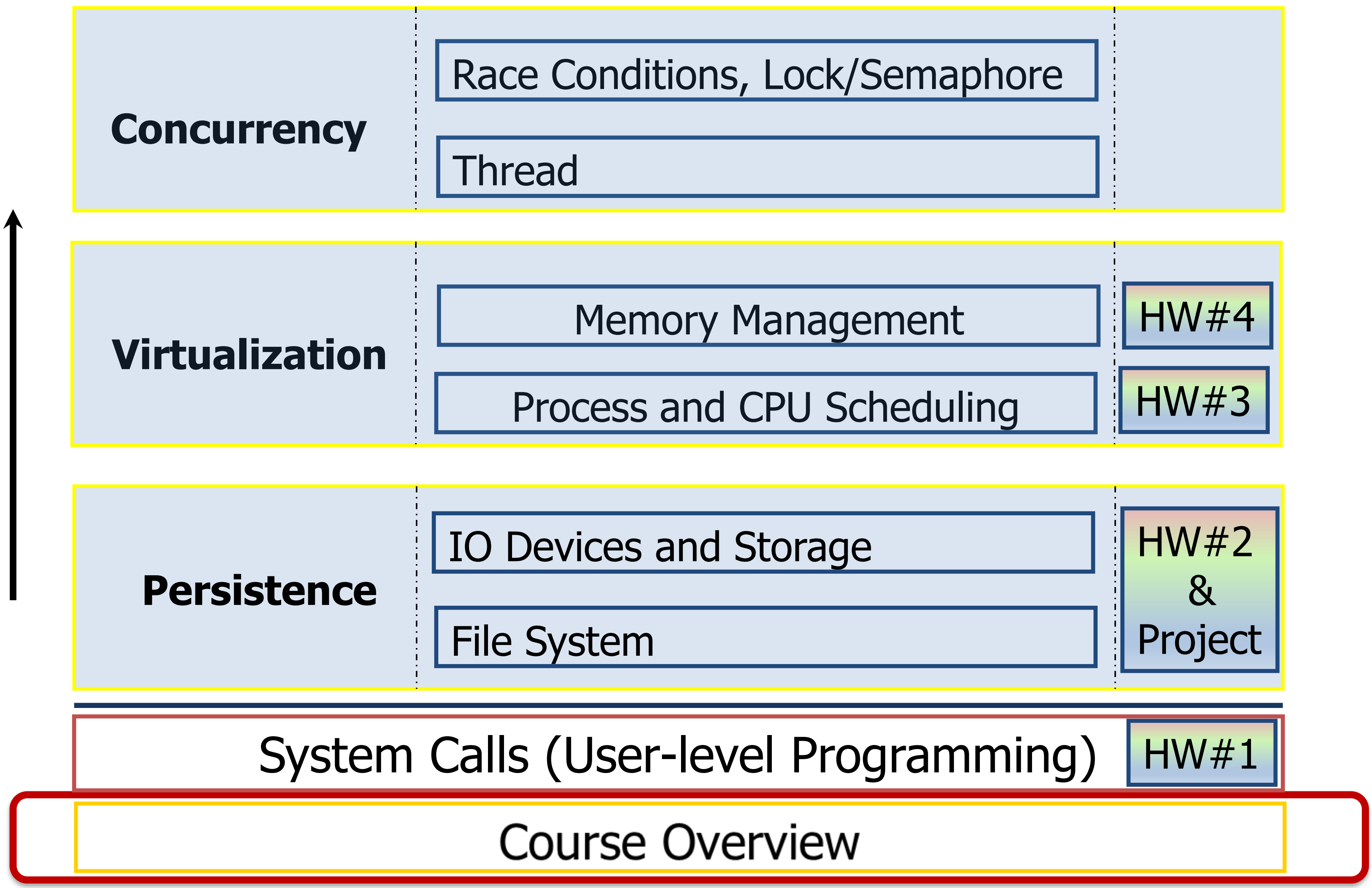
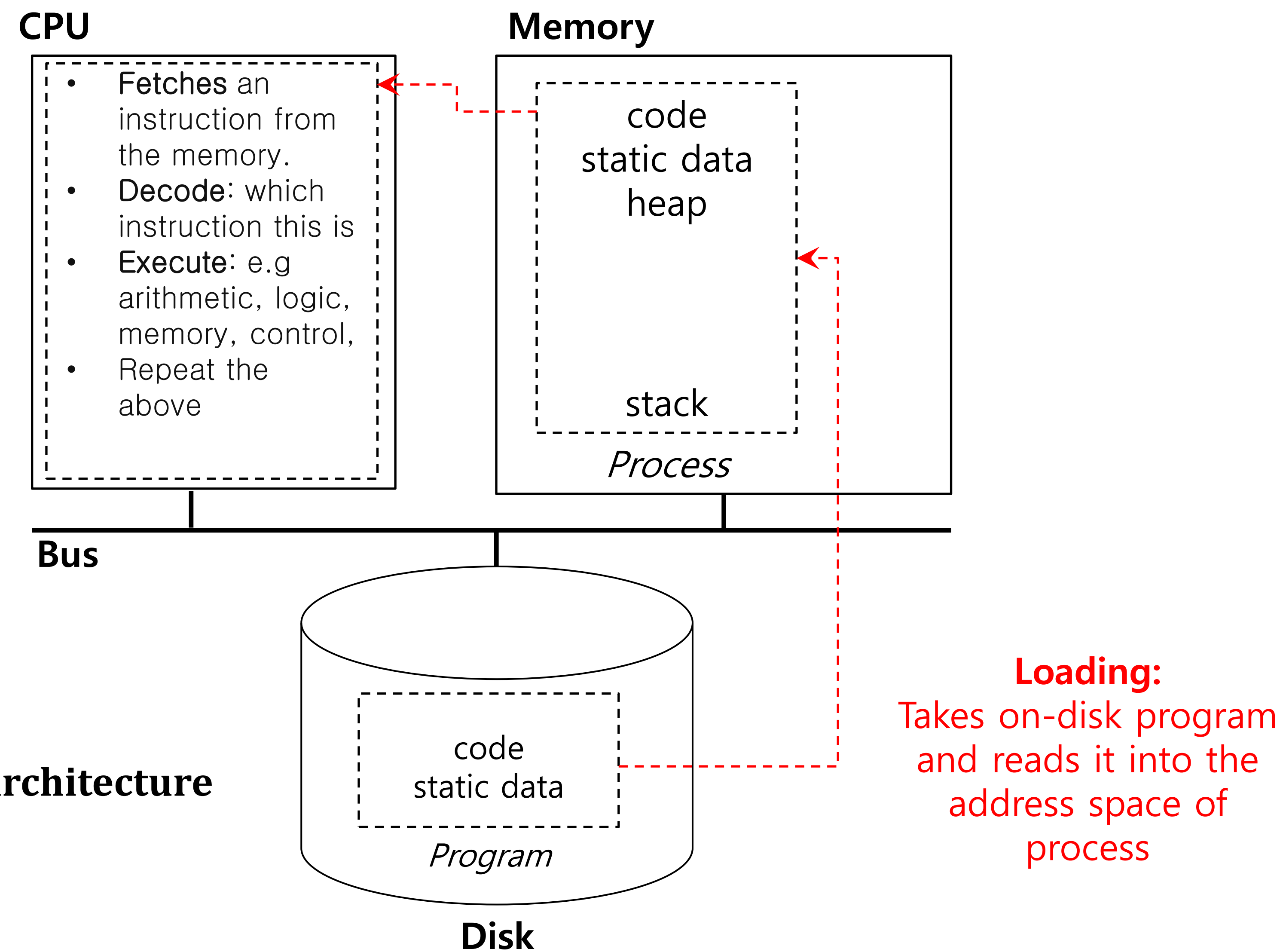
Lecture 1: Course Overview: Introduction to OS

The Course Organization (Bottom-up)

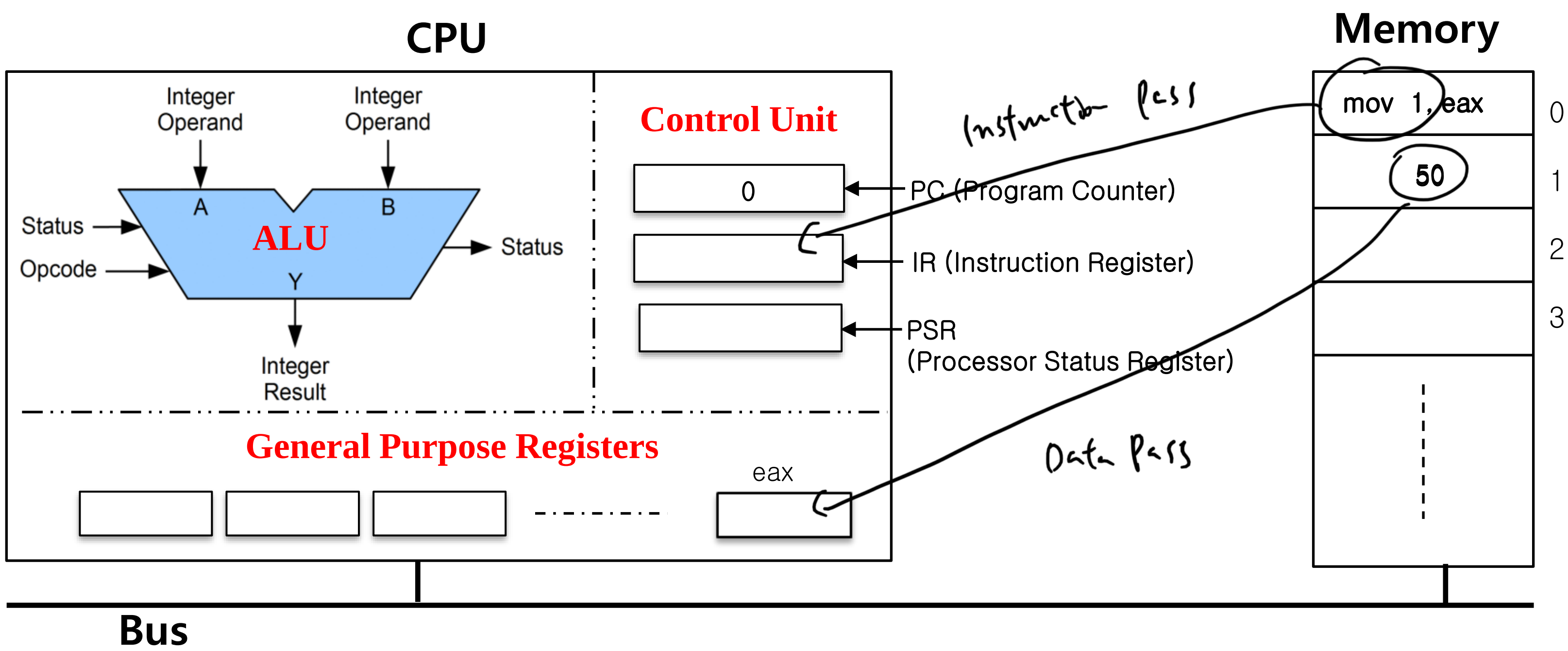


What happens when a program runs?



Von Neumann architecture

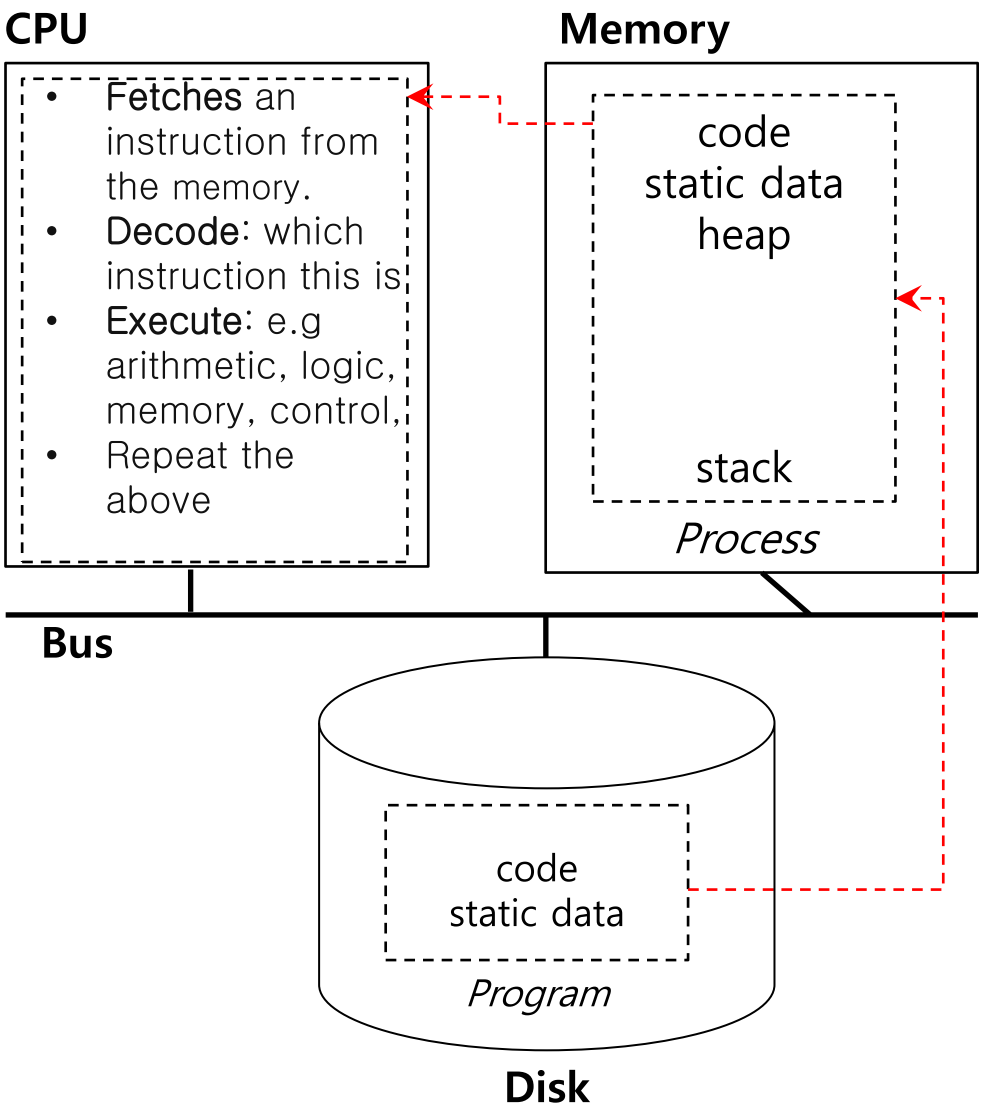
How CPU works?



1. **Fetch** an instruction from memory.
2. **Decode**: Figure out which instruction this is
3. **Execute**: i.e., arithmetic/logic operations (e.g. add/and), memory, condition, branch, etc.

Repeat the above steps

Operating System (OS)



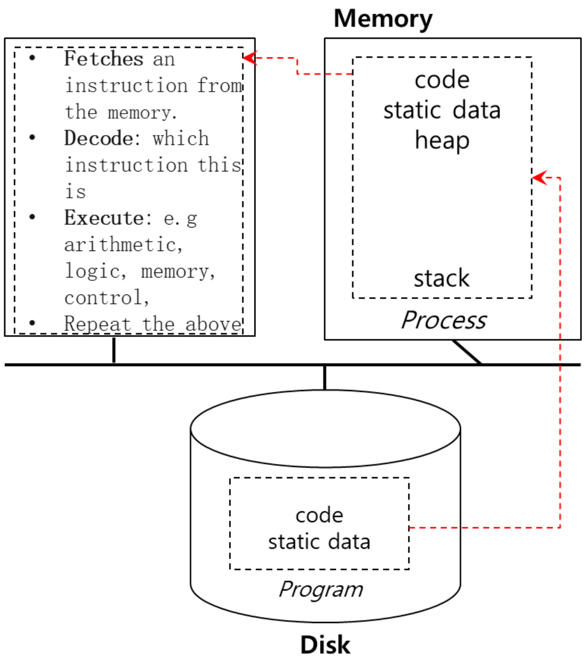
- OS's major functions
  - Manage Resources (*virtualization*)
    - CPU, Memory, Disk, etc.
  - Provide Services (*system call*)
    - **Process (running program)**: Easily **run** programs
    - **Memory**: **Share** memory among processes
    - **IO devices**: Enable processes to interact with or operate/utilize devices

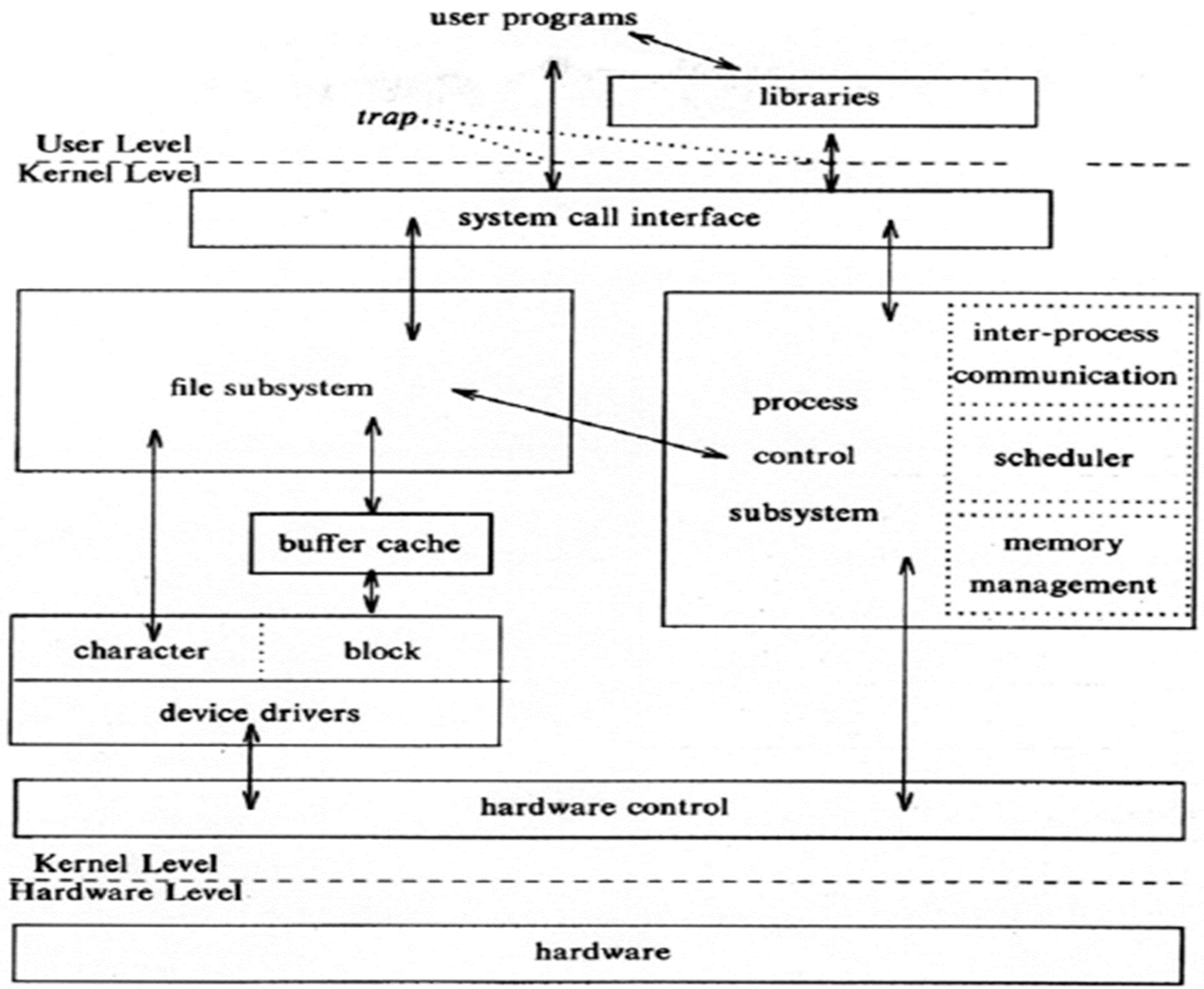
OS – Resource management via virtualization

- The OS **manages resources** such as *CPU*, *memory* and *disk*.
  - many programs to run (processes) → Sharing the CPU
  - many processes to *concurrently* access their own instructions and data → Sharing memory
  - many processes to access devices → Sharing disks
  - etc.
- How to achieve this – **Virtualization**
  - OS takes a physical resource and transforms it into a virtual form of itself.
    - **Physical resource**: Processor, Memory, Disk …
  - The virtual form is more general, powerful and easy-to-use.
  - We can refer to the OS as a **virtual machine**.

