CS 310

CONCURRENCY

ADMINISTRIVIA

- **Shell homework due Wednesday 9/10 11:59 PM.**

REVIEW FROM LAST TIME: CREATING A NEW PROCESS

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

  printf("Both!\n");
  return 0;
}
```

PRACTICAL MOTIVATION FOR CONCURRENCY

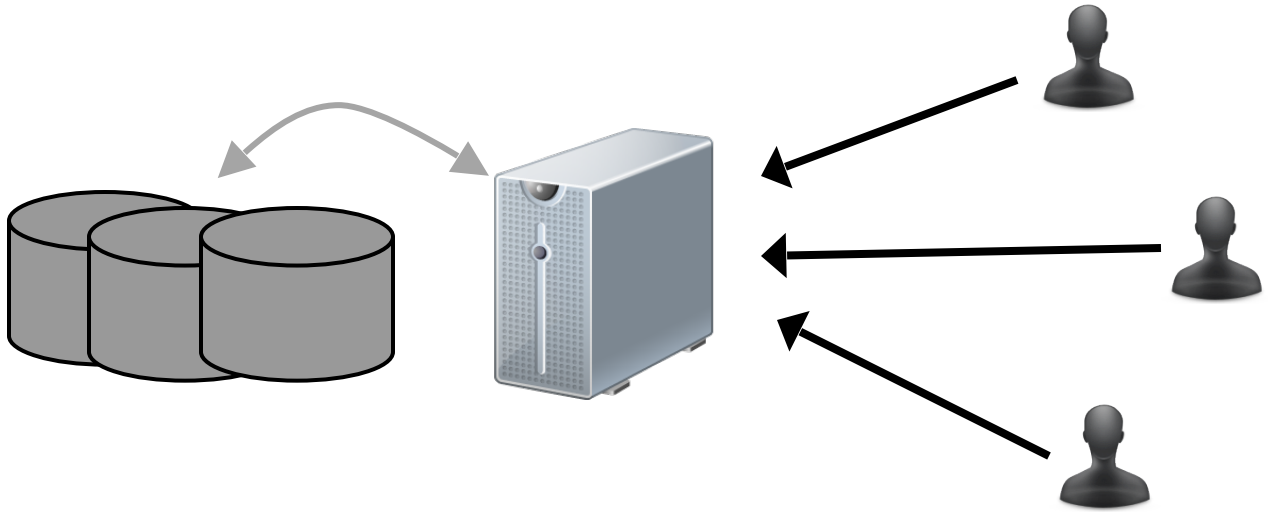
Back to Jeff Dean's "Numbers Everyone Should Know"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

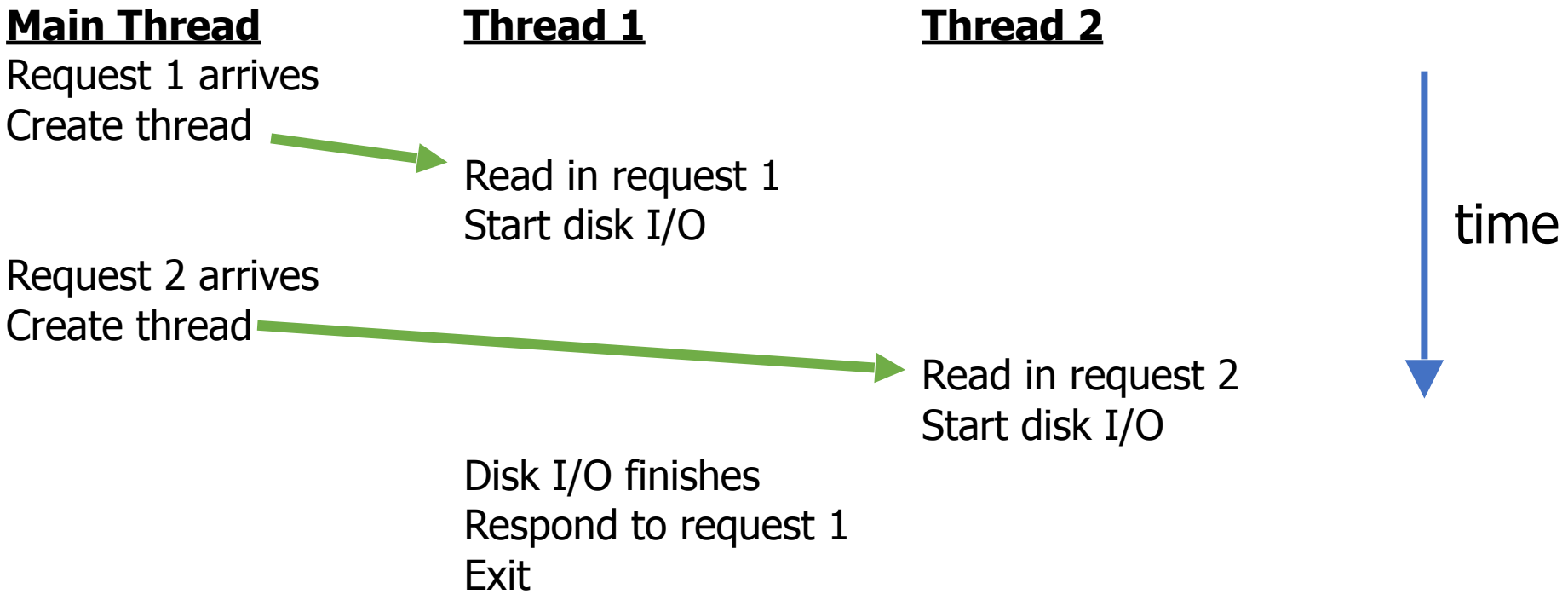
Handle I/O in separate thread, avoid blocking other progress

THREAD USE CASE: WEB SERVER

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



MULTI-THREADED WEB SERVER



- One thread per request. Thread handles only that request.
- Easy to program (maybe), and fast!
  - State is stored in the stacks of each thread and the thread scheduler
  - Simple to program if they are independent…

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

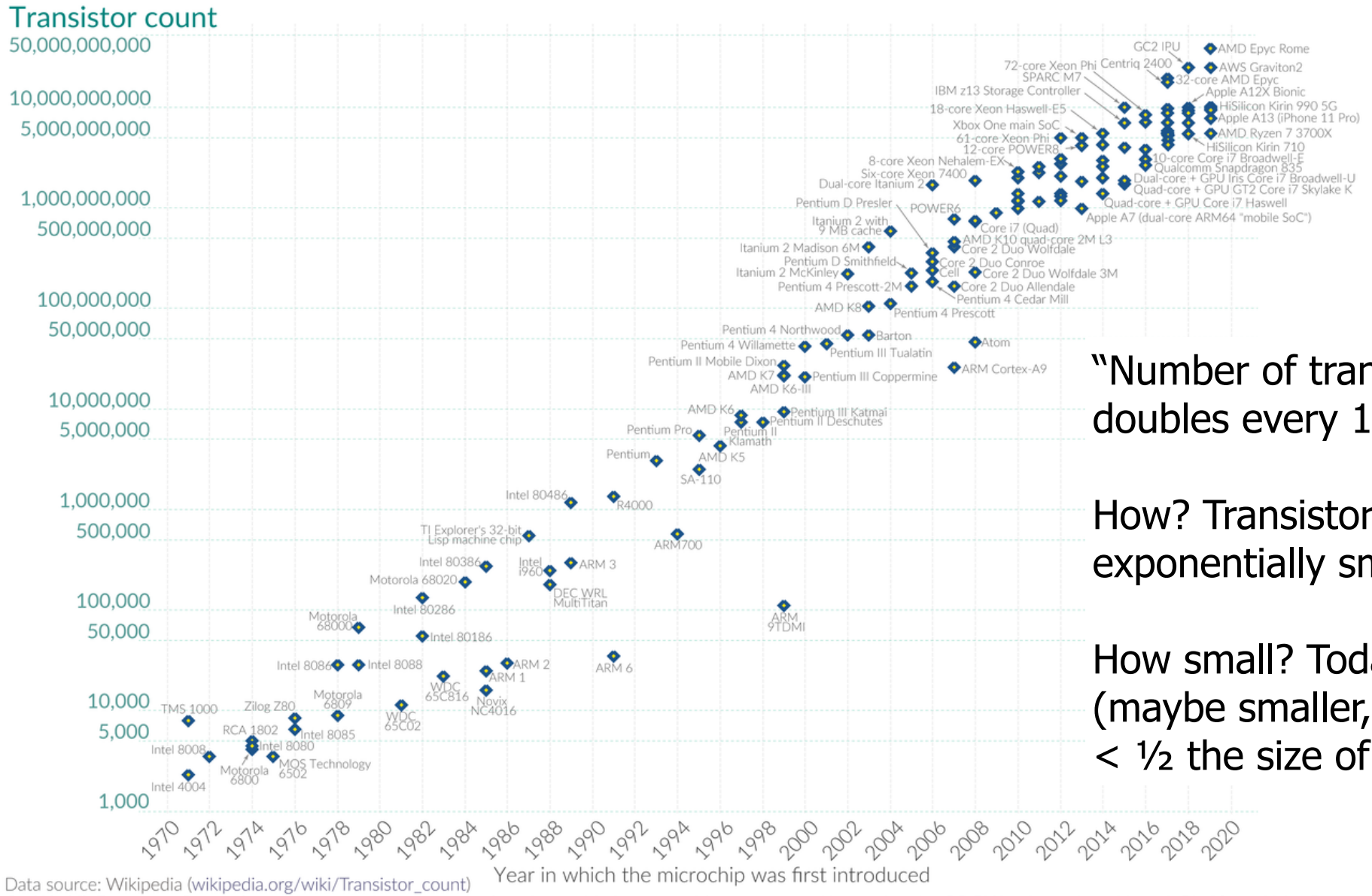
What do you do?

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

What do you do? **Take a vacation**

MOORE'S LAW – CPU TRANSISTORS COUNTS



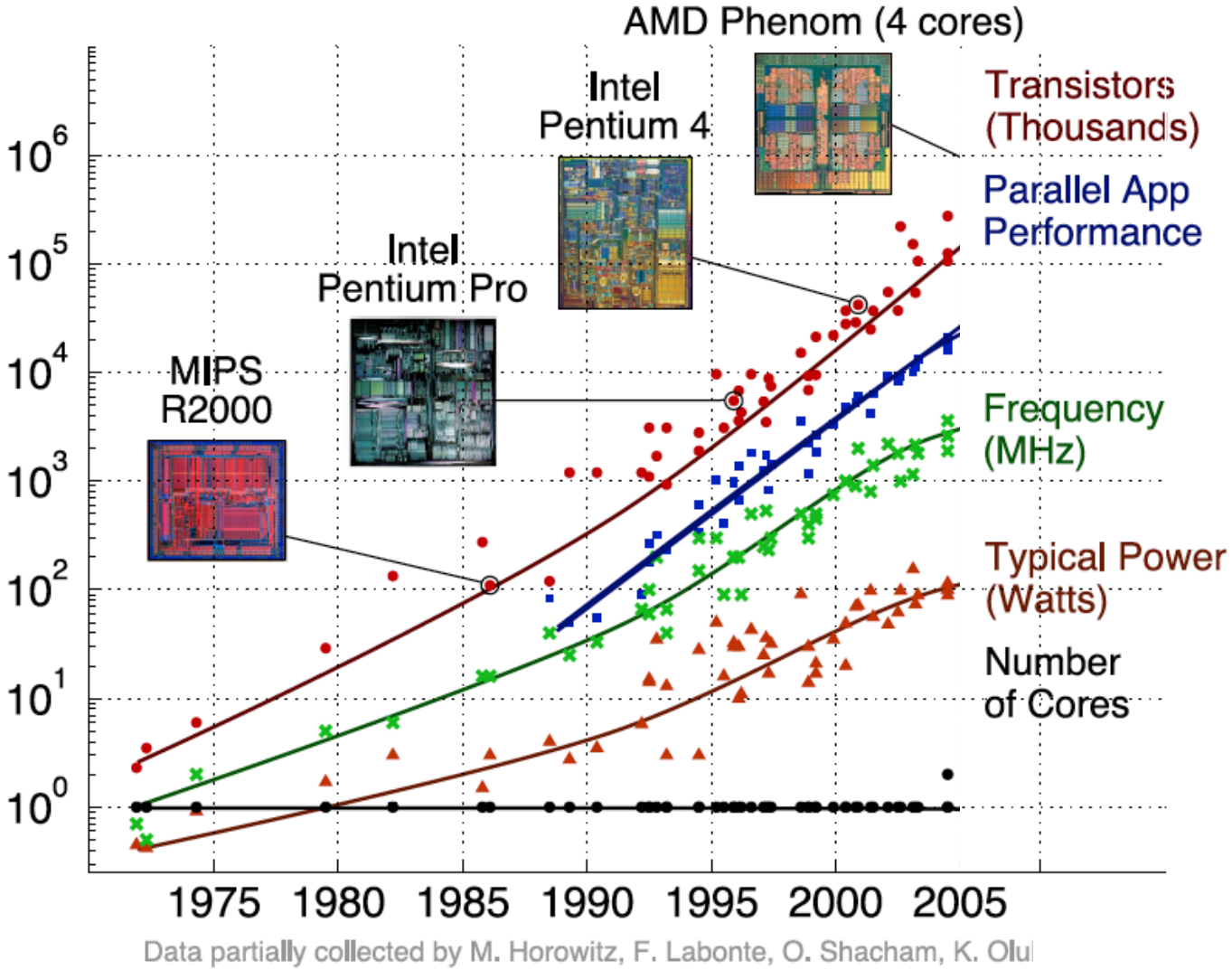
"Number of transistors in a chip doubles every 18 months"

How? Transistors are getting exponentially smaller!

How small? Today: 7nm!
(maybe smaller, kind of complicated)
< ½ the size of most viruses!

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

PROCESSORS KEPT GETTING FASTER TOO



POWER IS A MAJOR LIMITING FACTOR ON SPEED

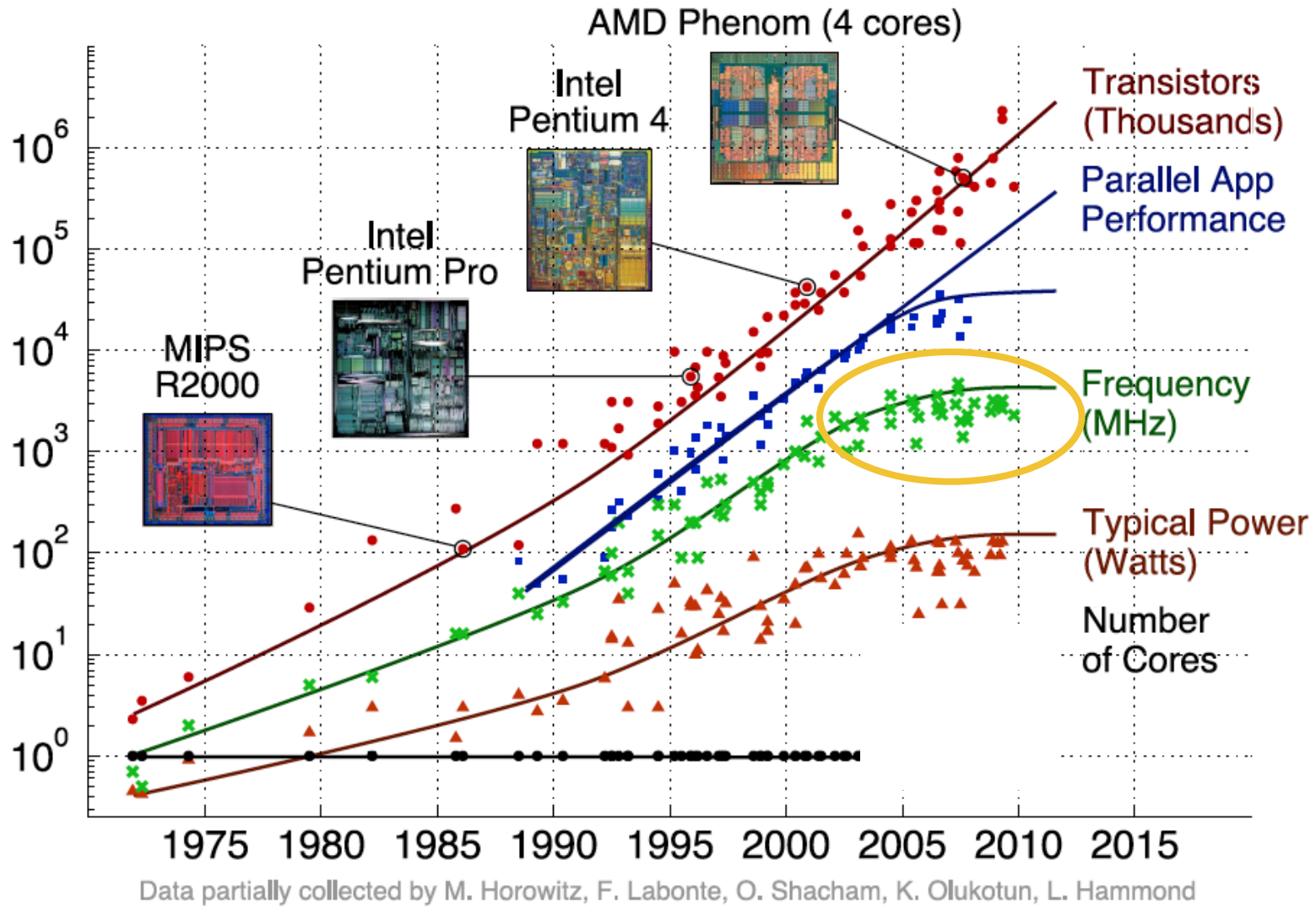
- We could make processors go very fast
  - But doing so uses more and more power
- More power means more heat generated
  - And chips typically work up to around 100°C
