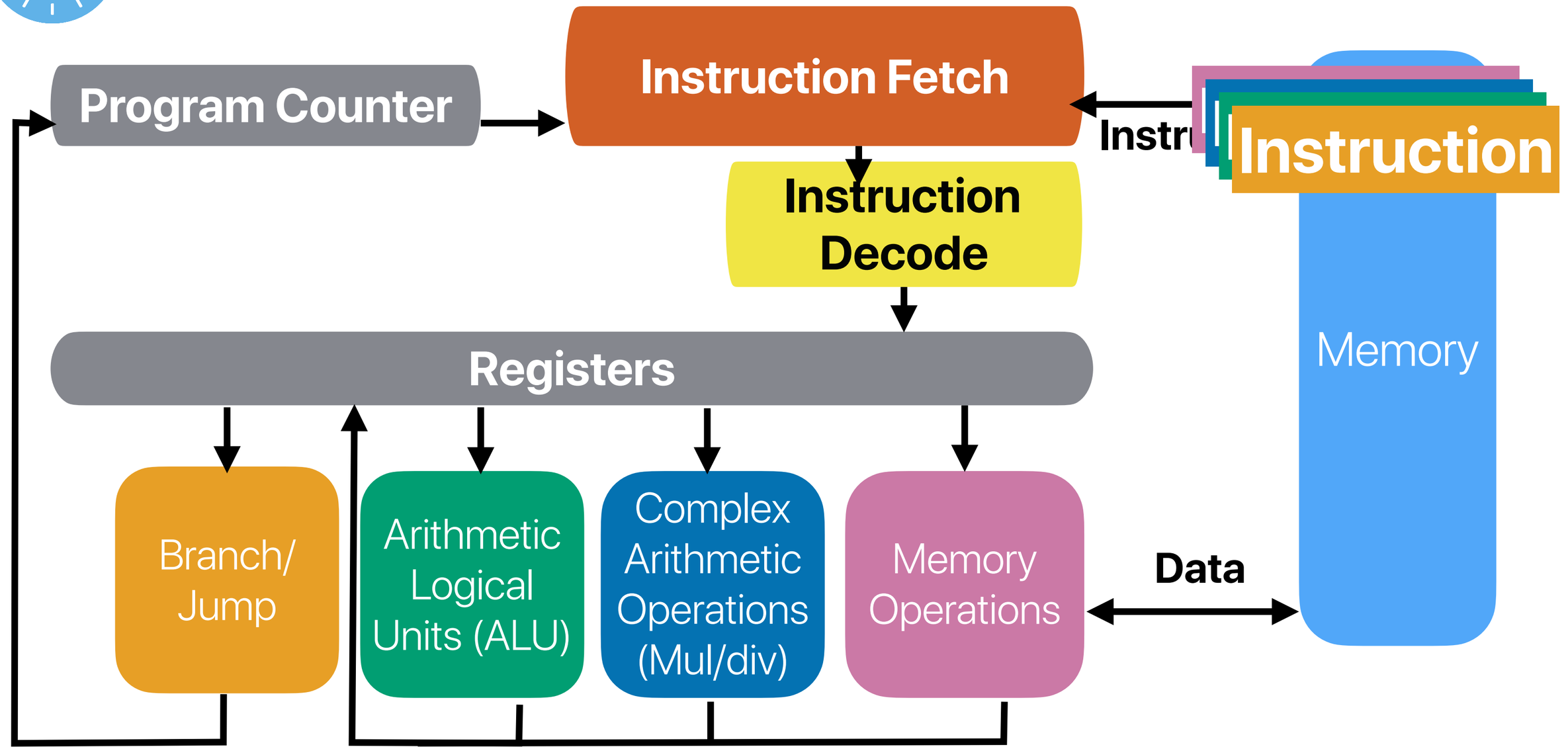


One instruction at time in a processor...

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

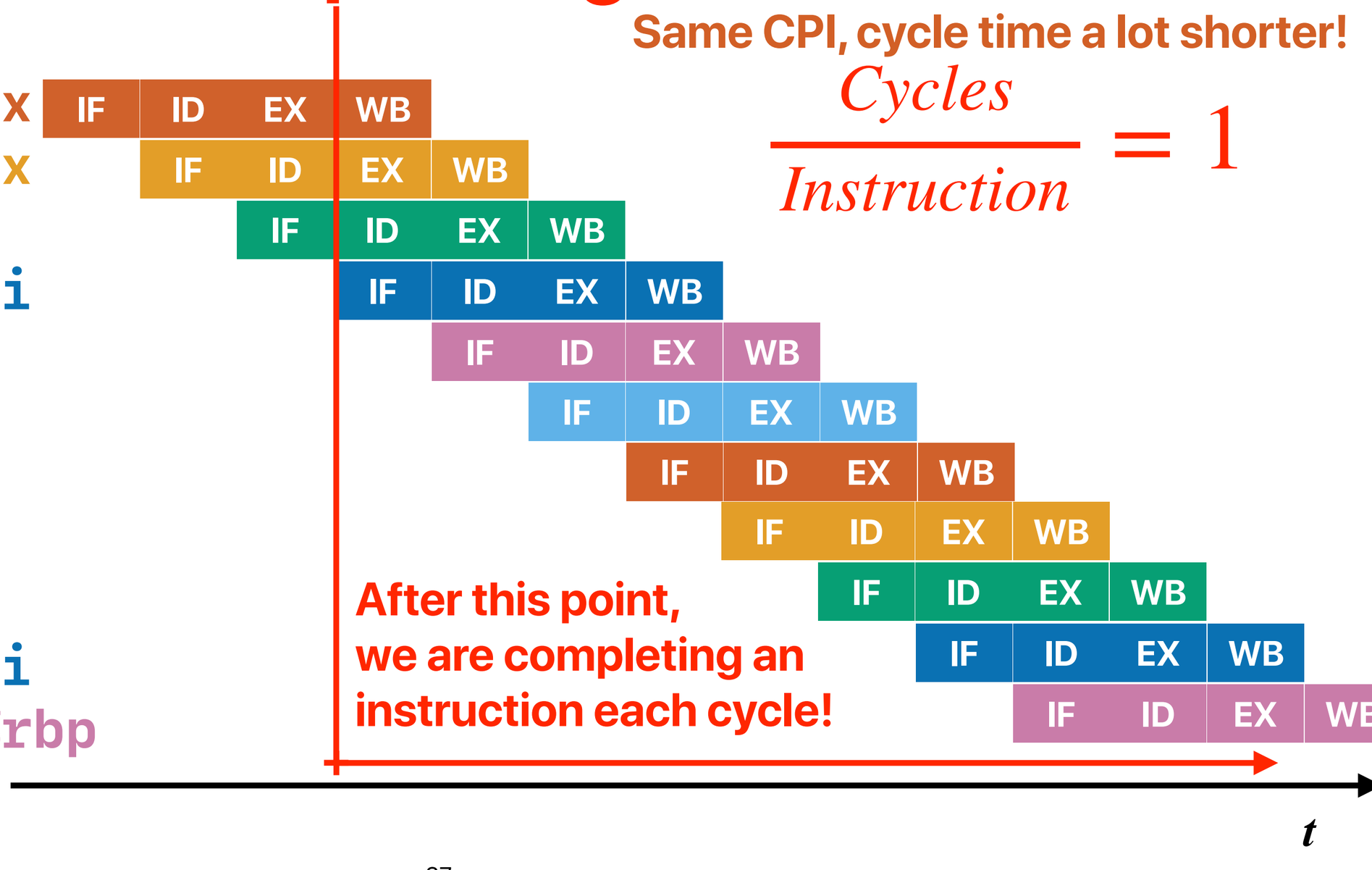
Cycle time is long, lots of functional units sitting idle!

"Pipeline" the processor!