System call

- OS provides services (to utilize resources) via **System Call** (typically a few hundred) to run process, access memory/devices, etc.





<u>The Design Of The Unix Operating System (Maurice Bach, 1986)</u>

Virtualizing the CPU

- The system has a very large number of virtual CPUs.
  - Turn a single CPU into a seemingly infinite number of CPUs.
  - Allow many processes to seemingly run at the same time

    → **Virtualizing the CPU**

Virtualizing the CPU (Cont.)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }

        char *str = argv[1];
        int i=0;

        while (i<100) {
                sleep(1);
                i++;
                printf("%s\n", str);
        }
        return 0;
}

```

Simple Example(`cpu.c`): Code that loops and prints

Virtualizing the CPU (Cont.)

- Execution result 1.

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A
A
A
A
^C
prompt>
```

Pressing “Ctrl-c” to halt the program

Virtualizing the CPU (Cont.)

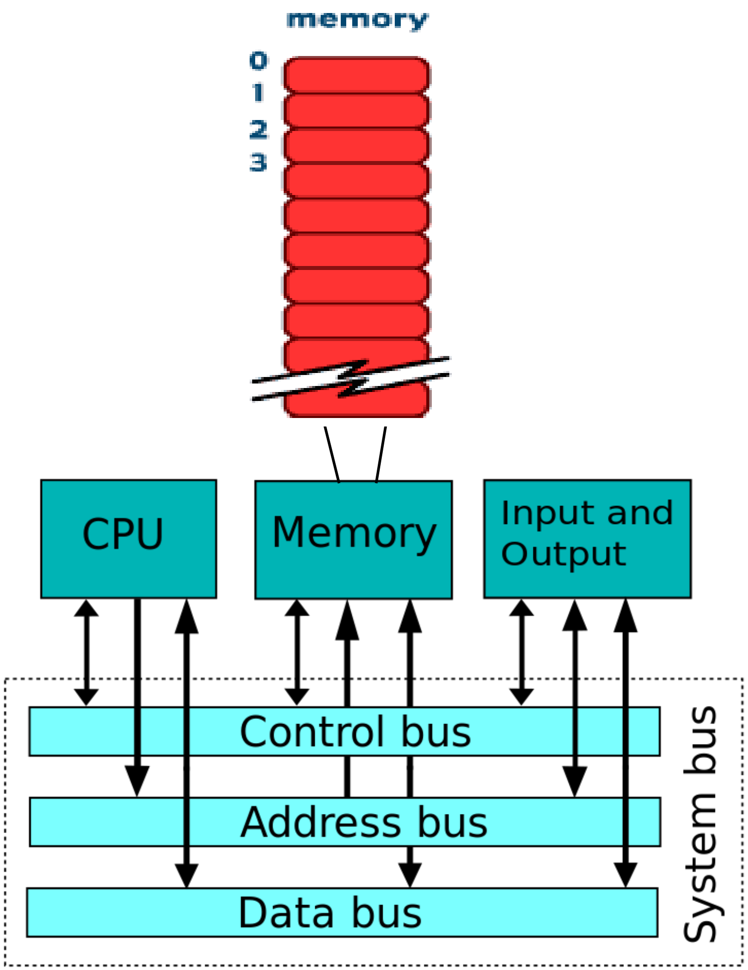
- Execution result 2.

```
prompt> ./cpu A &  ./cpu B &  ./cpu C &  ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Even though we have only one CPU, the four processes seem to be running at the same time!

Virtualizing Memory

- The physical memory is *an array of bytes*.
- A program keeps all of its data structures in memory.
  - **Read memory** (load):
    - Specify an address to be able to access the data
  - **Write memory** (store):
    - Specify the data to be written to the given address



Virtualizing Memory (Cont.)

- A program that Accesses Memory (`mem.c`)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define MAX 100000000
int
main(int argc, char *argv[])
{
        int *p = malloc(MAX*sizeof(int));  // a1: allocate memory
        assert(p != NULL);
        printf("(%d) Address Range: %08x %08x\n",getpid(),
                (unsigned) p, (unsigned) (p+MAX-1));
        *p = 0;  // a3: put zero into the first slot of the memory
        int i=0;
        while (i<10) {
                sleep(1);
                *p = *p + 1; i=i+1;
                printf("(%d) p: %d\n", getpid(), *p);
        }
        return 0;
}
```

Virtualizing Memory (Cont.)

- The output of the program `mem.c`

```
prompt>  ./mem
(25187) Address Range: 9f7eb008 - b7563404
(25187) p: 1
(25187) p: 2
(25187) p: 3
(25187) p: 4
(25187) p: 5
(25187) p: 6
(25187) p: 7
(25187) p: 8
(25187) p: 9
(25187) p: 10
```

  - The newly allocated memory range is at `9f7eb008 - b7563404`.
  - It updates the value and prints out the results.

Virtualizing Memory (Cont.)

- Running `mem.c` multiple times

```
prompt> ./mem & ./mem &
(25274) Address Range: 9f7eb008 - b7563404
(25275) Address Range: 9f8c0008 - b7638404
(25274) p: 1
(25275) p: 1
(25274) p: 2
(25275) p: 2
(25274) p: 3
(25275) p: 3
(25274) p: 4
(25275) p: 4
...
```

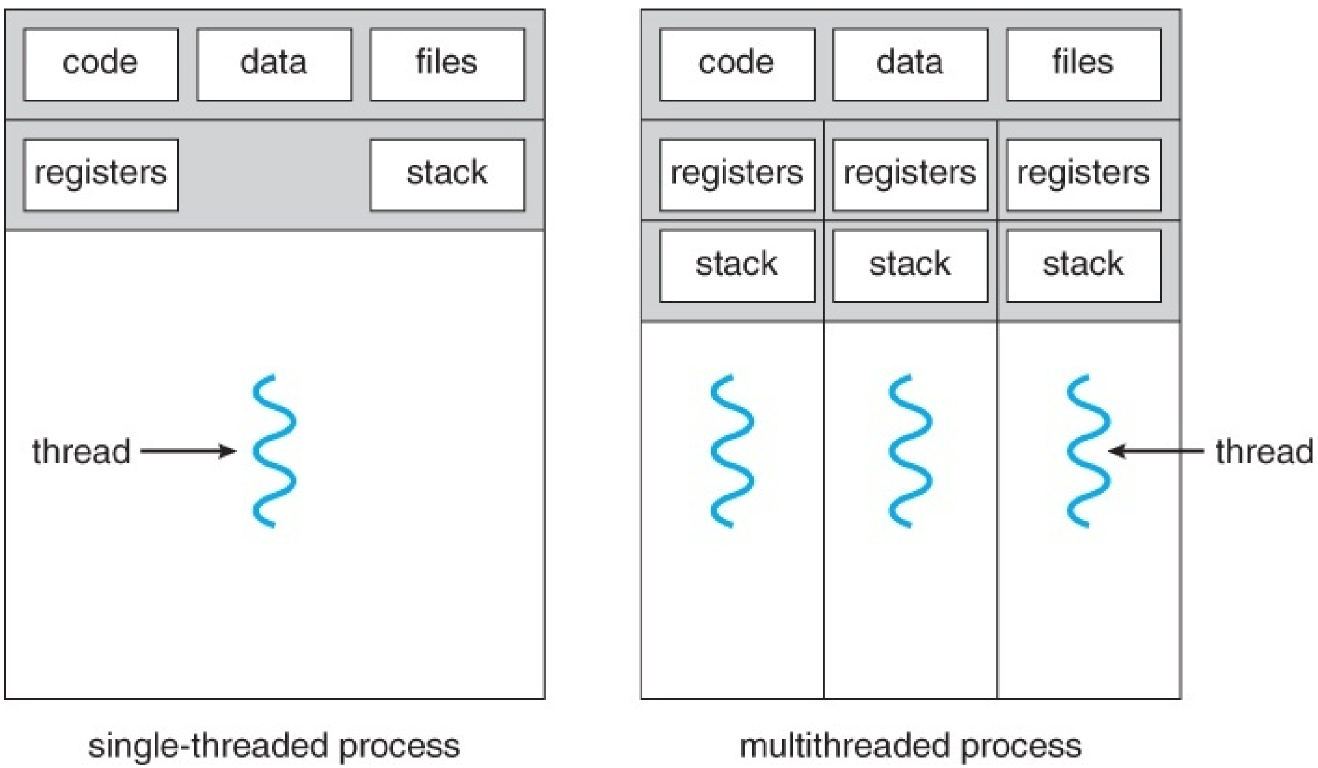
  - The memory ranges from two processes are overlapped – Each process has its **own private memory**.
    - Each running program has allocated memory at <u>its own address space</u>.
    - Each seems to be updating the value independently.

Virtualizing Memory (Cont.)

- Each process accesses its own private **virtual address space**.
  - The OS maps address space onto the physical memory.
  - A memory reference within one running program does not affect the address space of other processes.
  - Physical memory is a shared resource, managed by the OS.

Concurrency Problem

- The OS is juggling many things at once, first running one process, then another, and so forth.
- Modern multi-threaded programs also exhibit the concurrency problem.



Concurrency Example

- A Multi-threaded Program (`thread.c`)

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

volatile int counter = 0;
int loops;

void *worker(void *arg) {
        int i;
        for (i = 0; i < loops; i++) {
                counter++;
        }
        return NULL;
}
```

← 1000

2 workers : 2,000

Concurrency Example (Cont.)

```
int
main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: threads <value>\n");
                exit(1);
        }
        loops = atoi(argv[1]);
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);

        pthread_create(&p1, NULL, worker, NULL);
        pthread_create(&p2, NULL, worker, NULL);
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

- The main program creates **two threads**.
  - <u>Thread</u>: a function running within the same memory space. Each thread starts running in a routine called `worker()`.
  - `worker()`: increments a counter

Concurrency Example (Cont.)

- `loops` determines how many times each of the two workers will **increment the shared counter** in a loop.
  - `loops`: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

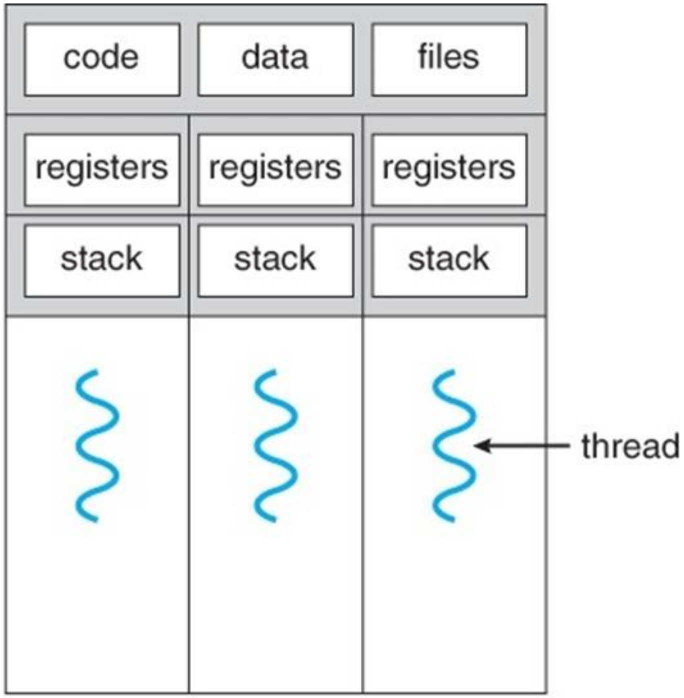
  - `loops`: `100000000`

```
prompt> ./thread 100000000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000000
Initial value : 0
Final value : 137298 // what the??
```

Why is this happening?

- Increment a shared counter → take three instructions.
  1. Load the value of the counter from the memory into a register.
  2. Increment it
  3. Store it back into the memory

- These three instructions do not execute atomically. → Problem of **concurrency** happen.



multithreaded process

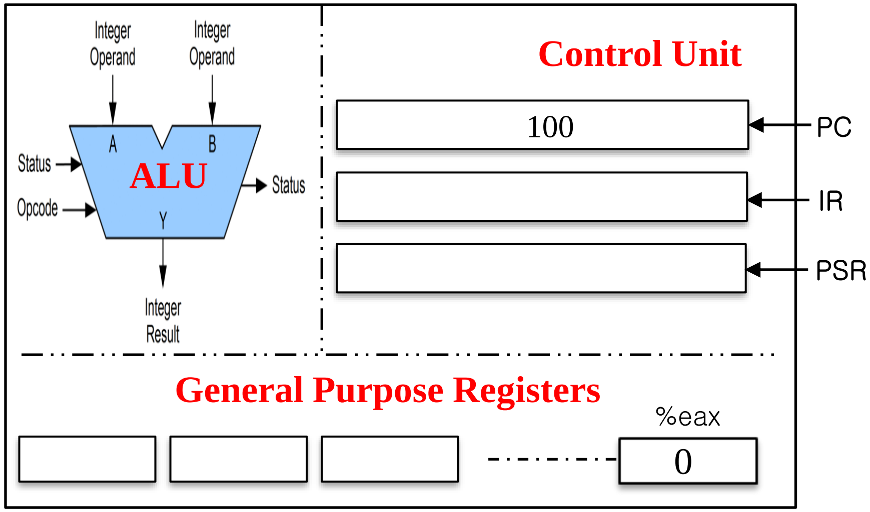
What happened?

- Example with two threads
  - counter = counter + 1 (**initial value: 50**)
  - We expect the result is 52. However,

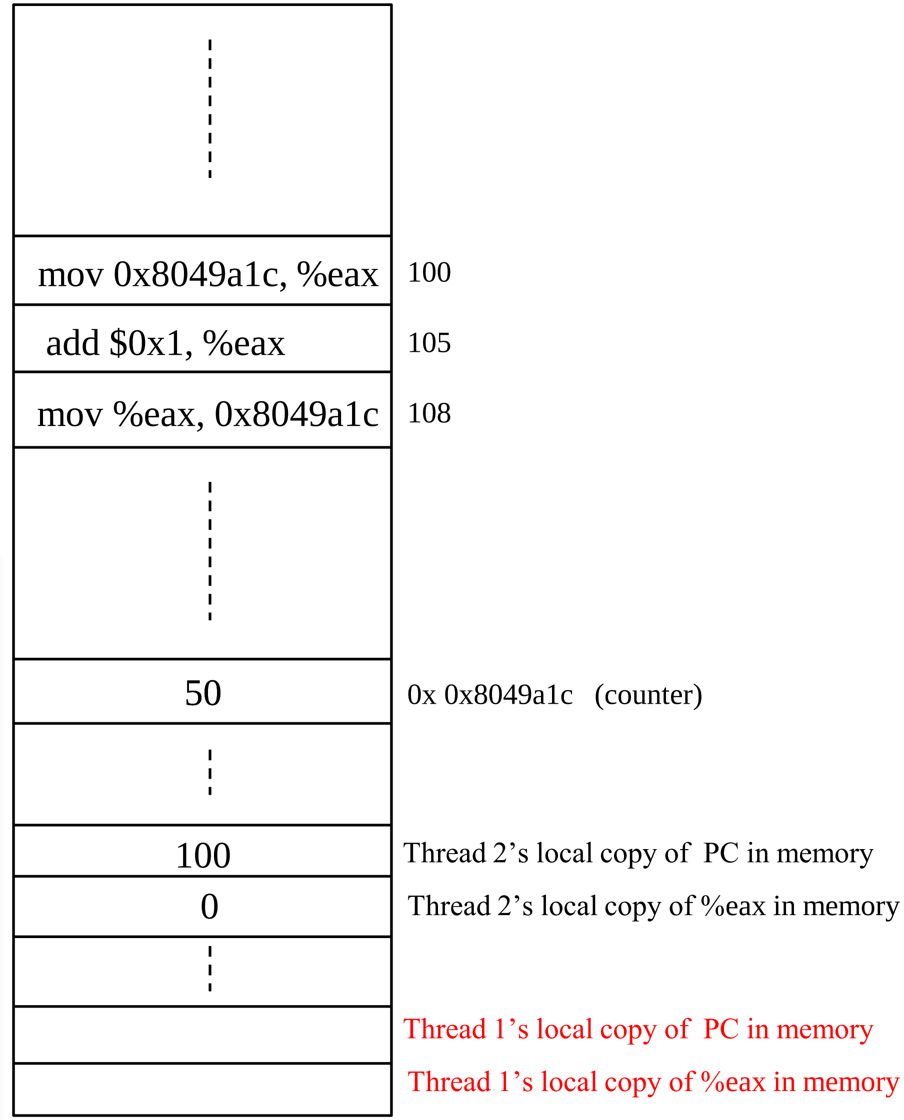
OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	51
	`mov %eax, 0x8049a1c`		113	51	**51**

Step 0: Initial Value (PC=100)