  - Hotter than that and things stop working
- We add heat sinks and fans and water coolers to keep chips cool
  - But it's hard to remove heat quickly enough from chips
- So, power consumption ends up limiting processor speed

DENARD SCALING

- Moore's Law corollary: Denar Scaling
  - As transistors get smaller, the power density stays the same
  - Which is to say that the power-per-transistor decreases!

- Making the processor clock speed faster uses more power
  - But the two balance out for roughly net even power
  - So not only do we get more transistors, but chip speed can be faster too
- From our Excel example:
  - In two years, new hardware would run the existing software twice as fast

THEN THEY STOPPED GETTING FASTER



~2006: Leakage current becomes significant

Now smaller transistors doesn't mean lower power

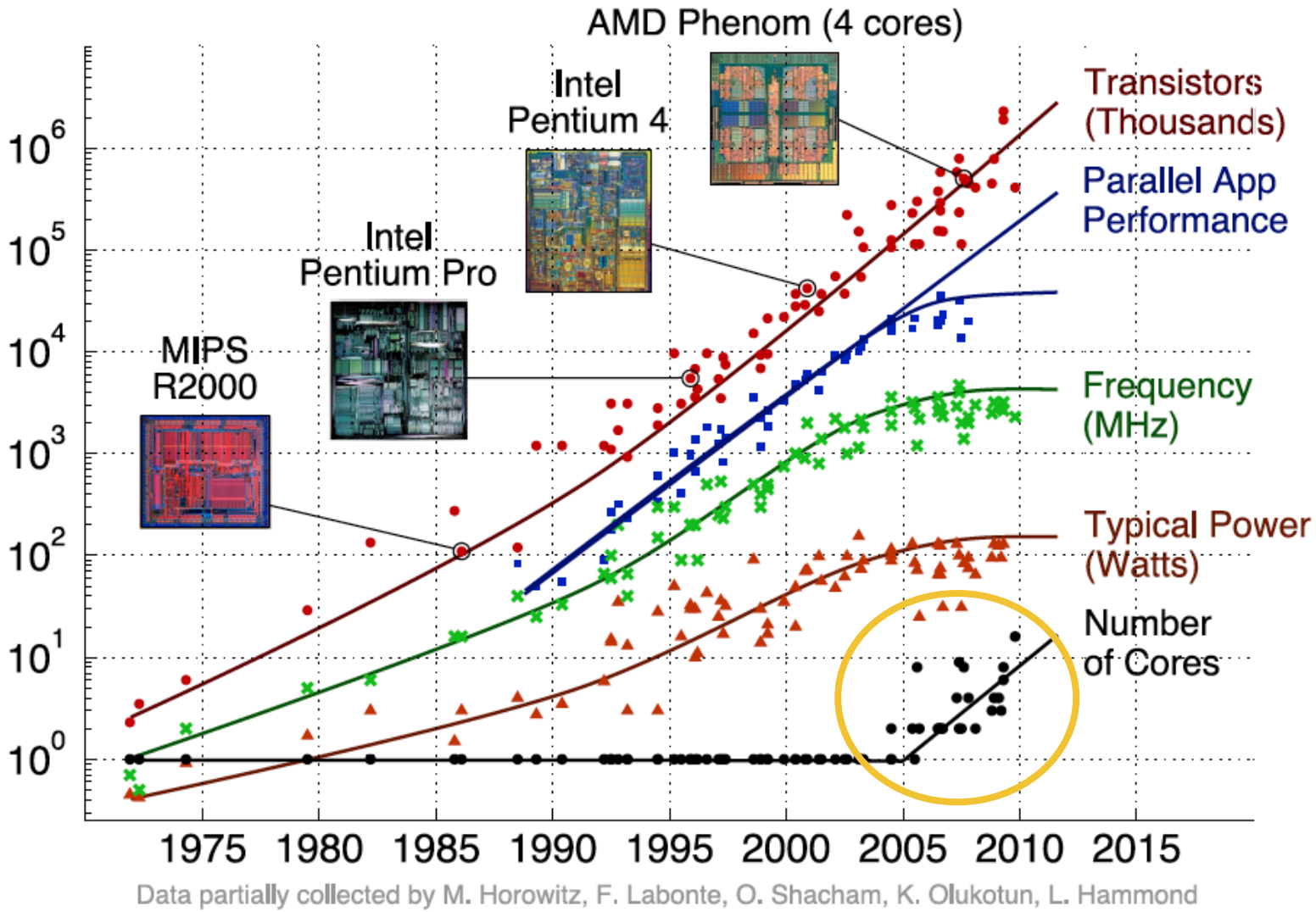
SO… NOW WHAT?

In summary:

- Making transistors smaller doesn't make them lower power,
- so if we were to make them faster, they would take more power,
- which will eventually lead to our processors melting…
- and because of that, we can't reliably make performance better by waiting for clock speeds to increase.

How do we continue to get better computation performance?

EXPLOIT PARALLELISM!



PARALLELISM ANALOGY

- I want to peel 100 potatoes as fast as possible:
  - I can learn to peel potatoes faster

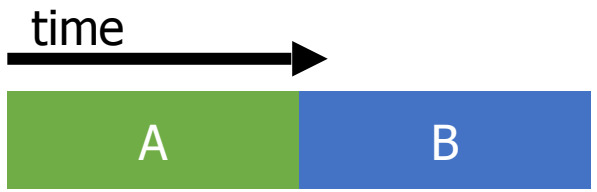
  OR

  - I can get 99 friends to help me
- Whenever one result doesn't depend on another, doing the task in parallel can be a big win!

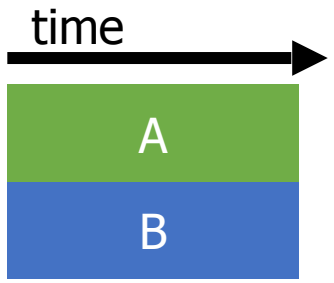
Parallelism versus Concurrency

Two processes A and B

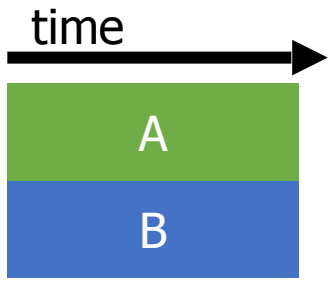
