Lecture 13: Concurrency – Thread and Thread API

The Course Organization (Bottom-up)

Concurrency	Race Conditions, Lock/Semaphore Thread	
Virtualization	Memory Management Process and CPU Scheduling	HW#4 HW#3
Persistence	IO Devices and Storage File System	HW#2 & Project

System Calls (User-level Programming) HW#1

Course Overview

Part I: Introduction to Thread

Thread

- A new abstraction for a single running process

- Multi-threaded program
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They share the same address space.

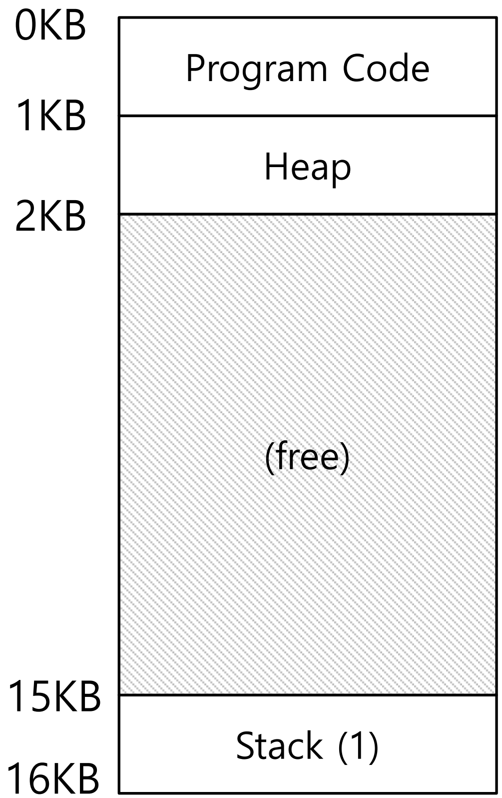
Context switch between threads

- Each thread has its own program counter and set of registers.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.

- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The address space remains the same.

The stack of the relevant thread

- There will be one stack per thread.

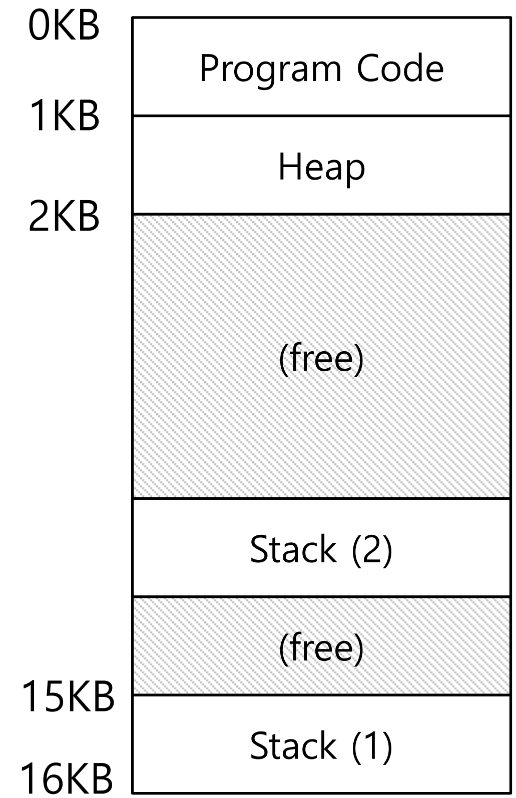


A Single-Threaded Address Space

The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

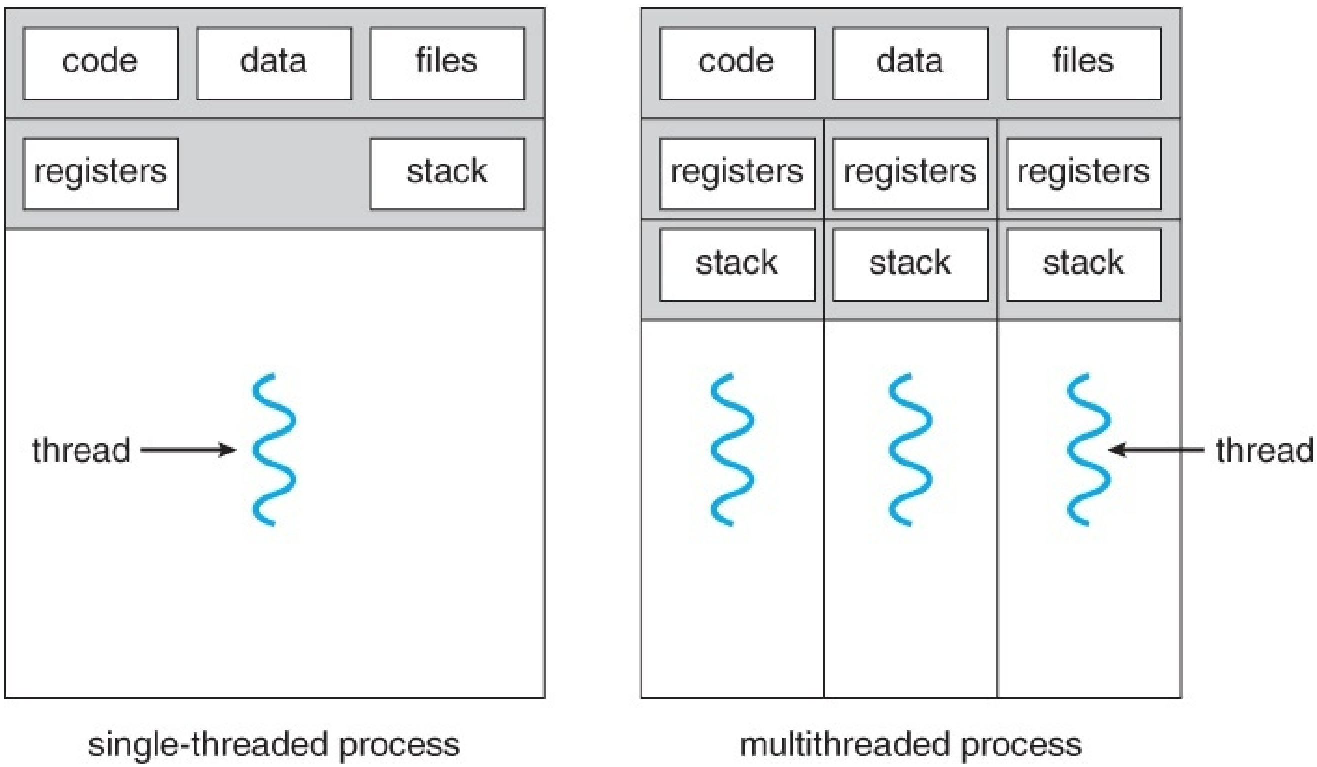
(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.



Two threaded Address Space

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;
}
```

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 start 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.

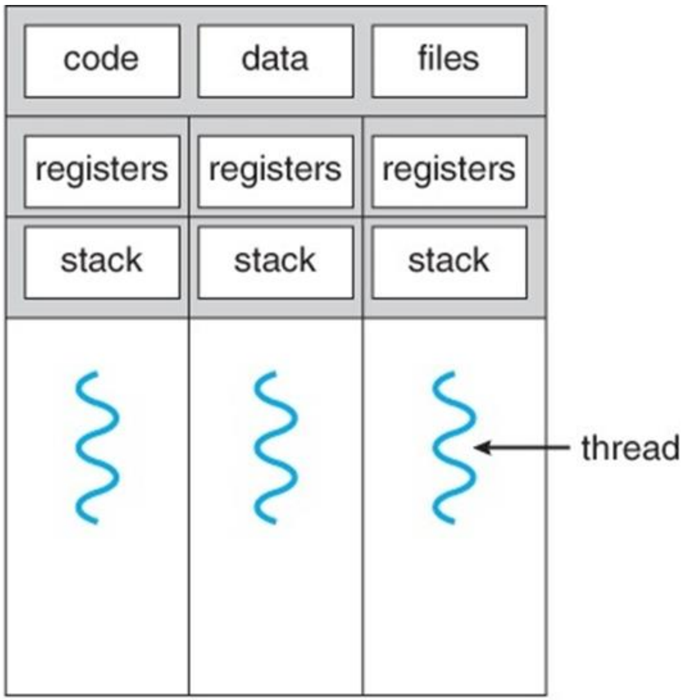
```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `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



multithreaded process

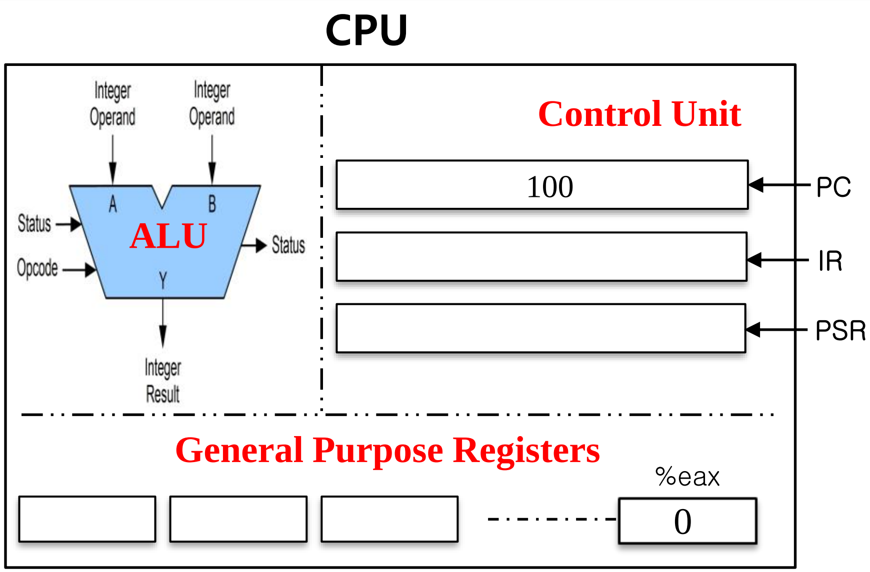
- These three instructions do not execute atomically. → Problem of **concurrency** happen.

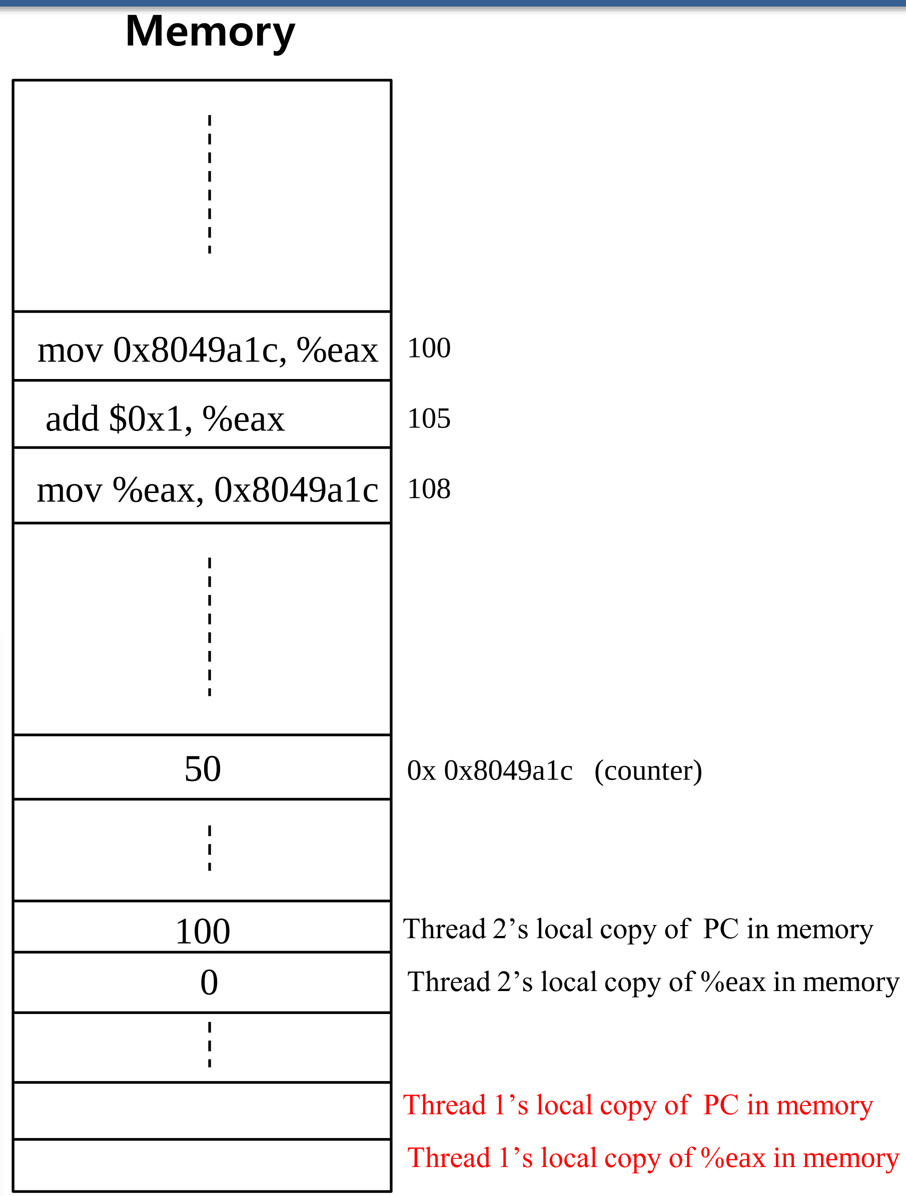
What happened?

