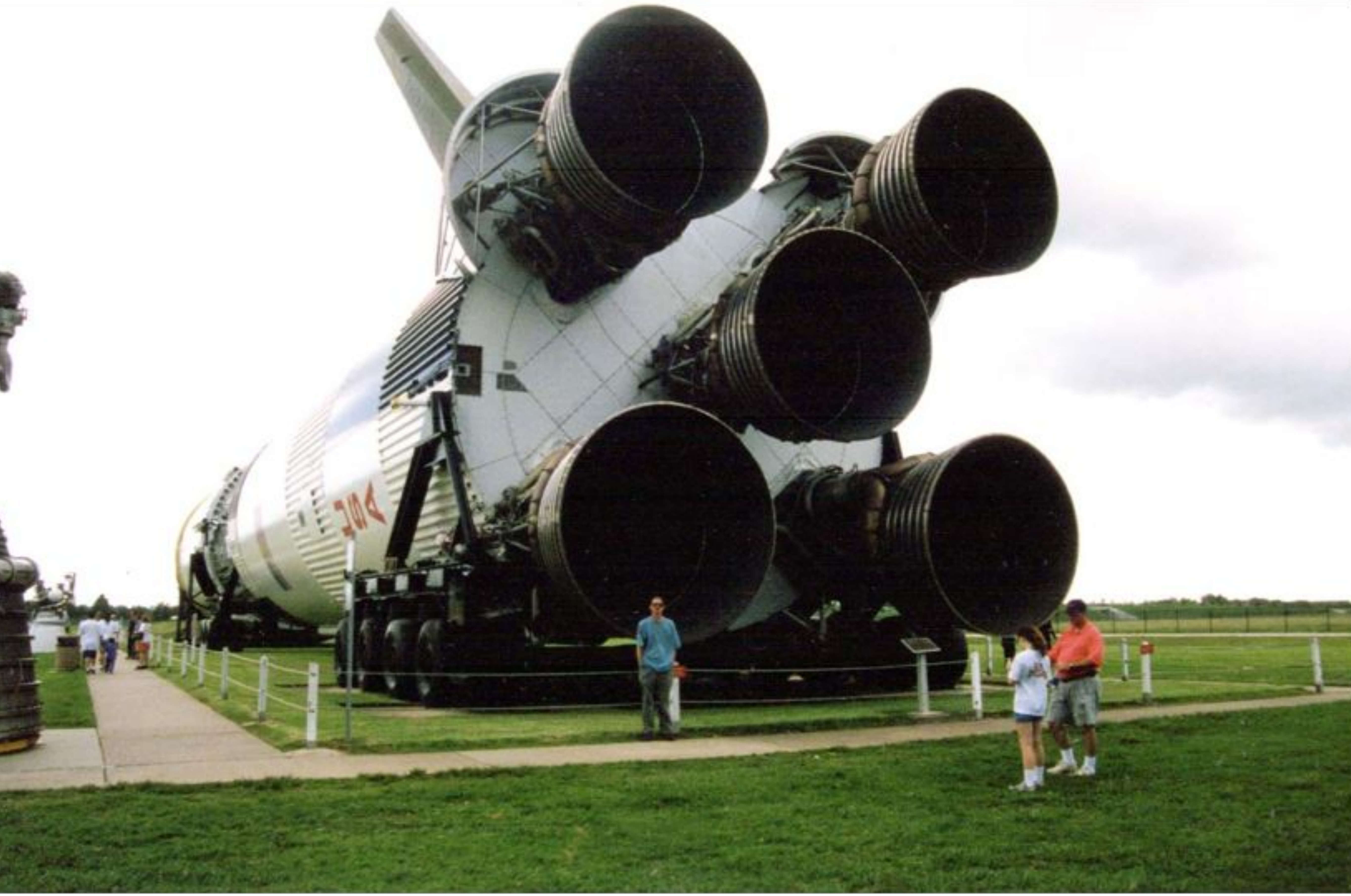
Performance

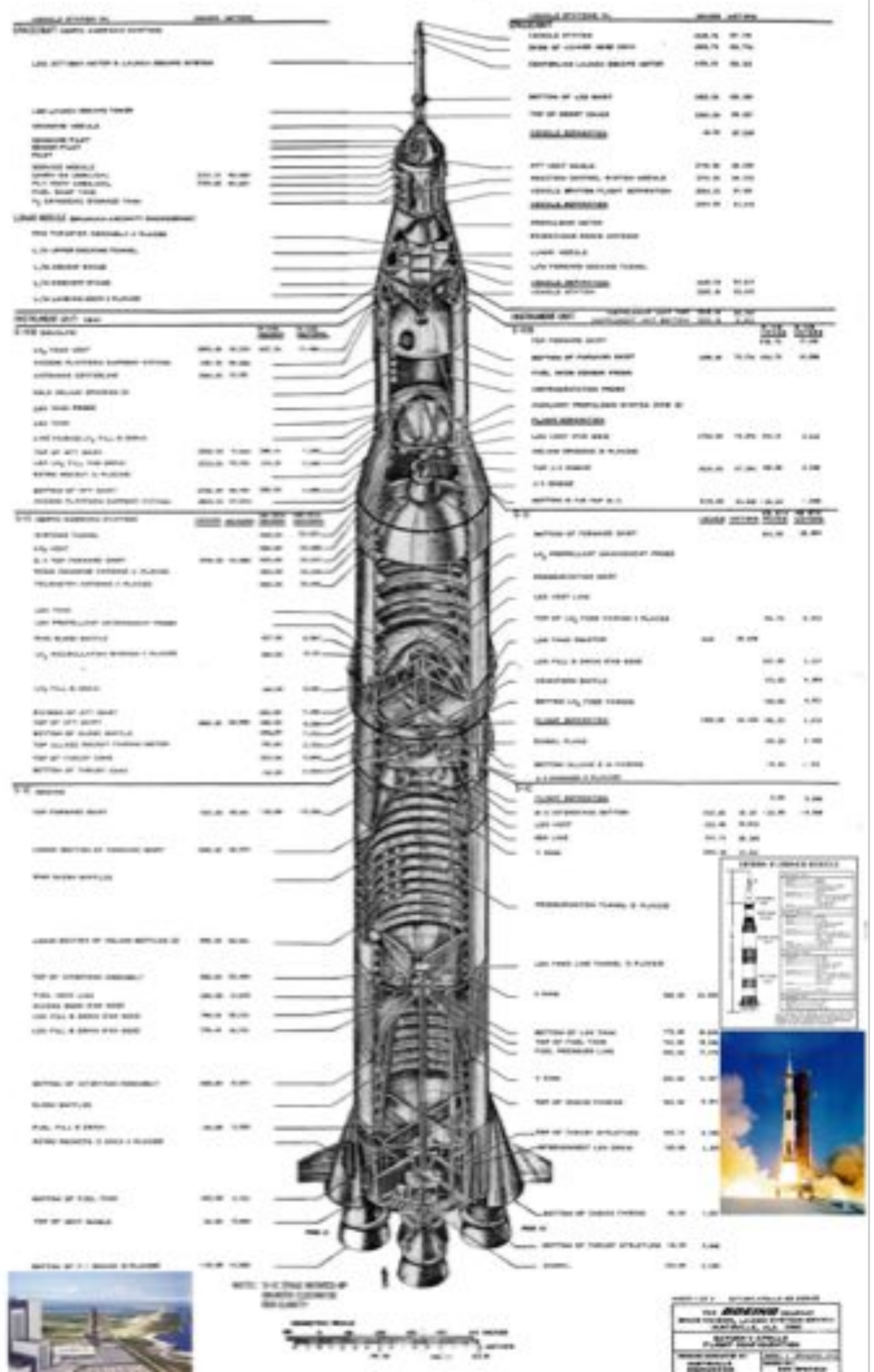
Admin:

Lab this week: discuss projects.

Ordering parts: keep all receipts! If paying with a credit card, you must have a receipt that shows at least the last four digits of the card.

If you need a particular part, ask and we can either order it for you or see if we have it already (e.g., ADC and DAC chips).


USA



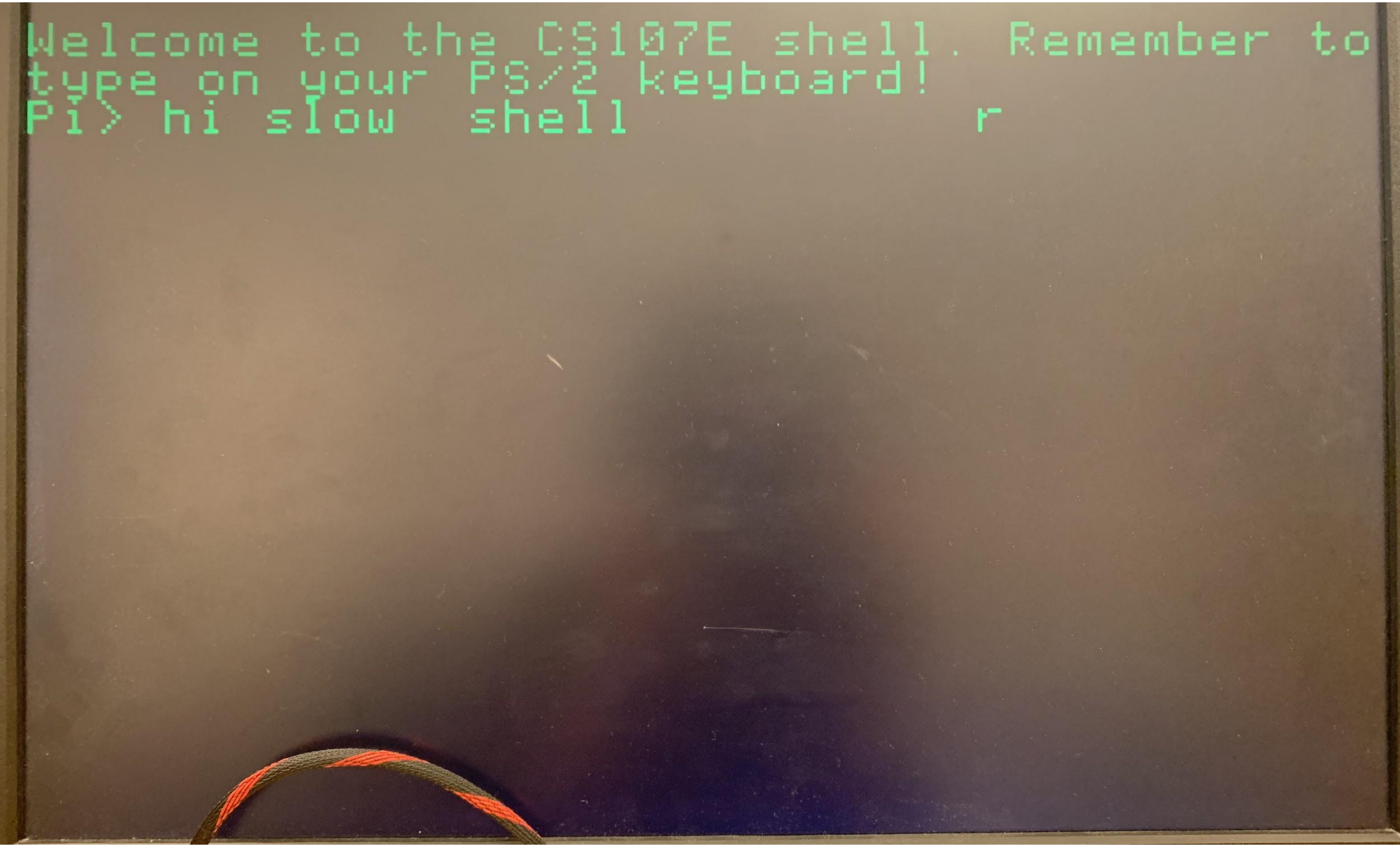
Command Module
64,000 lbs

Saturn V
6,200,000 lbs

Payload
1.5% of total weight

Falcon 9
Elon Musk

Why is performance optimization important?



```
Welcome to the CS107E shell. Remember to
type on your PS/2 keyboard!
Pi> hi slow  shell              r
```

User experience matters

BIG DATA
AI
11001

Demo: clear

SW Optimization Strategies

- **Compiler flags** - leverage automatic optimizations
  - **`gcc -O0 [source files] [object files] -o output file`**
  - Common options: -O0,-O1, -O2, -O3
  - Other options: -Ofast, -Og, -Os

    Ofast May produce inaccurate math results

```
draw_pixel:
    8010:   f8 40 2d e9     push    {r3, r4, r5, r6,
r7, lr}
    8014:   00 40 a0 e1     mov r4, r0
    8018:   01 50 a0 e1     mov r5, r1
    801c:   02 70 a0 e1     mov r7, r2
    8020:   c9 01 00 eb     bl  #1828 <fb_get_width>
    8024:   00 60 a0 e1     mov r6, r0
    8028:   c7 01 00 eb     bl  #1820 <fb_get_width>
    802c:   28 30 9f e5     ldr r3, [pc, #40]
    8030:   00 30 93 e5     ldr r3, [r3]
    8034:   97  <unknown>
    8035:   06 02 e0 02     rsceq   r0, r0,


    …
    8050:   03 20 c3 e5     strb    r2, [r3, #3]
    8054:   f8 40 bd e8     pop {r3, r4, r5, r6, r7,
lr}
```

```
draw_pixel:
    8010:   f8 40 2d e9     push    {r3, r4, r5, r6,
r7, lr}
    8014:   00 40 a0 e1     mov r4, r0
    8018:   01 50 a0 e1     mov r5, r1
    801c:   02 70 a0 e1     mov r7, r2
    8020:   c9 01 00 eb     bl  #1828 <fb_get_width>
    8024:   00 60 a0 e1     mov r6, r0
    8028:   c7 01 00 eb     bl  #1820 <fb_get_width>
    802c:   28 30 9f e5     ldr r3, [pc, #40]
    8030:   00 30 93 e5     ldr r3, [r3]
    8034:   97  <unknown>
    8035:   06 02 e0 02     rsceq   r0, r0,

    …
    8050:   03 20 c3 e5     strb    r2, [r3, #3]
    8054:   f8 40 bd e8     pop {r3, r4, r5, r6, r7,
lr}
```

Overhead for creating stack frame, saving regs

```
draw_pixel:
    8010:   f8 40 2d e9     push    {r3, r4, r5, r6,
r7, lr}
    8014:   00 40 a0 e1     mov r4, r0
    8018:   01 50 a0 e1     mov r5, r1
    801c:   02 70 a0 e1     mov r7, r2
    8020:   c9 01 00 eb     bl  #1828 <fb_get_width>
    8024:   00 60 a0 e1     mov r6, r0
    8028:   c7 01 00 eb     bl  #1820 <fb_get_width>
    802c:   28 30 9f e5     ldr r3, [pc, #40]
    8030:   00 30 93 e5     ldr r3, [r3]
    8034:   97  <unknown>
    8035:   06 02 e0 02     rsceq   r0, r0,

    …
    8050:   03 20 c3 e5     strb    r2, [r3, #3]
    8054:   f8 40 bd e8     pop {r3, r4, r5, r6, r7,
lr}
```

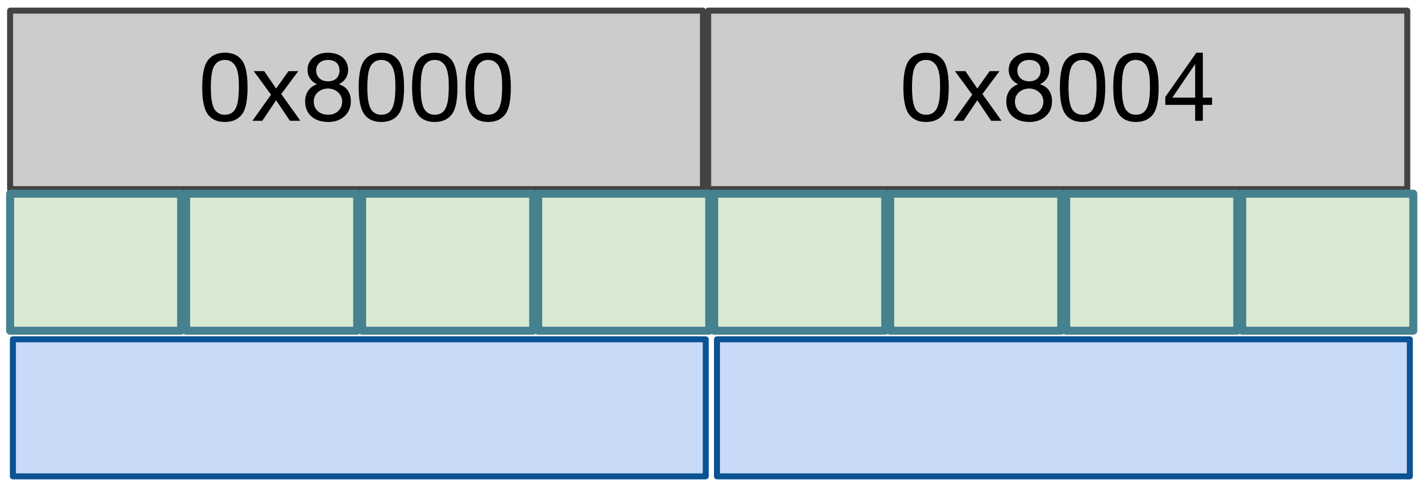
Overhead to call functions

SW Optimization Strategies

- **Compiler flags**
- **Inlining/Remove repetitive function calls**
  - Reduce function calls overhead
  - Can use the `inline` keyword to inline short helper function

SW Optimization Strategies

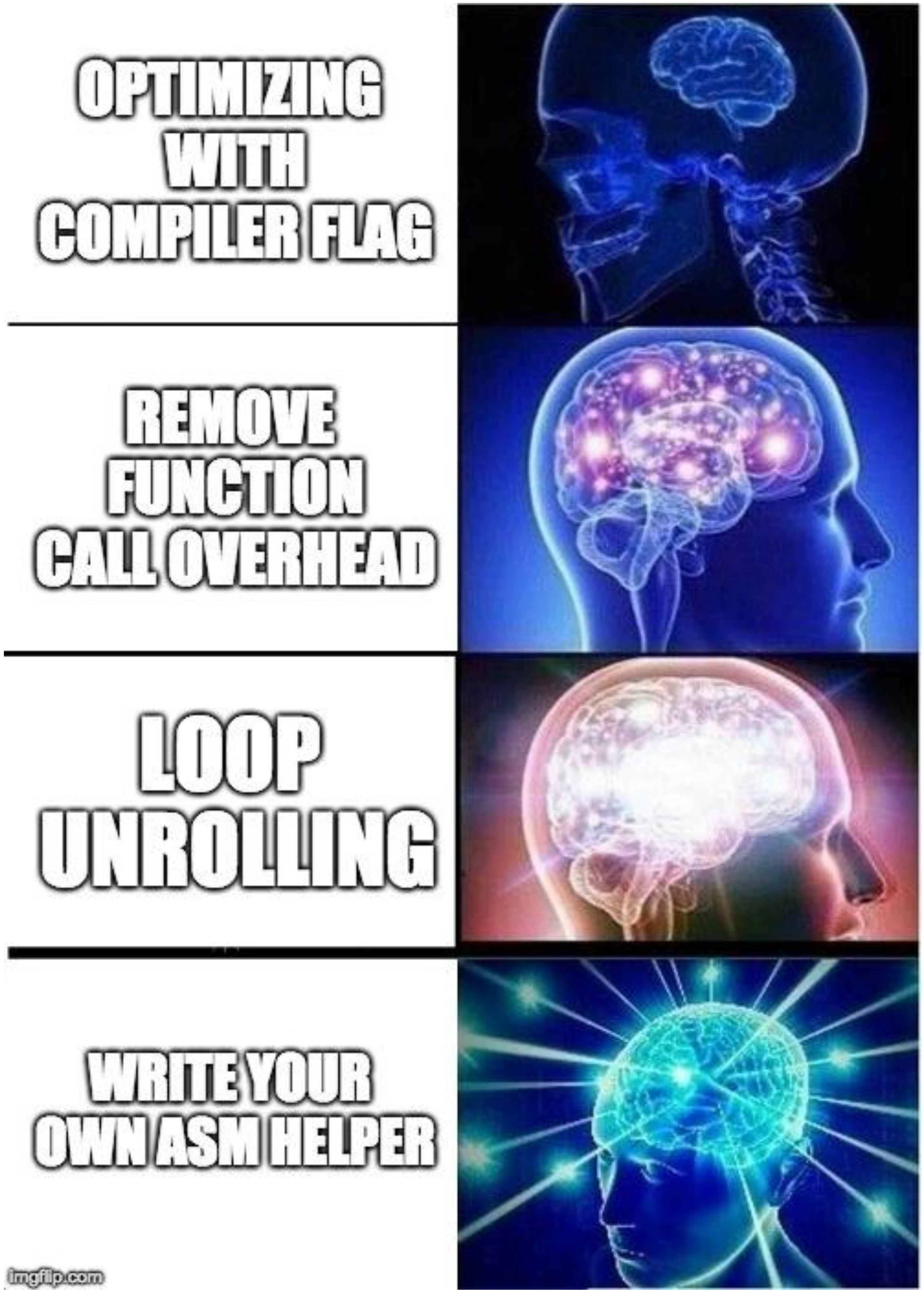
- **Compiler flags**
- **Inlining/Remove repetitive function calls**
- **Aggregate loads/stores**
  - can load/store a word once instead of 4 bytes



8 ldr/str
2 ldr/str

SW Optimization Strategies

- **Compiler flags**
- **Inlining/Remove repetitive function calls**
- **Aggregate loads/stores**
- **Loop optimization**
  - Code Hoisting: remove loop-invariant ops from loop body
  - Flatten nested loop
  - Loop unrolling: combine operations from multiple iteration

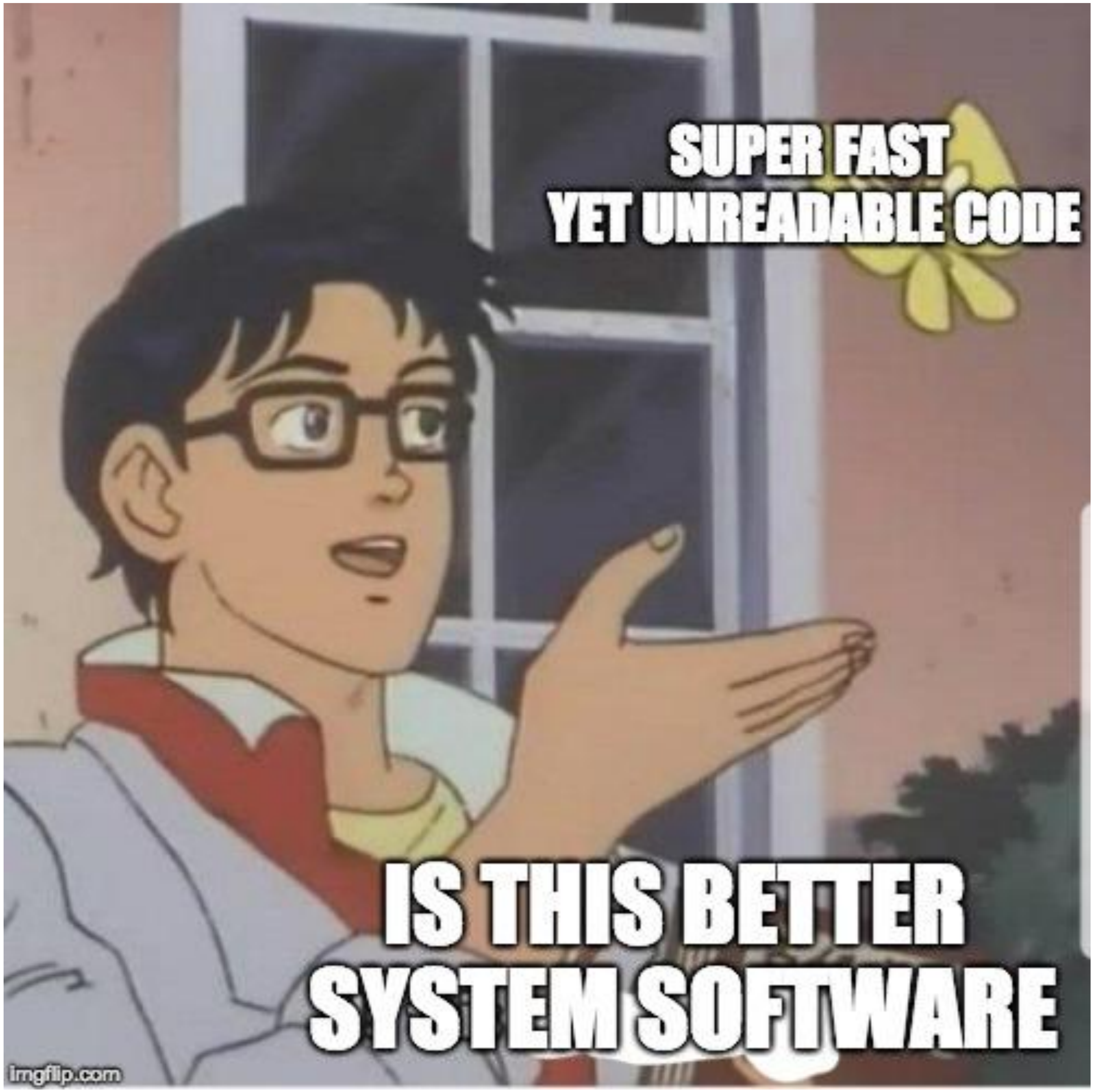

OPTIMIZING WITH COMPILER FLAG
REMOVE FUNCTION CALL OVERHEAD
LOOP UNROLLING
WRITE YOUR OWN ASM HELPER
imgflip.com

SW Optimization Strategies

- **Compiler flags**
- **Inlining/Remove repetitive function calls**
- **Aggregate loads/stores**
- **Loop optimization**
- **Manual Assembly**

SW Optimization Strategies

- **Compiler flags** - leverage automatic optimizations
- **Inlining** - Reduce function call overhead
- **Aggregate loads/stores** - less instructions per memory operation
- **Loop optimizations** - code hoisting, combination, unrolling
- **Manual assembly** - Be smarter than the compiler


SUPER FAST
YET UNREADABLE CODE
IS THIS BETTER
SYSTEM SOFTWARE
imgflip.com

When to optimize?

