Structures and Intro to Y86-64

CSCI 237: Computer Organization
13th Lecture, Mar 10, 2025

Jeannie Albrecht

Administrative Details

- Lab 2: final 2 phases due Tues/Wed
  - If you haven't at least partially figured out phase 4, I am a little worried. Come see me and/or TAs if you need help!
- HW 4 due Friday
- Midterm review session Tuesday 3/18 in the evening?
  - Details TBD
  - Sample midterms and solutions posted on course webpage
- Midterm during lab next week
  - I will try to reserve a classroom
  - Let me know ASAP if you have accommodations and need special arrangements

Last time

- Procedure register saving conventions
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Today: Wrap up Ch 3

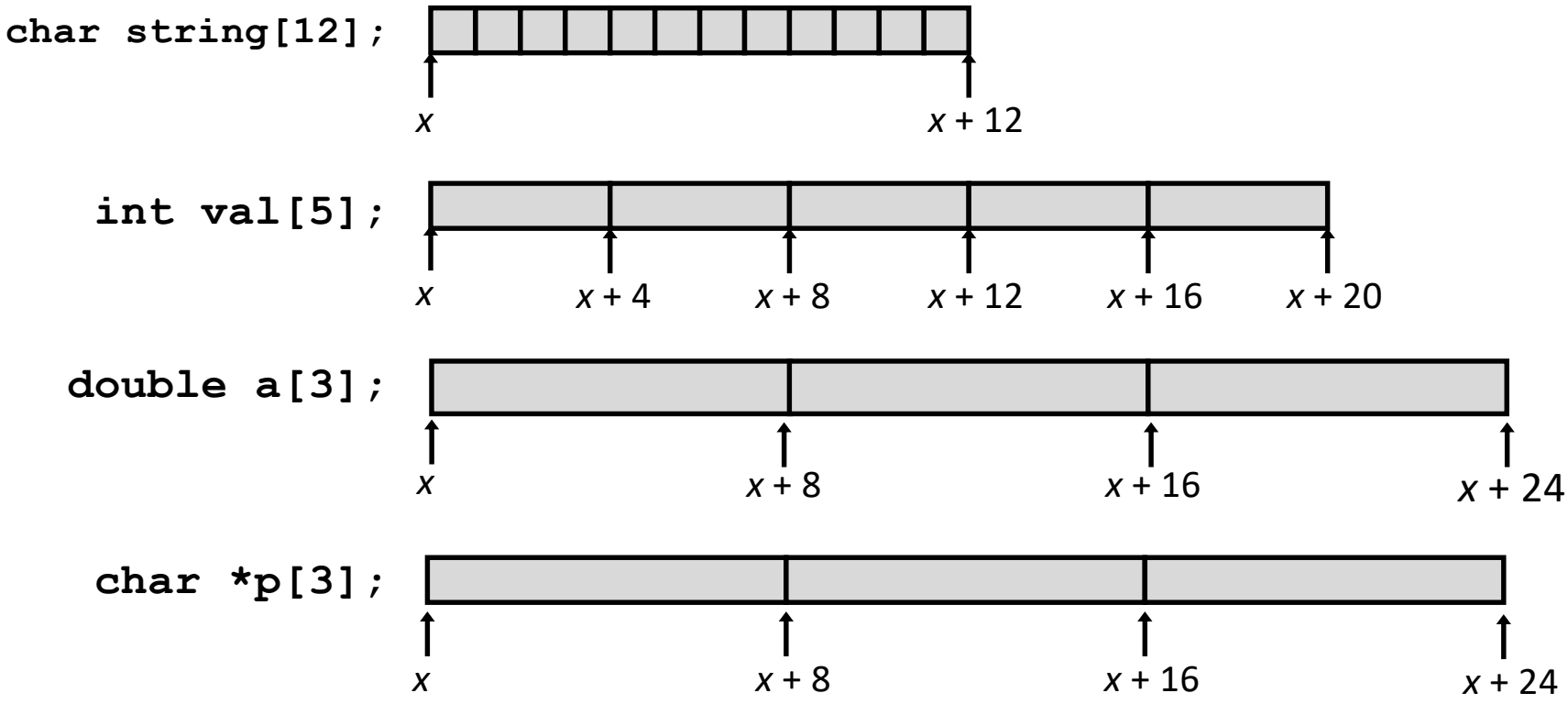
- Arrays (Ch 3.8)
  - Multi-dimensional (nested)
  - Multi-level
- Structures (Ch 3.9)
  - Allocation
  - Access
  - Alignment
- Intro to Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC

Recap: Array Allocation

- Basic Principle

  T **`A[`***L***`];`**

  - Array A of data type *T* and length *L*
  - **Contiguously** allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory
  - *x* is an address