CPU

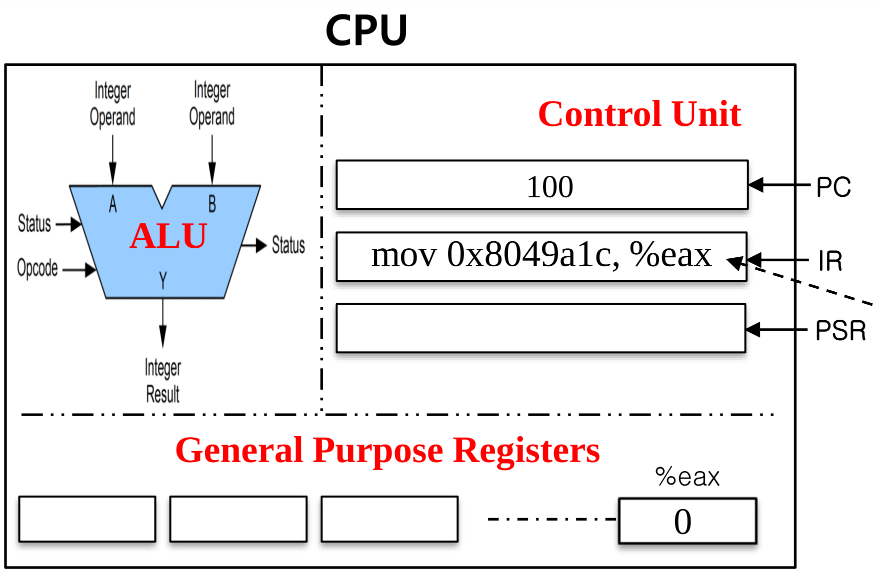


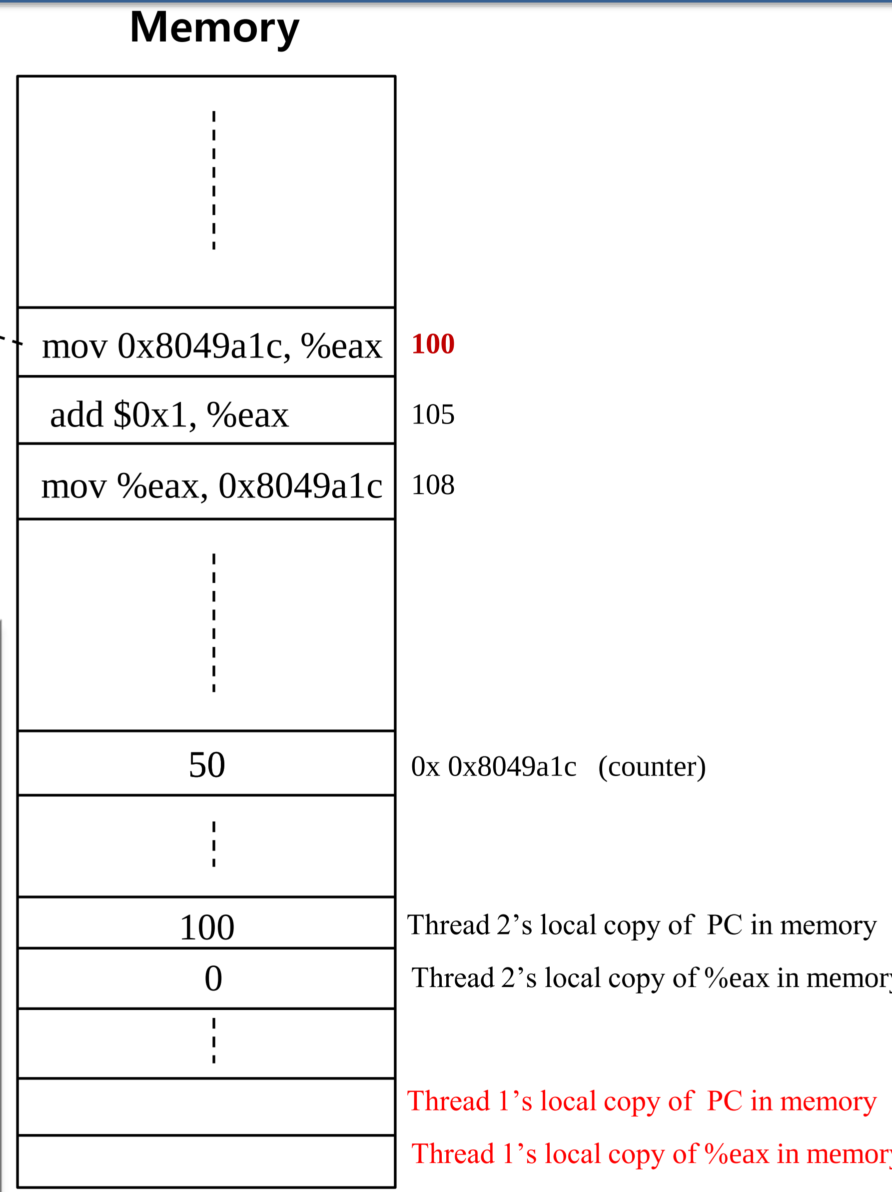
Memory



OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

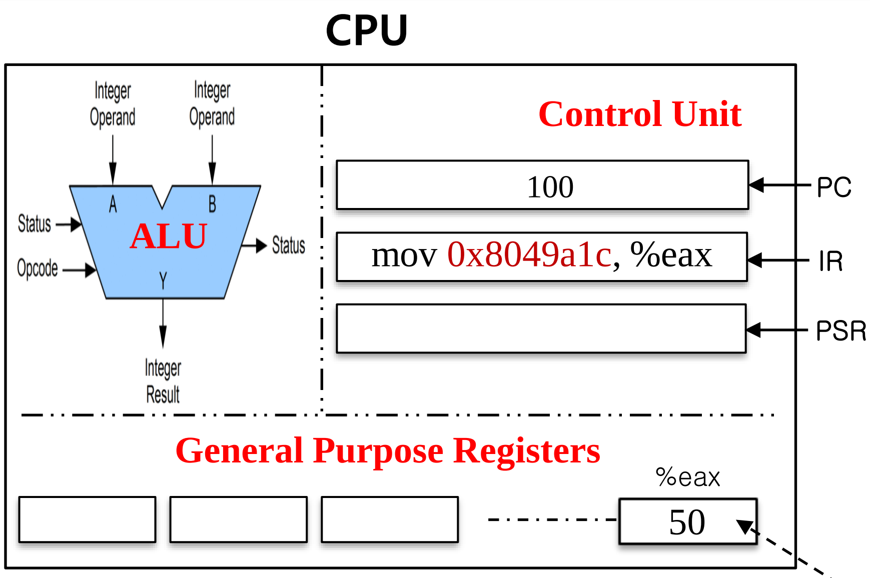
Step 1: Fetch instruction from the memory (PC->Address=100)

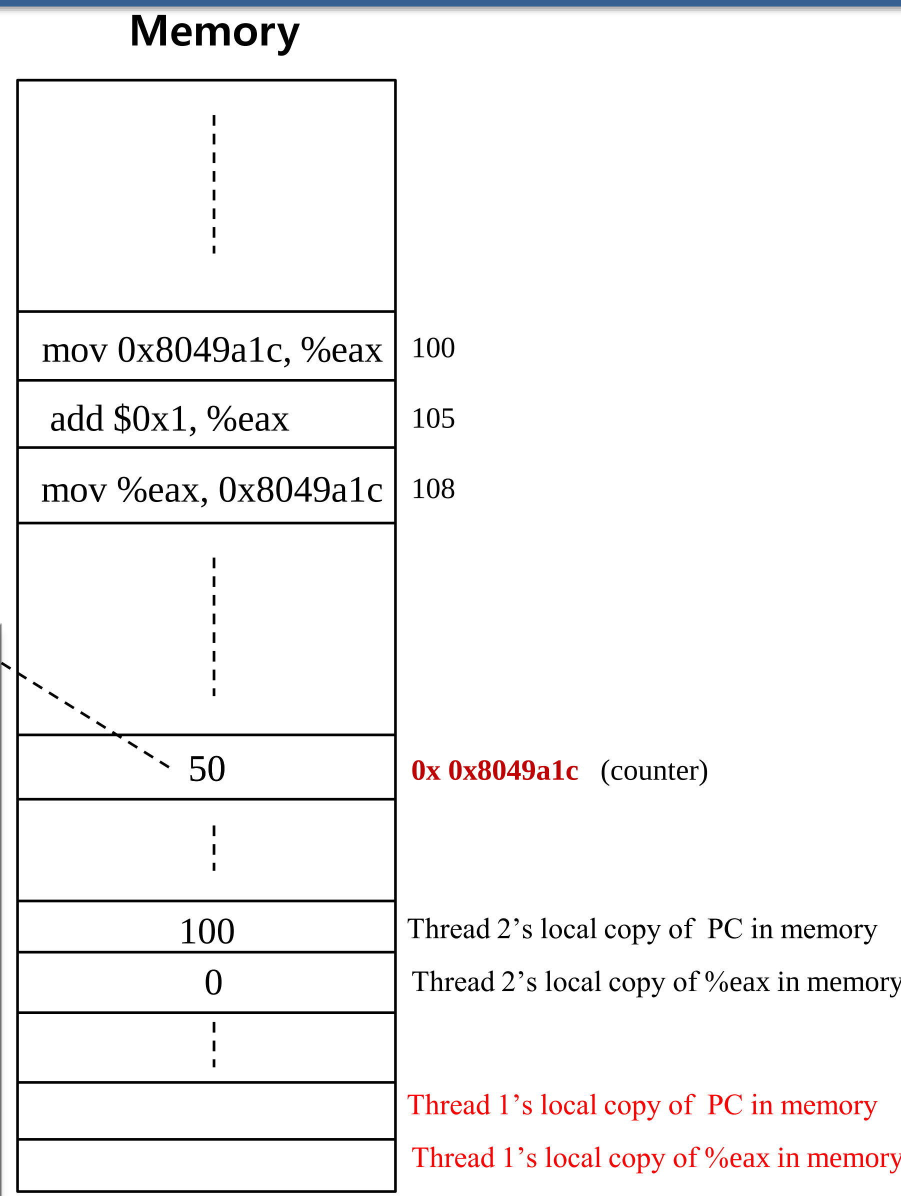




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Steps 2 & 3: Decode & Execution

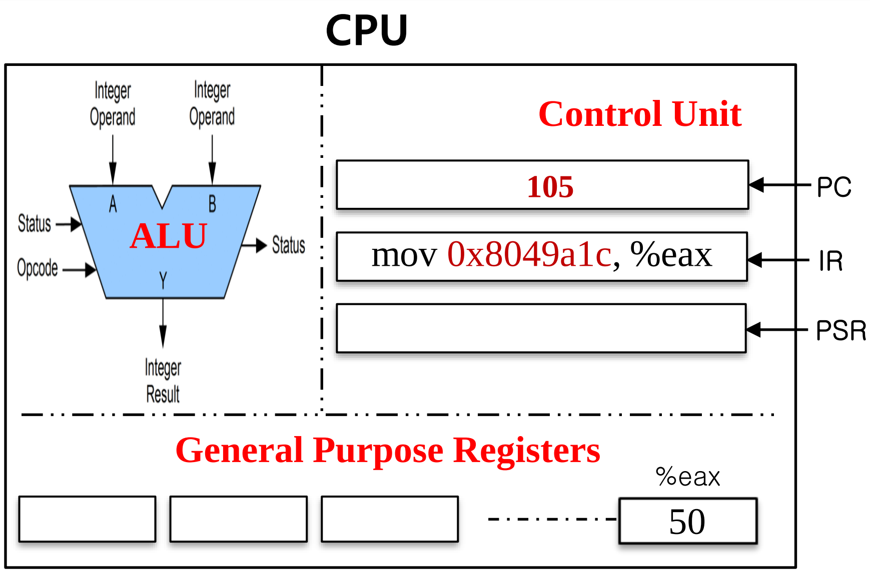




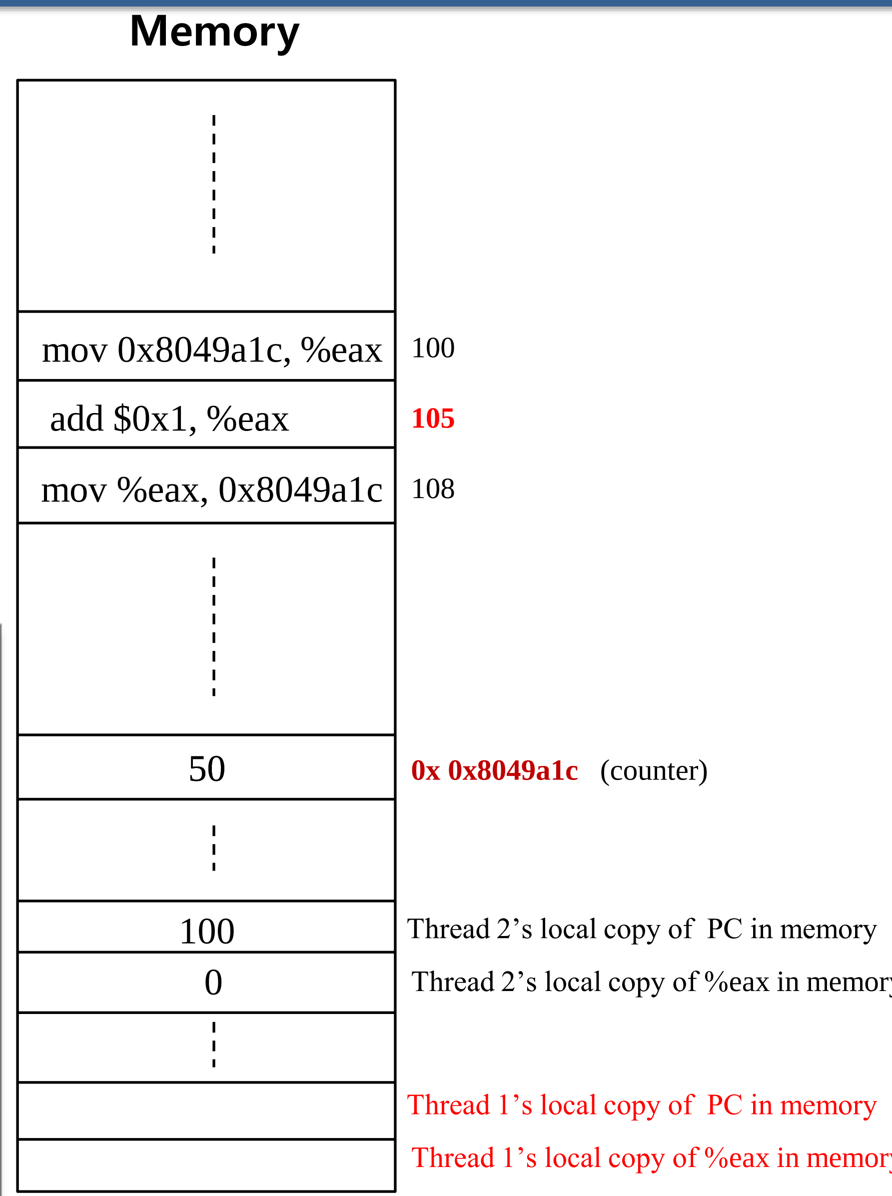
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Step 4: Increase PC (pointed to the next instruction in the memory)

CPU

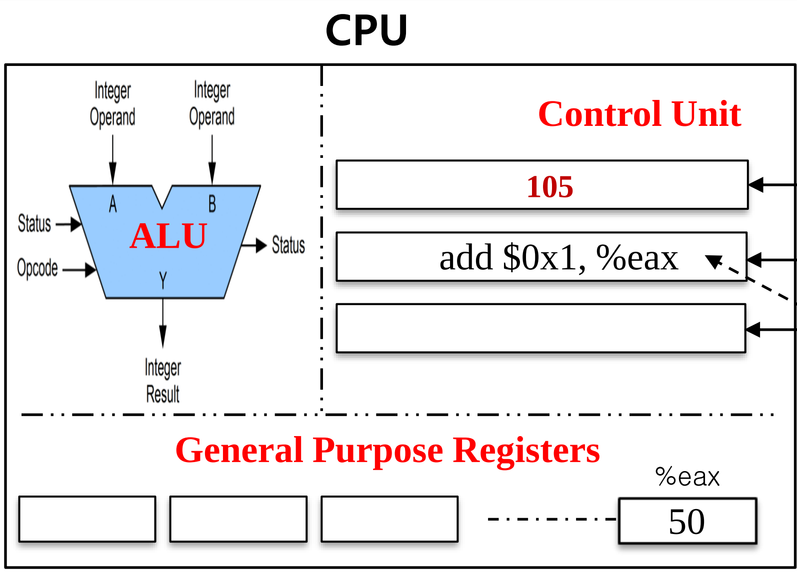


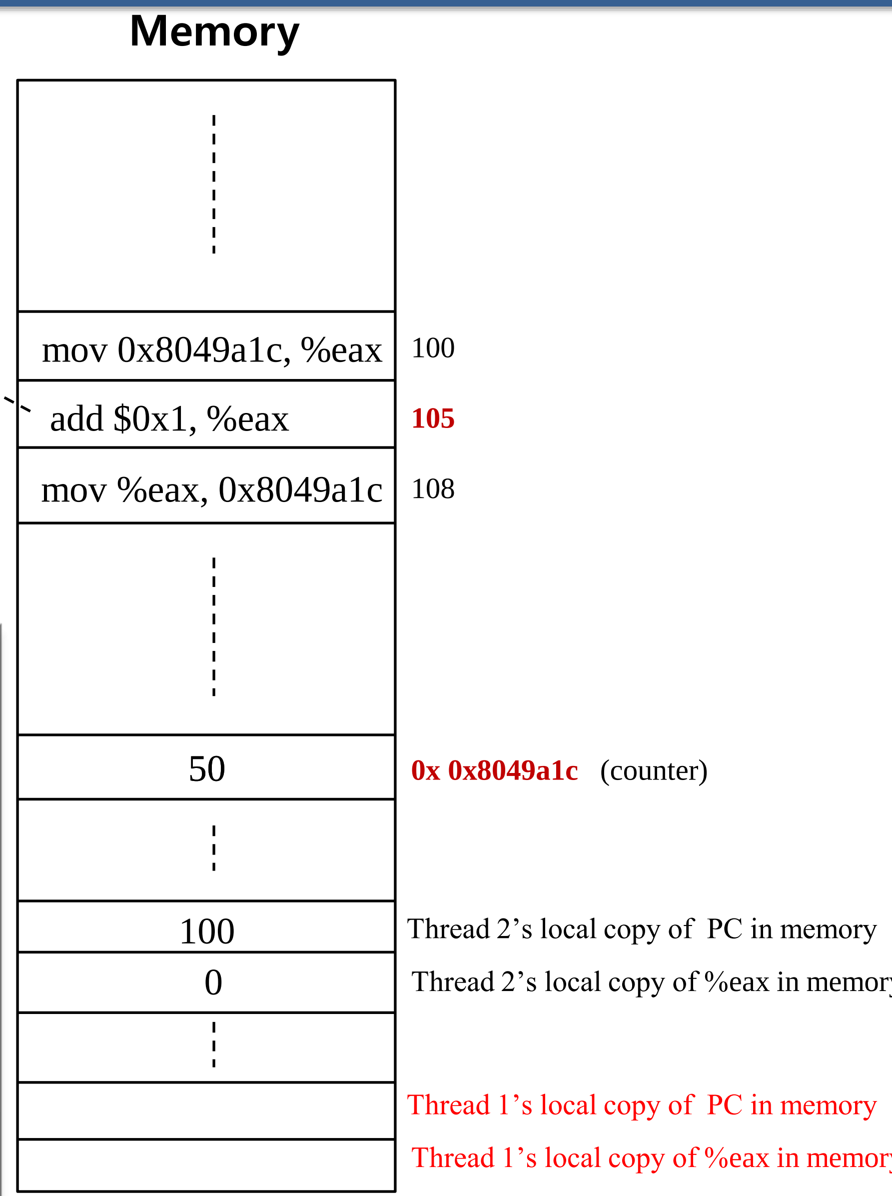
Memory



OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Step 1: Fetch instruction from the memory (PC->Address=105)