- 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**

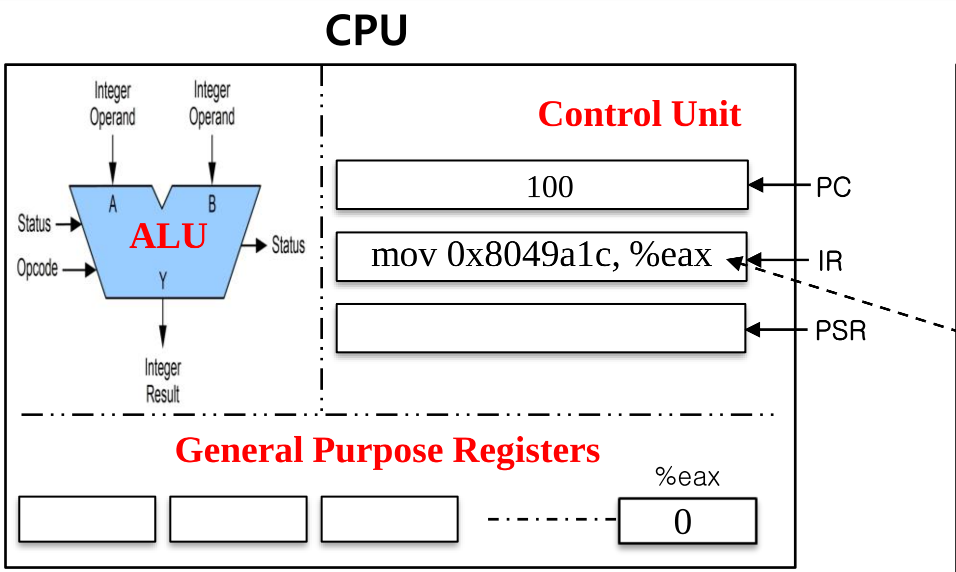
Step 0: Initial Value (PC=100)

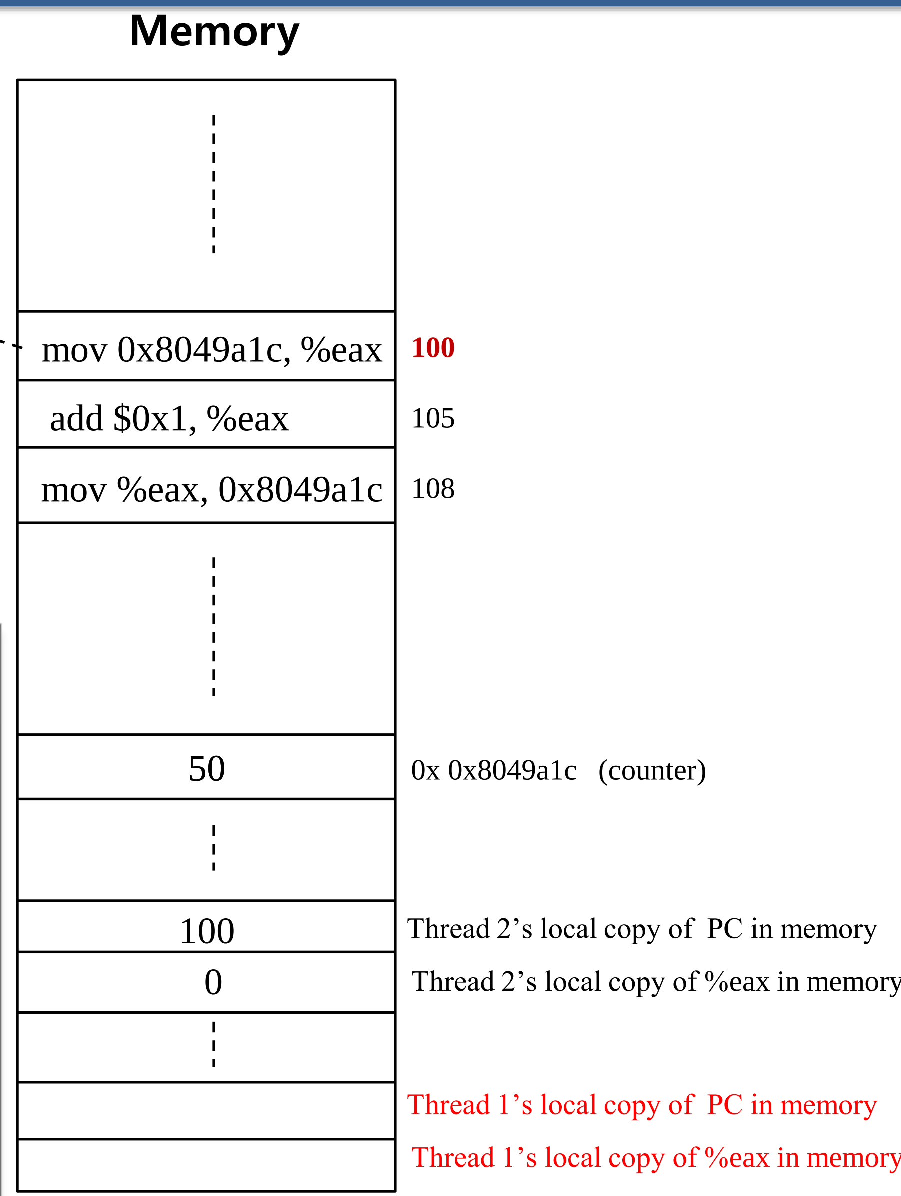




OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51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

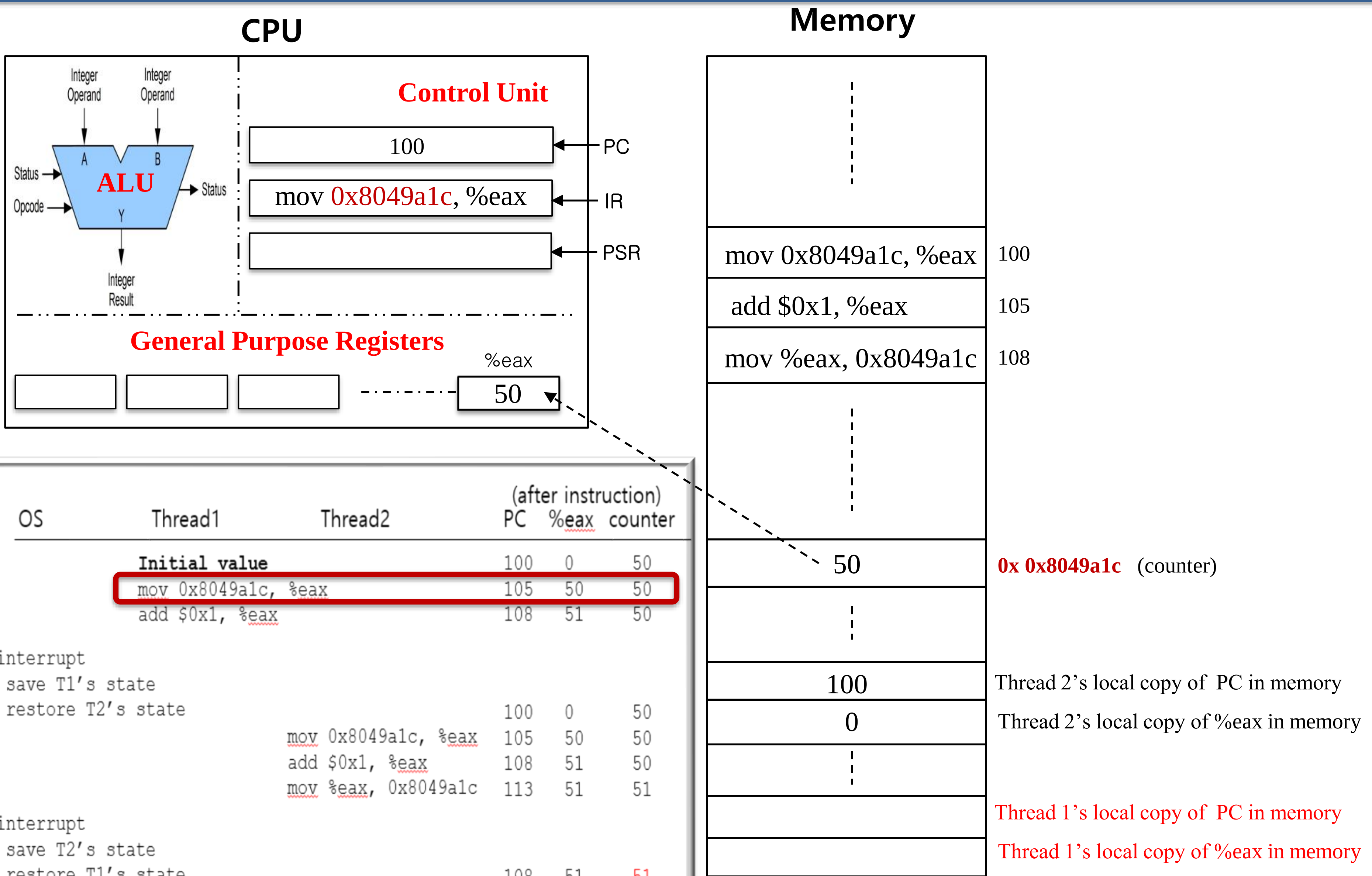
Step 1: Fetch instruction from the memory (PC->Address=100)





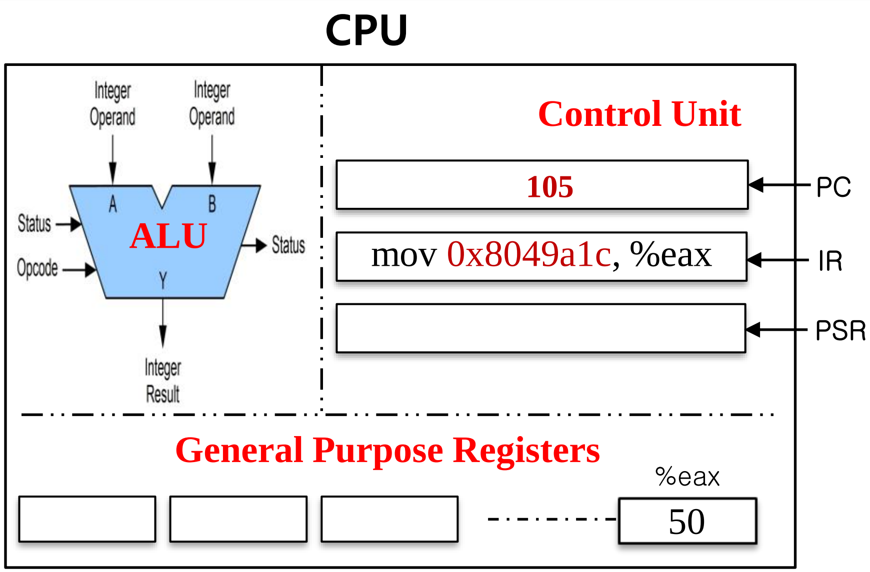
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

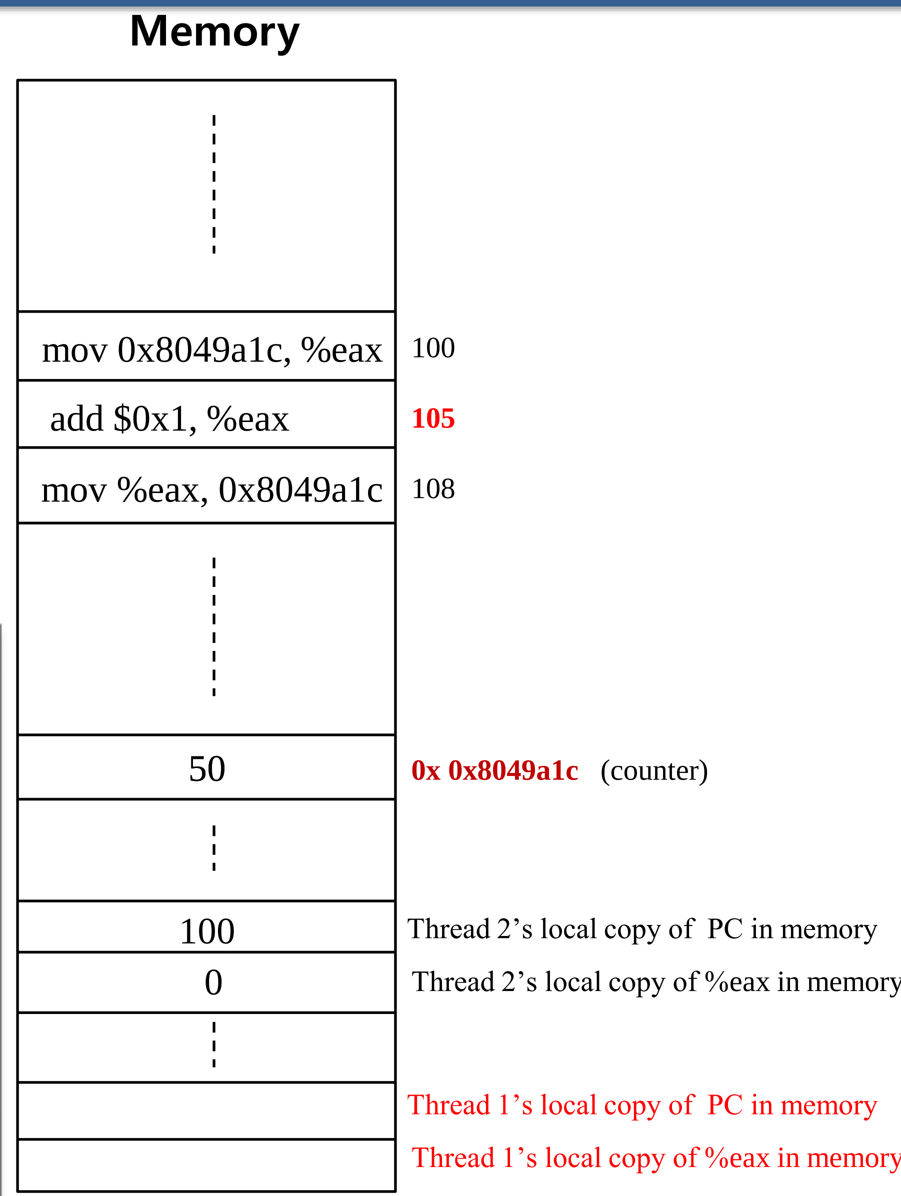
Steps 2 & 3: Decode & Execution



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)

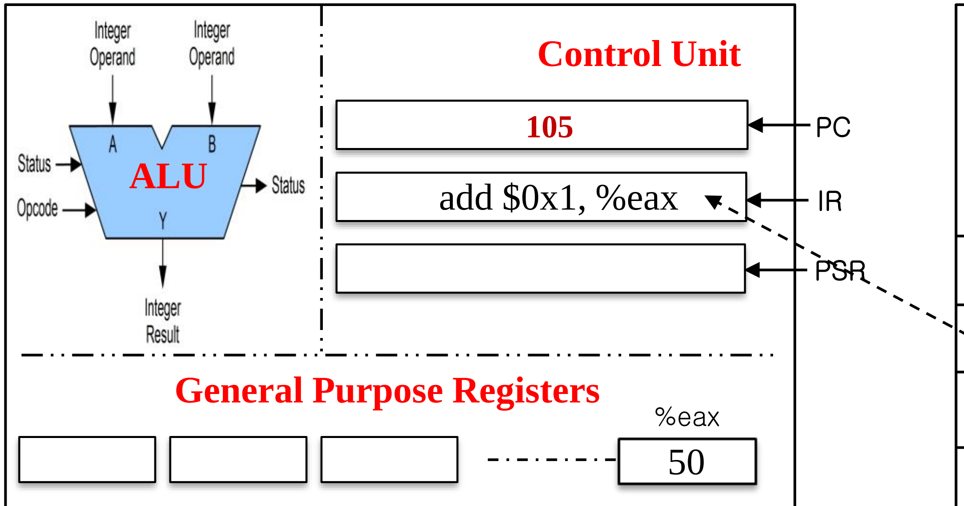




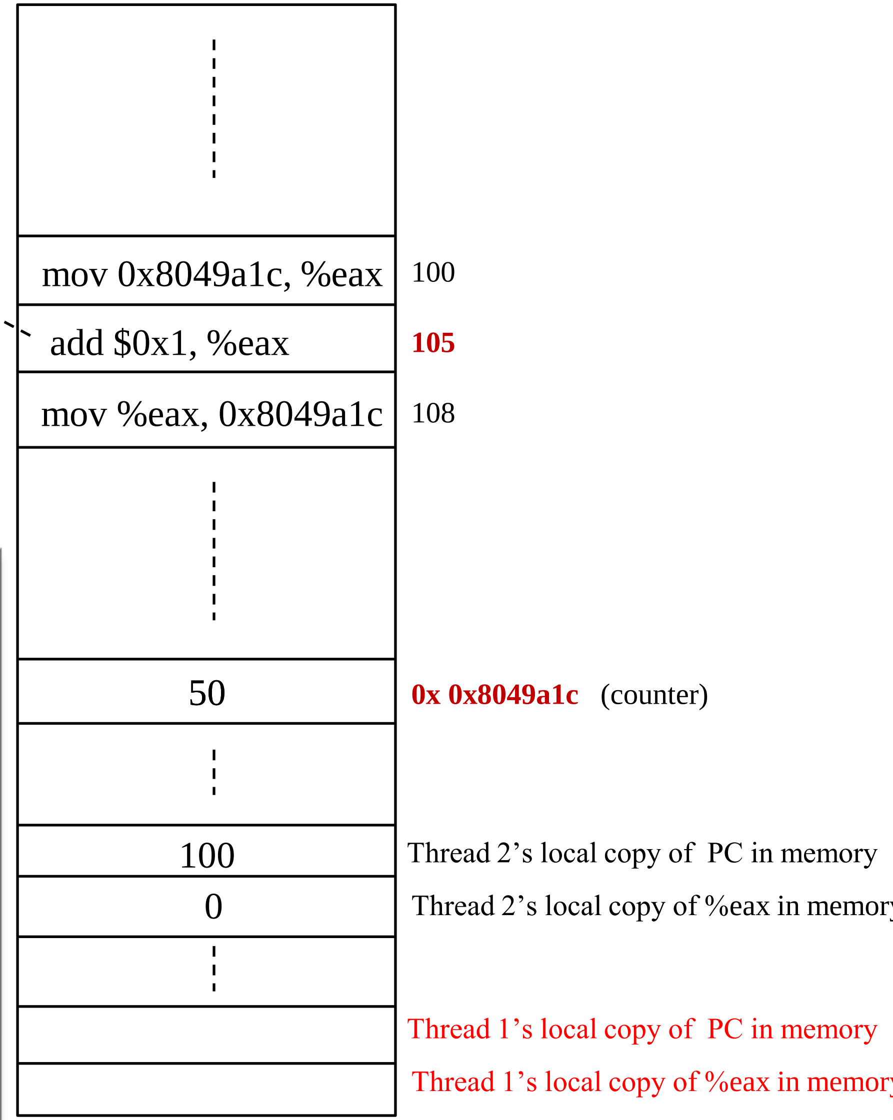
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