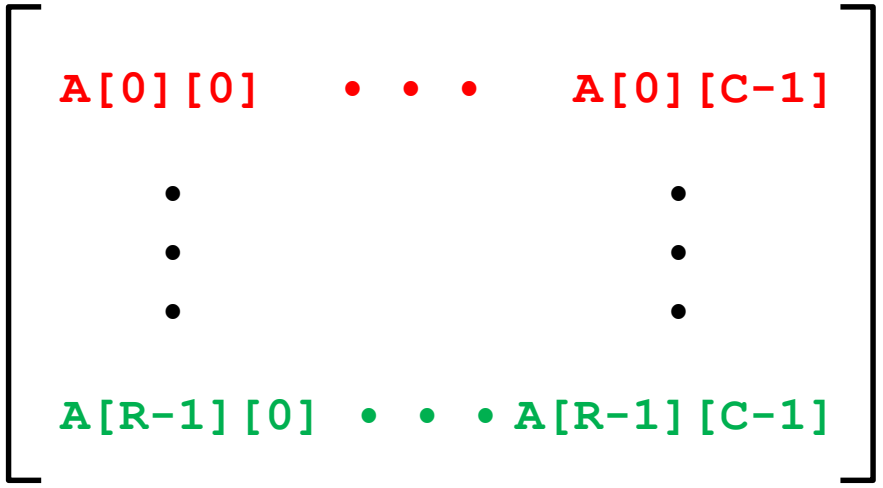


Multidimensional (Nested) Arrays

- Declaration

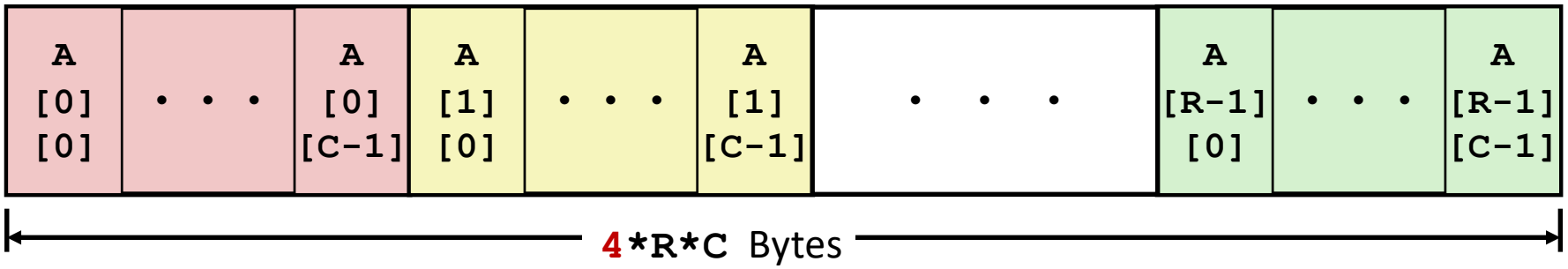
  T **A**[*R*][*C*];