Serial execution



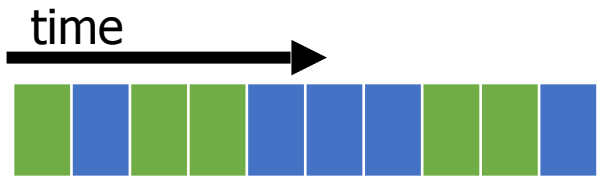
Parallel execution



Concurrent execution



OR

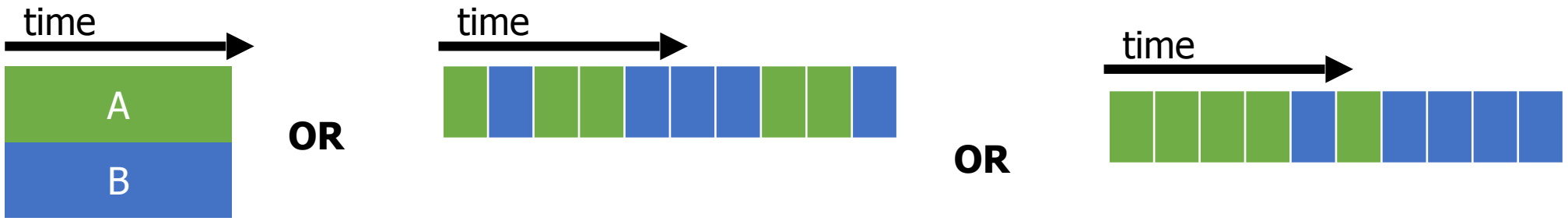


PARALLELISM VERSUS CONCURRENCY

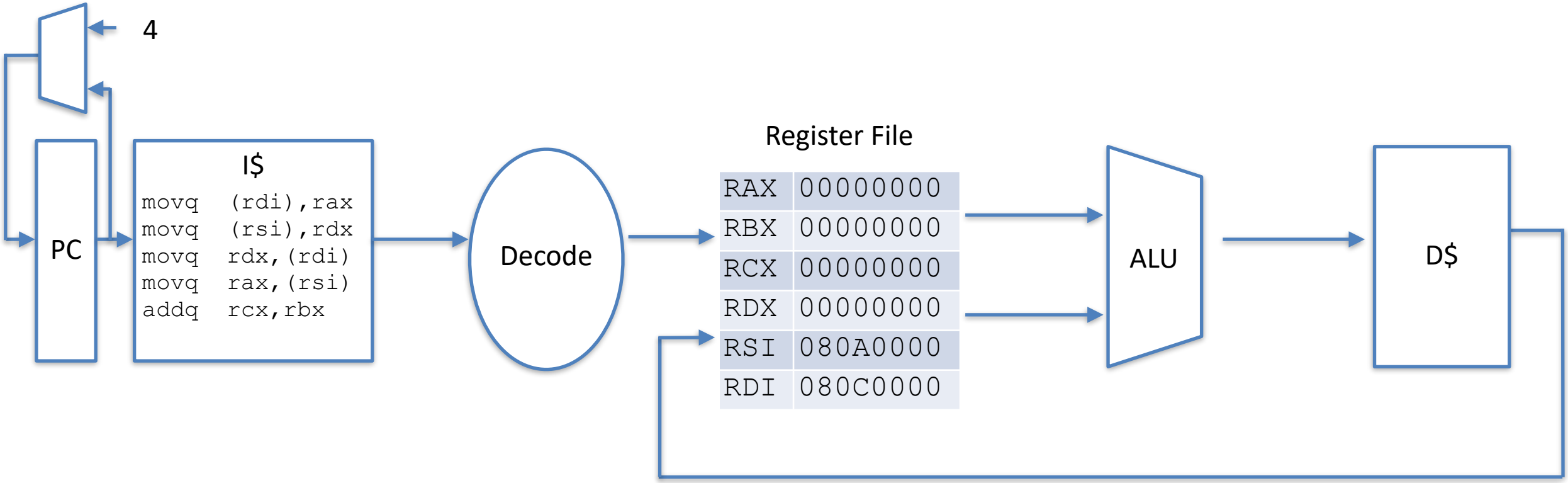
- Parallelism
  - Two things happen strictly simultaneously

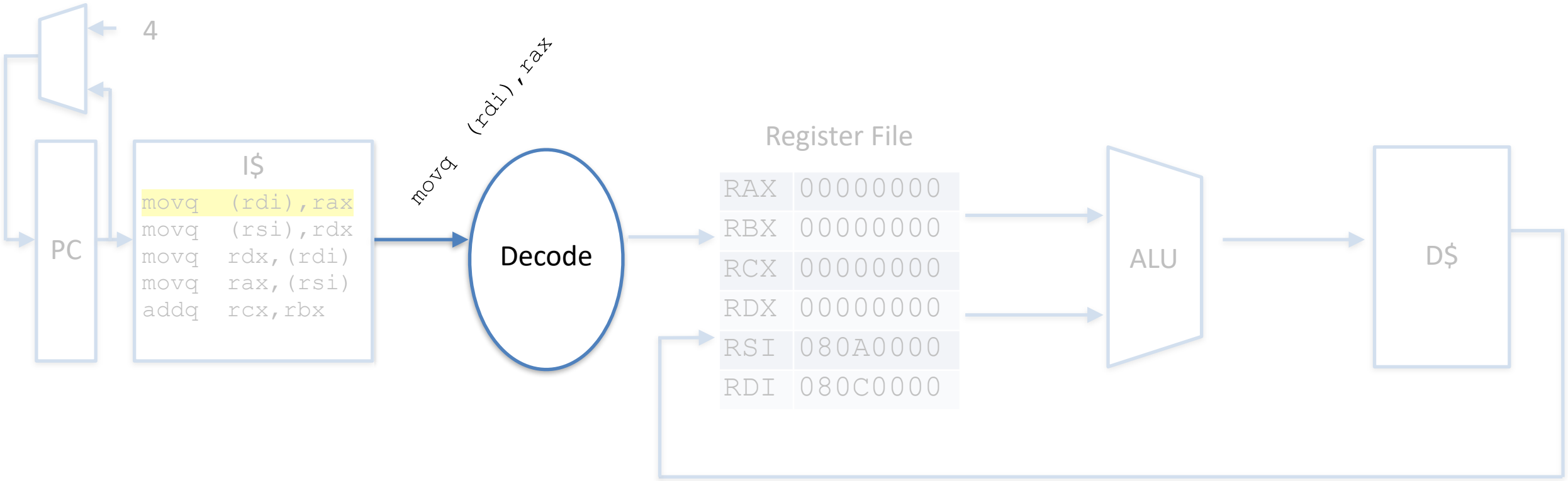
- Concurrency
  - More general term
  - Two things happen in the same time window
    - Could be simultaneous, could be interleaved

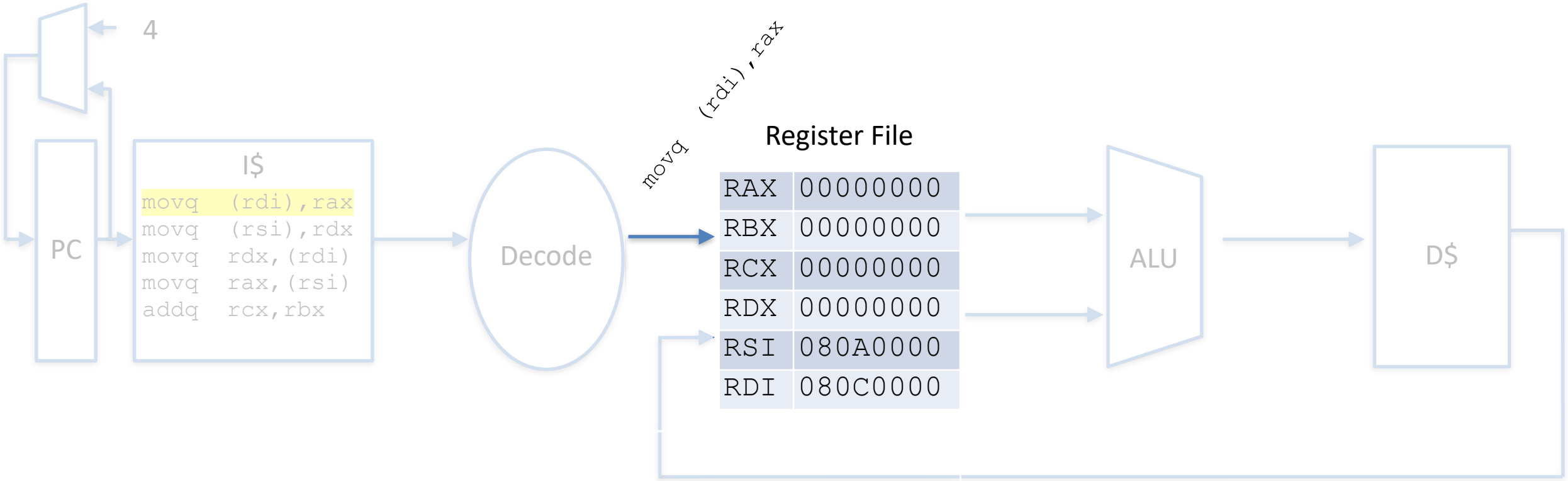
  - Concurrent execution occurs whenever two processes are both active

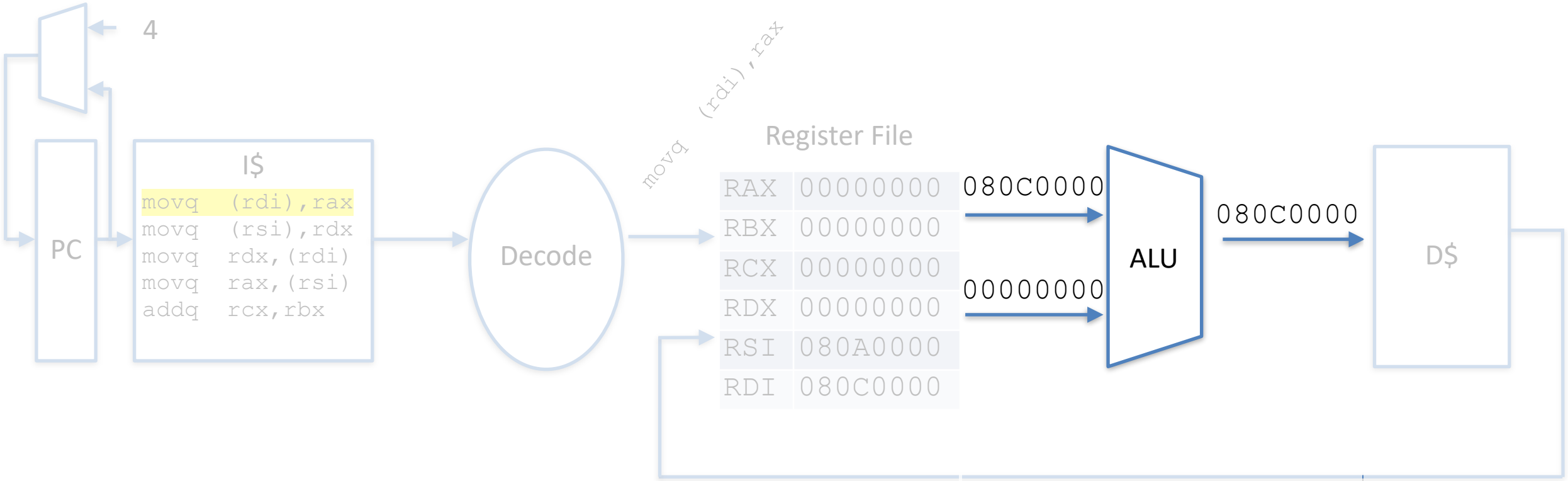


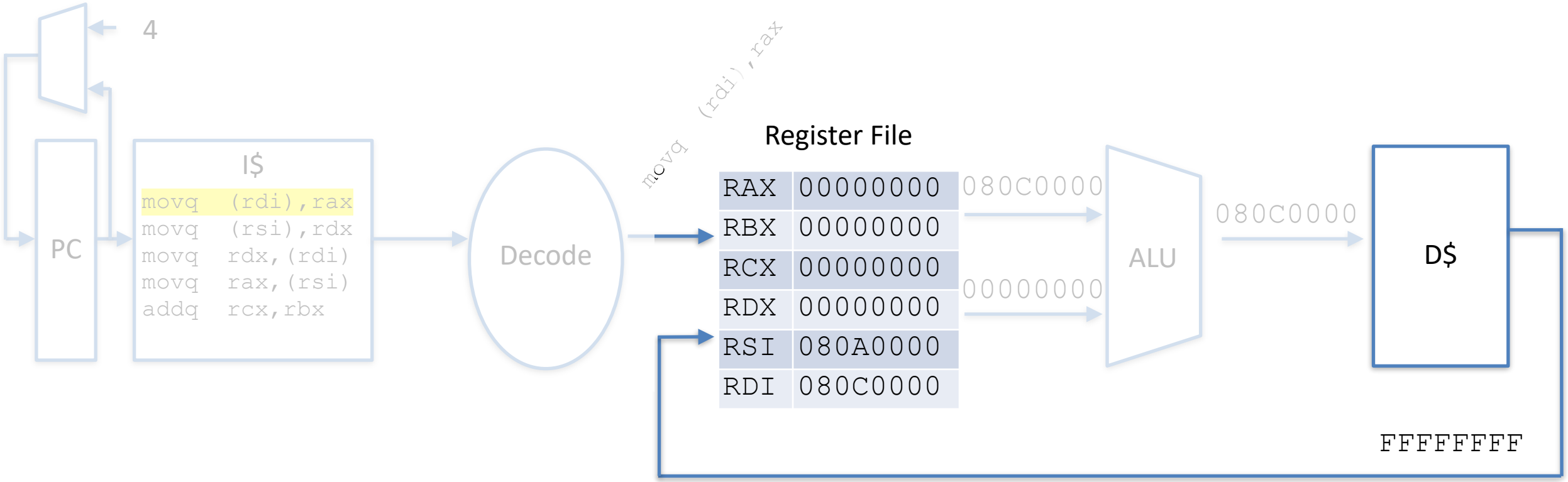
CONCURRENCY IN HARDWARE

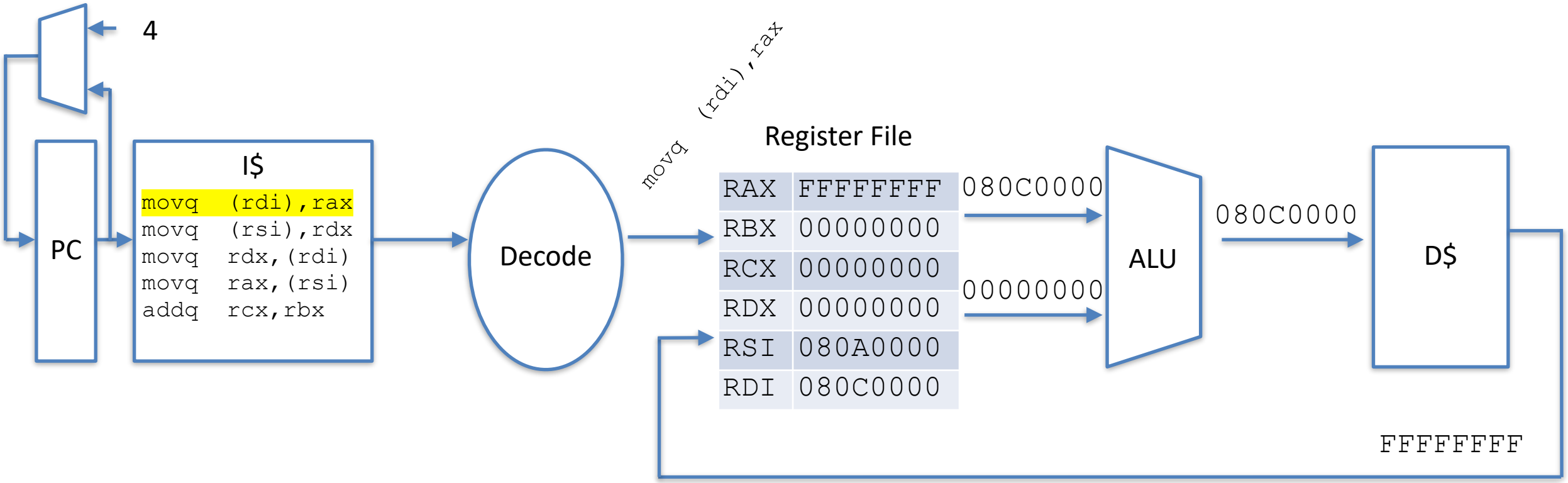

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

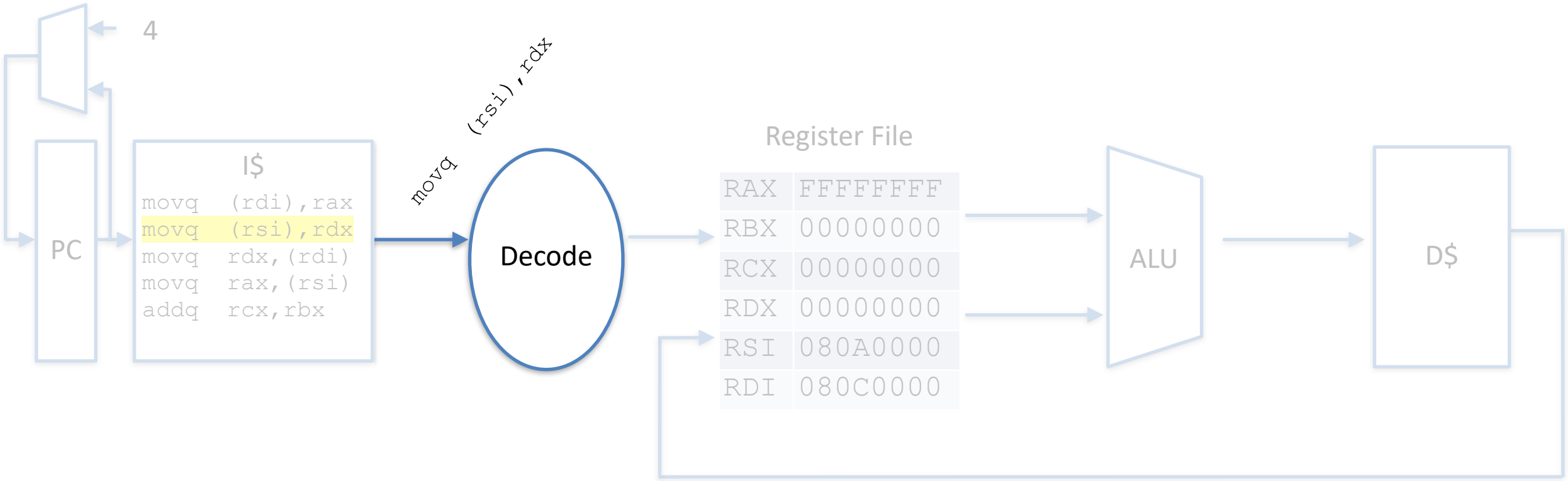

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
movq (rdi),rax
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

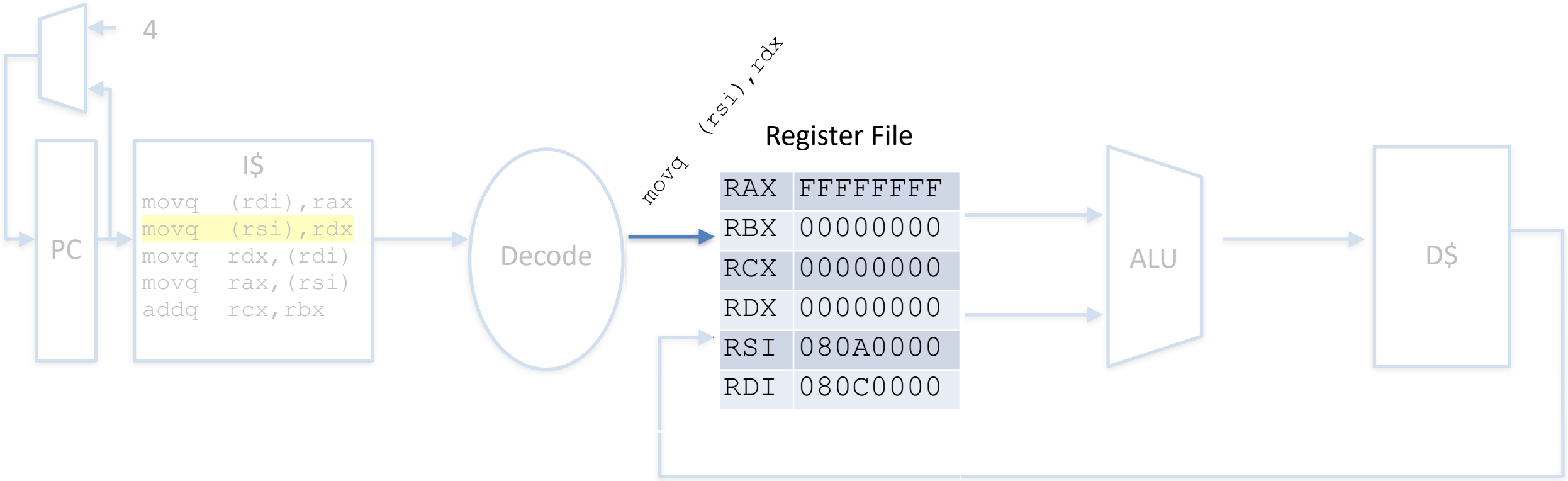

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rdi),rax
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

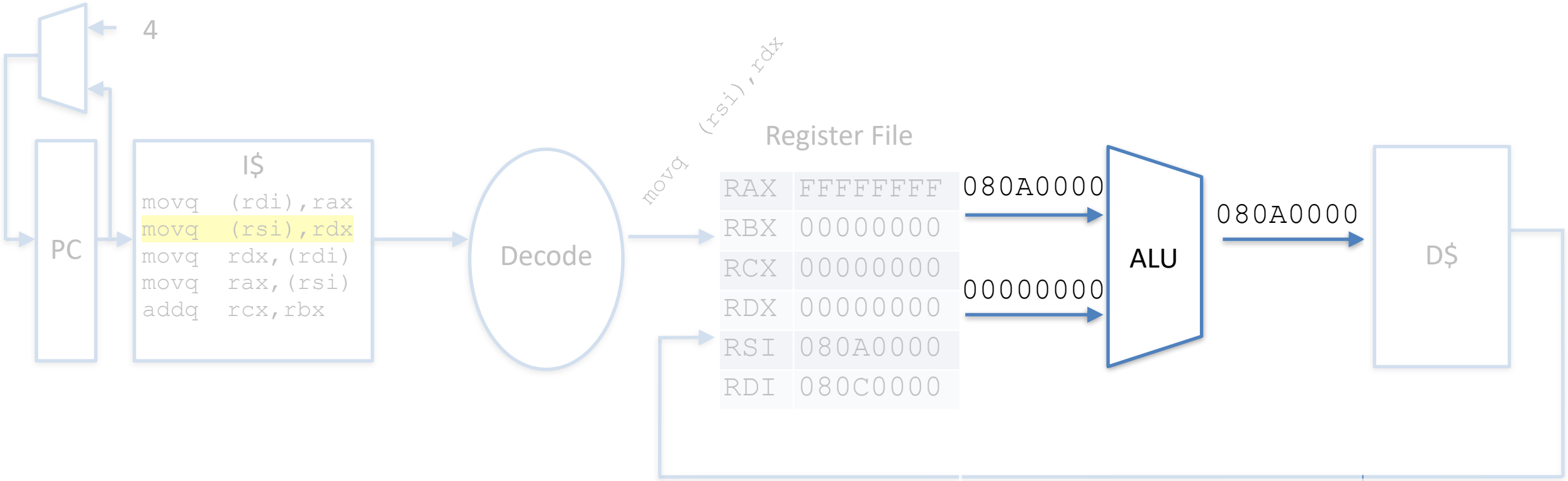

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rdi),rax
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
080C0000
00000000
ALU
080C0000
D$

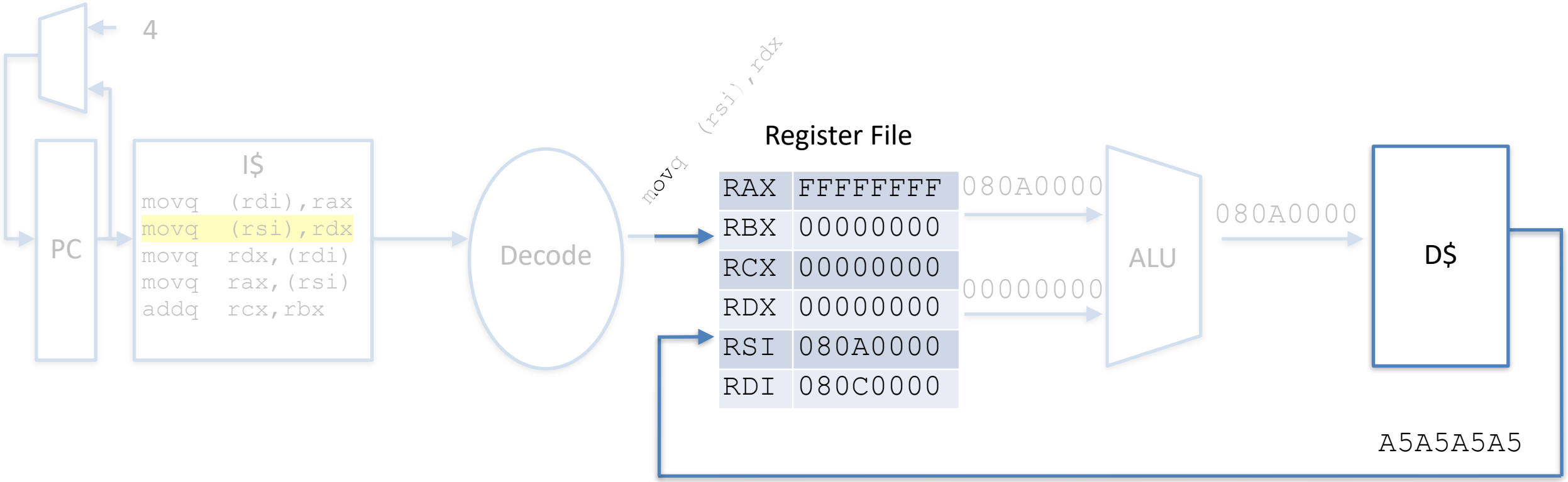

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rdi),rax
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
080C0000
00000000
ALU
080C0000
D$
FFFFFFFF

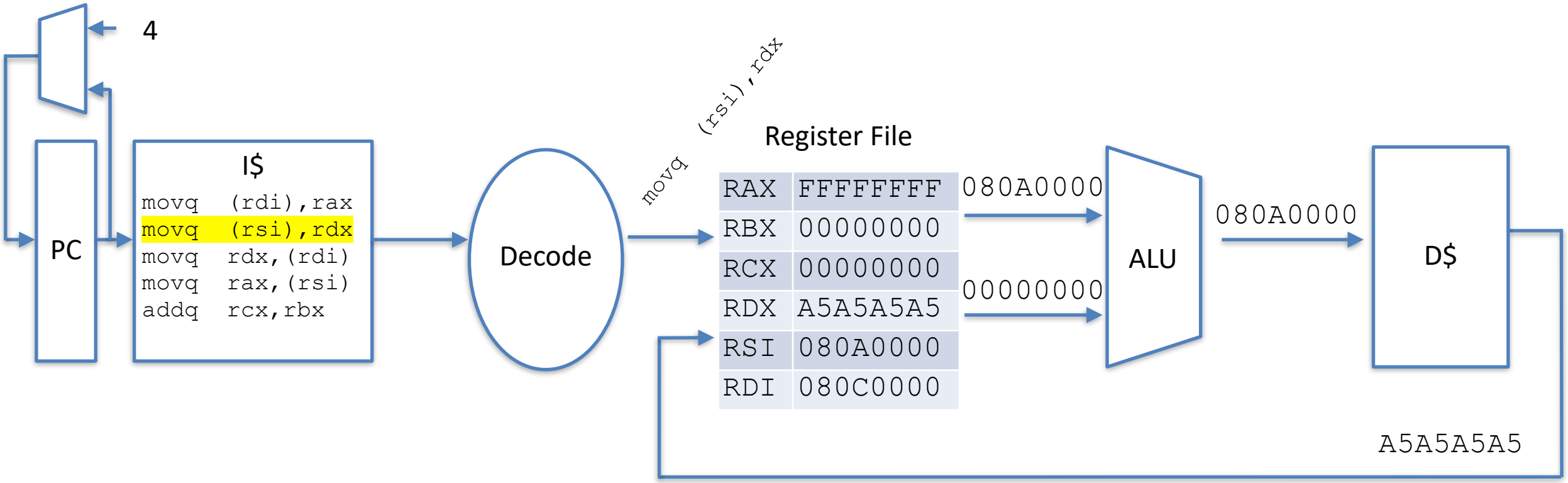

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rdi),rax
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
080C0000
00000000
ALU
080C0000
D$
FFFFFFFF


