Some Review

Test Logistics

- Date: Monday, March 20th
- Time: 3-6pm (but likely many will finish sooner)
- Place: Kinsey Science Teaching Pavilion: 1220B
  - (Ignore the other room)

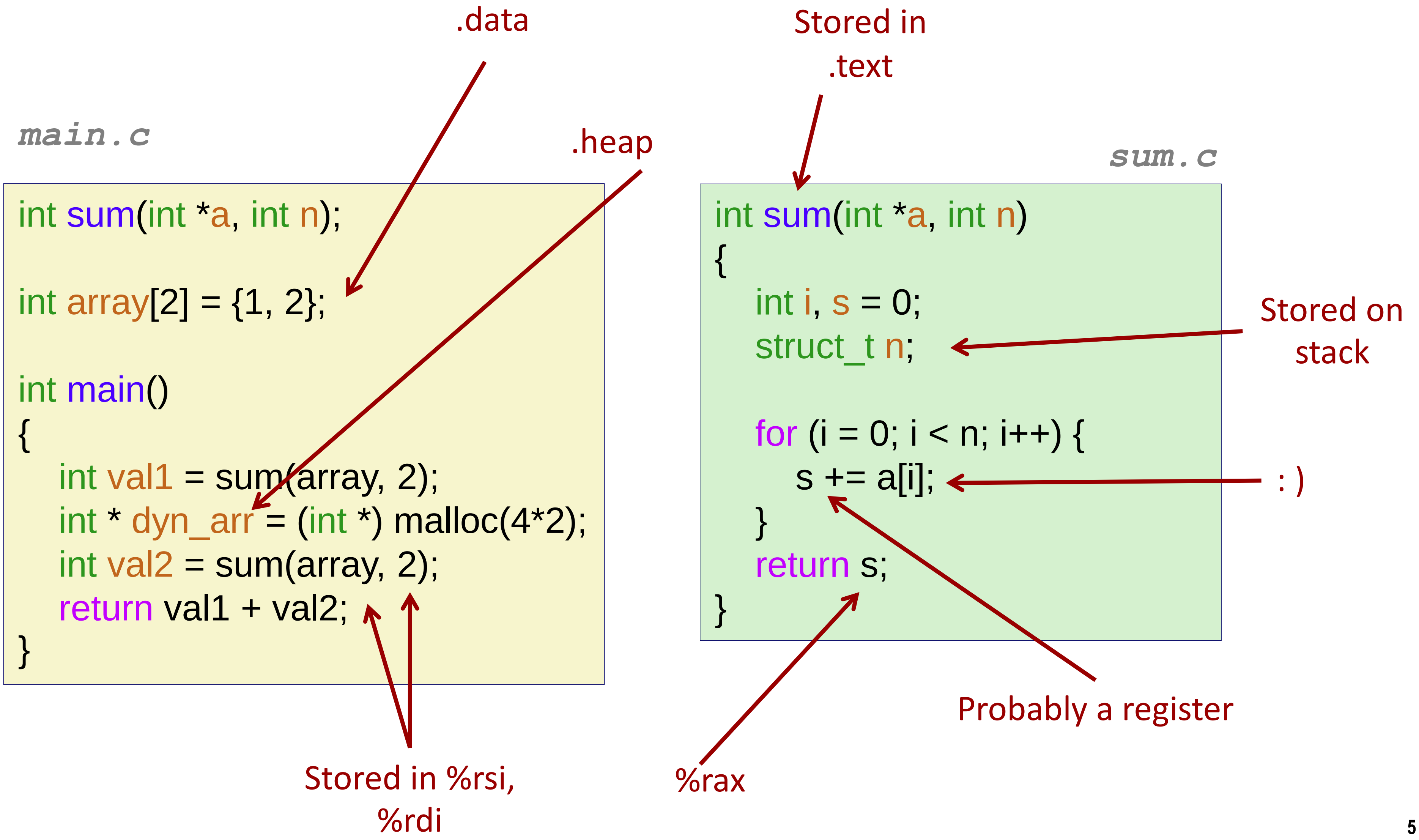
Review

- **Data Representations:**
  - Integers, Bit Manipulation, Floats, Casting & Data Types
- **Machine Level Programming:**
  - Instruction Types, Control Flow (conditionals, procedure calls), Data Types (struct, union), Buffer Overflow
- **Memory Hierarchy:**
  - Principle of caching, locality (temporal, spatial), Memory Hierarchy(disks, memory, cache, register), Virtual memory (TLBs)
- **Low-level Program Optimizations:**
  - Analysis: Calculating latency, CPE, cache miss rates
  - Optimizations for cache (blocking, loop ordering), function inlining, unrolling, strength reduction
- **Parallelism:**
  - Threading (pthreads), race conditions, semaphores, deadlocks.
- **System Guts:**
  - Linking, System Calls, Traps/Exceptions, Forking/Loading

What’s the test going to look like?

- **Many multiple choice / short answer questions**