CPU

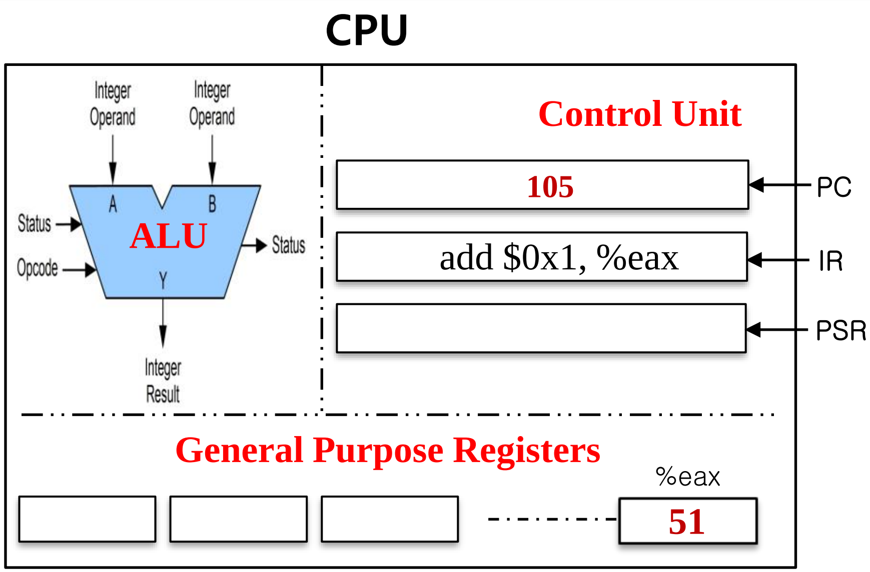


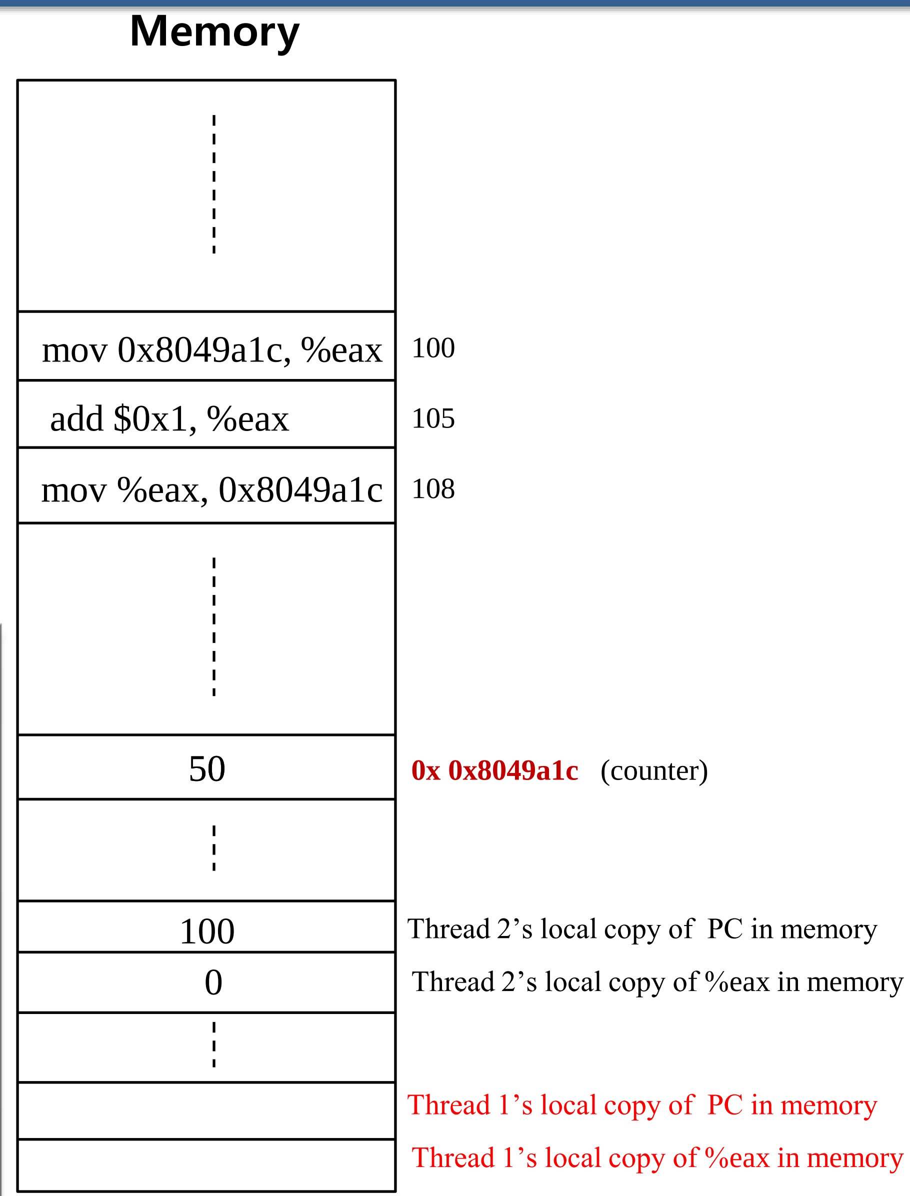
Memory



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**

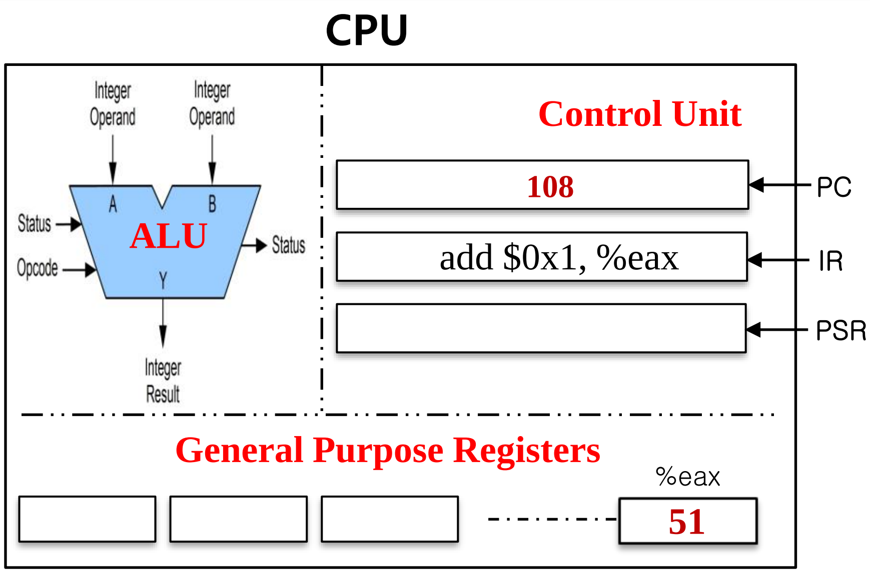
Steps 2 & 3: Decode & Execution

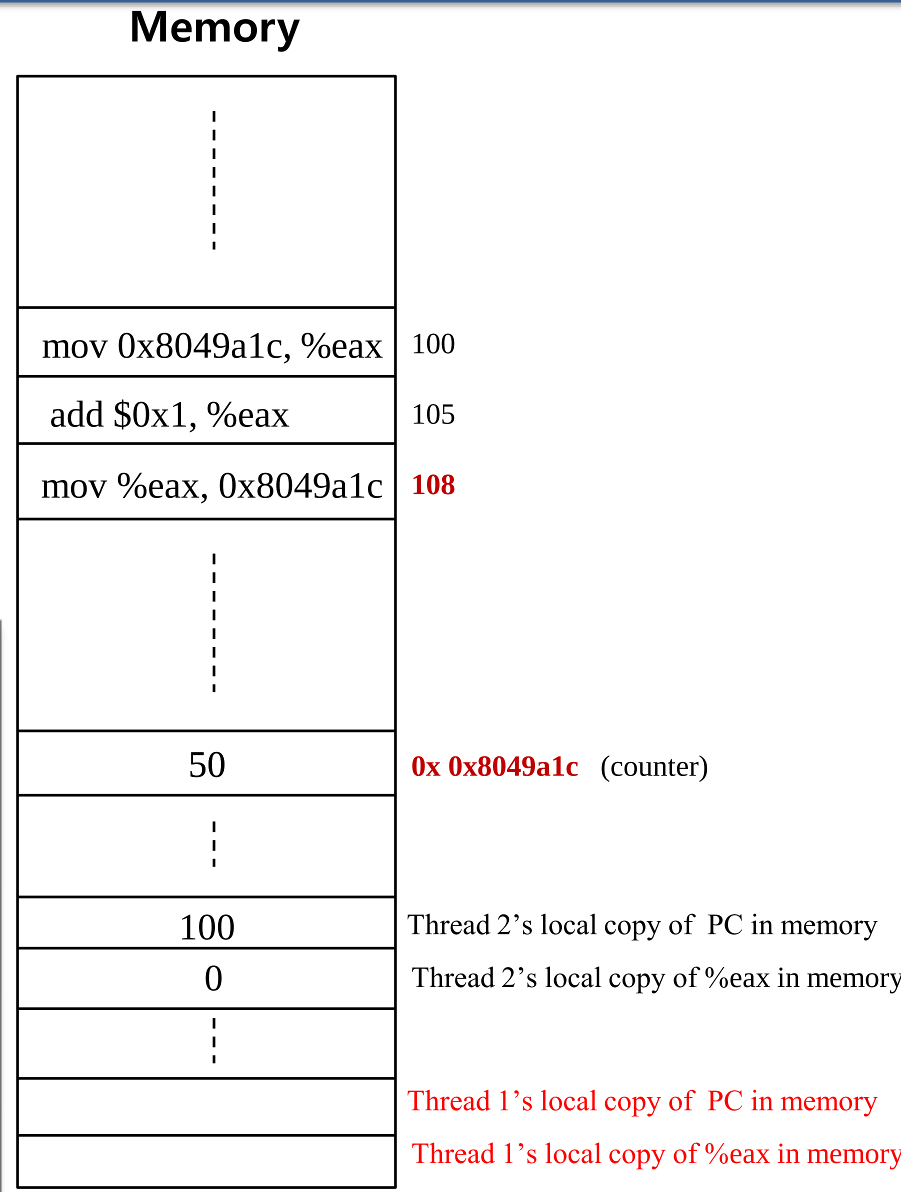




OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51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

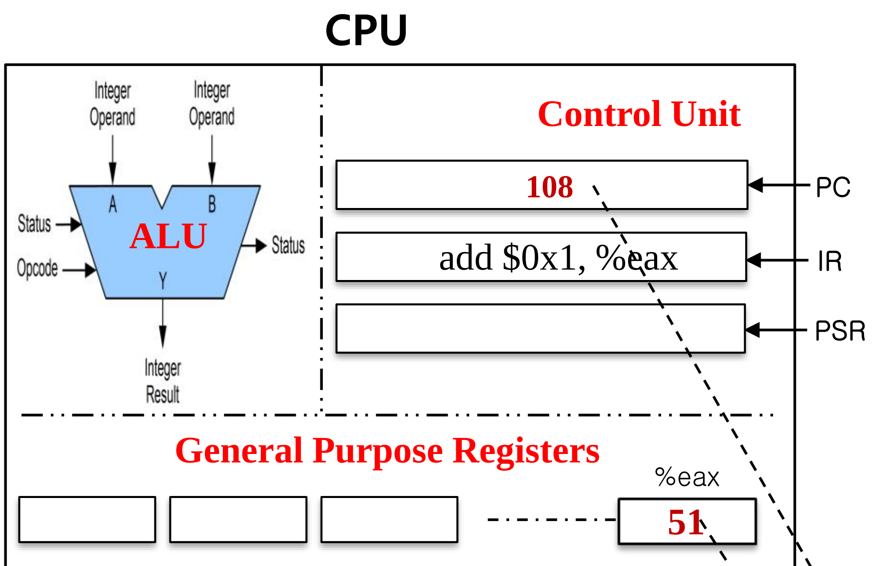
Step 3: Increase PC (pointed to the next instruction in the memory)

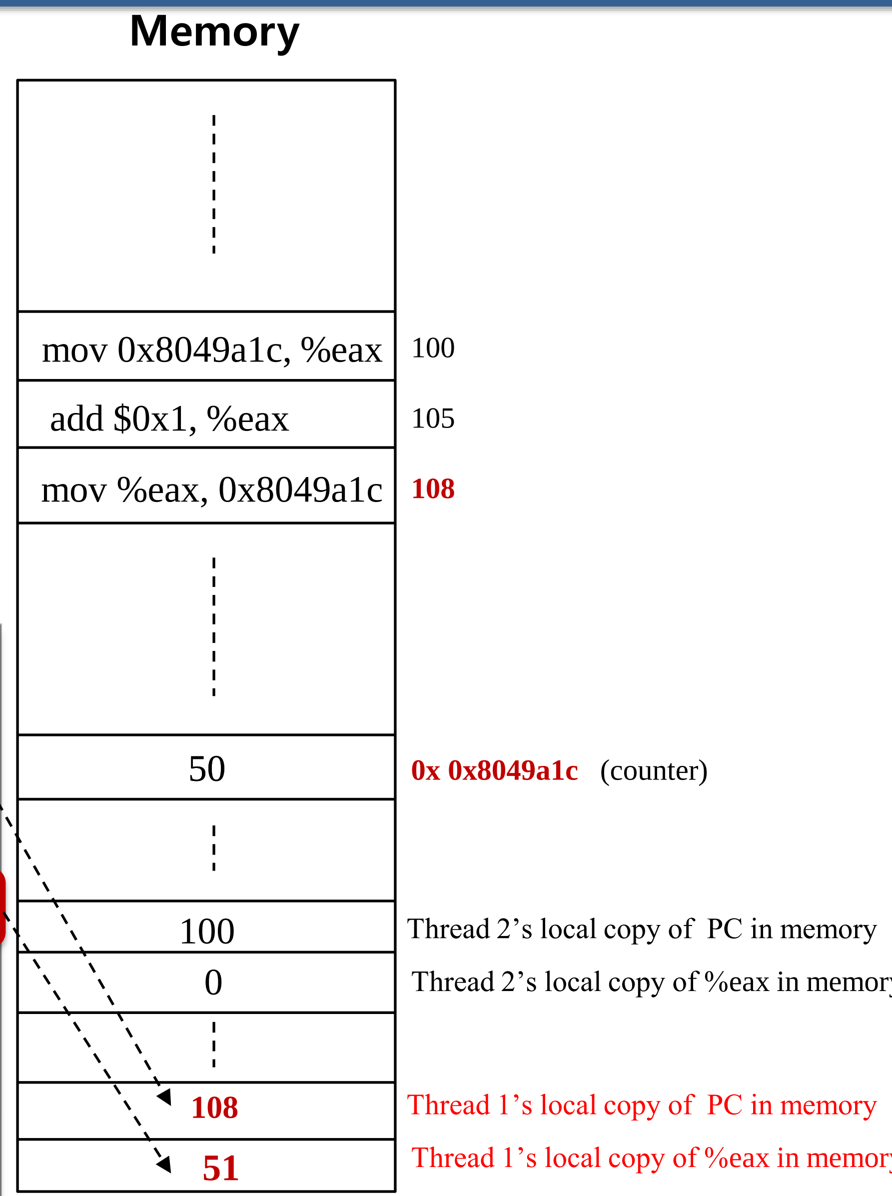




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	$\boxed{\text{add \$0x1, \%eax}}$		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c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