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
```



Same CPI, cycle time a lot shorter!

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

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

Three pipeline hazards

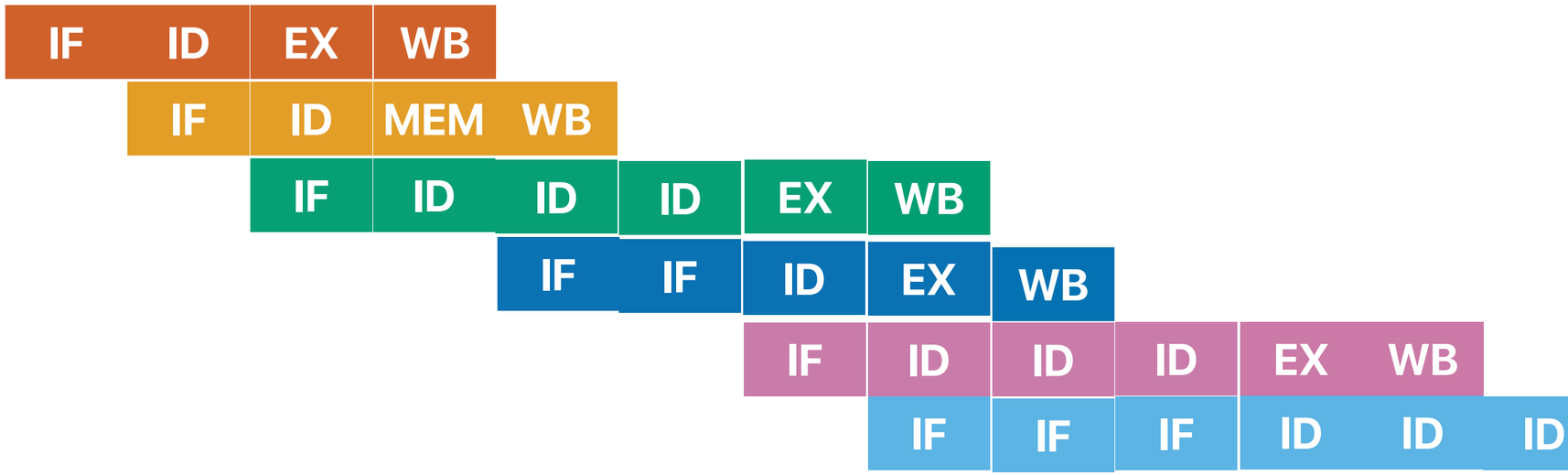
- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that's not yet generated or propagated when the instruction needs that

Stall — the universal solution to pipeline hazards

Stall whenever we have a hazard

- Stall: the hardware allows the earlier instruction to proceed, all later instructions stay at the same stage

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



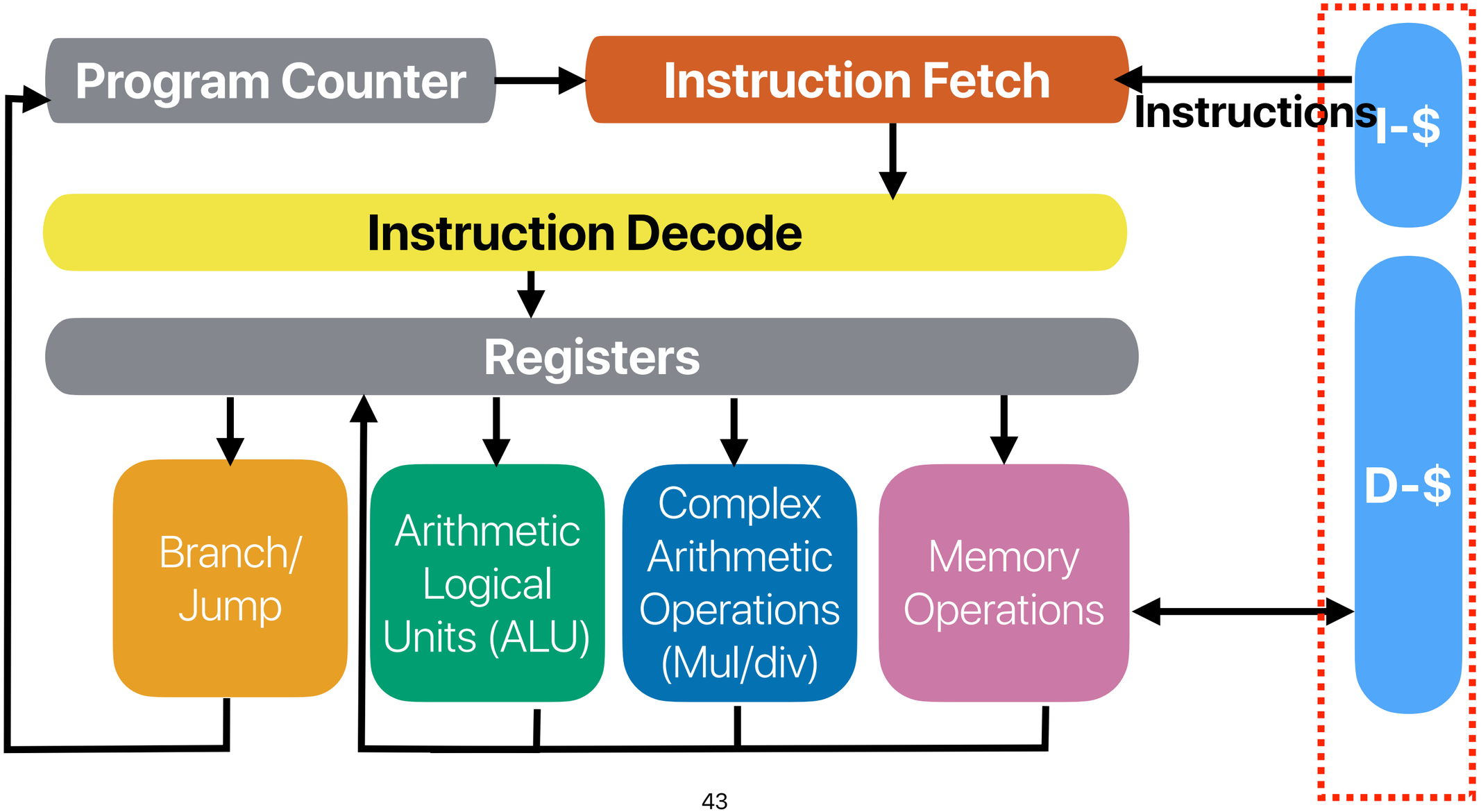
Slow! — 5 additional cycles

Structural Hazards

Solutions/work-around of pipeline hazards

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

Split L1-$


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

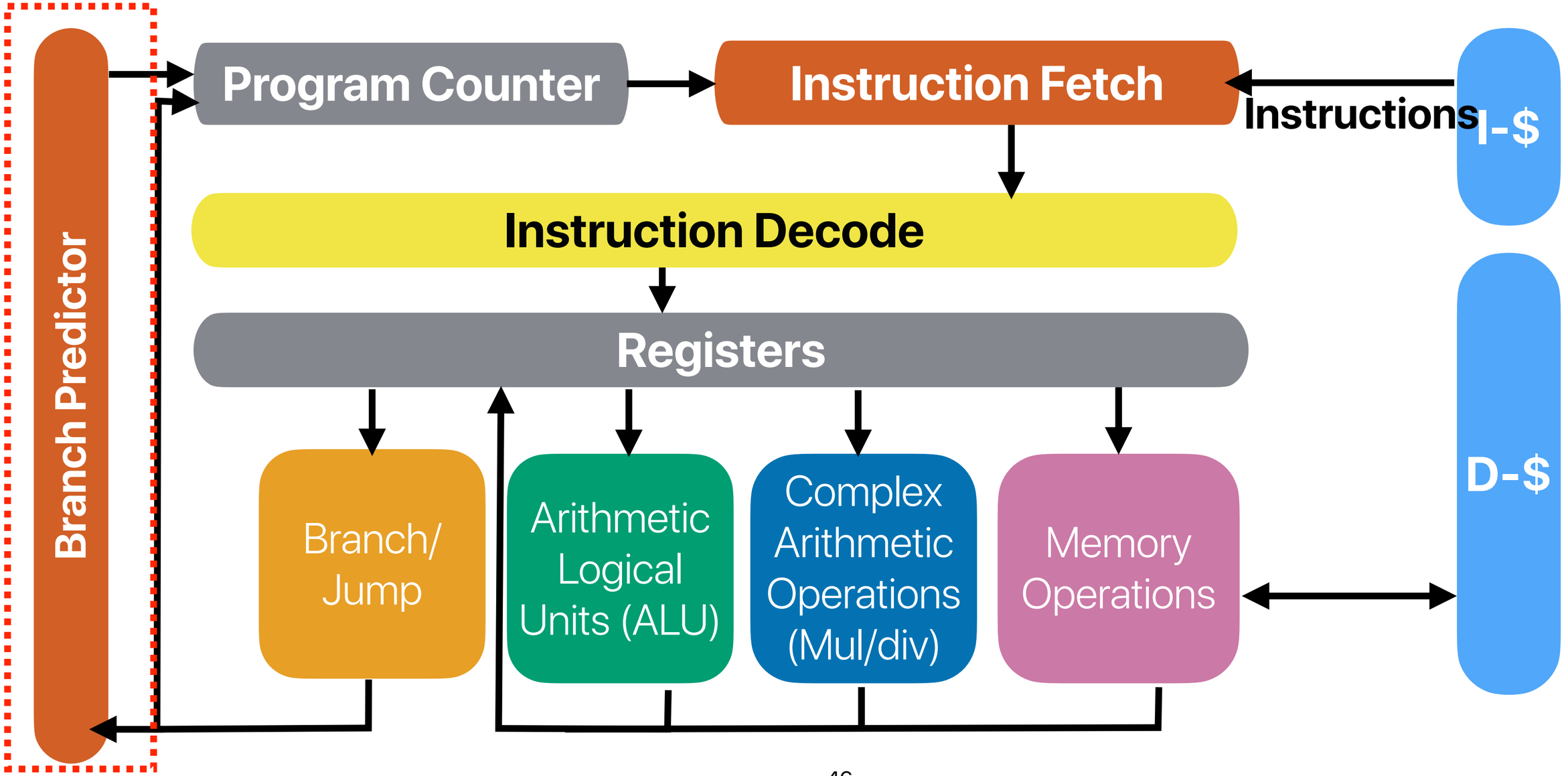
Control Hazards

The frequency of branch

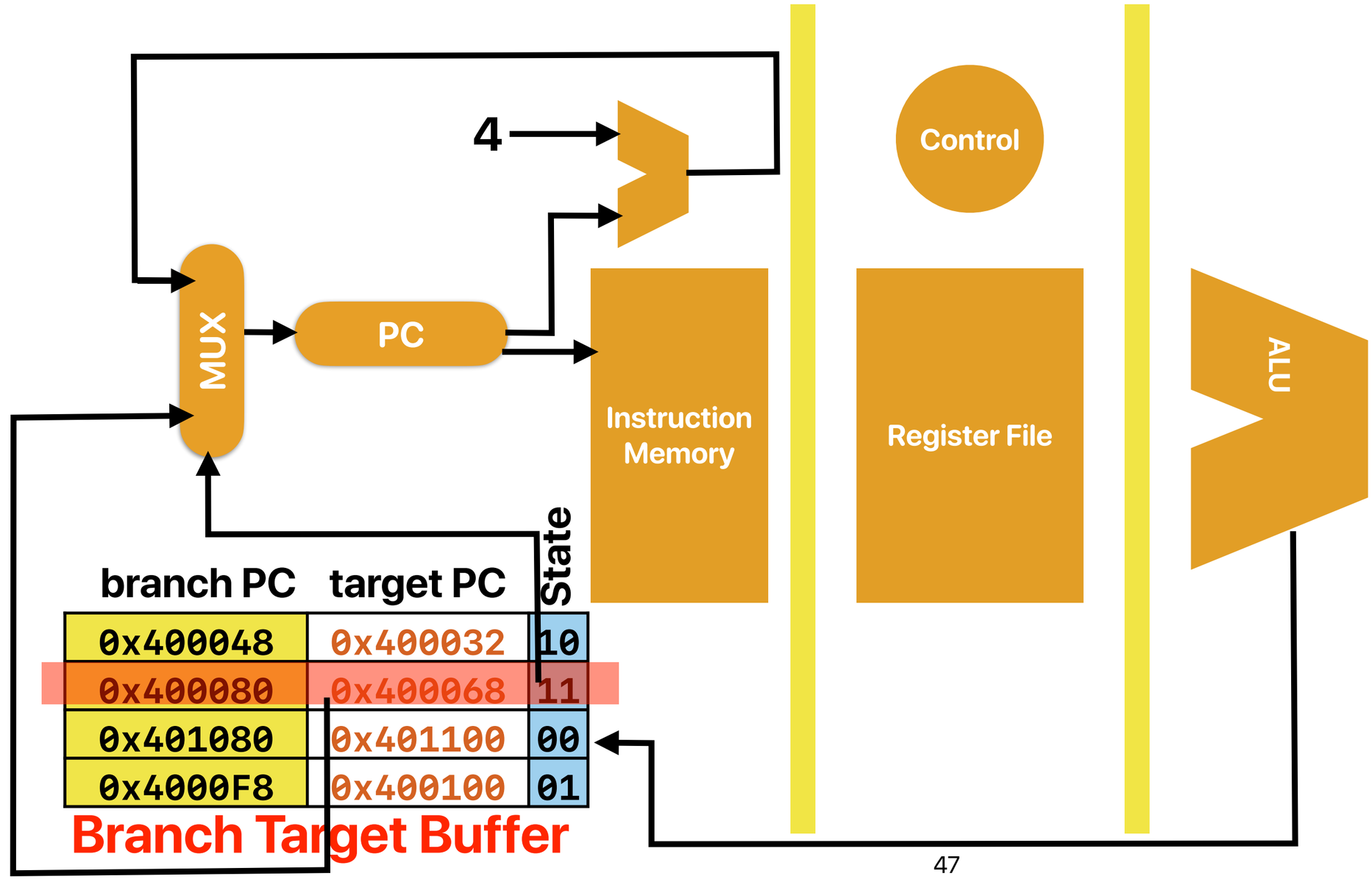
Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%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

Microprocessor with a “branch predictor”



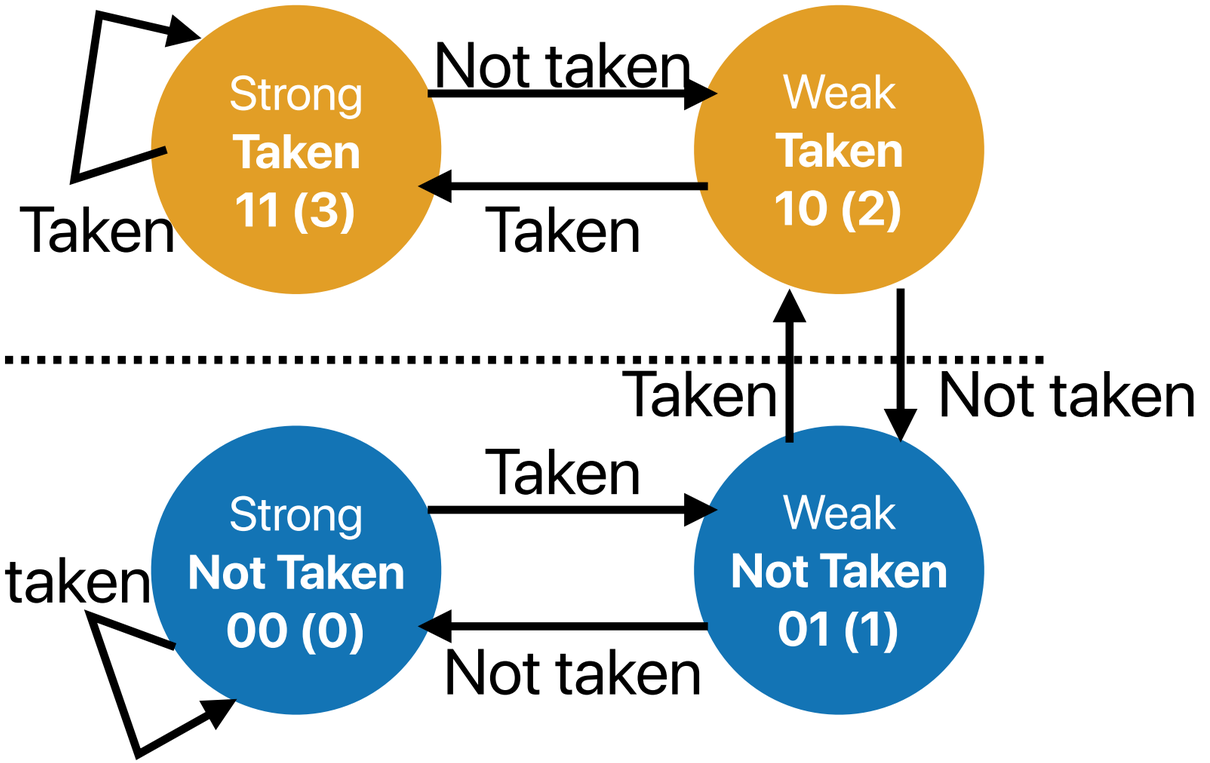
A basic dynamic branch predictor



2-bit/Bimodal local predictor

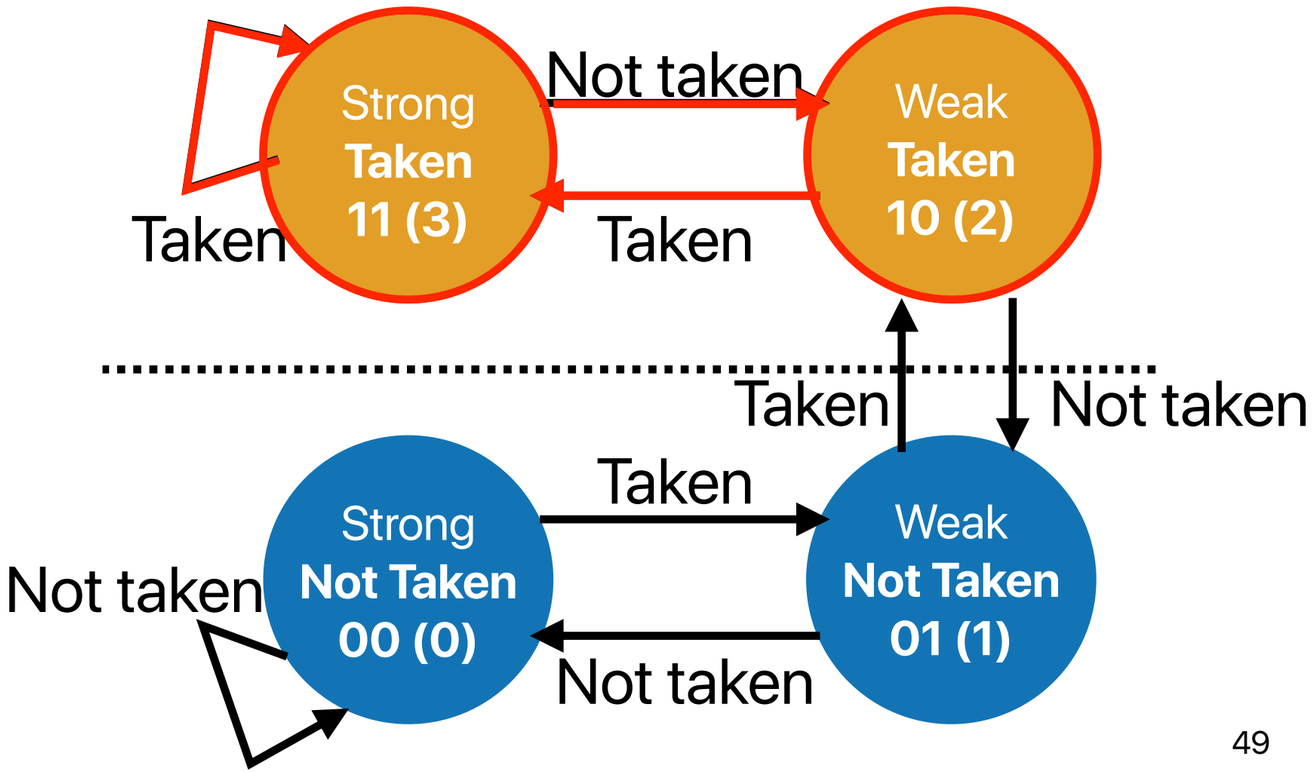
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF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

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



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%

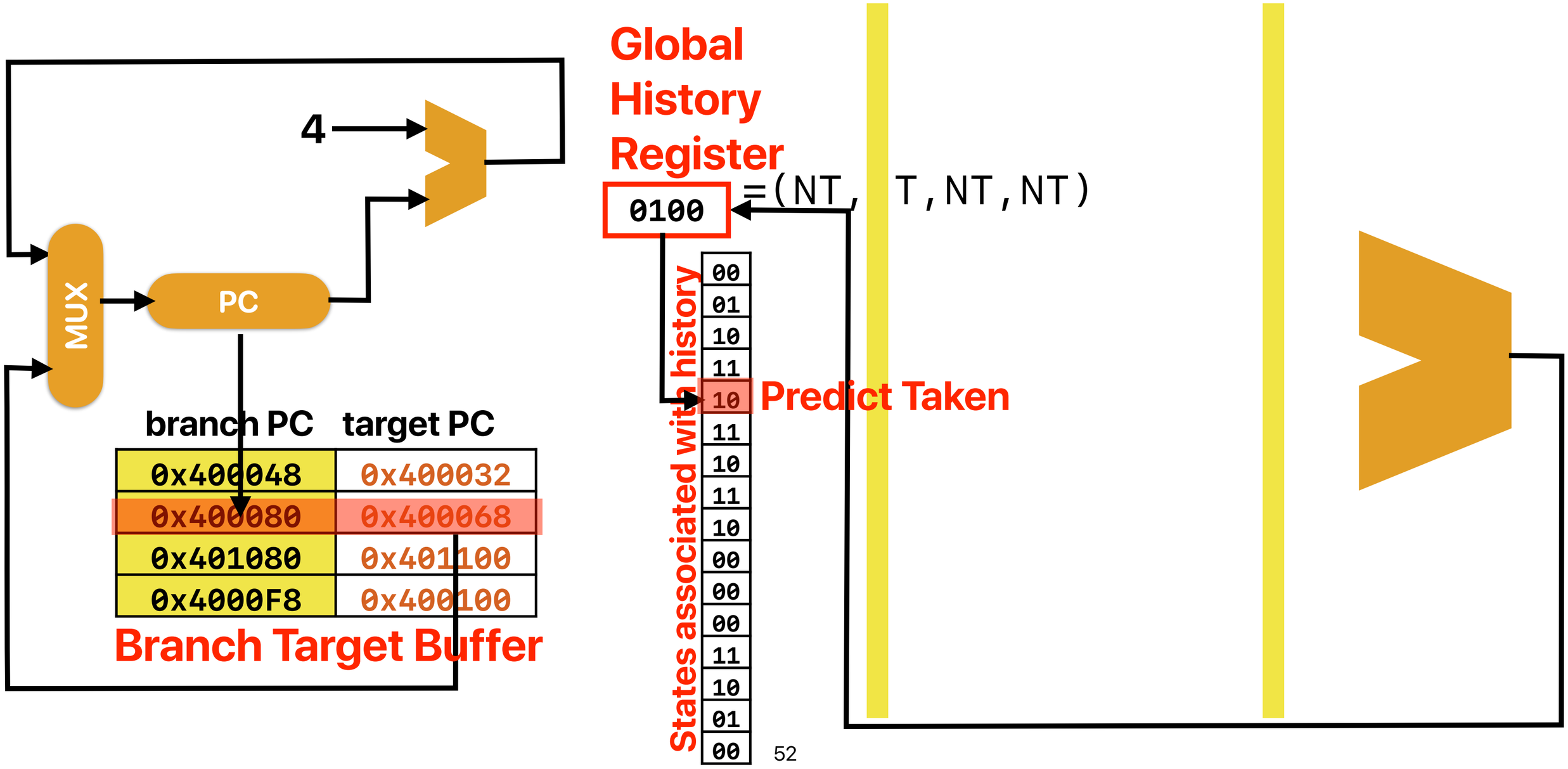
For branch Y, almost 100%,
For branch X, only 50%

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

Two-level global predictor

Reading: Scott McFarling. Combining Branch Predictors. Technical report WRL-TN-36, 1993.

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

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

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%

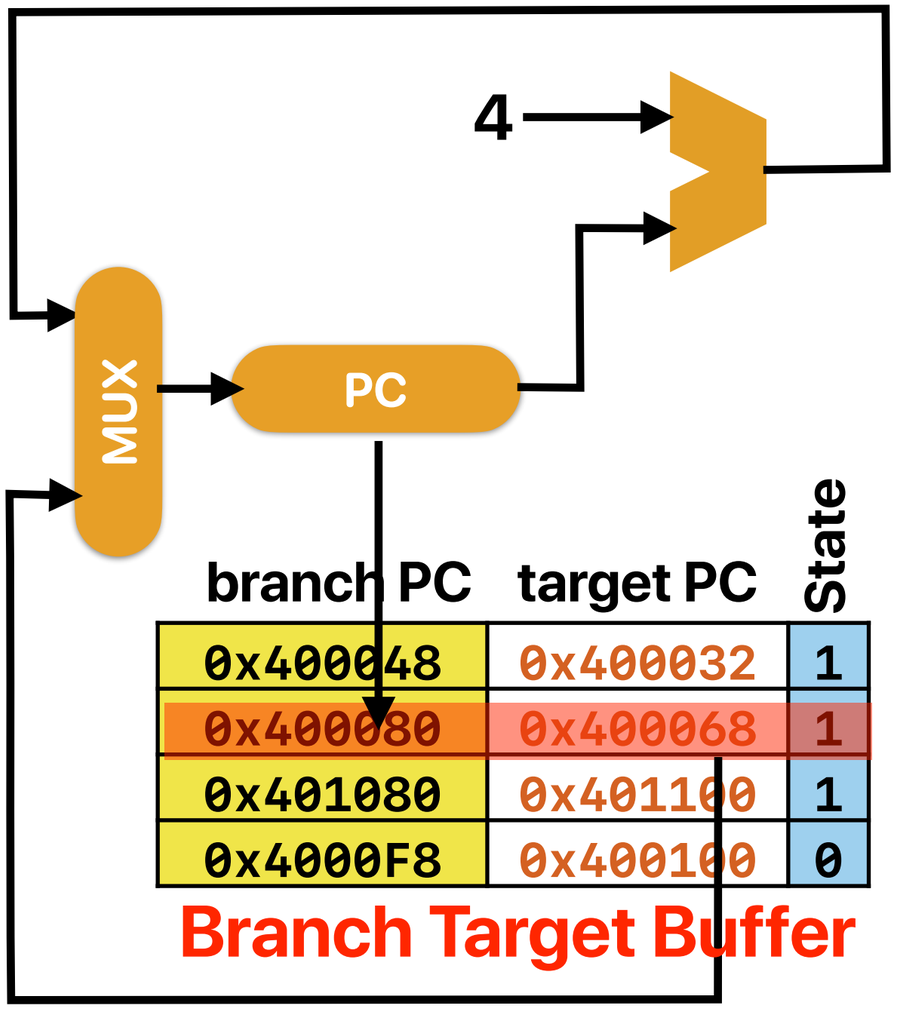
The **global** history

	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

Hybrid predictors

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

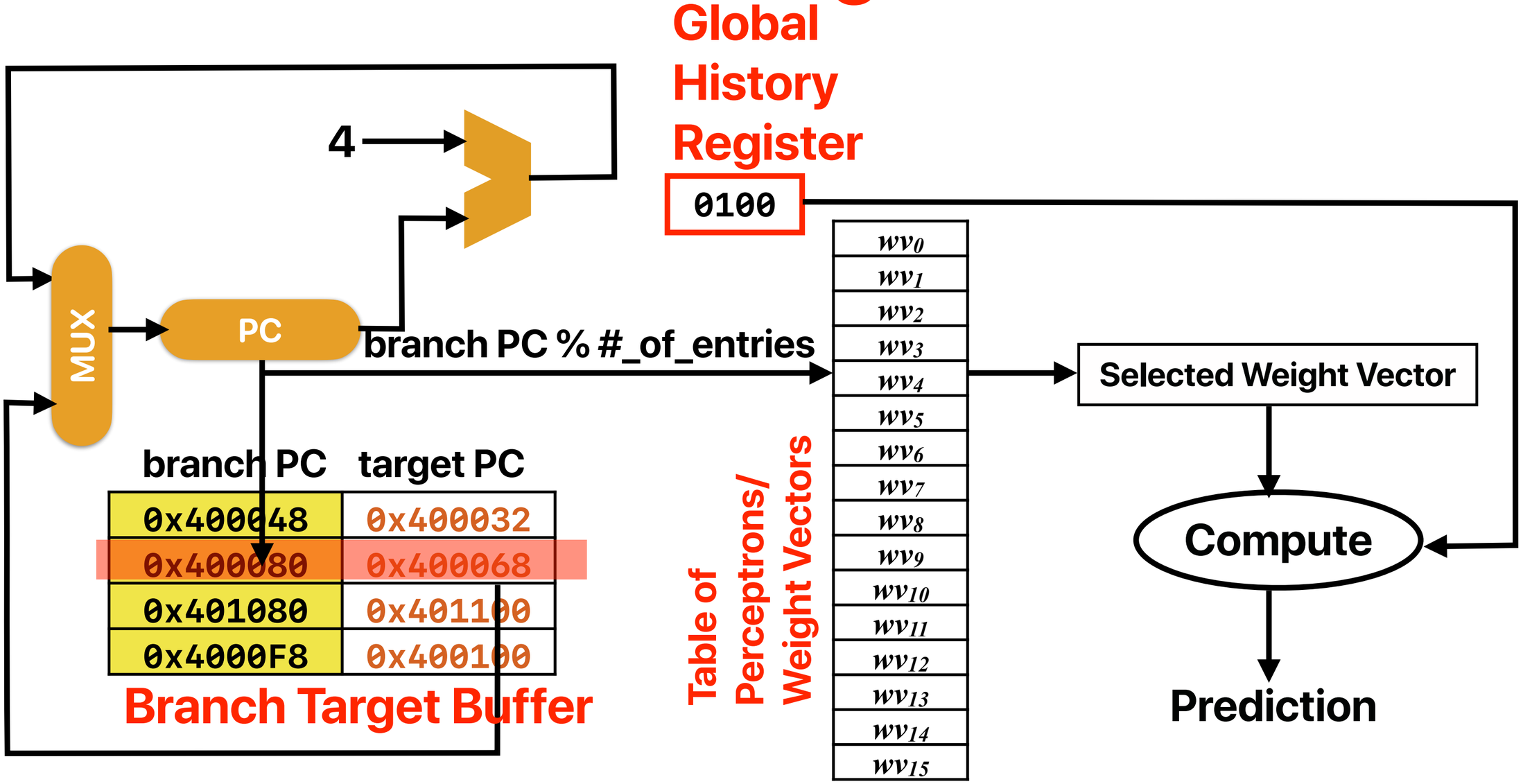
Tournament Predictor

- The state predicts “which predictor is better”
  - Local history
  - Global history
- The predicted predictor makes the prediction

Mapping Branch Prediction to NN

- The inputs to the perceptron are branch outcome histories
  - Just like in 2-level adaptive branch prediction
  - Can be global or local (per-branch) or both (alloyed)
  - Conceptually, branch outcomes are represented as
    - +1, for taken
    - -1, for not taken
- The output of the perceptron is
  - Non-negative, if the branch is predicted taken
  - Negative, if the branch is predicted not taken
- Ideally, each static branch is allocated its own perceptron

Predictor Organization



Branch predictor in processors

- 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.

Branch and programming

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 ++;
    }
}
```

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%

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 C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - ① C has lower dynamic instruction count than B — C only needs one load, one add, one shift, the same amount of iterations
  - ② C has significantly lower branch mis-prediction rate than B — the same number being predicted.
  - ③ C has significantly fewer branch instructions than B — the same amount of branches
  - ④ C can incur fewer data hazards — Probably not. In fact, the load may have negative effect without architectural supports

A. 0

B. 1

C. 2

D. 3

E. 4

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;
}
```

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;
}
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C — Compiler can do loop unrolling — no branches
  - ② D has significantly lower branch mis-prediction rate than C — Could be
  - ③ D has significantly fewer branch instructions than C — maybe eliminated through loop unrolling...
  - ④ D can incur fewer data hazards than C — about the same

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

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;
 }
```

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;
          c += table[(x & 0xF)];
          x = x >> 4;
     return c;
}
```

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

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;
}
```

Solutions/work-around of pipeline hazards

- Structural
  - Stall
  - More read/write ports
  - Split hardware units (e.g., instruction/data caches)
- Control
  - Stalls
  - Branch predictions
  - Compiler optimizations with ISA supports (e.g., delayed branch)

Data Hazards

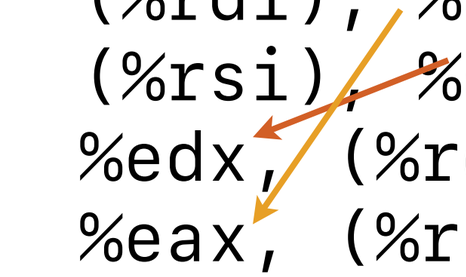
Data hazards

- 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

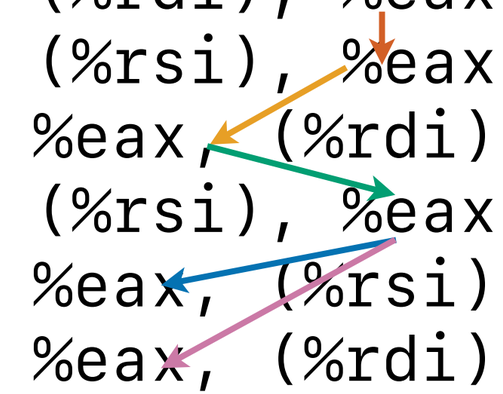
```
int temp = *a;
*a = *b;
*b = temp;
```