4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
movq (rsi),rdx
Decode
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

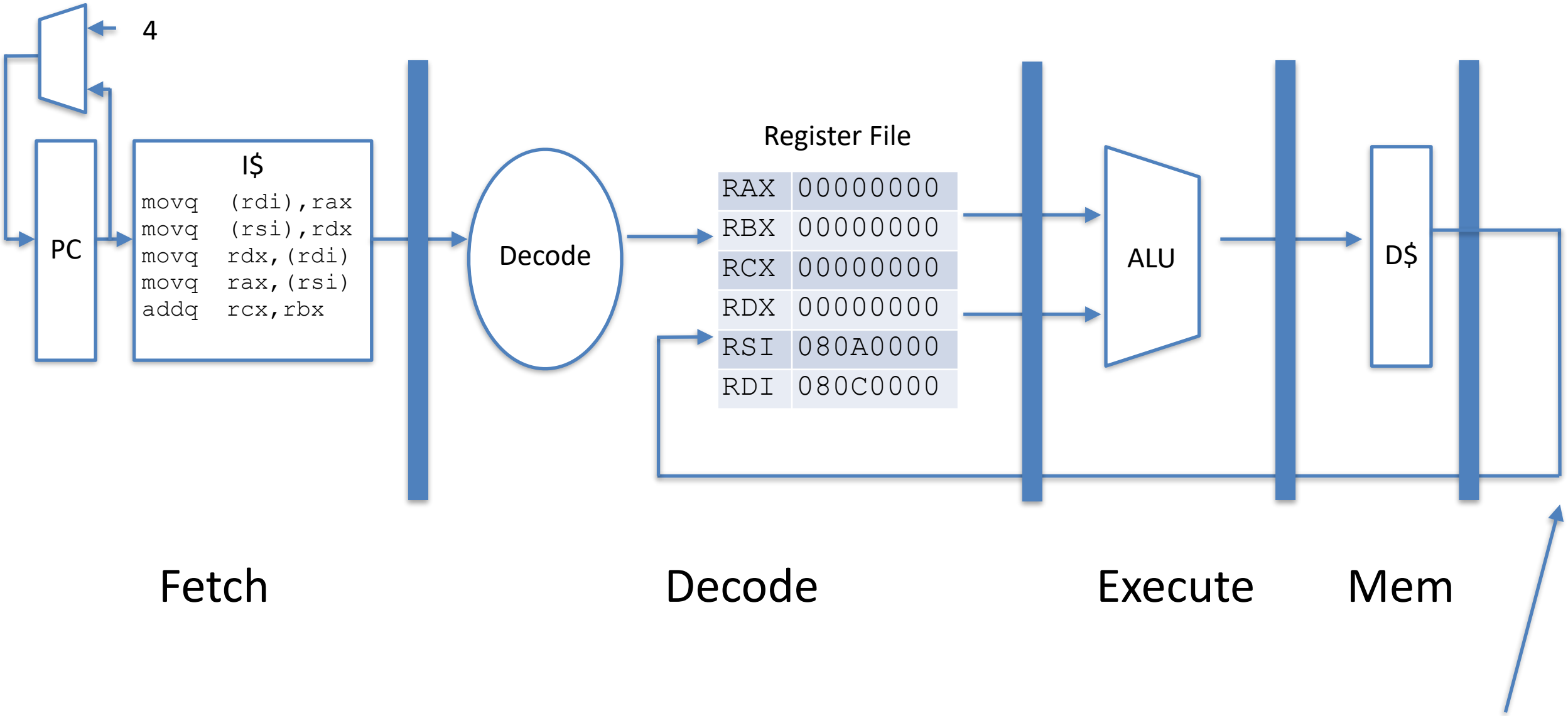

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rsi),rdx
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

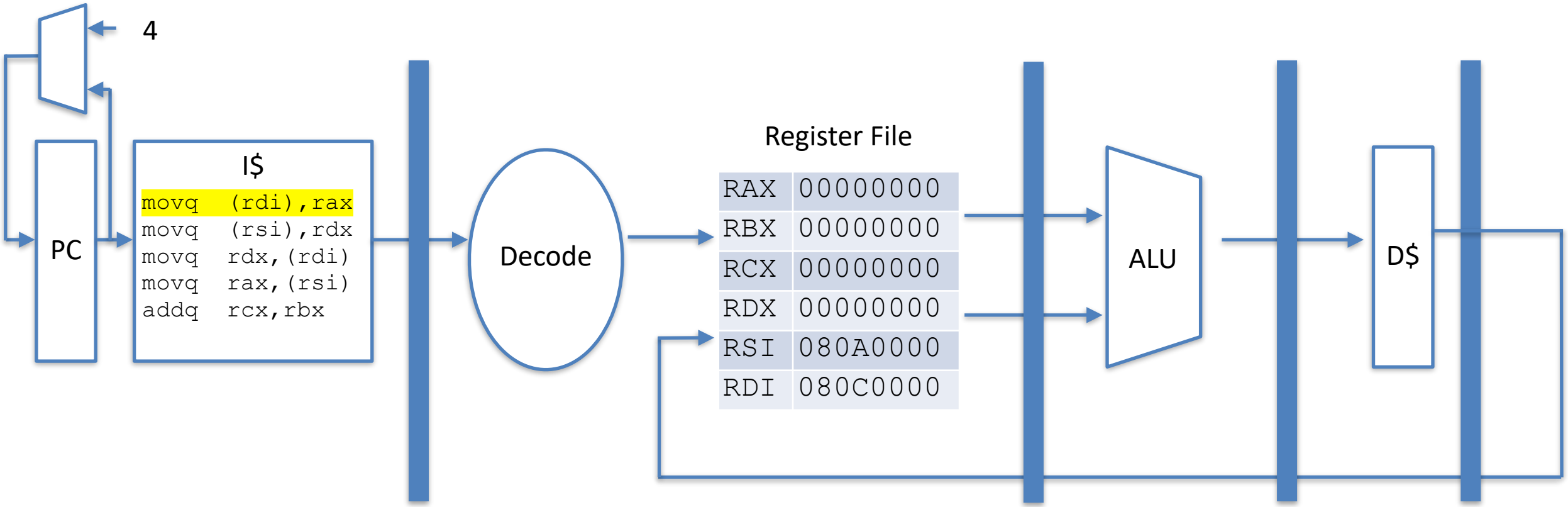

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rsi),rdx
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
080A0000
00000000
ALU
080A0000
D$

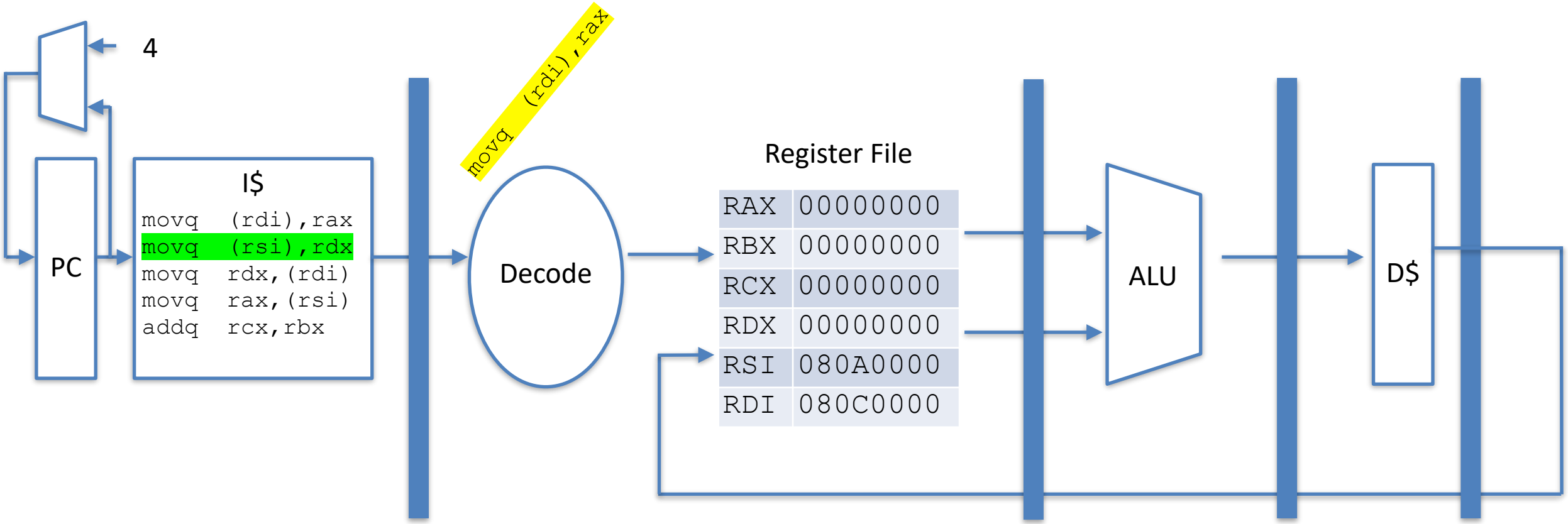

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rsi),rdx
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
080A0000
00000000
ALU
080A0000
D$
A5A5A5A5

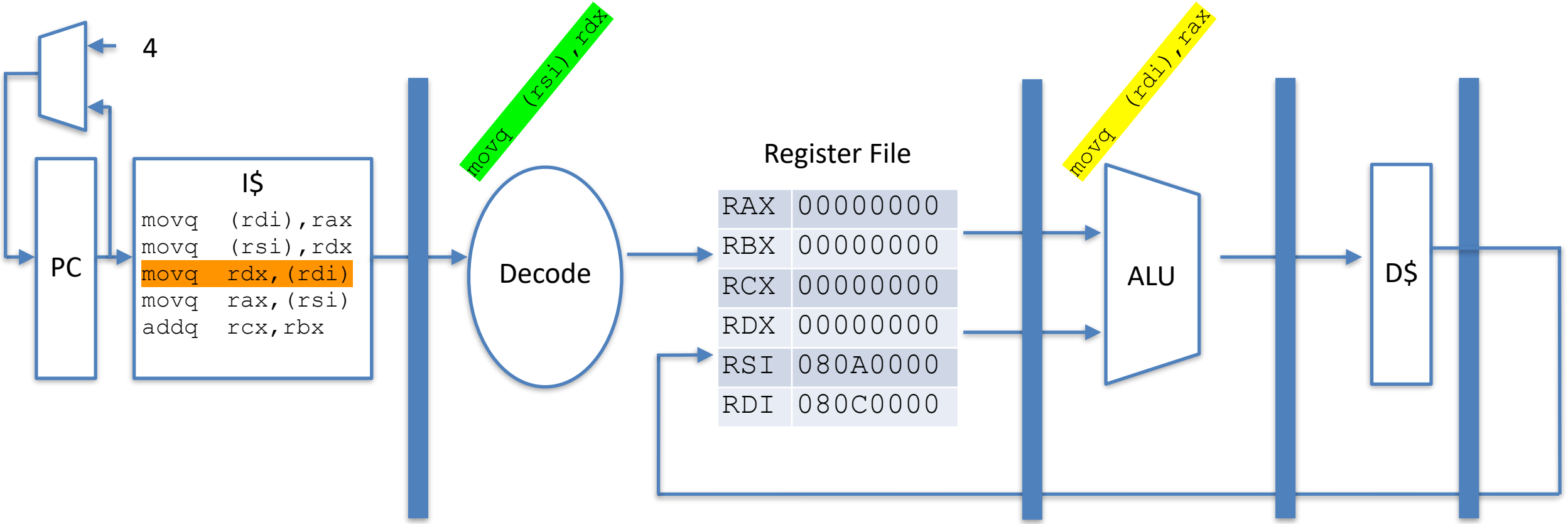

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq (rsi),rdx
Register File
RAX FFFFFFFF
RBX 00000000
RCX 00000000
RDX A5A5A5A5
RSI 080A0000
RDI 080C0000
080A0000
00000000
ALU
080A0000
D$
A5A5A5A5

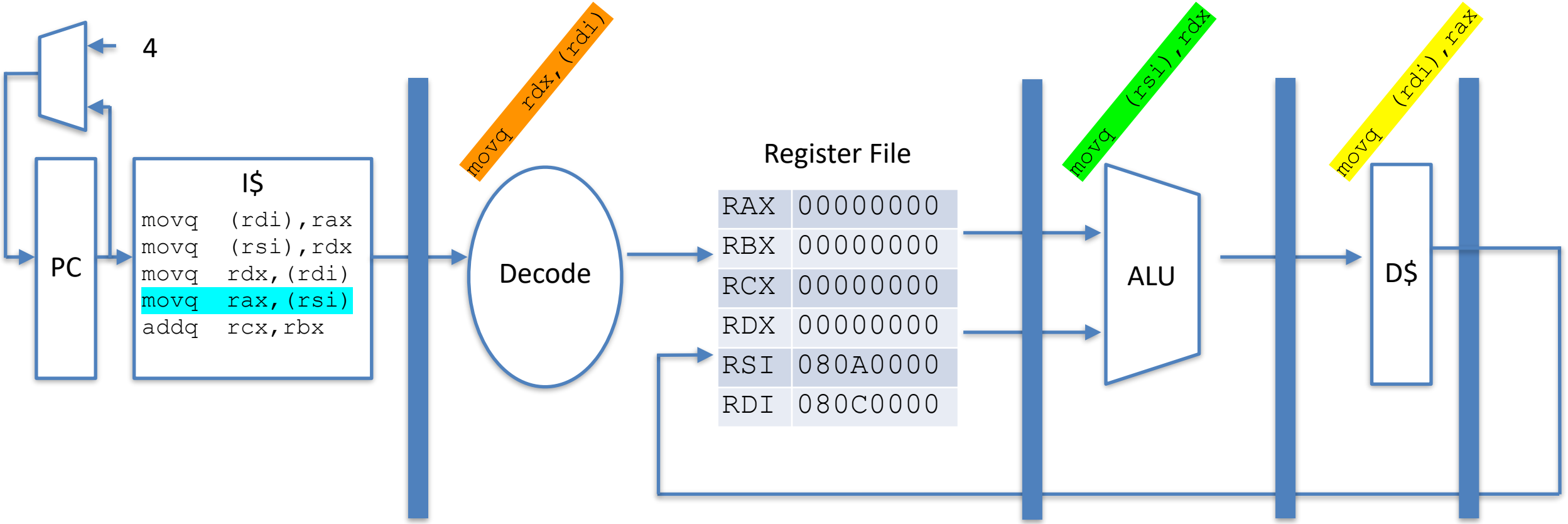
HARDWARE PARALLELISM

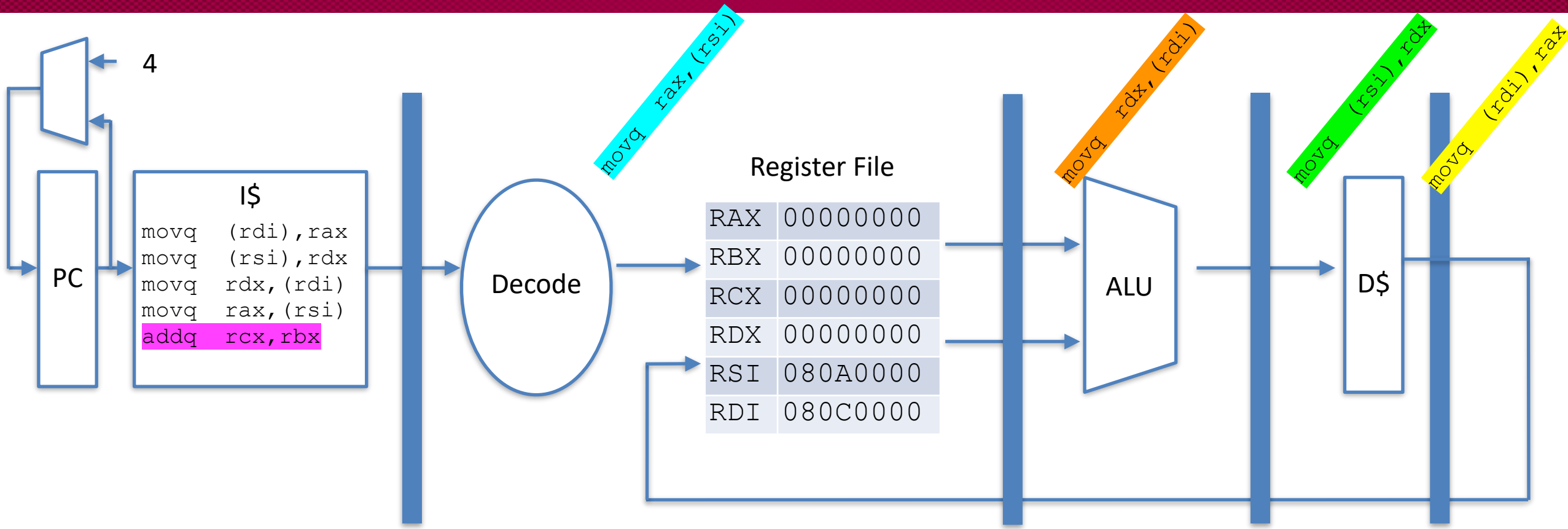

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$
Fetch
Decode
Execute
Mem
Writeback

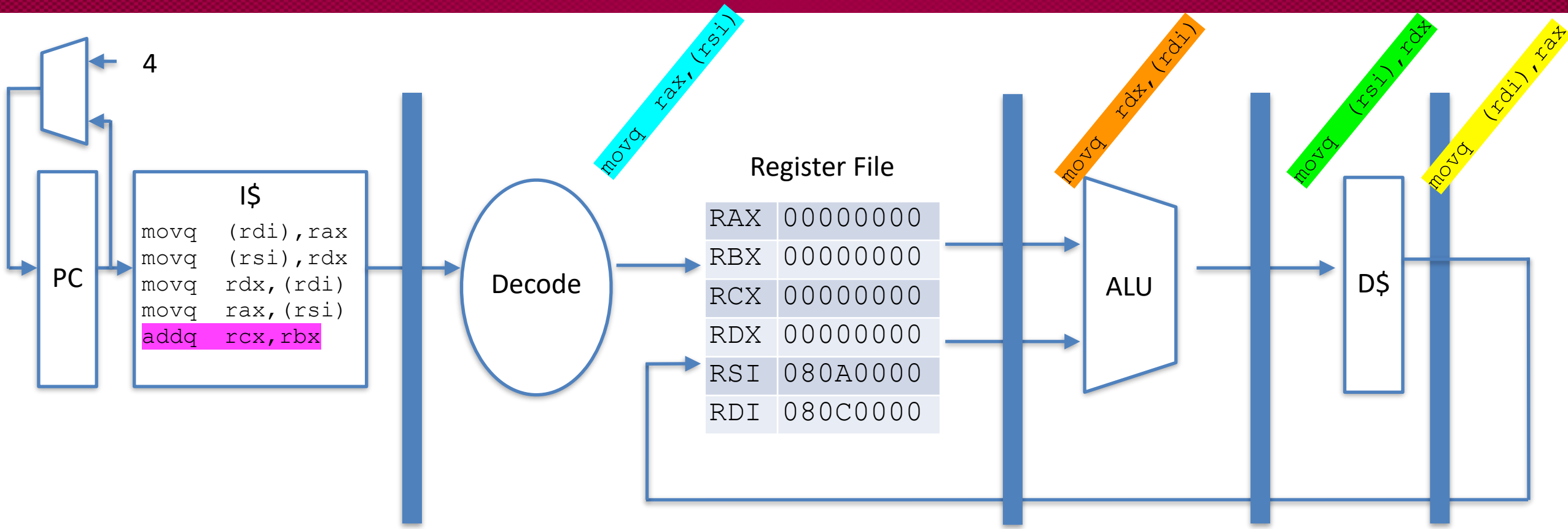

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$

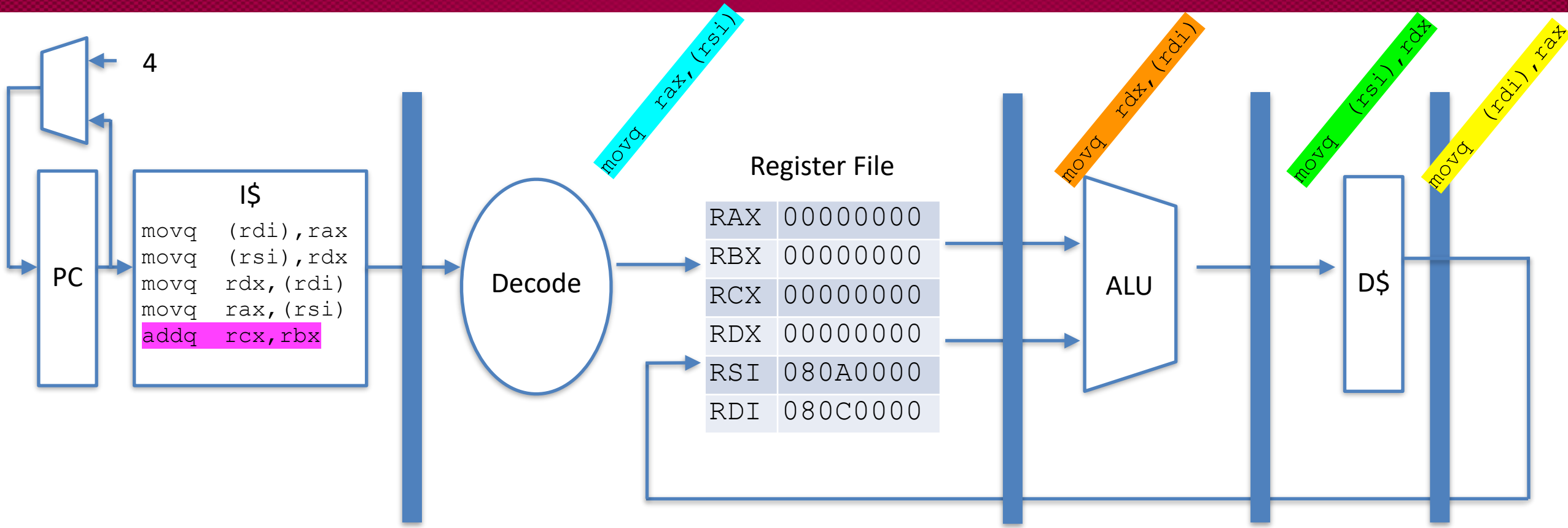

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
movq (rdi),rax
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
ALU
D$


4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
movq (rsi),rdx
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
movq (rdi),rax
ALU
D$

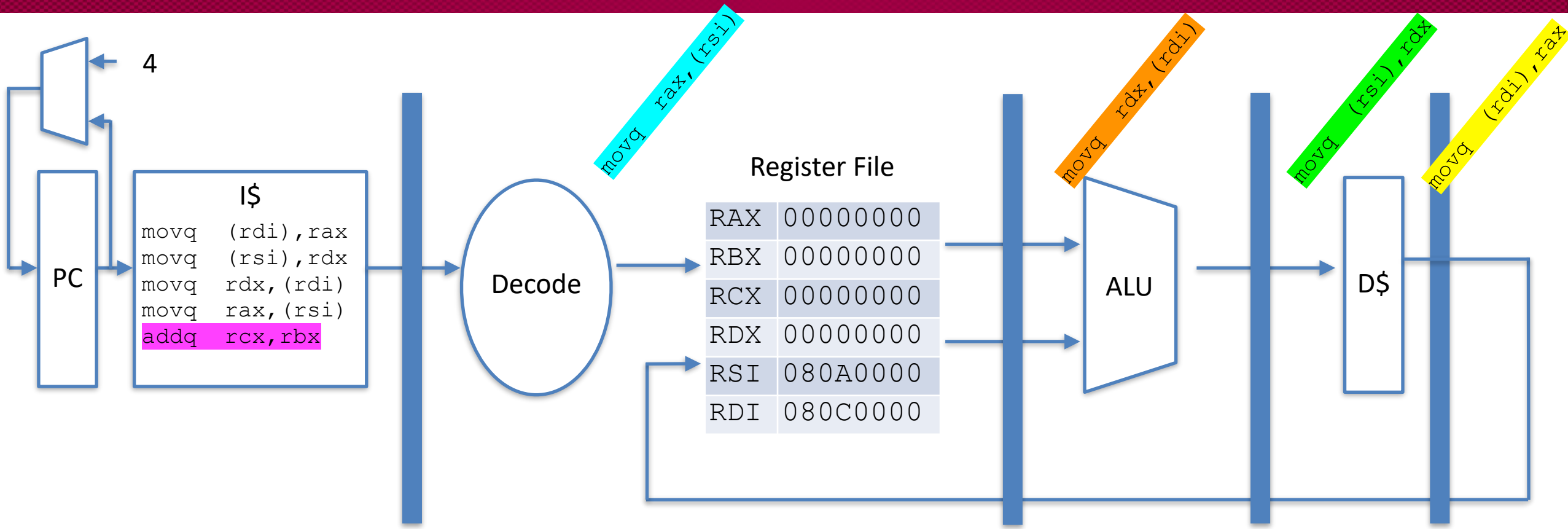

4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
movq rdx,(rdi)
Decode
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
