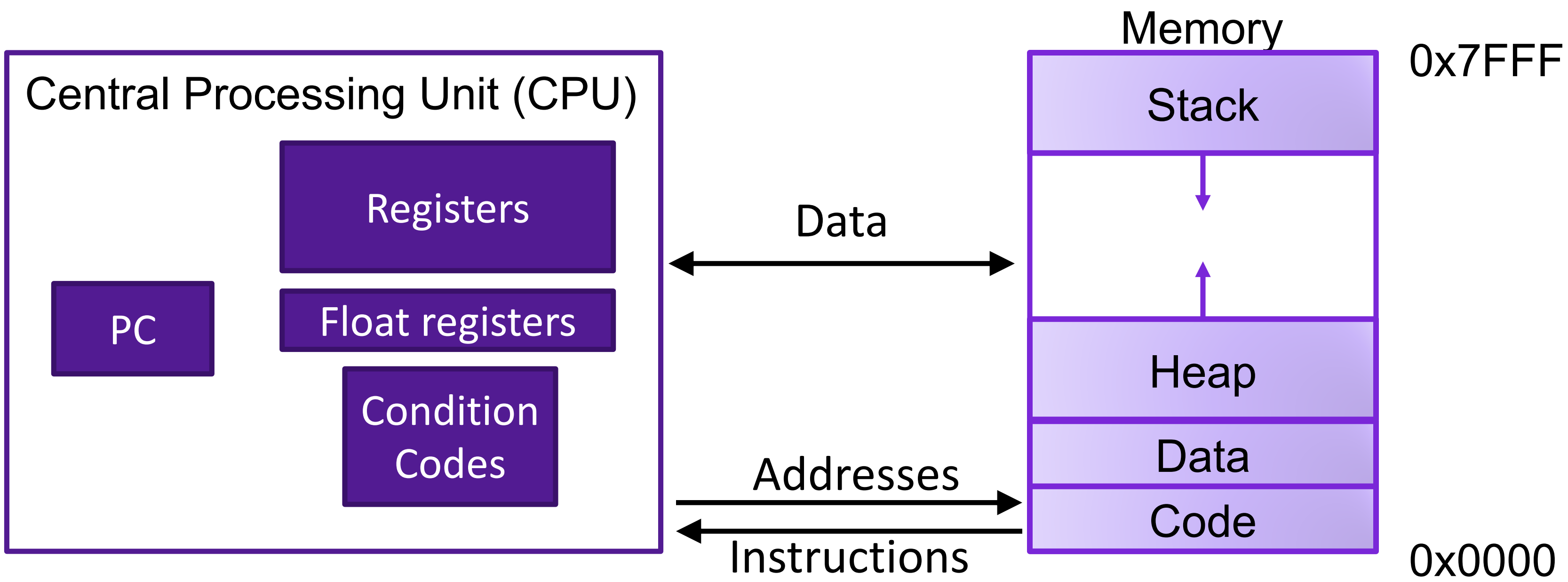
Lecture 7: Loops in Assembly

CS 105

Review: Assembly/Machine Code View



Programmer-Visible State

- PC: Program counter (%rip)
- Register file: 16 Registers
- Float registers
- Condition codes

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Review: Condition Codes

- Single bit registers
  - SF Sign Flag (for signed)
  - ZF Zero Flag
  - OF Overflow Flag (for signed)

- Implicitly set (as a side effect) by arithmetic operations and comparison operations
  - Not set by **`leaq`** instruction

- Explicitly set by special instructions
  - **`cmpq a,b`** (sets same condition codes as computing **`b-a`**)
  - **`testq a,b`** (sets same condition codes as computing **`a&b`**)

Review: Conditional Jumps

- jX instructions
  - Jump to different part of code if condition is true

jX	Condition	Description
jmp	[redacted]	Unconditional
je	[redacted]	Equal / Zero
jne	[redacted]	Not Equal / Not Zero
jl	[redacted]	Less (Signed)
jle	[redacted]	Less or Equal (Signed)
jg	[redacted]	Greater (Signed)
jge	[redacted]	Greater or Equal (Signed)

Recommendation: don't think about condition codes,
compare output of prev operation to 0

Review: Conditionals

Register	Use
%rdi	x
%rsi	y
%rax	result

```
long absdiff(long x, long y){
  long result;
  if (x > y)
  {
    result = x-y;
  } else
  {
    result = y-x;
  }
  return result;
}
```

```
long absdiff(long x, long y){
  long result;
  if (!(x > y)) goto cond2
cond1:
  result = x-y;
  goto cond3
cond2:
  result = y-x;
cond3:
            ult;
```

```
absdiff:
   cmpq    %rsi, %rdi
   jle     .L2
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L2        # x-y <= 0
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Review: Conditionals