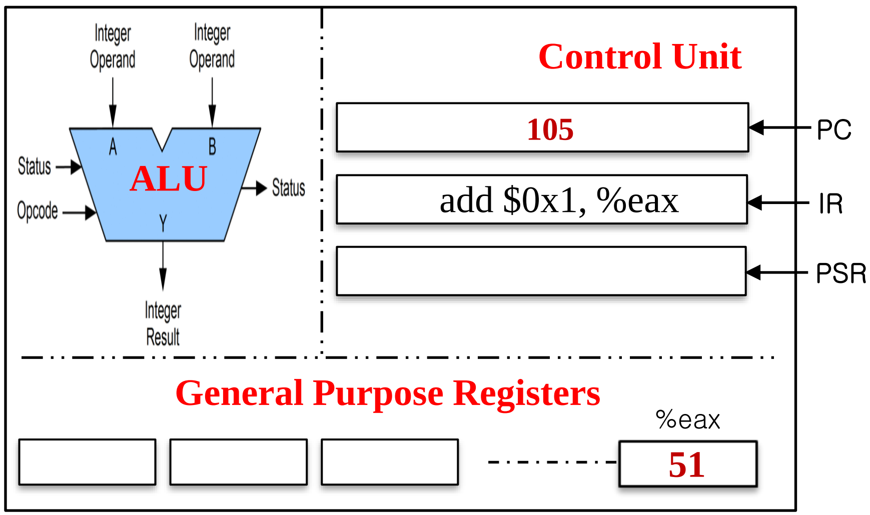




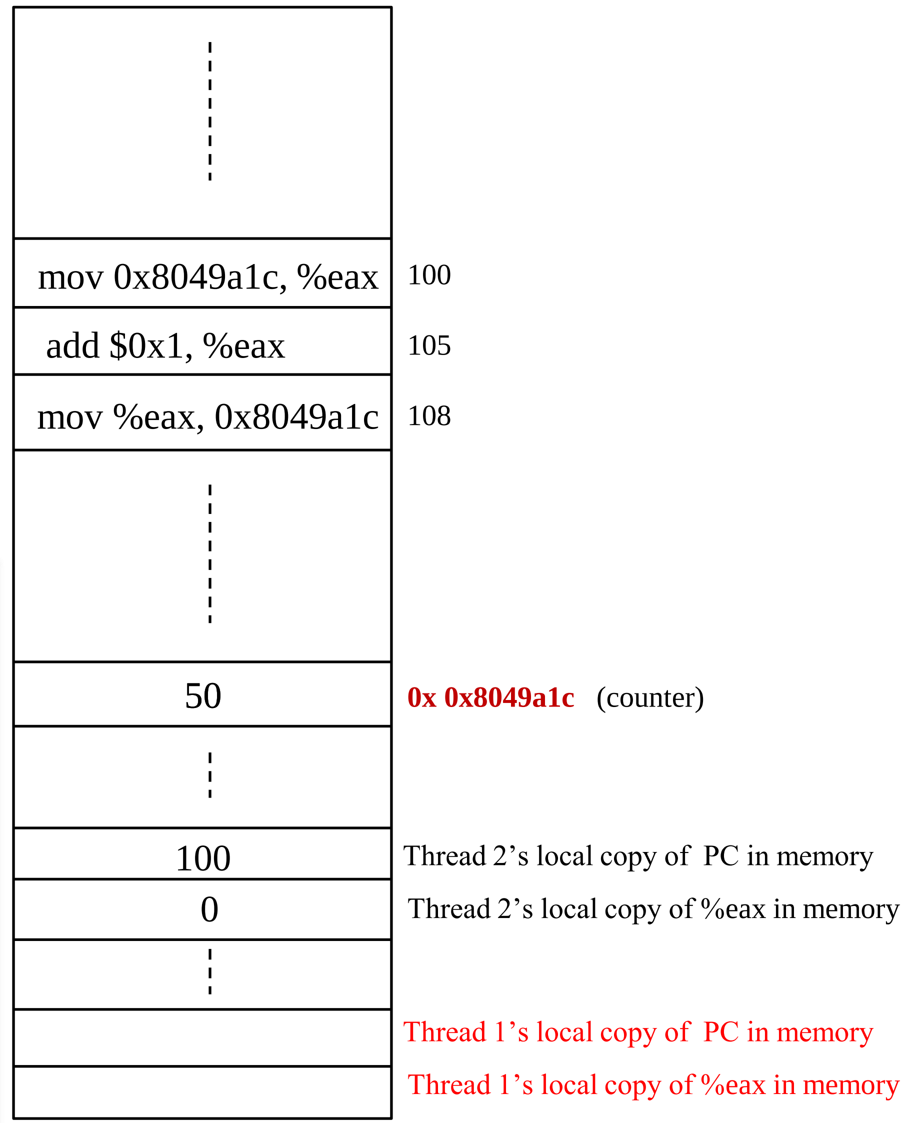
			(after instruction)		
OS	Thread1	Thread2	PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Steps 2 & 3: Decode & Execution

CPU

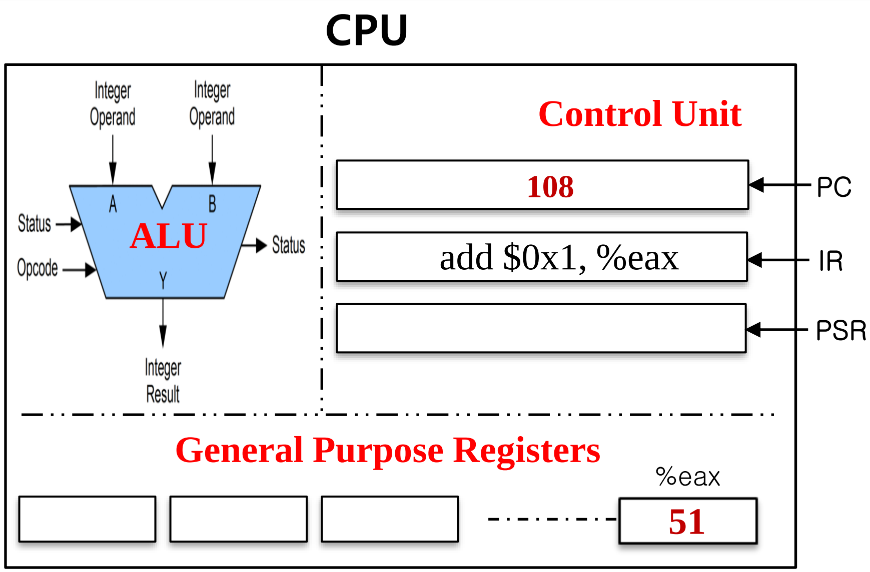


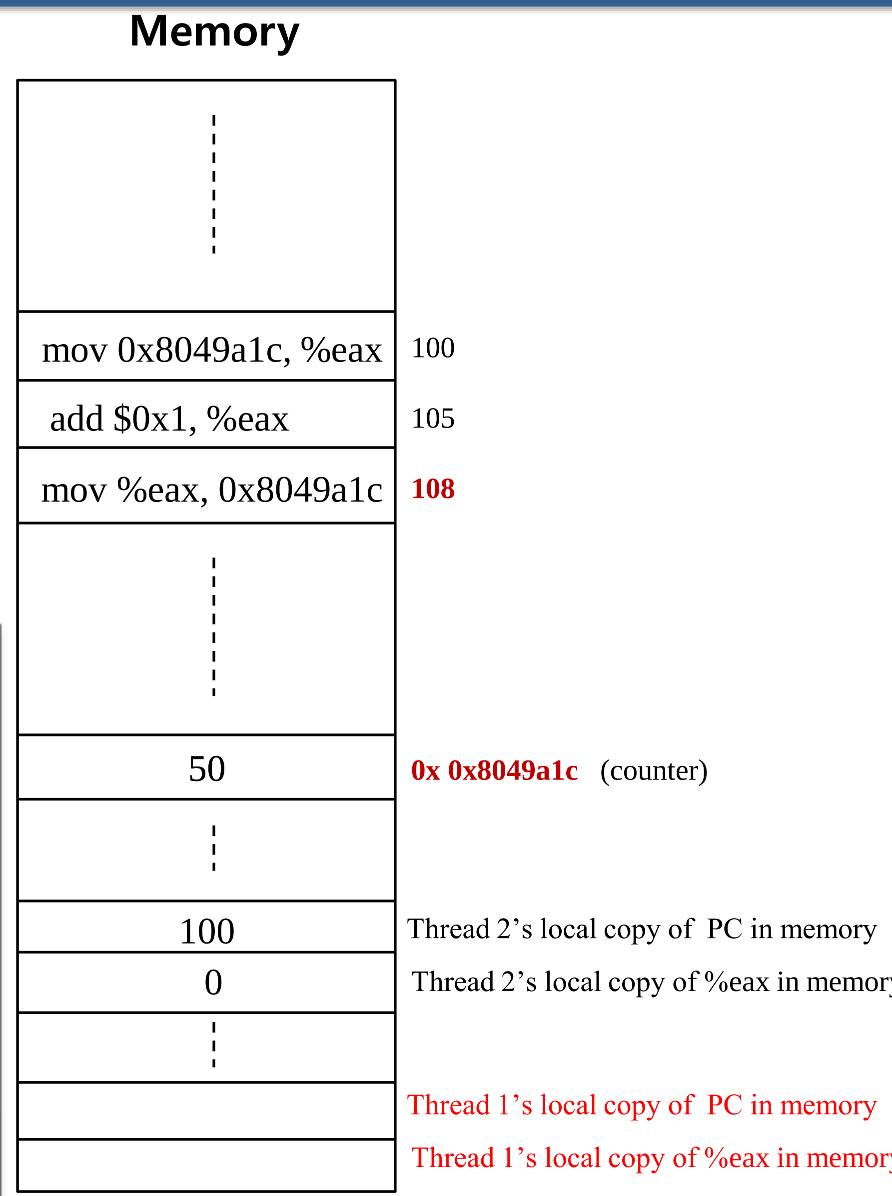
Memory



OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

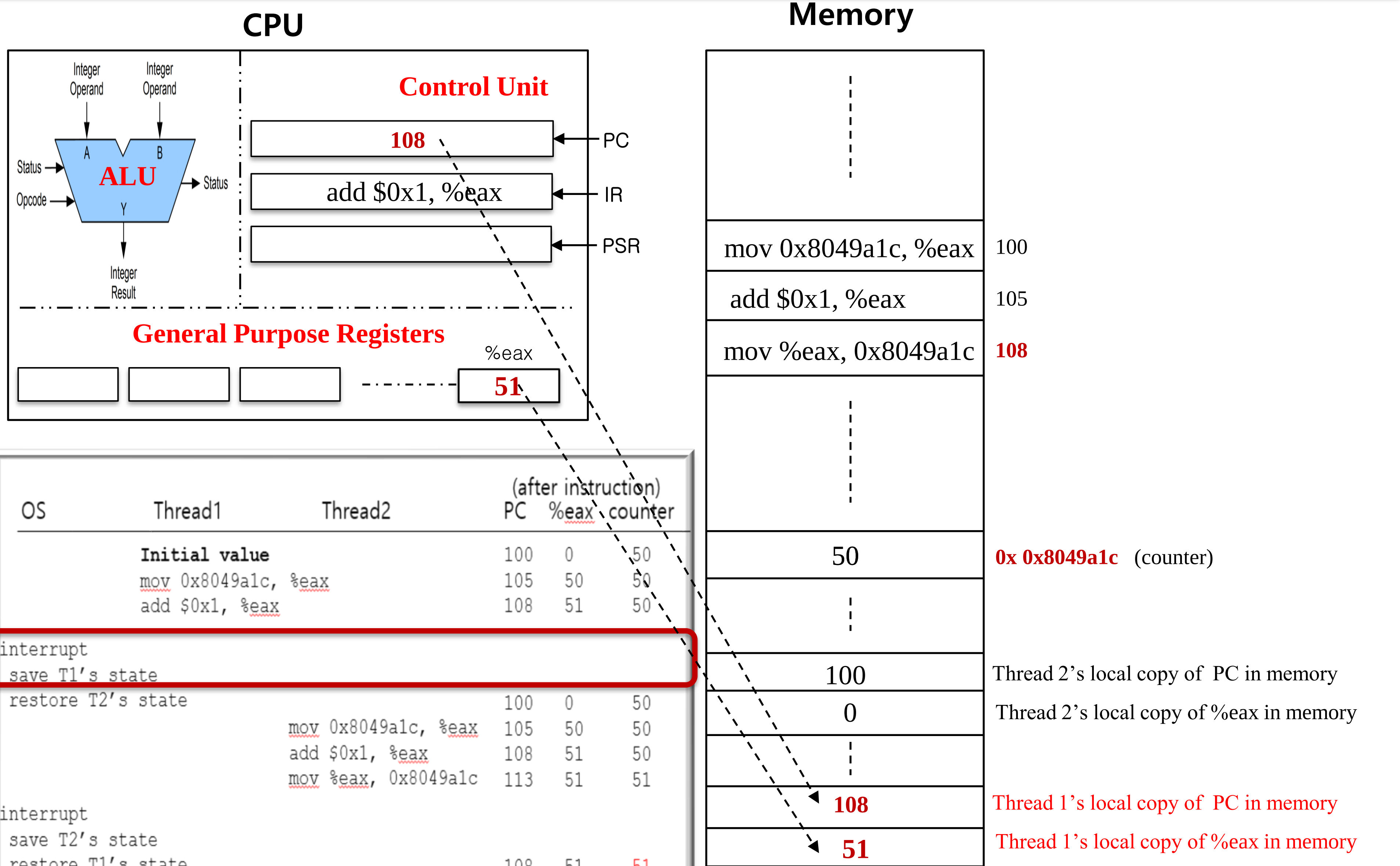
Step 3: Increase PC (pointed to the next instruction in the memory)





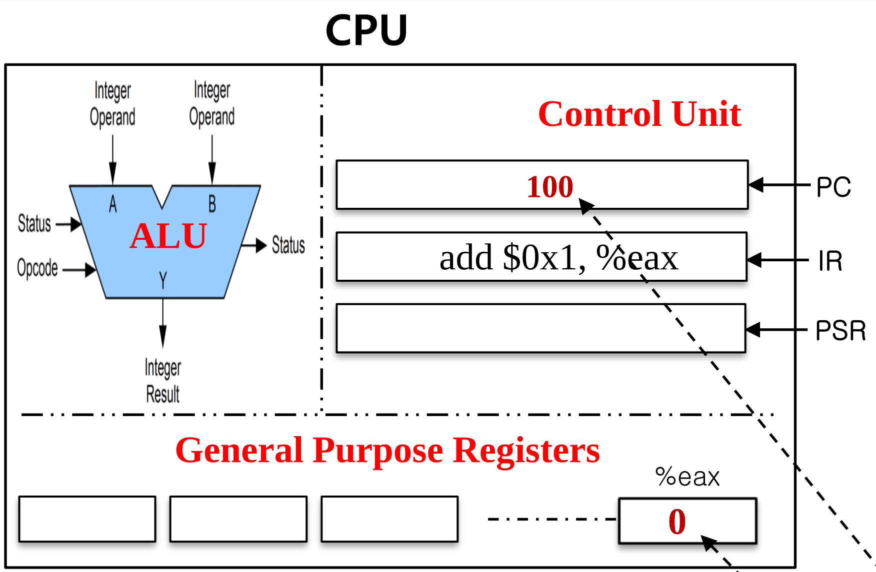
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

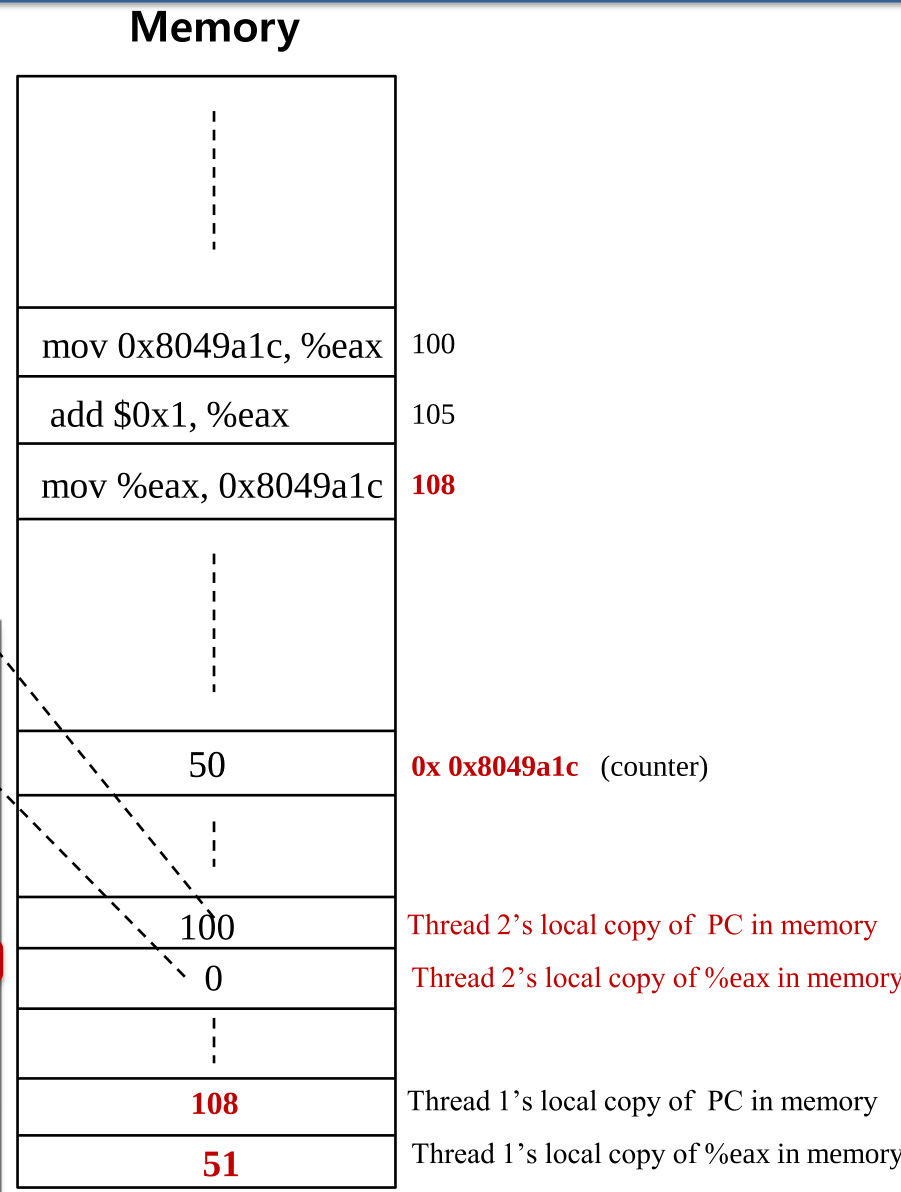
Interrupt occurred; Save Thread 1’s state