  - 2D array A of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes



- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering

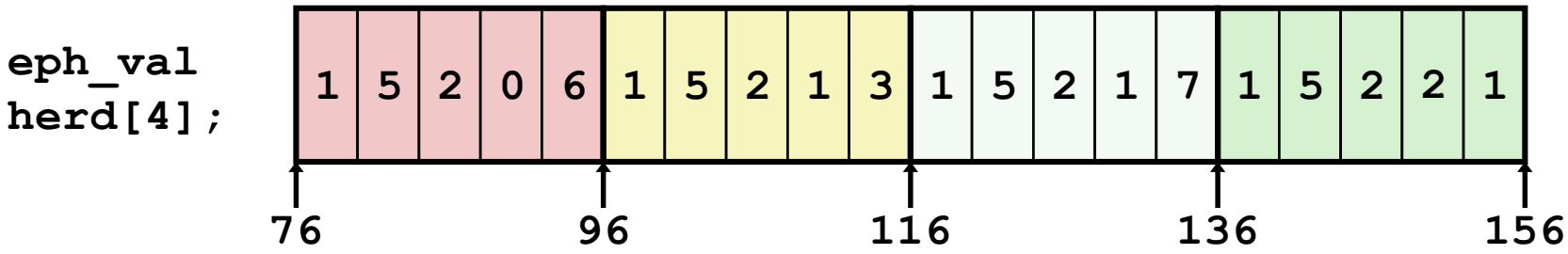
```
int A[R][C];
```



Nested Array Example

```
#define LEN 5
typedef int eph_val[LEN];
```

```
#define COUNT 4
eph_val herd[COUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

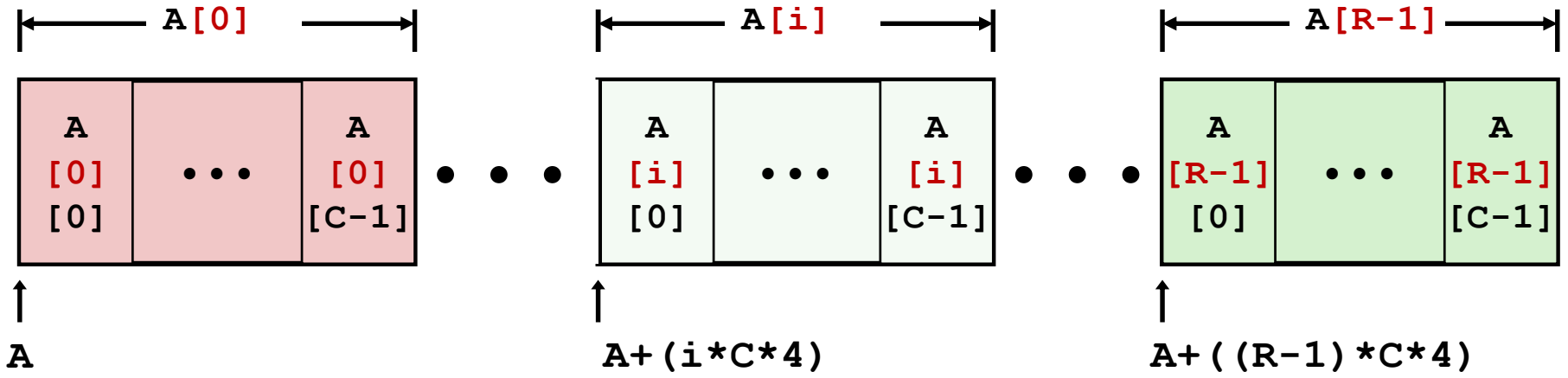


- `eph_val herd[4]` equivalent to `int herd[4][5]`
  - Variable **`herd`**: array of 4 elements, allocated contiguously
  - Each element in **`herd`** is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

Nested Array Row Access

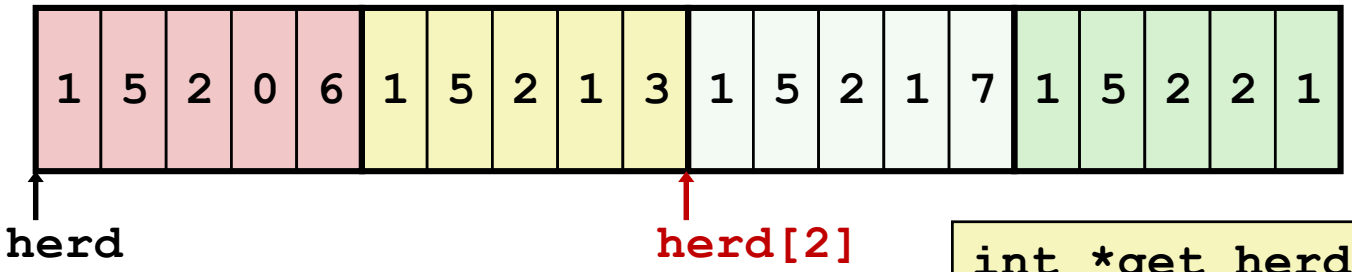
- Row Vectors
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A + i * (C * K)`**

```
int A[R][C];
```



Nested Array Row Access

```
//LEN=5, COUNT=4
typedef int eph_val[LEN];
eph_val herd[COUNT];
```




```
int *get_herd_eph(int index){
  return herd[index];
}
```

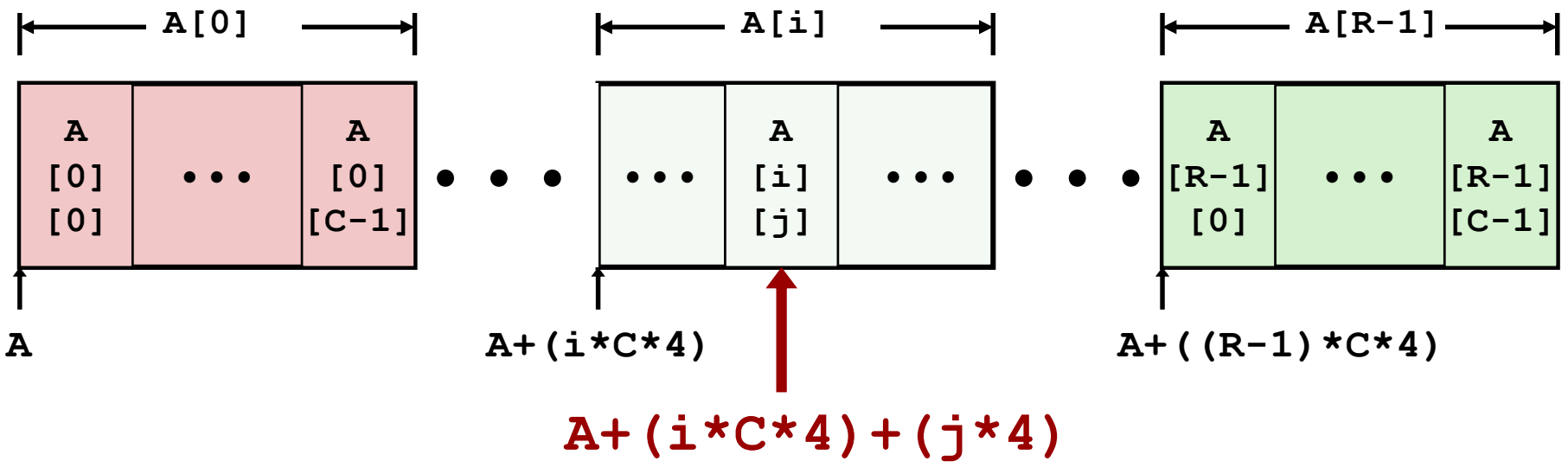
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq herd(,%rax,4),%rax # herd + (20 * index)
```