Amdahl's Law

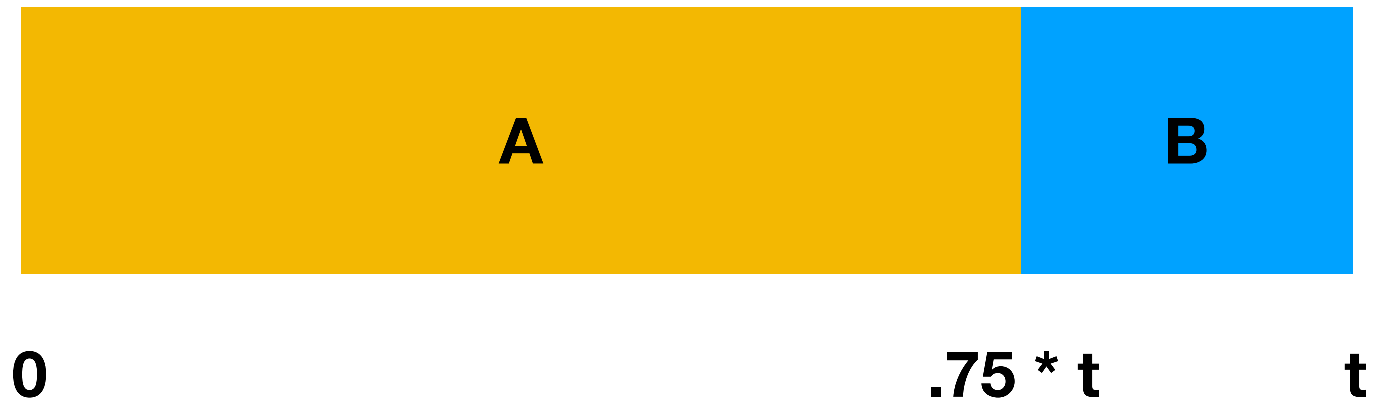
$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

- $S_{latency}$ - the theoretical speedup of the execution of the entire program
- s - the speedup of the part of the program you're optimizing
- p - the proportion of execution time that the part the program you're optimizing originally occupied

Amdahl’s Law

Suppose your original program took **t** cycles to execute

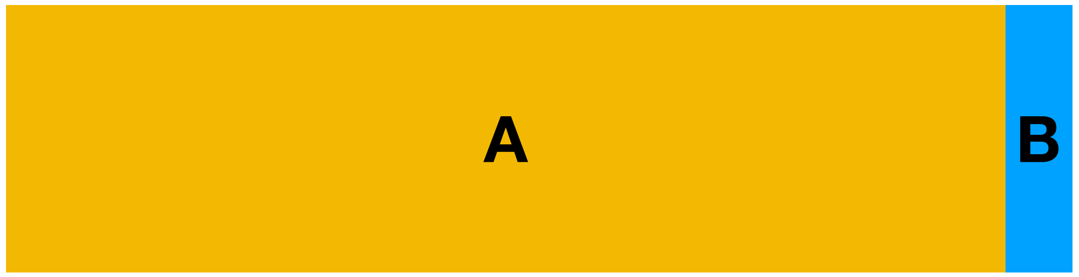
Amdahl's Law



The program is divided into two distinct portions:

- Part A takes 75% of the time
- Part B takes 25% of the time

Amdahl's Law



If we optimize part B to make it 5 times faster, this only reduces the overall computation time slightly

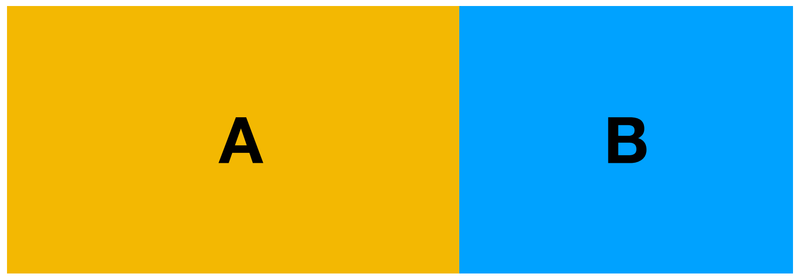
Amdahl's Law

Part B is 25% of the overall program (*p = .25*) and we speed it up by a factor of 5 (*s = 5)*

$$S_{latency} = \frac{1}{1 - .25 + \frac{.25}{5}} = 1.25$$

Overall program speedup is 1.25

Amdahl's Law



If we optimize part A to make it just twice as fast, we get a greater overall speedup

Amdahl's Law

Part A is 75% of the overall program (*p = .75*) and we speed it up by a factor of 2 (*s = 2)*

$$S_{latency} = \frac{1}{1 - .75 + \frac{.75}{2}} = 1.60$$

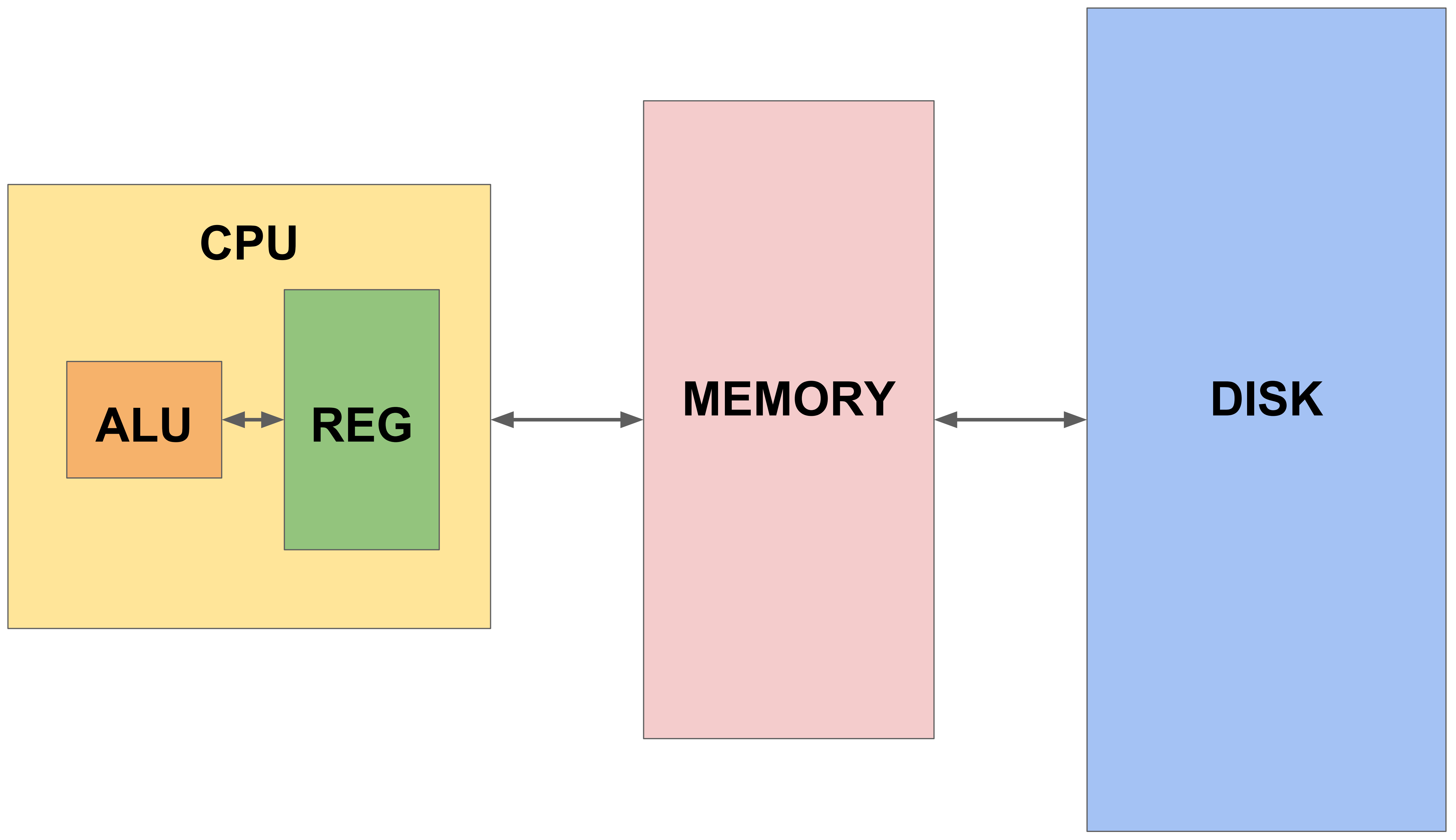
Overall program speedup is 1.60

Profiling

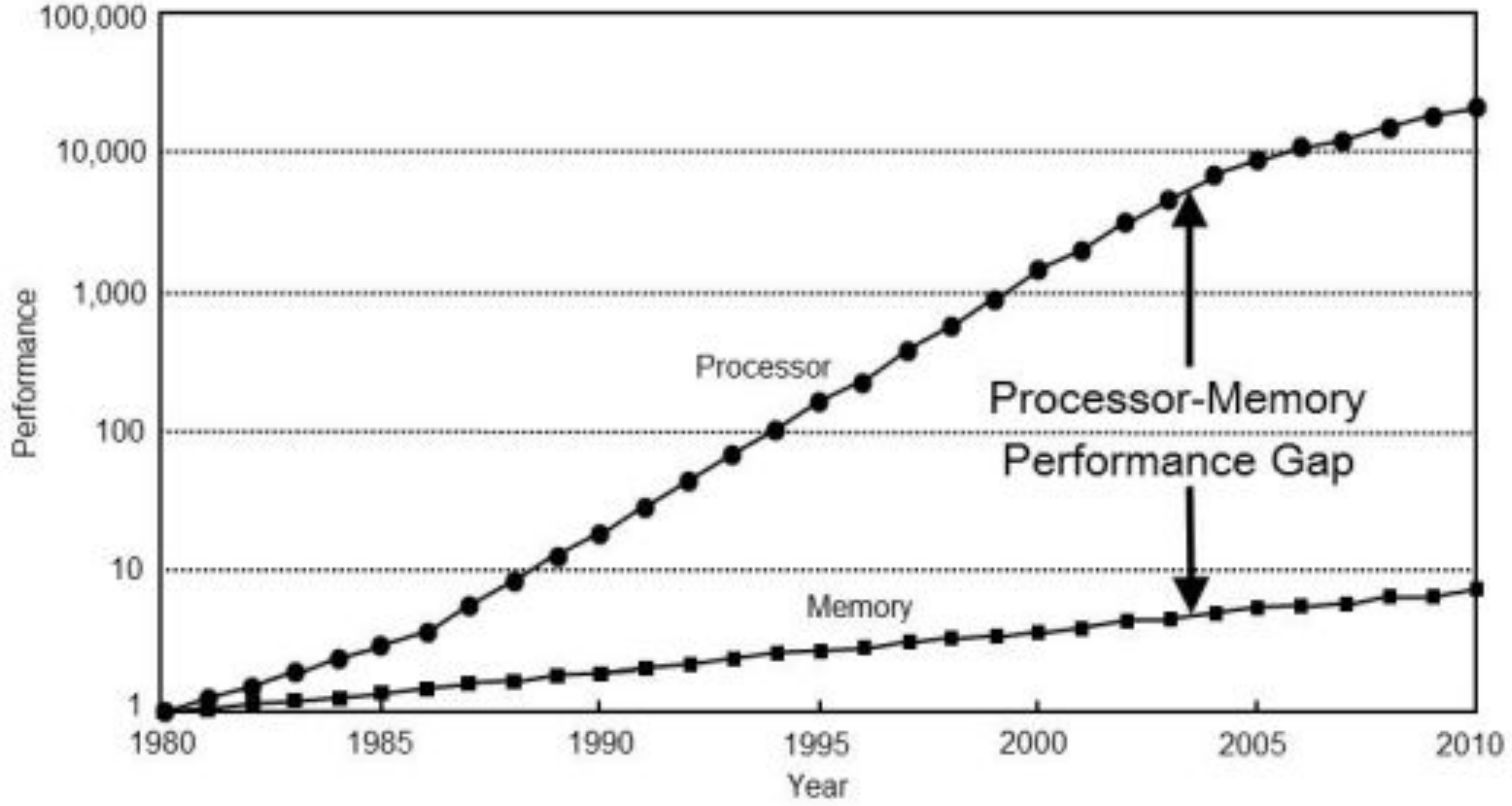
- Analyze your program at runtime to measure characteristics of interest
  - Space/time complexity
  - Frequency of certain instructions
  - Frequency and Duration of Function Calls
- Useful for guiding optimization or debug

Demo: stackprof

Beyond Software Optimization

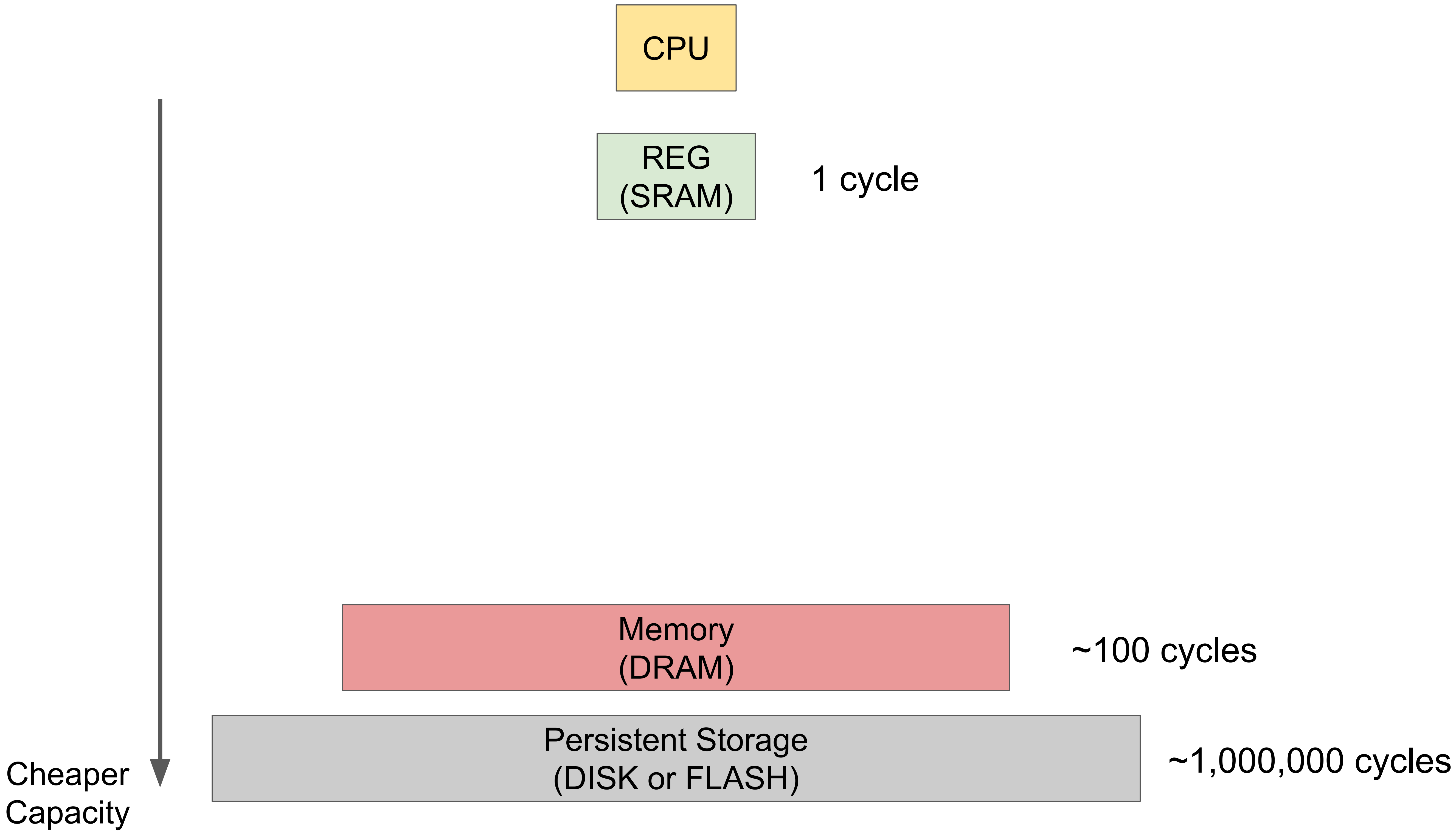

CPU
ALU
REG
MEMORY
DISK

Processor-Memory Performance Gap



[Hennessy, J.L.; Patterson, D.A. Computer Organization and Design, 2nd ed.]

Memory Hierarchy



```
// [r0]++;
ldr  r1, [r0]
add  r1,  #1
str  r1,   [r0]
```

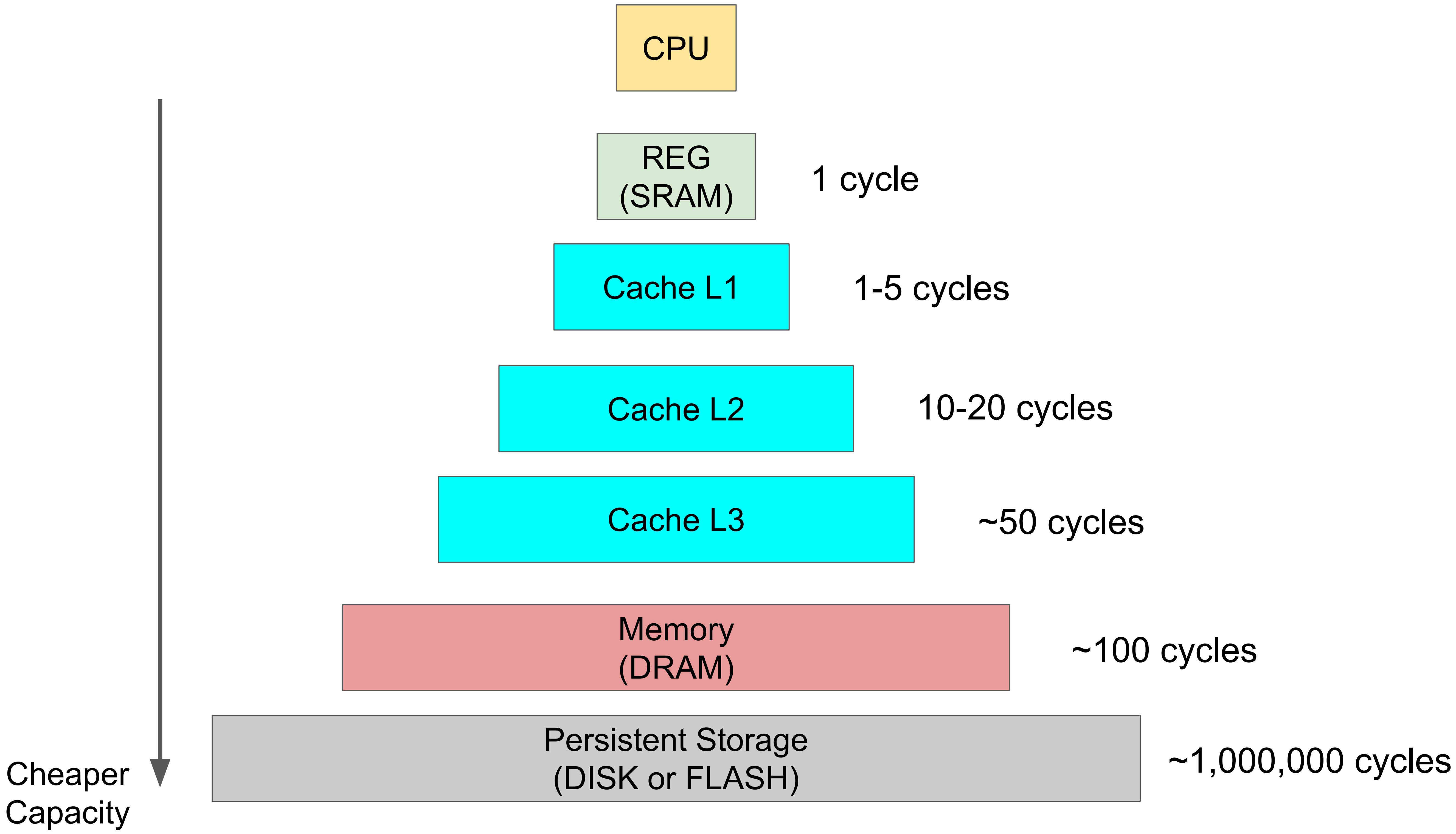
Wait for 100 clocks?!

Me: optimizes assembly code

Memory: stalls

Me: for 100 cycles

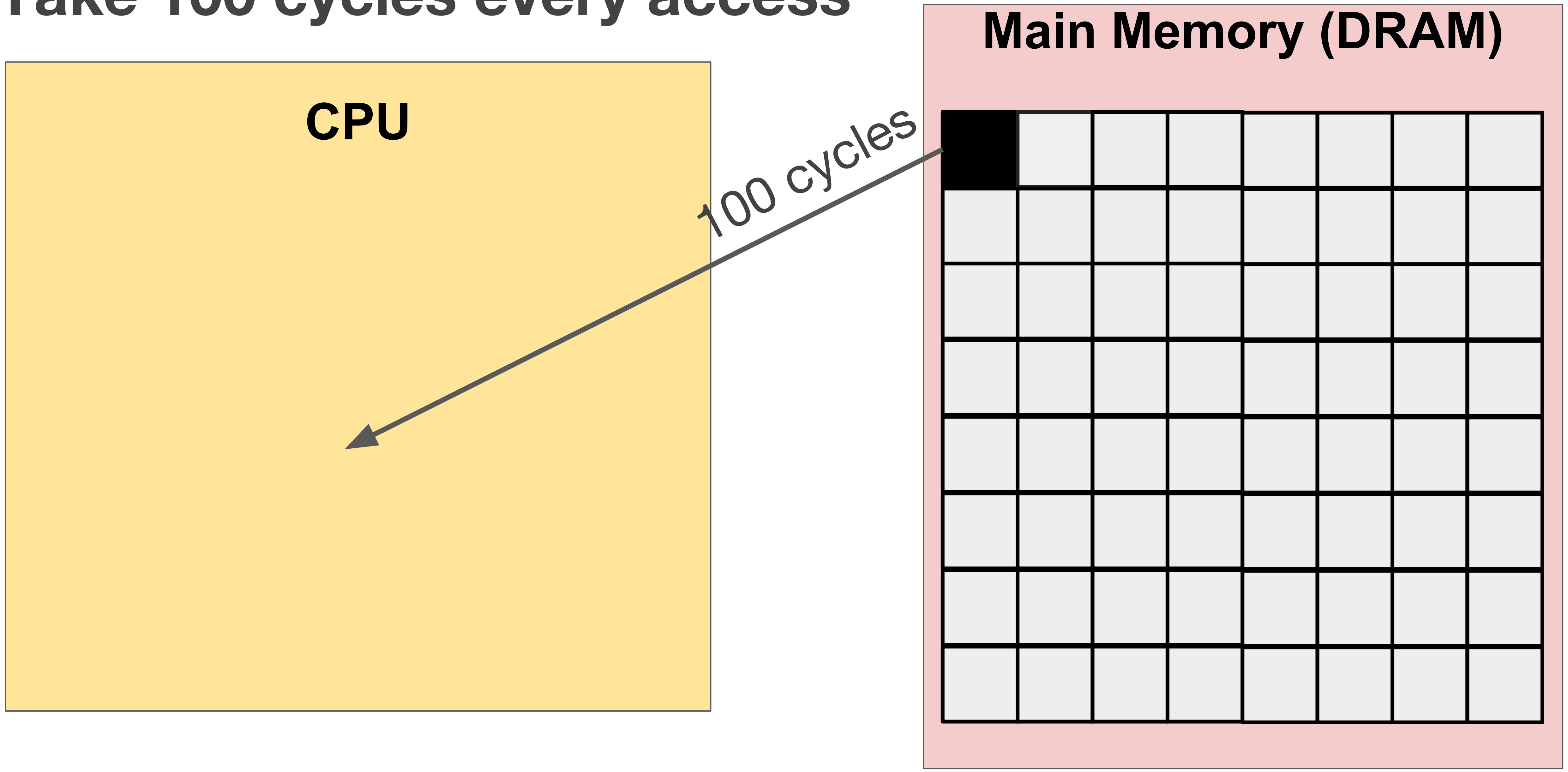
Memory Hierarchy



w/o caches

CPU wants to access memory location in black.

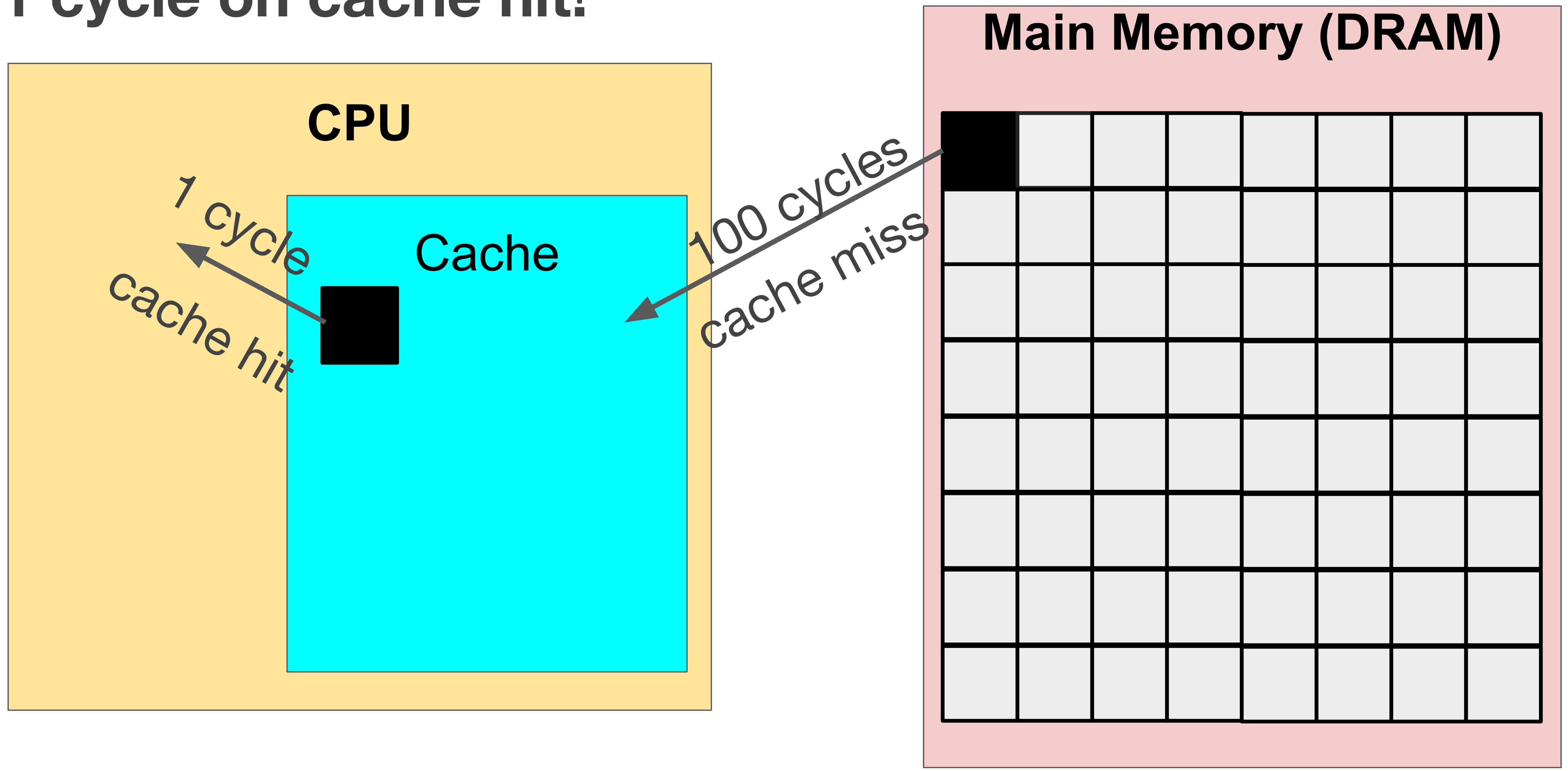
Take 100 cycles every access



Caches

100 cycles on cache miss (main memory access).