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

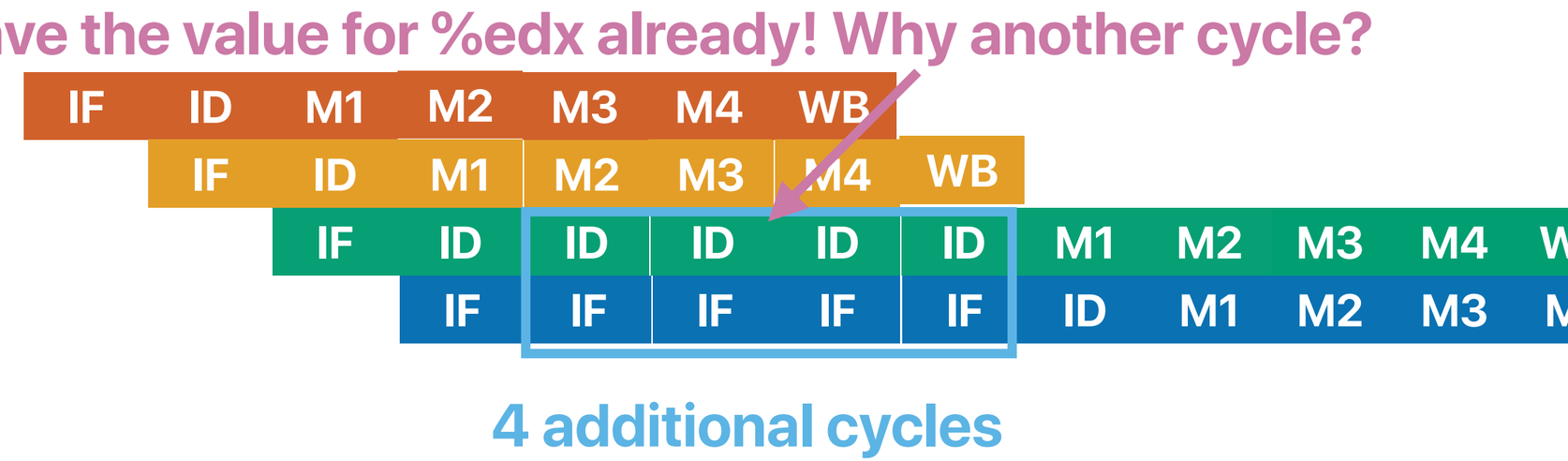
```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

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

Solution 1: Let's try "stall" again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

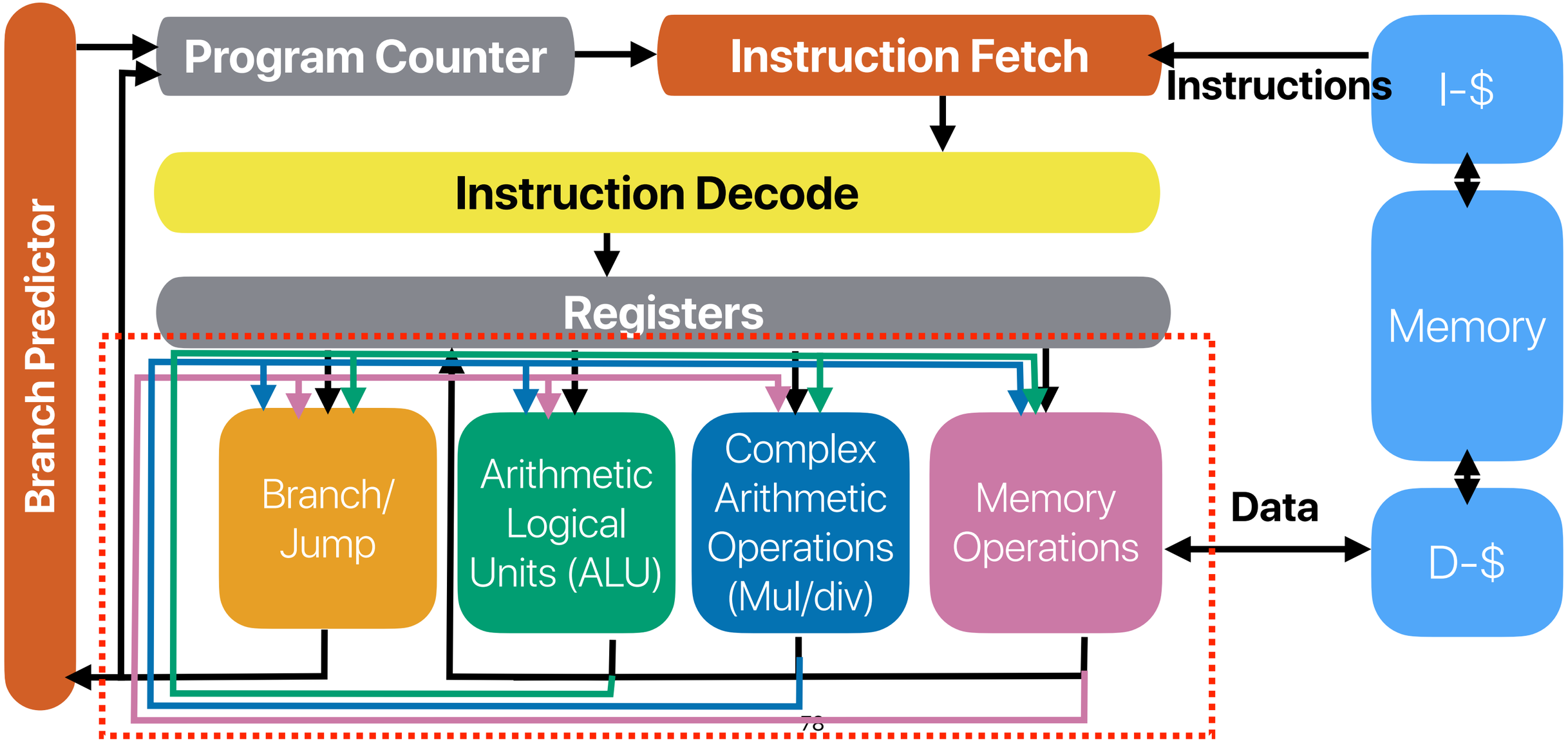
```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions

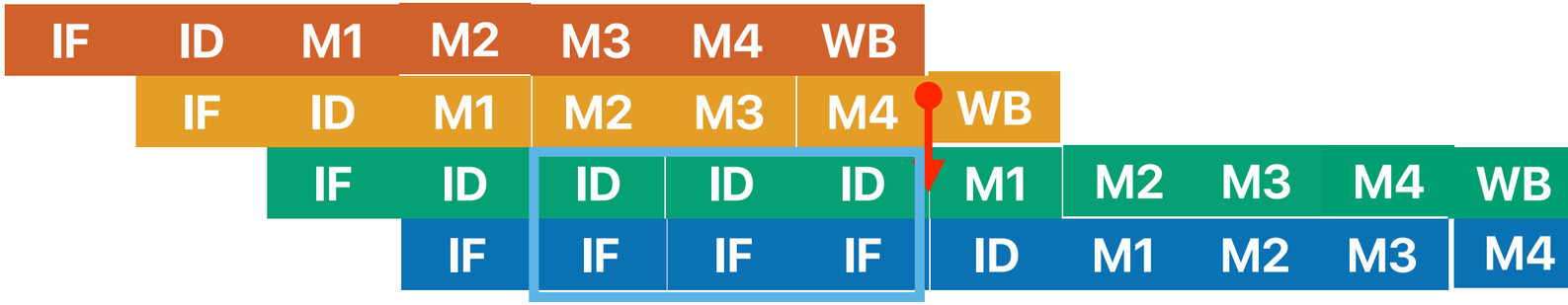
Data "forwarding"



How many data dependencies are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation (assume 100% cache hit rate) takes 4 cycles?

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

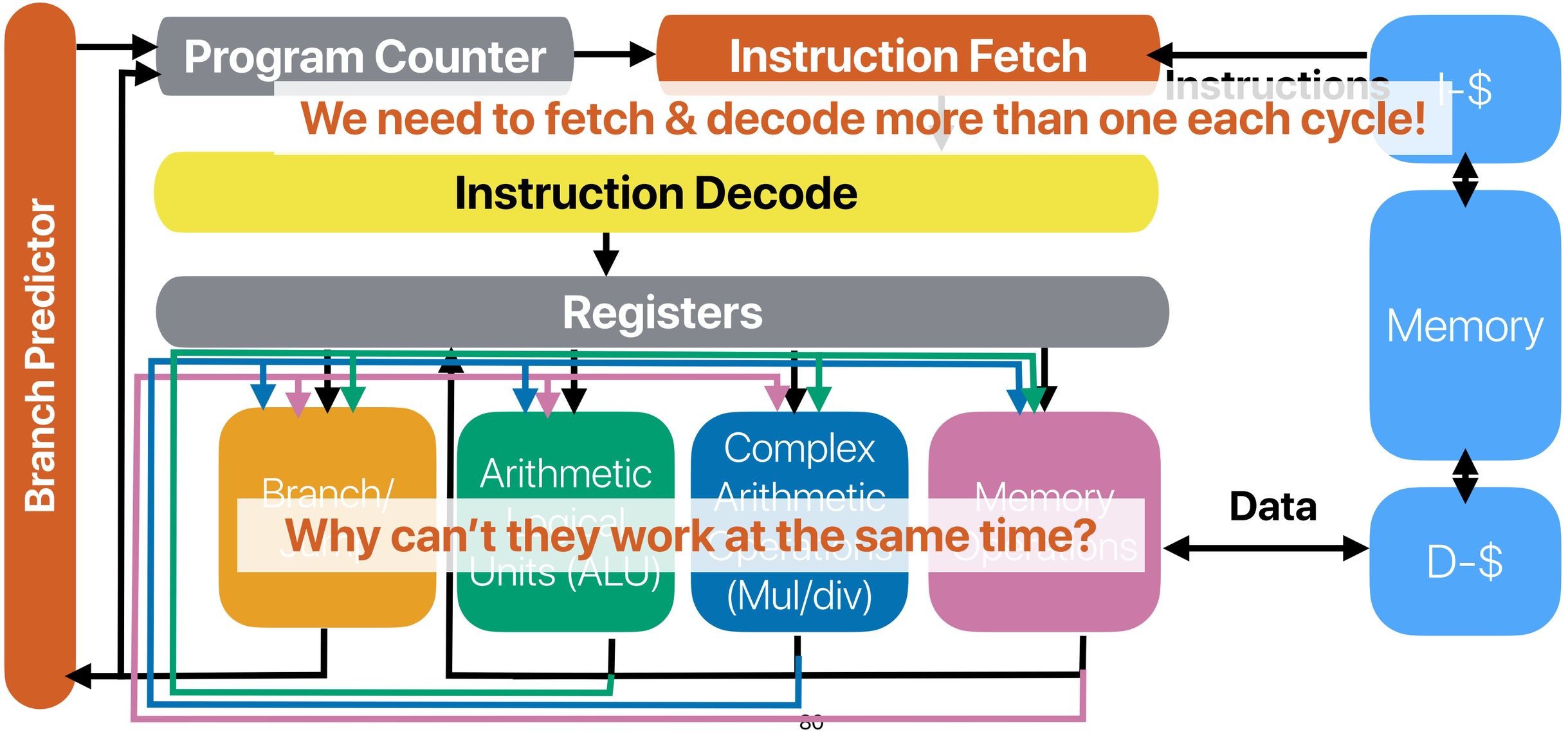


3 additional cycles

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

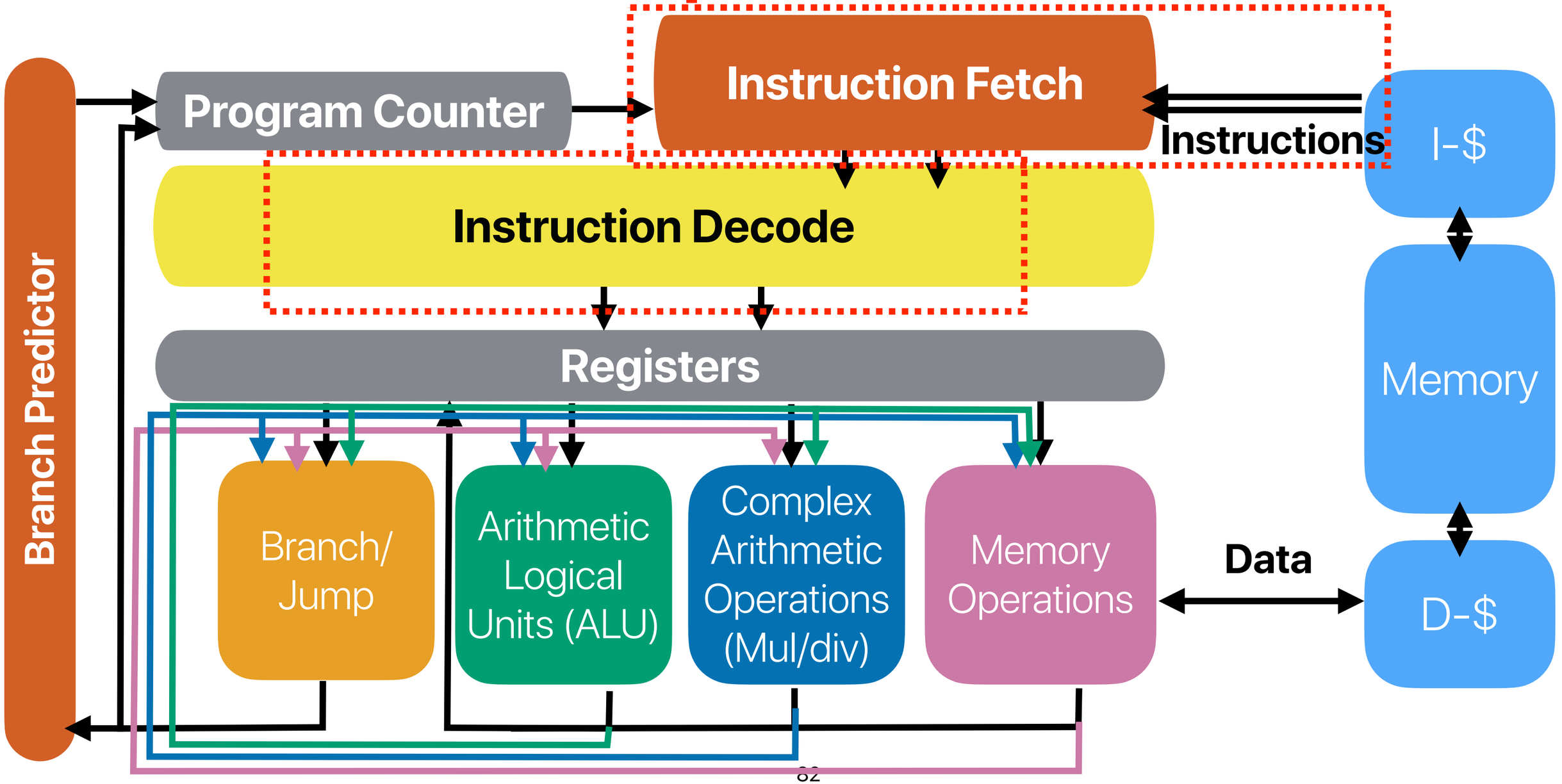
```
int temp = *a;
*a = *b;
*b = temp;
```

Data “forwarding”



Super Scalar

Super Scalar



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

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

If we loop many times (assume perfect predictor)

```
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
```

	IF	ID	M1/ALU/BR	M2	M3	M4	WB
1	(1) (2)						
2	(3)(4)	(1) (2)					
3	(5)(6)	(3)(4)	(1)(2)				
4	(5)(6)	(3)(4)		(1)(2)			
5	(5)(6)	(3)(4)			(1)(2)		
6	(5)(6)	(3)(4)				(1)(2)	
7	(7)(8)	(5)(6)	(3)(4)				(1)(2)
8	(9)(10)	(7)(8)	(5)(6)	(3)(4)			
9	(9)(10)	(8)	(7)	(5)(6)	(3)(4)		
10	(9)(10)	(8)		(7)	(5)(6)	(3)(4)	
11	(9)(10)	(8)			(7)	(5)(6)	(3)(4)
12	(11)(12)	(9)(10)	(8)			(7)	(5)(6)
	(11)(12)	(10)	(9)	(8)			(7)
		(11)(12)	(10)	(9)	(8)		
			(11)(12)	(10)	(9)	(8)	
				(11)(12)	(10)	(9)	(8)
					(11)	(10)	(9)

Everything we need for (4) is ready here! Why can't we execute it?

Why can't I start loading (6) & (11)?

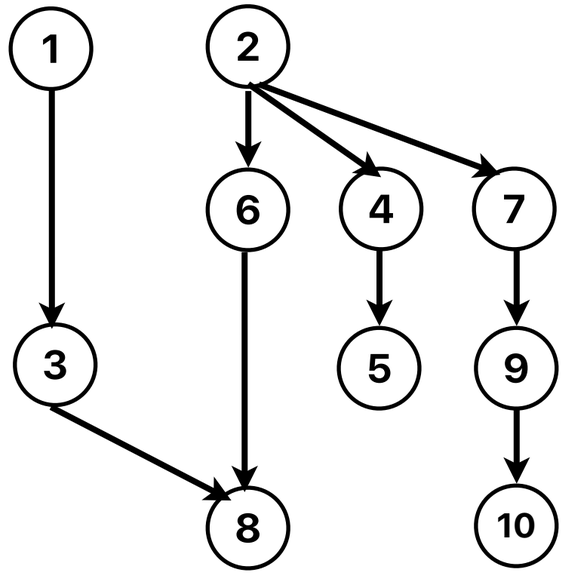
What do you need to execution an instruction?

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

Scheduling instructions: based on data dependencies

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

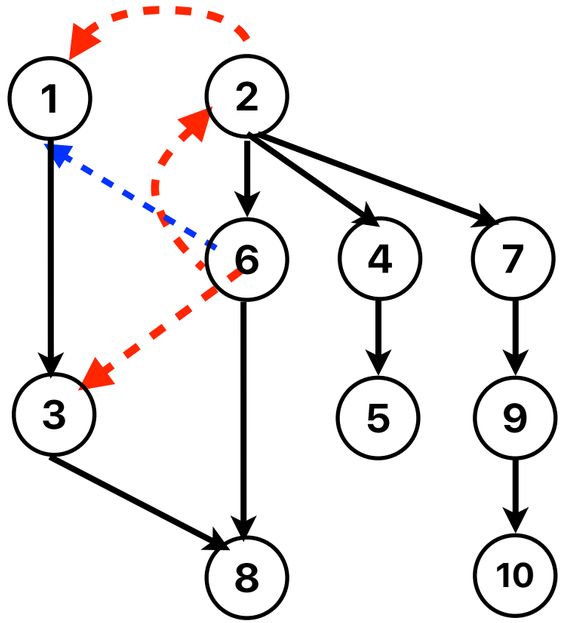
```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```



- **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
```

What if we can use more registers...

```
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
```

```
① movl    (%rdi), %ecx
② addq    $4, %rdi, %t0
③ addl    %ecx, %eax, %t1
④ cmpq    %rdx, %t0
⑤ jne     .L3
⑥ movl    (%t0), %t2
⑦ addq    $4, %t0, %t3
⑧ addl    %t1, %t2, %t4
⑨ cmpq    %rdx, %t3
⑩ jne     .L3
```

All false dependencies are gone!!!

Compiler cannot do this!!!