movq (rsi),rdx
ALU
movq (rdi),rax
D$


4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq rax,(rsi)
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
movq rdx,(rdi)
ALU
movq (rsi),rdx
D$
movq (rdi),rax


4
PC
I$
movq (rdi),rax
movq (rsi),rdx
movq rdx,(rdi)
movq rax,(rsi)
addq rcx,rbx
Decode
movq rax,(rsi)
Register File
RAX 00000000
RBX 00000000
RCX 00000000
RDX 00000000
RSI 080A0000
RDI 080C0000
movq rdx,(rdi)
ALU
movq (rsi),rdx
D$
movq (rdi),rax



- **Pipelining (Instruction-Level Parallelism)**
- **Increase the clock frequency because signals don't travel as far**



- **Program Counter controls which instruction executes.**
- **Normally, the PC advances one instruction on each cycle.**
- **BUT, it can be manually changed by the software.**

HOW ELSE DO PROCESSORS EMPLOY CONCURRENCY?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. Use multiple cores to run multiple tasks in parallel
2. Run multiple tasks on a single core concurrently

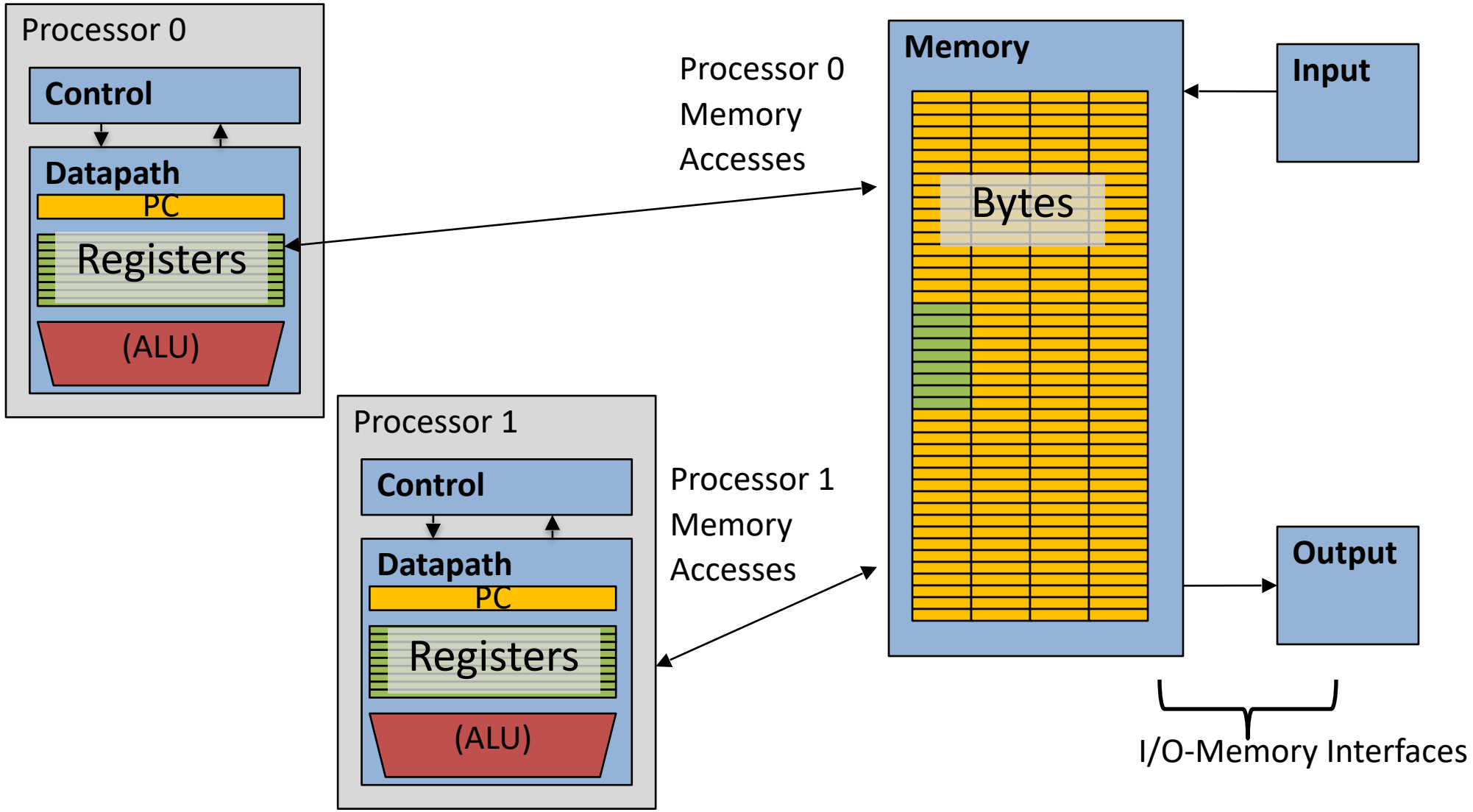
HOW ELSE DO PROCESSORS EMPLOY CONCURRENCY?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. **Use multiple cores to run multiple tasks in parallel**
2. Run multiple tasks on a single core concurrently

Multiprocessor Systems (in pictures)



MULTIPROCESSOR SYSTEMS (IN WORDS)

- A computer system with at least 2 processors or cores
  - Each core has its own registers
  - Each core executes independent instruction streams
  - Processors share the same system memory
    - But use different parts of it
  - Communication possible through memory accesses
- Deliver high throughput for independent jobs via task-level parallelism

MULTIPROCESSOR EXAMPLE

