Structures

Structures

Complex data type defined by programmer

- Keeps together pertinent information of an object
- Contains simple data types or other complex data types
- Similar to a class in C++ or Java, but without methods

Example from graphics: a point has two coordinates

```
struct point {
        double x;
        double y;
};
```

- x and y are called members of struct point

Since a structure is a data type, you can declare variables:

```
struct point p1, p2;
```

What is the size of struct point? 16

Accessing structures

```
struct point {
        double x;
        double y;
};
struct point p1;
```

Use the “.” operator on structure objects to obtain members

```
p1.x = 10;
p1.y = 20;
```

Use the “->” operator on structure pointers to obtain members

```
struct point *pp=&p1;
double d;
```

- Long-form for accessing structures via pointer

```
d = (*pp).x;
```

- Short-form using “->” operator

```
d = pp->x;
```

Initializing structures like other variables:

```
struct point p1 = {320, 200};
```

- Equivalent to: `p1.x = 320; p1.y = 200;`

More structures

Structures can contain other structures as members:

```
struct rectangle {
        struct point pt1;
        struct point pt2;
};
```

What is the size of a struct rectangle? **32**

Structures can be arguments of functions

- Passed by value like most other data types
- Compare to arrays

More structures

Arrays within structures are passed by value!

```
#include <stdio.h>
struct two_arrays {
   char a[200];
   char b[200];
};
void func(struct two_arrays t, long i) {
    printf("t.a is at: %p     t.b is at: %p\n",&t.a,&t.b);
    if (i>0) func(t,i-1);
}
main() {
   struct two_arrays a;
   func(a,2);
}
% ./a.out
t.a is at: 0x7ffe77b2b8d0     t.b is at: 0x7ffe77b2b998
t.a is at: 0x7ffe77b2b720     t.b is at: 0x7ffe77b2b7e8
t.a is at: 0x7ffe77b2b570     t.b is at: 0x7ffe77b2b638
% objdump -d a.out
...
  40061c:     mov    $0x32,%eax
  400621:     mov    %rdx,%rdi
  400624:     mov    %rax,%rcx
  400627:     rep movsq %ds:(%rsi),%es:(%rdi)
  40062a:     mov    $0x2,%edi
  40062f:     callq  40059d <func>
```

More structures

Avoid copying via pointer passing...

```
#include <stdio.h>
struct two_arrays {
  char a[200];
  char b[200];
};
void func(struct two_arrays *t, int i) {
   printf("t->a is at: %p     t->b is at: %p\n",&t->a,&t->b);
   if (i>0) func(t,i-1);
}
main() {
  struct two_arrays a, *ap;
  ap = &a;
  func(ap,2);
}
% ./a.out
t.a is at: 0x7ffdea1f79d0     t.b is at: 0x7ffdea1f7a98
t.a is at: 0x7ffdea1f79d0     t.b is at: 0x7ffdea1f7a98
t.a is at: 0x7ffdea1f79d0     t.b is at: 0x7ffdea1f7a98
% objdump -d a.out
   …
  400619:     mov    $0x2,%esi
  40061e:     mov    %rsp,%rdi
  400621:     callq  4005bd <func>
```

Operations on structures

Legal operations

- Copy a structure (assignment equivalent to memcpy)
- Get its address
- Access its members

Illegal operations

- Compare content of structures in their entirety
- Must compare individual parts

Structure operator precedences

- "`.`" and "`->`" higher than other operators
- `*p.x` is the same as `*(p.x)`
- `++p->x` is the same as `++(p->x)`

C typedef

C allows us to declare new datatypes using "`typedef`" keyword

- **The thing being named is then a data type, rather than a variable**

```
typedef int Length;

Length sideA; // may be more intuitive than
int sideA;
```