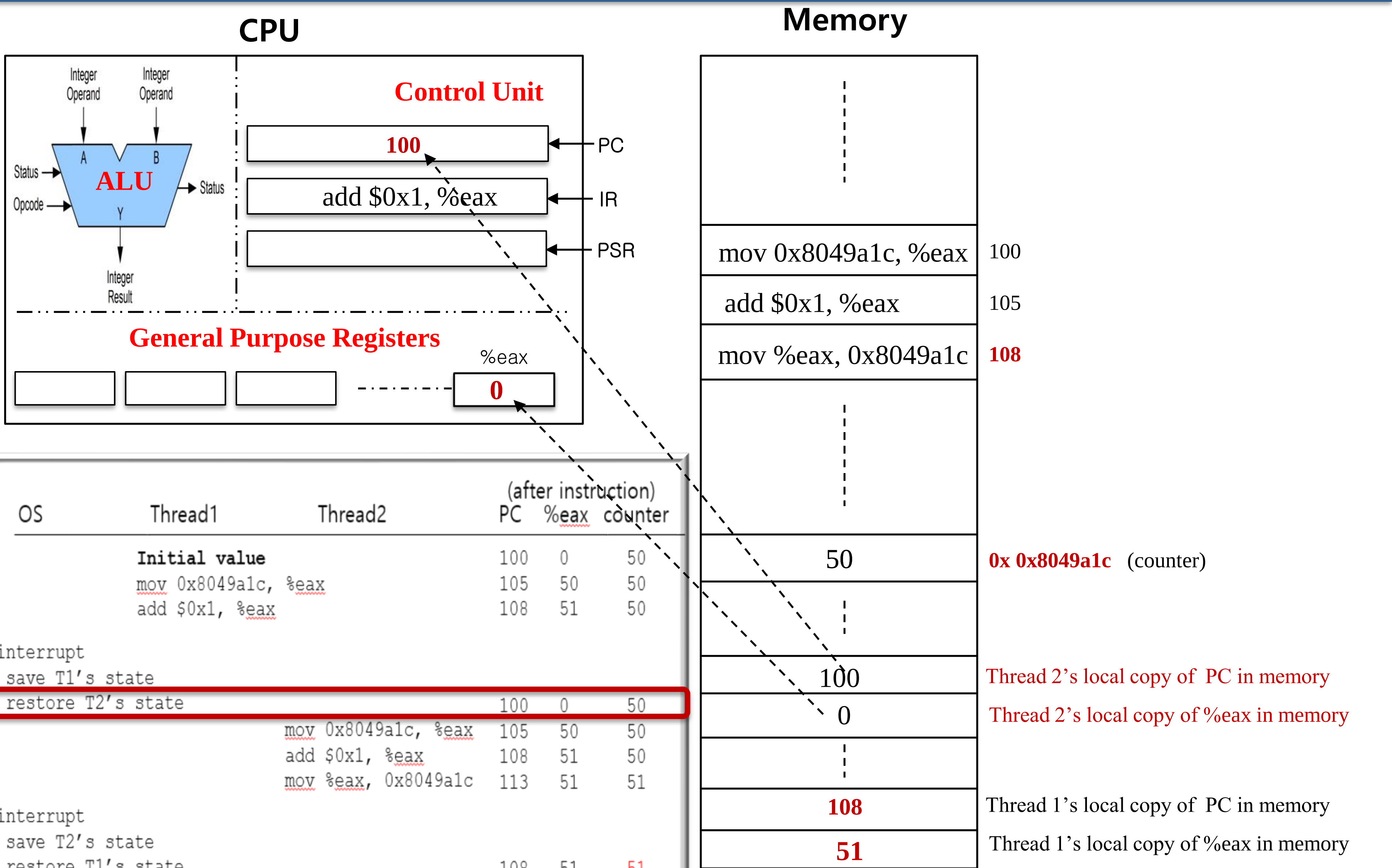
Interrupt occurred; Save Thread 1's state





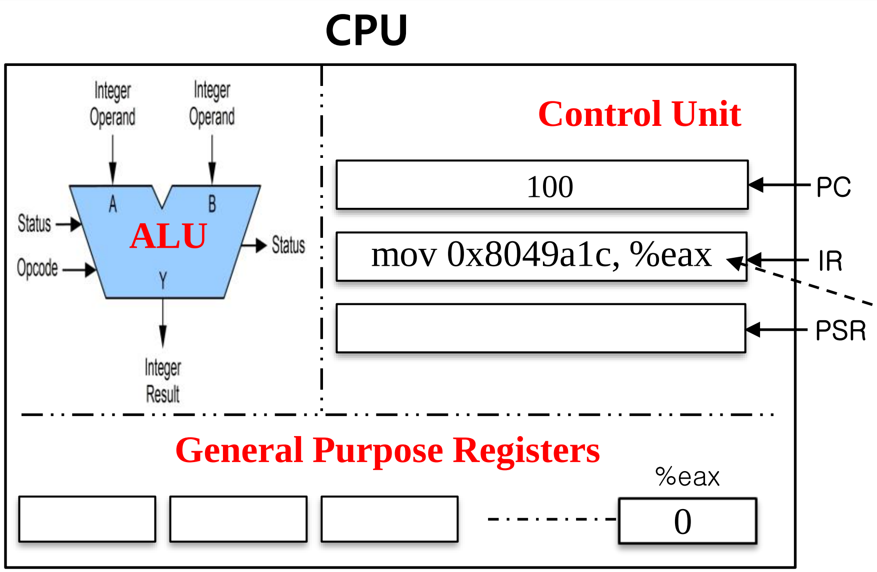
OS	Thread1	Thread2	PC (after instruction)	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

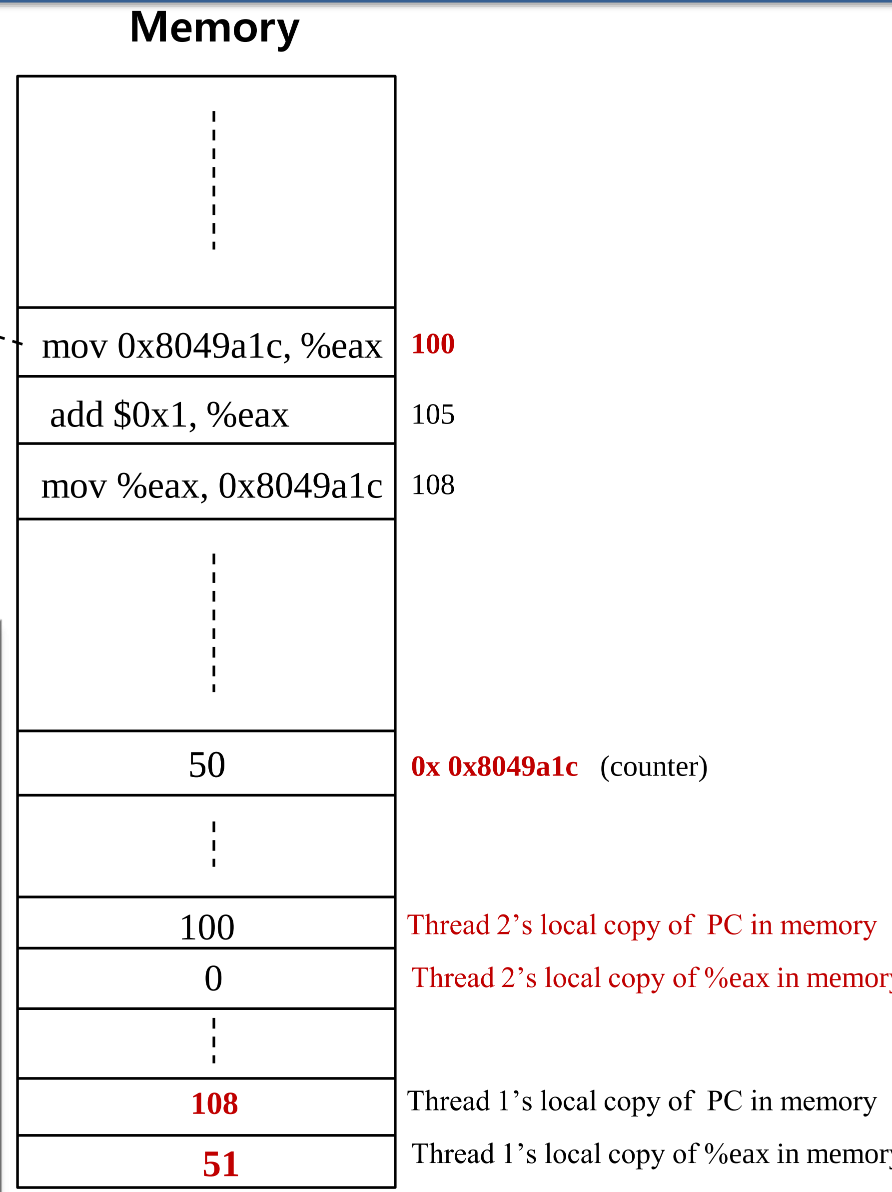
Restore Thread 2's state



OS	Thread1	Thread2	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**

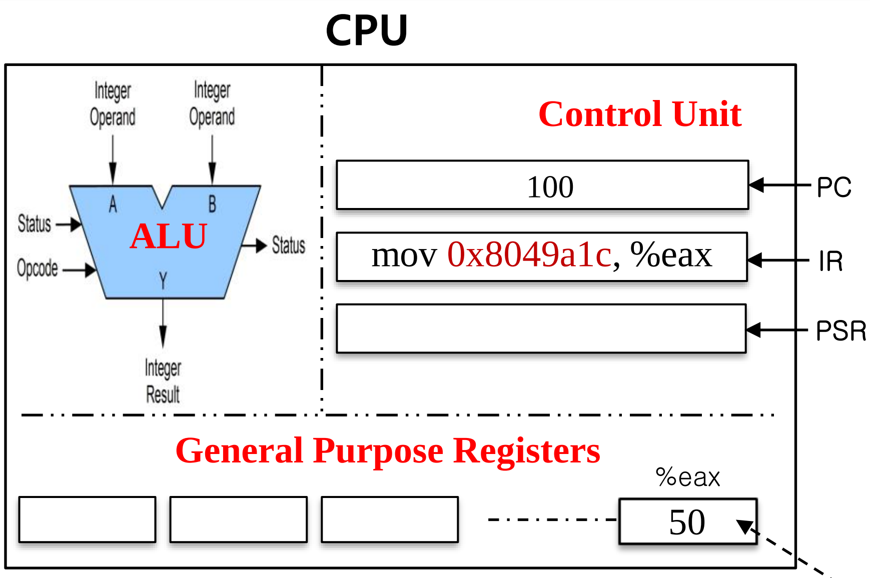
Step 1: Fetch instruction from the memory (PC->Address=100)