Run Chrome and Spotify simultaneously

- Each are separate programs
- Each has a different memory space
- Each can run on a separate core

Don’t even need to communicate...

Note: OS can fake this by interleaving processes,
but hardware can make it actually simultaneous

HOW ELSE DO PROCESSORS EMPLOY CONCURRENCY?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. Use multiple cores to run multiple tasks in parallel
2. **Run multiple tasks on a single core concurrently**

MULTITHREADING PROCESSORS

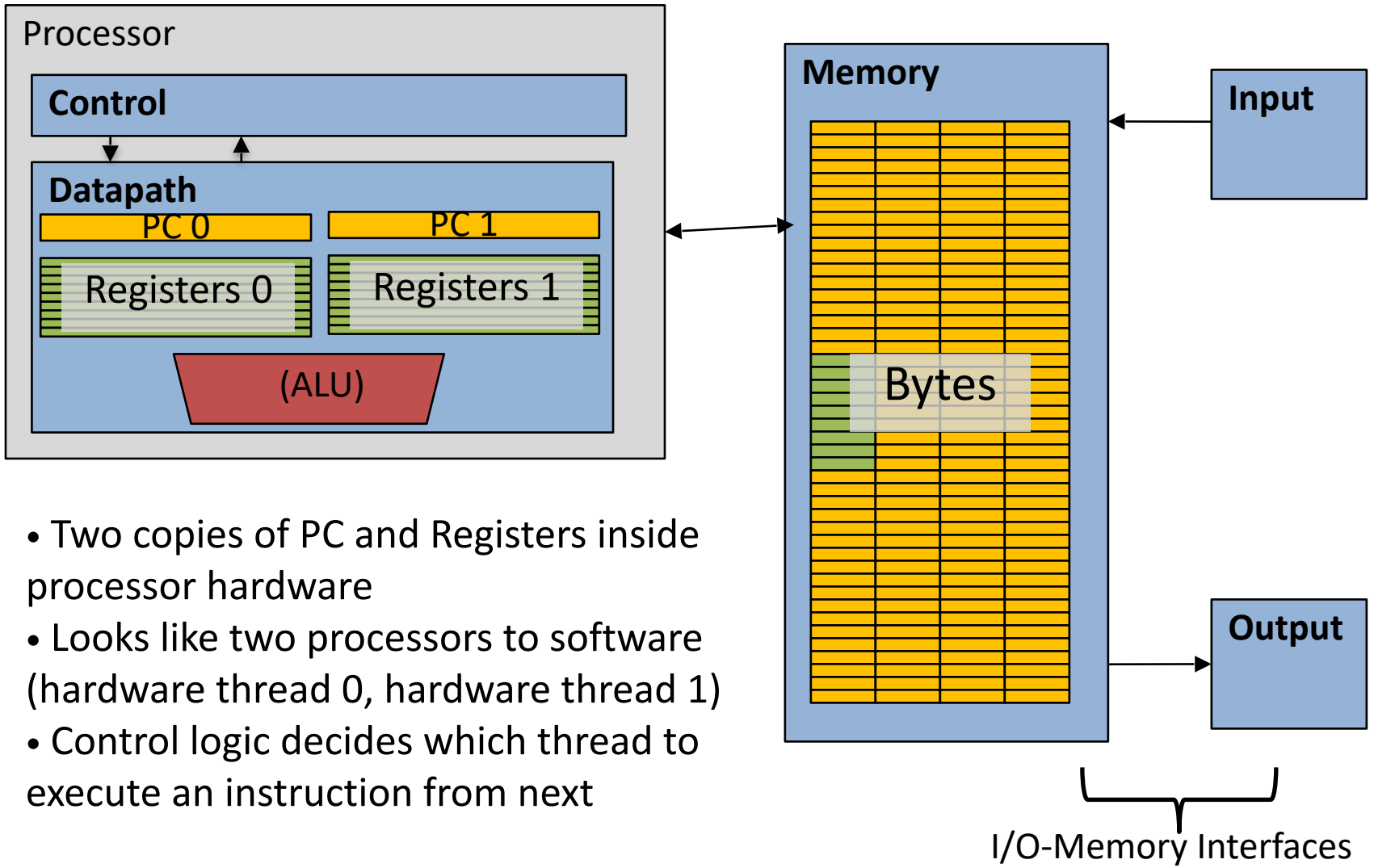
Basic idea: Processor resources are expensive and should not be left idle

Long memory latency to memory on cache miss?

- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency

- Switching threads is less expensive than processes because they share memory

HARDWARE SUPPORT FOR MULTITHREADING



- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

MULTITHREADING VERSUS MULTICORE

- Multithreading => Better utilization
  - ≈5% more hardware for ≈1.3x better performance?
  - Gets to share ALUs, caches, memory controller
- Multicore => Duplicate processors
  - ≈50% more hardware for ≈2x better performance?
  - Share some caches (L2 cache, L3 cache), memory controller
- Modern machines do both
  - Multiple cores with multiple threads per core

BACK UP TO THE OS PERSPECTIVE

- Modern operating systems must manage concurrency
  - Both parallel operation and interleaving operations

- Concurrency is valuable
  - Performance gains are the reason

SPEEDUP & AMDAHL'S LAW

ADMINISTRIVIA

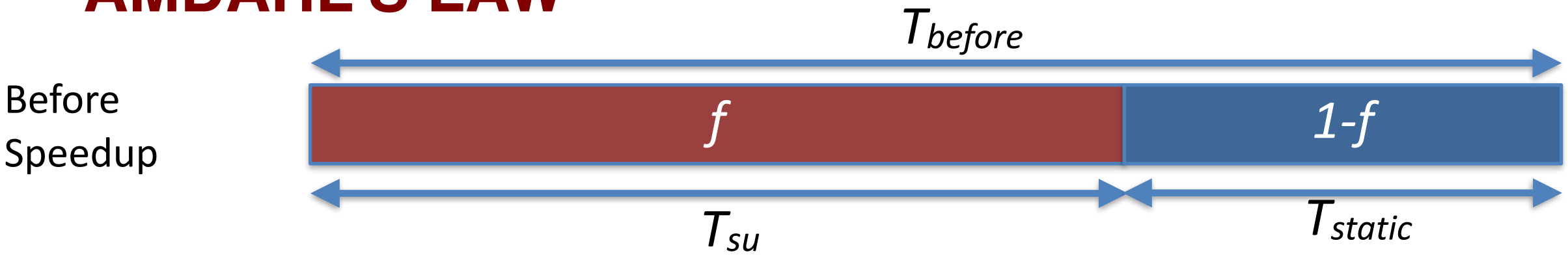
- Shell homework due tonight 11:59 PM.
- Quiz Wednesday 9/20 on Processes & Concurrency
- Homework 2 is Out. Due next Wednesday 9/20.