Often used when working with structs

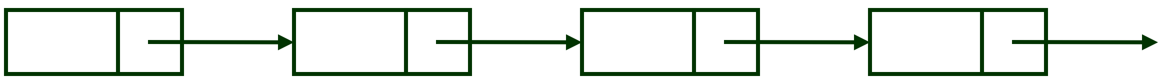
```
typedef struct tnode {
     char *word;
     int count;
     Treeptr left;
     Treeptr right;
} Treenode;

Treenode td;          // struct tnode td;
```

Self-referential structures

A structure can contain members that are pointers to the same struct (i.e. nodes in linked lists)

```
struct tnode {
    char *word;
    int count;
    struct tnode *next;
} p;
```



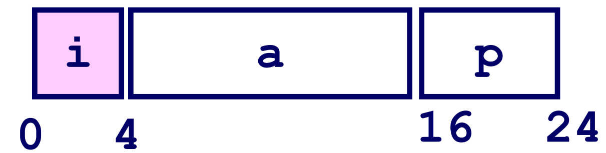
Structures in assembly

Concept

- Contiguously-allocated region of memory
- Members may be of different types
- Accessed statically, code generated at compile-time

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout



Accessing Structure Member

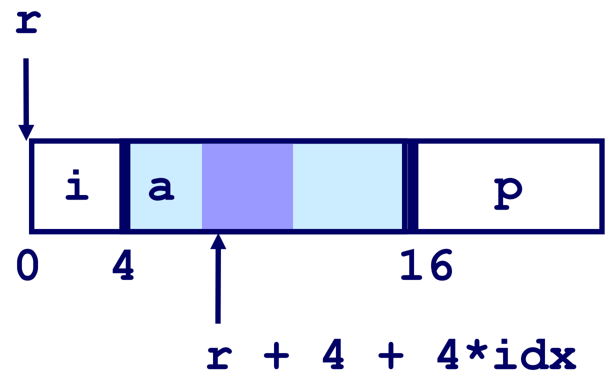
```
void
set_i(struct rec *r, int val)
{  r->i = val;}
```

Assembly

```
# %eax = val
# %rdx = r
movl %eax,(%rdx)     # Mem[r] = val
```

Example

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```




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

```
# %rcx = idx
# %rdx = r
leaq 0(,%rcx,4),%rax    # 4*idx
leaq 4(%rax,%rdx),%rax  # r+4*idx+4
```

Practice problem 3.39

```
struct prob {
     int *p;
     struct {
          int x;
          int y;
     } s;
     struct prob *next;
};
```

How many total bytes does the structure require?

24

What are the byte offsets of the following fields?

p	0
s.x	8
s.y	12
next	16

Consider the following C code:

```
void sp_init(struct prob *sp)
{
     sp->s.x  = __sp->s.y__;
     sp->p    = _&(sp->s.x)_;
     sp->next = ____sp_____;
}
```

Fill in the missing expressions

```
/* sp in %rdi     */
sp_init:
  movl   12(%rdi), %eax
  movl   %eax, 8(%rdi)
  leaq   8(%rdi), %rax
  movq   %rax, (%rdi)
  movq   %rdi, 16(%rdi)
  ret
```

Aligning structures

Data must be aligned at specific offsets in memory

Align so that data does not cross access boundaries and cache line boundaries

Why?

- Low-level memory access done in fixed sizes at fixed offsets
- Alignment allows items to be retrieved with one access
  - Storing a long at 0x00
    - » Single memory access to retrieve value
  - Storing a long at 0x04
    - » Two memory accesses to retrieve value
- Addressing code simplified
  - Scaled index addressing mode works better with aligned members

Compiler inserts gaps in structures to ensure correct alignment of fields

Alignment in x86-64

Aligned data required on some machines; advised on x86-64

If primitive data type has size *K* bytes, address must be multiple of *K*

- **char is 1 byte**
  - **Can be aligned arbitrarily**
- **short is 2 bytes**
  - **Member must be aligned on even addresses**
  - **Lowest bit of address must be 0**
- **int, float are 4 bytes**
  - **Member must be aligned to addresses divisible by 4**
  - **Lowest 2 bits of address must be 00**
- **long, double, pointers, … are 8 bytes**
  - **Member must be aligned to addresses divisible by 8**
  - **Lowest 3 bits of address must be 000**