Register renaming + speculative execution

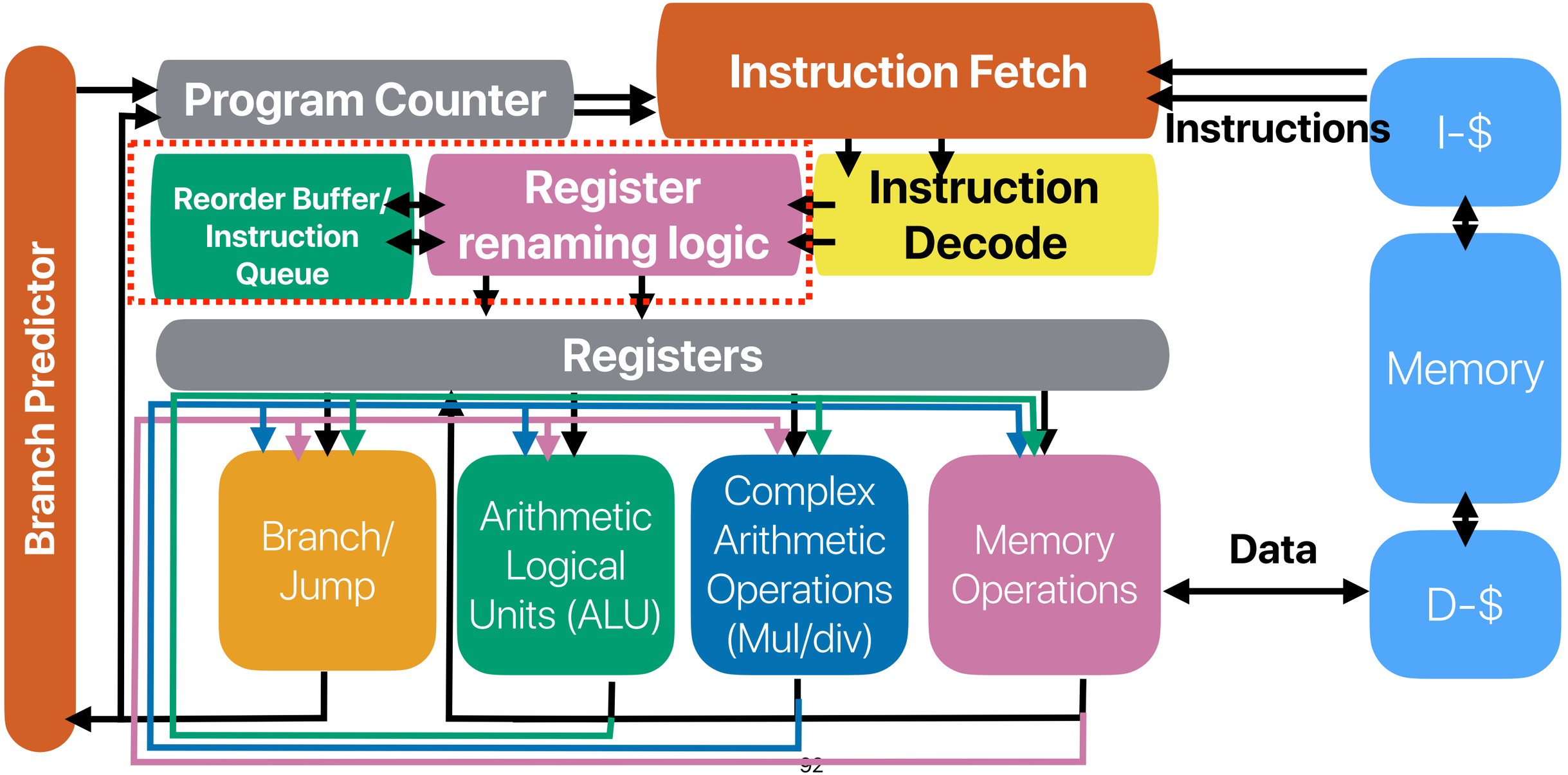
Register renaming

- 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

Speculative Execution

- Exceptions (e.g. divided by 0, page fault) may occur anytime
  - A later instruction cannot write back its own result otherwise the architectural states won't be correct
- 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
- Execute instructions across branches
  - Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Execute an instruction all operands are ready (the values of depending physical registers are generated)
  - Store results in **reorder buffer** before the processor knows if the instruction is going to be executed or not.

Recap: Register renaming



Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
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
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4											
5											
6											
7											
8											
9											
10											
11											

	Physical Register
eax	
ecx	
rdi	
rdx	

	Valid	Value	In use		Valid	Value	In use
P1				P6			
P2				P7			
P3				P8			
P4				P9			
P5				P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
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
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4		(5)(6)	(3)(4)	(1)				(2)			
5											
6											
7											
8											
9											
10											
11											

	Physical Register
eax	
ecx	P1
rdi	P2
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6			
P2	0		1	P7			
P3				P8			
P4				P9			
P5				P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
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
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6											
7											
8											
9											
10											
11											

(4) is now executing before (3)!

	Physical Register
eax	
ecx	P1
rdi	P2
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6			
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5				P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
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
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7											
8											
9											
10											
11											

	Physical Register
eax	P6
ecx	P1
rdi	P5
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5	0		1	P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
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
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8											
9											
10											
11											

	Physical Register
eax	P6
ecx	P1
rdi	P5
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5	0		1	P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			(1)(2)(4)(5)(7)
9											
10											
11											

	Physical Register
eax	P6
ecx	P7
rdi	P8
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7	0		1
P3	0		1	P8	0		1
P4	0		1	P9			
P5	1		1	P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			(1)(2)(4)(5)(7)
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			(3)(4)(5)(7)
10											
11											

	Physical Register
eax	P9
ecx	P7
rdi	P8
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	1		0	P6	0		1
P2	1		0	P7	0		1
P3	1		1	P8	0		1
P4	0		1	P9	0		1
P5	1		1	P10			

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)~~(2)(4)(5)(7)
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	(6)(7)(9)
11											

	Physical Register
eax	P9
ecx	P7
rdi	P8
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	1		0	P6	0		1
P2	1		0	P7	0		1
P3	1		0	P8	0		1
P4	0		1	P9	0		1
P5	1		1	P10	0		1

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)~~(2)(4)(5)(7)
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	~~(6)(7)~~(9)
11		(19)(20)	(13)(14)(15)(16)(17)(18)			(11)		(12)			(8)(9)(10)

	Physical Register
eax	P6
ecx	P1
rdi	P5
rdx	P4

	Valid	Value	In use		Valid	Value	In use
P1	1		0	P6	1		1
P2	1		0	P7	0		1
P3	1		0	P8	0		1
P4	1		0	P9	0		1
P5	1		1	P10	0		1

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)(2)(4)(5)(7)~~
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	~~(6)(7)~~(9)
11		(19)(20)	(13)(14)(15)(16)(17)(18)			(11)		(12)			~~(8)(9)(10)~~
12			(13)(15)(17)(18)(19)(20)	(16)			(11)	(14)			(12)
13											
14											
15											

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)(2)(4)(5)(7)~~
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	~~(6)(7)~~(9)
11		(19)(20)	(13)(14)(15)(16)(17)(18)			(11)		(12)			~~(8)(9)(10)~~
12			(13)(15)(17)(18)(19)(20)	(16)			(11)	(14)			(12)
13			(17)(18)(19)(20)		(16)			(13)		(15)	(11)(12)(14)
14											
15											

Register renaming

2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)(2)~~(4)(5)(7)
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	~~(6)(7)~~(9)
11		(19)(20)	(13)(14)(15)(16)(17)(18)			(11)		(12)			~~(8)(9)(10)~~
12			(13)(15)(17)(18)(19)(20)	(16)			(11)	(14)			(12)
13			(17)(18)(19)(20)		(16)			(13)		(15)	~~(11)(12)~~(14)
14						(16)		(17)			(13)(14)(15)
15											

Register renaming

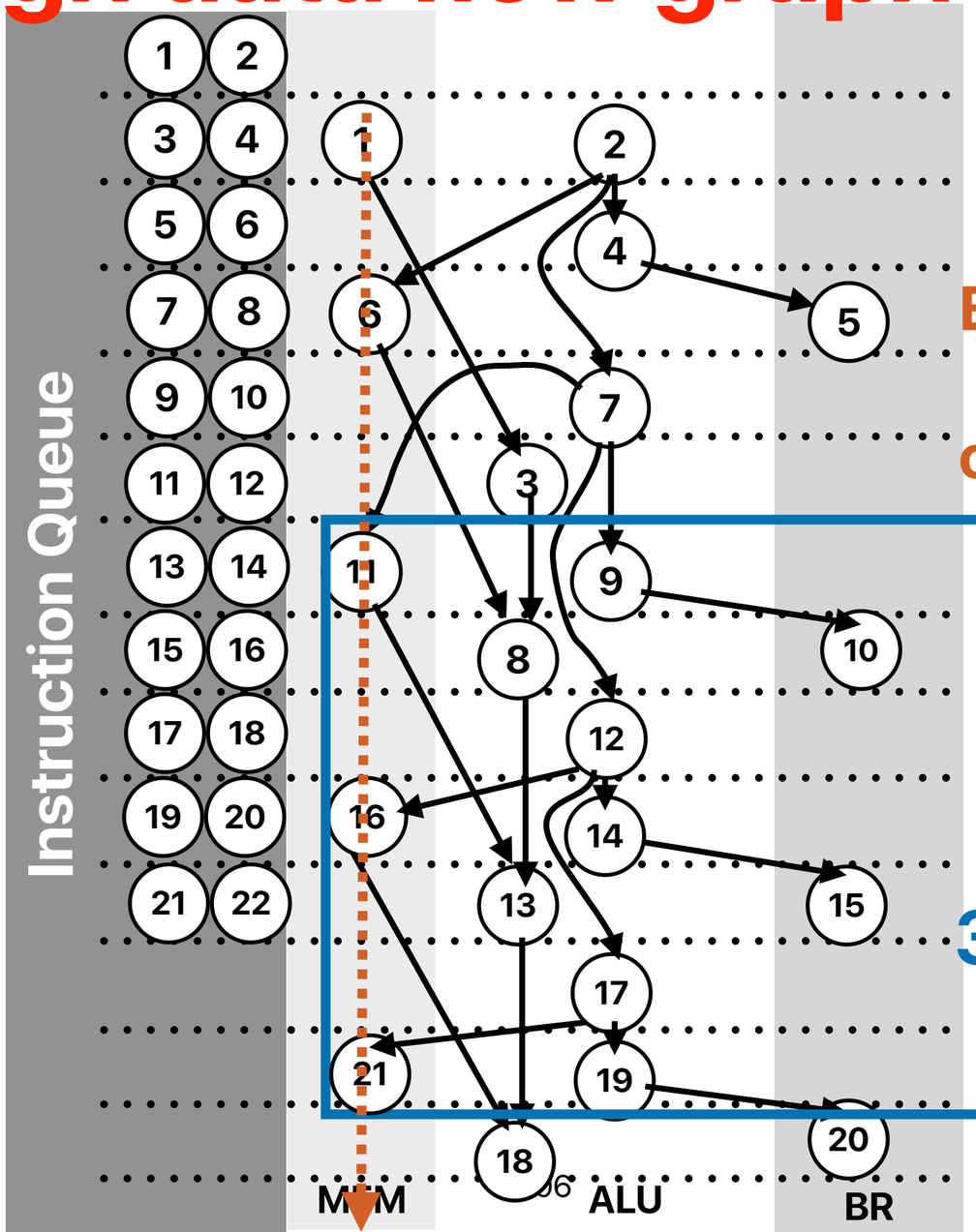
2-issue: Only 2 of them can have instructions at the same cycle

```
① movl   (%rdi), %ecx → P1
② addq   $4, %rdi → P2
③ addl   %ecx, %eax → P3
④ cmpq   %rdx, %rdi
⑤ jne    .L3
⑥ movl   (%rdi), %ecx → P4
⑦ addq   $4, %rdi → P5
⑧ addl   %ecx, %eax → P6
⑨ cmpq   %rdx, %rdi
⑩ jne    .L3
⑪ movl   (%rdi), %ecx → P7
⑫ addq   $4, %rdi → P8
⑬ addl   %ecx, %eax → P9
⑭ cmpq   %rdx, %rdi
⑮ jne    .L3
```

	IF	ID	REN	M1	M2	M3	M4	ALU	MUL	BR	ROB
1	(1) (2)										
2	(3)(4)	(1) (2)									
3	(5)(6)	(3)(4)	(1) (2)								
4	(7)(8)	(5)(6)	(3)(4)	(1)				(2)			
5	(9)(10)	(7)(8)	(3)(5)(6)		(1)			(4)			(2)
6	(11)(12)	(9)(10)	(3)(7)(8)	(6)		(1)				(5)	(2)(4)
7	(13)(14)	(11)(12)	(3)(8)(9)(10)		(6)		(1)	(7)			(2)(4)(5)
8	(15)(16)	(13)(14)	(8)(9)(10)(11)(12)			(6)		(3)			~~(1)(2)(4)(5)(7)~~
9	(17)(18)	(15)(16)	(8)(10)(12)(13)(14)	(11)			(6)	(9)			~~(3)(4)(5)(7)~~
10	(19)(20)	(17)(18)	(12)(13)(14)(15)(16)		(11)			(8)		(10)	~~(6)(7)~~(9)
11		(19)(20)	(13)(14)(15)(16)(17)(18)			(11)		(12)			~~(8)(9)(10)~~
12			(13)(15)(17)(18)(19)(20)	(16)			(11)	(14)			(12)
13			(17)(18)(19)(20)		(16)			(13)		(15)	~~(11)(12)~~(14)
14						(16)		(17)			~~(13)(14)(15)~~
15							(16)	(19)			(17)

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

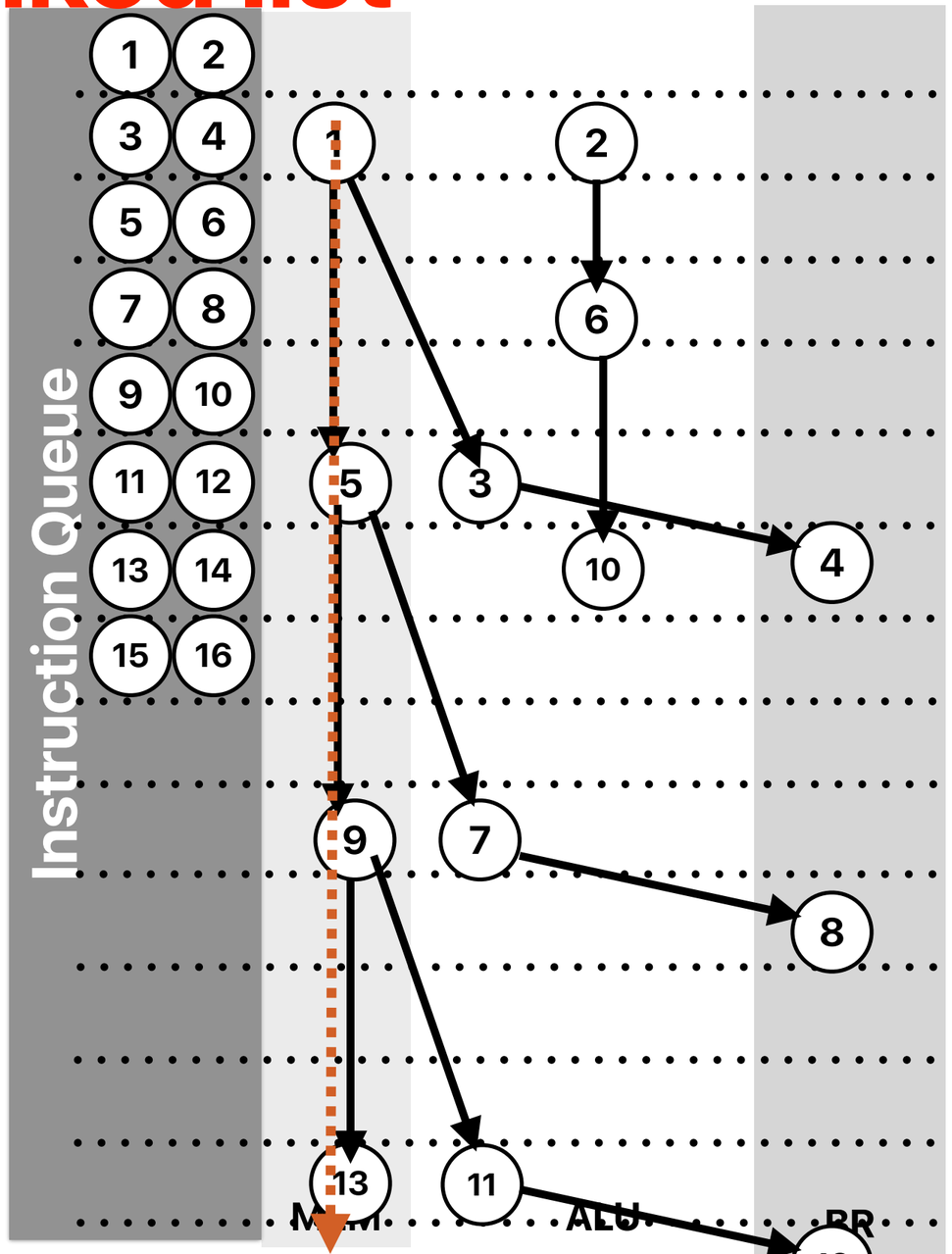
4 cycles each iteration

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
  - Stall
  - More read/write ports
  - Split hardware units (e.g., instruction/data caches)
- Control
  - Stalls
  - Branch predictions
  - Compiler optimizations with ISA supports (e.g., delayed branch)
- Data
  - Stalls
  - Data forwarding
  - Compiler optimizations
  - Dynamic scheduling

Why compiler optimization is insufficient

- Compiler cannot predict “dynamic” events
  - Compiler doesn’t know if the branch is going or likely to taken or not
    - Loop unrolling doesn’t always work
  - Compiler doesn’t know if the memory access is going to be a miss or not
- Compiler can only optimize on features exposed by hardware
  - Compiler can only see “architectural registers”, but cannot utilize “physical registers”
    - Has very limited power in “renaming”
    - Creates “false dependencies”
  - Compiler optimization cannot be adaptive to micro architectural changes — what if the pipeline changes?

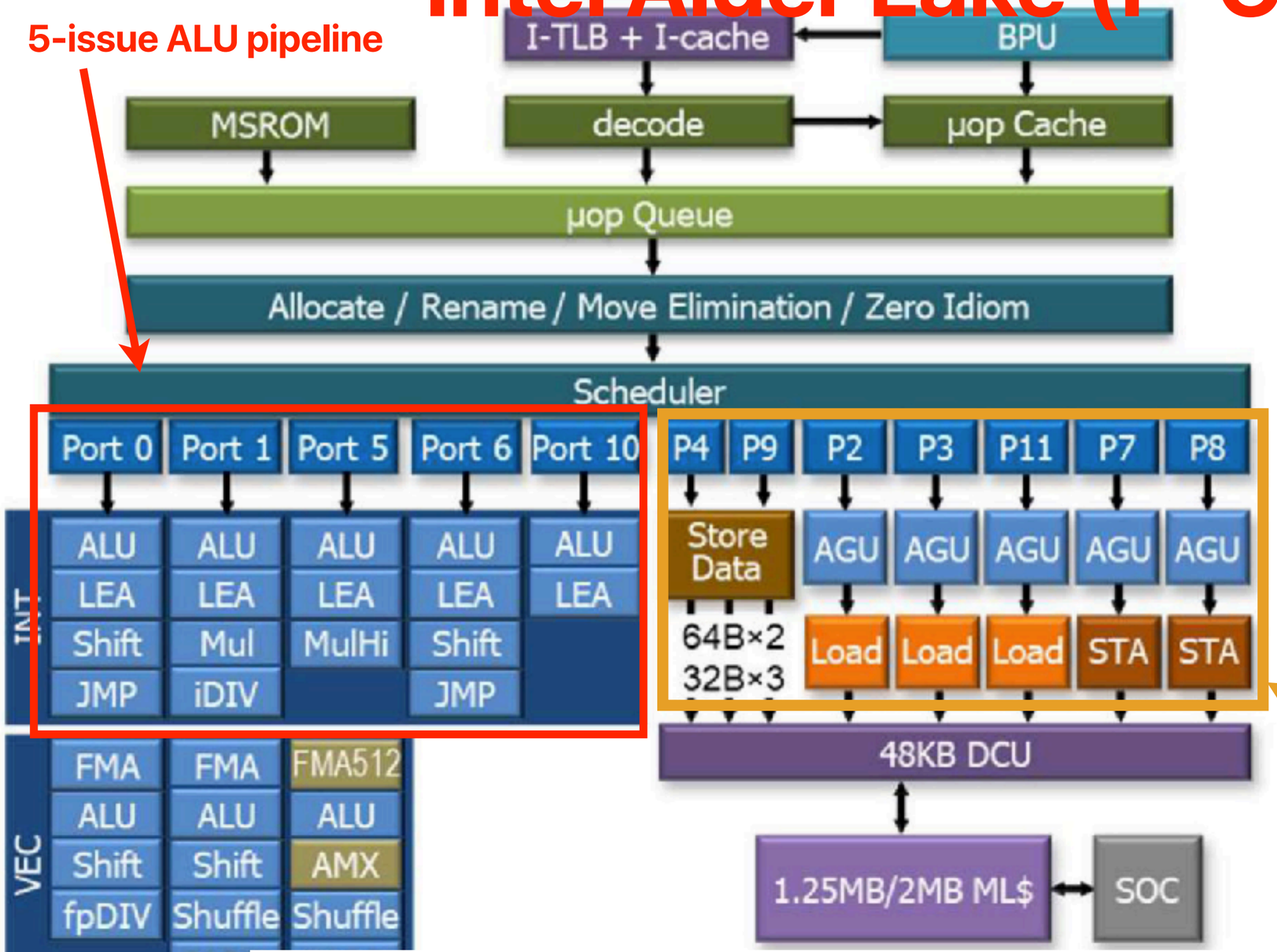
Intel Alder Lake (P-Core)