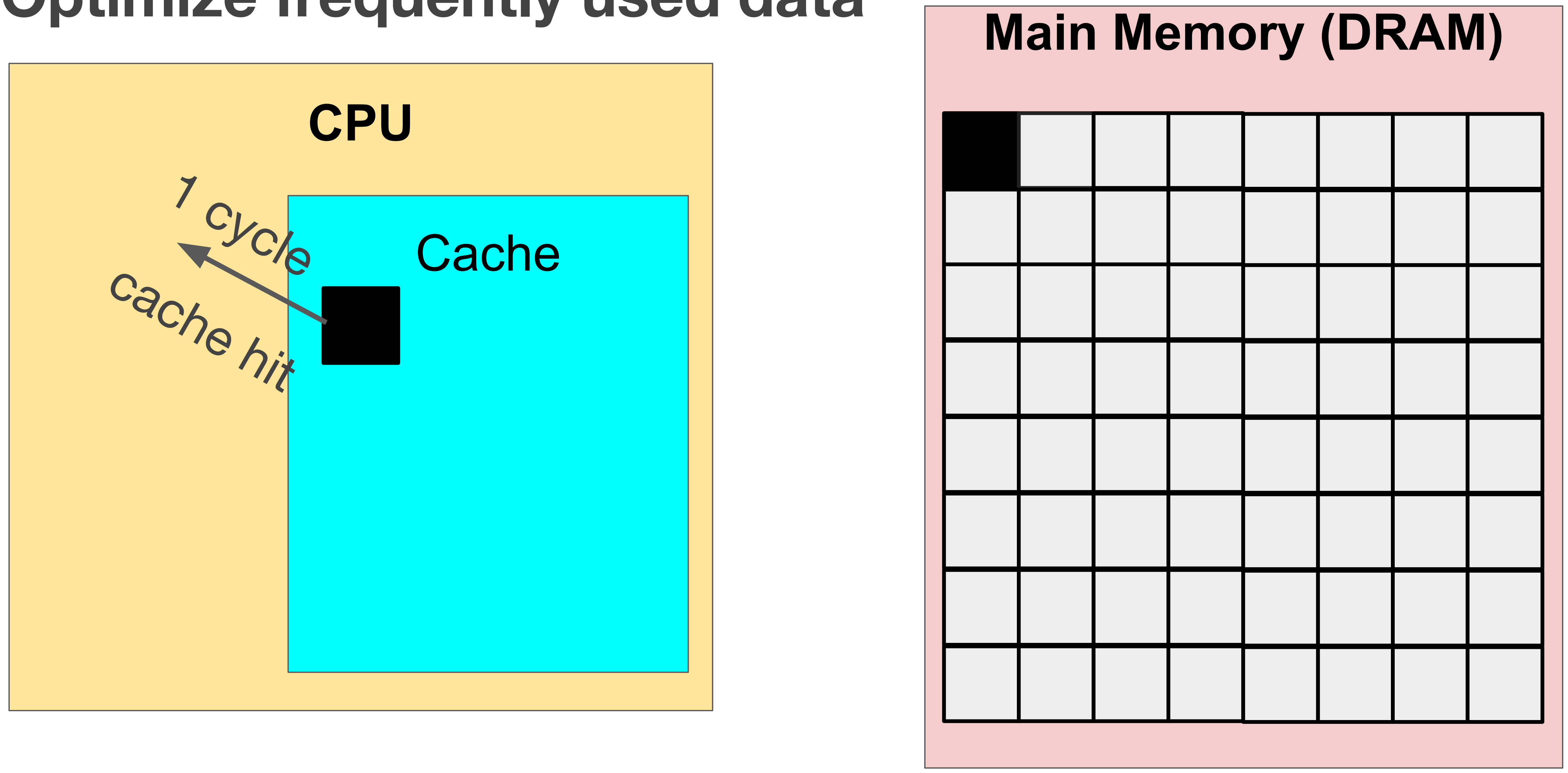
1 cycle on cache hit!



Caches

Concept 1: *Temporal Locality*

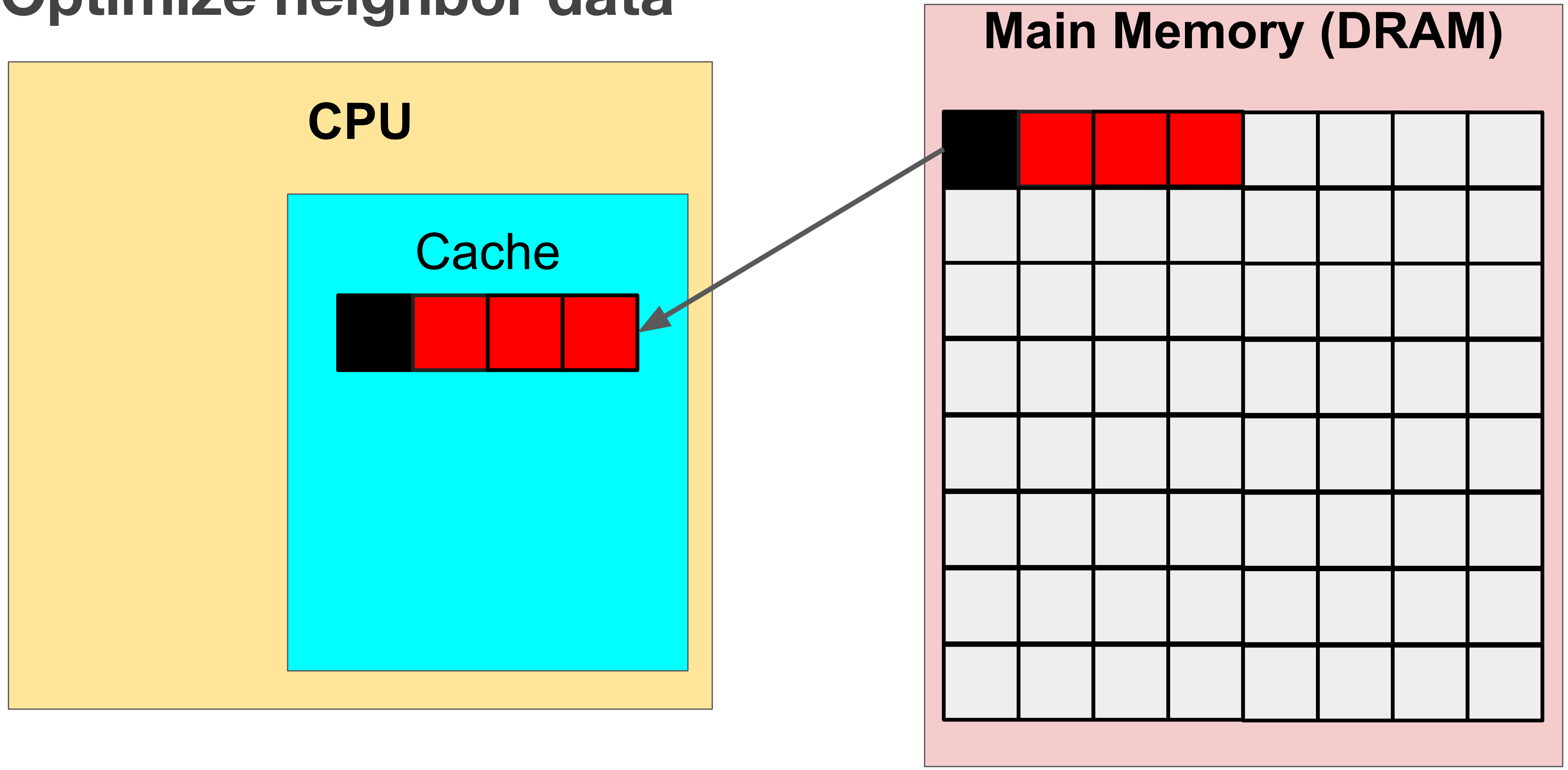
Optimize frequently used data



Caches

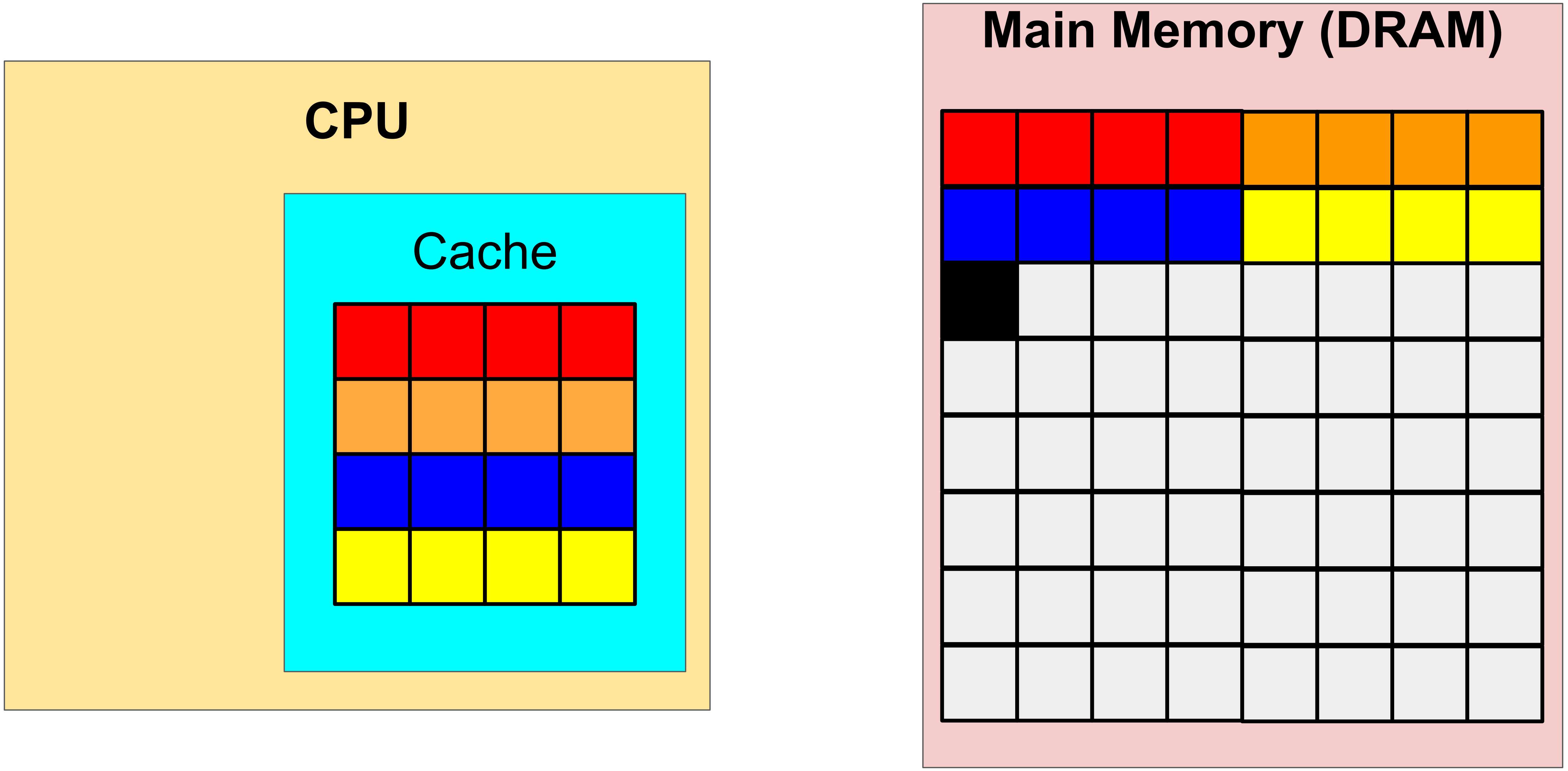
Concept 2: *Spatial Locality*

Optimize neighbor data



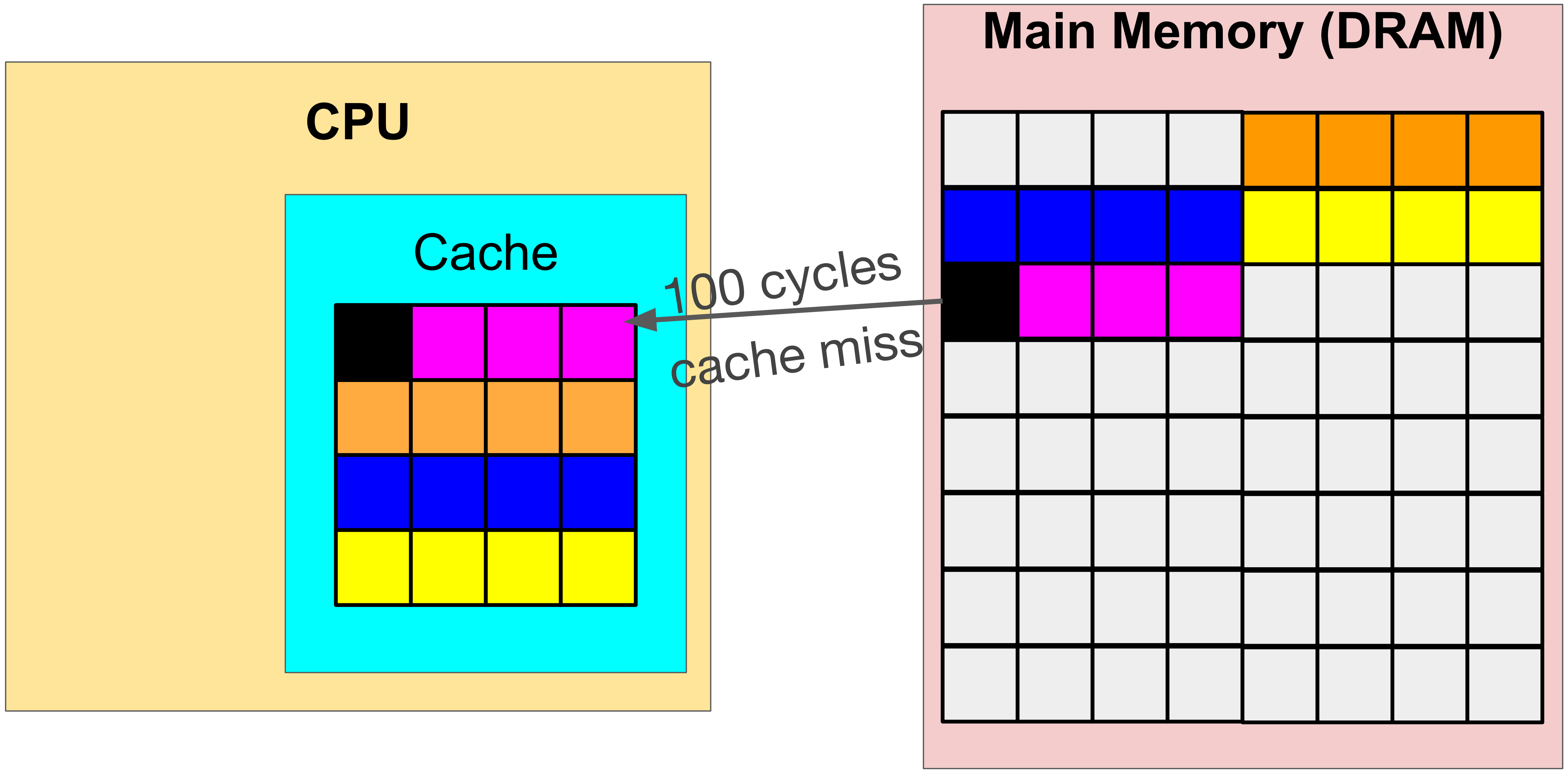
Caches

Limited cache size



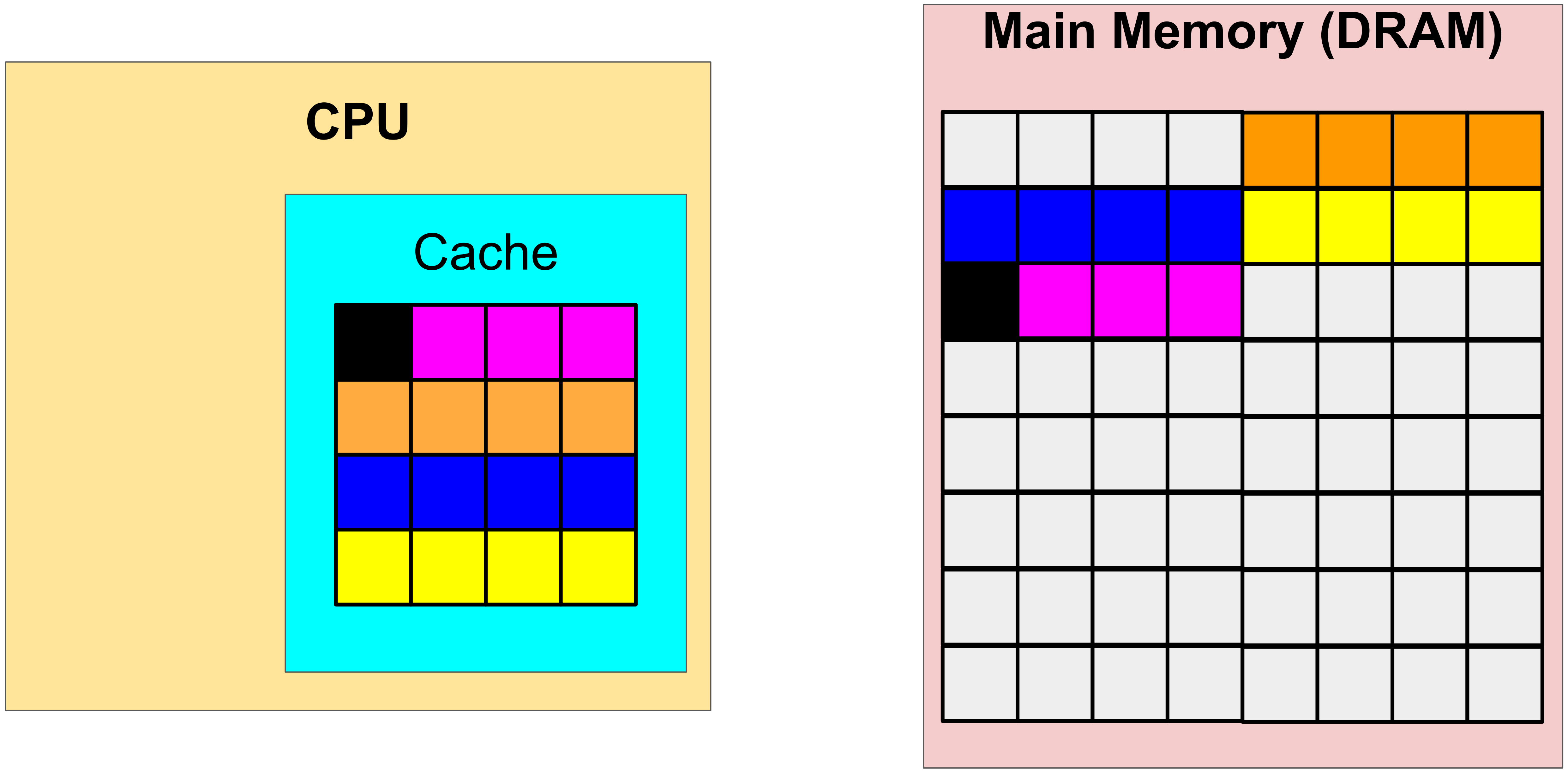
Caches

Eviction



Caches

Eviction

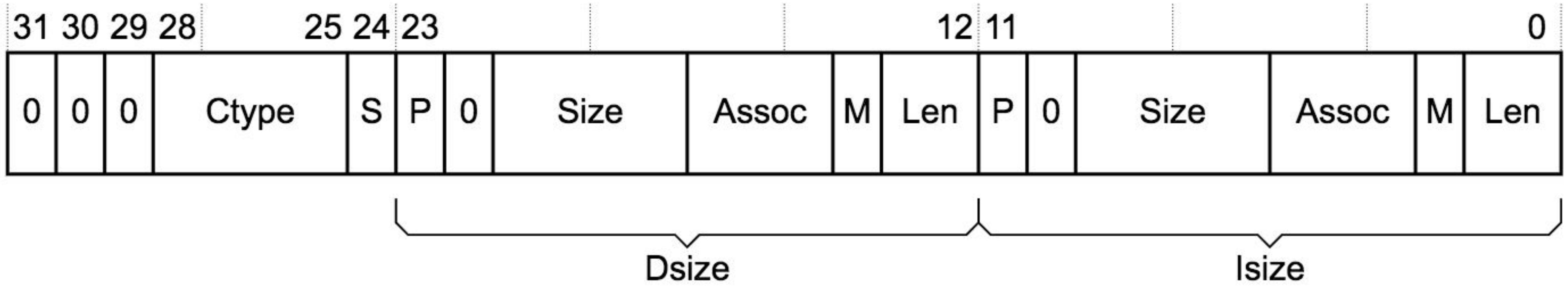


BCM 2835 Data Cache

L1 Total Size: 16 KB

L1 Block Size: 32 Bytes

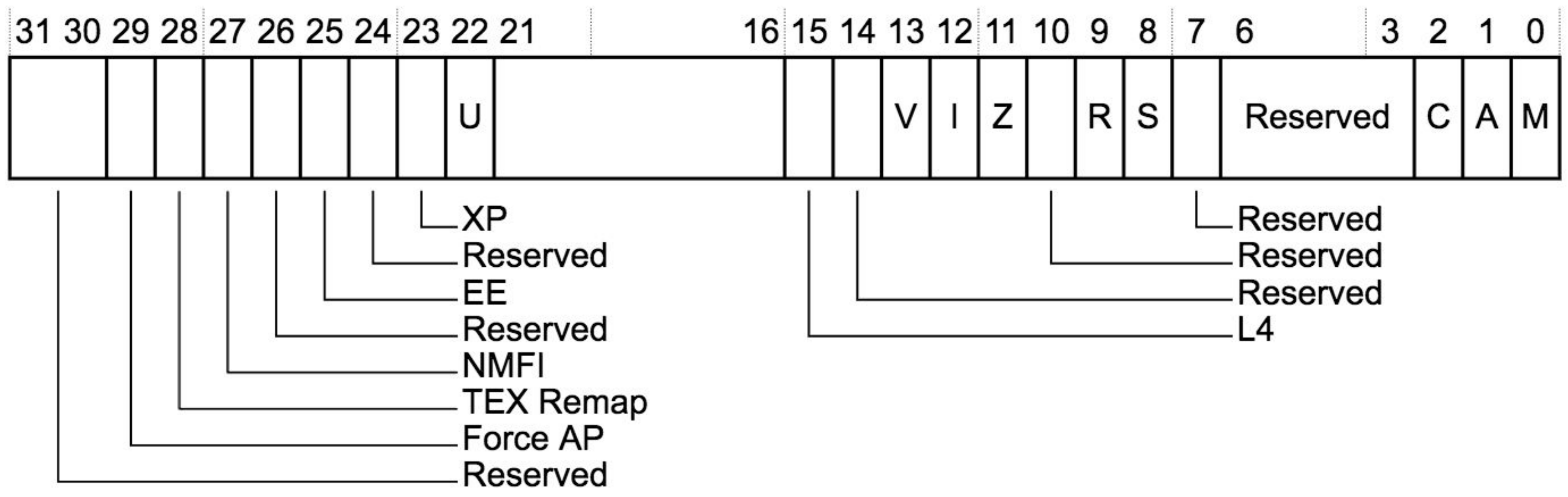
Figure 3.3. Cache Type Register format



```
// system.c
unsigned system_get_cache_type(void) {
    // See section 3.3.2 in arm1176
    unsigned reg;
    __asm__ volatile("mrc   p15, 0, %0, c0, c0, 1" : "=r"(reg));
    return reg;
}
```