OS	Thread1	Thread2	PC (after instruction)	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1’s state					
restore T2’s state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2’s state					
restore T1’s state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

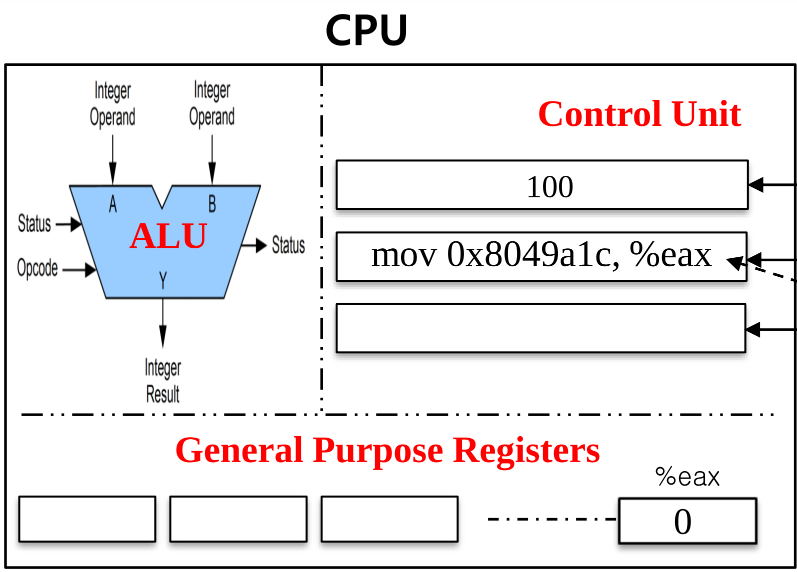
Restore Thread 2's state

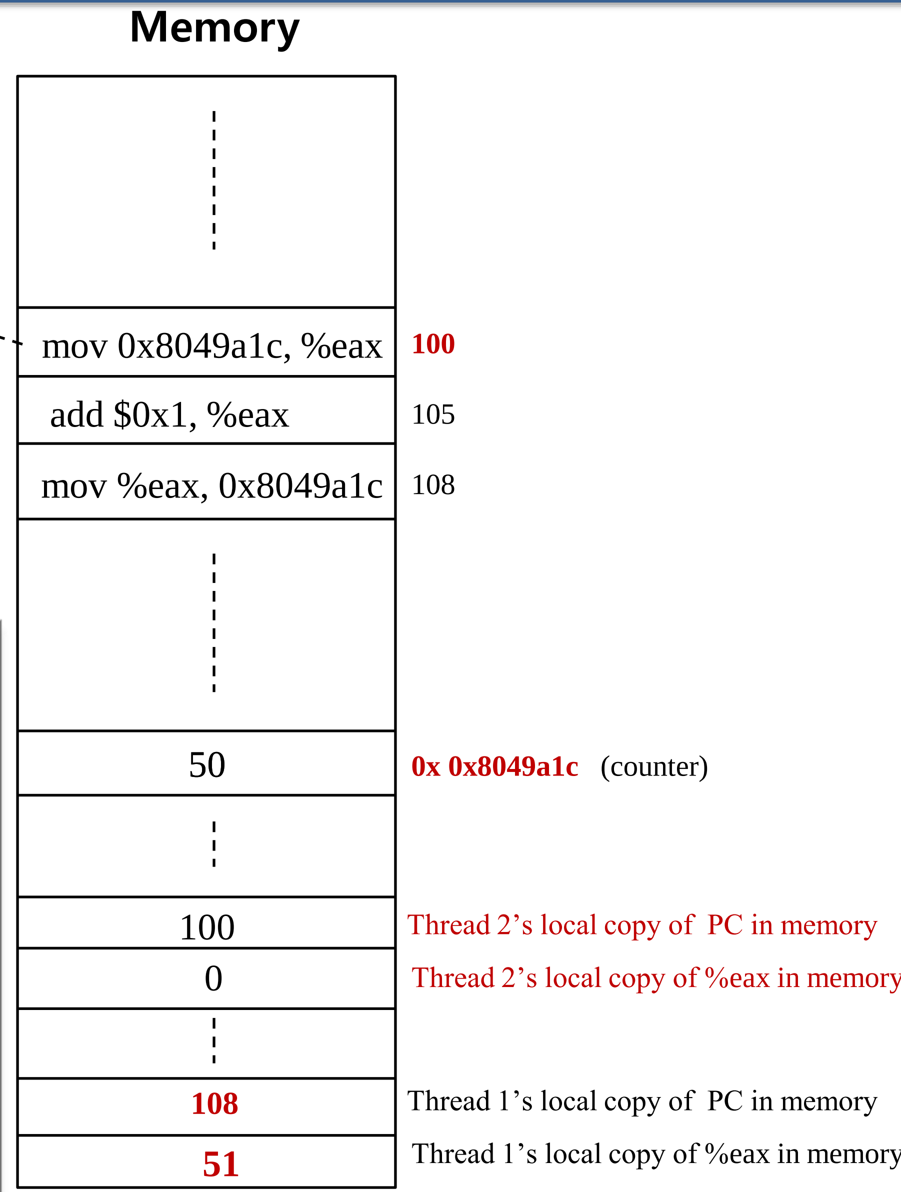




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

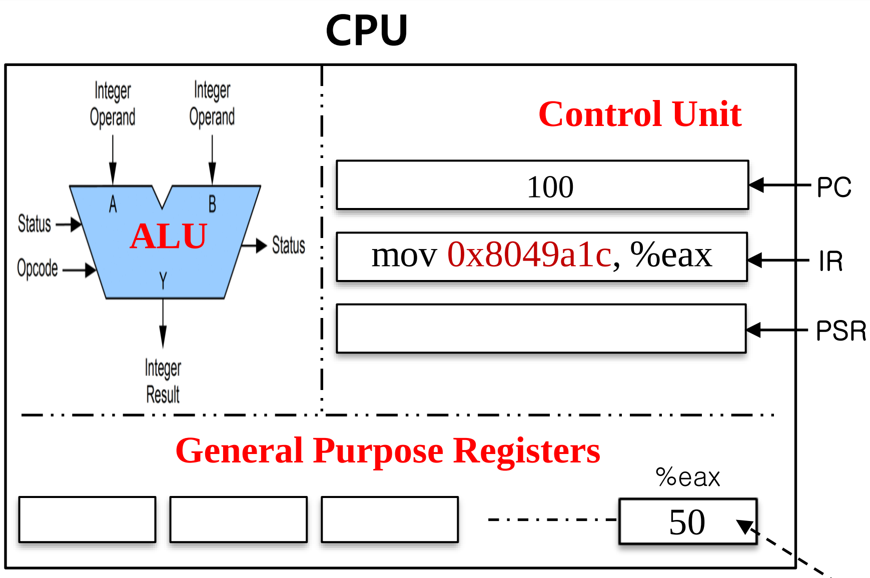
Step 1: Fetch instruction from the memory (PC->Address=100)

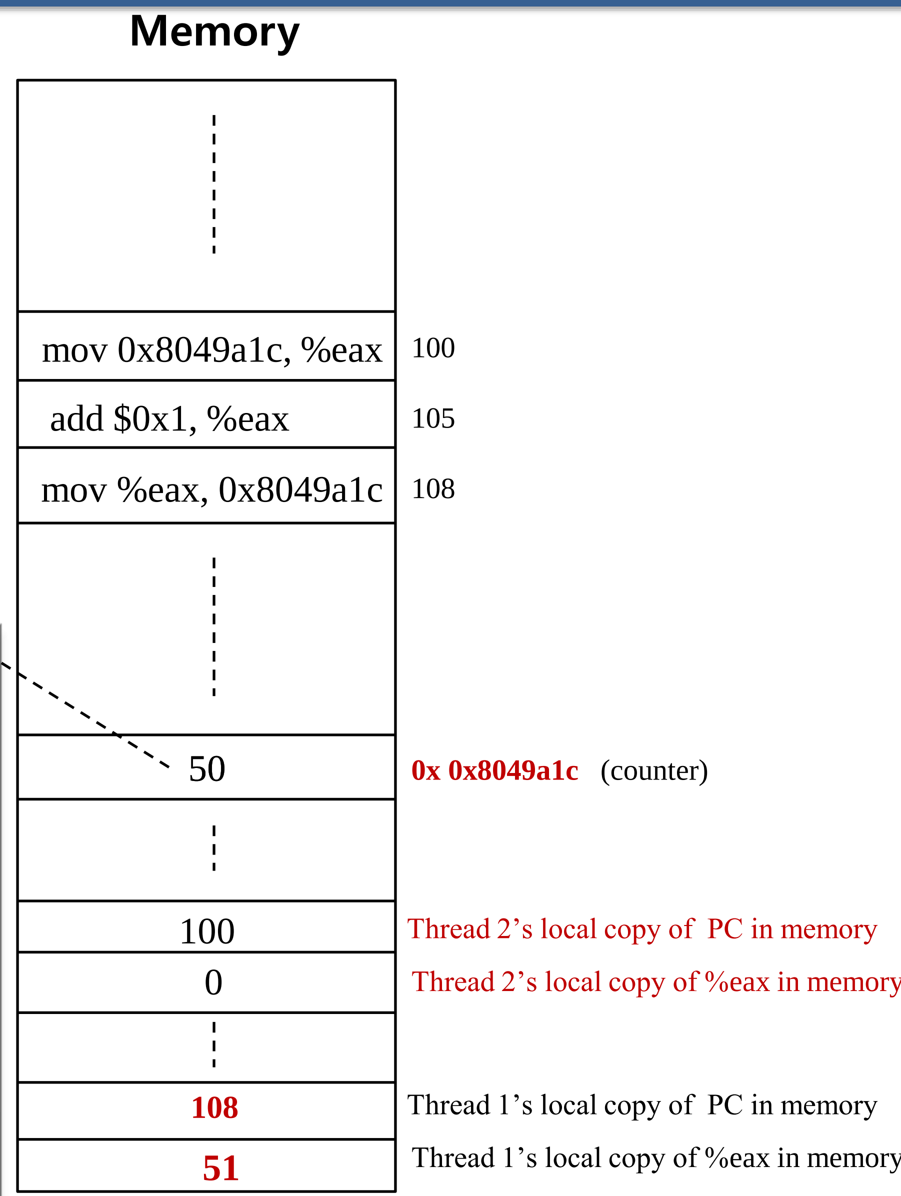




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

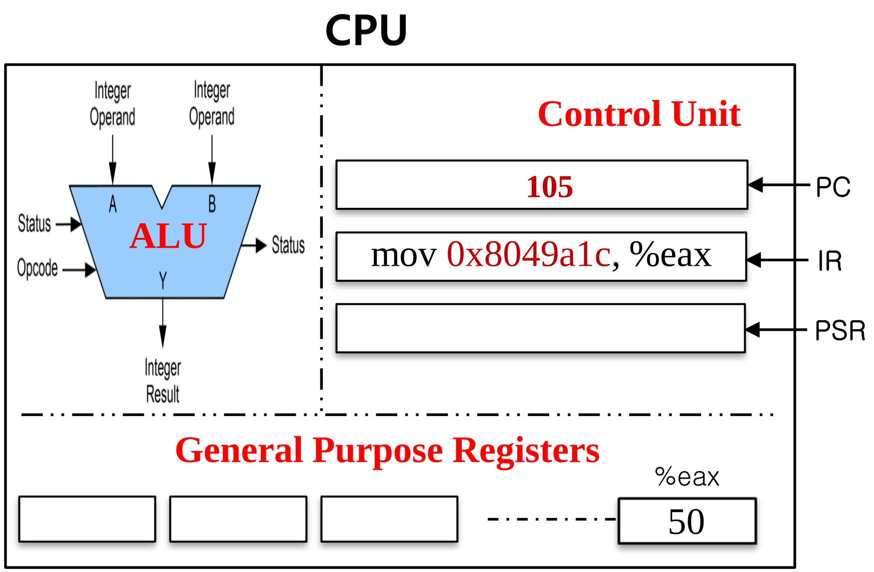
Steps 2 & 3: Decode & Execution

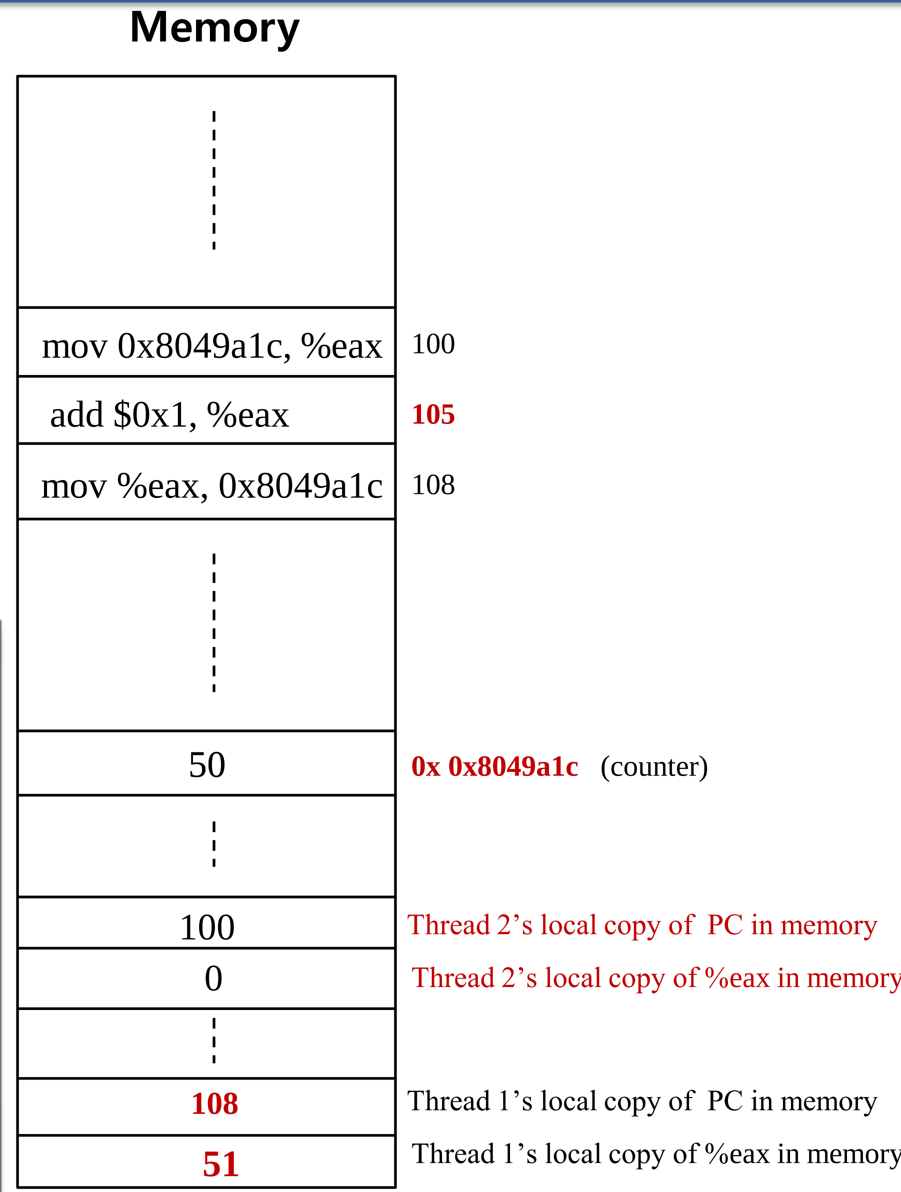




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

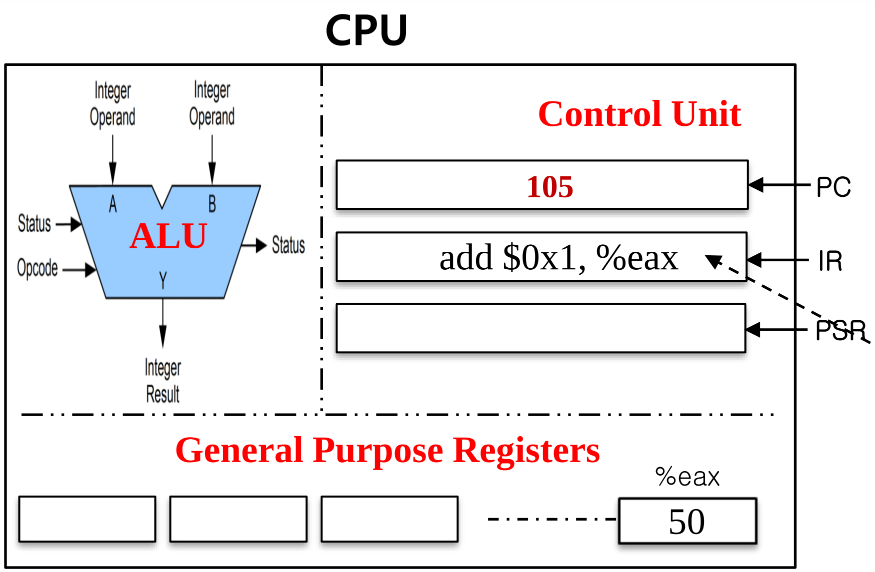
Step 4: Increase PC (pointed to the next instruction in the memory)