Alignment with Structures

- Each member must satisfy its own alignment requirement
- Overall structure must also satisfy an alignment requirement "K"
  - K = Largest alignment of any element
  - Initial address must be multiple of K
  - Structure length must be multiple of K
    - For arrays of structures

Example

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

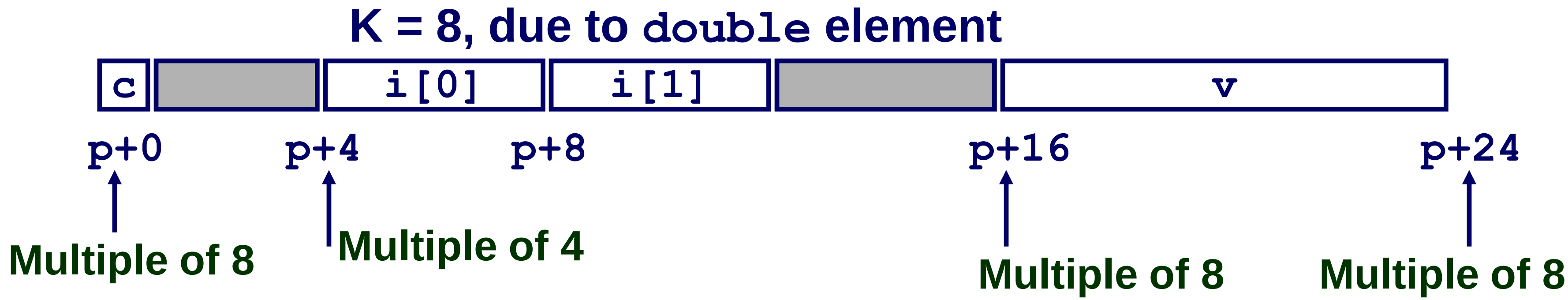
What is K for S1?

- K = 8, due to `double` element

What is the size of S1?

- 24 bytes

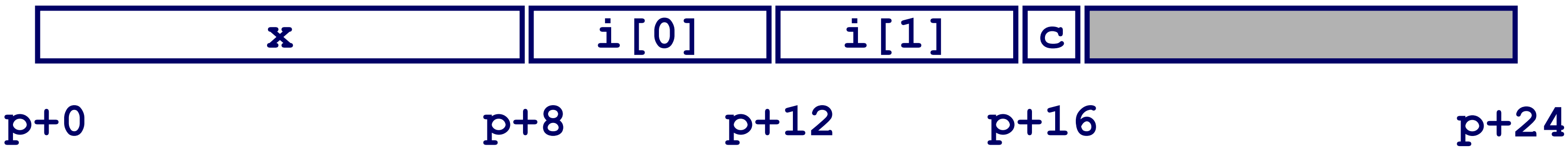
Draw S1



Examples

```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

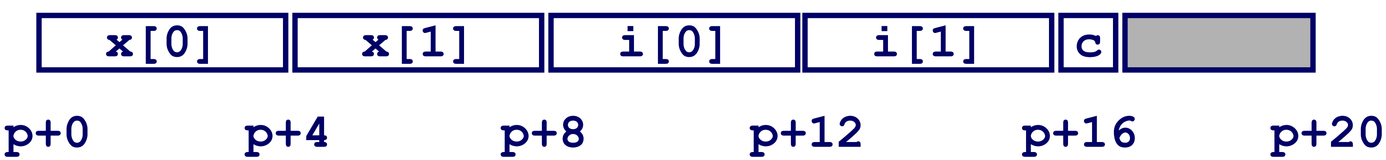
Draw the allocation for this structure




```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

Draw the allocation for this structure
What is K?