Enabling Cache

Figure 3.18. Control Register format



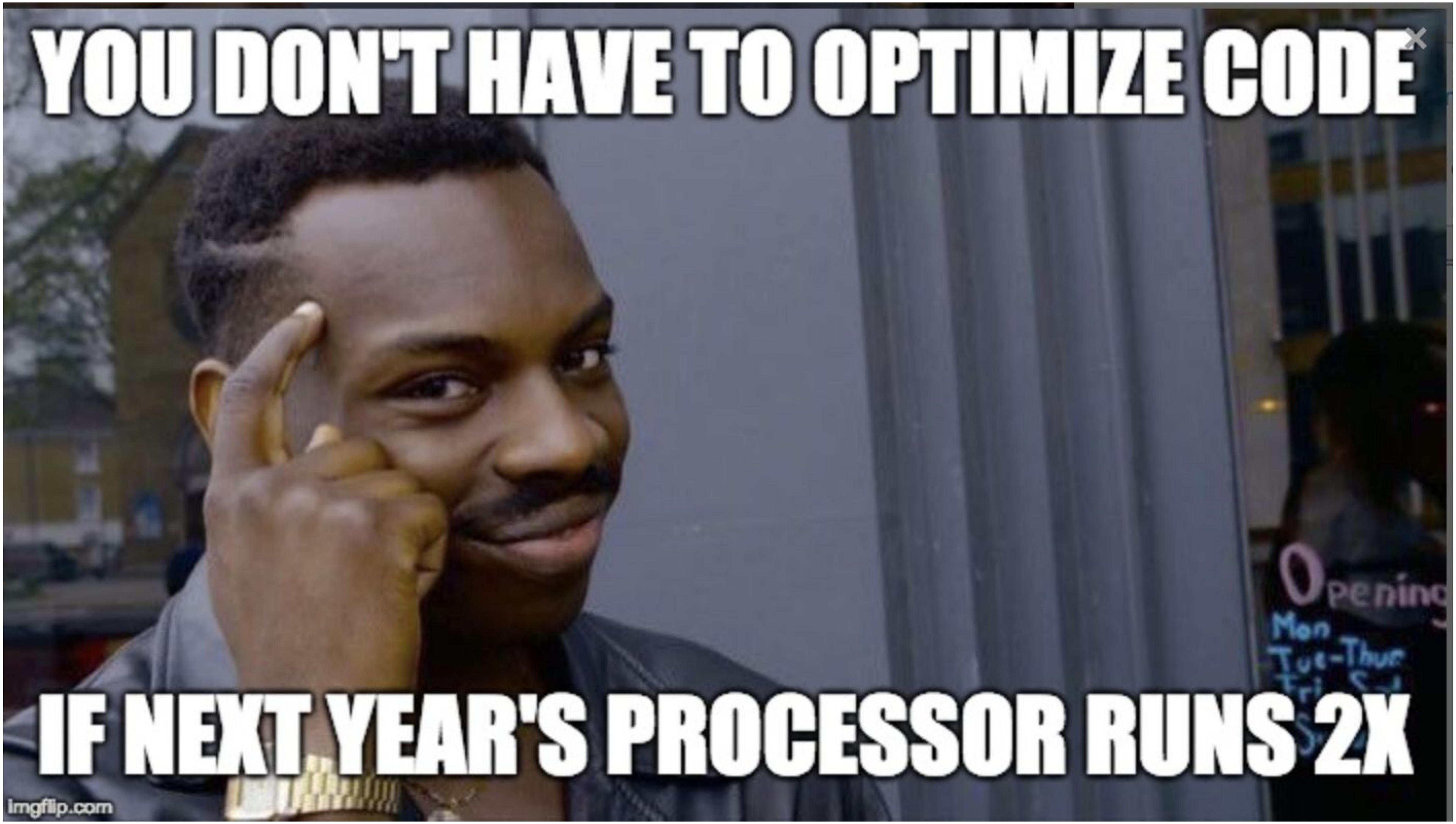
```
// system.c

system_enable_dcache(void) {
    // See section 3.2.7 in arm1176
    unsigned reg;
    __asm__ volatile("mrc   p15, 0, %0, c1, c0, 0" : "=r"(reg));
    reg |= SYSTEM_DCACHE_ENABLE; //(1<<2)
    __asm__ volatile("mcr   p15, 0, %0, c1, c0, 0" : : "r"(reg));
}
```

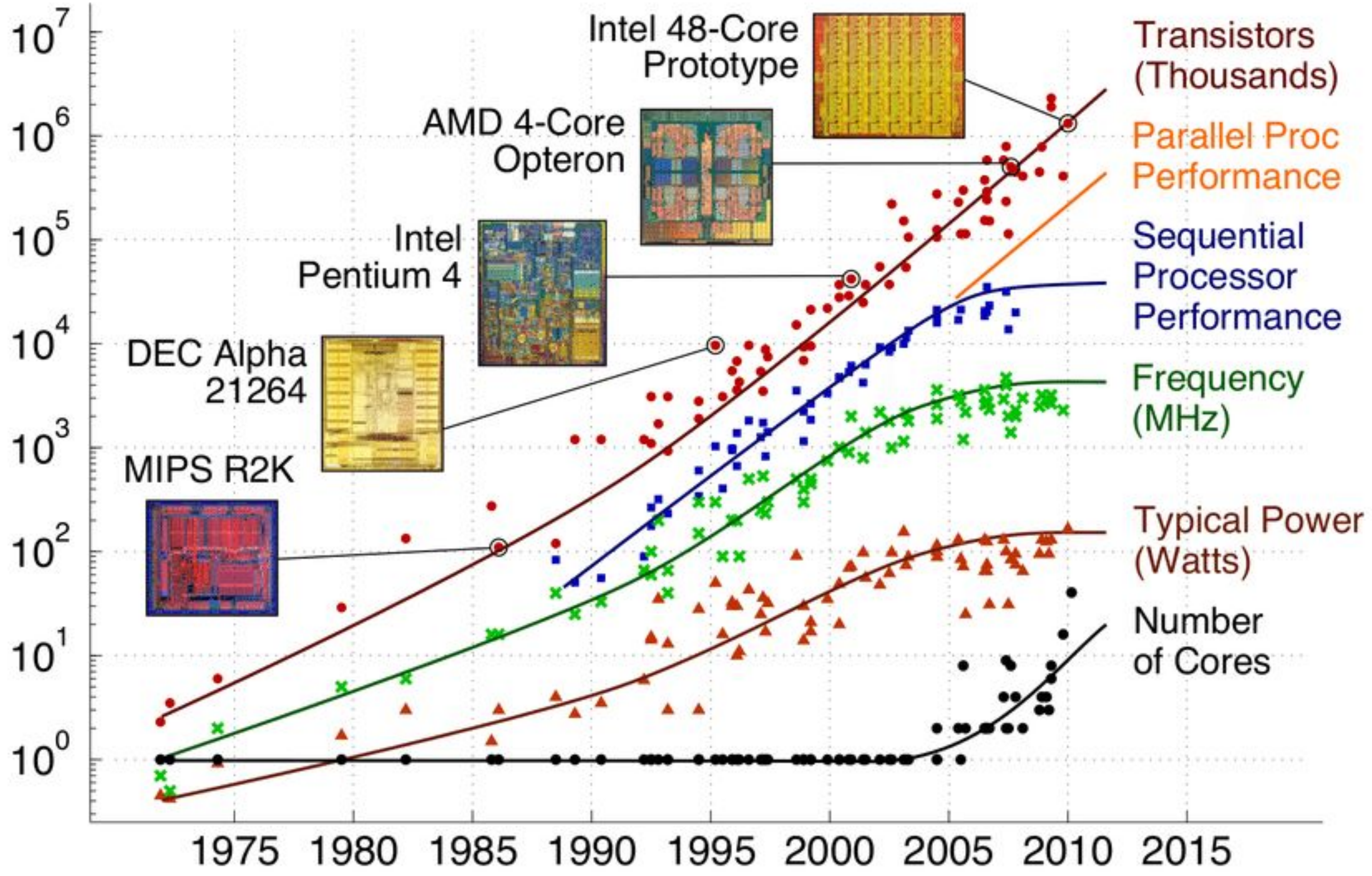
Demo: clear

w/ cache enabled

Know your hardware!

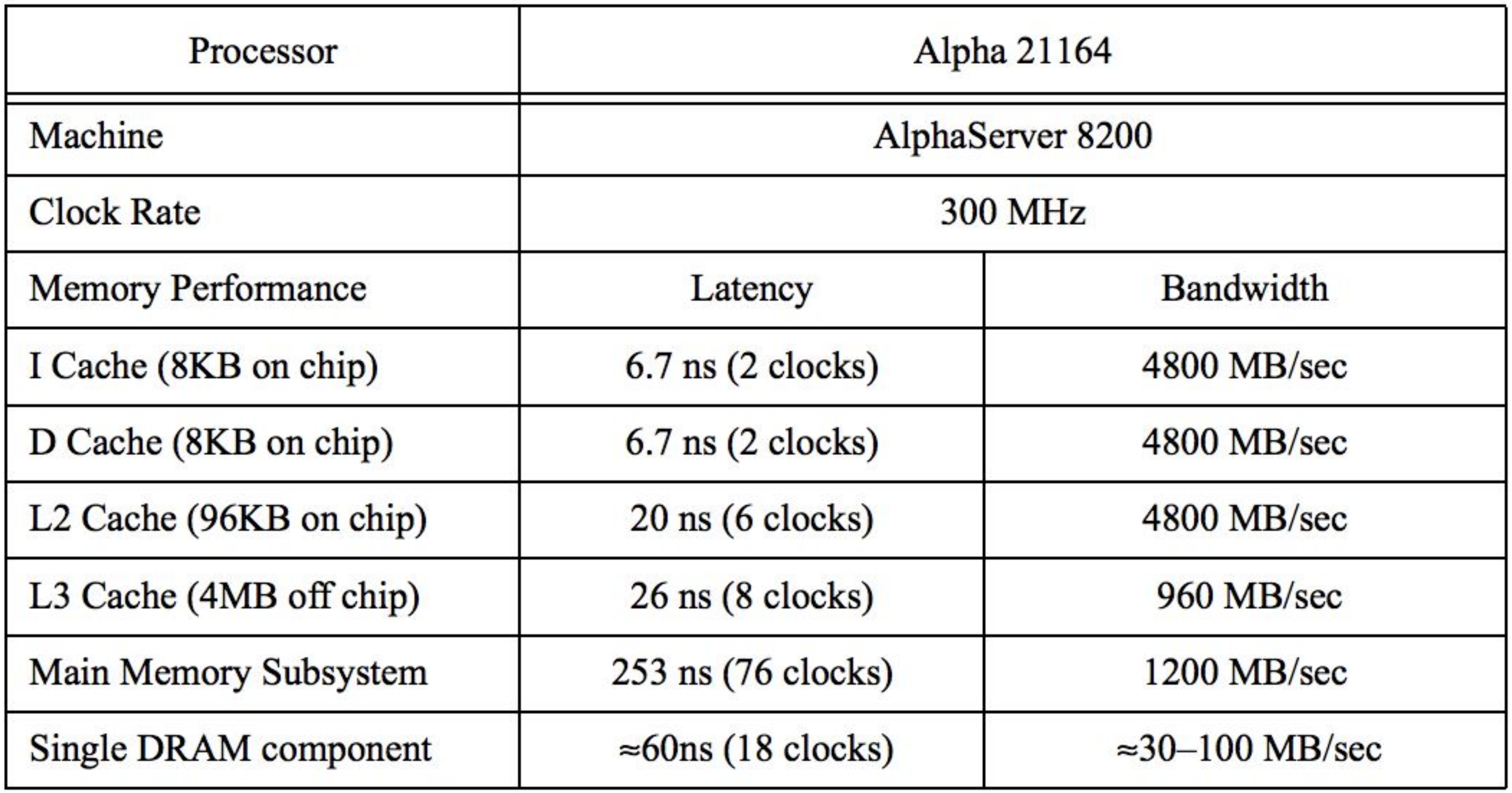

YOU DON'T HAVE TO OPTIMIZE CODE
IF NEXT YEAR'S PROCESSOR RUNS 2X
Opening
Mon
Tue-Thur
imgflip.com

Evolution of Processors



Source: Pranav Tendulkar, 2014, https://www.researchgate.net/figure/Evolution-of-multi-core-processors-1_fig1_281534326

Memory System Performance



Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

[Patterson, David, et al. "A case for intelligent RAM.”]

Moving data between the cpu and memory is the bottleneck

strcpy

```
for (int i = 0; i <= strlen(src); i++) {
    dst[i] = src[i]
}
```

All we’re doing is loading data from memory into the CPU and storing it back into memory

Avoiding the Memory Bottleneck

- Raspberry Pi has a **DMA Controller** that allows us to read and write memory without having to go through the processor (avoiding the load/store latency)
- Section 4 of BCM2835-ARM-Peripherals.pdf

Demo: dma

Measuring Performance

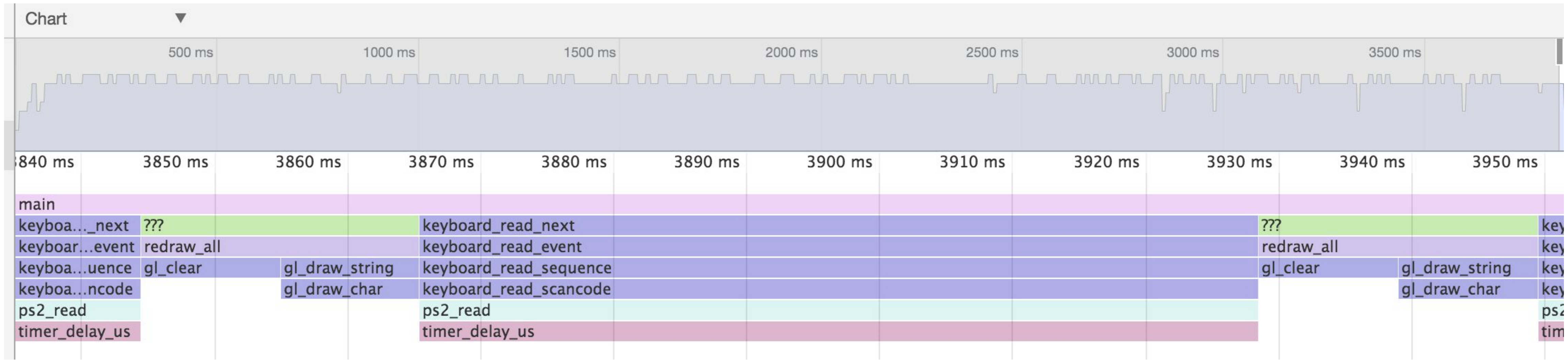
Don’t optimize blind

Taking Measurements

- **Hardware interrupts** - gprof.c
- **Code Instrumentation** - timer
- Also: instruction set simulation, OS hooks, performance counters
- Many techniques rely on sampling (statistical profilers) to trade off accuracy for speed

Visualizing Measurements

Chrome Developer Tools



- Output profiling information in standard format (linux *perf*)
- Use **thlorenz/cpuprofilify** to convert into *.cpuprofile* format

Demo: stackprof

Performance Metrics

Latency: time to complete an action

- How fast to load data from memory?
- How fast can instruction X complete?

Throughput: actions over a period of time

- How many instructions/second?
- How many compute operations per second? (GOPs, GFLOPS)

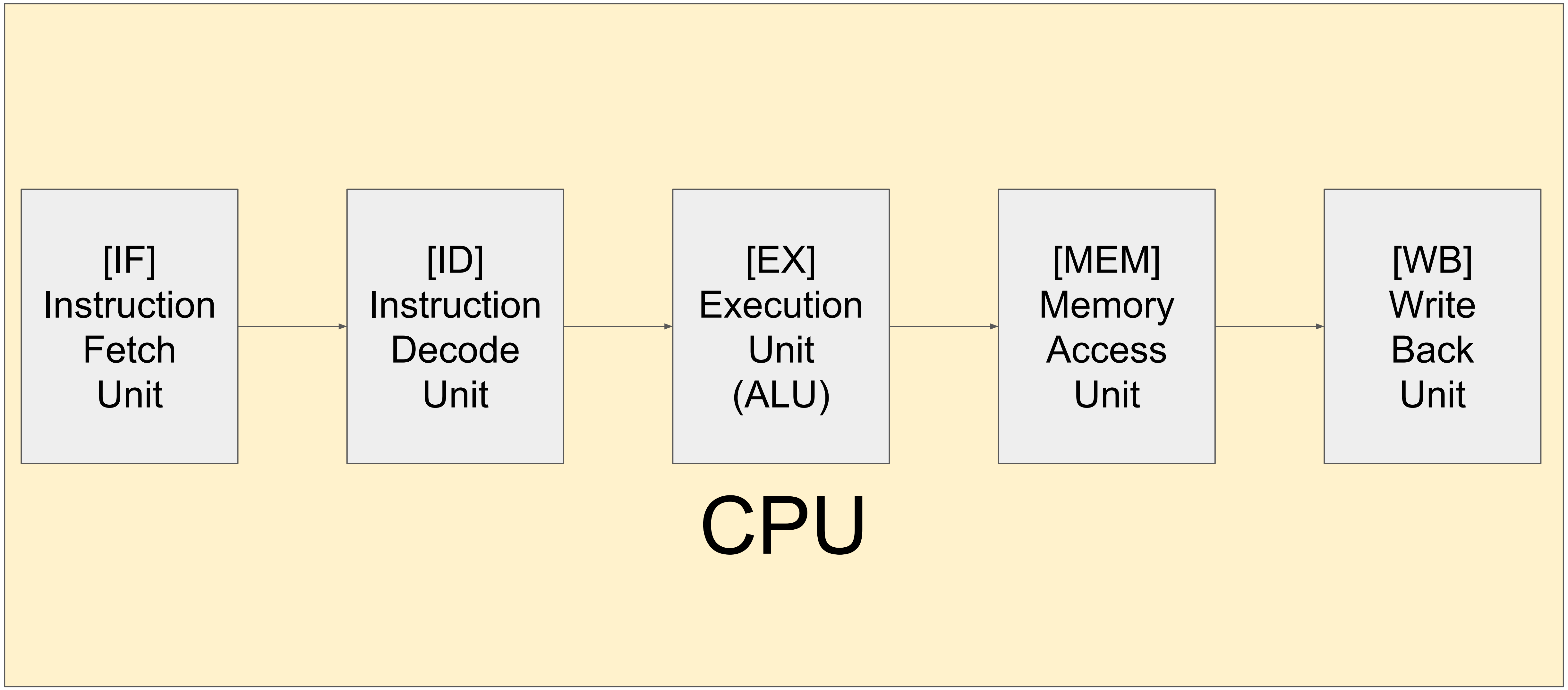
Stages in Instruction Execution

How does an instruction execute on a processor?

```
add r0, r1, r2
```

[IF] Instruction Fetch:	load instruction from memory based on PC
[ID] Instruction Decode:	decode the instruction operation, dst, src
[EX] Execute (ALU):	execute arithmetic
[MEM] Memory Access:	load or store data value to memory
[WB] Write Back:	write value back to register

Processor Units


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit
CPU

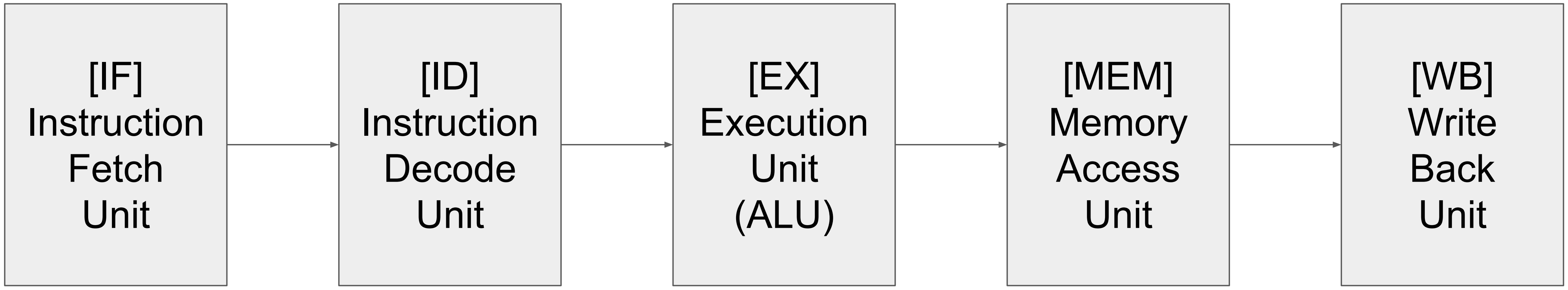
Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x0000
r1	0x8000
r2	0x0004

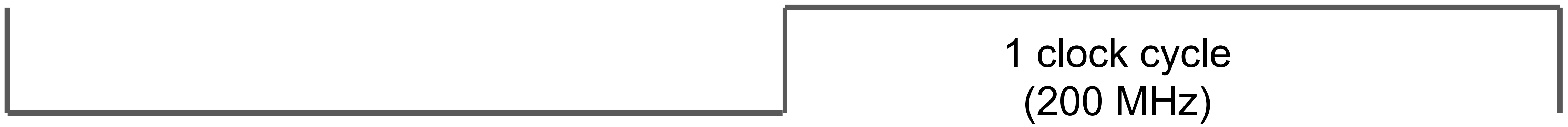
	MEMORY
0x8000	0xFFFF
0x8004	0x107E

PC = 0



[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

ldr r0, [r1, r2]

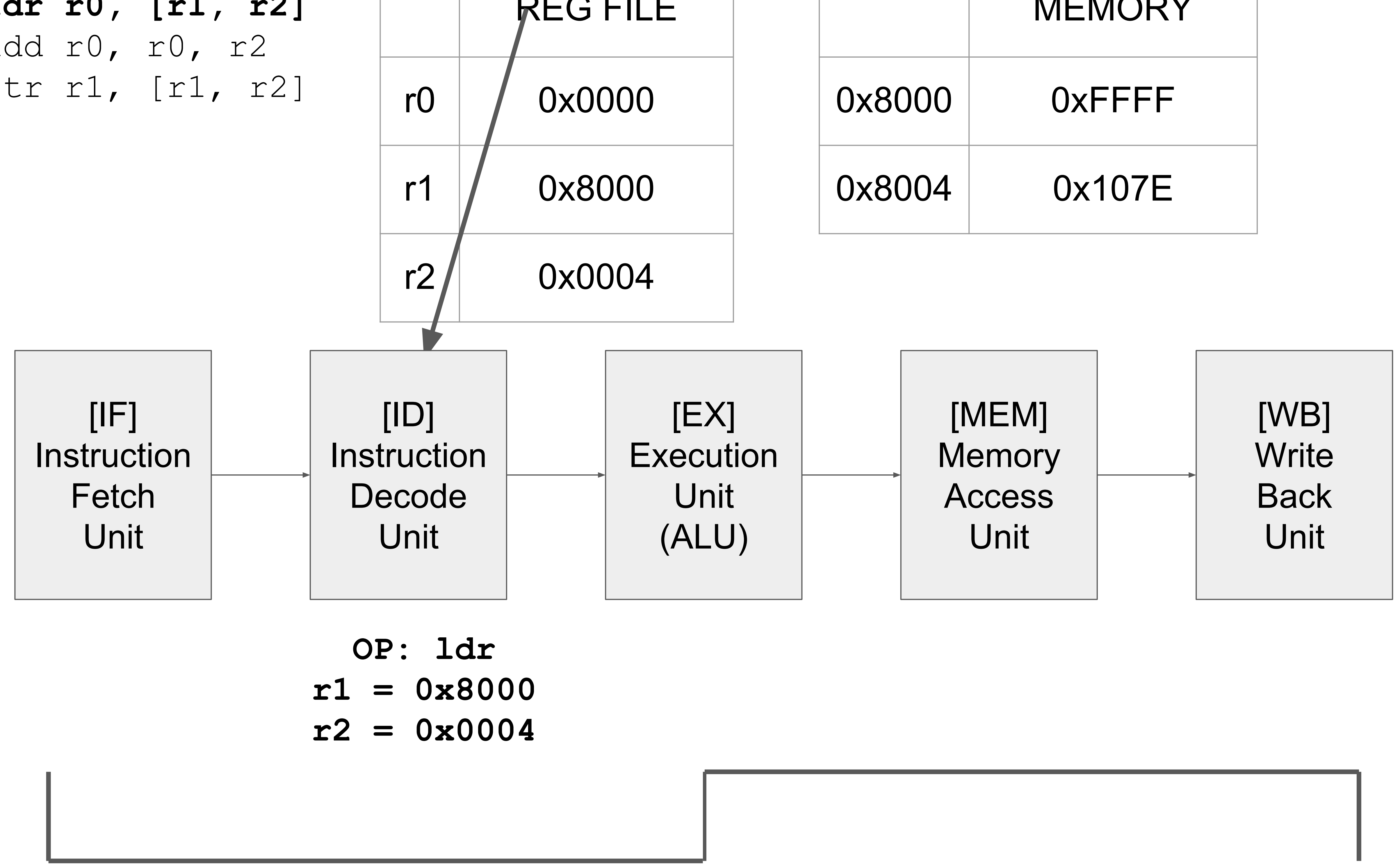


Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x0000
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

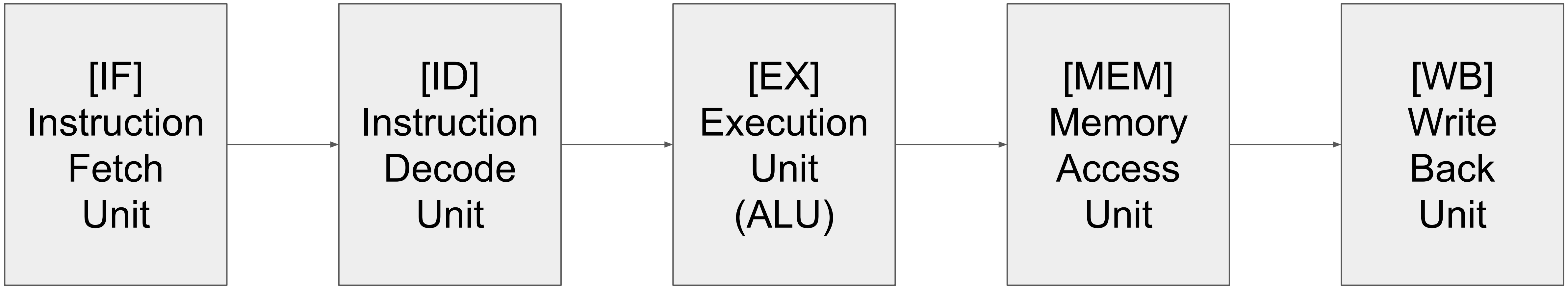
```
OP: ldr
r1 = 0x8000
r2 = 0x0004
```

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x0000
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

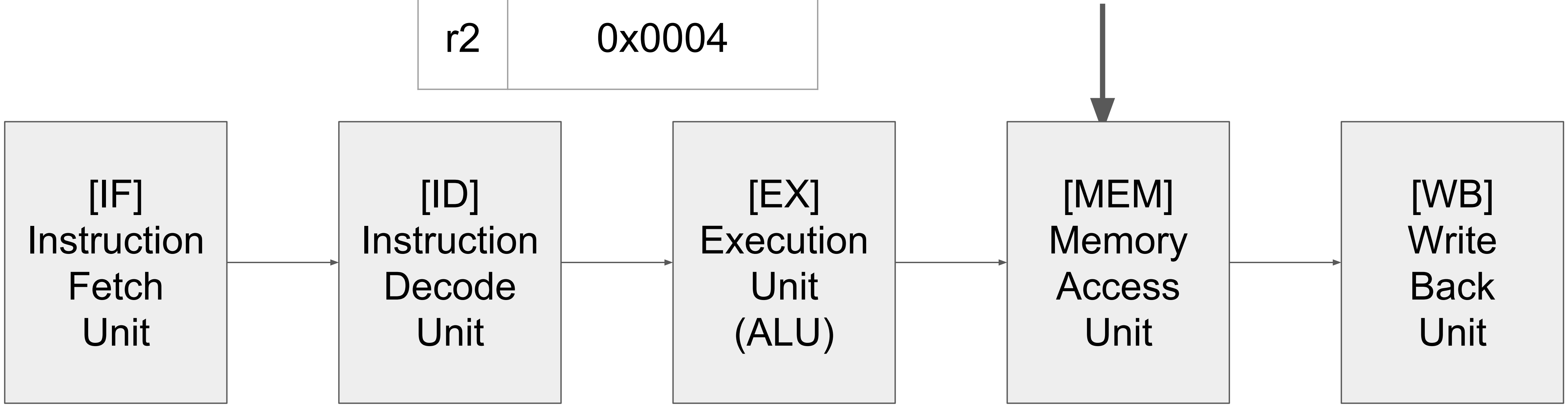
```
OP: ldr
ALU_val
= r1 + r2
= 0x8000 + 0x4
= 0x8004
```