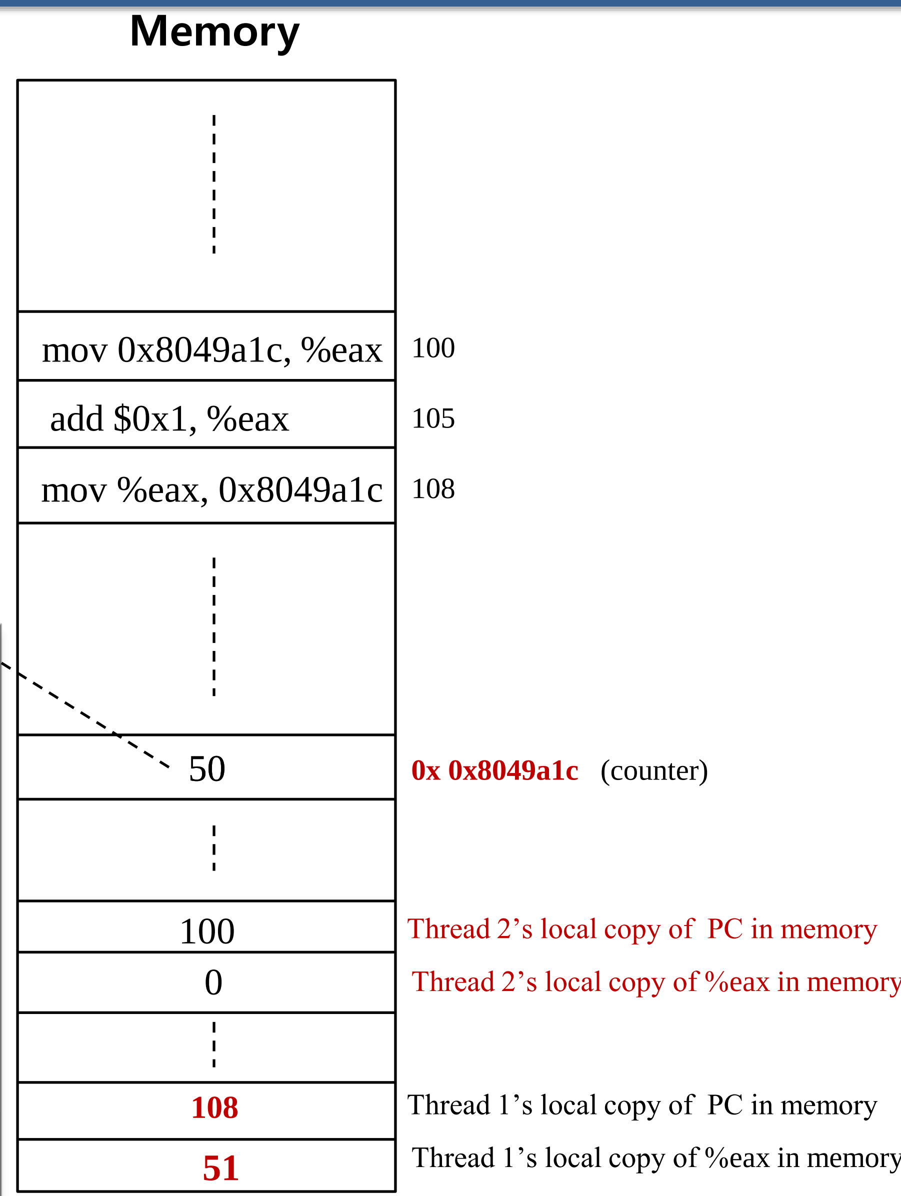




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

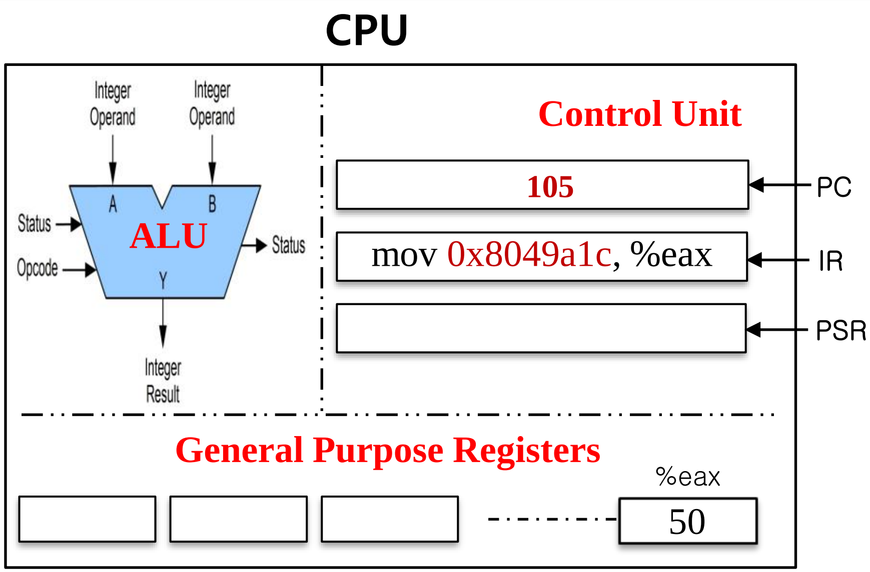
Steps 2 & 3: Decode & Execution

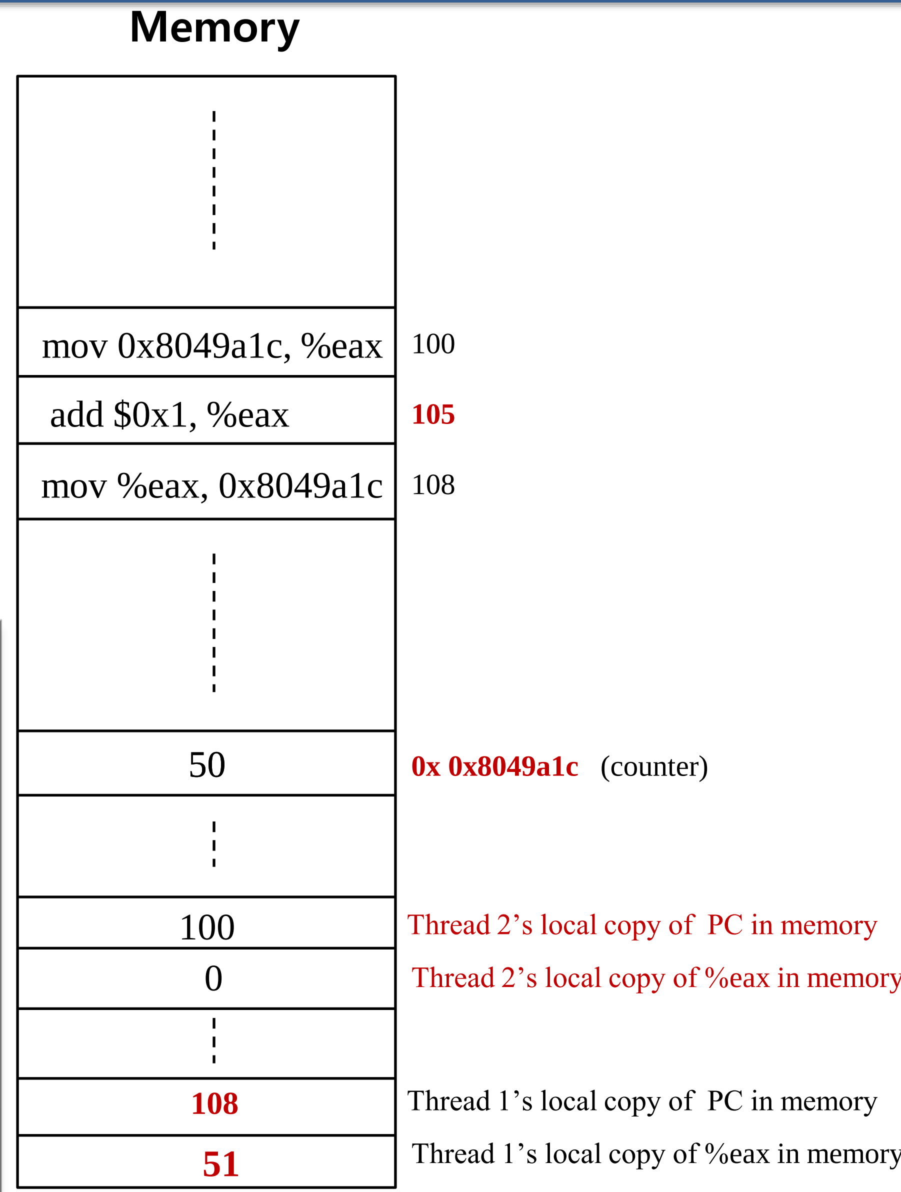




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

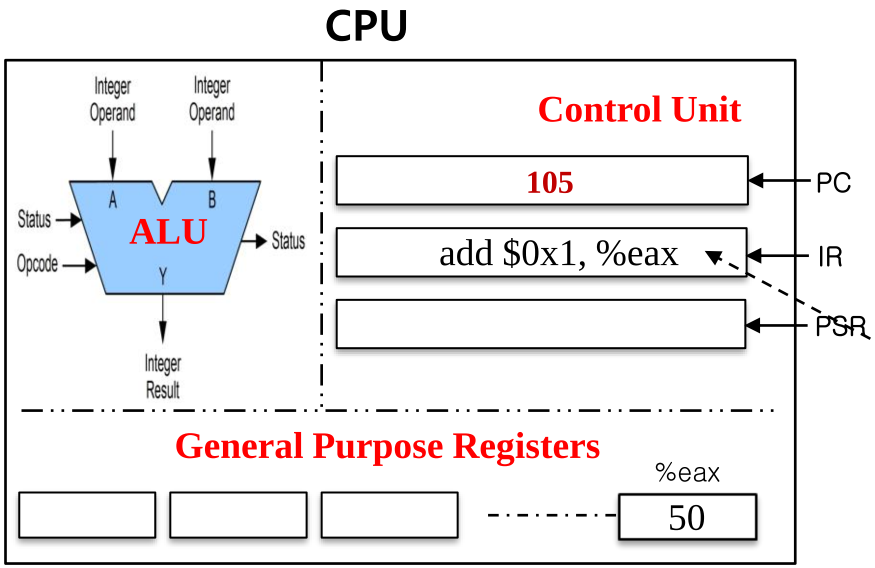
Step 4: Increase PC (pointed to the next instruction in the memory)

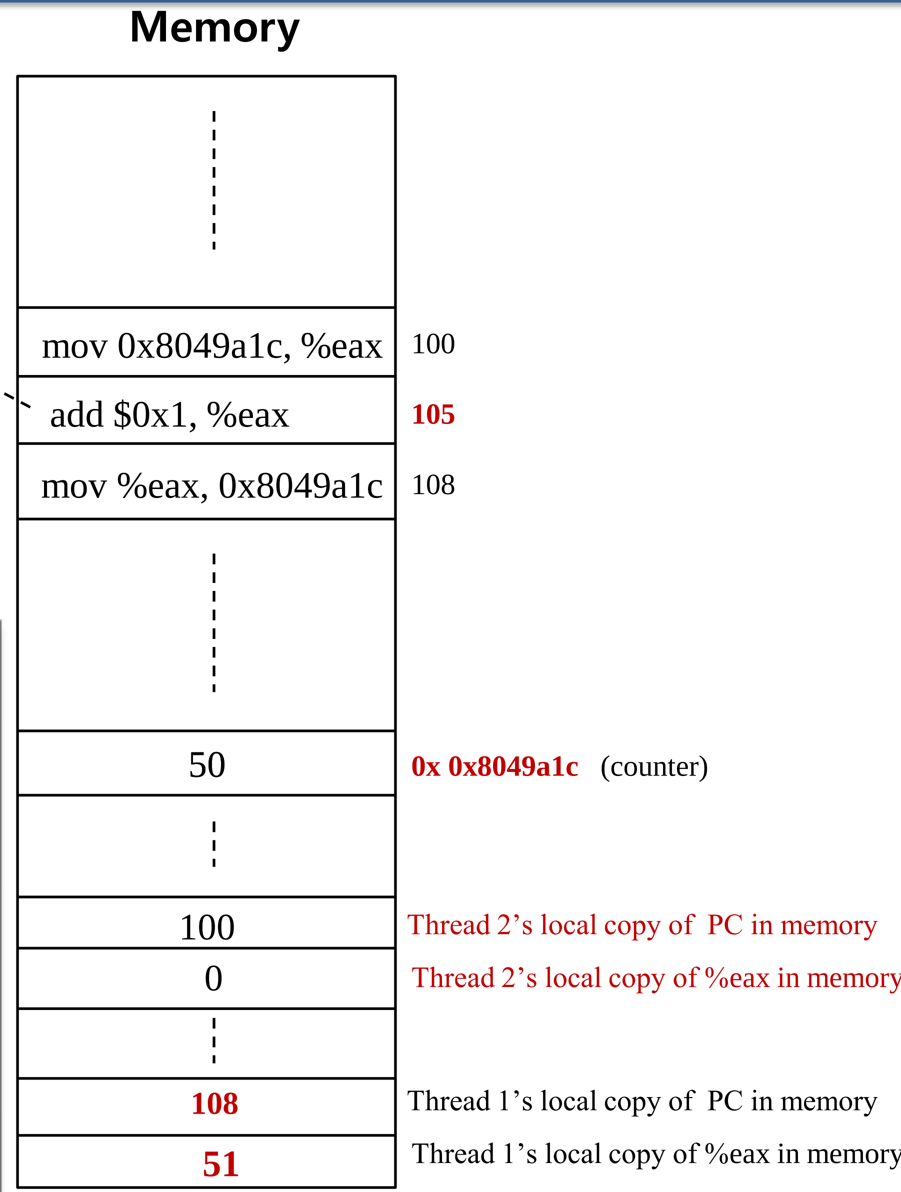




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

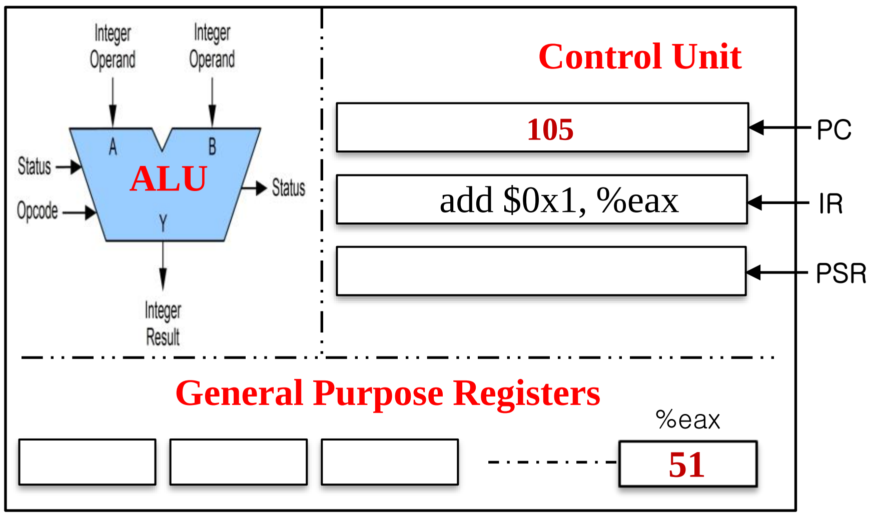




OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51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