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

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

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

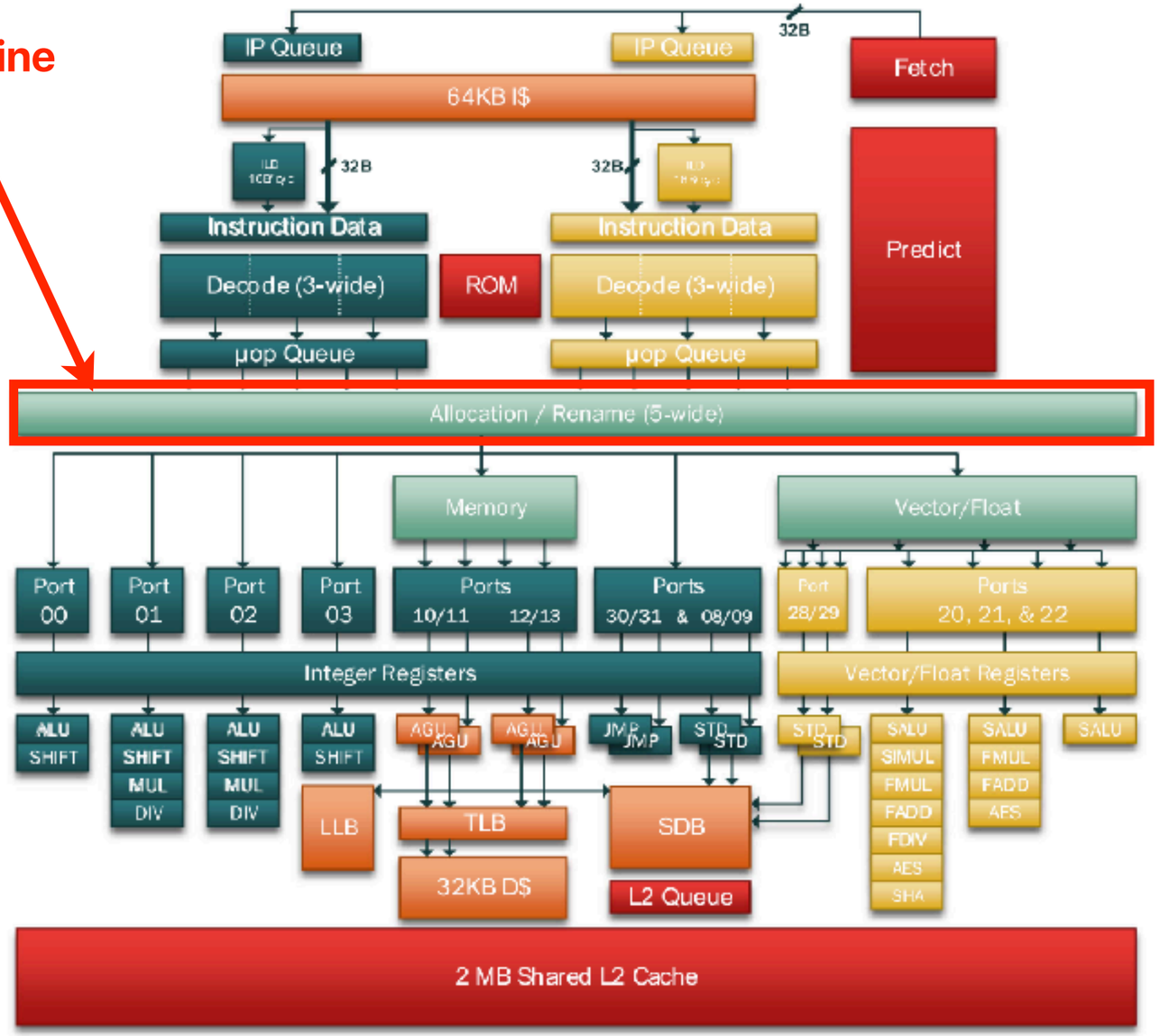
$$MinBRInst.CPI = \frac{1}{2}$$

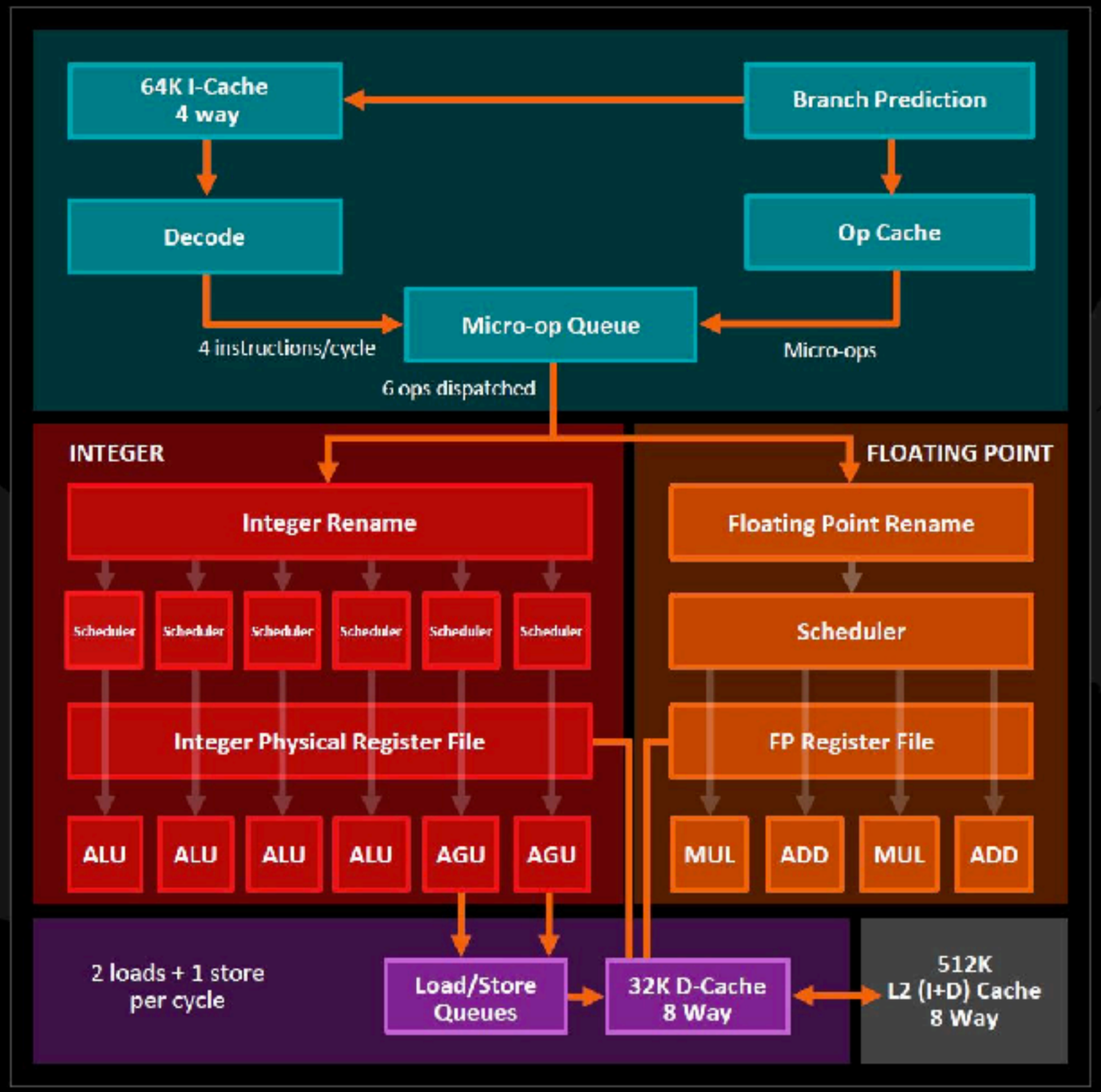


https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

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

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

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

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

```
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;
}
```

Recap: What about “linked list”

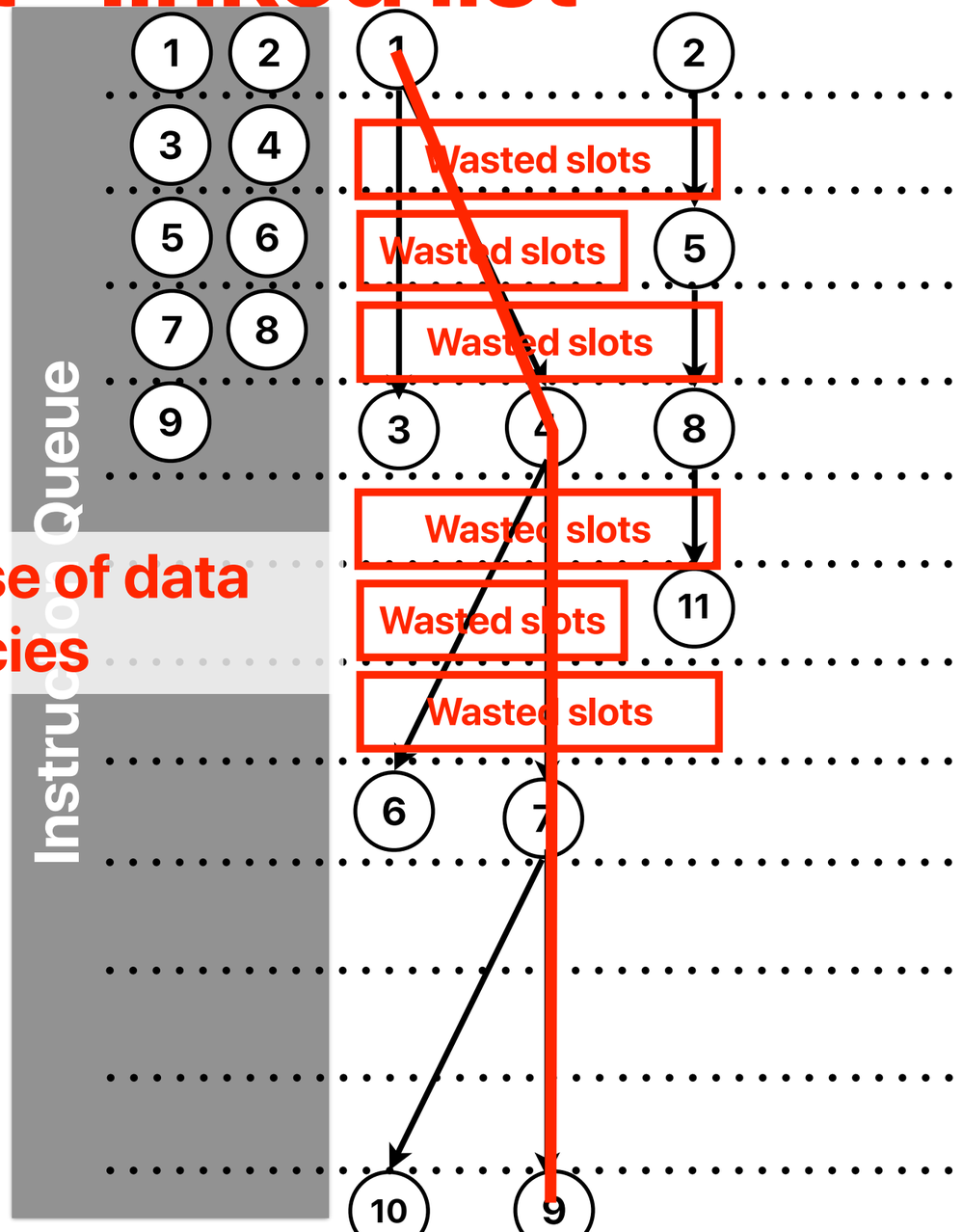
Static instructions

```
LOOP: ld   X10, 8(X10)
      addi  X7, X7, 1
      bne  X10, X0, LOOP
```

Dynamic instructions

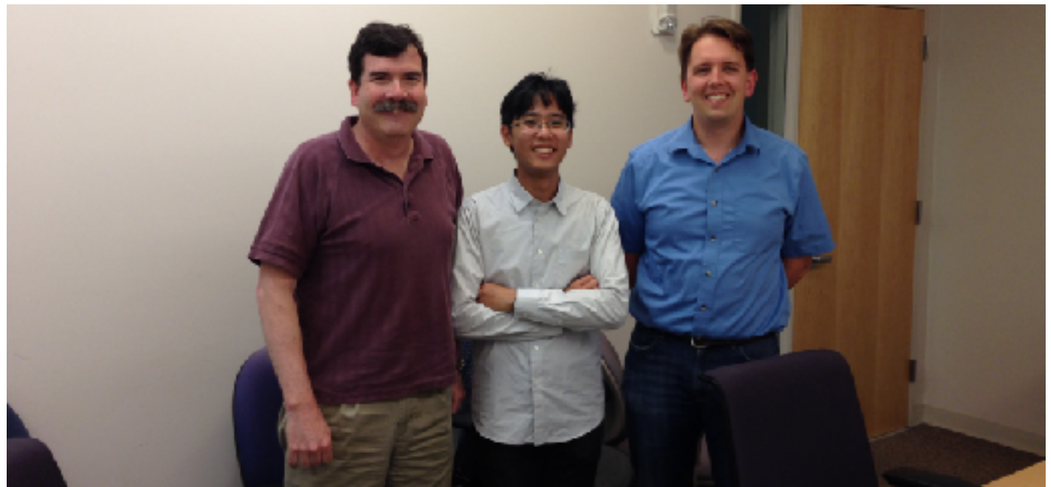
```
① ld   X10, 8(X10)
② addi  X7, X7, 1
③ bne  X10, X0, LOOP
④ ld   X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne  X10, X0, LOOP
⑦ ld   X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne  X10, X0, LOOP
```

ILP is low because of data dependencies



Simultaneous multithreading: maximizing on-chip parallelism

Dean M. Tullsen, Susan J. Eggers, Henry M. Levy
Department of Computer Science and Engineering, University of Washington

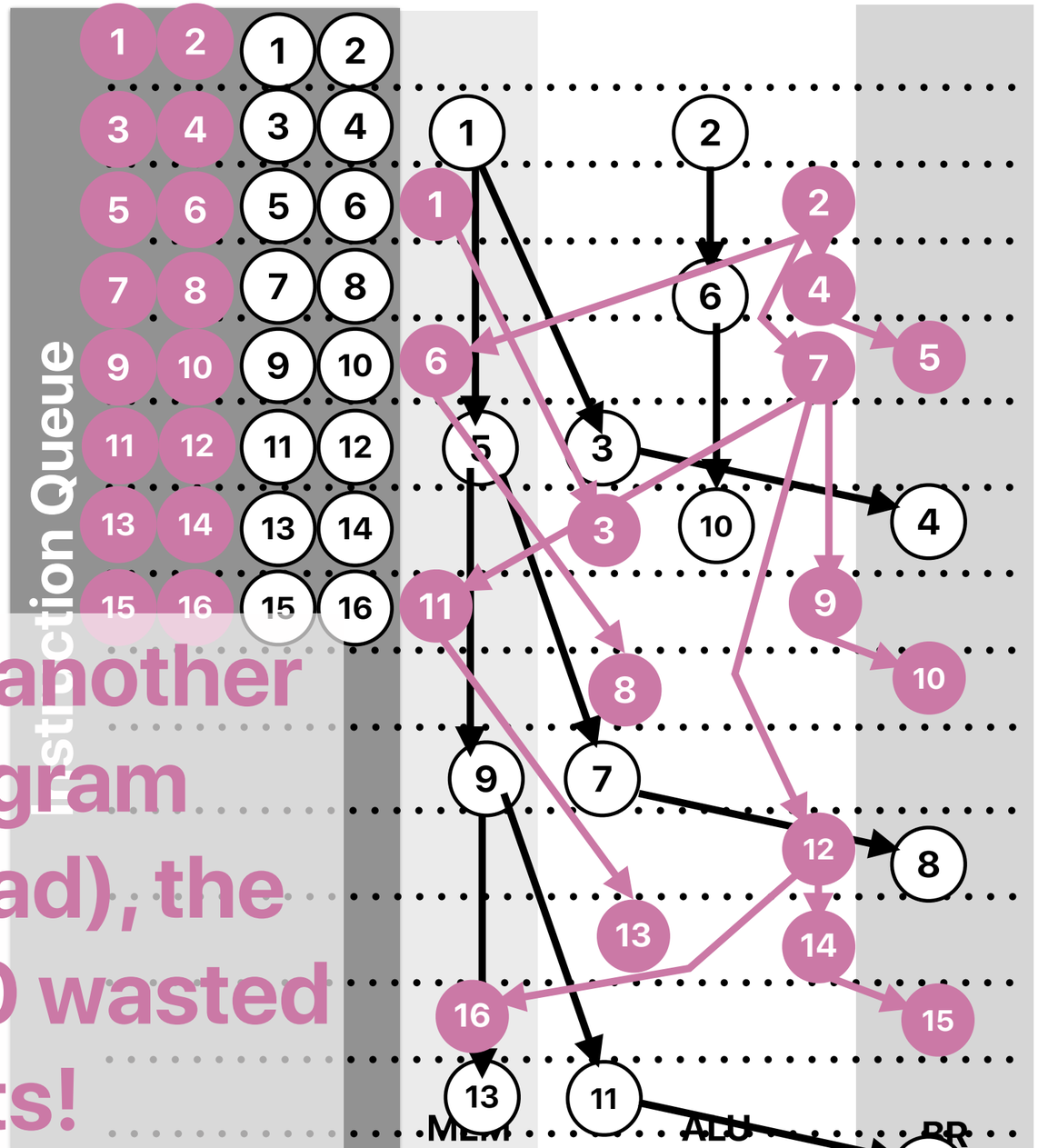


Simultaneous multithreading

- The processor can schedule instructions from different threads/processes/programs
- Fetch instructions from different threads/processes to fill the not utilized part of pipeline
  - Exploit “thread level parallelism” (TLP) to solve the problem of insufficient ILP in a single thread
  - You need to create an illusion of multiple processors for OSs

Concept: Simultaneous Multithreading (SMT)

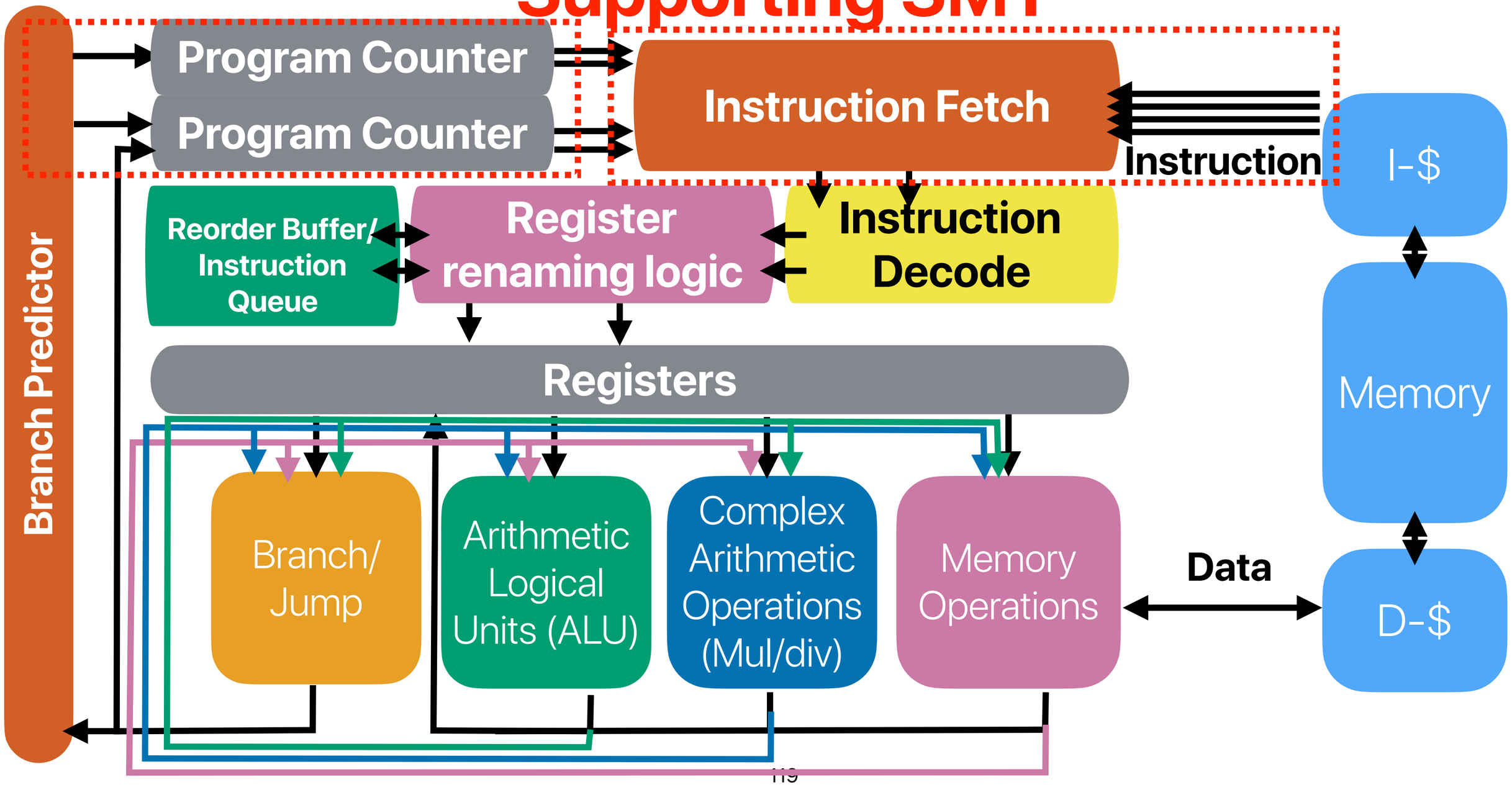
```
①  movq    8(%rdi), %rdi
②  addl    $1, %eax
③  testq   %rdi, %rdi
④  jne     .L3
⑤  movq    8(%rdi), %rdi
⑥  addl    $1, %eax
⑦  testq   %rdi, %rdi
⑧  jne     .L3
⑨  movq    8(%rdi), %rdi
⑩  addl    $1, %eax
⑪  testq   %rdi, %rdi
⑫  jne     .L3
```