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x0000
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

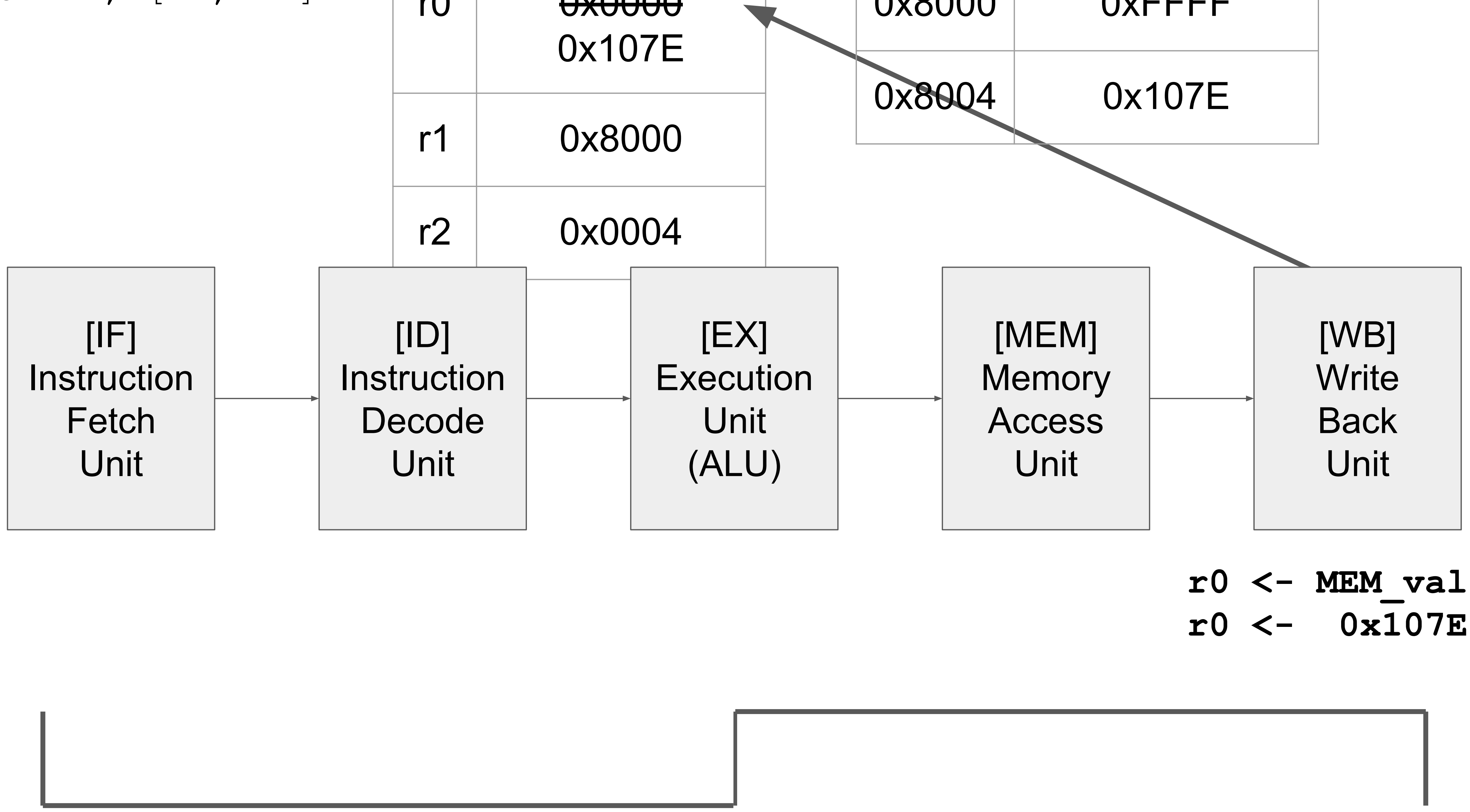
```
OP: ldr
MEM_val
= MEM[ALU_val]
= MEM[0x8004]
= 0x107E
```

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	~~0x0000~~ 0x107E
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

```
r0 <- MEM_val
r0 <-  0x107E
```

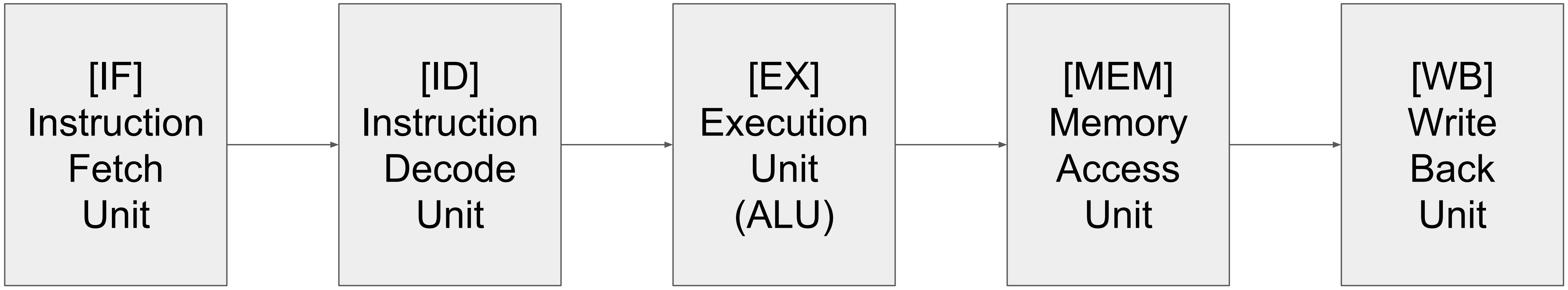
Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x107E
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E

PC = 4


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

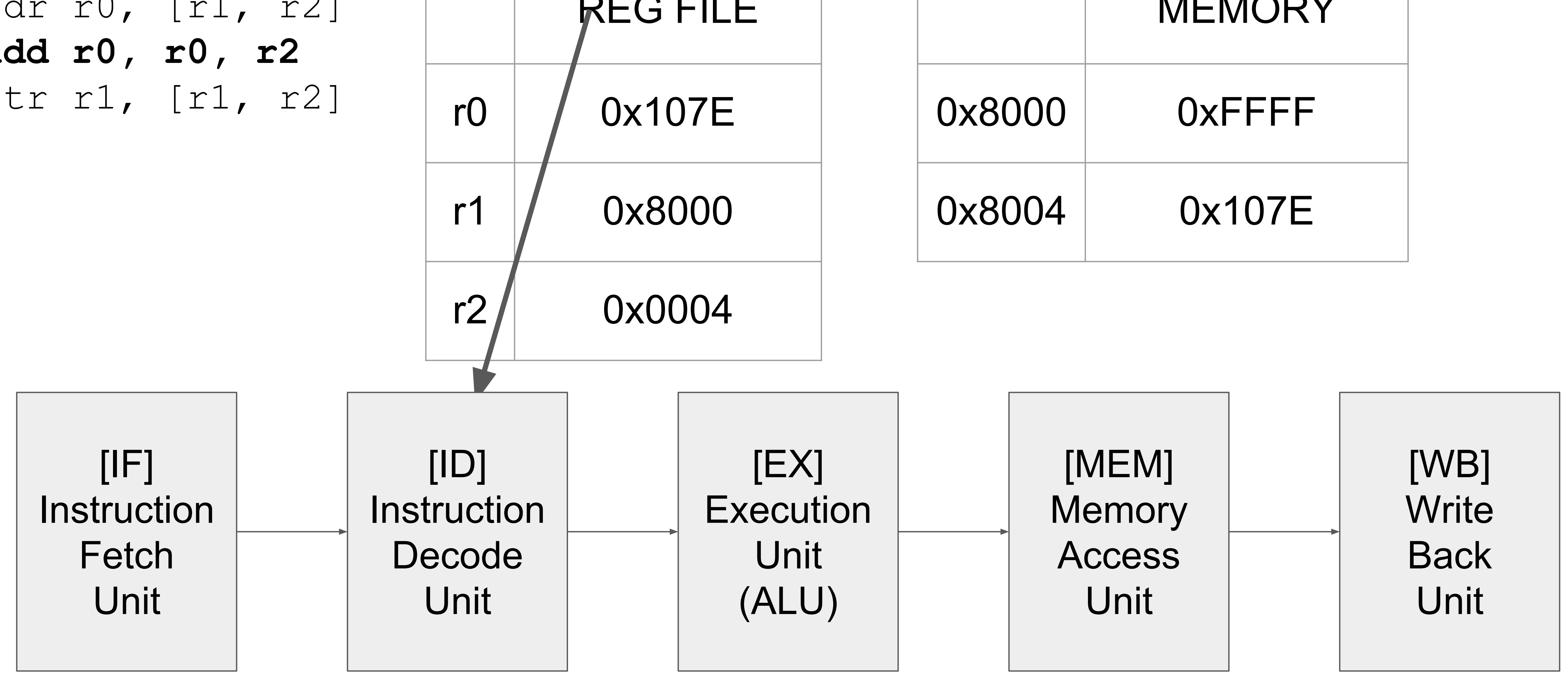
add r0, r0, r2

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x107E
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

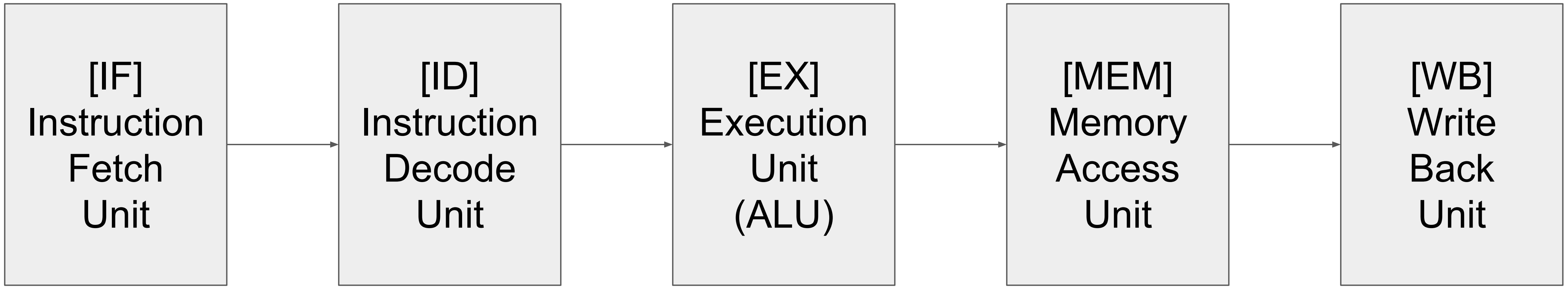
```
  OP: add
r0 = 0x107E
r2 = 0x0004
```

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x107E
r1	0x8000
r2	0x0004

	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

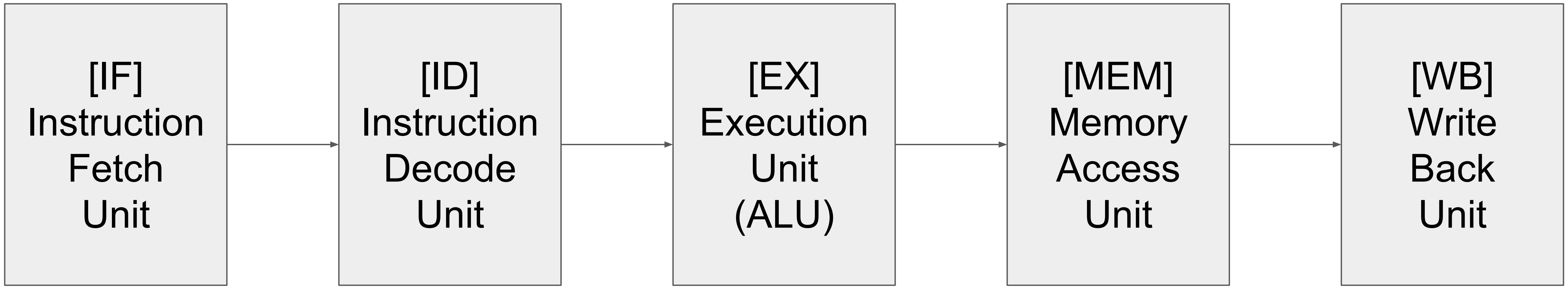
```
OP: ldr
ALU_val
= r0 + r2
= 0x107E + 0x4
= 0x1083
```

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```

	REG FILE
r0	0x107E
r1	0x8000
r2	0x0004

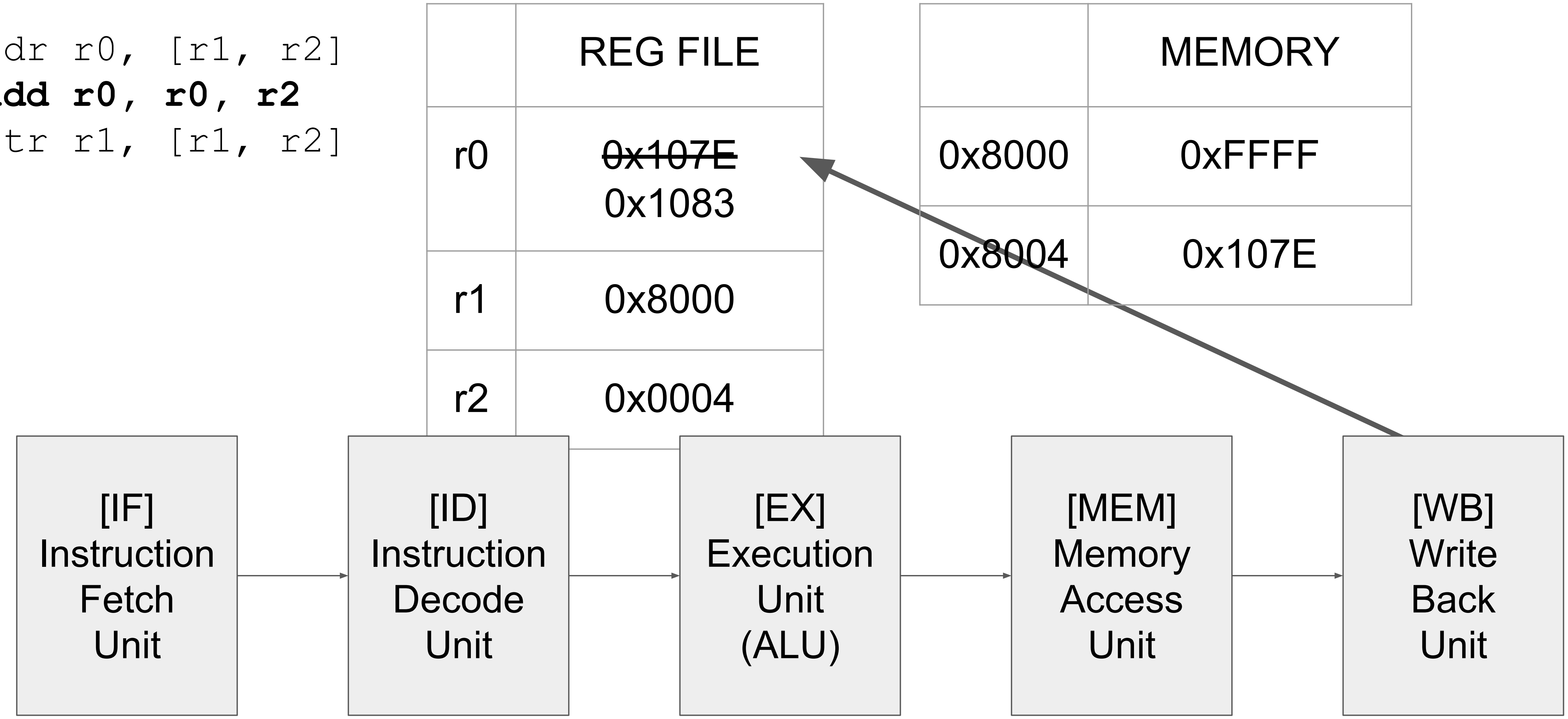
	MEMORY
0x8000	0xFFFF
0x8004	0x107E


[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

OP: add

Single-Cycle Processor

```
ldr r0, [r1, r2]
add r0, r0, r2
str r1, [r1, r2]
```



[IF] Instruction Fetch Unit
[ID] Instruction Decode Unit
[EX] Execution Unit (ALU)
[MEM] Memory Access Unit
[WB] Write Back Unit

	REG FILE
r0	~~0x107E~~ 0x1083
r1	0x8000
r2	0x0004

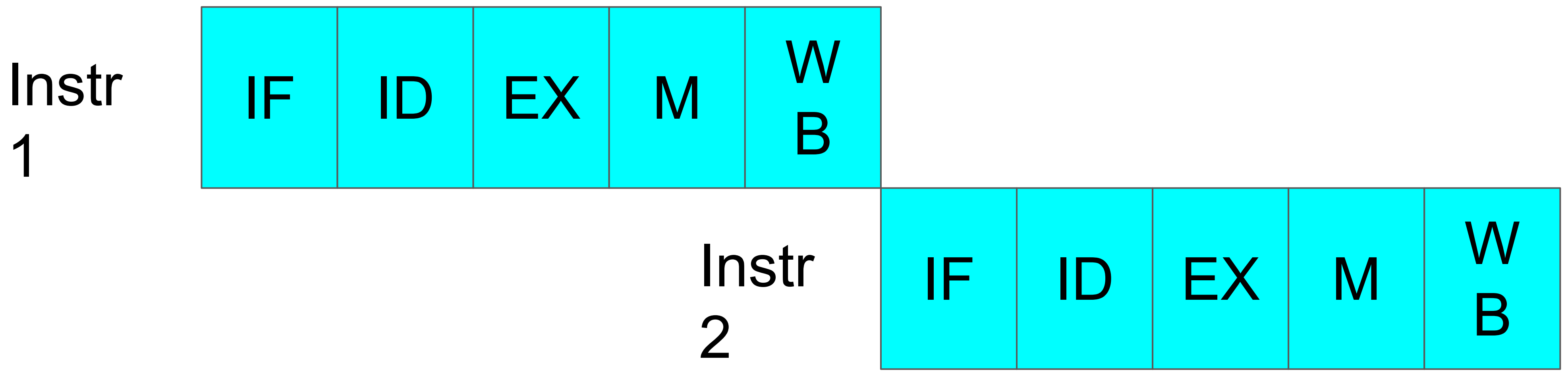
	MEMORY
0x8000	0xFFFF
0x8004	0x107E

```
r0 <- ALU_val
r0 <-  0x1083
```

Disadvantages of Single Cycle CPU

CPU resources not fully utilized

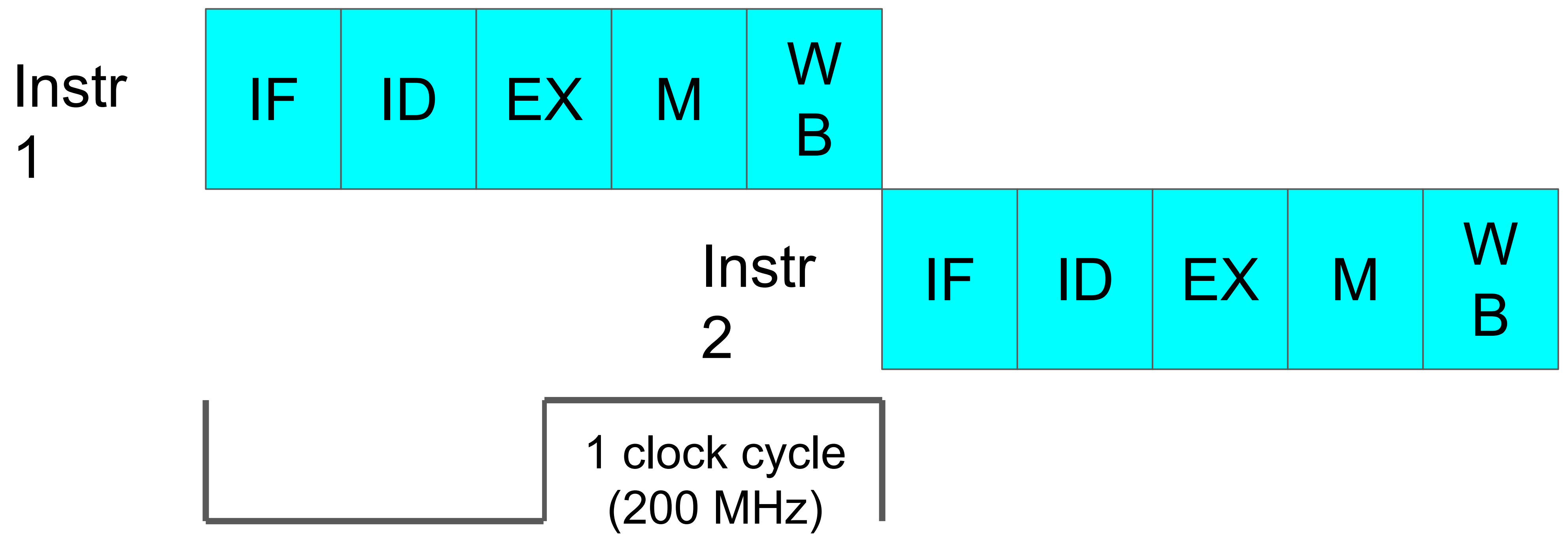
- 5 resource units: IF, ID, EX, MEM, WB
- Only 1 unit used at a time → 20% utilization



Disadvantages of Single Cycle CPU

CPU clock cycle is *slow* & 1 instruction per clock cycle

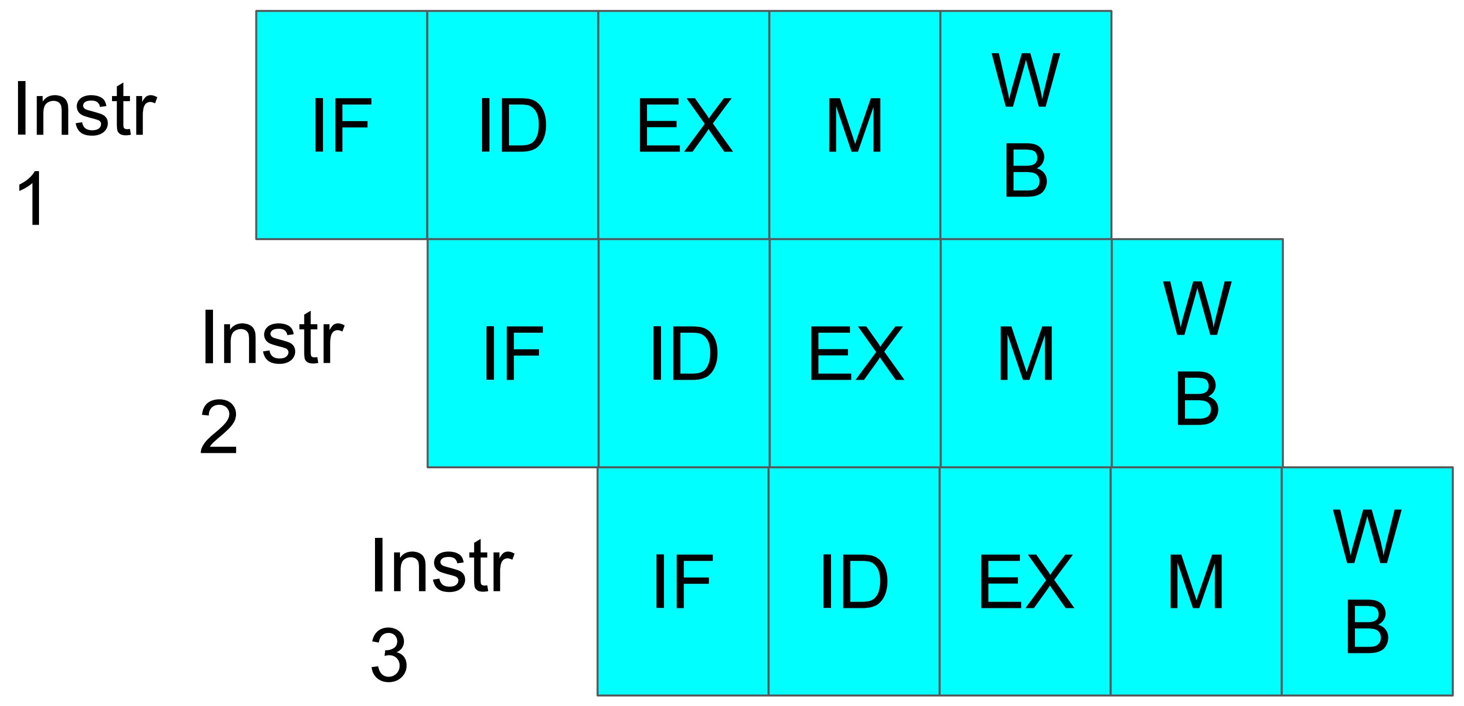
- @ 200 MHz clock → 200,000,000 instructions per second
- Latency: 5 ns per instruction
- Throughput: 200 M-instr/sec



Fixing the Single Cycle CPU

Pipelining

- Each instruction can start using a free unit before the previous instruction is completed
- Increases *throughput* and CPU *utilization* (100%)