  - Cover miscellaneous topics – there is no material that is out of bounds. (though I will try to not be mean :)
  - Naming which part of the system is responsible for what
- **Straightforward Questions:**
  - Given a piece of C code, identify where those variables are stored, and other properties (shared, private)
  - Analyze a given piece of code for performance, explain which optimizations could be applied
  - Virtual Memory + Caching

Where’s that Memory?



main.c

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val1 = sum(array, 2);
    int * dyn_arr = (int *) malloc(4*2);
    int val2 = sum(array, 2);
    return val1 + val2;
}
```

sum.c

```
int sum(int *a, int n)
{
    int i, s = 0;
    struct_t n;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

.data

.heap

Stored in .text

Stored on stack

:)

Probably a register

Stored in %rsi, %rdi

%rax

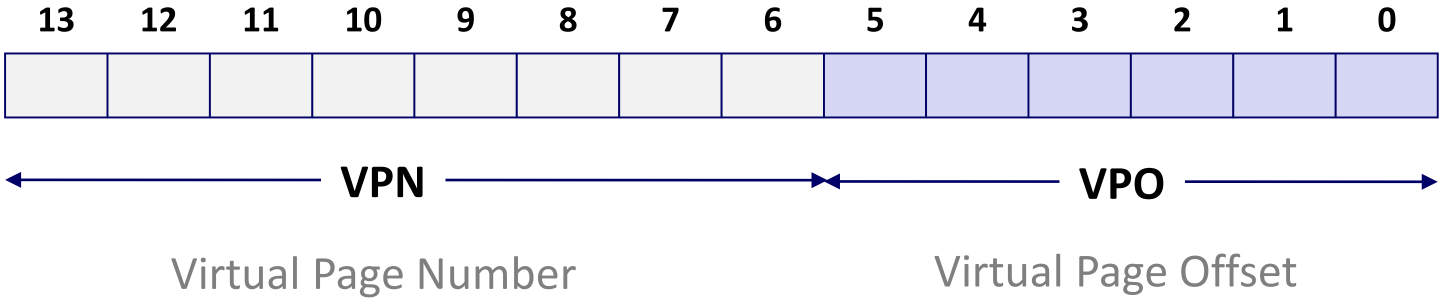
Code Optimizations

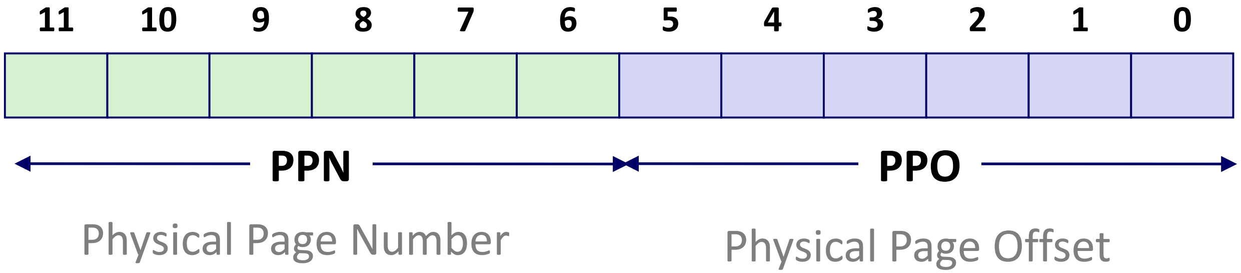
```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
     for (j = 0; j < n; j++)
        for (i = 0; i < n; i++) {
            b[i] += a[i*n + j];
    }
}
```