SPEEDUP

$$Speedup = \frac{T_{old}}{T_{new}}$$

- A task that used to take 1s now takes 0.5s
  - Speedup is 2x
- Speedup > 1 means new way is faster
- Speedup < 1 means new way is slower

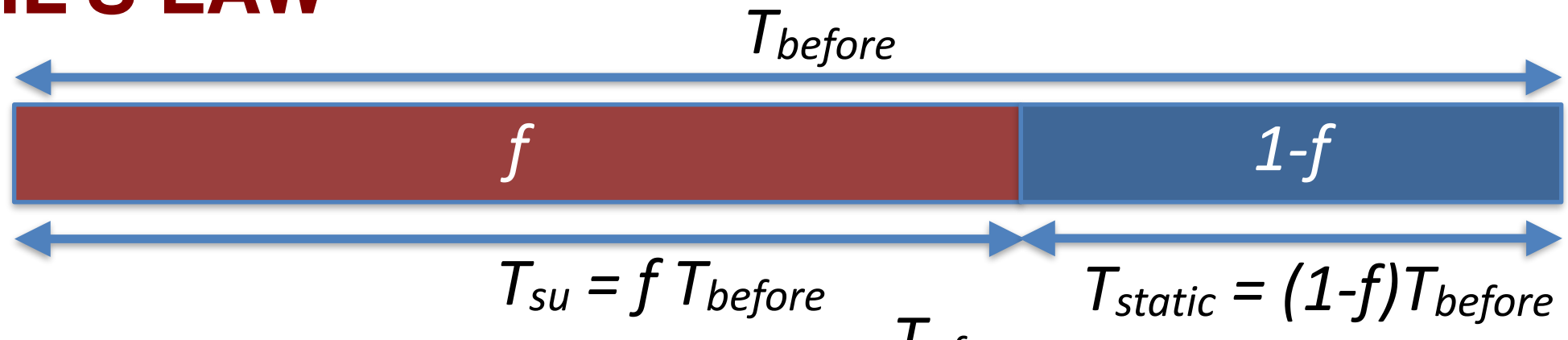
AMDAHL'S LAW



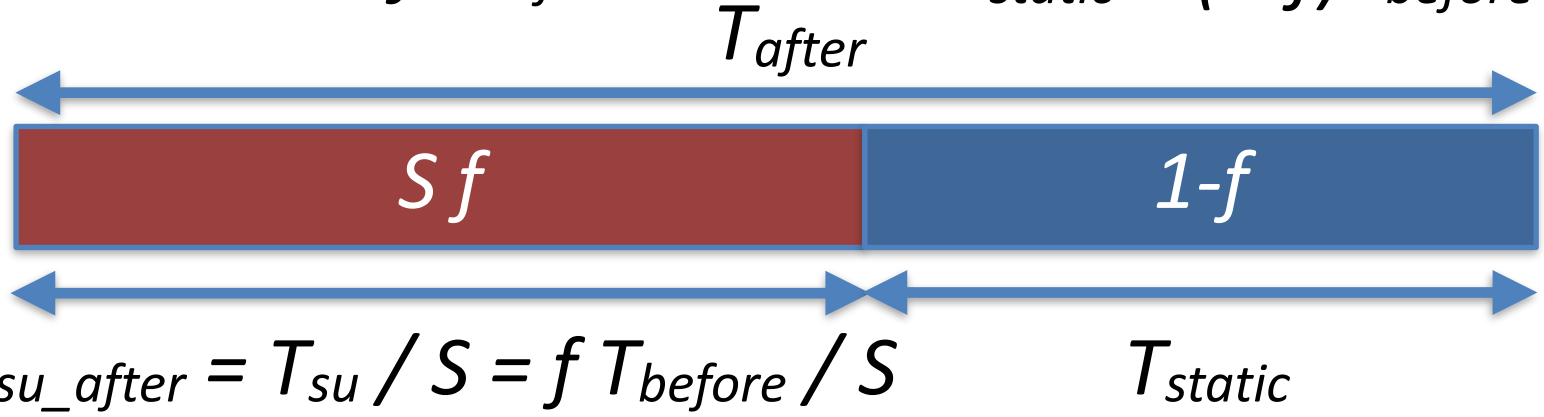
- **We have some task that we want to speed up.**
  - **The red fraction f_{su} can be sped up.**
  - **The blue fraction f_{static} can't be changed.**
- **Initially, the task takes T seconds to complete.**
  - $T = f\ T_{su} + (1-f)\ T_{static}$

AMDAHL'S LAW

Before Speedup



After Speedup

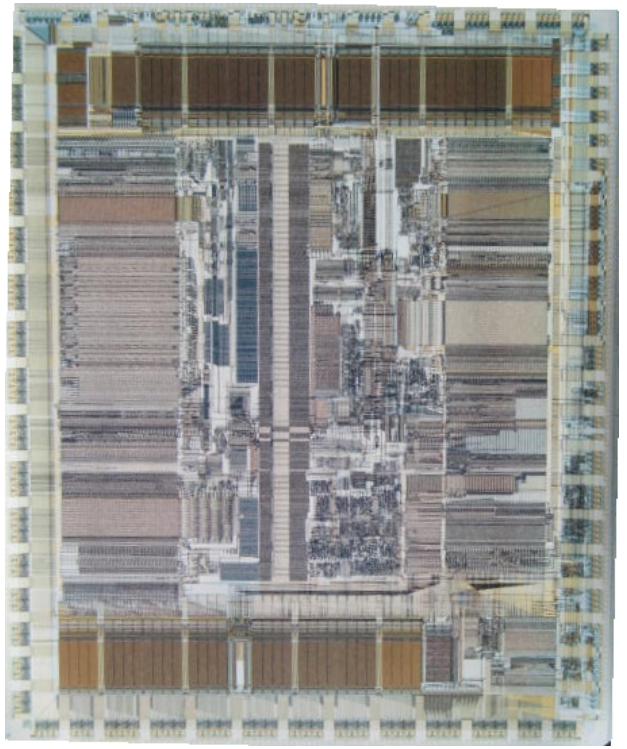


$$S_{overall} = \frac{T_{before}}{T_{after}} = \frac{T_{before}}{\left(\frac{f}{S}\right) T_{before} + (1-f)T_{before}} = \frac{1}{\frac{f}{S} + 1 - f}$$

WHAT WE CAN LEARN FROM AMDAHL'S LAW

- **Always focus on improving the longest-running part first.**
- **Even a huge improvement in a small component will give modest gains.**

CASE STUDY: DEC ALPHA 21264



Reduce the OoO Logic Power by 10%

$$P = \frac{1}{\frac{0.193}{1.1} + 1 - 0.193} \approx 1.01786$$

Gives a 1.8% overall power reduction! That's nothing!

Functional Unit	Power Fraction
Caches	16.1%
Out-of-Order Issue Logic	19.3%
Memory Management Unit	8.6%
Floating Point Execution Unit	10.8%
Integer Execution Unit	10.8%
Clock Tree	34.4%

Data from: *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*

CHALLENGES TO CONCURRENCY

Concurrency is great! We can do so many things!!

But what's the downside…?

1. How much speedup can we get from it?
2. **How hard is it to write parallel programs?**

CONCURRENCY PROBLEM: DATA RACES

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 1;
}
```