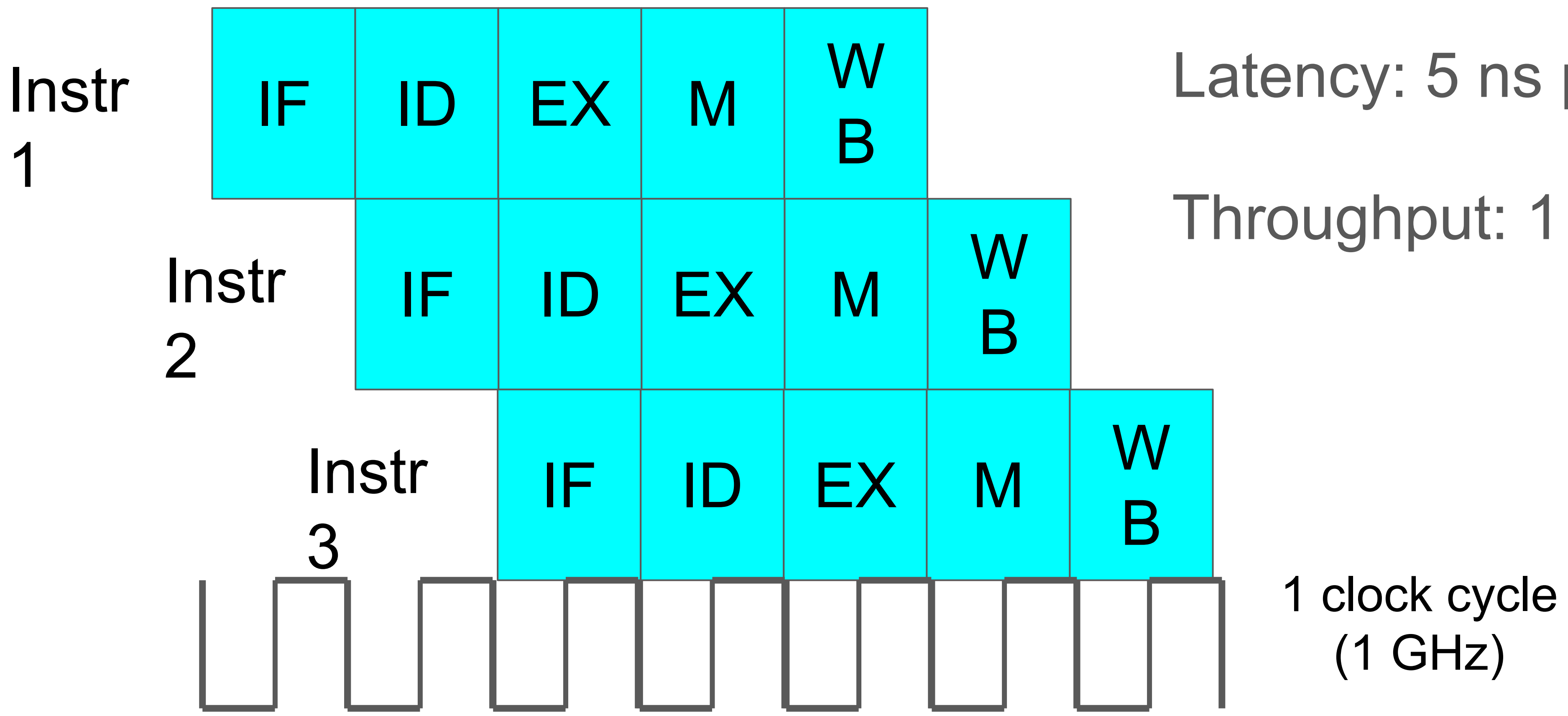


Fixing the Single Cycle CPU

Multi-Cycle

- Each instruction takes more clock cycles, but CPU can be clocked faster
- 1 clock cycle per functional unit (200 Mhz x 5 = 1 Ghz)

@ 1 GHz clock → 1,000,000,000 instructions per second



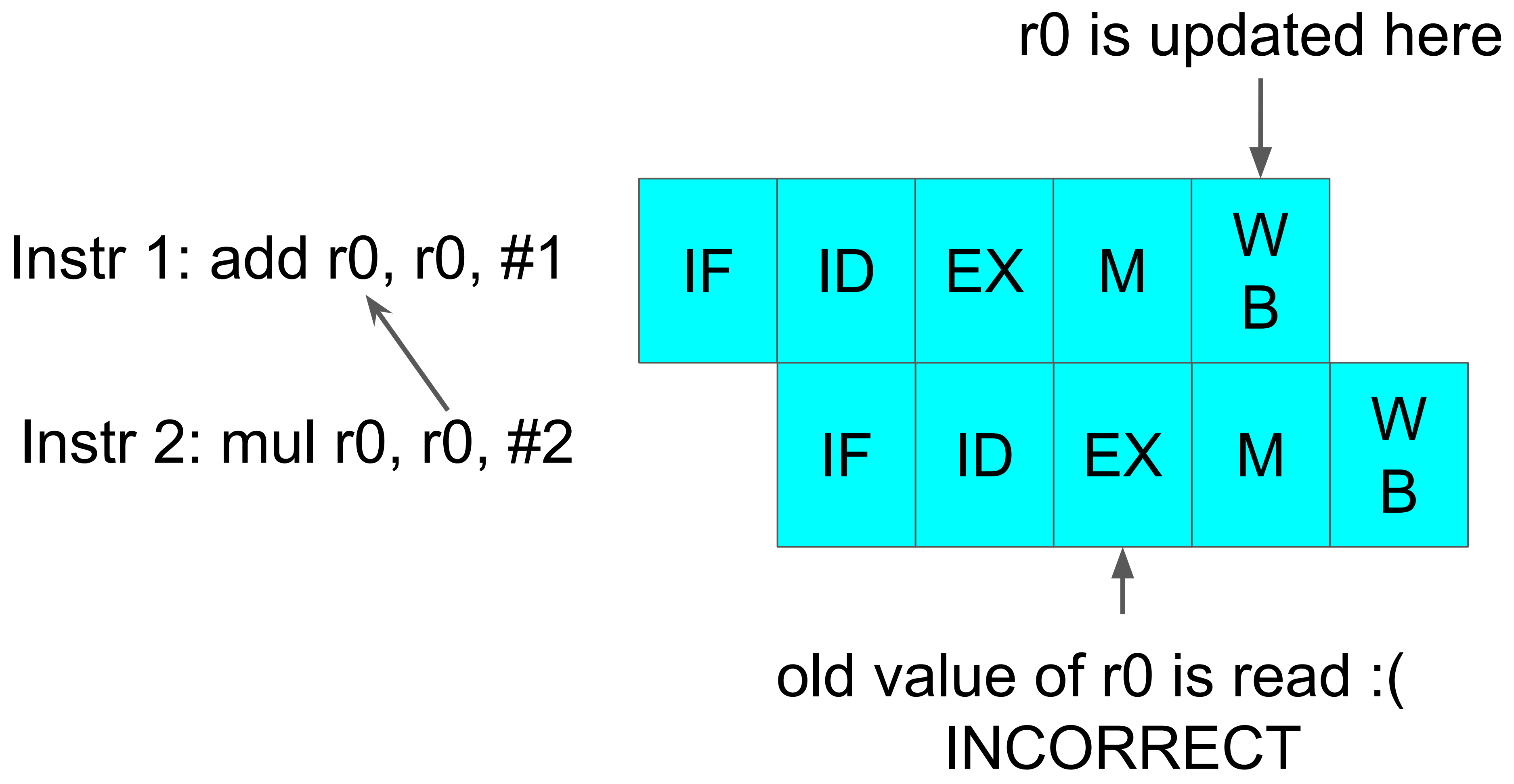
Latency: 5 ns per instruction

Throughput: 1 G-instr/sec

Data Hazards

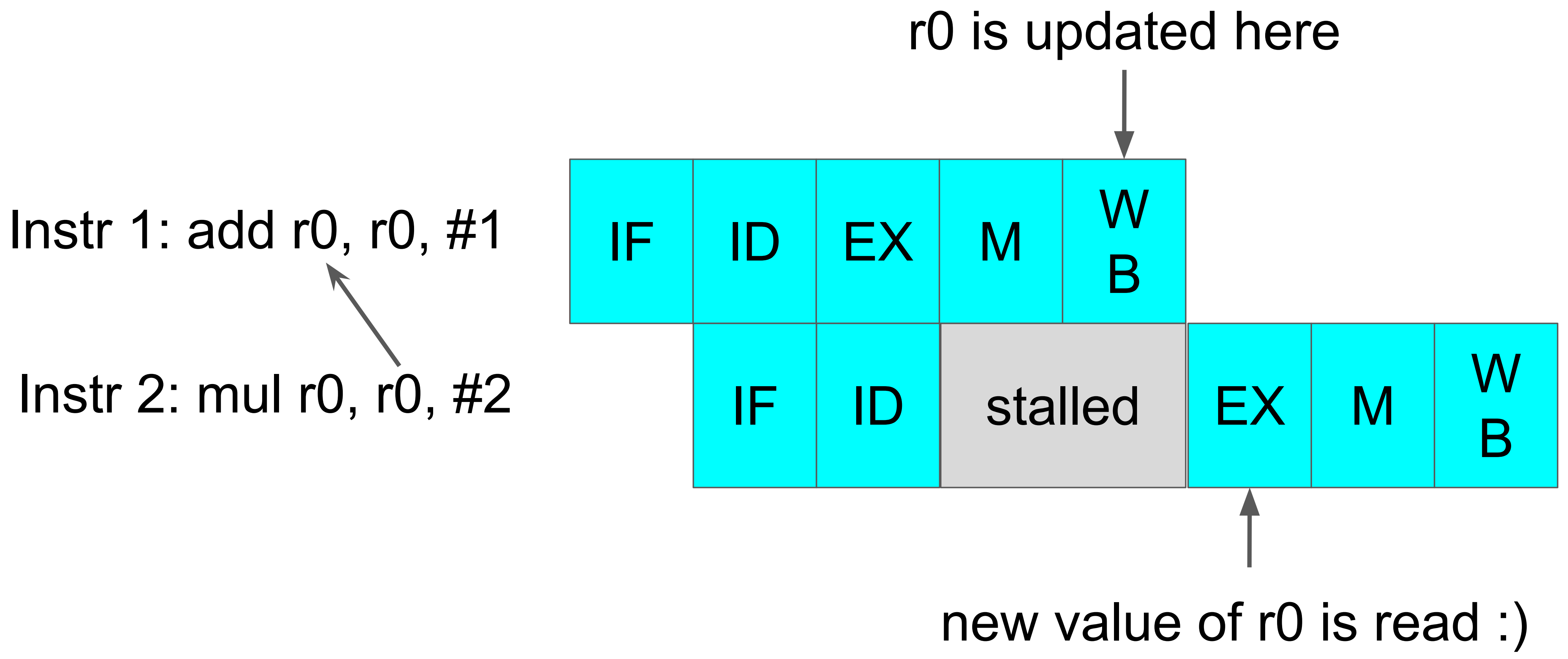
Instruction 2 in the pipeline depends on result of Instruction 1.

If Instruction 2 reads the register file (EX stage) before Instruction 1 has updated it (WB stage).



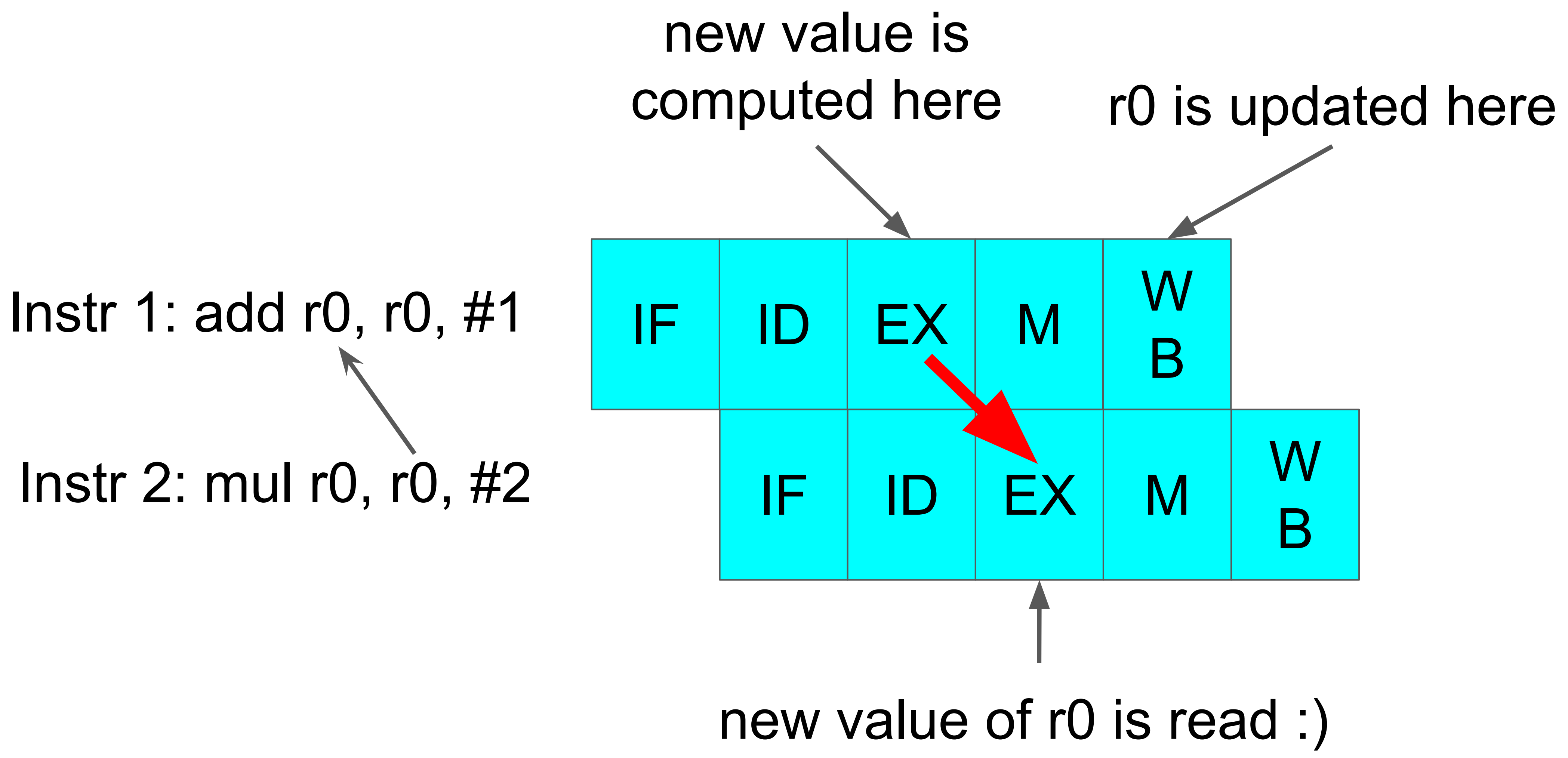
Data Hazards

Solution 1: *Detect* the dependency & *stall* until previous WB is complete.



Data Hazards

Solution 2: *Detect* the dependency & *forward* the result to next instruction.



Control Hazards

Which instruction to fetch after a branch instruction?

```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1     // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3       //not taken
```

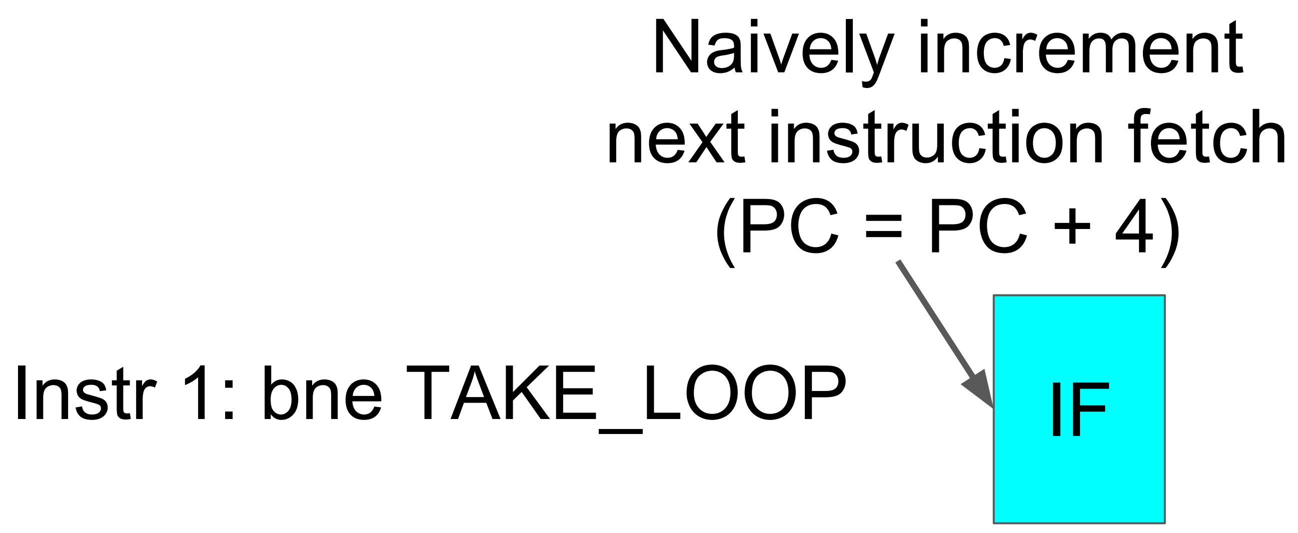
What stage of the pipeline do we compute where to branch?

Hint: PC + offset

Control Hazards

Which instruction to fetch after a branch instruction?

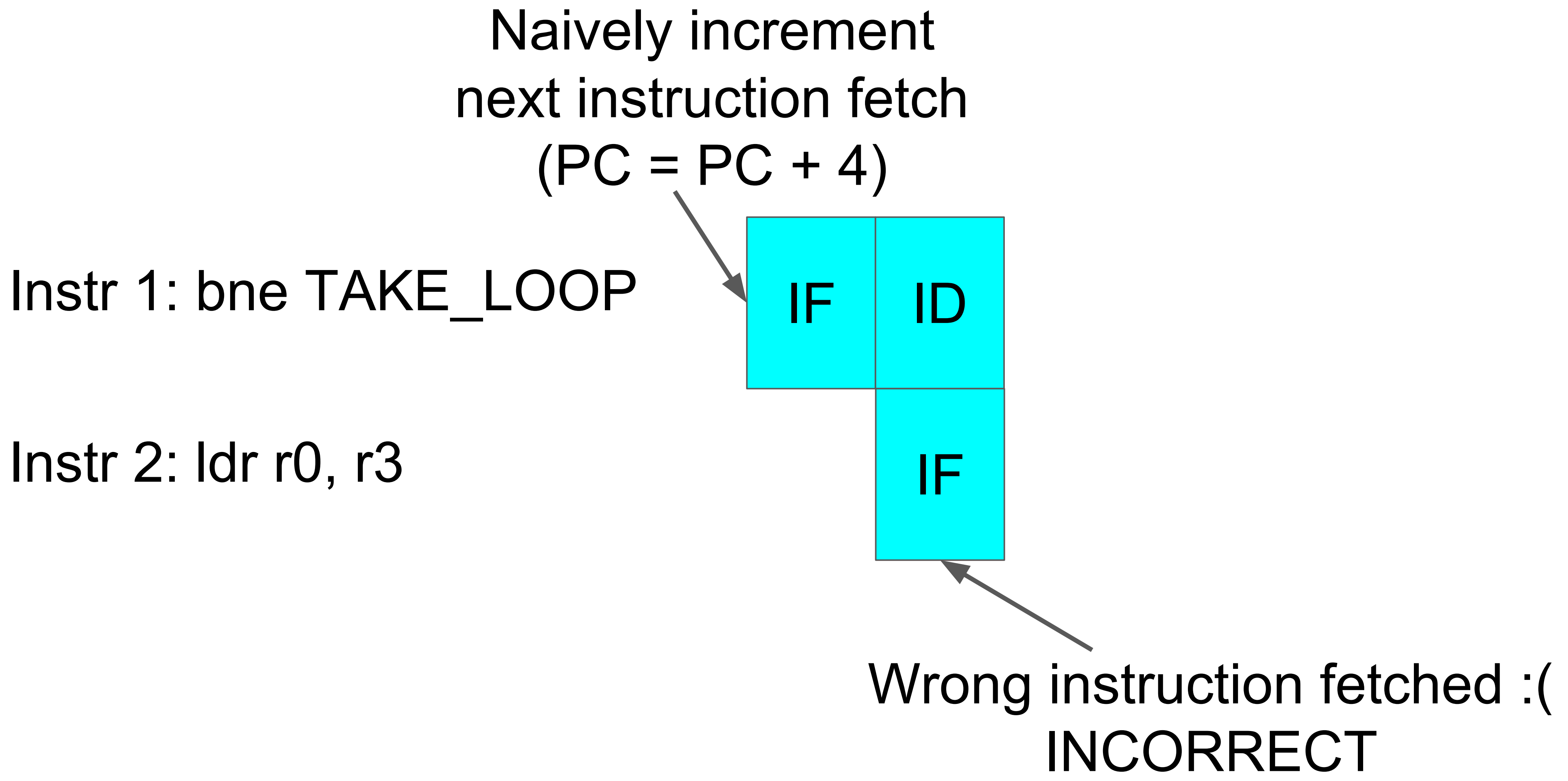
```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1      // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3        //not taken
```



Control Hazards

Which instruction to fetch after a branch instruction?

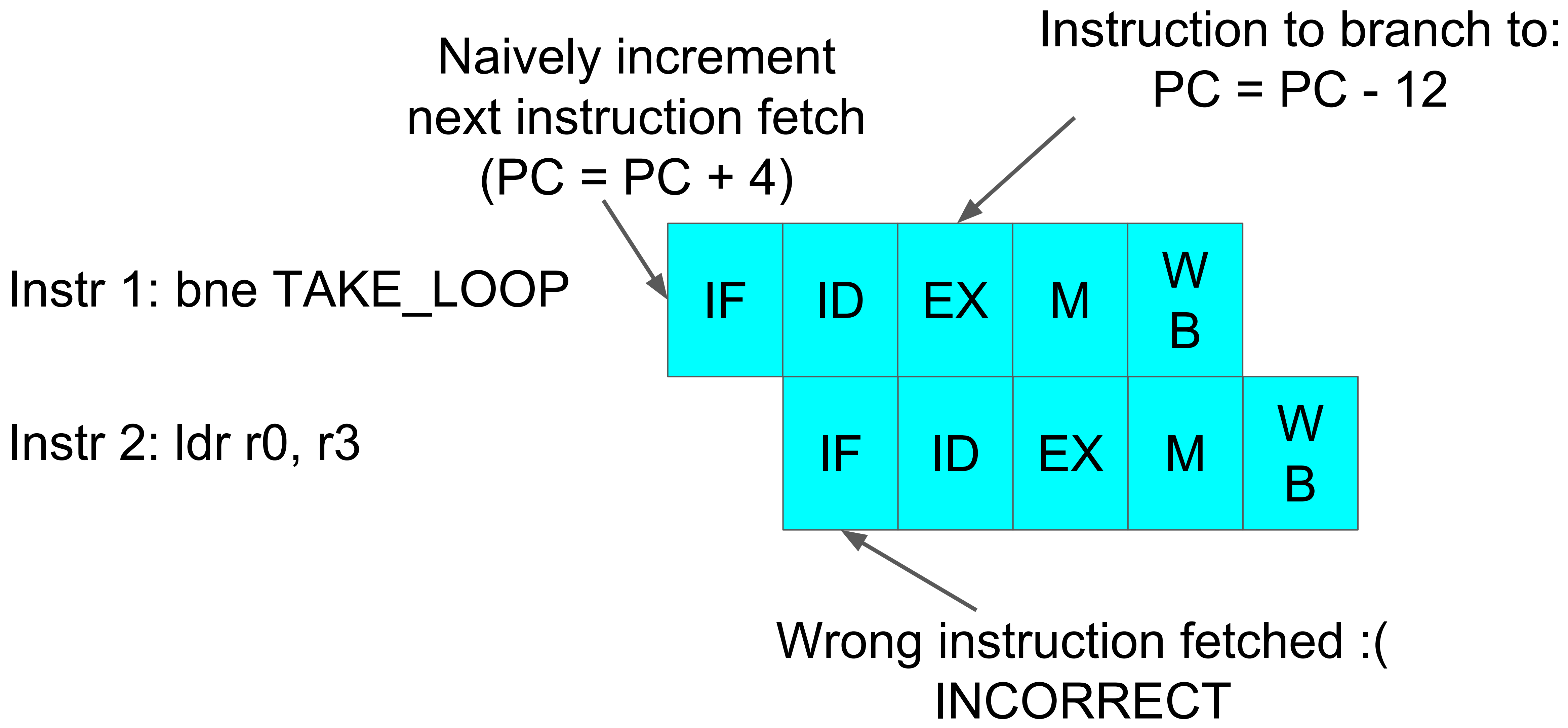
```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1      // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3        //not taken
```


Naively increment next instruction fetch (PC = PC + 4)
Instr 1: bne TAKE_LOOP
IF
ID
Instr 2: ldr r0, r3
IF
Wrong instruction fetched :(INCORRECT

Control Hazards

Which instruction to fetch after a branch instruction?

```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1      // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3        //not taken
```

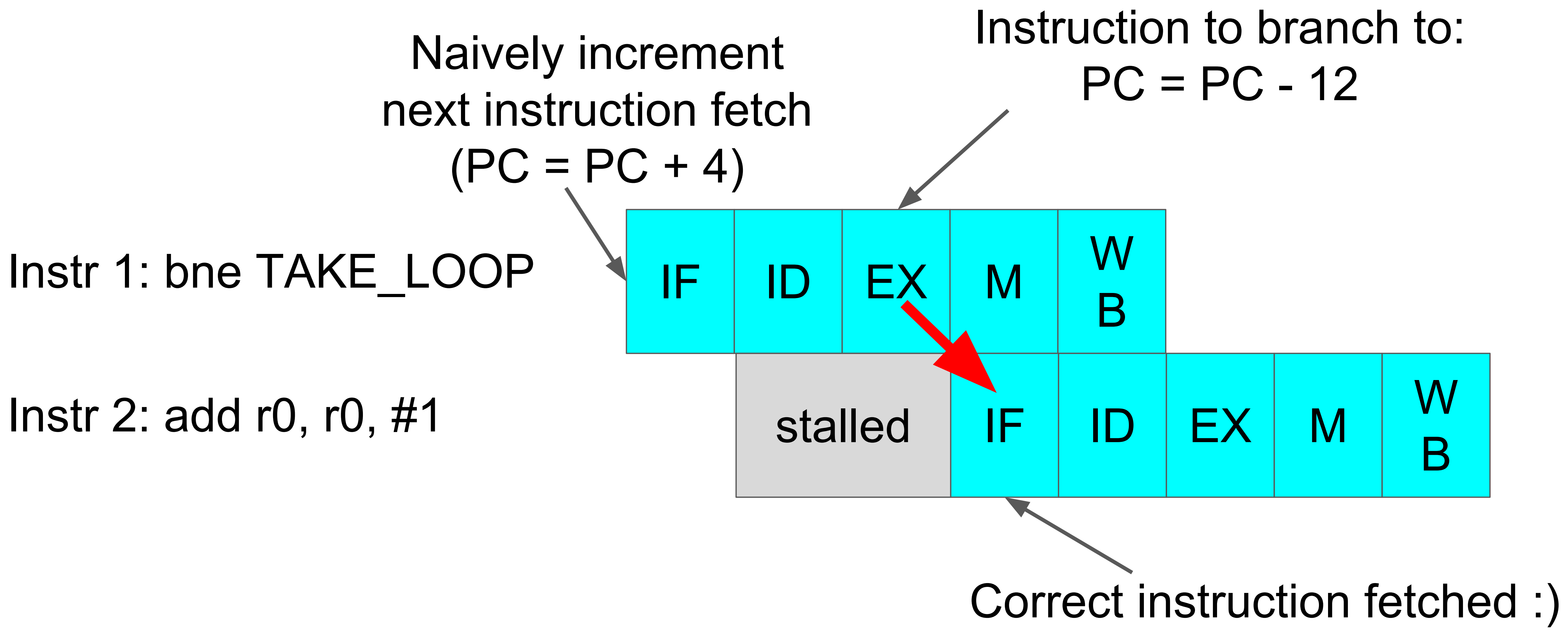


Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC - 12
Instr 1: bne TAKE_LOOP
IF
ID
EX
M
WB
Instr 2: ldr r0, r3
IF
ID
EX
M
WB
Wrong instruction fetched :(INCORRECT