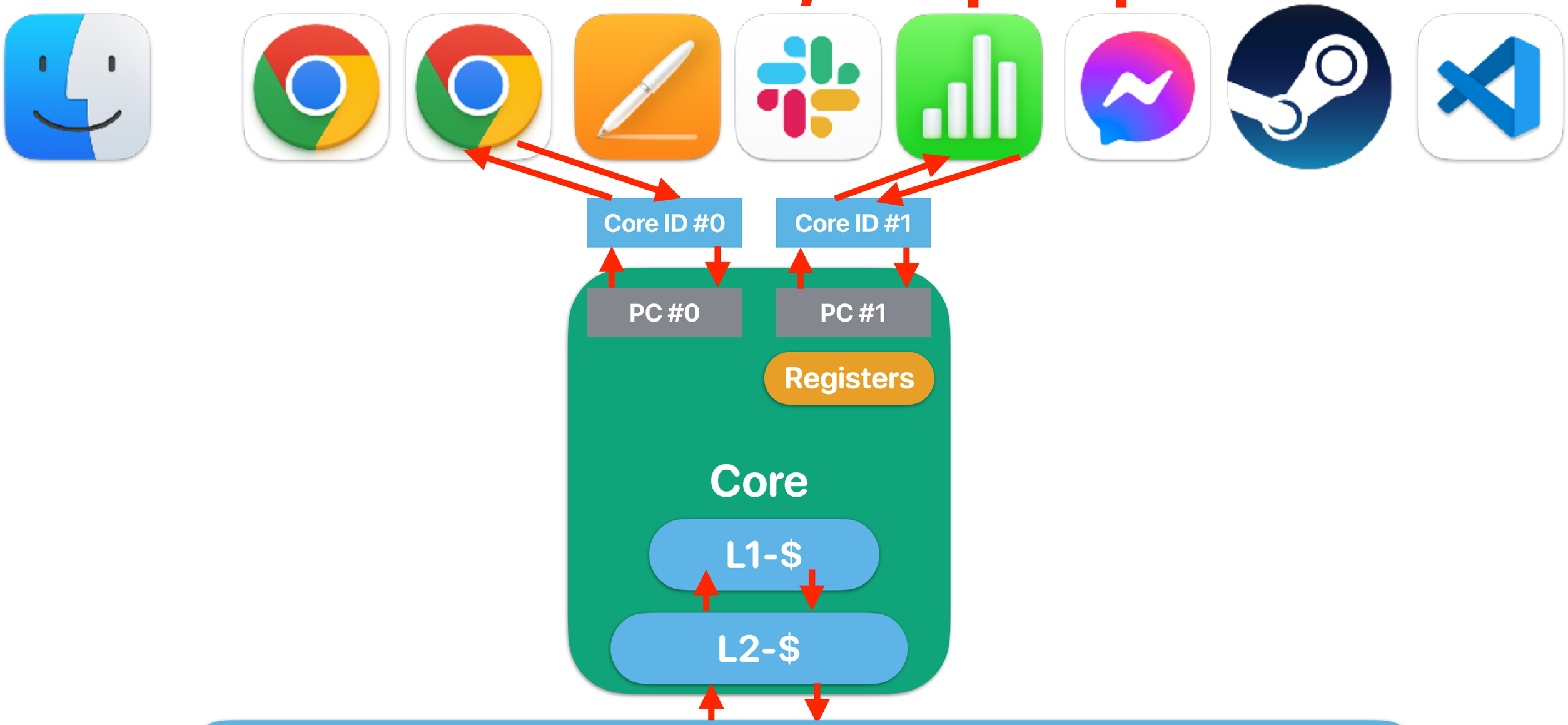
By scheduling another running program instance (thread), the processor has 0 wasted issue slots!

```
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
```

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

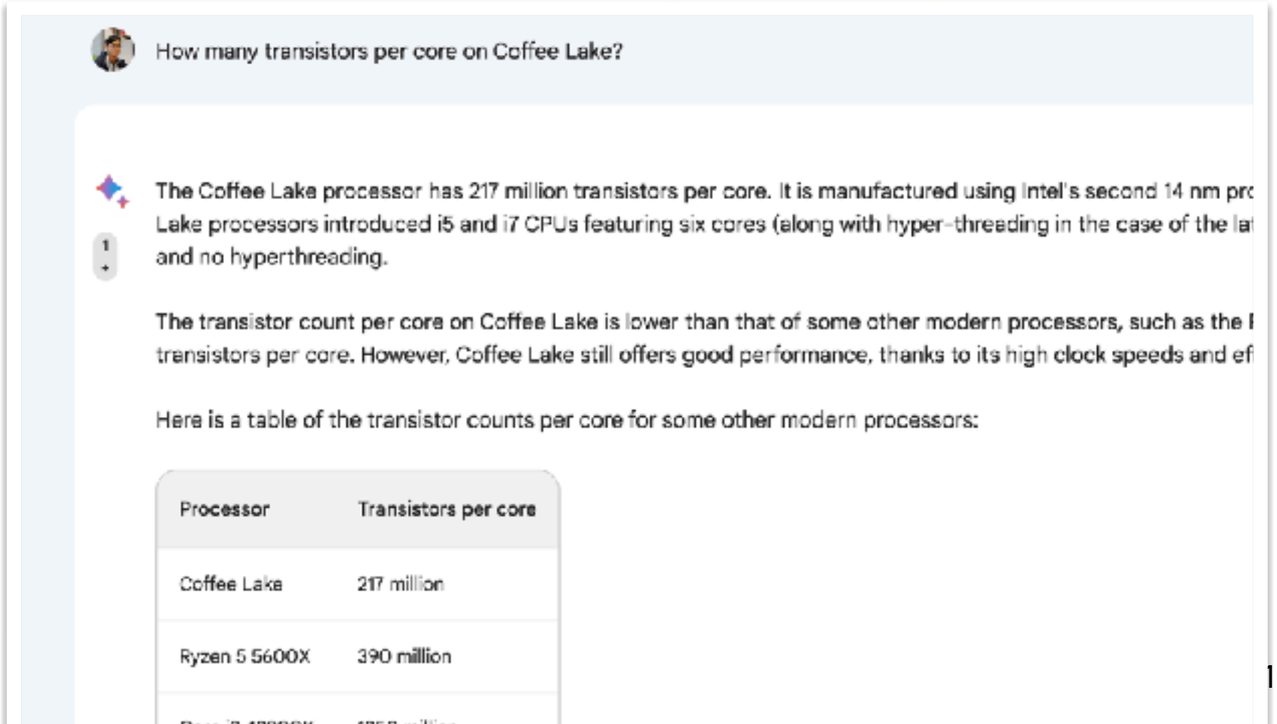
hurt, b/c you are sharing resource with other threads.

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

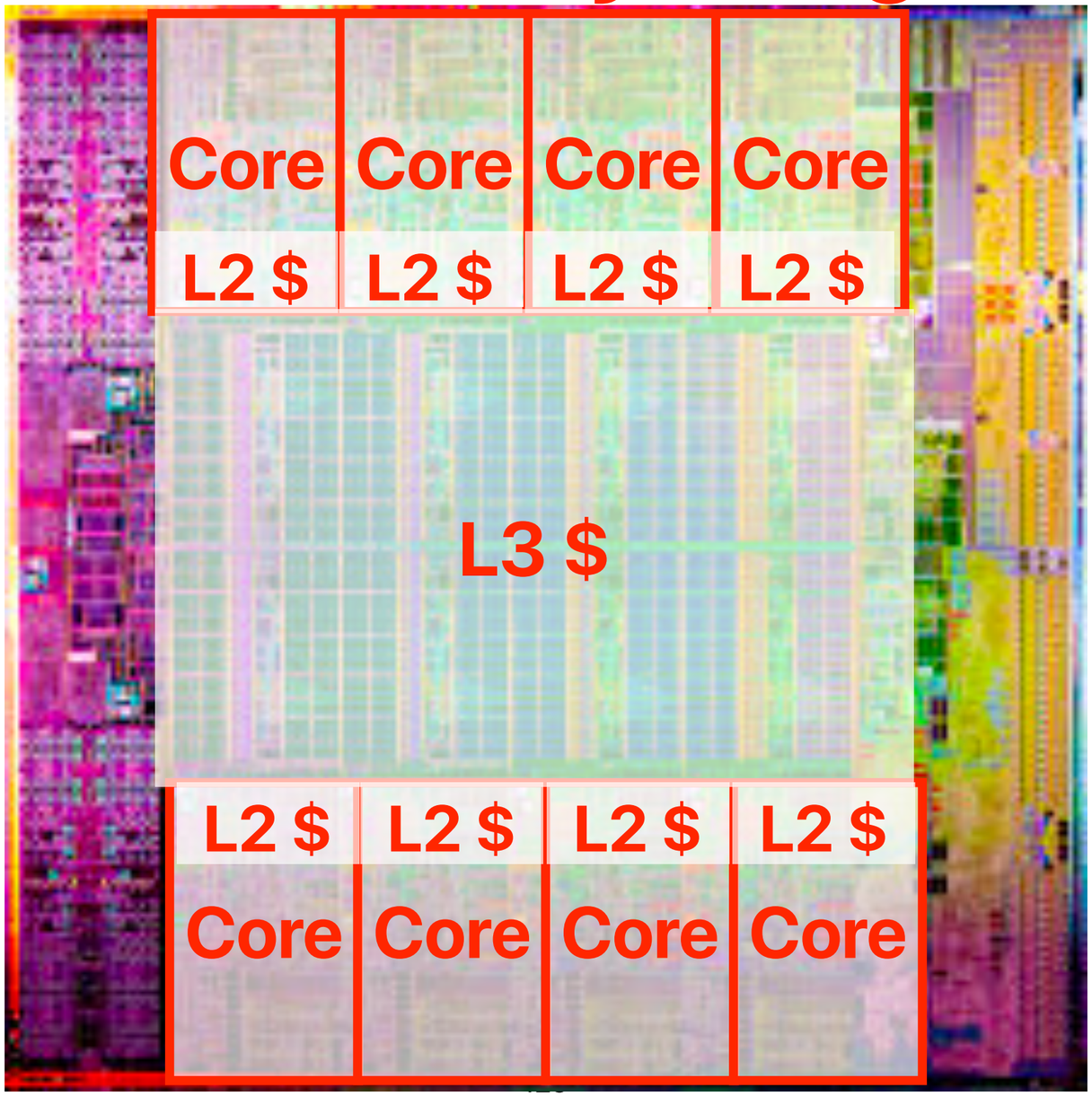
2x 3-issue ALUs Nehalem

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

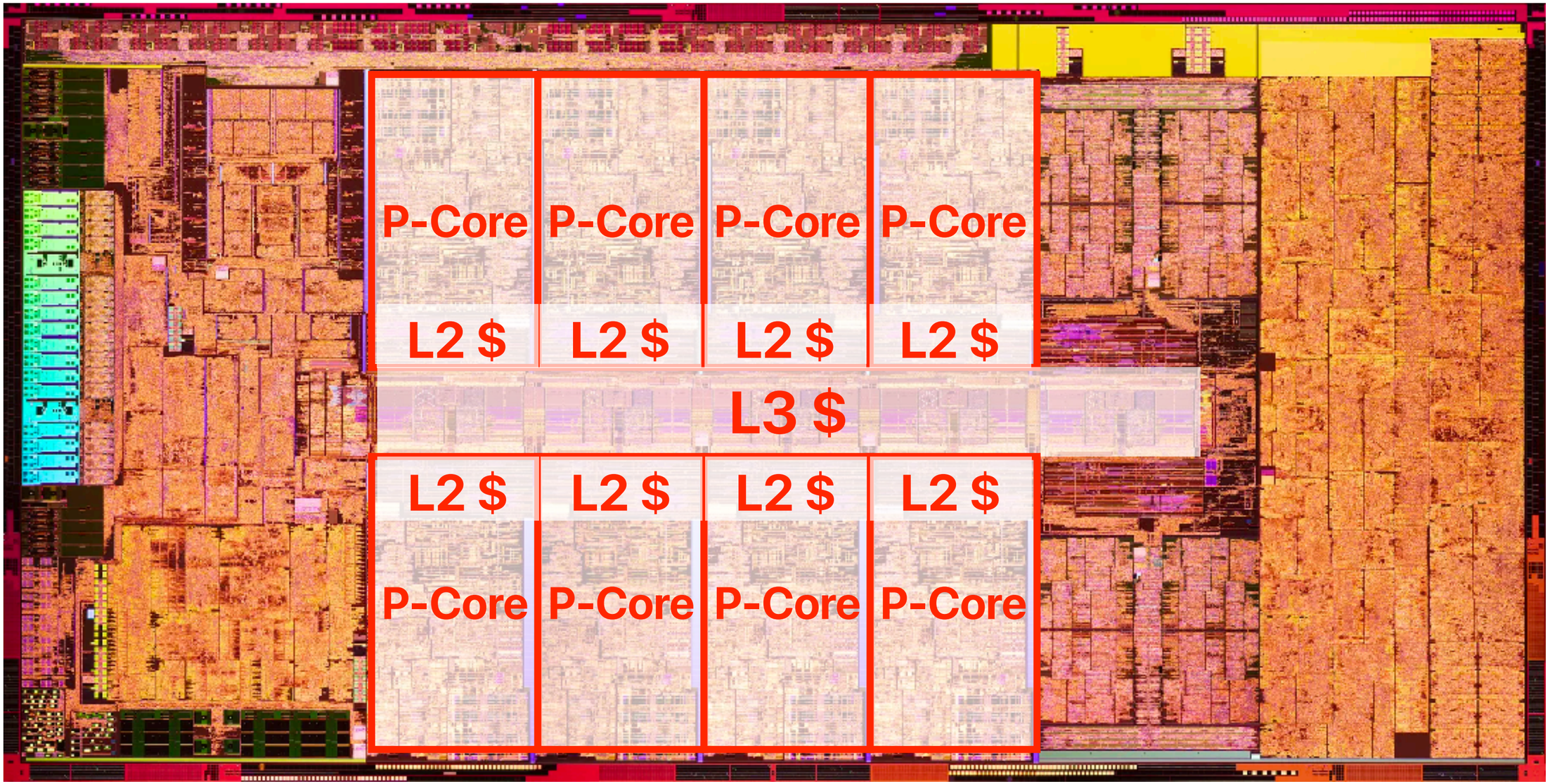
1x 5-issue ALUs Alder Lake

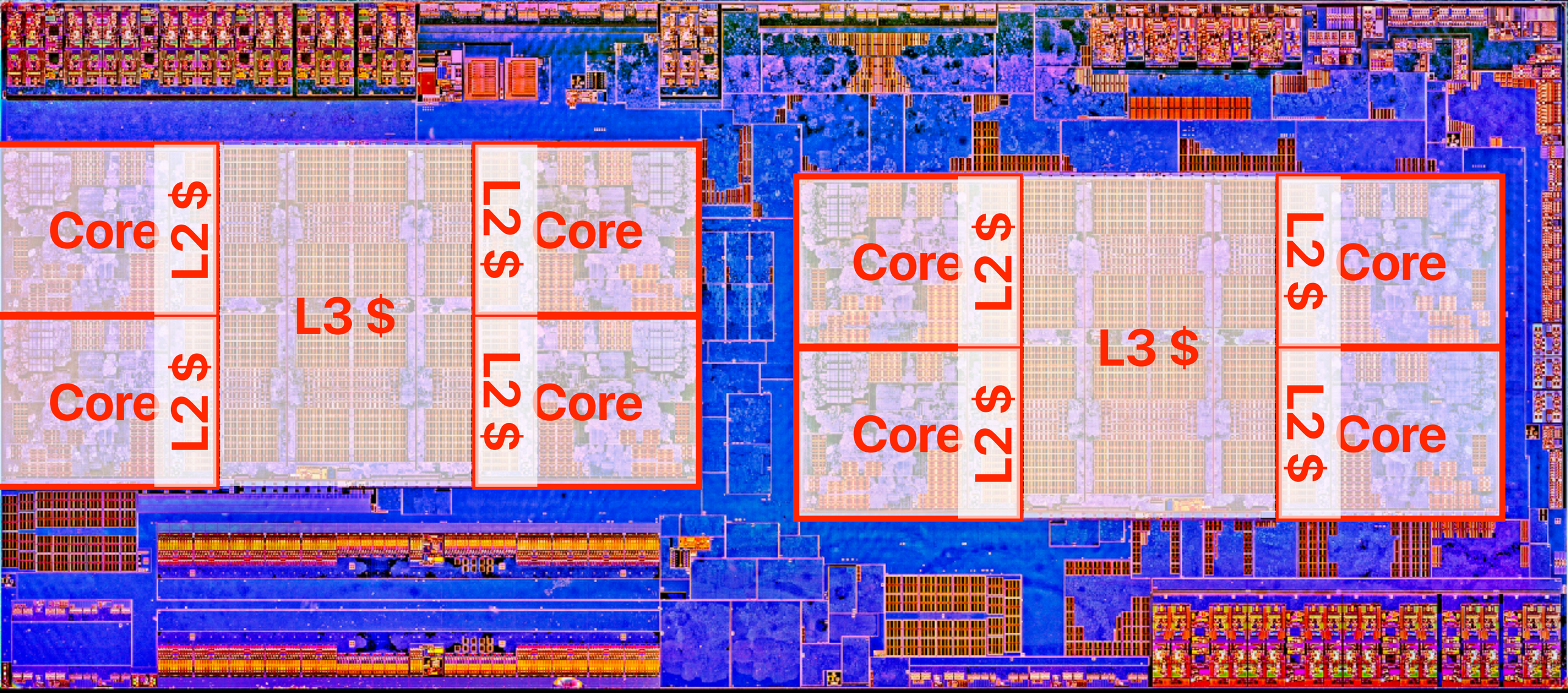
Based on https://en.wikipedia.org/wiki/Transistor_count