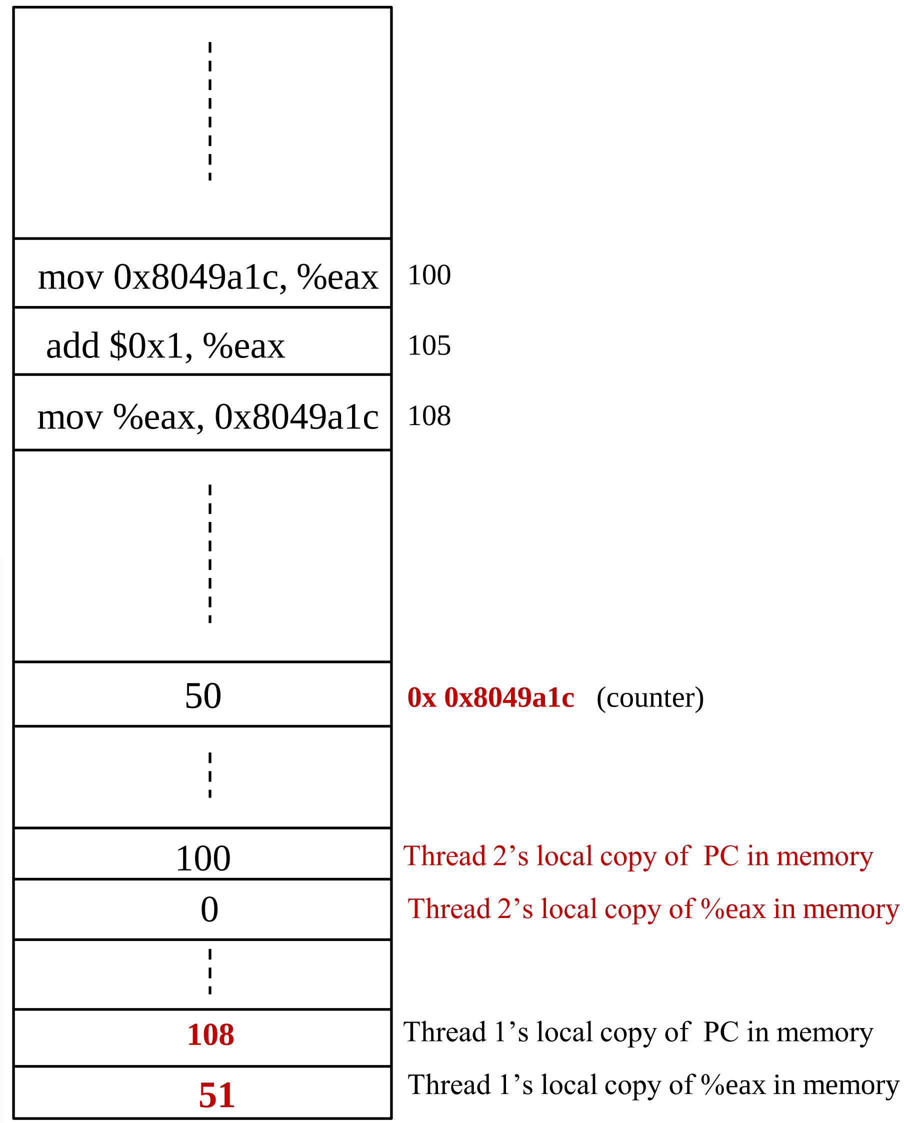
Steps 2 & 3: Decode & Execution

CPU

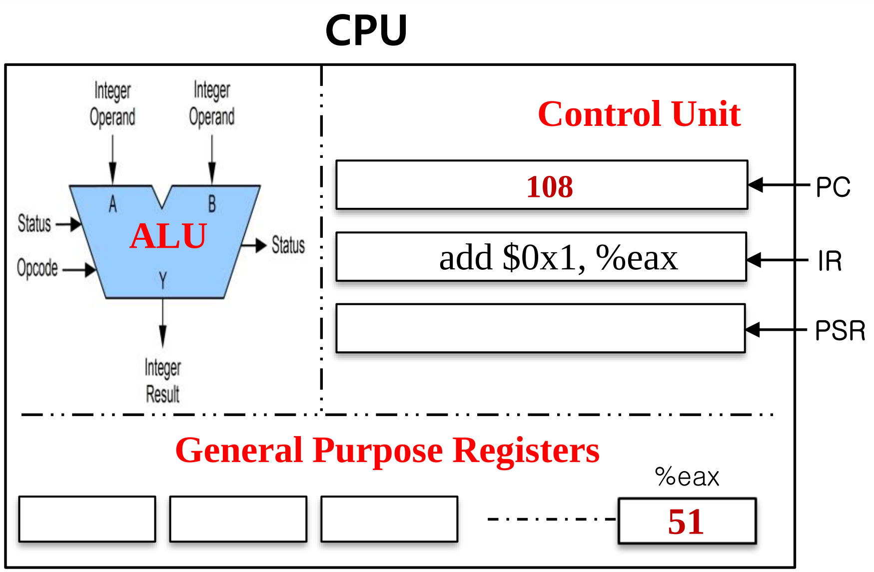


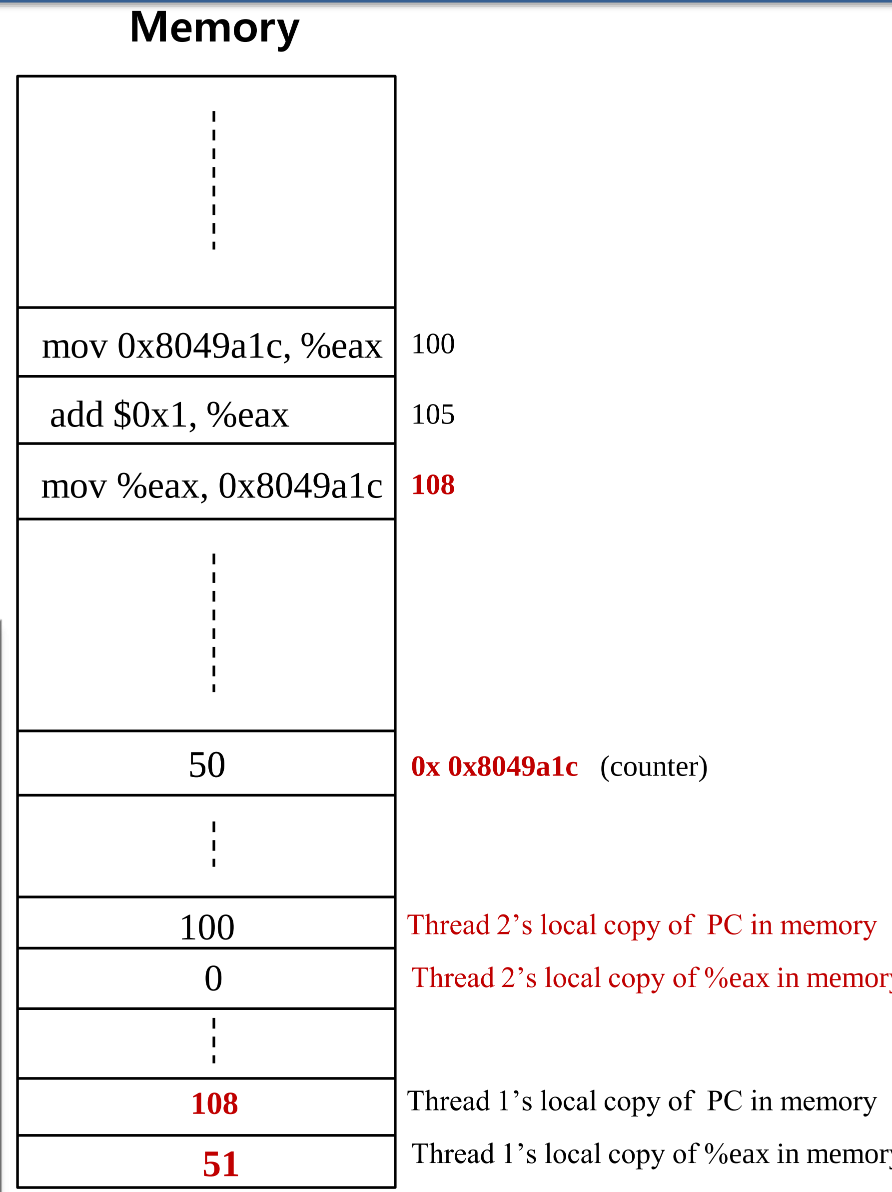
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

Memory



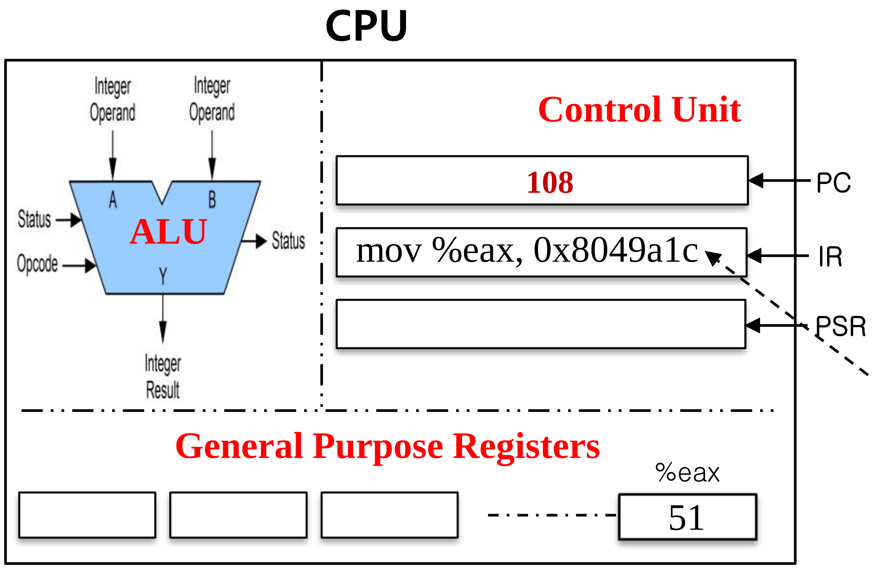
Step 4: Increase PC (pointed to the next instruction in the memory)

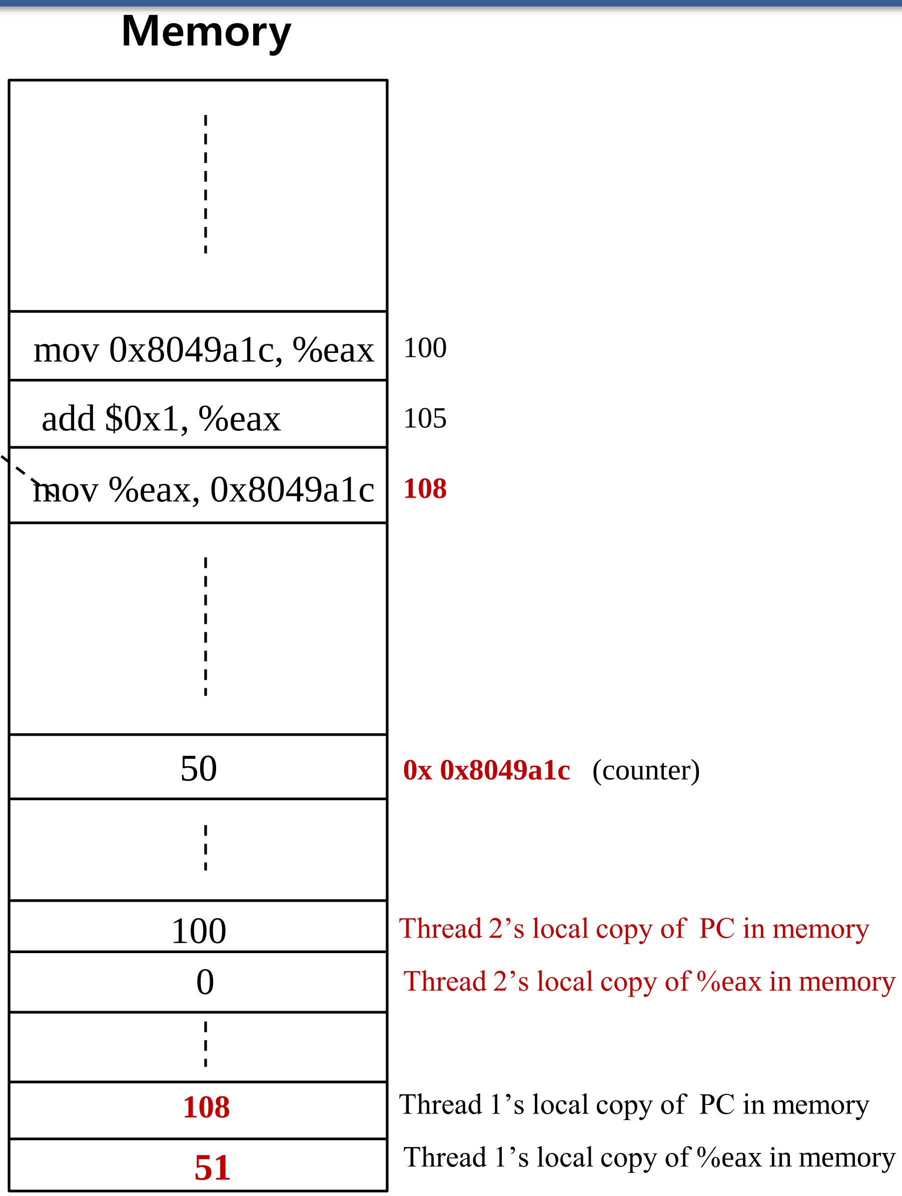




OS	Thread1	Thread2	PC (after instruction)	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

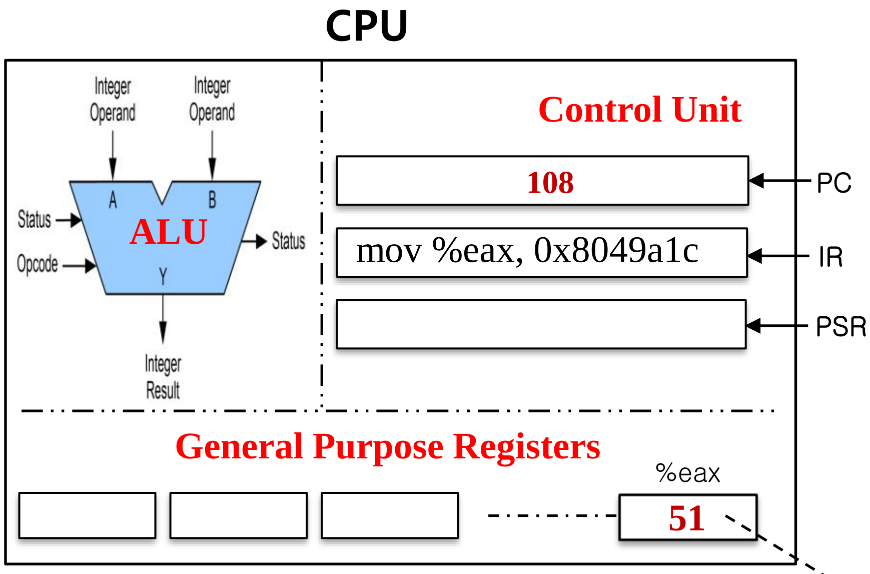
Step 1: Fetch instruction from the memory (PC->Address=108)



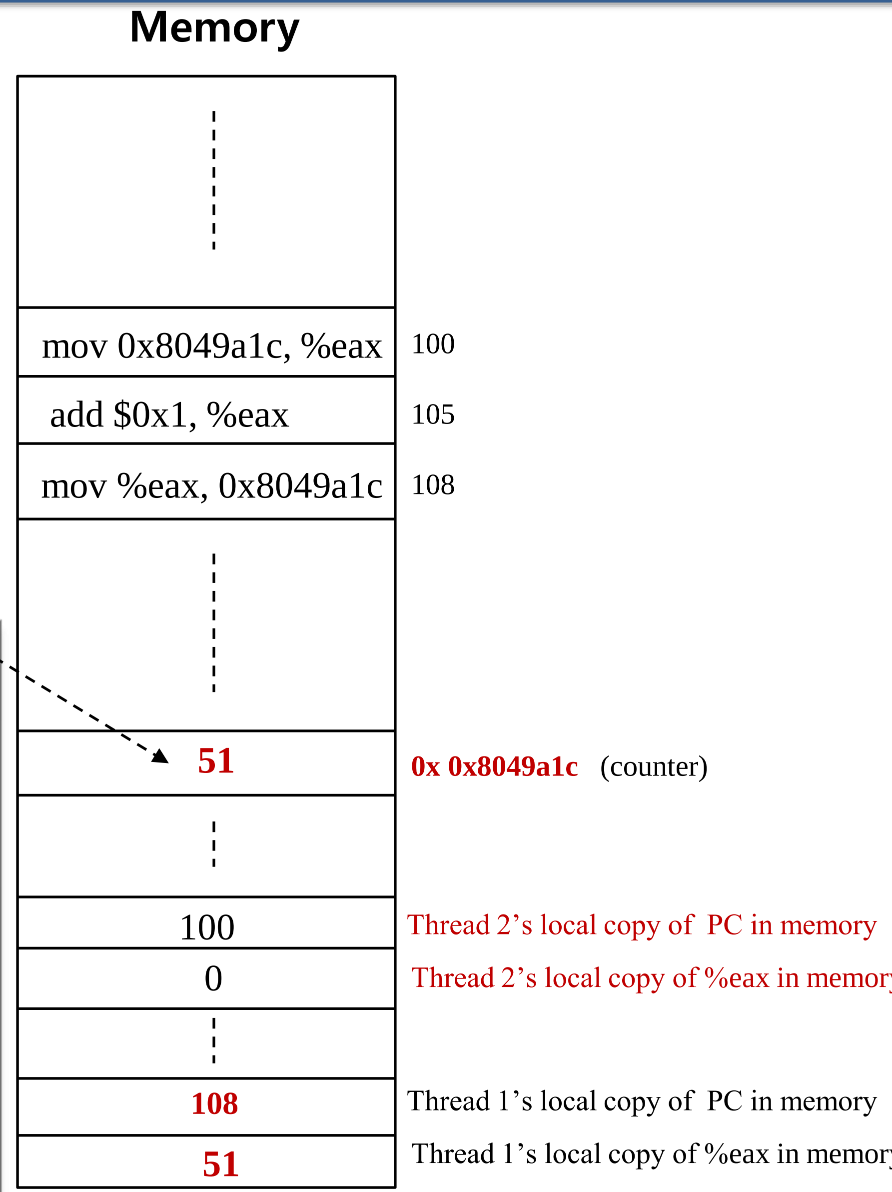


OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51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