- Row Vector
  - **`herd[index]`** is an **`eph_val`** (an array of 5 **`int`**'s)
  - Starting address **`herd + 20*index`**
- Machine Code
  - Computes and returns address
  - Compute as **`herd + 4*(index + 4*index)`**

Nested Array Element Access

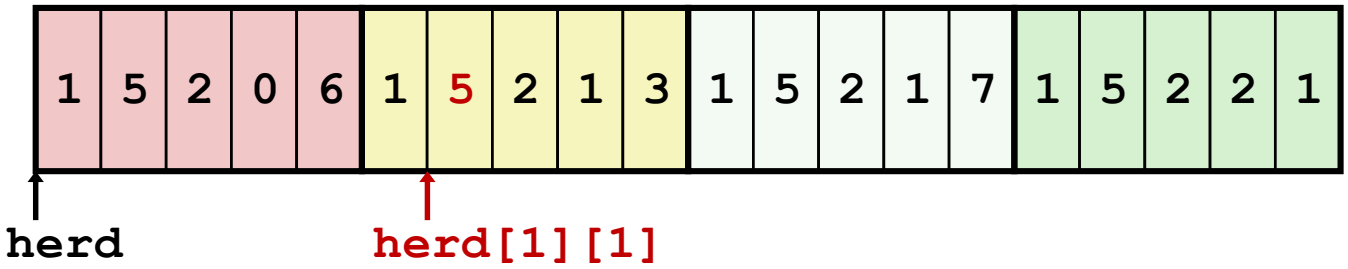
- Array Elements
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address `A + i * (C * K) + j * K`
    `= A + (i * C + j) * K`

```
int A[R][C];
```



Nested Array Element Access

```
//LEN=5, COUNT=4
typedef int eph_val[LEN];
eph_val herd[COUNT];
```

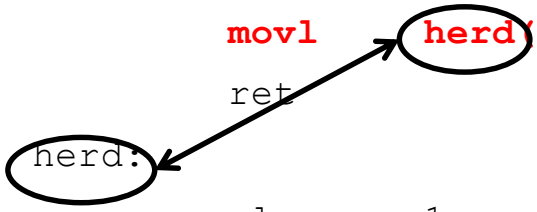



```
int get_herd_value(int index, int val){
  return herd[index][val];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+val
movl   herd(,%rsi,4), %eax    # M[herd + 4*(5*index+val)]
```

- Array Elements
  - **`herd[index][val]`** is **`int`**
  - Address: **`herd + 20*index + 4*val`**
    = **`herd + 4*(5*index + val)`**

```
get_herd_value:
.LFB0:
        movslq  %esi, %rsi
        movslq  %edi, %rdi
        leaq    (%rdi,%rdi,4), %rax
        addq    %rax, %rsi
        movl    herd(,%rsi,4), %eax
        ret
herd:
        .long   1
        .long   5
        .long   2
        .long   0
        .long   6
        .long   1
        .long   5
        .long   2
        .long   1
        .long   3
        .long   1
        . . .
```



Nested Arrays Summary

- Allocated contiguously in memory
- We can conveniently locate any element using math
- We will see later that these arrays are also “cache friendly”

- However, there are other ways to make 2-D arrays.
  - What if we wanted to assemble an array out of pointers to existing arrays?

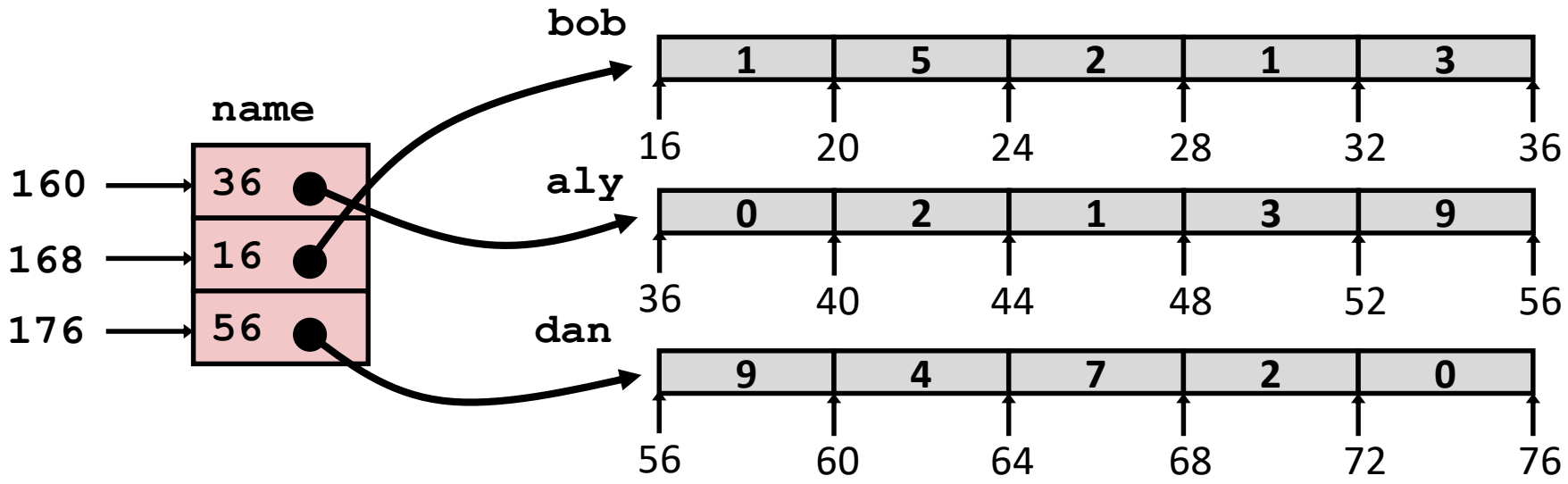
Multi-Level Array Example

```
//LEN=5
typedef int eph_val[LEN];
```

```
eph_val bob = { 1, 5, 2, 1, 3 };
eph_val aly = { 0, 2, 1, 3, 9 };
eph_val dan = { 9, 4, 7, 2, 0 };
```