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
         b[i] = val;
    }
}
```

Simple Memory System Example

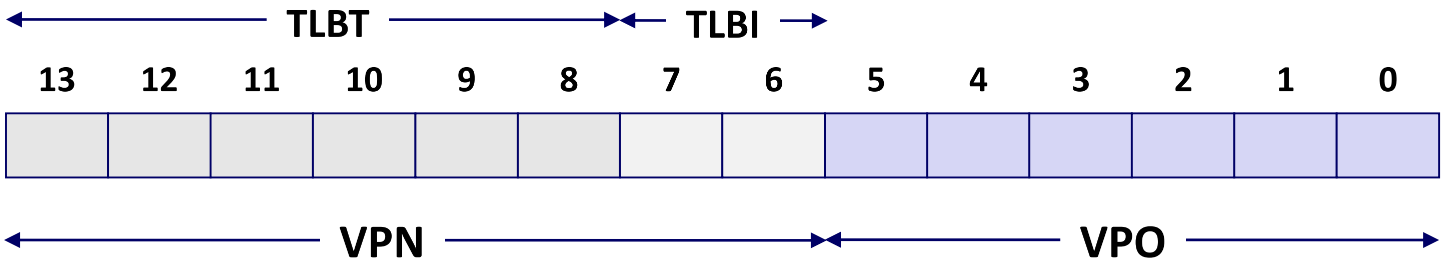
- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes





1. Simple Memory System TLB

- 16 entries
- 4-way associative



TLBT TLBI

13 12 11 10 9 8 7 6 5 4 3 2 1 0

VPN VPO

Set	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. Simple Memory System Page Table

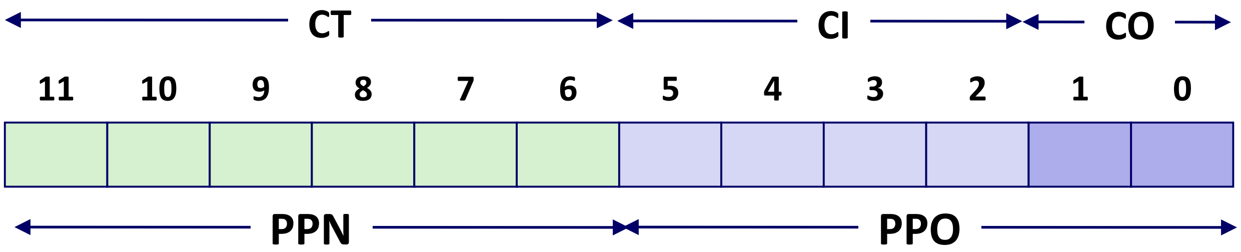
Only show first 16 entries (out of 256)

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

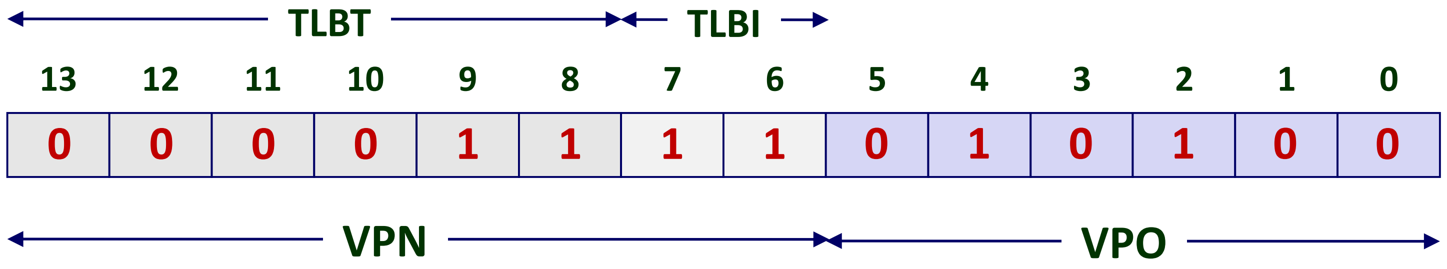


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Address Translation Example #1

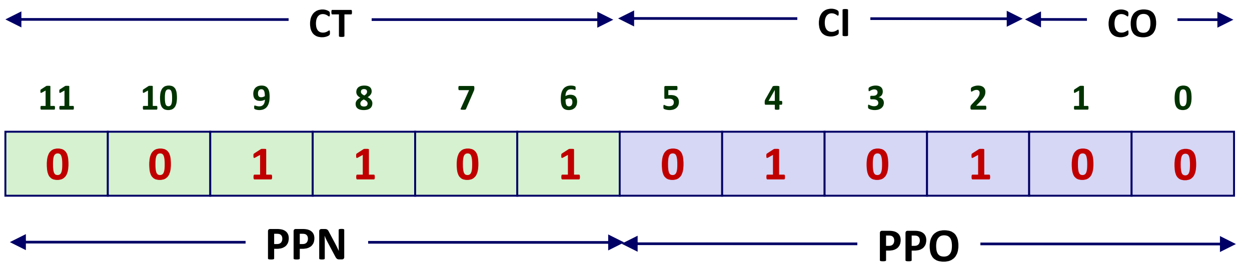
Virtual Address: 0x03D4



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN 0x0F   TLBI 0x3   TLBT 0x03   TLB Hit? Y   Page Fault? N   PPN: 0x0D

Physical Address



11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

CO 0   CI 0x5   CT 0x0D   Hit? Y   Byte: 0x36

Address Translation Example #2

Virtual Address: 0x0020

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN (bits 13–6) | VPO (bits 5–0)

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

PPN (bits 11–6) | PPO (bits 5–0)

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

Hardest Test Questions?

- **Race Question**
  - Pthreads / Joining
  - Forking / Shared Memory
  - Semaphores
- **Attack Lab**
  - Stacks (you will draw a stack based on assembly)
  - Construct / Identify an attack
  - Test knowledge of datatypes
- **Maybe one or two more questions???**

Races and Semaphores