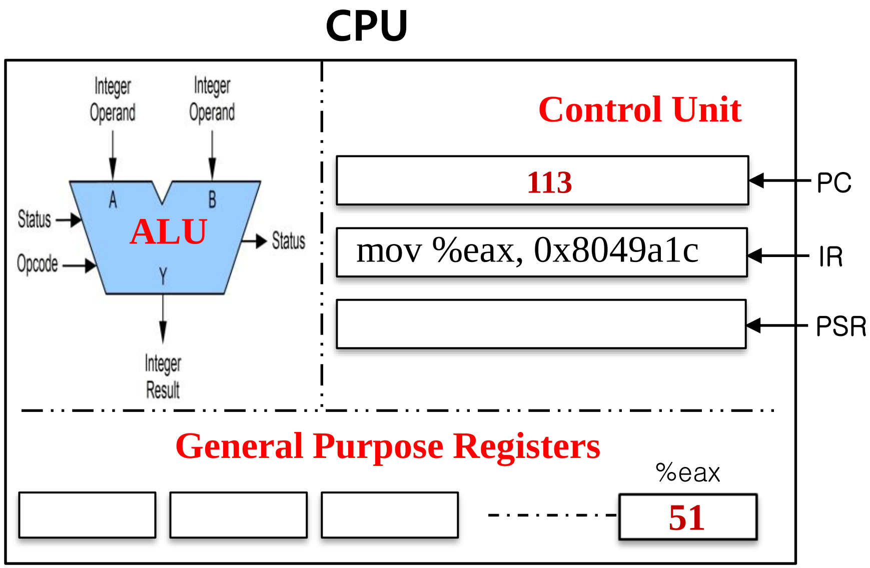
Steps 2 & 3: Decode & Execution

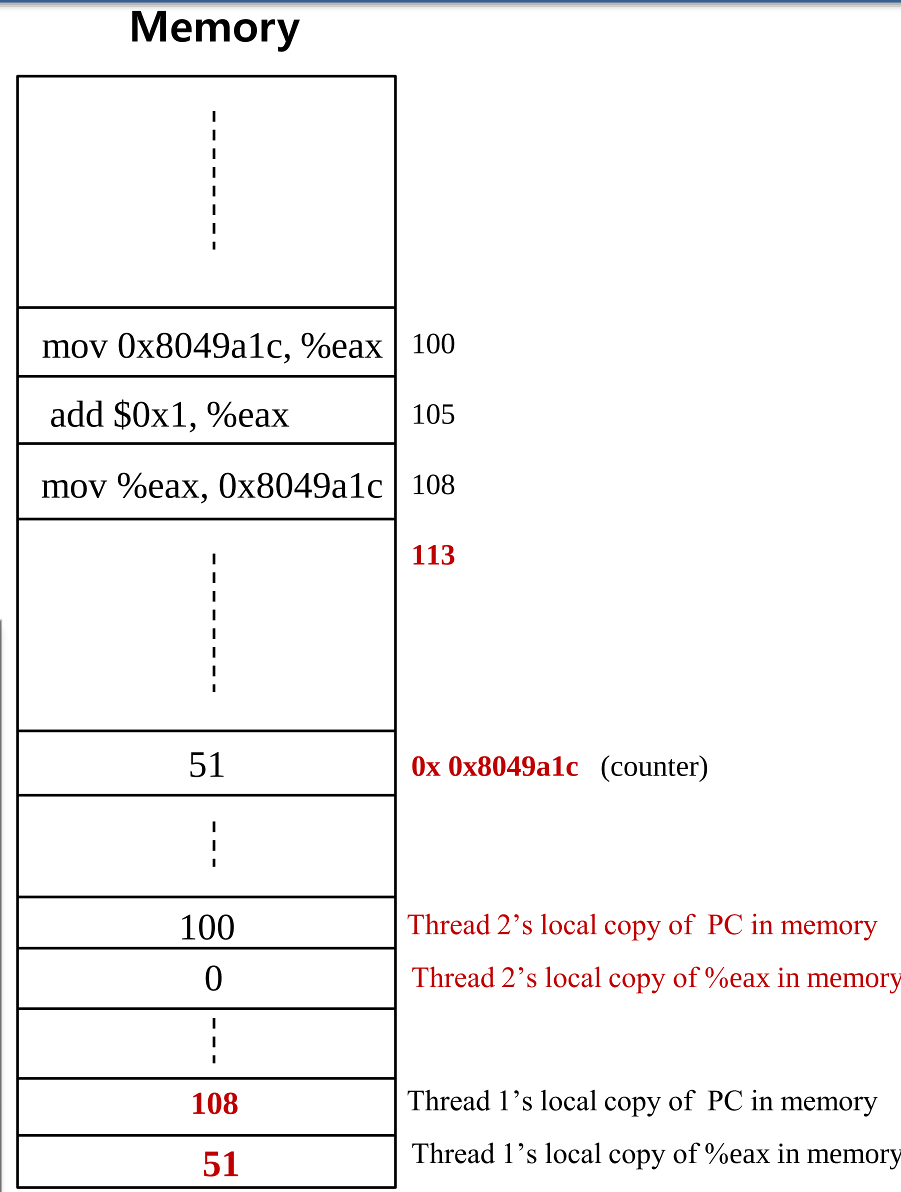


OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51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



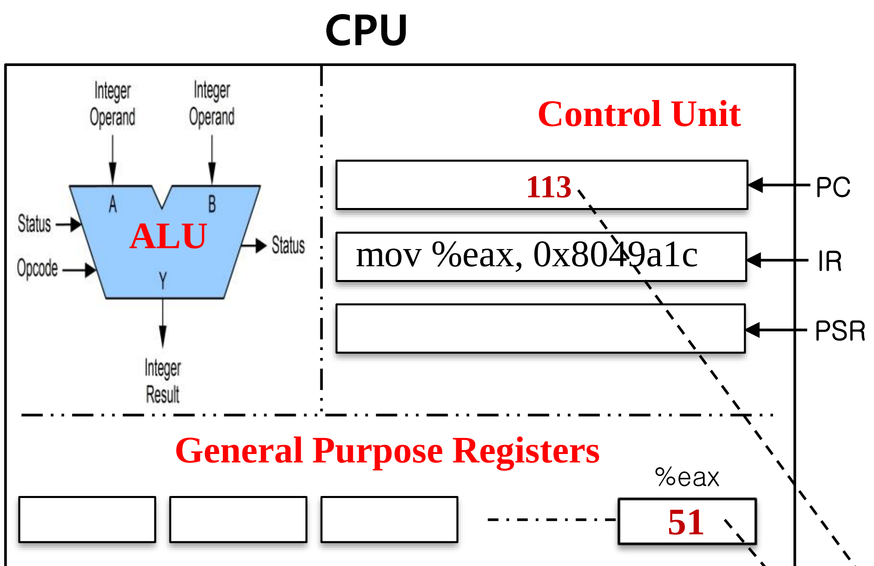
Step 4: Increase PC (pointed to the next instruction in the memory)





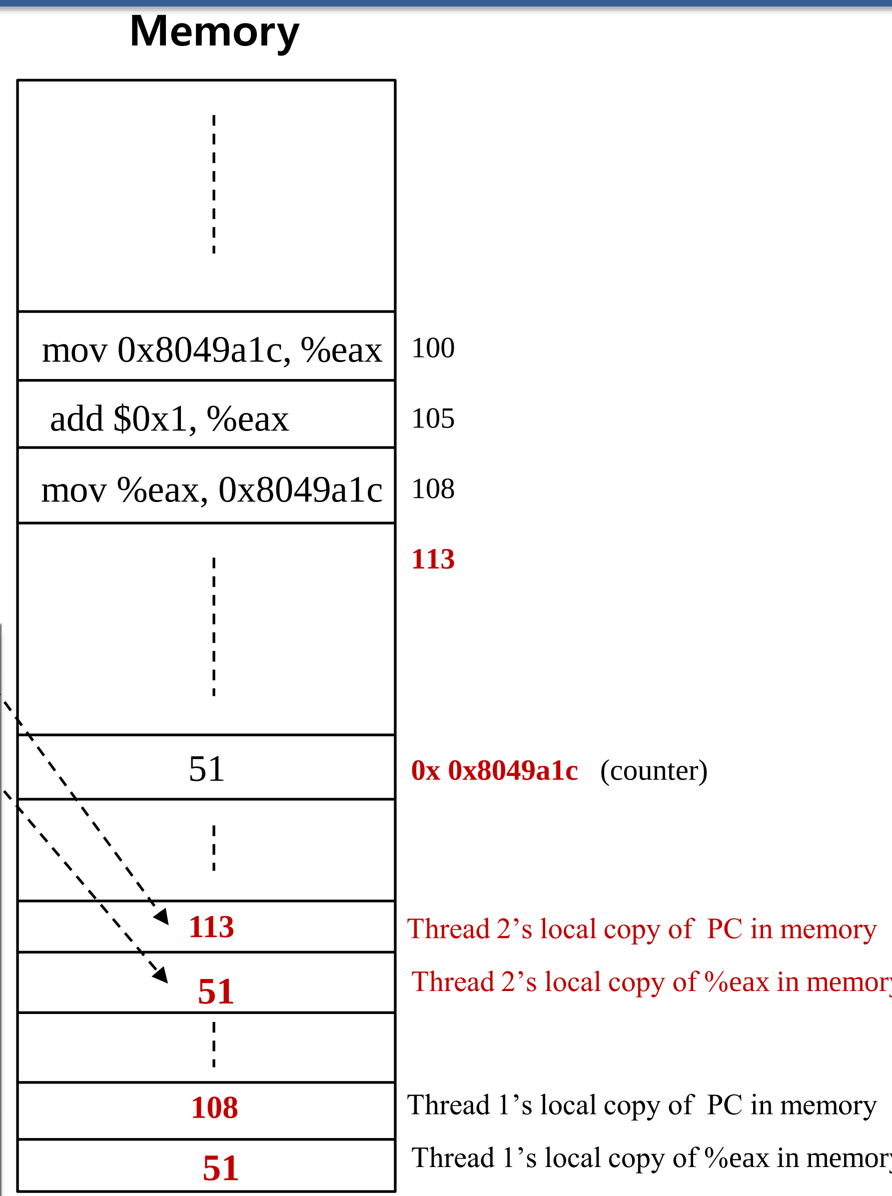
OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

Interrupt occurred; Save Thread 2’s state

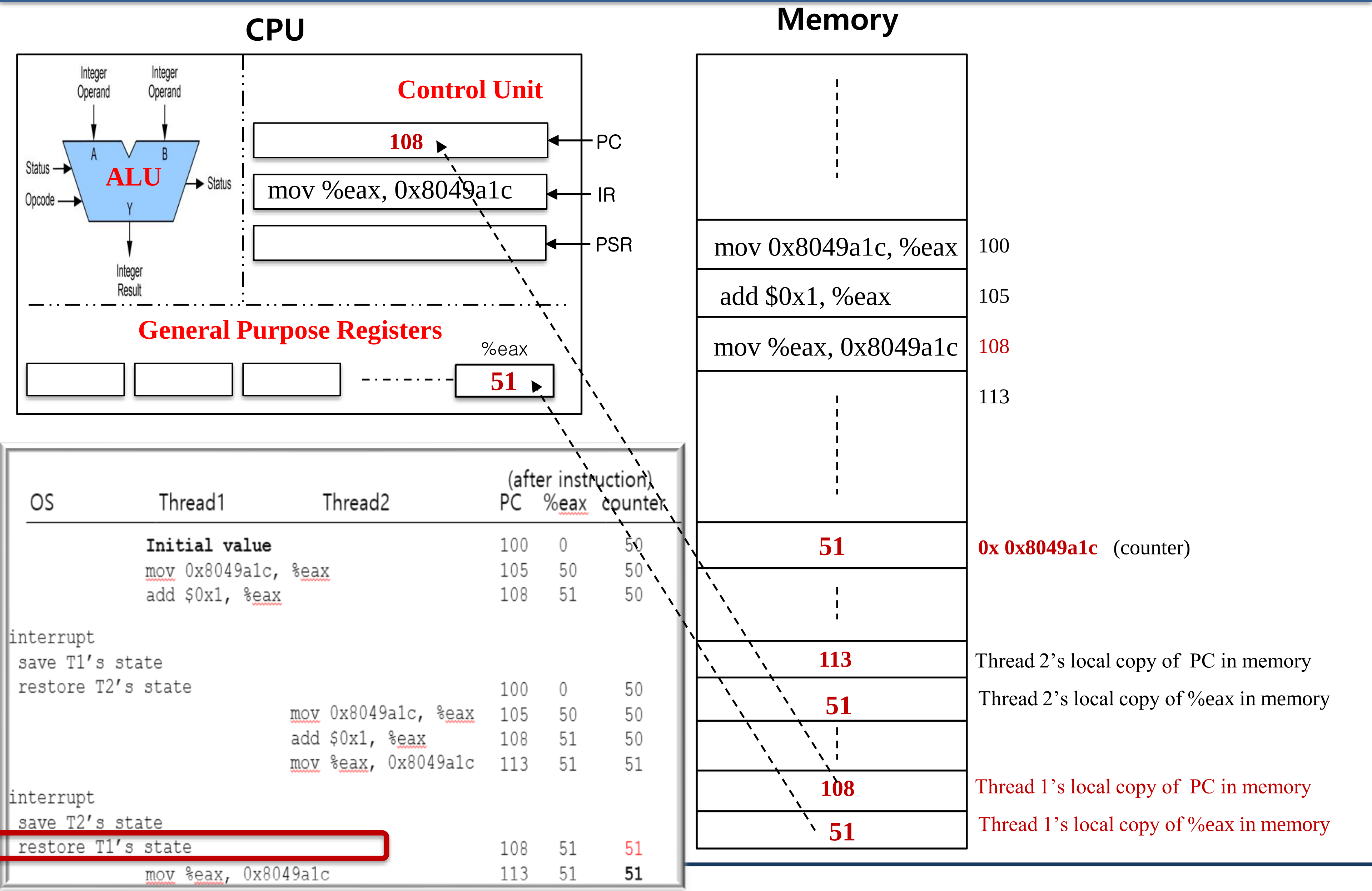


(after instruction)

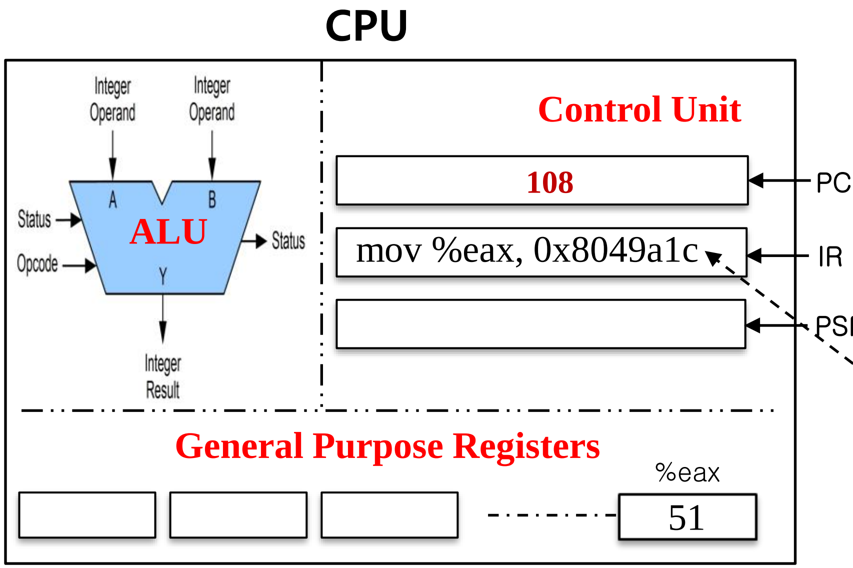
OS	Thread1	Thread2	PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1’s state					
restore T2’s state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2’s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