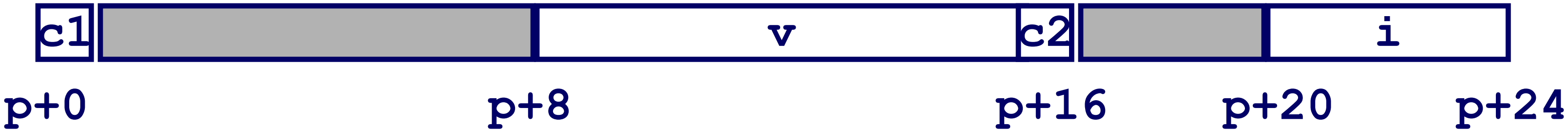
`p` must be multiple of 4



Reordering to reduce wasted space

```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

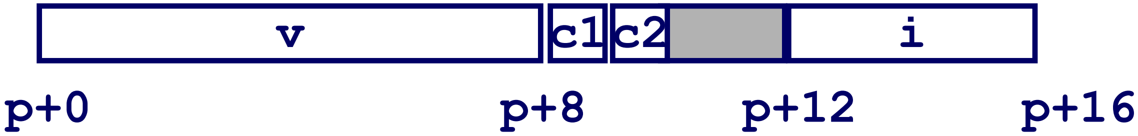
10 bytes wasted



Largest data first

```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

2 bytes wasted



Practice problem 3.44

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement

```
struct P1 {int i; char c; int j; char d;};
            0, 4, 8, 12  : 16 bytes : 4
struct P2 {int i; char c; char d; long j;};
             0, 4, 5, 8  : 16 bytes : 8
struct P3 {short w[3]; char c[3];};
                0, 6  : 10 bytes : 2
struct P4 {short w[5]; char *c[3];};
               0, 16  : 40 bytes : 8
struct P5 {struct P3 a[2]; struct P2 t}
                0, 24  : 40 bytes : 8
```

Practice problem 3.45

What are the byte offsets of each field?

```
0 8 16 24 28 32 40 48
```

```
struct {
  char *a;
  short b;
  double c;
  char d;
  float e;
  char f;
  long g;
  int h;
} rec;
```

What is the total size of the structure?

```
Must be multiple of K (8) => 56
```

Rearrange the structure to minimize space

```
a, c, g, e, h, b, d, f
```

Answer the two questions again

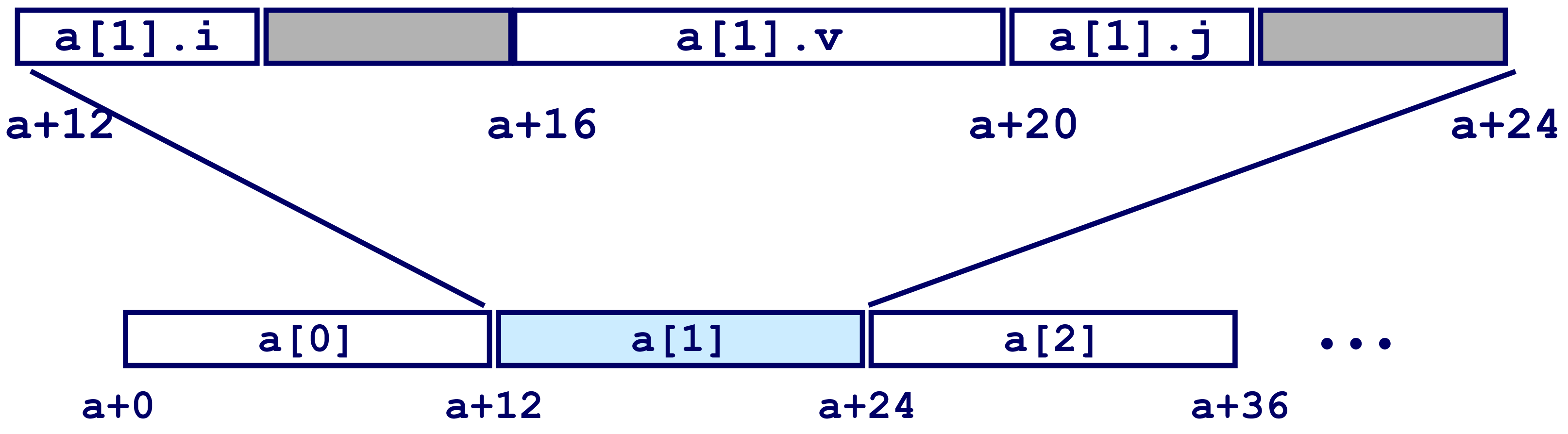
```
0 8 16 24 28 32 34 35
Multiple of 8 => 40
```