```
int a;
sem_t mutex[2];
int main() {
  Sem_init(&mutex[0], 0, 0);  /* mutex[0] = 0 */
  Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
}
```

Thread 1

```
sem_wait(&mutex[0]);
printf(“what’s up?”);
sem_post(&mutex[1]);
```

Thread 2

```
printf(“hi”);
sem_post(&mutex[0]);
sem_wait(&mutex[1]);
printf(“not much”);
```

Thread 1

```
sem_wait(&mutex[0]);
printf(“what’s up?”);
sem_post(&mutex[0]);
```

Thread 2

```
printf(“hi”);
sem_post(&mutex[0]);
sem_wait(&mutex[0]);
printf(“not much”);
```

Thread 1

```
sem_wait(&mutex[1]);
printf(“what’s up?”);
sem_post(&mutex[1]);
```

Thread 2

```
printf(“hi”);
sem_post(&mutex[1]);
sem_wait(&mutex[1]);
printf(“not much”);
```

Stack and Data

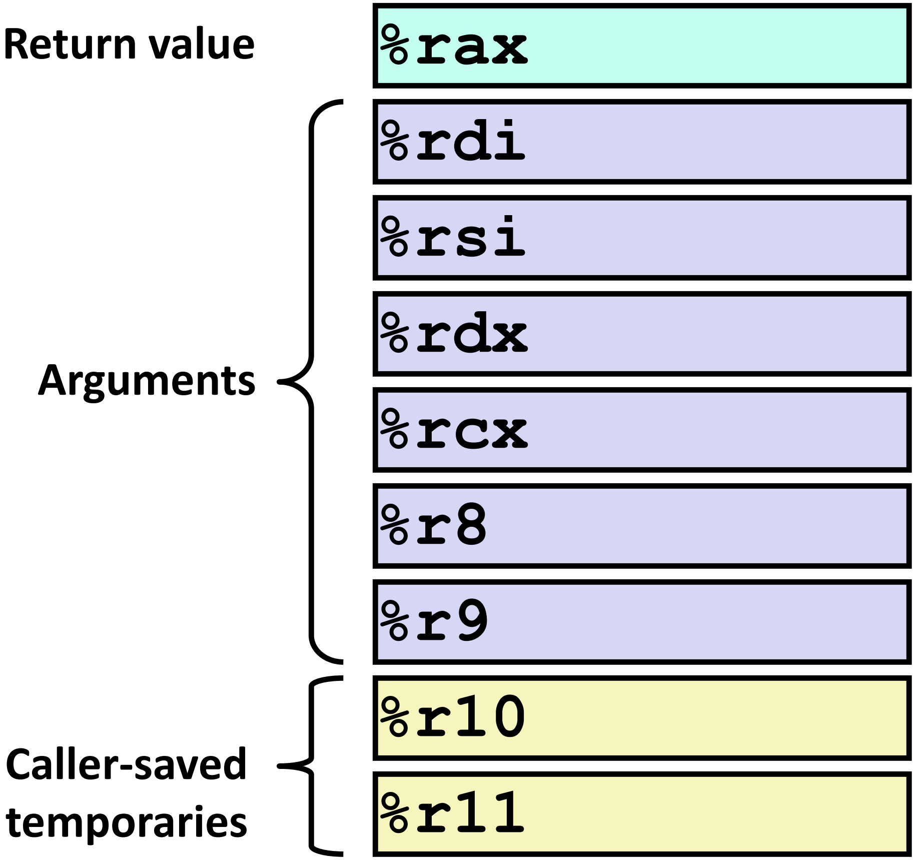
```
int func(int i) {