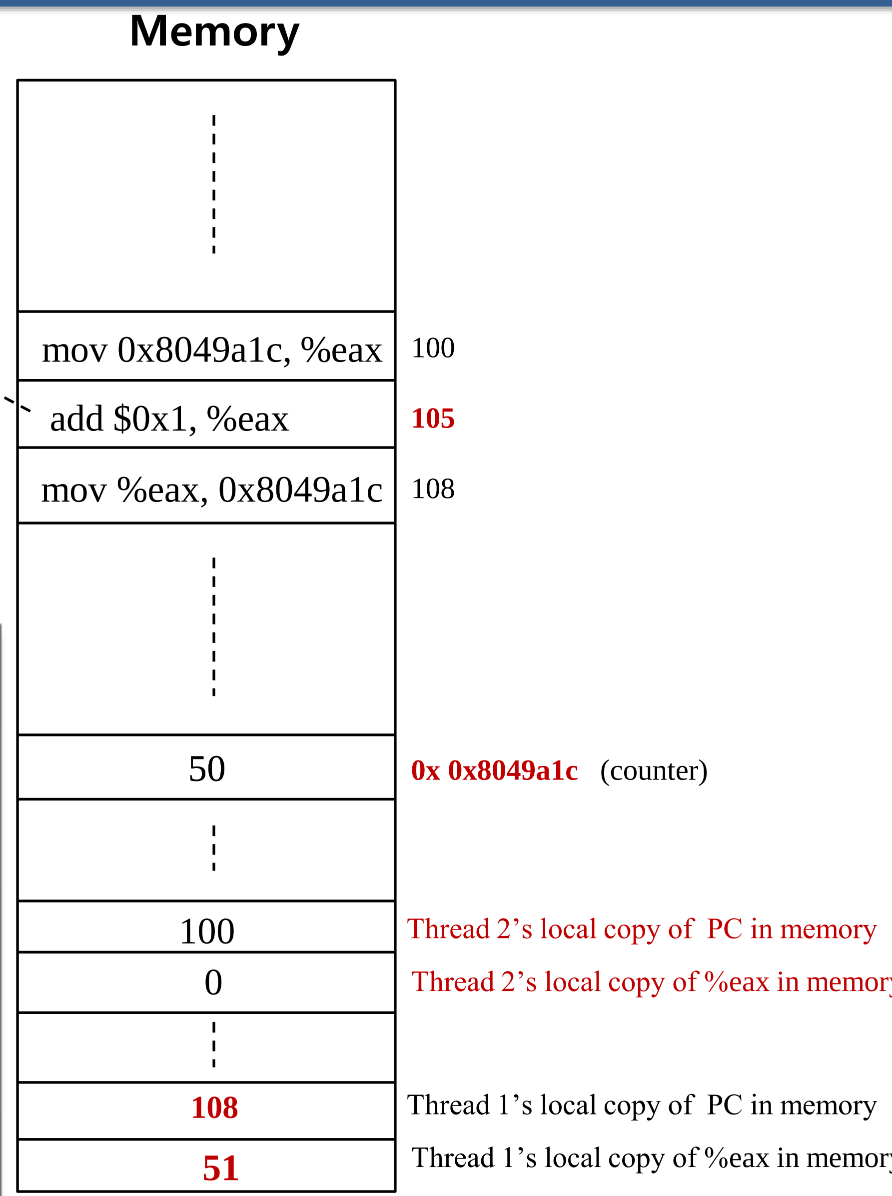




OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

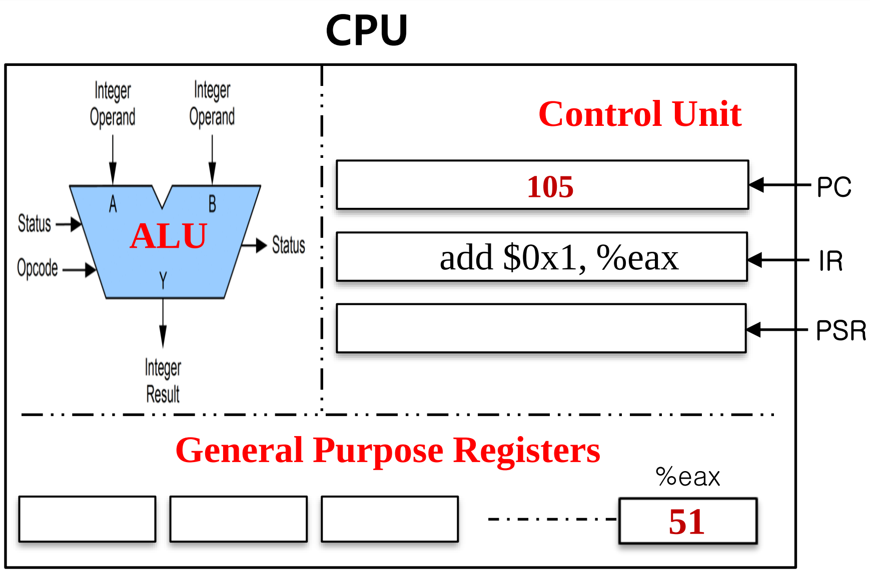
Step 1: Fetch instruction from the memory (PC->Address=105)

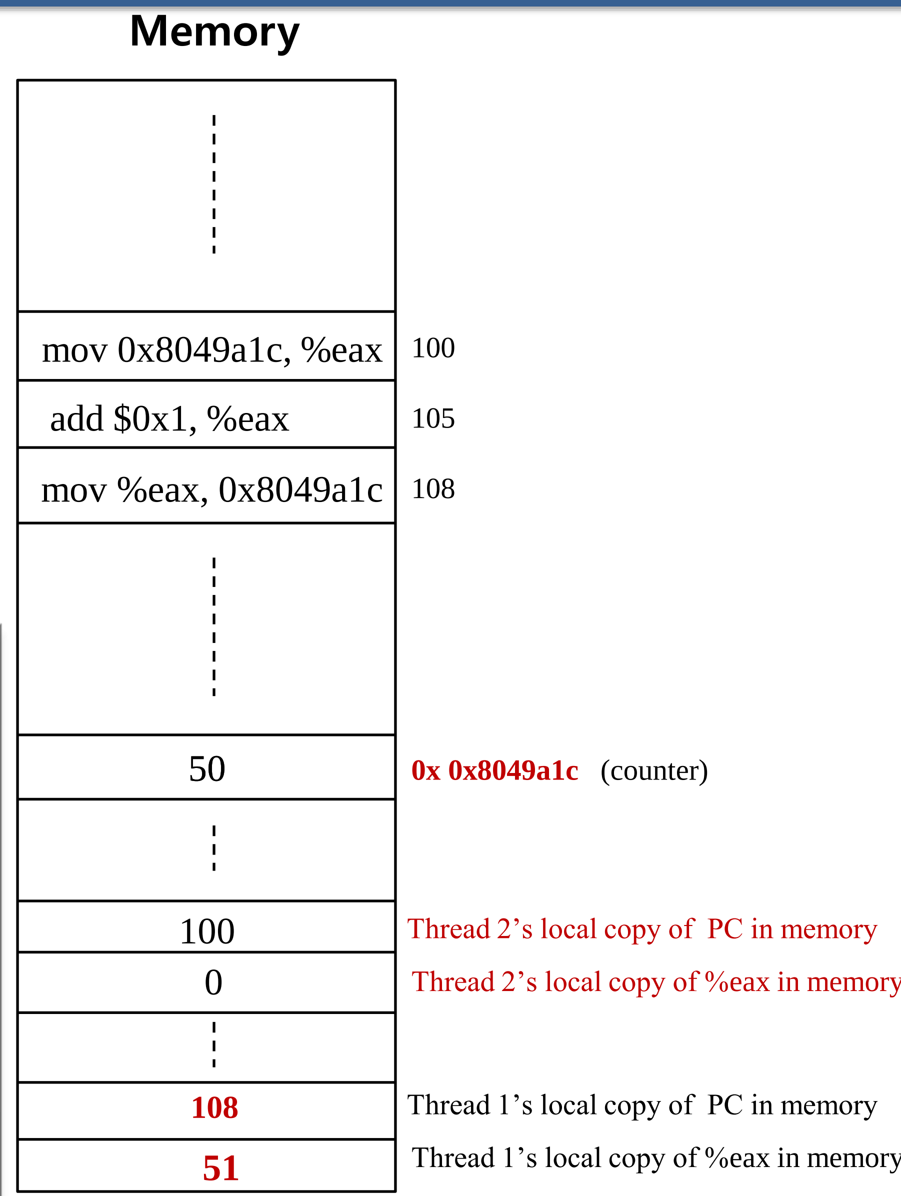




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Steps 2 & 3: Decode & Execution

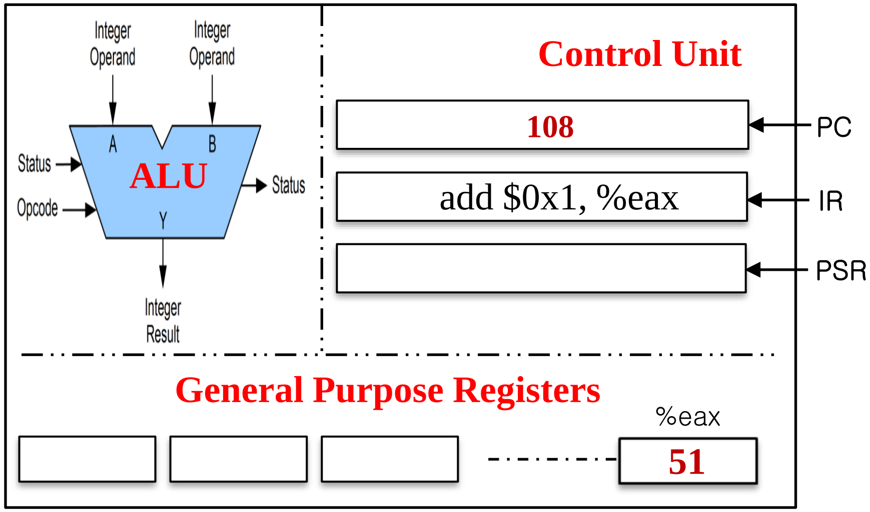




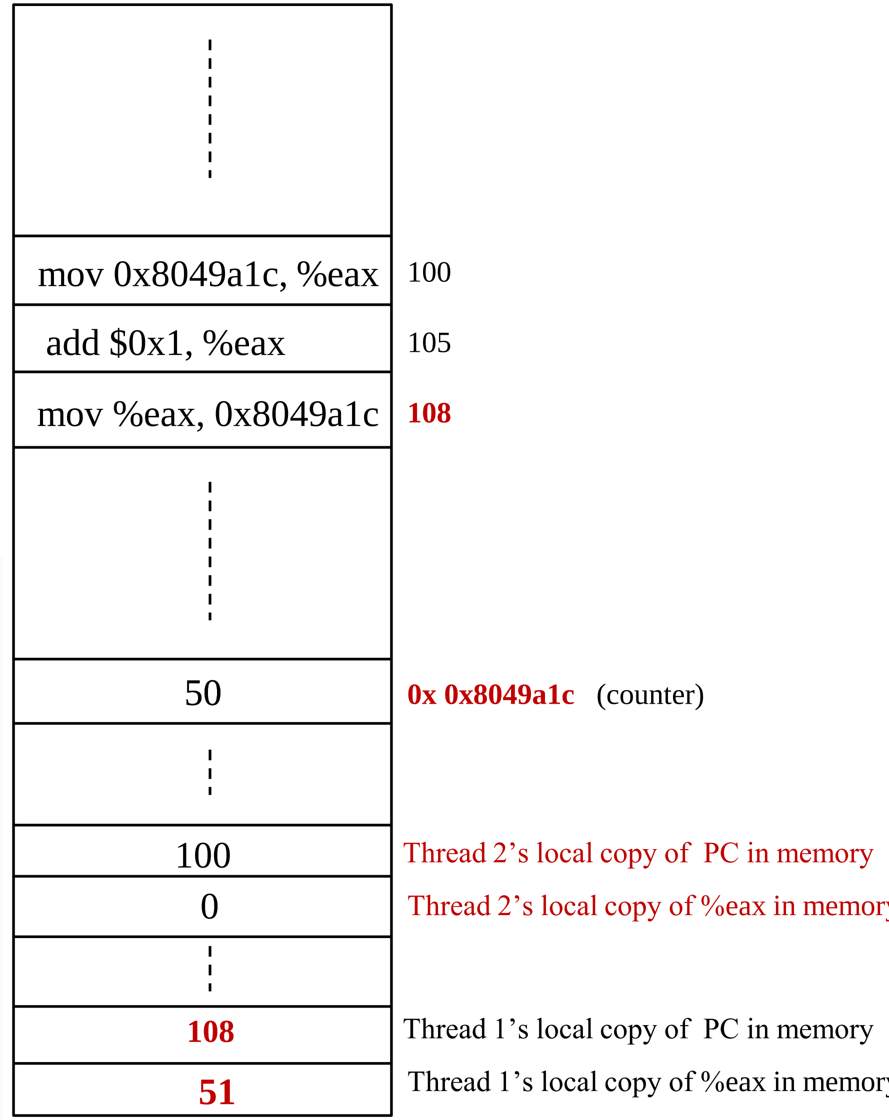
			(after instruction)		
OS	Thread1	Thread2	PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Step 4: Increase PC (pointed to the next instruction in the memory)

CPU

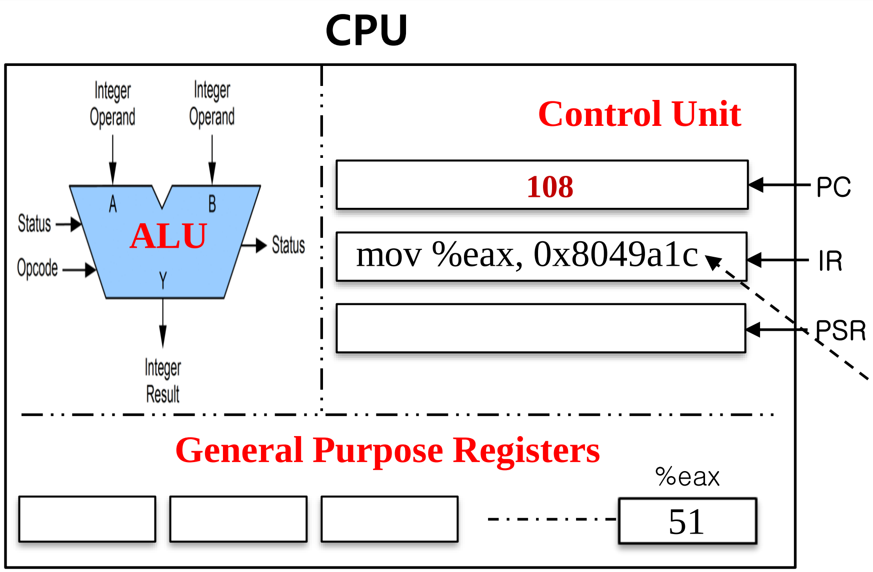


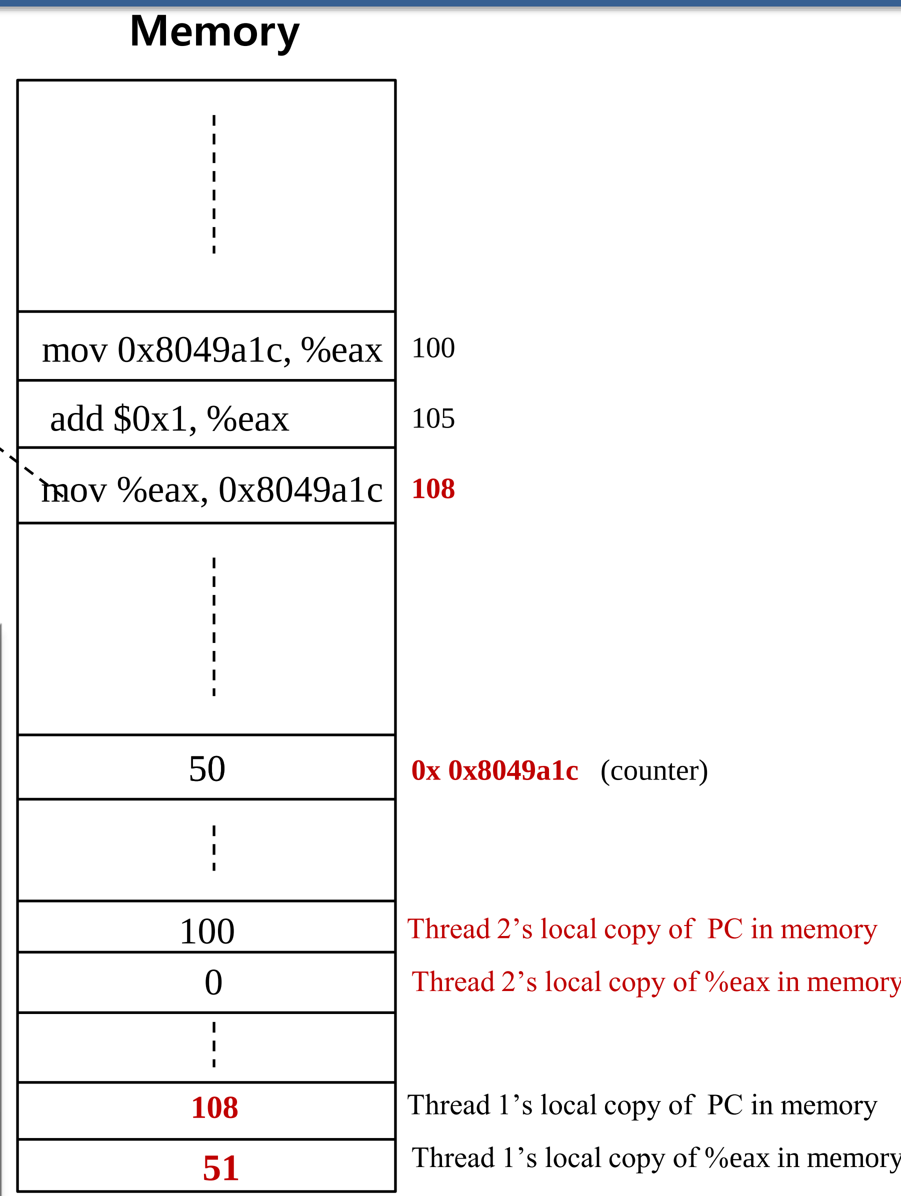
Memory



OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

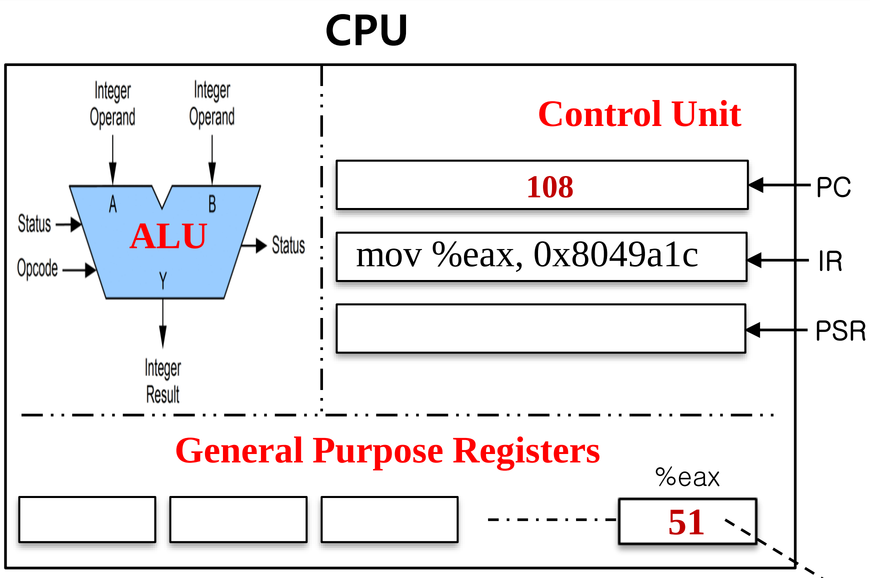
Step 1: Fetch instruction from the memory (PC->Address=108)