Thread 2:

```
void main(){
  count += 1;
}
```

`count` could end up with a final value of 1 or 2. How?

CONCURRENCY PROBLEM: DATA RACES

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Thread 2:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Assuming "count" is in memory location 0x8049a1c

`count` could end up with a final value of 1 or 2. How?

These instructions could be interleaved in any way.

DATA RACE EXAMPLE

Assuming "count" is in memory location pointed to by `%edi`

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Final value of count: 2

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Final value of count: 1

DATA RACE EXPLANATION

- Thread scheduling is **non-deterministic**
  - There is no guarantee that any thread will go first or last or not be interrupted at any point

- If different threads write to the same variable
  - The final value of the variable is also non-deterministic
  - This is a data race

CHECK YOUR UNDERSTANDING: DATA RACES WITH MULTIPLE THREADS

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 2;
}
```

Thread 2:

```
void main(){
  count -= 2;
}
```

Thread 3:

```
void main(){
  count += 3;
}
```

What are the possible values of count?

CHECK YOUR UNDERSTANDING: DATA RACES WITH MULTIPLE THREADS

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 2;
}
```

Thread 2:

```
void main(){
  count -= 2;
}
```

Thread 3:

```
void main(){
  count += 3;
}
```

What are the possible values of count? **-2, 0, 1, 2, 3, 5**

How are you supposed to reason about this?!
Need mechanisms for sharing memory.

RACE CONDITION

- Two or more things are happening at the same time
- It's not clear which will run when
- The result will be different depending on execution order
- Result becomes indeterminate (non-deterministic)

- Data race
  - Two or more threads access shared memory at the same time and at least one modifies it

CRITICAL SECTION

- Code that interacts with a shared resource must not be executed concurrently

- Part of code that accesses a shared resource is a **Critical Section**
  - In other words, code that would lead to a data race
  - May be multiple, unrelated critical sections for multiple shared resources

- Critical sections need to be addressed for correctness
  - Races can be avoided by never overlapping multiple critical sections
  - We must execute critical sections “atomically” (all or none)

Critical section occurs when shared memory is accessed

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

static volatile int counter = 0;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter++;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  printf("main: begin (counter = %d)\n",
counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d,
goal was %d)\n", counter, 2*LOOPS);
  return 0;
}
```

Example: Where is the critical section?

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

static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  person1 = "Jack";
  person2 = "Jill";
  printf("main: begin (%s, %s)\n",
         person1, person2);
  pthread_create(&p1, NULL, mythread,
"A");
  pthread_create(&p2, NULL, mythread,
"B");


  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: end (%s, %s)\n",
         person1, person2);
}
```

Buggy concurrent swap. What can go wrong?

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

static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  person1 = "Jack";
  person2 = "Jill";
  printf("main: begin (%s, %s)\n",
         person1, person2);
  pthread_create(&p1, NULL, mythread,
"A");
  pthread_create(&p2, NULL, mythread,
"B");


  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: end (%s, %s)\n",
         person1, person2);
}
```

Buggy concurrent swap. What can go wrong?

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

static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  person1 = "Jack";
  person2 = "Jill";
  printf("main: begin (%s, %s)\n",
         person1, person2);
  pthread_create(&p1, NULL, mythread,
"A
"B
  printf("main: end (%s, %s)\n",
         person1, person2);
}
```

For a brief period in time:

person1: “Jill”
person2: “Jill”

Example: Is there a problem here?

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

static volatile int sum_amount = 2;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  int counter = 0;
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter += sum_amount;
  }
  printf("%s: done %d\n", (char*)arg,
                          counter);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  printf("main: begin\n");
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done\n");
  return 0;
}
```

Check your understanding. Is there a problem here?

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

static volatile int sum_amount = 2;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  int counter = 0;
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter += sum_amount;
  }
  printf("%s: done %d\n", (char*)arg,
                          counter);
  return NULL;
}
```

This code will work!

All threads only read from shared memory.

If at least one wrote to shared memory, it would be a problem.

SOLUTION REQUIREMENTS

We MUST stop data races from occurring in our programs.

1. **No two processes may simultaneously be in their critical sections.**

2. **Processes outside of critical sections should have no impact.**

3. **No assumptions should be made about number of cores, speed of cores, or scheduler choices.**

LOCKS (ALSO KNOWN AS A MUTEX)

- **Locks are the simplest mutual exclusion primitive**
  - **Represent a resource that can be reserved and freed**

- **Acquire/lock:**
  - **Used before a critical section to reserve the resource**
  - **If the lock is free (unlocked), then lock it and proceed.**
  - **If the lock is already taken (someone else called acquire/lock), then wait until it's free before proceeding.**
- **Release/unlock:**
  - **Used at the end of a critical section to free the resource**
  - **Only the thread holding the lock can release it**
  - **Allows one waiting (or future) thread to acquire the lock**

TWO DIFFERENT METAPHORS & ETYMOLOGY

Lock

- **Think about locking a bathroom door**
- **Our virtual lock works as follows:**
  - **Anyone can lock or unlock (there is no "key").**
  - **Trying to enter (lock) if the lock is already-locked will cause you to wait until it's unlocked.**

Token

- Holding the token gives you permission to do something.
- There is only one token.
- Thus, you:
  1. Try to **acquire** the token ("lock"). You have to wait your turn if someone else is holding it.
  2. When done, **release** the token/ lock.
- The token represents exclusive access to a shared resource or a critical section.

Locks prevent data races

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

static volatile int counter = 0;
static const int LOOPS = 1e7;
static pthread_mutex_t lock;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  pthread_mutex_init(&lock, 0);
  printf("main: begin (counter = %d)\n",
counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d,
goal was %d)\n", counter, 2*LOOPS);
  return 0;
}
```

1. Approach for single-core machines: disable interrupts

```
void lock() {
  disable_interrupts();
}

void unlock() {
  enable_interrupts();
}
```

- Disable interrupts to prevent preemption during critical section
  - Scheduler can’t run if the OS never takes control
  - Also stops data races in interrupt handlers
- Problems
  - Doesn’t work on multicore machines
  - Bad Idea to let processes disable the OS
    - Process could freeze the entire computer
  - Might screw up timing for interrupt handling

2. Straightforward approach: lock variable with loads/stores

```
// wait for lock released
while (lock != 0);
// lock == 0 now (unlocked)

// set lock
lock = 1;

      // access shared resource ...
      // sequential execution!

// release lock
lock = 0;
```

Initialization:

```
boolean lock = false;
```

Is this going to work though?

Race condition on lock variable

Thread 1

```
while (lock != 0);

lock = 1;

// critical section

lock = 0;
```

Thread 2

```
while (lock != 0);

lock = 1;

// critical section

lock = 0;
```

Race condition on lock variable

Thread 1

```
while (lock != 0);

lock = 1;

// critical
section


lock = 0;
```

Thread 2

```
while (lock != 0);

lock = 1;

// critical section

lock = 0;
```

- Thread 2 finds lock is not set before Thread 1 sets it
- Both threads believe they acquired and set the lock!

Race condition on lock variable

Thread 1	**Thread 2**
`while (lock != 0);`	
	`while (lock != 0);`
`lock = 1;`	
`// critical section`	
	`lock = 1;`
`lock = 0;`	`// critical section`
	`lock = 0;`

- Lock is released and available while Thread 2 is in critical section!

2. Straightforward approach: lock variable with loads/stores

```
while(locklock != 0);
locklock = 1;
   // wait for lock released
   while (lock != 0);
   // lock == 0 now (unlocked)

   // set lock
   lock = 1;
locklock = 0;

     // access shared resource ...

// release lock
lock = 0;
```

Initialization:

```
boolean lock = false;
boolean locklock =
false;
```

2. Straightforward approach: lock variable with loads/stores

```
while(locklock != 0);
locklock = 1;
   // wait for lock released
   while (lock != 0);
   // lock == 0 now (unlocked)

   // set lock
   lock = 1;
locklock = 0;

      // access shared resource ...

// release lock
lock = 0;
```

Initialization:

```
boolean lock = false;
boolean locklock =
false;
```

- This is not going to work…
- **Problem**: the lock itself is a shared resource!

2. Algorithmic approach: Peterson's Algorithm

- There are indeed several algorithmic approaches to create a lock!

- See textbook (or other sources) for Peterson's Solution for two threads

- Advantages:
  - Algorithm, so it works on any platform no matter the hardware
- Disadvantages:
  - Solution for N threads gets complicated
  - Performance is slow

3. Hardware approach: atomic instructions

- **Atomic** instructions perform operations on memory in one uninterruptable instruction
  - Guarantees that all parts of the instruction occur before the next instruction
  - In multicore, guarantees that entire access to memory is serialized

- Commonly read, modify, and write in a single instruction

Atomic Instruction: Exchange

- Example atomic_exchange

pseudocode for the instruction: remember, this is actually in hardware NOT C

```
int atomic_exchange(int* pointer, int new_value) {
    int old_value = *pointer; // fetch old value from memory
    *pointer = new_value;     // write new value to memory
    return old_value;         // return old value
}
```

- atomic_exchange(destptr, newval)
  - Write a new value to memory, and return the old one
  - Also known as test-and-set when operating on boolean data
  - x86-64 instruction: `lock; xchg`

Atomic Instruction: Compare And Swap

- Example atomic_compare_and_swap (remember, this is pseudocode for hardware)

```
bool atomic_compare_and_swap (int* pointer, int expected_value, int
new_value) {
    int actual_value = *pointer;
    if (actual_value == expected_value) {
        *pointer = new_value;
        return true;
    }
    return false;
}
```

- atomic_compare_and_swap(destptr, oldval, newval)
  - x86-64 instruction: `lock; cmpxchg`
  - Generalization of exchange
    - Exchange(ptr, new) -> CompareAndSwap(ptr, *ptr, new)

Sequential memory consistency

- Memory barrier
  - Guarantees that all load/stores **before** this line of code are completed before any load/stores **after** this line of code are started
  - Comes in software (compiler orders things) and hardware (processor orders things) forms
    - Both are necessary for correct execution!
  - C wrappers for atomics allow you to specify a memory barrier

- Atomic Load/Store C-wrappers
  - Guarantee sequential consistency
  - Remember: memory could be reordered by compiler or processor!

Spinlock implementation

```
typedef struct {
    int flag; // 0 indicates that mutex is available, 1 that it is held
} lock_t;

void mutex_init(lock_t* mutex) {
    mutex->flag = 0; // lock starts available
}
void mutex_acquire(lock_t* mutex) {
    while (atomic_exchange(&(mutex->flag), 1) == 1); // spin-wait until available
}
void mutex_release(lock_t* mutex) {
    atomic_store(&(mutex->flag), 0); // make lock available
}
```