  volatile union U {
    char s[16];
    int x;
  } u;

  u.x=0;
  u.s[i]=i;
  return u.x;
}
```

```
0x00000000004005d0 <+0>:    movslq %edi,%rax
0x00000000004005d3 <+3>:    movl   $0x0,-0x18(%rsp)
0x00000000004005db <+11>:   mov    %dil,-0x18(%rsp,%rax,1)
0x00000000004005e0 <+16>:   mov    -0x18(%rsp),%eax
0x00000000004005e4 <+20>:   retq
```

func(0)=0x0
func(1)=0x100
func(2)=0x20000
func(3)=0x3000000

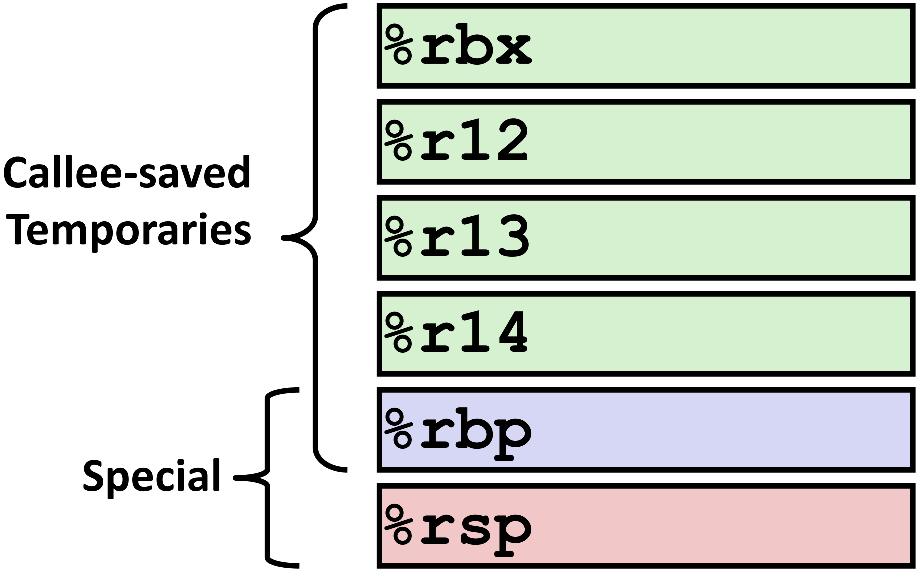
x86-64 Linux Register Usage #1

- **`%rax`**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **`%rdi`**, …, **`%r9`**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **`%r10`**, **`%r11`**
  - Caller-saved
  - Can be modified by procedure



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



Other Questions