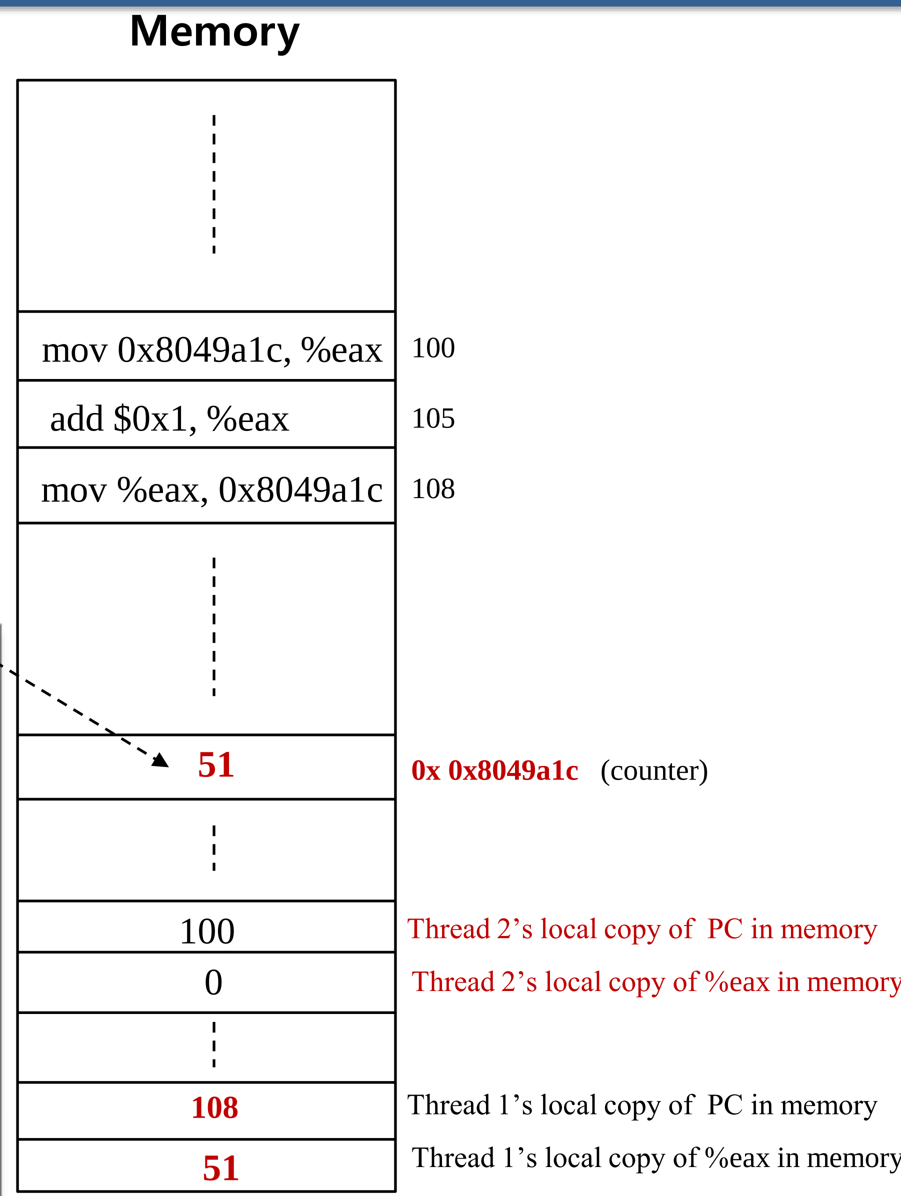




			(after instruction)		
OS	Thread1	Thread2	PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

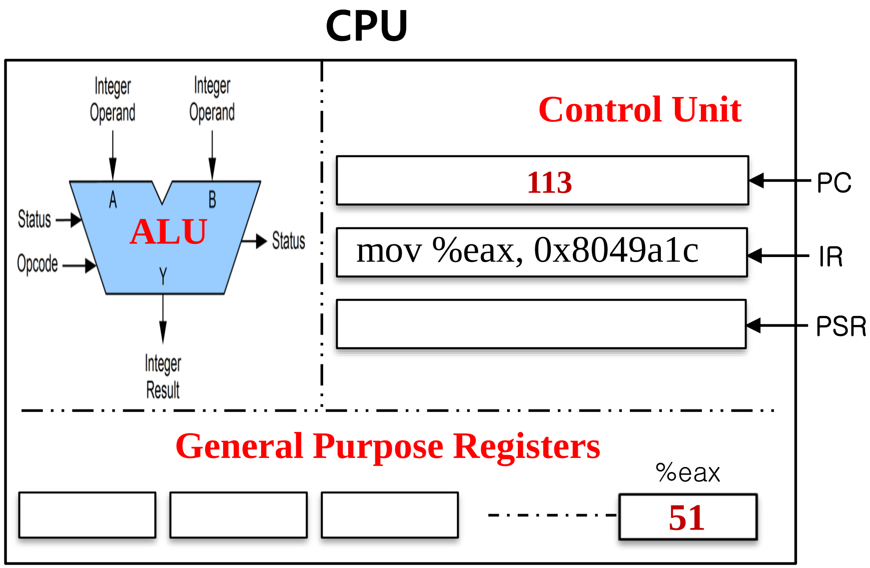
Steps 2 & 3: Decode & Execution

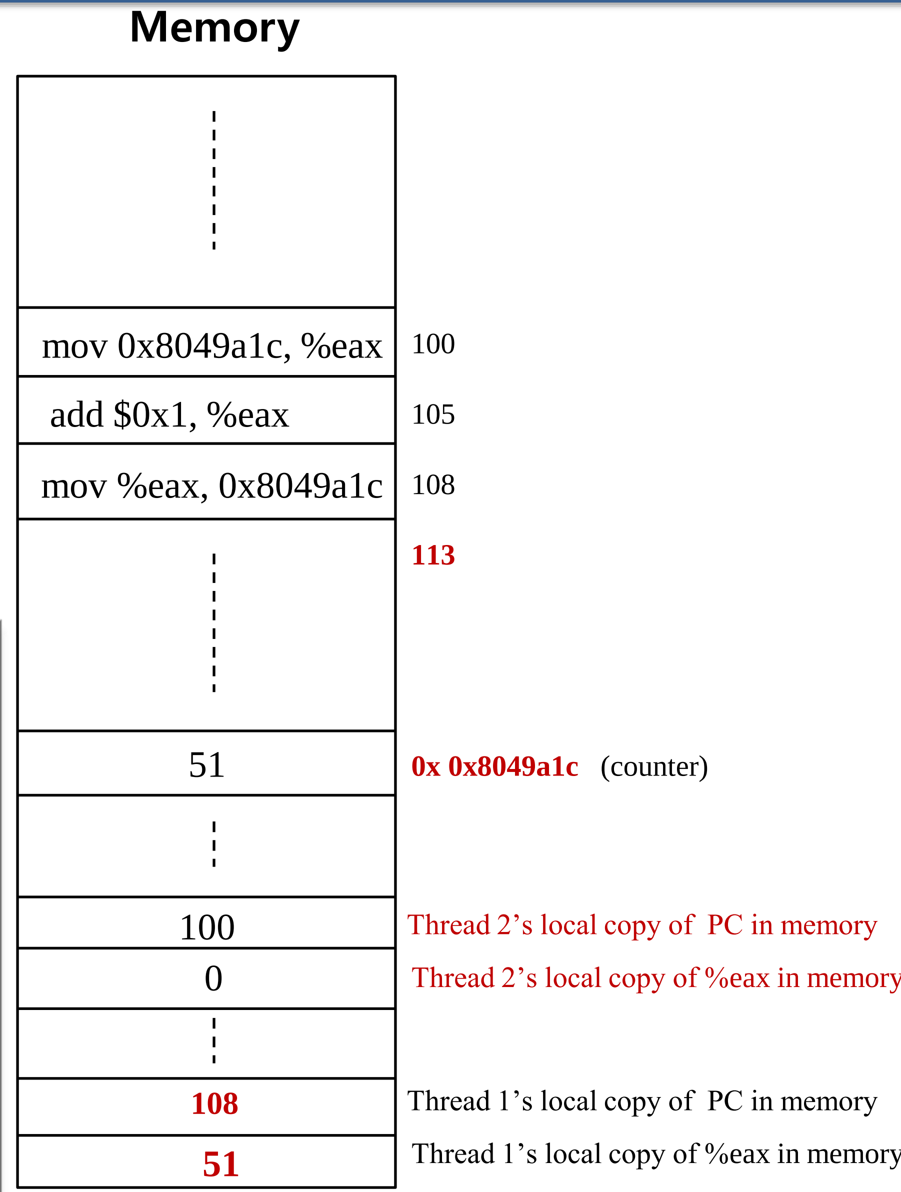




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	51

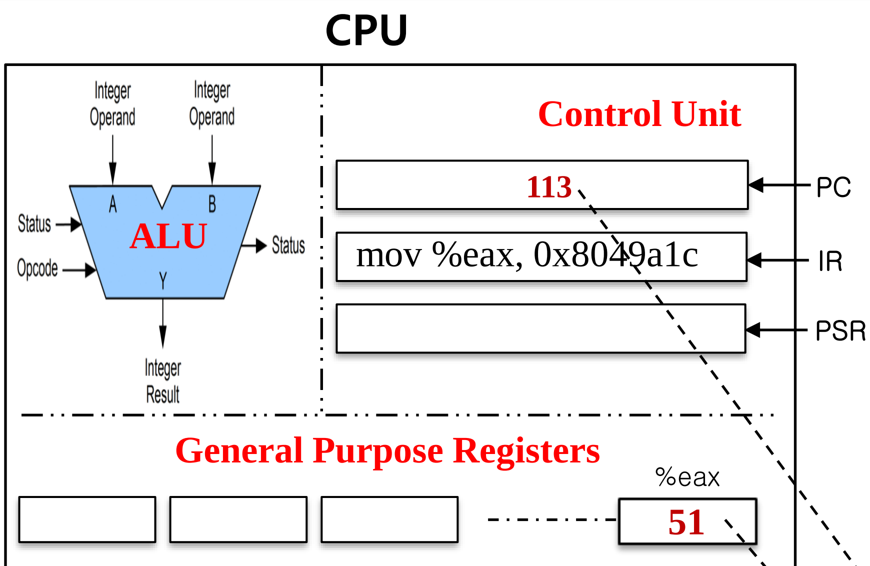
Step 4: Increase PC (pointed to the next instruction in the memory)

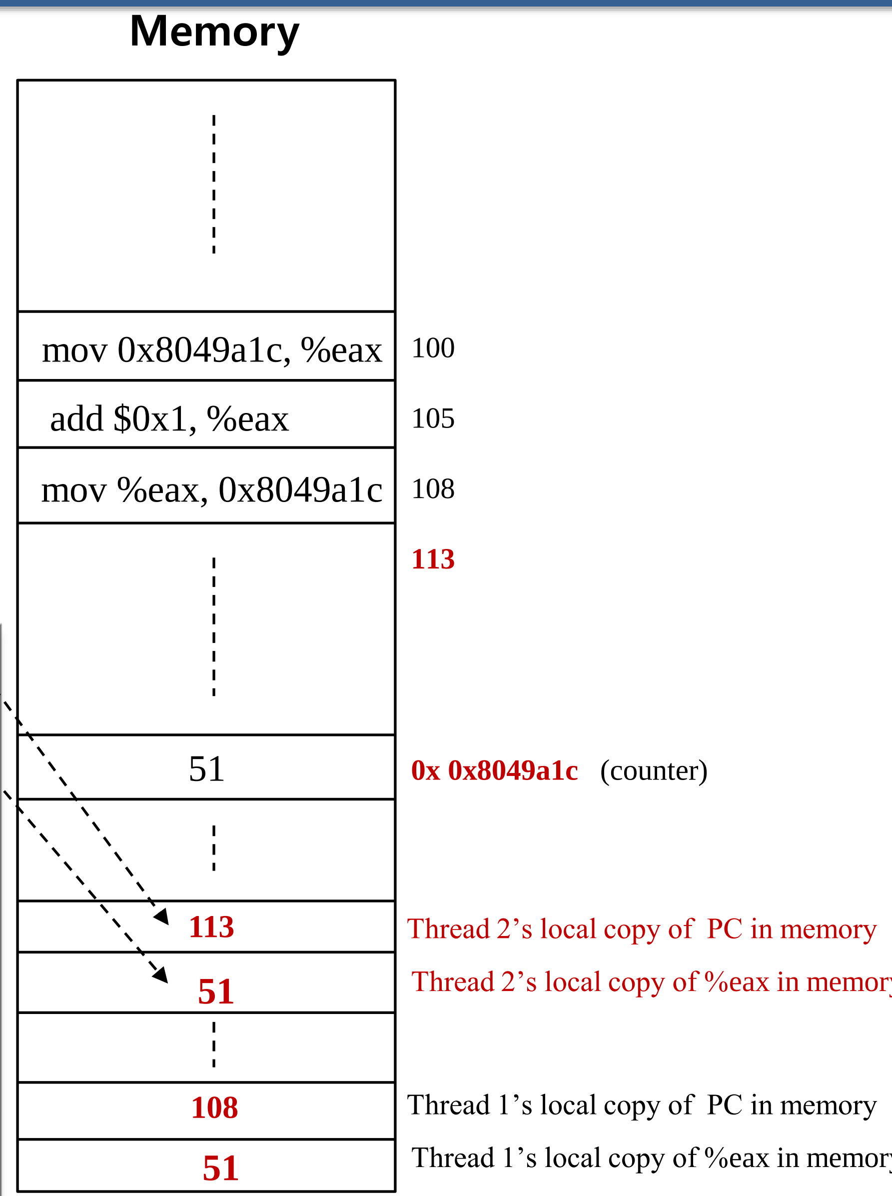




OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	51

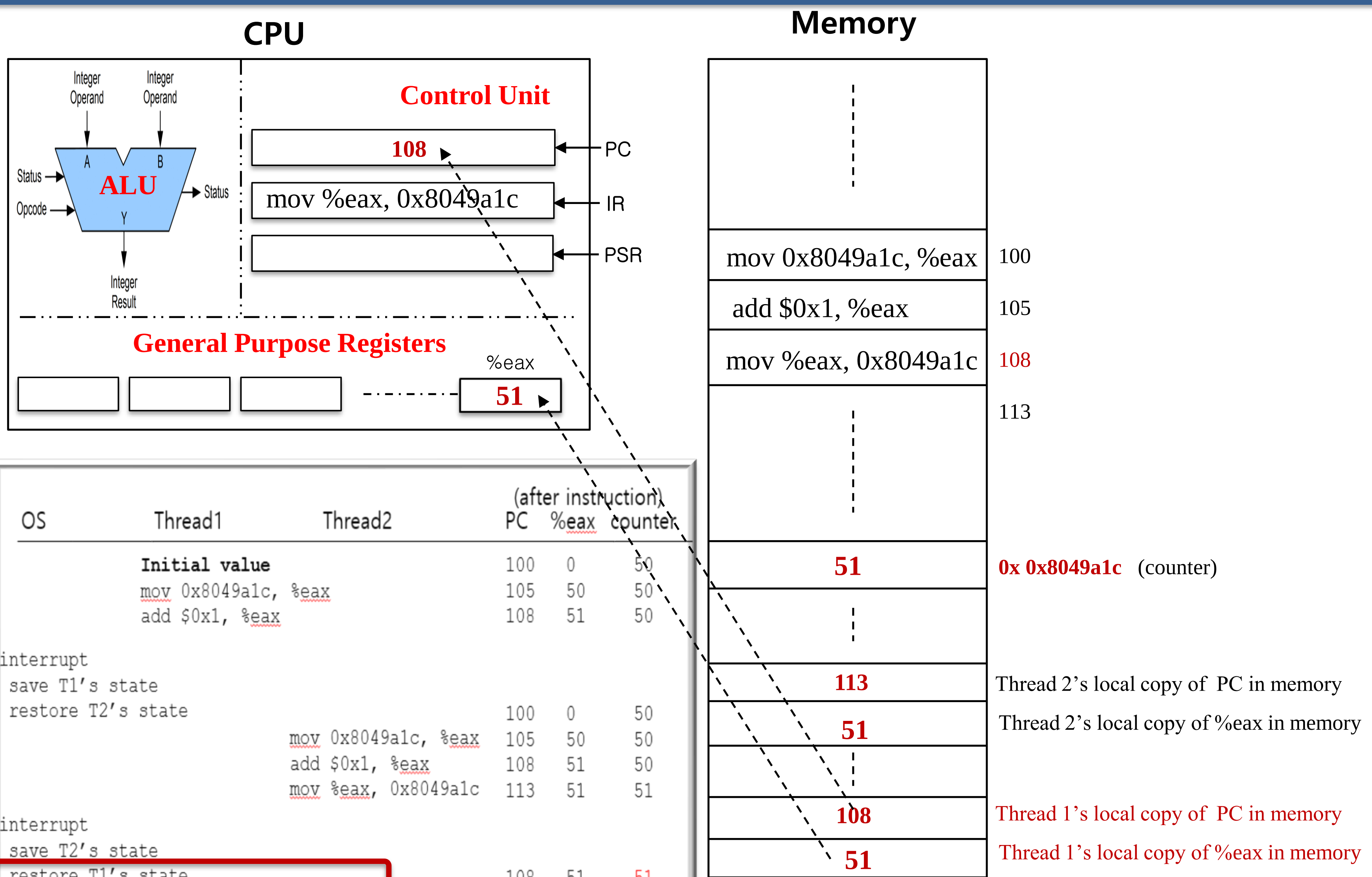
Interrupt occurred; Save Thread 2's state