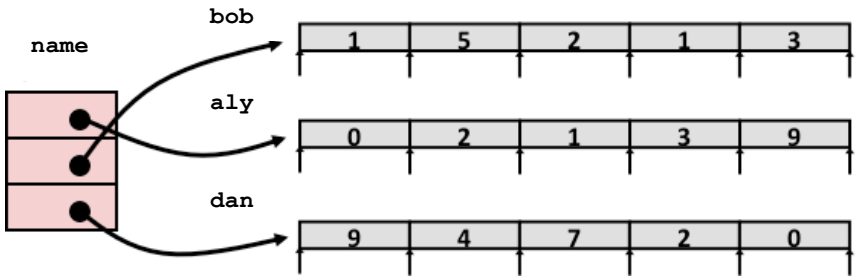
```
#define COUNT 3
int *name[COUNT] = {aly, bob, dan};
```

- Variable `name` denotes array of 3 elements
- Each element is a **pointer**
  - 8 bytes
- Each pointer points to an **`eph_val`** (an array of `int`'s)



Element Access in Multi-Level Array

```
int get_name_value
   (size_t index, size_t val) {
   return name[index][val];
}
```




```
   salq    $2, %rsi                 # 4*val
   addq    name(,%rdi,8), %rsi      # p = name[index] + 4*val
   movl    (%rsi), %eax             # return *p
   ret
```

- Computation
  - Element access **Mem[Mem[name+8*index]+4*val]**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

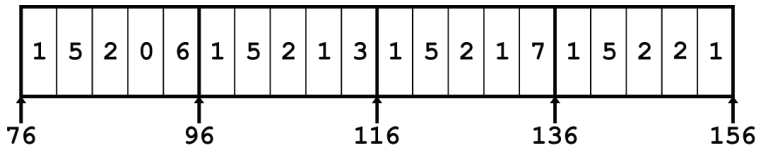
Array Element Accesses

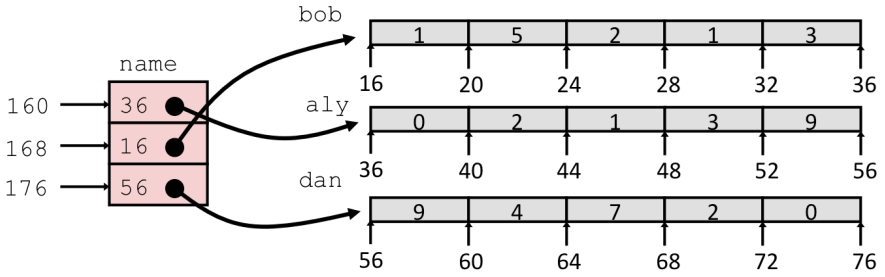
Nested array

```
int get_herd_value
   (size_t index, size_t val) {
   return herd[index][val];
}
```

Multi-level array

```
int get_name_value
   (size_t index, size_t val) {
   return name[index][val];
}
```





Accesses looks similar in C, but address computations very different:

`Mem[herd+20*index+4*val]`

`Mem[Mem[name+8*index]+4*val]`

Moving on: Struct Overview

- Structs are a way to make “composite types” in C
- Syntax:

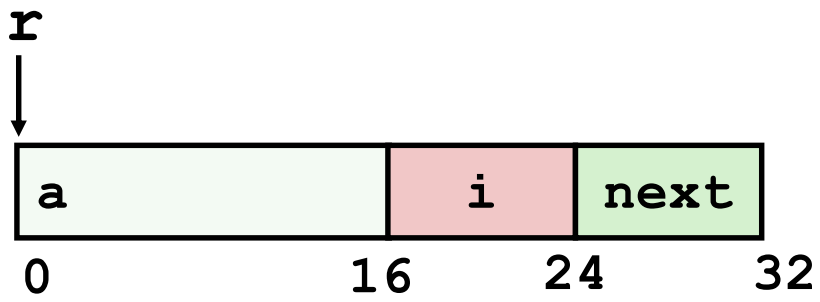
```
struct name {
        type_0 name_0;
        type_1 name_1;
        type_2 name_2;
        …
};

…

struct name var;
var.name_0 = val;
var.name_2 = another_val;
…
```