Arrays of Structures

Principle

- Allocated by repeating allocation for array type

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

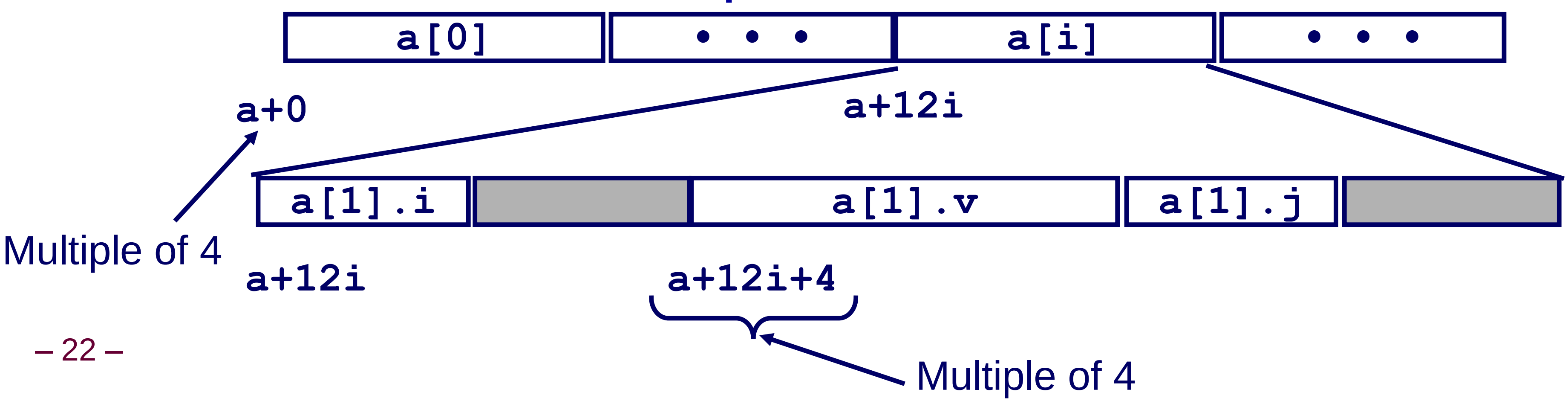


Satisfying Alignment within Arrays

Achieving Alignment

- Starting address must be K aligned
  - `a` must be multiple of 4
- Individual array elements must be K aligned
  - Structure padded with unused space to be 12 bytes (multiple of 4)
  - As a result, size of structure is a multiple of K
- Structure members must meet their own alignment requirement
  - `v`'s offset of 4 is a multiple of 4

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



Exercise

```
struct point {
      double x;
      double y
};

struct octagon {
  // An array can be an element of a structure ...
  struct point points[8];
} A[34];

struct octagon *r = A;
r += 8;
```

What is the size of a struct octagon? **16*8 = 128**

What is the difference between the address r and the address A?

128*8 = 1024

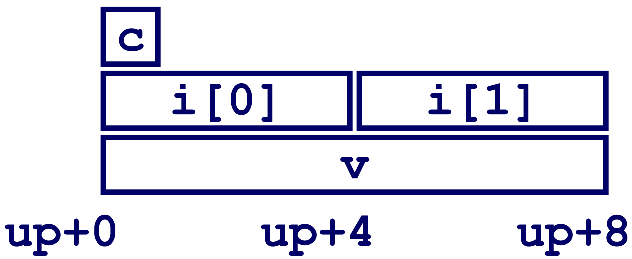
Unions

A *union* is a variable that may hold objects of different types and sizes

Sort of like a structure with all the members on top of each other.