Intel Sandy Bridge

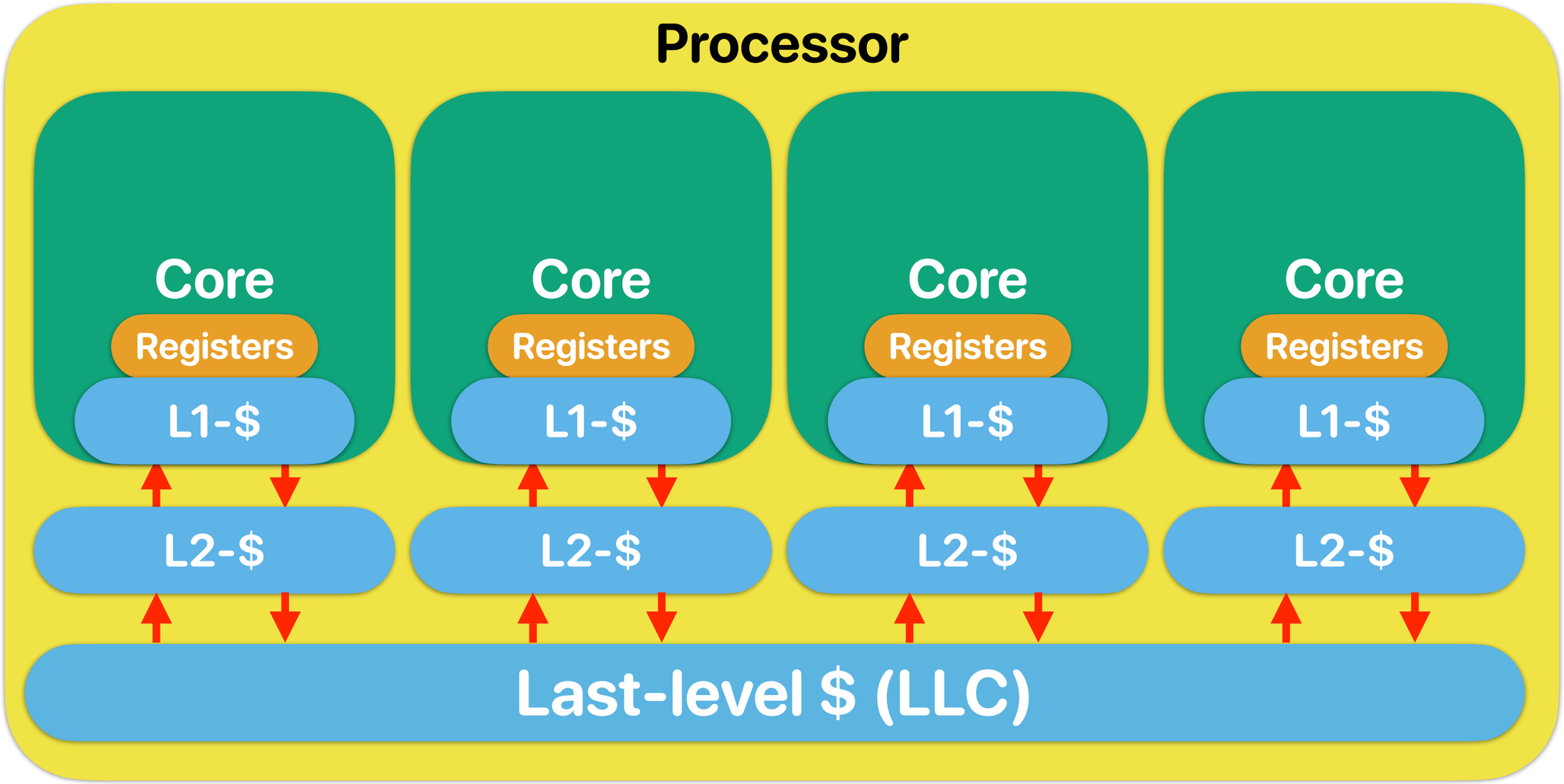


Intel Alder Lake




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

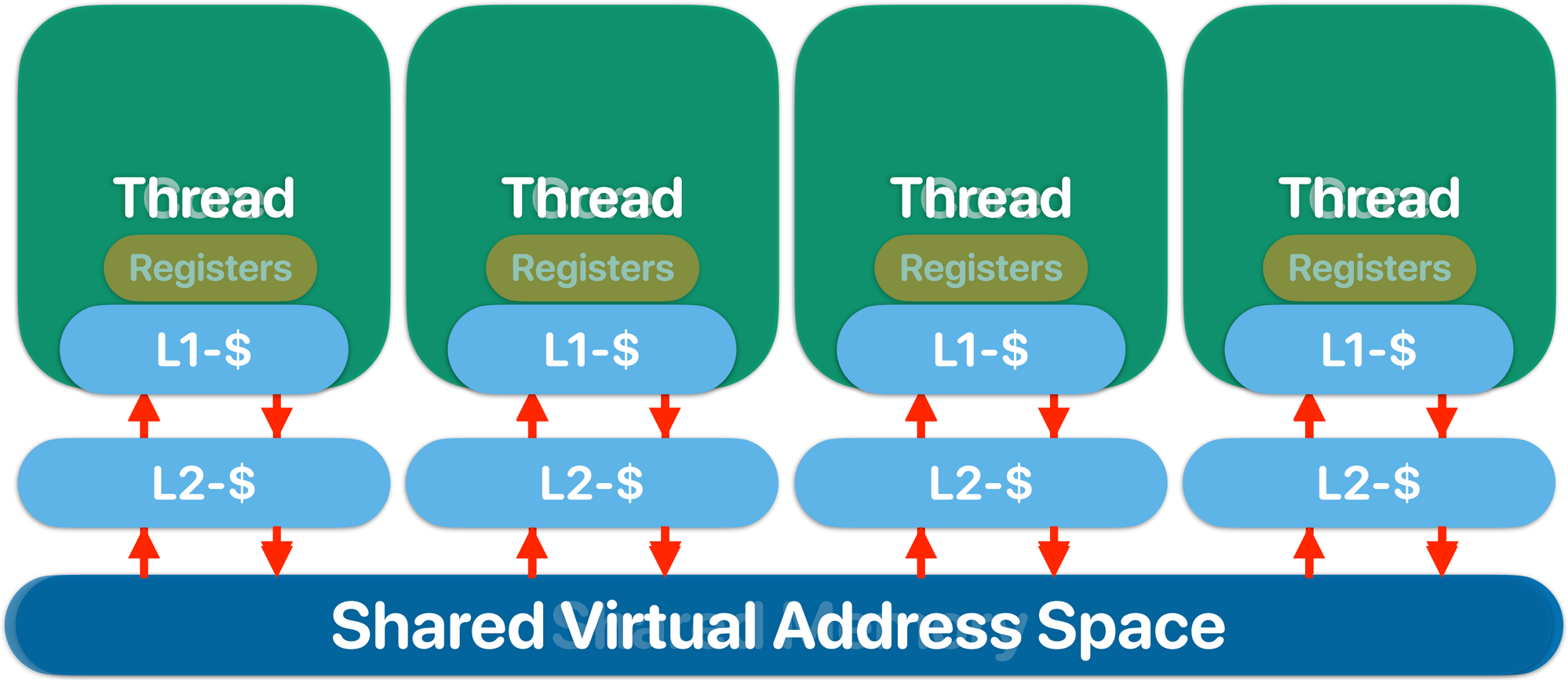
There is no clear win on each — why not having both?

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

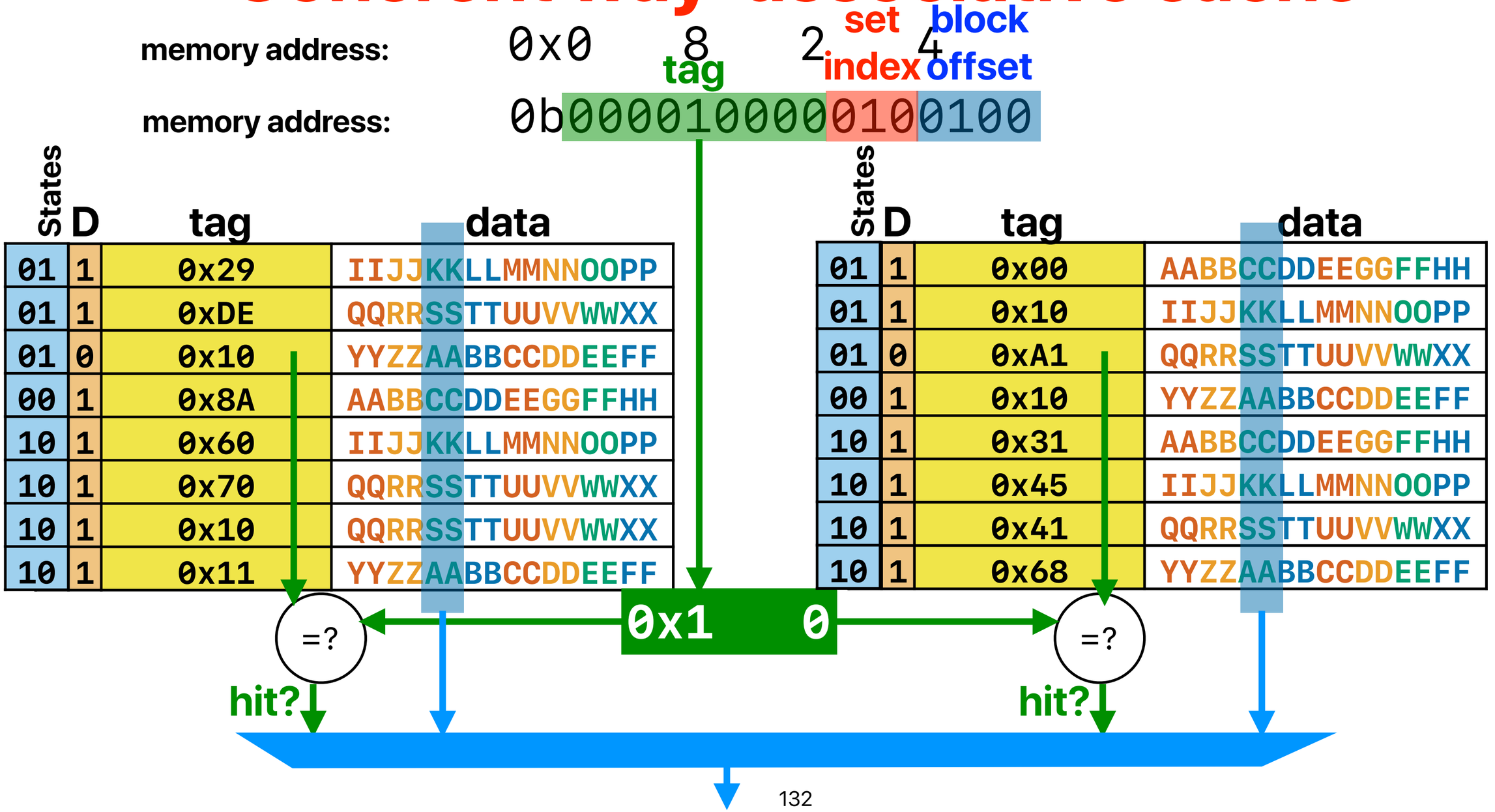
What software thinks about “multiprogramming” hardware



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

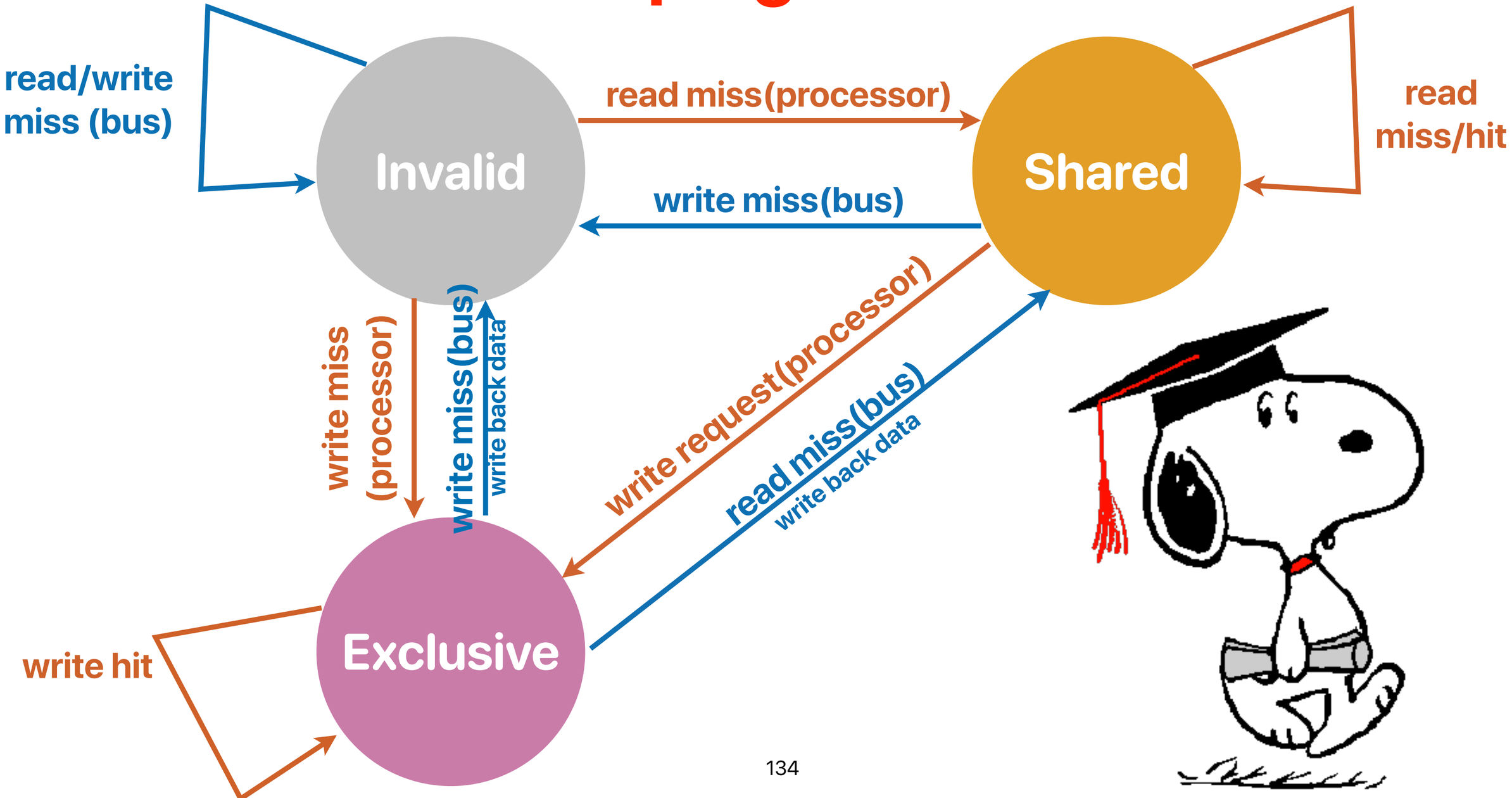
Coherent way-associative cache



Simple cache coherency protocol

- Snooping protocol
  - Each processor broadcasts / listens to cache misses
- State associate with each block (cacheline)
  - Invalid
    - The data in the current block is invalid
  - Shared
    - The processor can read the data
    - The data may also exist on other processors
  - Exclusive
    - The processor has full permission on the data
    - The processor is the only one that has up-to-date data

Snooping Protocol



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.

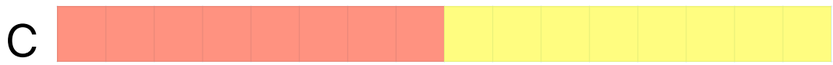
L v.s. R

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;
}
```



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);
```

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?

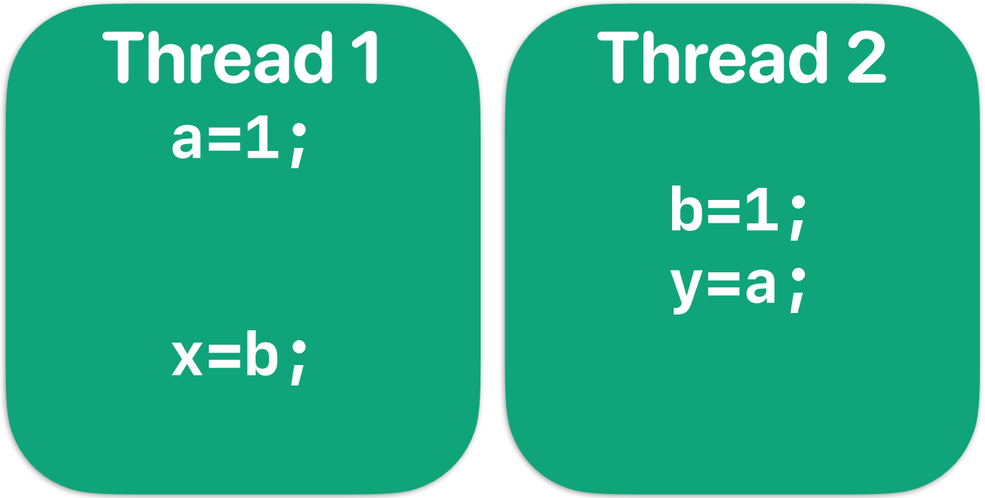
① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

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

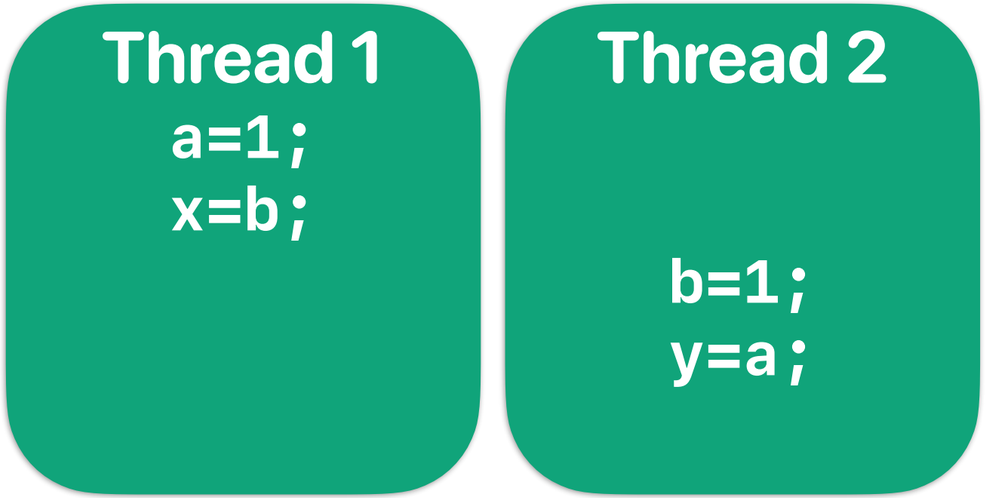
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