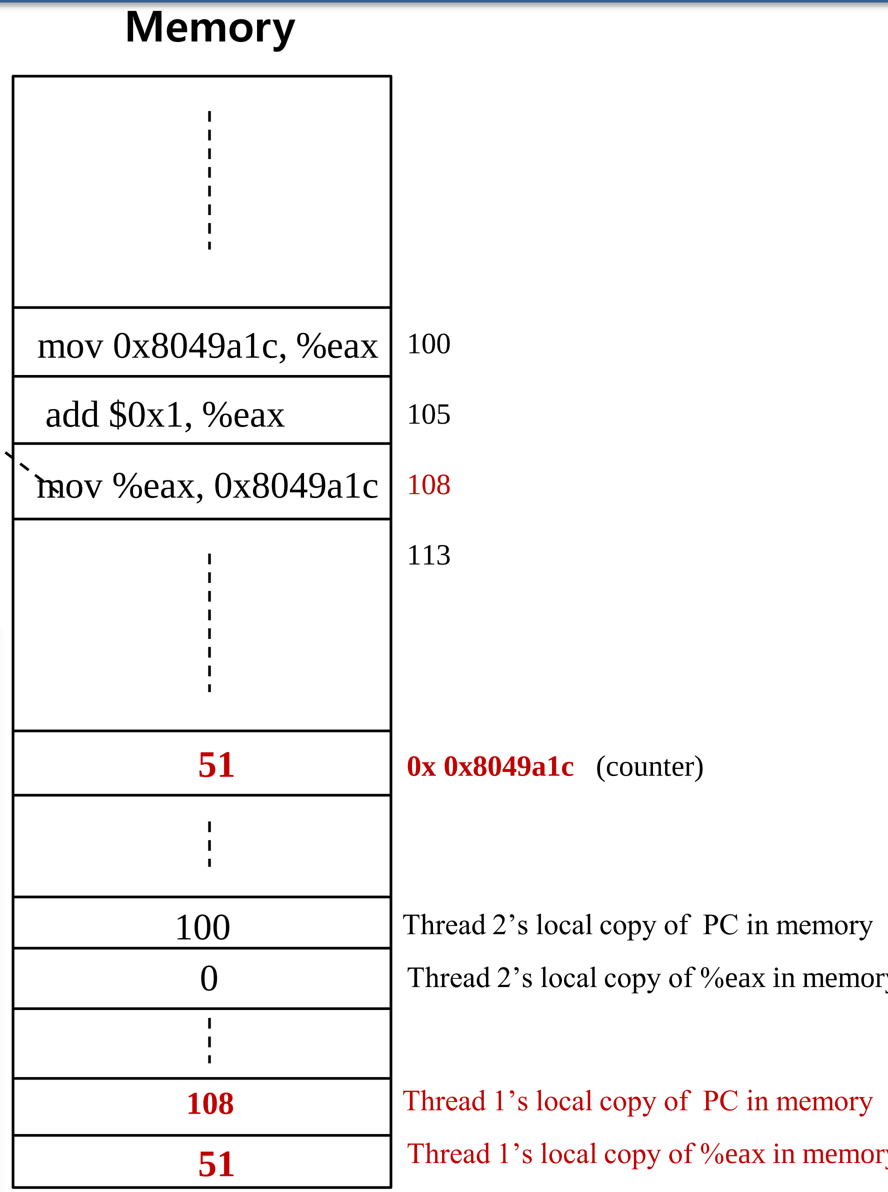


Restore Thread 1’s state



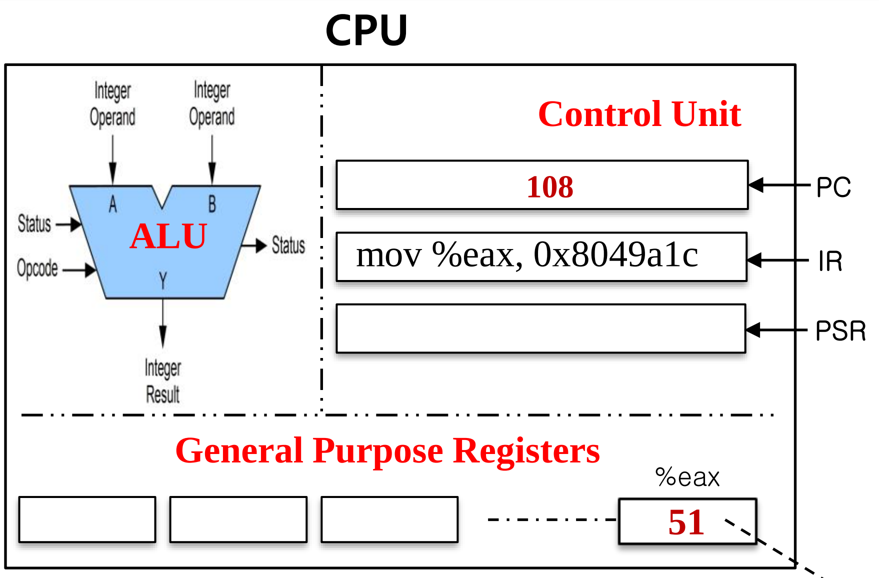
Step 1: Fetch instruction from the memory (PC->Address=108)

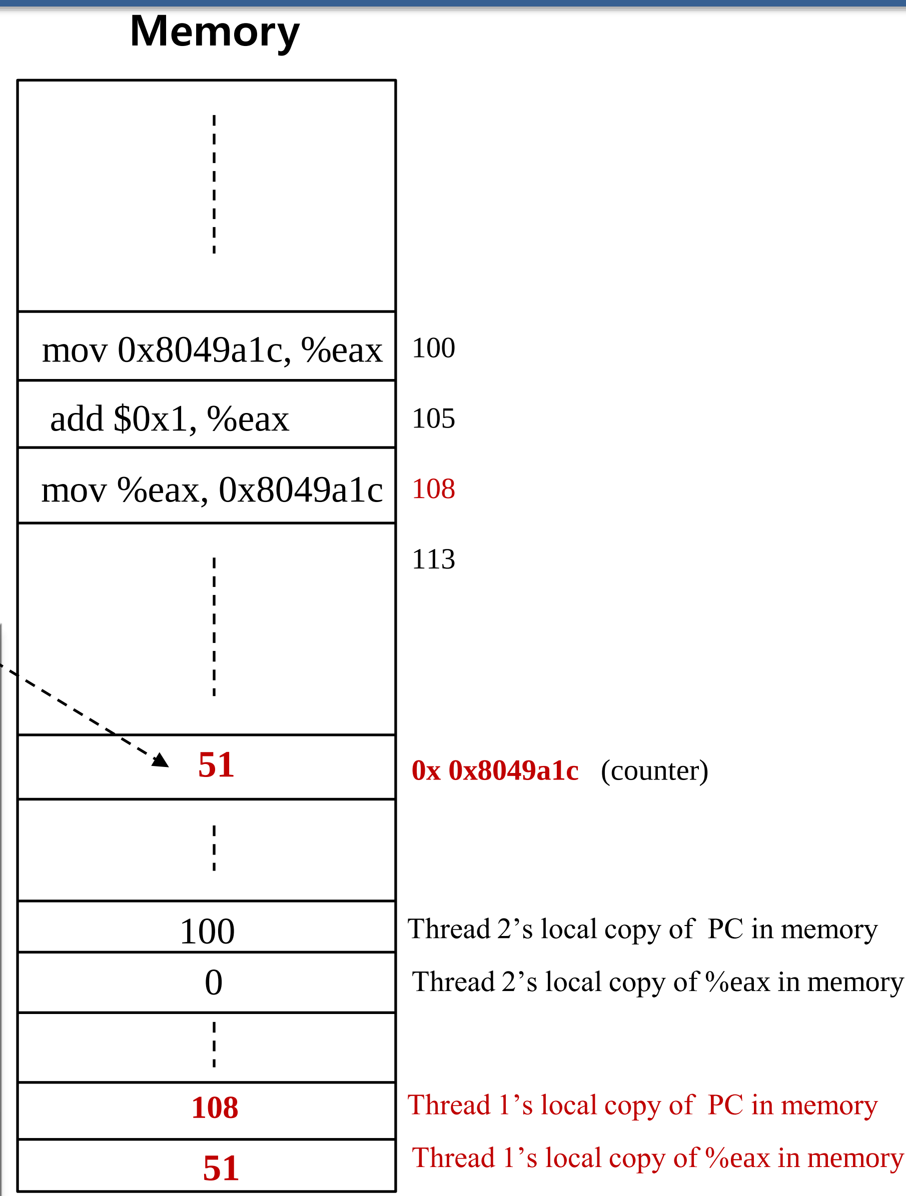




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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

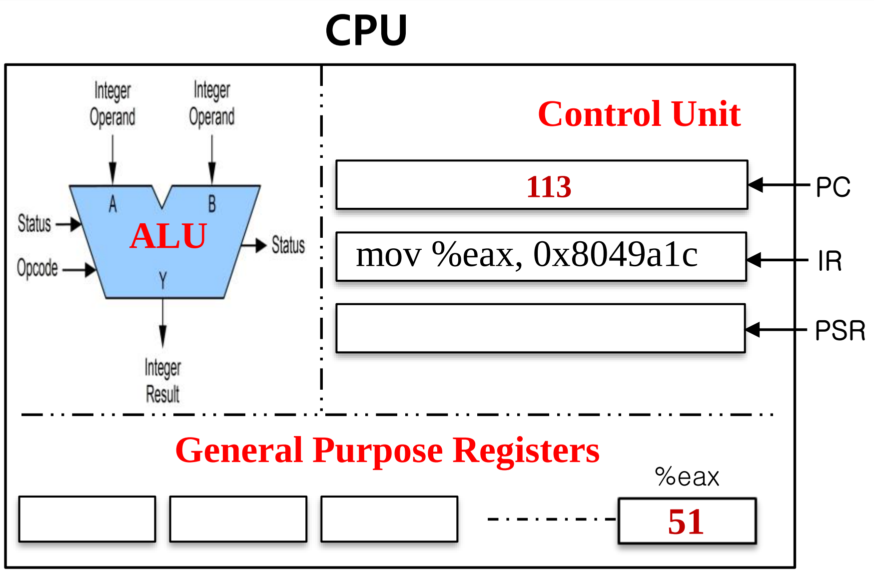
Step 2 & 3: Decode & Execution

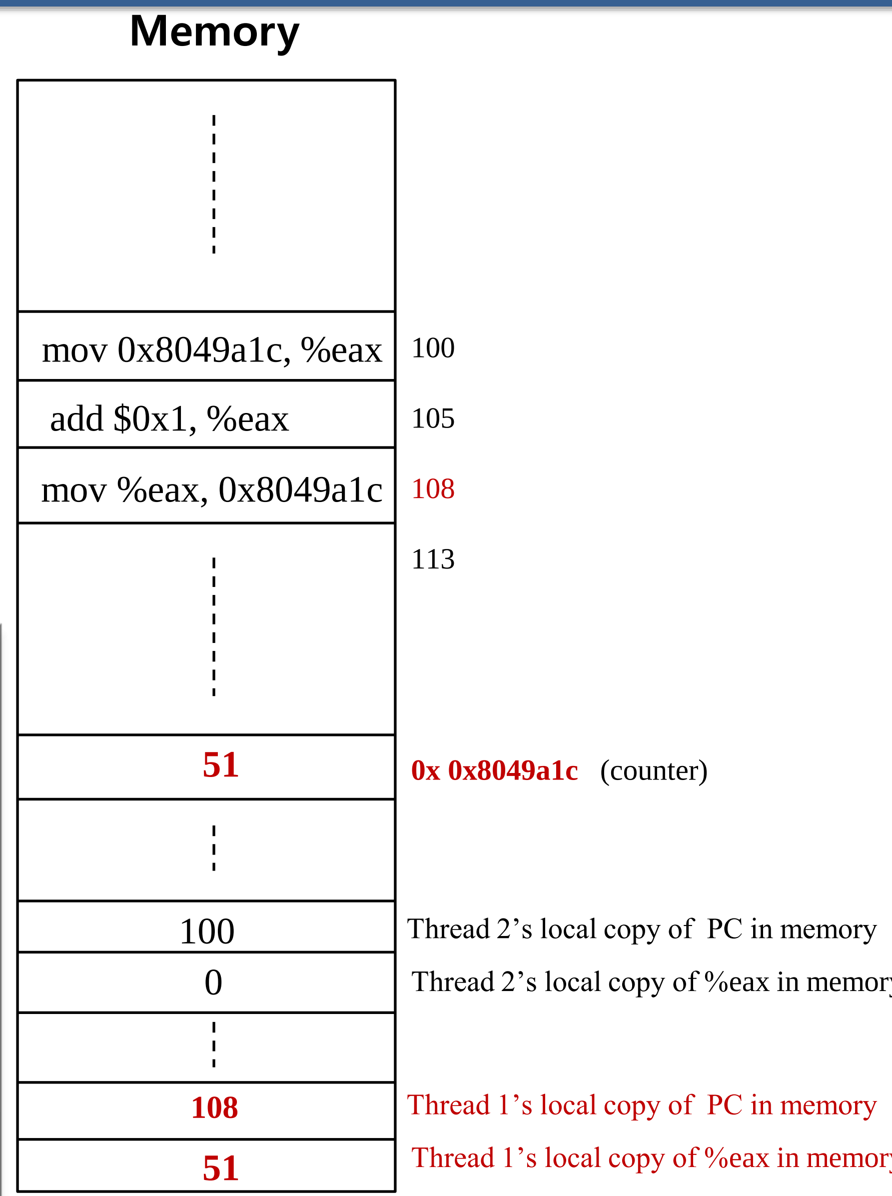




OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	Initial value		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt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





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**

Critical section

- A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

Part II: Thread API

Thread Creation

- How to create and control threads?

```
#include <pthread.h>

int
pthread_create(       pthread_t*        thread,
                const pthread_attr_t* attr,
                      void*             (*start_routine)(void*),
                      void*             arg);
```

- `thread`: Used to interact with this thread.
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, …
- `start_routine`: the function this thread start running in.
- `arg`: the argument to be passed to the function (`start routine`)
  - *a void pointer* allows us to pass in *any type of* argument.

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

Wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, the return value needs to be passed in a pointer to that value.

Example: Waiting for Thread Completion

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

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Waiting for Thread Completion (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                    // waiting inside of the
                                    // pthread_join() routine.
    printf(“returned %d %d\n”, m->x, m->y);
    return 0;
}
```

Example: Dangerous code

- Be careful with <u>how values are returned</u> from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

  - When the variable `r` returns, it is automatically de-allocated.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf(“%d\n”, m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf(“returned %d\n”, m);
    return 0;
}
```

Locks

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Locks (Cont.)

- These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                       struct timespec *abs_timeout);
```

  - `trylock`: return failure if the lock is already held
  - `timelock`: return after a timeout or after acquiring the lock

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait`:
    - Put the calling thread to sleep.
    - Wait for some other thread to signal it.
- `pthread_cond_signal`:
    - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - The wait call releases the lock when putting said caller to sleep.
  - Before returning after being woken, the wait call re-acquires the lock.

- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple **if** statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not* (the condition can be changed before an awakened thread returns from pthread_cond_wait()).

Condition Variables (Cont.)

- Don't ever to this.
  - A thread calling wait routine:

```
while(initialized == 0)
        ; // spin
```

  - A thread calling signal routine:

```
initialized = 1;
```

  - It performs poorly in many cases. → just wastes CPU cycles.
  - It is error prone.

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information,

```
man -k pthread
```

Summary

- A multi-threaded program
  - More than one point of execution (with multiple PCs)
  - Multiple threads share the same address space but with different stacks
- Race condition may occur when multiple threads enter critical sections.
- Thread API
  - Thread creation: pthread_create
  - Thread wait: pthread_join()
  - Lock & unlock: pthread_mutex_lock() / pthread_mutex_unlock()
  - Conditional variable: pthread_cond_wait() / pthread_cond_signal()
- Next: Semaphore