The size of the union is the *maximum* of the size of the individual datatypes

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



Unions

```
union u_tag {
        int ival;
        float fval;
        char *sval;
} u;
```

```
u.ival = 14;
u.fval = 31.3;
u.sval = (char *) malloc(strlen(string)+1);
```

What's the size of u?
What exactly does u contain after these three lines of code?

Bit Fields

If you have multiple Boolean variables, you may save space by just making them bit fields

- **Used heavily in device drivers**
- **Simplifies code**

The Linux system call to open a file:

```
int fd = open("file", O_CREAT|O_WRONLY|O_TRUNC);
```

- **Second argument is an integer, using bit fields to specify how to open it.**
- **In this case, create a new file if it doesn't exist, for writing only, and truncate the file if it already exists.**

Implementing Bit Fields

You can use an integer and create bit fields using bitwise operators:

- 32 bit-field flags in a single integer

Via #defines

```
#define A 0x01
#define B 0x02
#define C 0x04
#define D 0x08
```

- Note that they are powers of two corresponding to bit positions

Via enum

- Constant declarations (i.e. like #define, but values are generated if not specified by programmer)

```
enum { A = 01, B = 02, C = 04, D = 08 };
```

Example

```
int flags;
flags |= A | B;
```

Bit field implementation via structs

Use bit width specification in combination with struct

Give names to 1-bit members

```
struct {
  unsigned int is_keyword : 1;
  unsigned int is_extern : 1 ;
  unsigned int is_static : 1;
};
```

Data structure with three members, each one bit wide

- **What is the size of the struct? 4 bytes**

Embedded Assembly

Assembly in C

Motivation

- Performance
- Access to special processor instructions or registers
  - (e.g. cycle counters)

Mechanisms specific to processor architecture (x86) *and* compiler (gcc)

- Must rewrite for other processors and compilers

Two forms

- Basic: `asm ( code-string );`
- Extended:

```
asm ( code-string
    [ : output-list
    [ : input-list
    [ : overwrite-list ] ] ] );
```

Basic Inline Assembly

Implement ok_smul(int x, int y, int *dest)

- Calculate x*y and put result at dest
- Return 1 if multiplication does not overflow and 0 otherwise

Use `setae` instruction to get condition code

- `setae D    (D <- ~CF)`

Strategy

- `%eax` stores return value
- Declare `result` and use it to store status code in `%eax`

```
int ok_smul1(int x, int y, int *dest)
{
    int result = 0;
    *dest = x*y;
    asm("setae %al");
    return result;
}
```

Basic Inline Assembly

Code does not work!

- Return `result` in `%eax`
- Want to set `result` using `setae` instruction beforehand
- Compiler does not know you want to link these two
  - (i.e. `int result` and `%eax`)

```
int ok_smul1(int x, int y, int *dest)
{
    int result = 0;
    *dest = x*y;
    asm("setae %al");
    return result;
}
```

http://thefengs.com/wuchang/courses/cs201/class/12/ok_smul1

Extended form asm

```
asm ( code-string
        [ : output-list
        [ : input-list
        [ : overwrite-list ] ] ] );
```

Allows you to bind registers to program values

Code-string

- Sequence of assembly separated by “;”

Output-list: get results from embedded assembly to C variables

- Tell assembler where to put result and what registers to use
  - “=r” (x) : dynamically assign a register for output variable “x”
  - Or use specific registers
    - “=a” (x) : use %eax for variable x
  - “+r” (x) : dynamically assign a register for both input and output variable “x”

Extended form asm

```
asm ( code-string
        [ : output-list
        [ : input-list
        [ : overwrite-list ] ] ] );
```

Input-list: pass values from C variables to embedded assembly

- **Tell assembler where to get operands and what registers to use**
  - **"r" (x) : dynamically assign a register to hold variable "x"**
  - **Or use specific registers**