```
test:
  leaq (%rdi, %rsi), %rax
  addq %rdx, %rax
  cmpq $-3, %rdi
  jge .L2
  cmpq %rdx, %rsi
  jge .L3
  movq %rdi, %rax
  imulq %rsi, %rax
  ret
.L3:
  movq %rsi, %rax
  imulq %rdx, %rax
  ret
.L2
  cmpq $2, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
.L4:
  rep; ret
```

Reg	Use
%rdi	x
%rsi	y
%rdx	z
%rax	val

```
long test(long x, long y, long z){
  long val = _x + y + z_;

  if(_x + 3 < 0_){

    if(_y - z < 0_){

      val = _x*y_;


    } else {
      val = _y*z_;

    }

 } else if (_x - 2 > 0_){

    val = _x*z_;

  }
  return val;
}
```

Loops

- All use conditions and jumps
  - do-while
  - while
  - for

Do-while Loops

Register	Use(s)
`%rdi`	Argument **x**
`%rax`	`result`

```
long bitcount(unsigned long x){
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x != 0);
  return result;
}
```

```
long bitcount(unsigned long x){
  long result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if(x != 0) goto loop;
  return result;
}
```

```
   movq     $0, %rax     #  result = 0
.L2:                     #  loop:
   movq     %rdi, %rdx
   andq     $1, %rdx     #  t = x & 0x1
   addq     %rdx, %rax   #  result += t
   shrq     %rdi, $1     #  x >>= 1
   jne      .L2          #  if (x) goto loop
   rep; ret
```

While Loops

Register	Use(s)
%rdi	Argument **x**
%rax	**result**

```
long bitcount(unsigned long x){
  long result = 0;
  while (x != 0) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

?

```
        movq    $0, %rax
        jmp     .L2
.L3:
        movq    %rdi, %rdx
        andq    $1, %rdx
        addq    %rdx, %rax
        shrq    %rdi, $1
.L2:
        testq   %rdi, %rdi
        jne     .L3
        rep ret
```

```
        movq    $0, %rax
.L1:
        test    %rdi,%rdi
        je      .L2
        movq    %rdi, %rdx
        andq    $1, %rdx
        addq    %rdx, %rax
        shrq    %rdi, $1
        jmp     .L1
.L2:
        rep ret
```

Exercise: Loops

Reg	Use(s)
%rdi	Argument **val**
%rdx	Local **i**
%rax	Local **ret**

```
loop:
  movq $0, %rax
  movq $0, %rdx
  jmp L1
L0:
  addq %rdx, %rax
  incq %rdx
L1:
  cmp %rdi, %rdx
  jl L0
  ret
```

```
long loop(long val){
  long ret = ______;
  long i   = ______;

  while(__________){

    ret = _______;
    i   = _______;

  }

  return ret;
}
```

For loops

Register	Use(s)
`%rdi`	Argument **x**
`%rax`	**result**

```
for (Init; Cond; Incr){
      Body
}
```

```
Init;
while (Cond) {
      Body;
      Incr;
}
```

Initial test can often be optimized away:

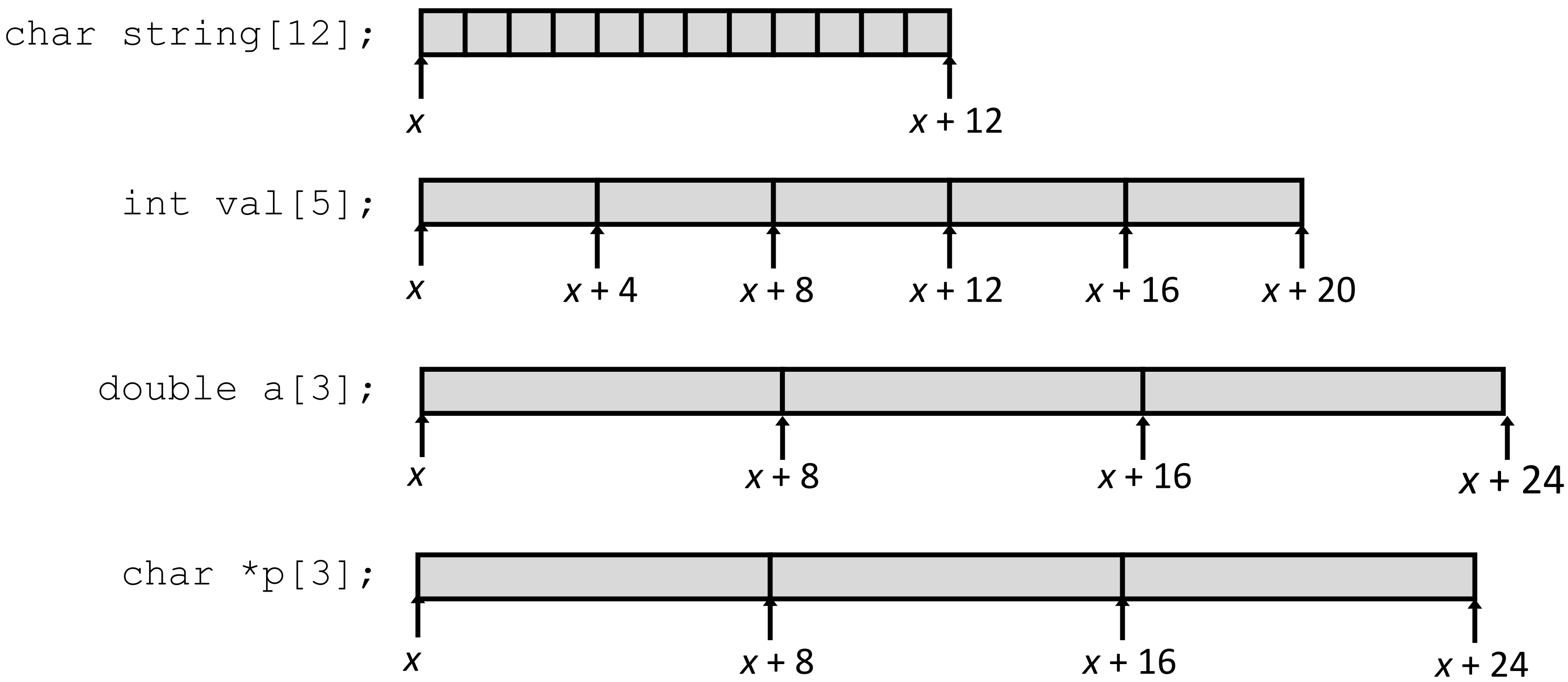
```
  for (j = 0; j < 99; j++)
```

```
long bitcount(unsigned long x) {
  long result;
  for (result = 0; x!=0; x >>= 1)
    result += x & 0x1;
  return result;
}
```

```
       movq    $0, %rax
.L1:
       test    %rdi,%rdi
       je      .L2
       movq    %rdi, %rdx
       andq    $1, %rdx
       addq    %rdx, %rax
       shrq    %rdi, $1
       testq   %rdi, %rdi
       jmp     .L1
.L2:
       rep ret
```

Review: Array Allocation

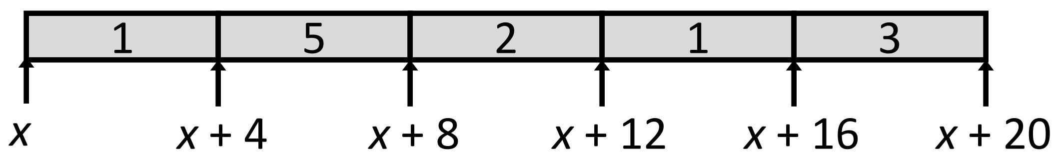
- Basic Principle *T* **`A[`***L***`]`***;*
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`** (*T*) bytes in memory
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**