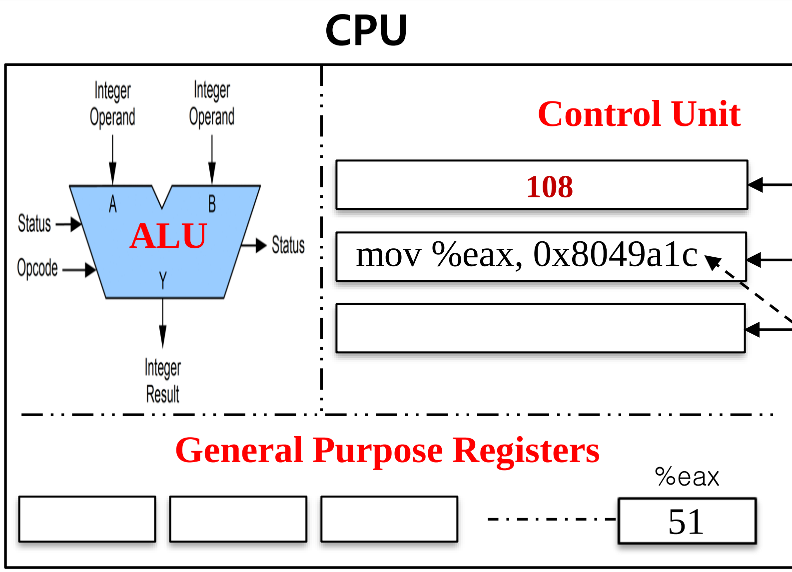
OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	Initial value		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	`mov %eax, 0x8049a1c`		113	51	**51**

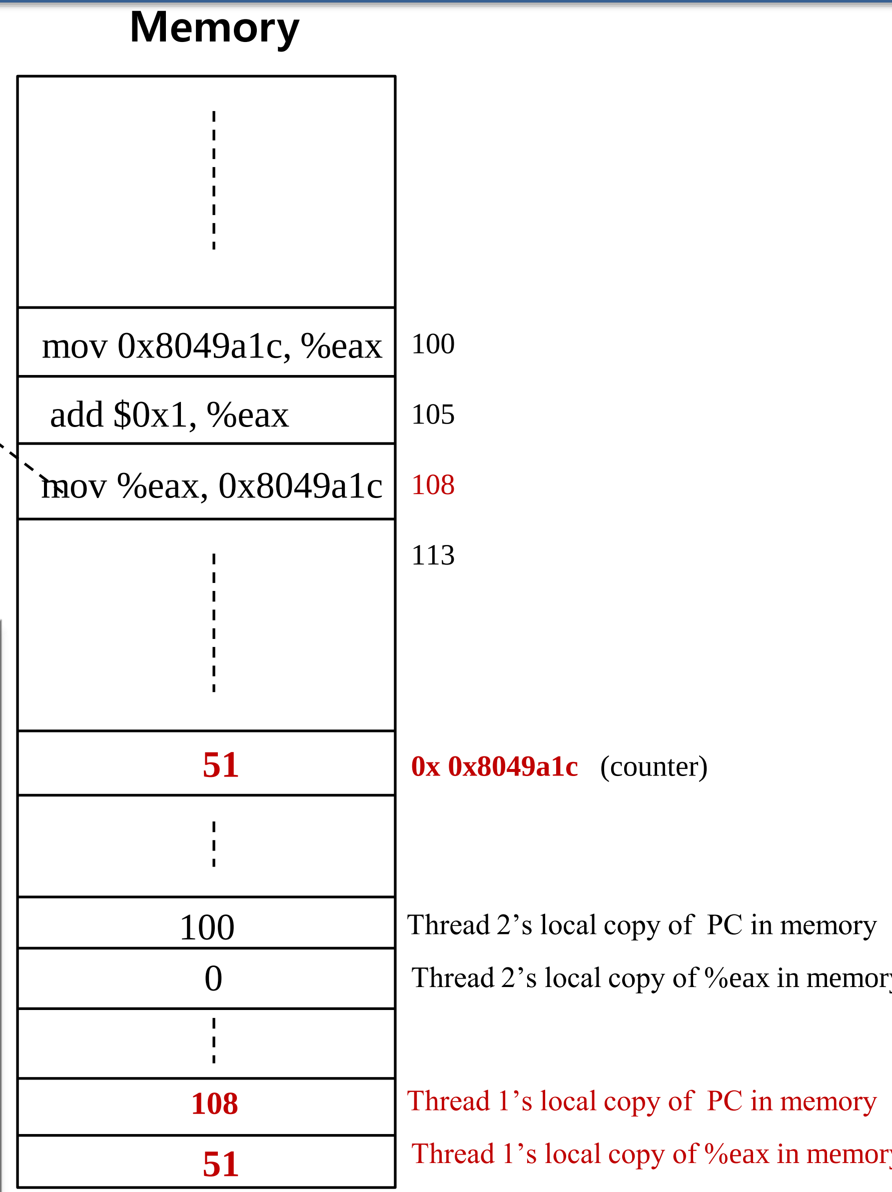
Restore Thread 1's state



OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

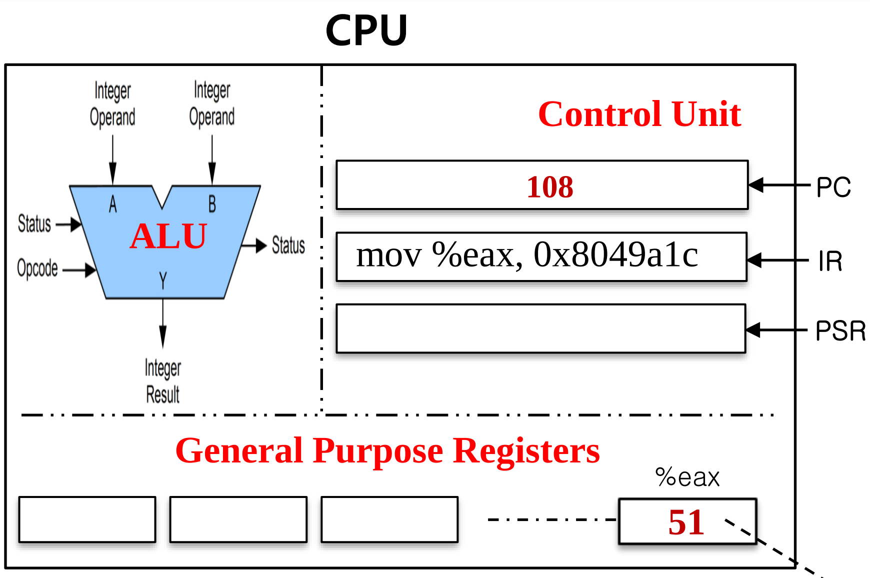
Step 1: Fetch instruction from the memory (PC->Address=108)

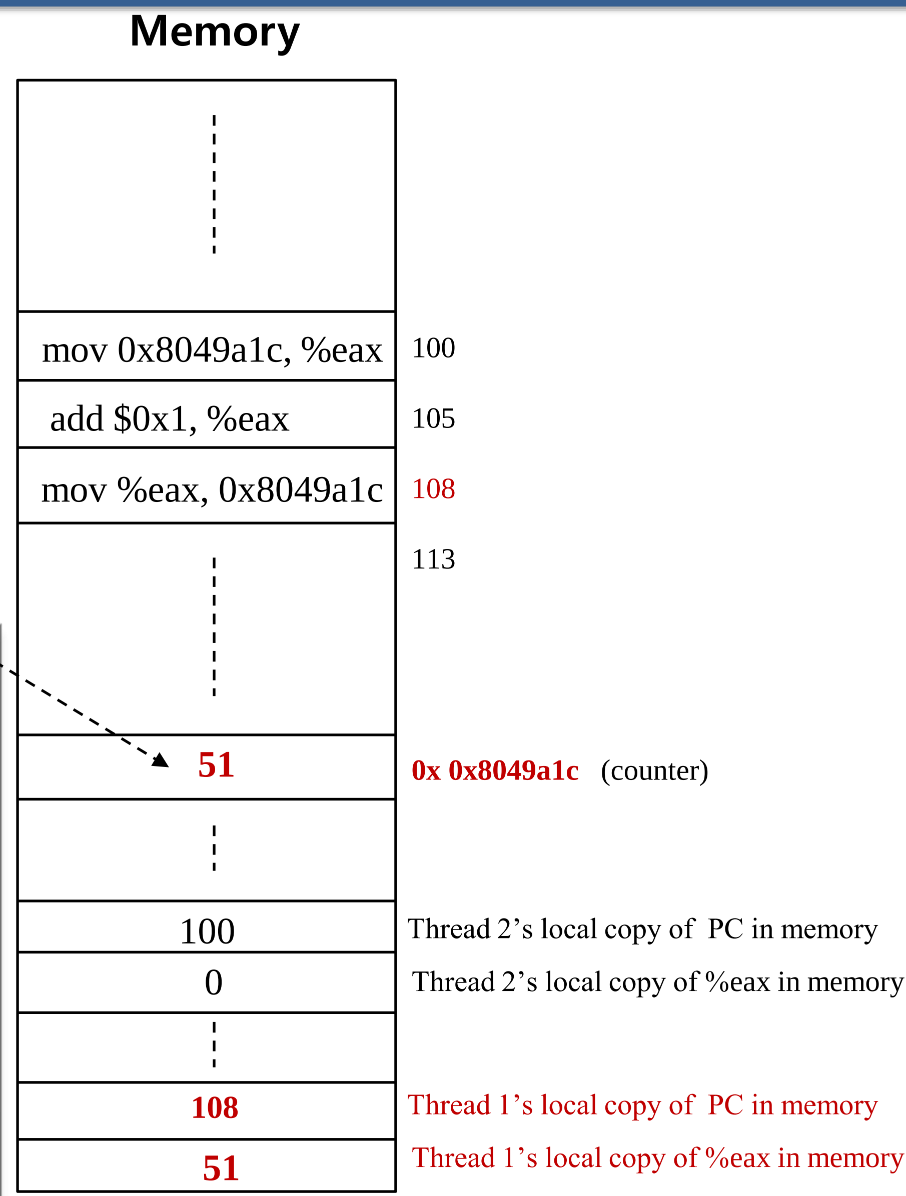




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Step 2 & 3: Decode & Execution

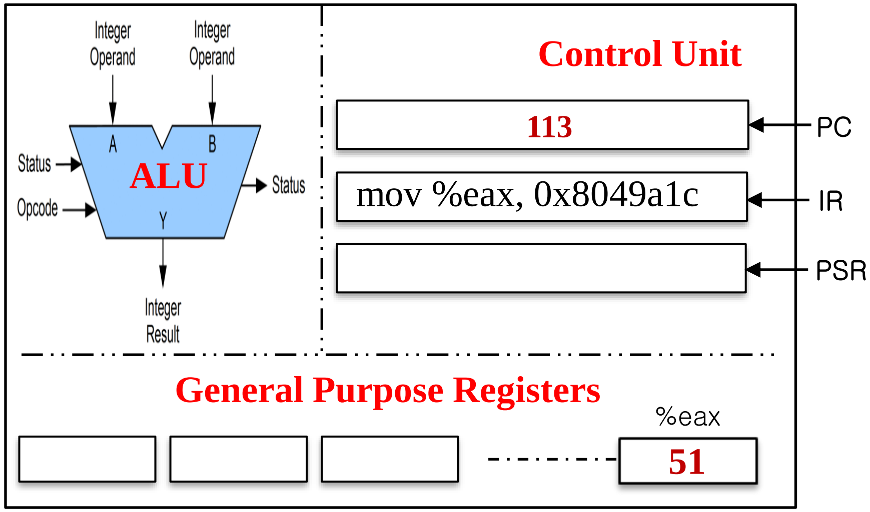




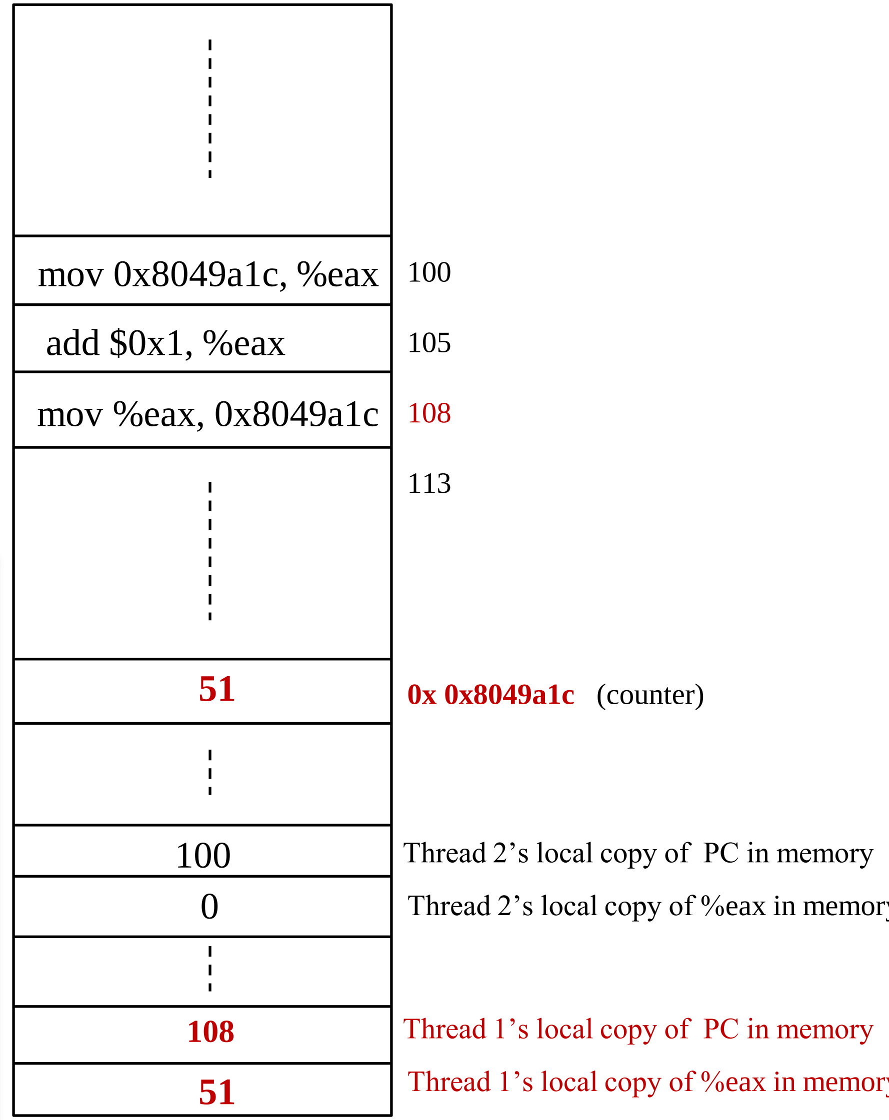
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Step 4: Increase PC (pointed to the next instruction in the memory)

CPU



Memory



OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Persistence

- Devices such as DRAM store values in a volatile.
- *Hardware* and *software* are needed to store data persistently.
  - **Hardware**: I/O device such as a hard drive, solid-state drives(SSDs)
  - **Software**:
    - File system manages storage devices (hard drives, SSDs, etc.).
    - File system is responsible for storing any files that users create.
      - Regular files
      - Directory files
      - Special files (e.g. devices – keyboards, monitors, etc.)
      - etc.

Persistence (Cont.)

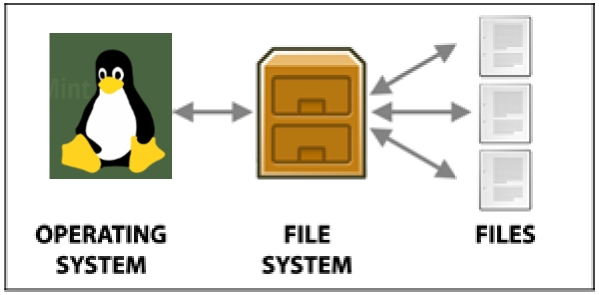
- Create a file (`myfile`) that contains the string "hello world" (`fs.c`)

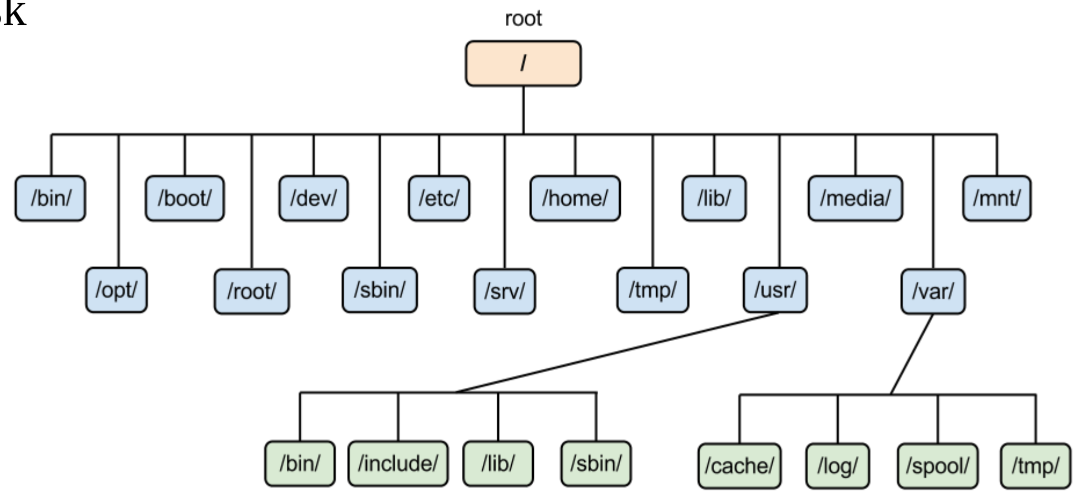
```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
        int fd = open("myfile", O_WRONLY | O_CREAT
                      | O_TRUNC, S_IRWXU);
        assert(fd > -1);
        int rc = write(fd, "hello world\n", 13);
        assert(rc == 13);
        close(fd);
        return 0;
}
```

`open()`, `write()`, and `close()` system calls are routed to the part of OS called the file system, which handles the requests

Persistence (Cont.)

- What OS does in order to write data to the disk?
  - Store file meta data in the disk (file name, creation time, etc.)
  - Figure out **where** on the disk this new data will reside
  - **Issue I/O** requests to the underlying storage device
- File system handles system crashes during write.
  - **Journaling** or **copy-on-write**
  - Carefully ordering writes to disk





Design Goals

- Build up **abstraction**
  - Make the system convenient and easy to use.
- Provide **high performance**
  - Minimize the overhead of the OS.
  - OS must strive to provide virtualization <u>without excessive overhead</u>.
- **Protection** between applications
  - <u>Isolation</u>: Bad behavior of one does not harm other and the OS itself.
- High degree of **reliability**
  - The OS must also run non-stop.
- Other issues: Energy-efficiency, Security, Mobility

Summary

- Computer architecture (Von Neumann): CPU, Memory, and IO devices
- OS main functions
  - **Manage resources via virtualization**
    - CPU, Memory, IO devices (e.g. storage (e.g. hard drives), keyboard, etc.)
  - **Provide services via system calls**
    - Process (run programs), memory management, file, etc.
- Design goals
  - Abstraction, high performance, protection, reliability etc.
- Related Chapter: Chapter 2 (2.1 – 2.5)
- Next: User-level programming using system calls (process, memory, file)
  - Chapter 5 (Process) , Chapter 14 (Memory) , Chapter 39 (Files)
  - Self study: Lab tutorial