Ch 3.9 - Structure Representation

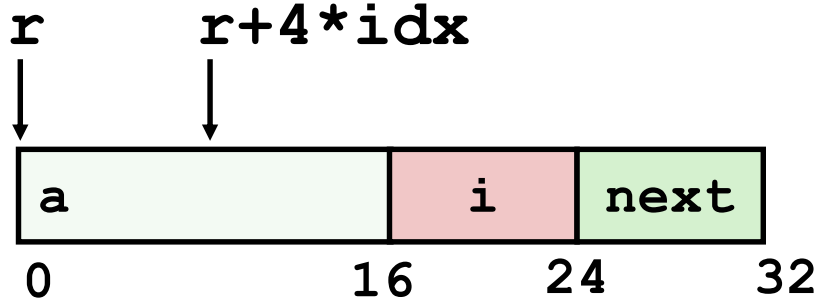
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
…
struct rec r;
```



- Above example is a recursive data structure (modified linked list)
- Structure represented as contiguous 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**

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap
  (struct rec *r, size_t idx) {
   return &r->a[idx];
}
```

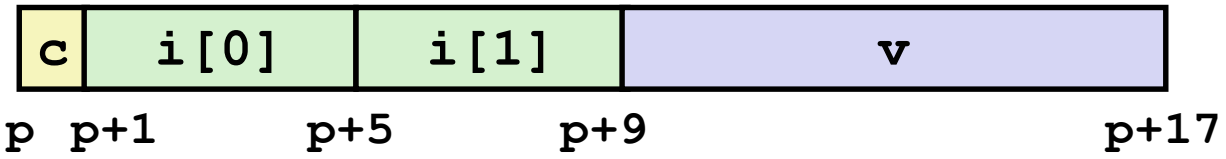
```
   # r in %rdi, idx in %rsi
   leaq   (%rdi,%rsi,4), %rax
   ret
```

Structures & Alignment

- Aligned Data
  - Primitive data type requires `k` bytes
  - Address must be multiple of `k`
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store data that spans quad word boundaries
- Compiler
  - **Inserts gaps** in structs to ensure correct alignment of fields

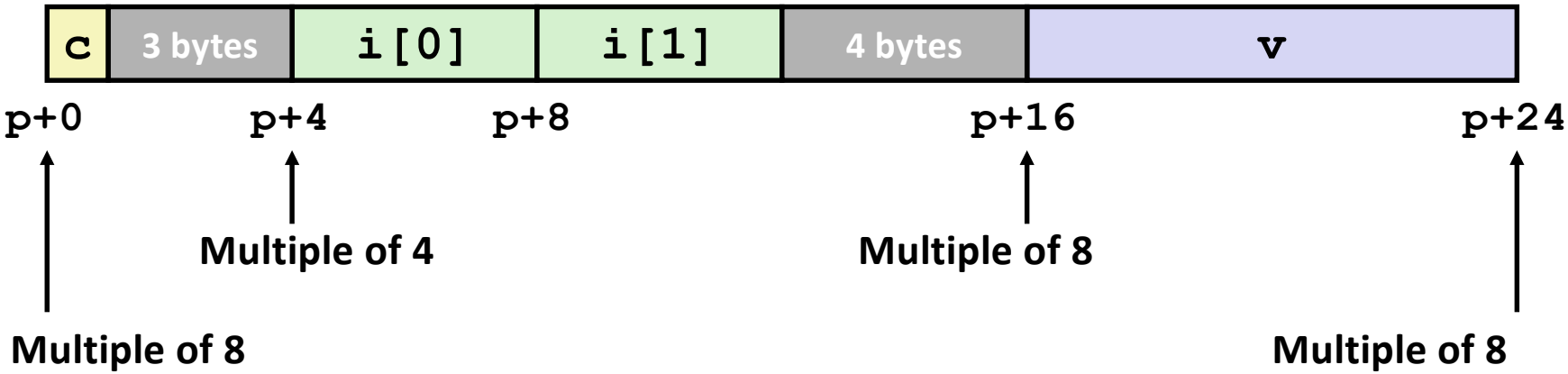
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

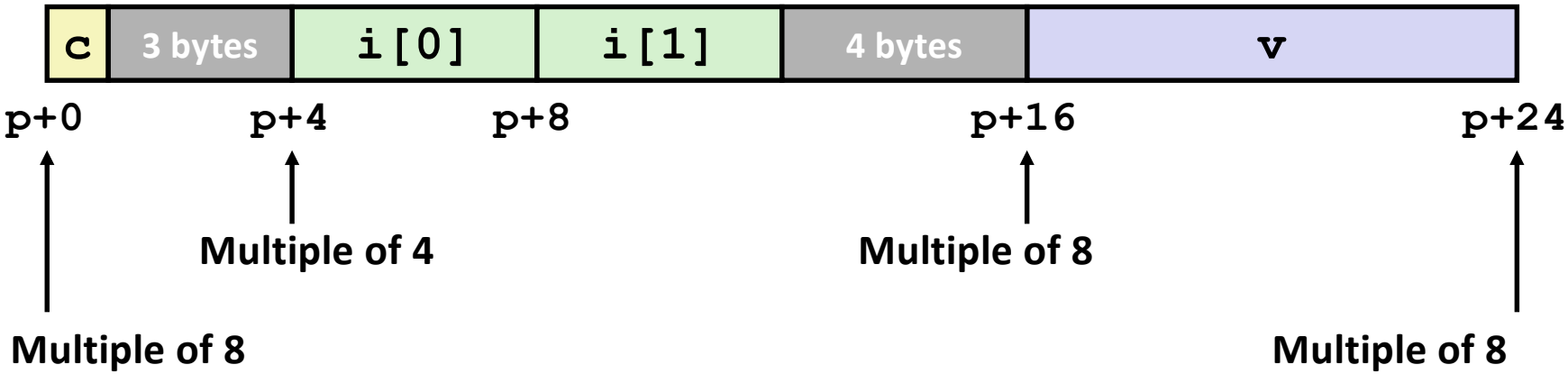
- Aligned Data
  - Primitive data type requires k bytes
  - Address must be multiple of k



Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - K = Largest alignment of any element in struct
  - Initial address & overall structure length must be multiples of K
- Example:
  - K = 8, due to **`double`** element

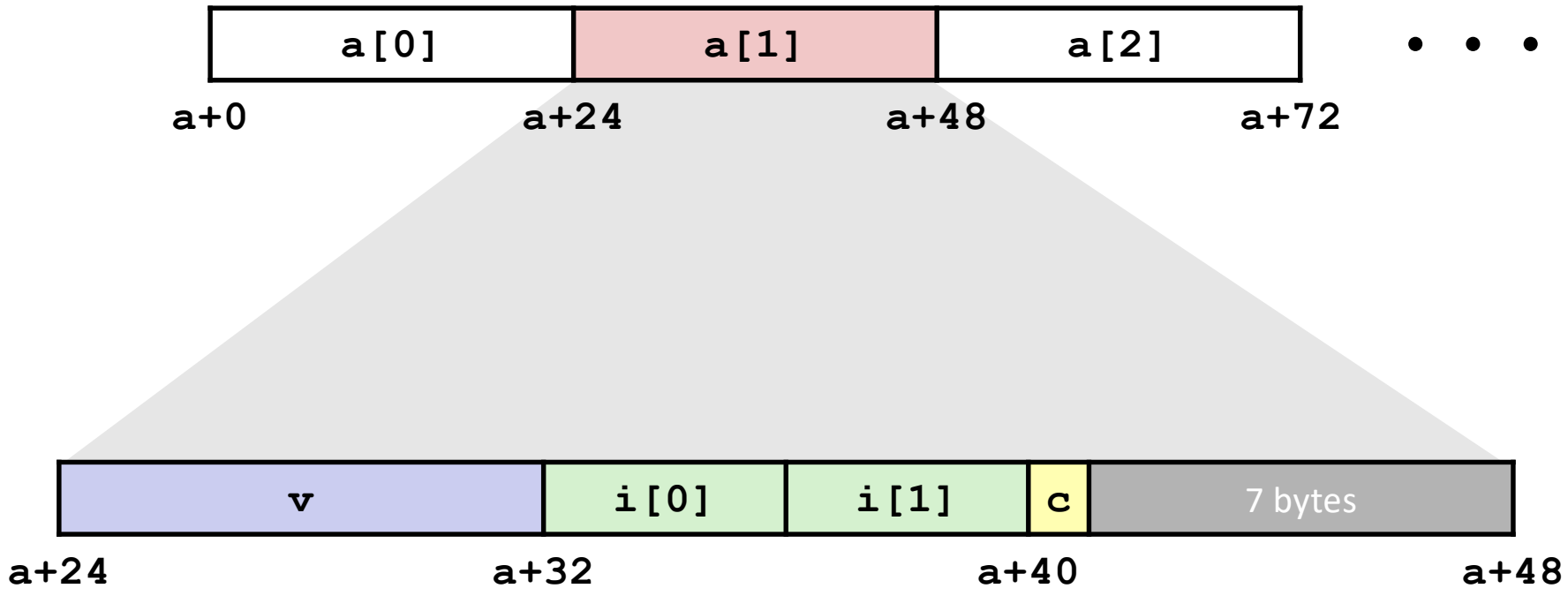
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Arrays of Structures

- Overall structure length multiple of K
  - K = Largest alignment of any element
- Satisfy alignment requirement for every element (little k)

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Arrays & Structures (Ch 3.8-3.9) Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment (little k and big K)

Today: Moving on to Ch 4

- Arrays (Ch 3.8)
  - Multi-dimensional (nested)
  - Multi-level
- Structures (Ch 3.9)
  - Allocation
  - Access
  - Alignment
- Intro to Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC

Ch 4 Processor Architecture Overview

- Background
  - Instruction sets (today)
  - Logic design
- Sequential Implementation
  - A simple, but not very fast processor design
- Pipelining
  - Several overlapping tasks running simultaneously
- Pipelined Implementation
  - Make it work in the presence of “hazards”