Exercise: Array Access

- Basic Principle *T* **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

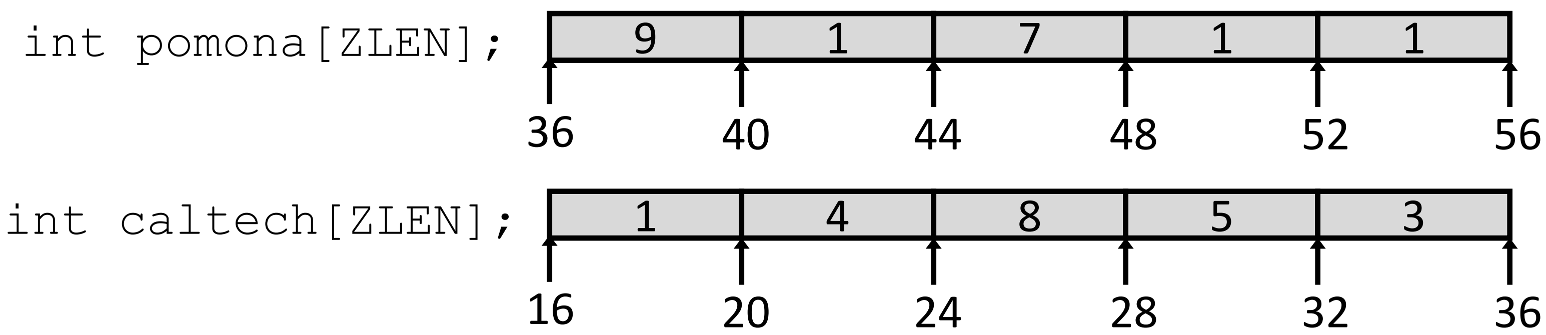
```
int val[5];
```



Reference	Type	Value
val[4]		
val		
val+1		
&(val[2])		
val[5]		
***(val+1)**		

Array Example

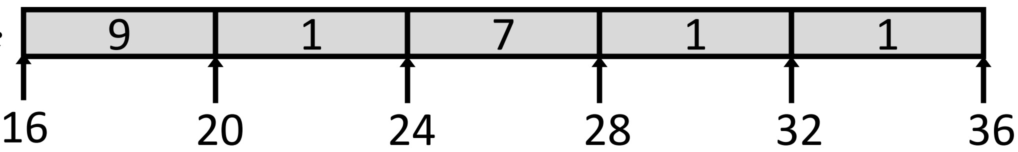
```
#define ZLEN 5

int pomona[ZLEN]  = { 9, 1, 7, 1, 1 };
int caltech[ZLEN] = { 9, 1, 1, 2, 5 };
```



Register	Use(s)
%rdi	z
%rsi	digit

Array Accessing Example

`zip_code pomona;`




```
int get_digit(int * z, int digit){
  return z[digit];
}
```

```
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Exercise : Array Loop

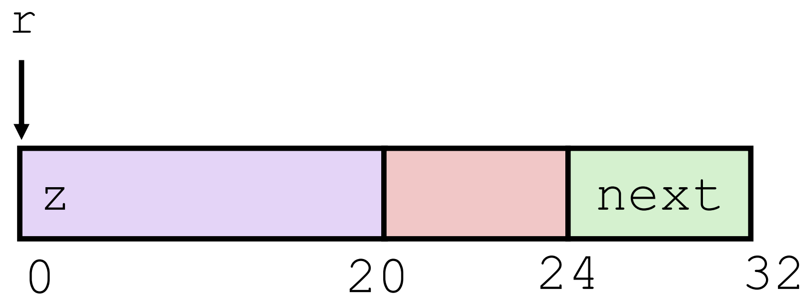
Variable	Register
z	
sum	
i	

```
array_loop:
  movl    $0, %esi
  xorl    %eax, %eax
  jmp     L2
L1:
  addl    (%rdi,%rsi,4), %eax
  incq    %rsi
L2:
  cmpq    $5, %rsi
  jl      L1
  retq
```

```
int array_loop(int * z) {
  int sum = _______;
  int i;

  for(i = ___; i < ___; ___ )

    sum = ________;

  }
  return ________;
}
```

Structure Representation

```
struct rec {
  int z[5];
  struct rec *next;
};
```

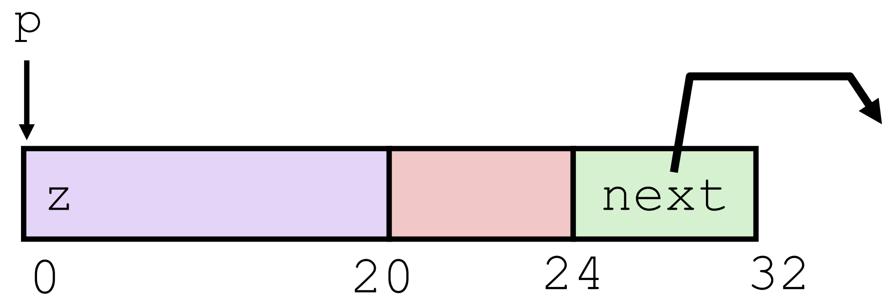


- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

Register	Value
%rdi	p

Following Linked List

```
typedef struct rec {
  int z[5];
  struct rec *next;
} zip_node;
```




```
zip_node*get_tail_ptr(zip_node *p){
  if(p == NULL){
    return NULL;
  }

  while(p->next != NULL){
    p = p->next;
  }

  return p;
}
```

```
get_tail_ptr:
  testq   %rdi, %rdi
  jne      L1
  xorl    %eax, %eax
  retq
L1:
  movq    %rdi, %rax
  movq    24(%rax), %rdi
  testq   %rdi, %rdi
  jne     L1
  retq
```