Control Hazards

Solution 1: *Stall* until branch address is resolved.

```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1      // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3        //not taken
```

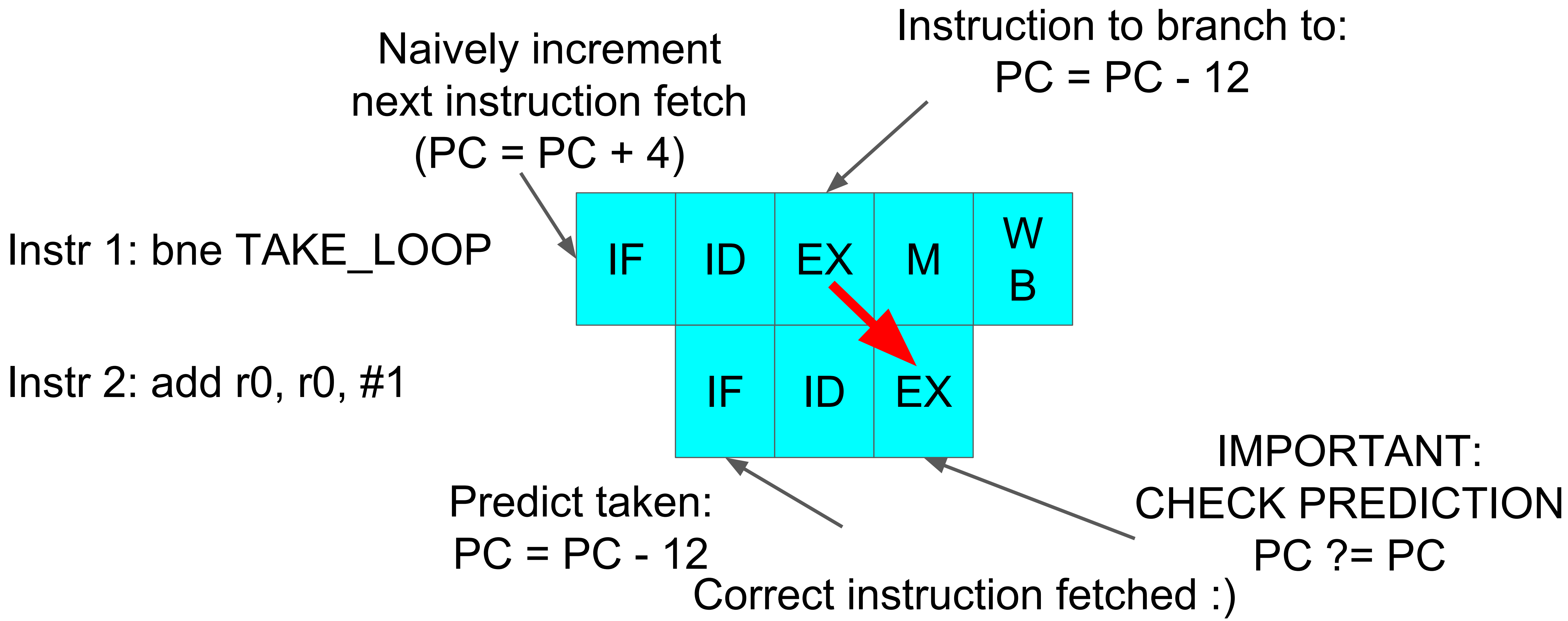


Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC - 12
Instr 1: bne TAKE_LOOP
IF
ID
EX
M
WB
Instr 2: add r0, r0, #1
stalled
IF
ID
EX
M
WB
Correct instruction fetched :)

Control Hazards

Solution 2: *Speculate - Try to predict* branch *taken* or *not taken*.

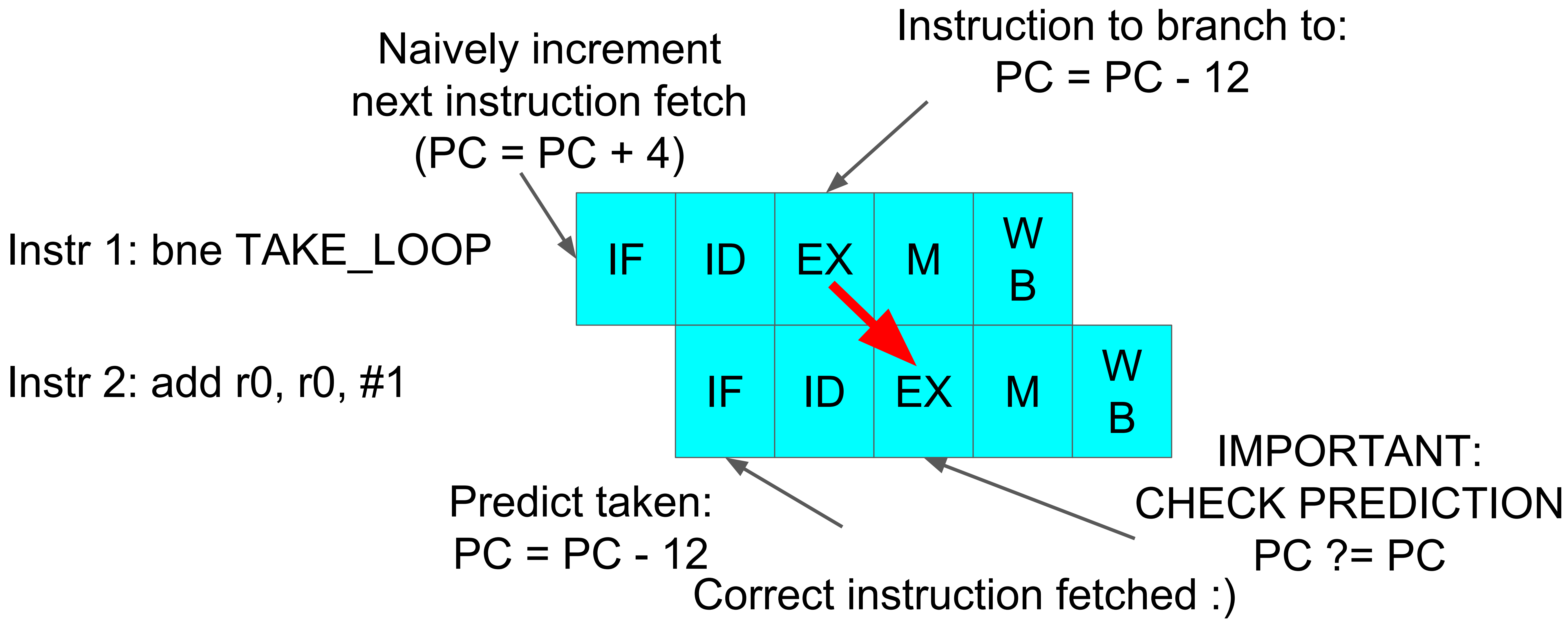
```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1     // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3       //not taken
```


Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC - 12
Instr 1: bne TAKE_LOOP
IF
ID
EX
M
WB
Instr 2: add r0, r0, #1
IF
ID
EX
Predict taken: PC = PC - 12
IMPORTANT: CHECK PREDICTION PC ?= PC
Correct instruction fetched :)

Control Hazards

Solution 2: *Speculate - Try to predict* branch *taken* or *not taken*.

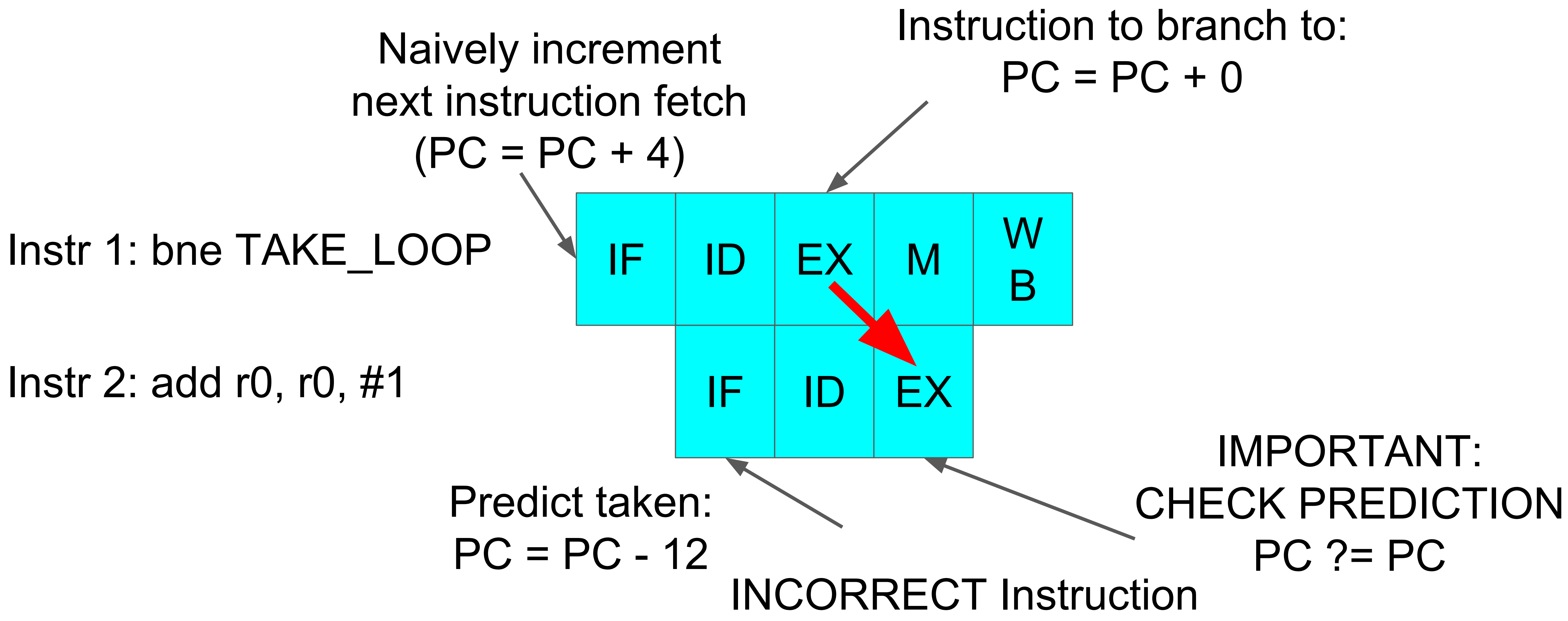
```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1      // r0=1, r1=10
   bne TAKE_LOOP
ldr r0, r3        //not taken
```


Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC - 12
Instr 1: bne TAKE_LOOP
IF
ID
EX
M
WB
Instr 2: add r0, r0, #1
IF
ID
EX
M
WB
Predict taken: PC = PC - 12
IMPORTANT: CHECK PREDICTION PC ?= PC
Correct instruction fetched :)

Control Hazards

Speculation: What if branch is predicted incorrectly?

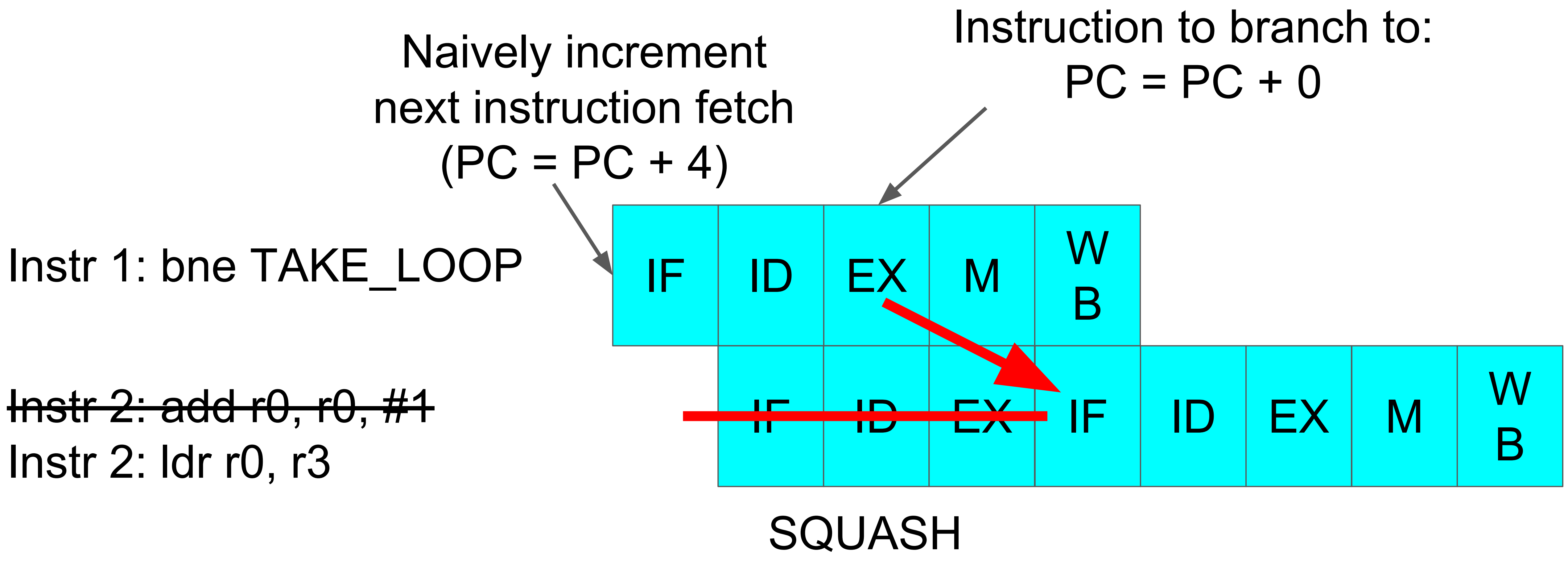
```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1     // r0=10, r1=10
   bne TAKE_LOOP
ldr r0, r3       //not taken
```


Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC + 0
Instr 1: bne TAKE_LOOP
IF
ID
EX
M
WB
Instr 2: add r0, r0, #1
IF
ID
EX
Predict taken: PC = PC - 12
IMPORTANT: CHECK PREDICTION PC ?= PC
INCORRECT Instruction

Control Hazards

Speculation: What if branch is predicted incorrectly?

```
TAKE_LOOP:
   add r0, r0, #1
   cmp r0, r1       // r0=10, r1=10
   bne TAKE_LOOP
ldr r0, r3         //not taken
```



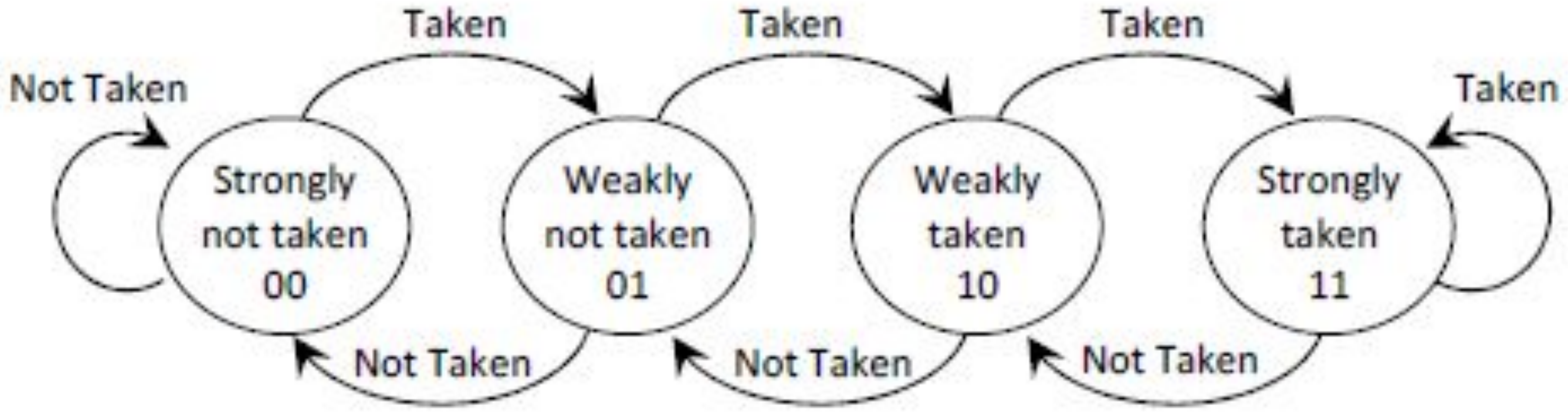
Naively increment next instruction fetch (PC = PC + 4)
Instruction to branch to: PC = PC + 0
Instr 1: bne TAKE_LOOP
~~Instr 2: add r0, r0, #1~~
Instr 2: ldr r0, r3
IF
ID
EX
M
WB
IF
ID
EX
IF
ID
EX
M
WB
SQUASH

Branch Prediction

Piece of *hardware* to predict if branch will be *taken* or *not taken.*

Simplest: Always predict TAKEN

2-bit Predictor:



Source: https://cw.fel.cvut.cz/old/courses/a0b36apo/en/tutorials/06/start

Demo: clear

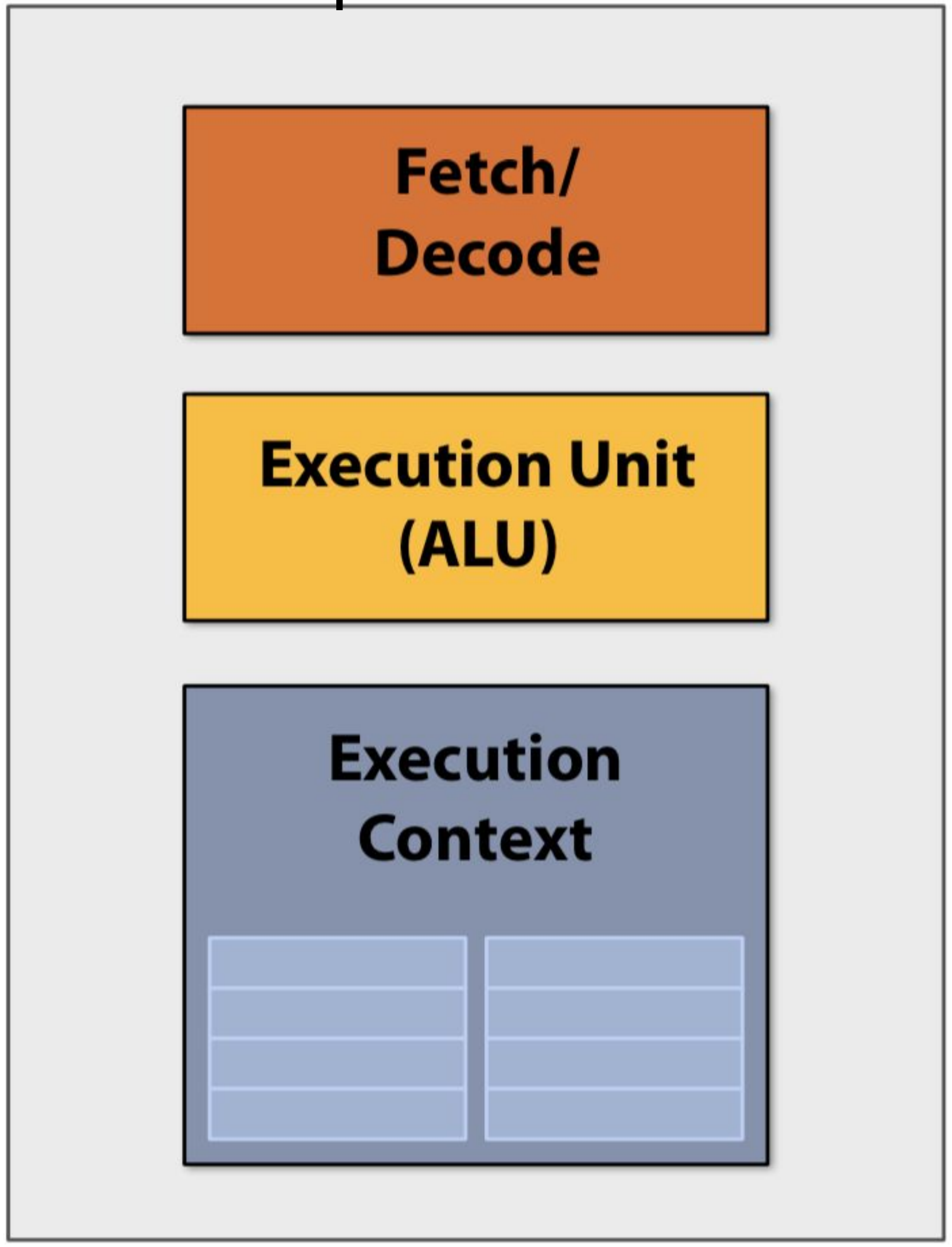
w/ branch prediction

Parallel Architecture

Images courtesy of CS 149 (cs149.stanford.edu)

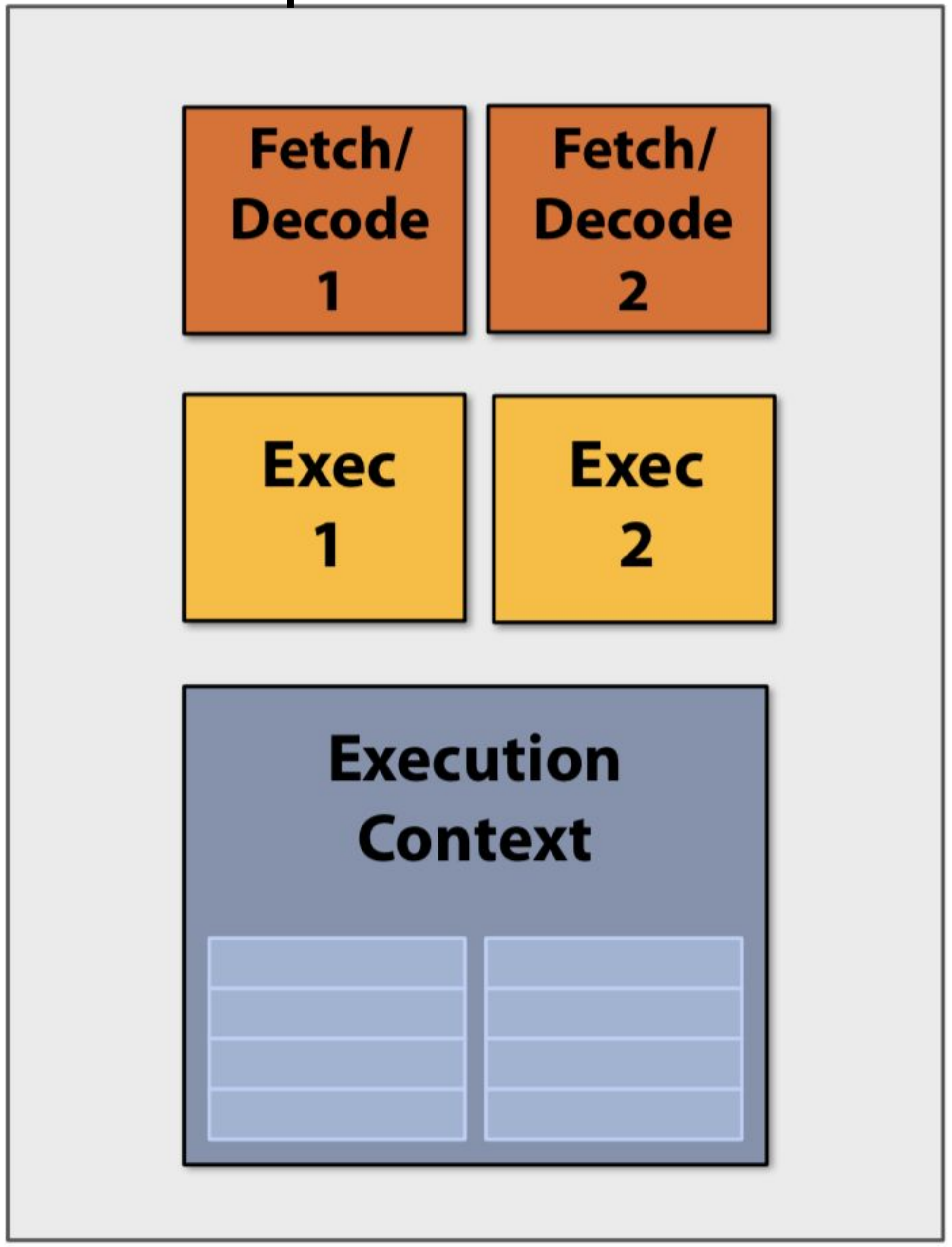
Instruction Level Parallelism (ILP)