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;
}
```

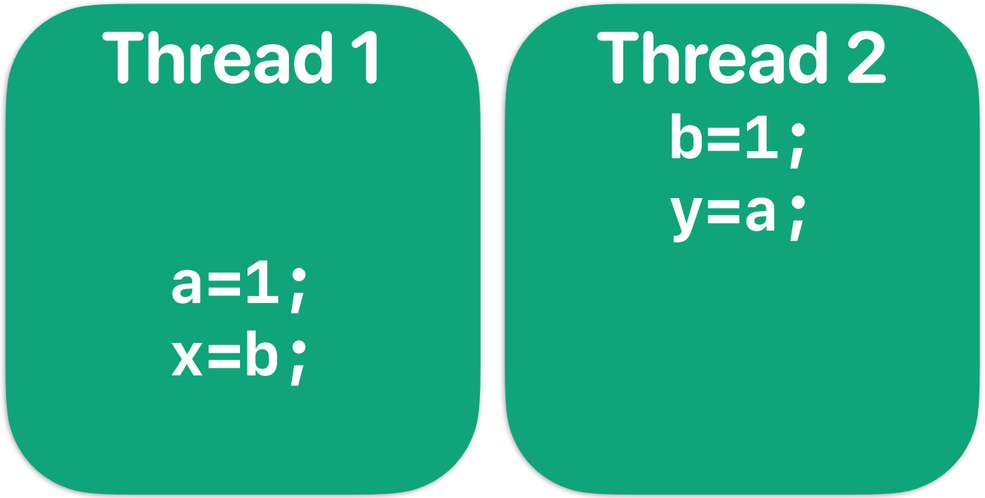
Possible scenarios



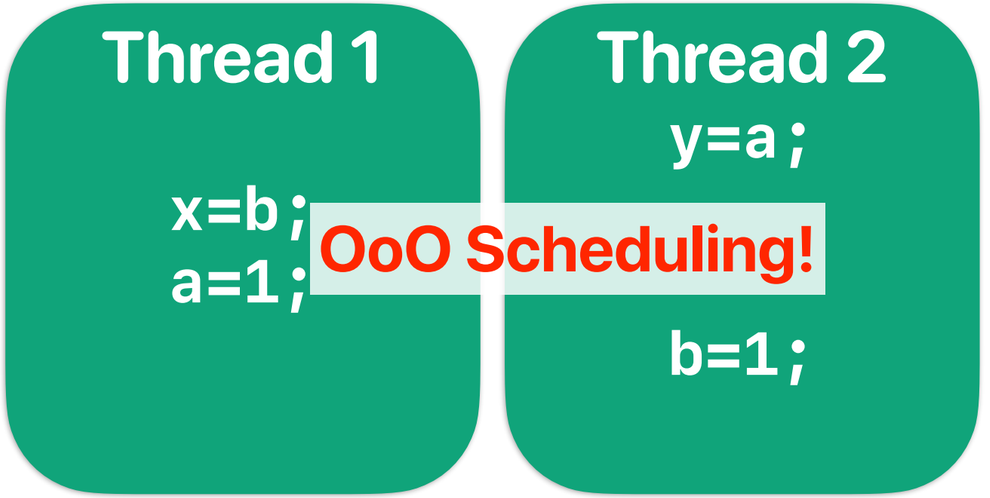
(1,1)



(0,1)



(1,0)



(0,0)

fence instructions

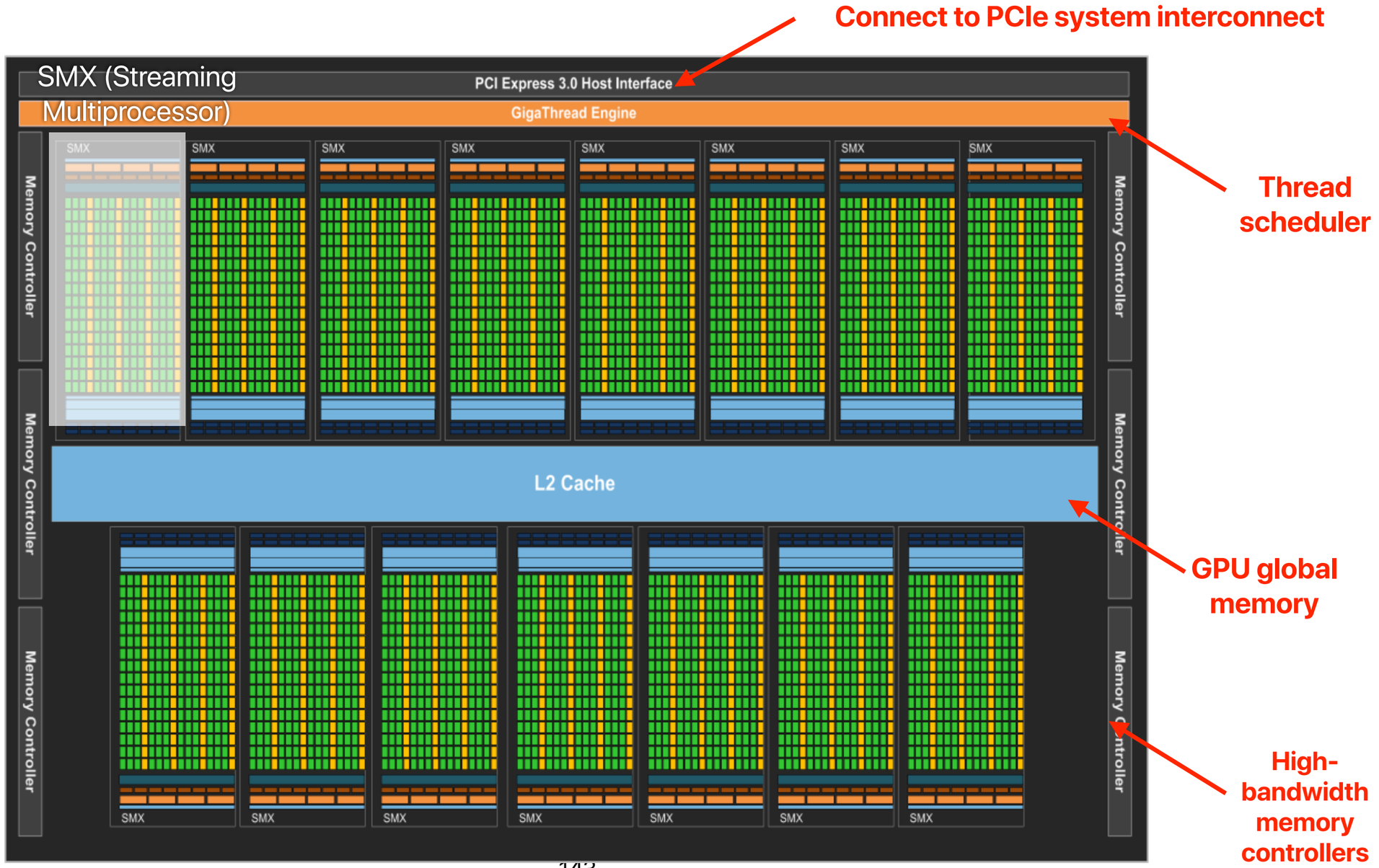
- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

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

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

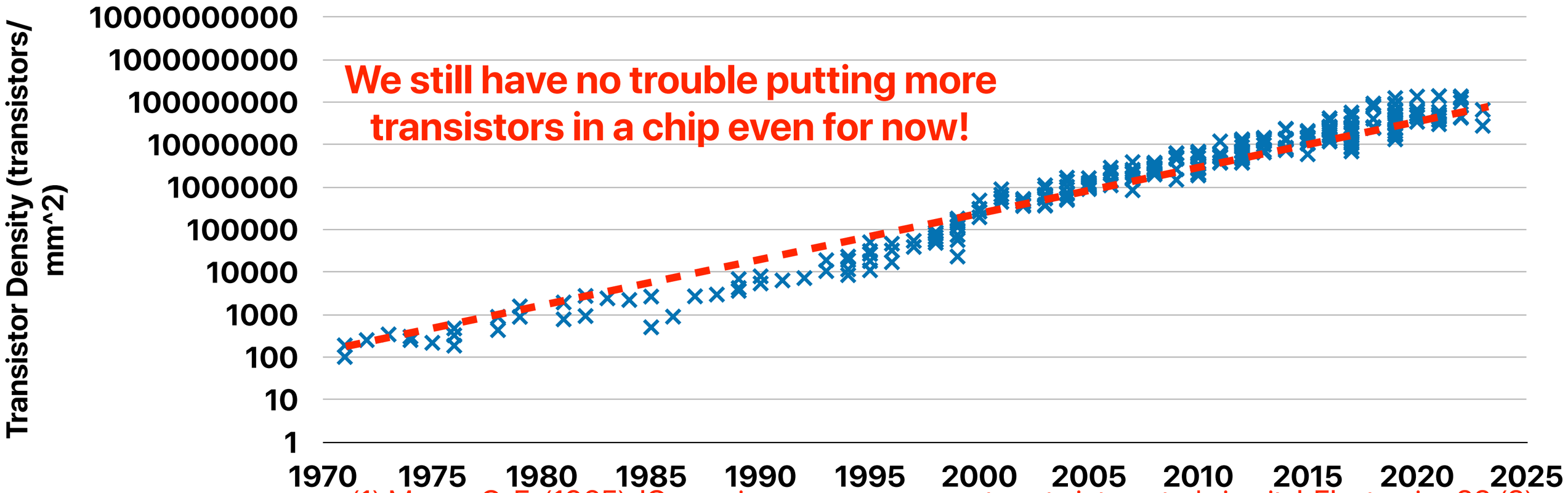
In terms of latency, even for 120 frames, you still have 8ms latency to get everything done!

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

Power/Energy/Dark Silicon

Moore's Law[1]

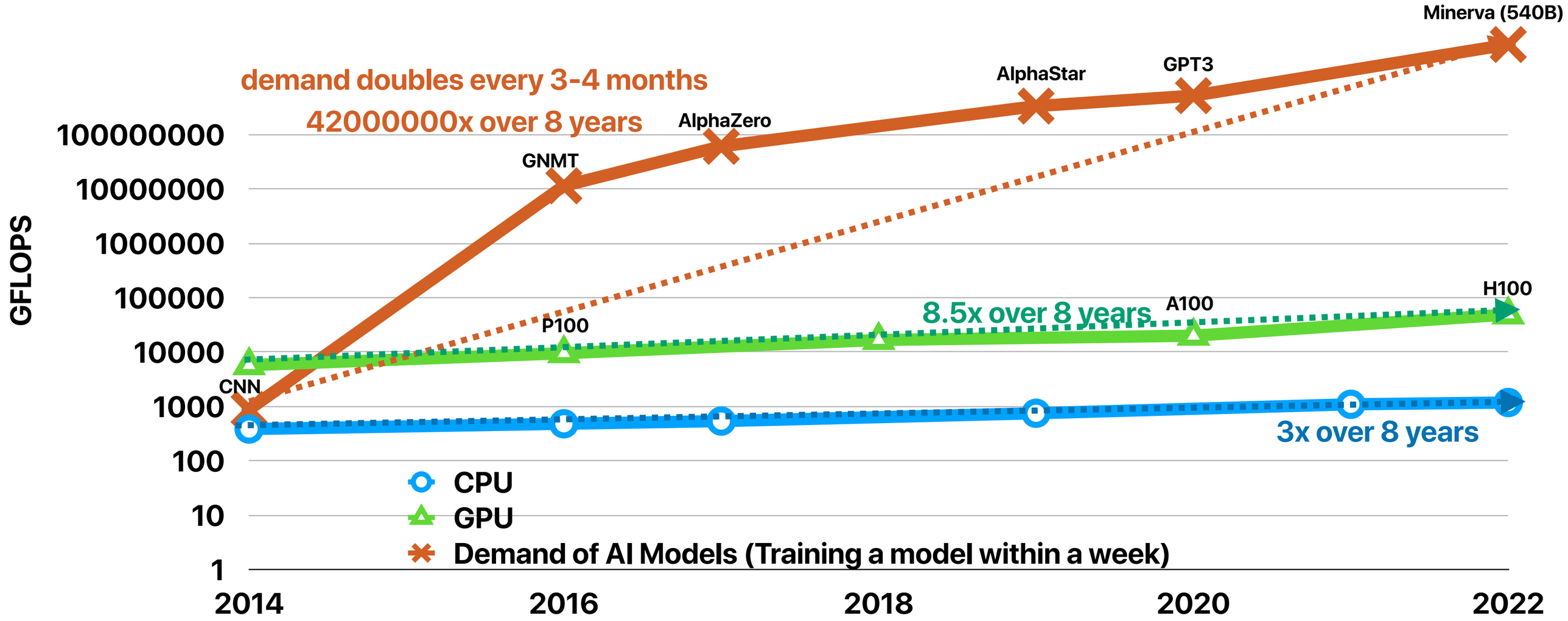
- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains



(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

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

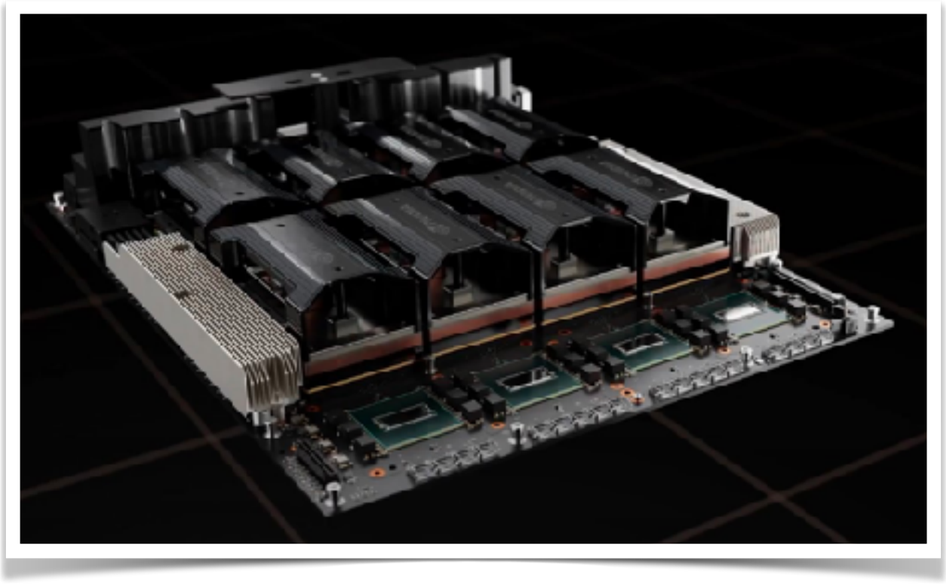


https://ourworldindata.org/grapher/artificial-intelligence-training-computation

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

Dynamic/Active Power

- The power consumption due to the switching of transistor states
- Dynamic power per transistor

  $P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$

  - α: average switches per cycle
  - C: capacitance
  - V: voltage
  - f: frequency, usually linear with V
  - N: the number of transistors

Static/Leakage Power

- The power consumption due to leakage — transistors do not turn all the way off during no operation
- Becomes the **dominant** factor in the most advanced process technologies.

$$P_{leakage} \sim N \times V \times e^{-V_t}$$

  - N: number of transistors
  - V: voltage
  - V_t: threshold voltage where transistor conducts (begins to switch)

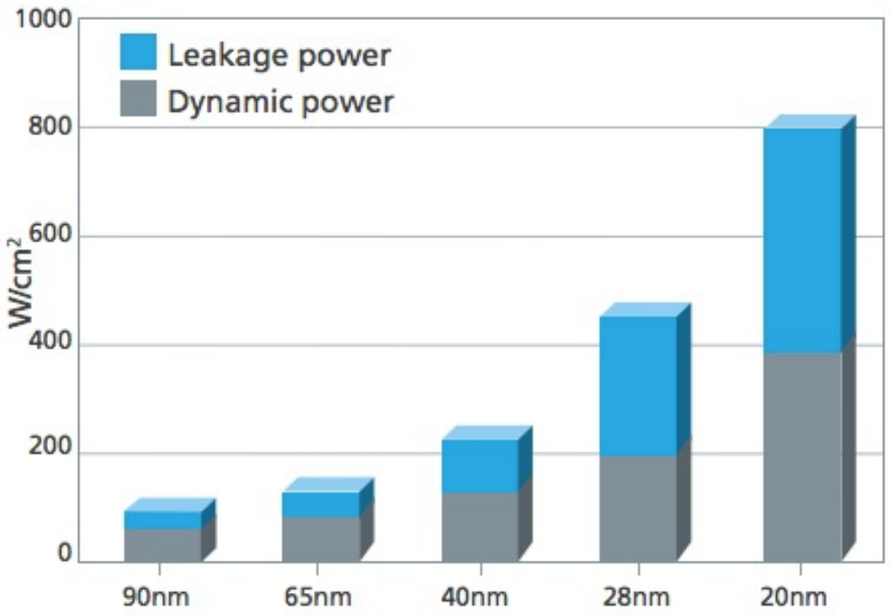


Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).

What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if we power the chip with the same power consumption but put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  - ① The power consumption per chip will increase
  - ② The power density of the chip will increase
  - ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  - ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Power consumption to light on all transistors

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Dennardian Scaling

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Dennardian Broken

Chip

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

On ~ 50W

Off ~ 0W

Dark!

=100W!

Or we have to dim down all transistors

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Dennardian Scaling

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Dennardian Broken

Chip

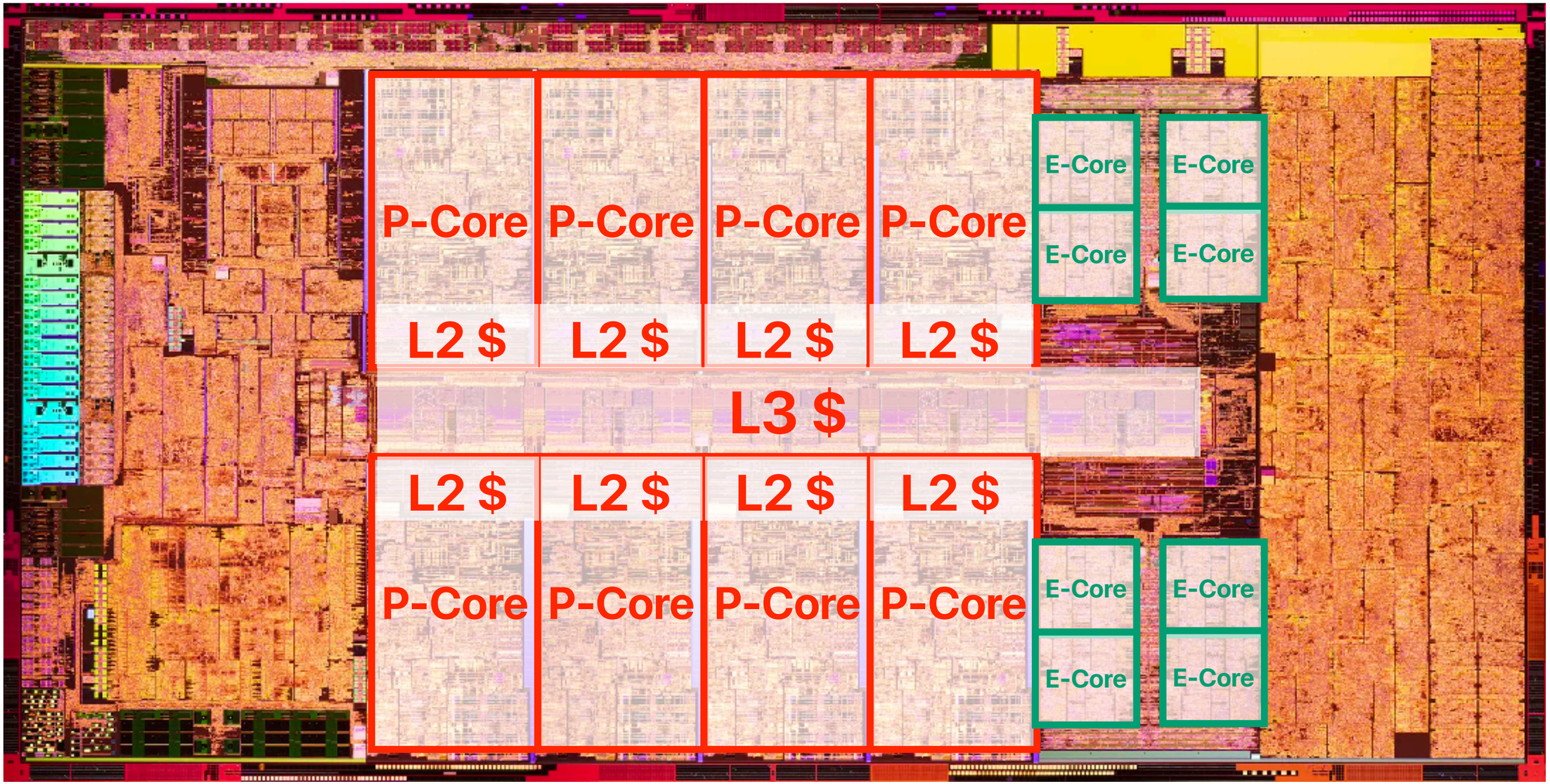
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Low frequency

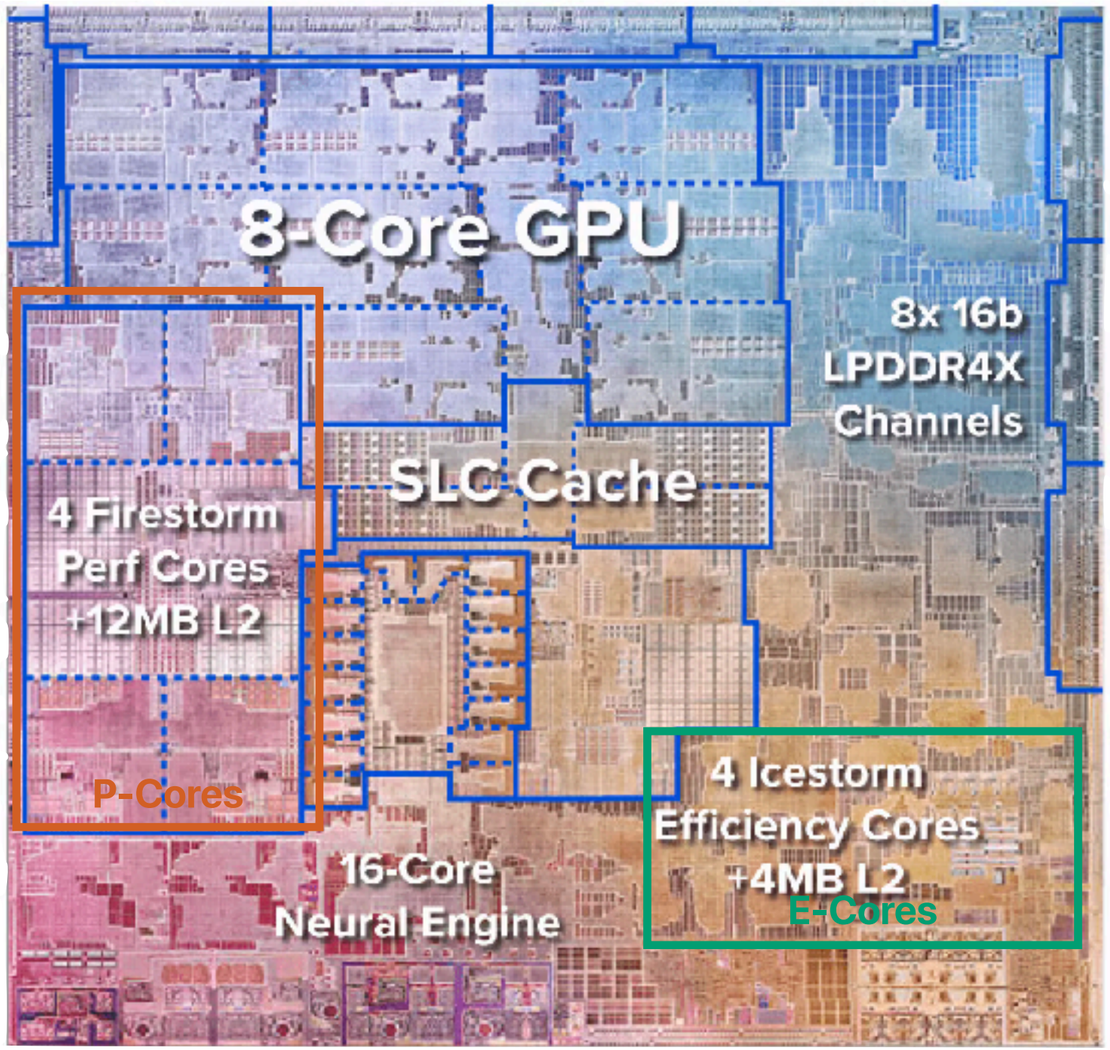
~ 0.5W

=50W!

Intel Alder Lake



Single ISA heterogeneous CMP in Apple's M1



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

Inside an SM

A total of 16*4 = 64 FP32 cores
A total of 16*4 = 64 INT32 cores
A total of 16*4 = 16 FP64 cores

- All of these can only perform the same operation at the same time, but each of these is named as a “thread” in CUDA
- You can only use either FP32, FP64, INT32 or “Tensor Cores” at the same time


Thank you!
CSE142 Summer 2023!

Computer
Science &
Engineering

142

ありがとう