Coverage

- Our Approach
  - Work through designs for particular instruction set
    - Y86-64 – **a simplified (gentler) version of the Intel x86-64 ISA**
    - A little closer to RISC than x86-64, but still CISC
  - Work at "micro-architectural" level
    - Assemble basic hardware blocks into overall processor structure
      - Memories, functional units, etc.
    - Surround by control logic to make sure each instruction flows through properly
  - Use simple hardware description language to describe control logic
    - Can extend and modify
    - Test via simulation

An Aside: CISC vs. RISC

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

https://www.microcontrollertips.com/risc-vs-cisc-architectures-one-better/

CISC Instruction Sets

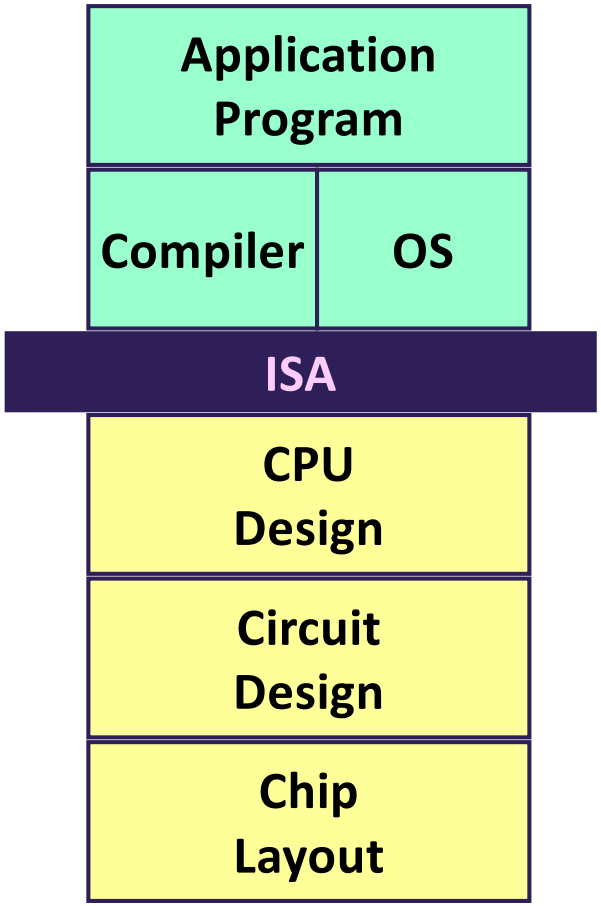
- Complex Instruction Set Computer
  - X86-64 is an example
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - `addq %rax, 12(%rbx,%rcx,8)`
    - Requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform “typical” programming tasks

RISC Instruction Sets

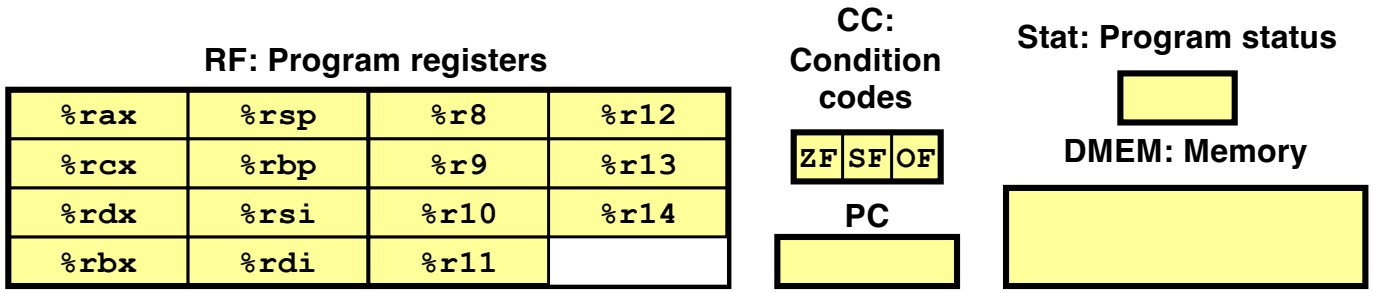
- Reduced Instruction Set Computer
  - Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
  - Examples: MIPS, ARM
- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
  - Similar to Y86-64 `mrmovq` and `rmmovq`
- No Condition codes
  - Test instructions return 0/1 in register

Ch 4.1 - Instruction Set Architecture

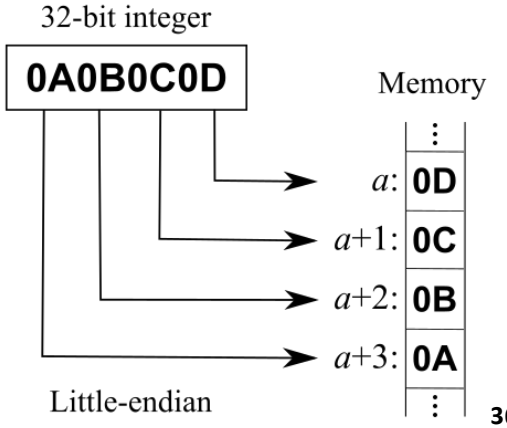
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Y86-64 Processor State



- Program Registers
  - 15 registers (omit `%r15`). Each 64 bits.
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF: Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order



Y86-64 Instruction Set #1