Pipelined CPU



1 instr/cycle

Superscalar CPU



2 instr/cycle

Limits of ILP

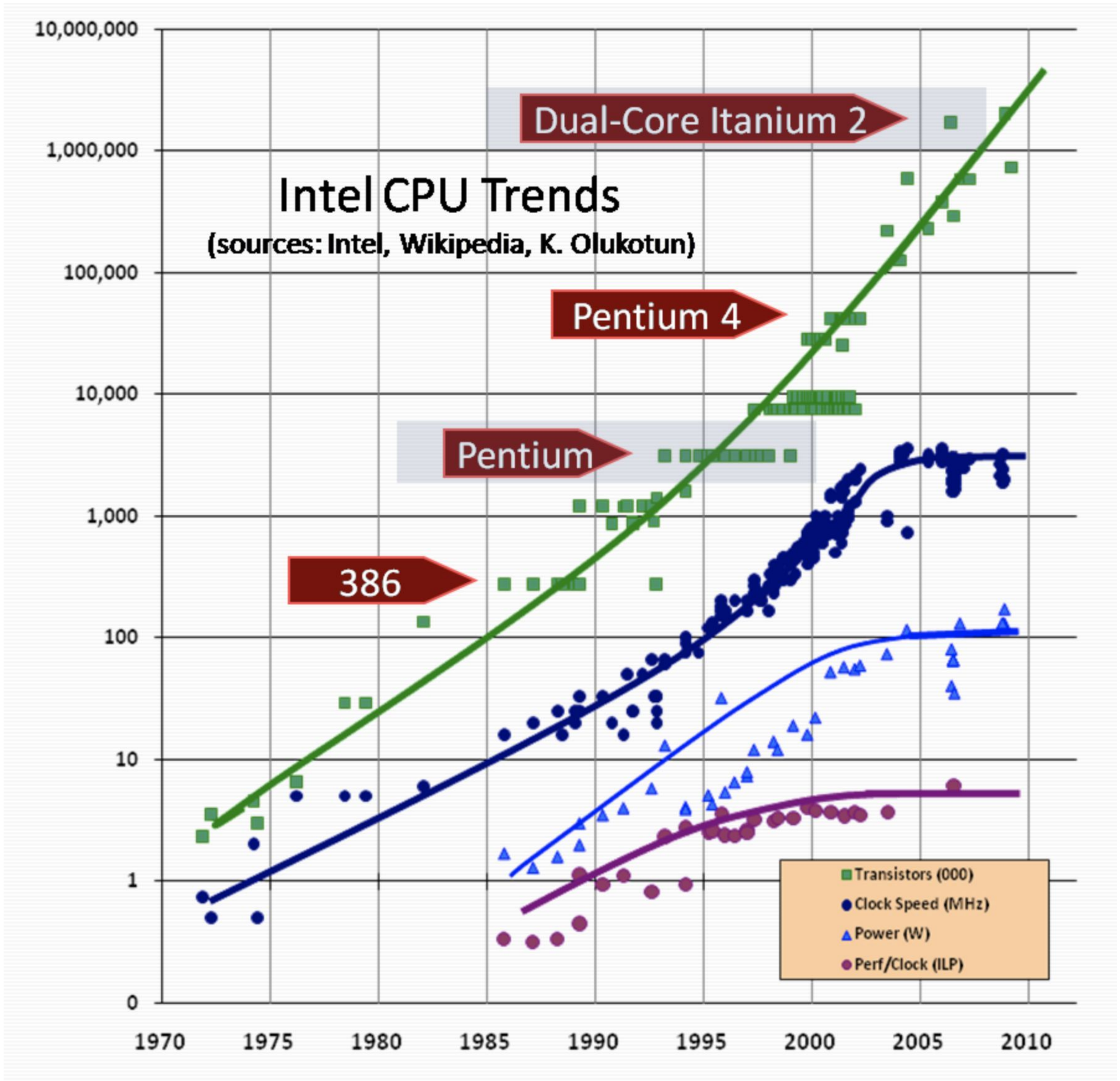
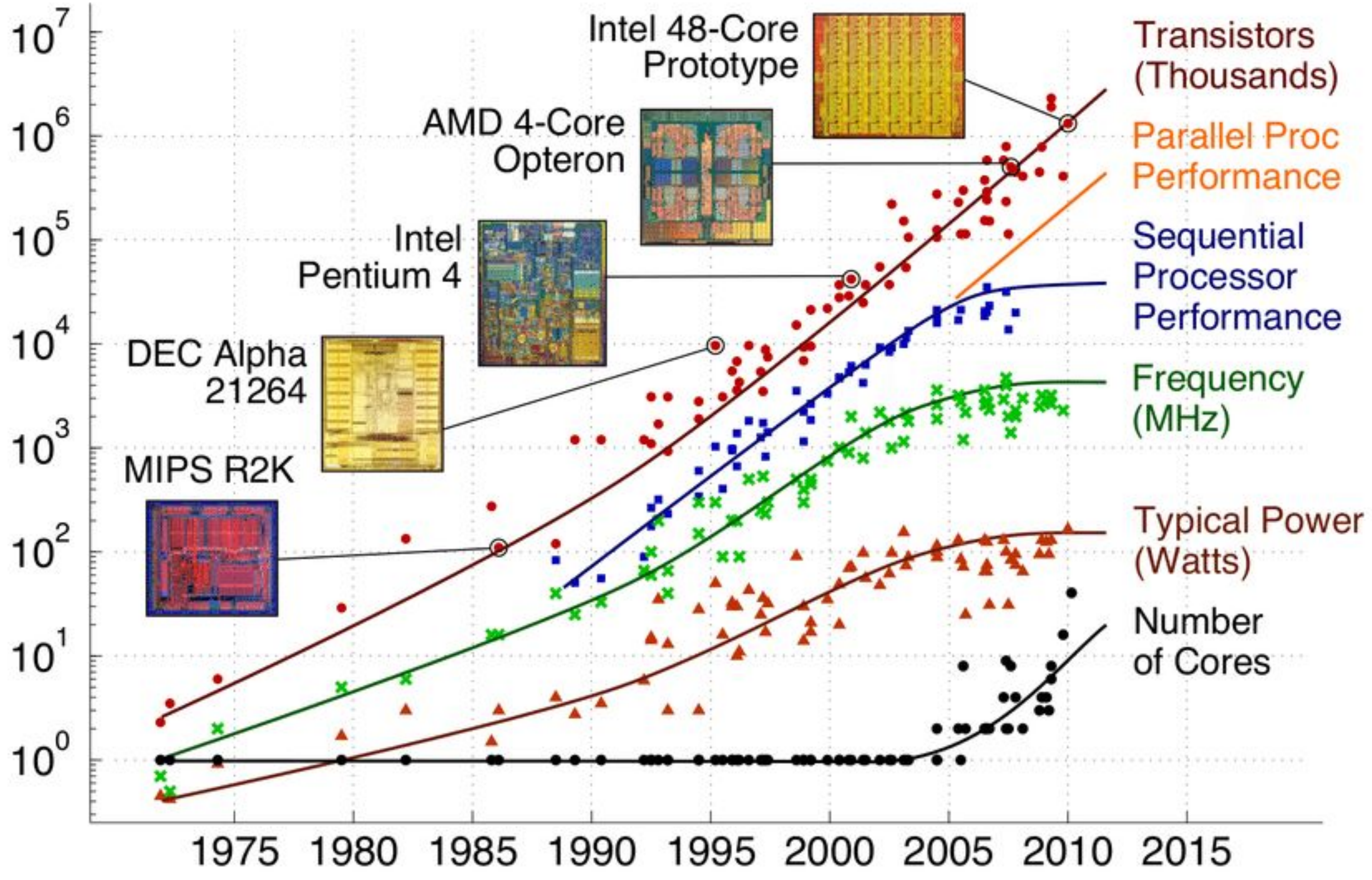


Figure 1: Intel CPU Introductions (graph updated August 2009; article text original from December 2004)

Source: Herb Sutter, “The Free Lunch is Over”, http://www.gotw.ca/publications/concurrency-ddj.htm

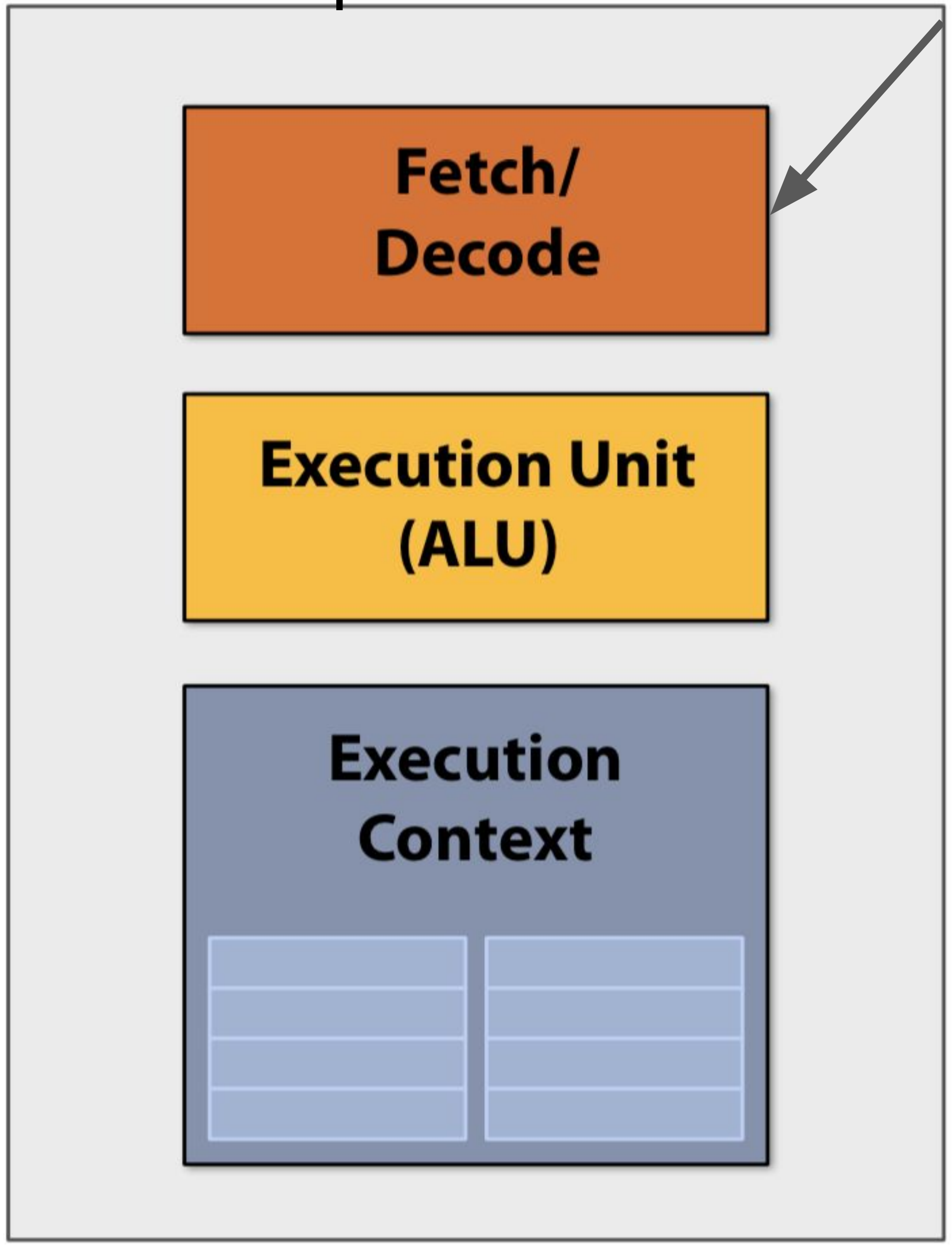
Why Parallelism?



Source: Pranav Tendulkar, 2014, https://www.researchgate.net/figure/Evolution-of-multi-core-processors-1_fig1_281534326

Data Level Parallelism (DLP)

Pipelined CPU

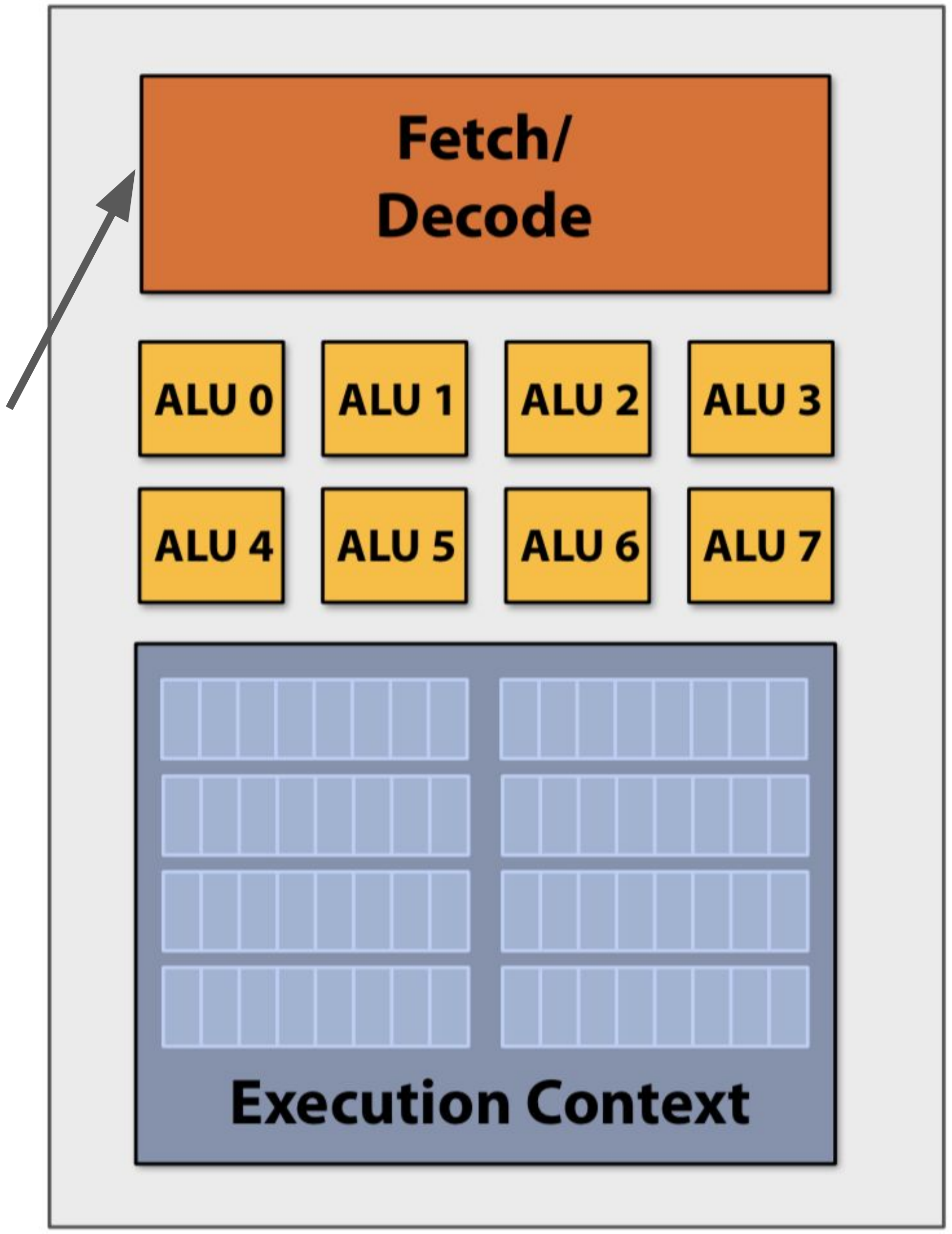


```
ldr r0, r1
ldr r2, r3
add r0, r2
```

1 instr/cycle

Single Instruction, Multiple Data
SIMD CPU

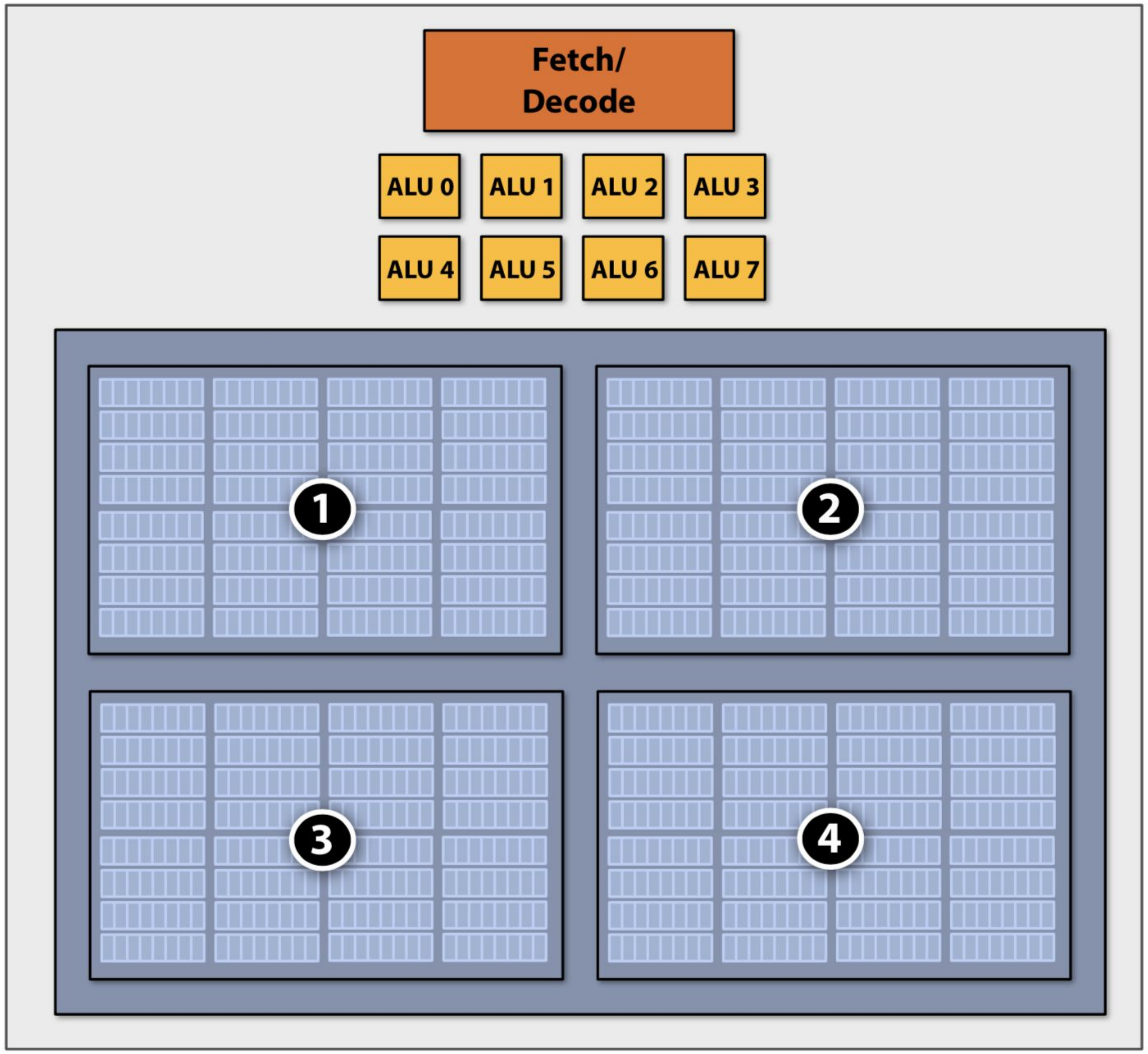
```
 vld v0, v1
 vld v2, v2
vadd v0, v2
```



1 instr/cycle
8 computations/cycle

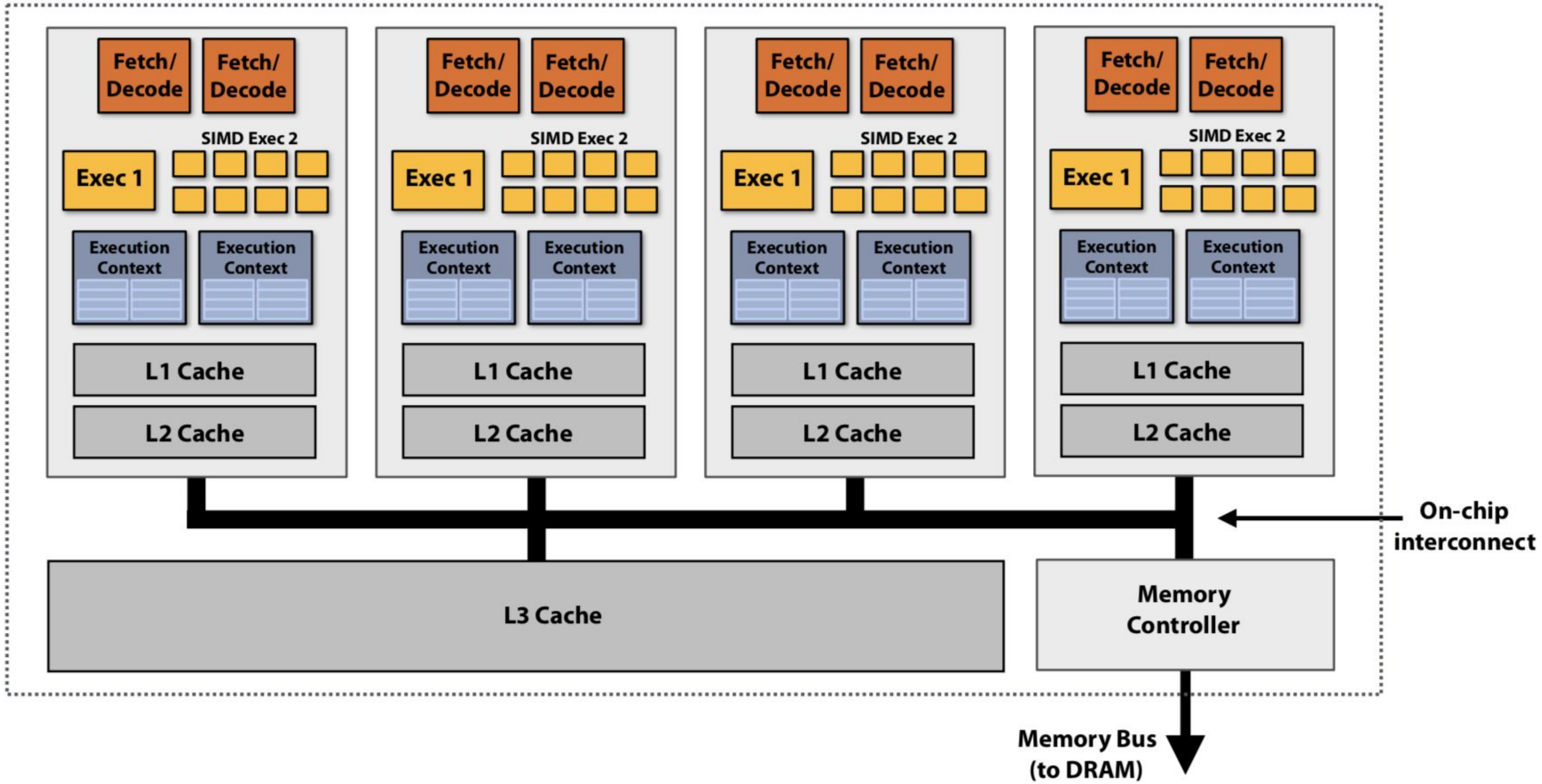
Thread Level Parallelism (TLP)

Multiple Execution Contexts = Hardware Multithreading

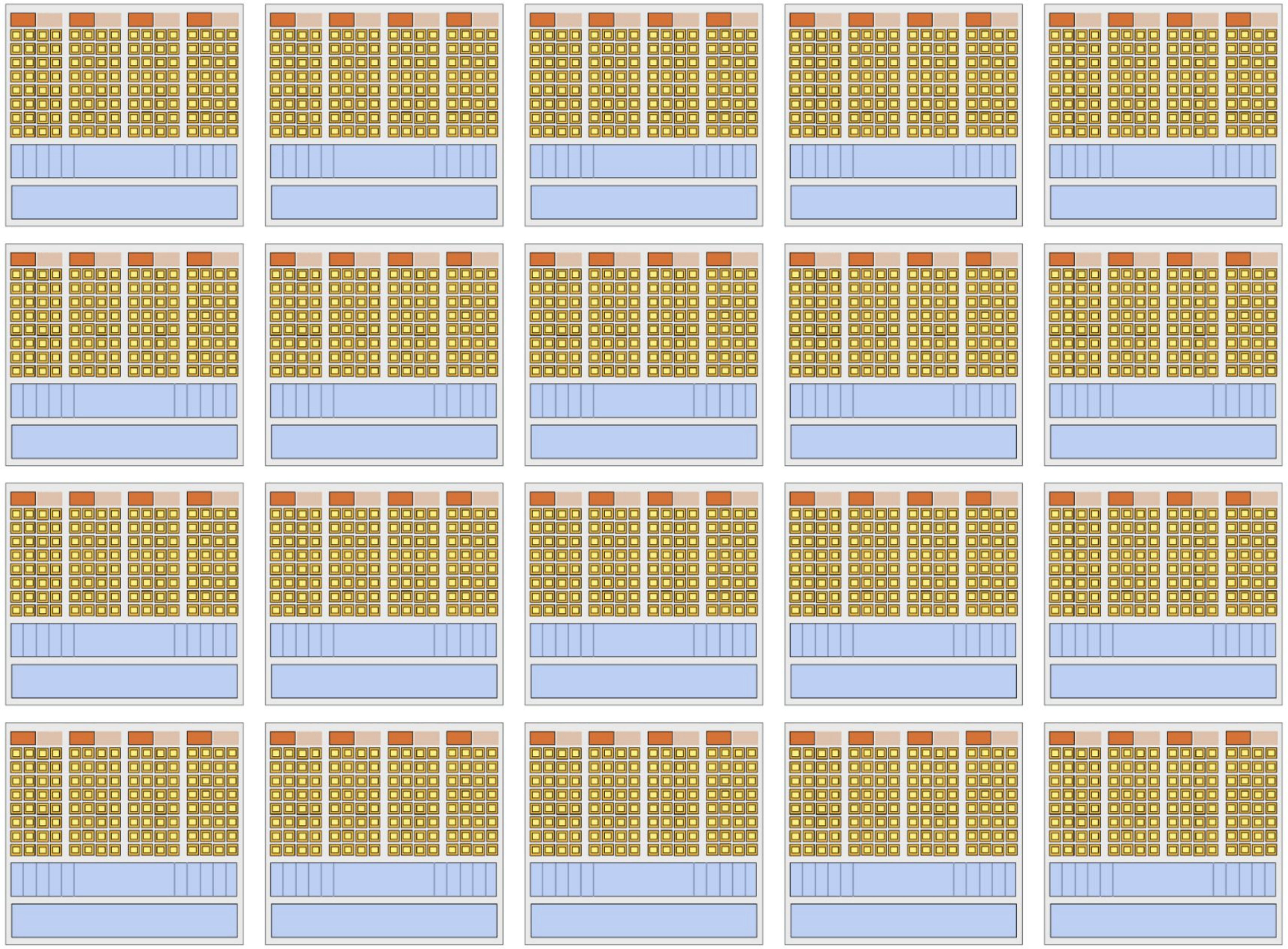


Multiple programs/threads can run together concurrently, but still have to share pipeline resources.

Modern Quad-Core CPU



Modern GPU (NVIDIA GTX 1080)



20 cores * 4 pipelines * 32 element SIMD ALUs = **2,560 parallel computations**

* 2 FLOPs (mult-add) * 1.6 Ghz = **8.1 TERAFLOPS**

Architecture Summary

Hardware makes optimizations for instruction latency & throughput.

Memory latency is very significant.

Single-thread performance is not increasing drastically on chips anymore.

Multicore chips and application-specific processors are rising.

Software/algorithm needs to match the architecture (or vice versa).