    "a" (x) : read in variable x into %eax

Overwrite-list: to write to registers

- **Tell assembler what registers will be overwritten in embedded code**
- **Allows assembler to**
  - **Arrange to save data it had in those registers**
  - **Avoid using those registers**

Extended form asm

Code-string

- Assembly instructions
- Specific registers
  - %%<register>
- Input and output operands
  - %<digit>
  - Ordered by output list, then input list

Output list

- Assembler assigns a register to store result
- Compiler adds code to save register to memory

Overwrite list

- Compiler saves %ebx or avoids using %ebx in code

```
int ok_smul3(int x, int y, int *dest)
{
    int result;

    *dest = x*y;

    /* Insert following assembly
        setae %bl
        movzbl %bl,result   */

    asm("setae %%bl; movzbl %%bl,%0"
            : "=r" (result)
            :
            : "%ebx"
        );
    return result;
}
```


http://thefengs.com/wuchang/courses/cs201/class/12/ok_smul3

Extended form asm

Unsigned multiplication example

```
int ok_umul(unsigned x, unsigned y, unsigned *dest)
{
    int result;
    asm("movl %2,%%eax; mull %3; movl %%eax,%0;
            setae %%dl; movzbl %%dl,%1"
                : "=r" (*dest), "=r" (result)
                : "r" (x), "r" (y)
                : "%eax", "%edx"
    );
    return result;
}
```

```
/*  movl x, %eax
    mull y
    movl %eax, *dest
    setae %dl
    movzbl %dl, result     */
```

http://thefengs.com/wuchang/courses/cs201/class/12/ok_umul

Problem

What is the output of the following code?

```
#include <stdio.h>
int myasm(int x, int y) {
        int result;

        asm("movl %1,%%ebx; movl %2,%%ecx;
              sall %%cl,%%ebx; movl %%ebx,%0"
                : "=r" (result)
                : "r" (x), "r" (y)
                : "%ebx", "%ecx"
        );
        return result;
}
main() {
    printf("%d\n", myasm(2,3));
}
```


http://thefengs.com/wuchang/courses/cs201/class/12/example_asm

Extended form asm

Something more useful

- rdtsc = read timestamp counter (Pentium)
  - Reads 64-bit timestamp counter into %edx:%eax
  - Accessed via asm
  - Key code

```
unsigned int lo, hi;

asm("rdtsc": "=a" (lo), "=d" (hi) );
```

http://thefengs.com/wuchang/courses/cs201/class/12/rdtsc.c

Exam practice

Chapter 3 Problems (Part 2)

3.18	C from x86 conds
3.20, 3.21	C from x86 (conditionals)
3.23	Cross x86 to C (loops)
3.24	C from x86 (loops)
3.28	Fill in C for loop from x86
3.30, 3.31	Switch case reverse engineering
3.32	Following stack in function calls
3.33	Function call params
3.35	Function call reversing
3.36, 3.37	Array element sizing
3.38	Array/Matrix dimension reversing
3.40	Refactor C Matrix computation to pointers
3.41, 3.44, 3.45	structs in assembly
3.58	C from assembly
3.62, 3.63	Full switch reverse engineering
3.65	Matrix dimension reversing

ARM

ARM history

Acorn RISC Machine (Acorn Computers, UK)

- Design initiated 1983, first silicon 1985
- Licensing model allows for custom designs (contrast to x86)
  - Does not produce their own chips
  - Companies customize base CPU for their products
  - PA Semiconductor (fabless, SoC startup acquired by Apple for its A4 design that powers iPhone/iPad)
  - ARM estimated to make $0.11 on each chip (royalties + license)
- Runs 98% of all mobile phones (2005)
  - Per-watt performance currently better than x86
  - Less “legacy” instructions to implement

ARM architecture

RISC architecture

- 32-bit reduced instruction set machine inspired by Berkeley RISC (Patterson, 1980-1984)
- Fewer instructions
  - Complex instructions handled via multiple simpler ones
  - Results in a smaller execution unit
- Only loads/stores to and from memory
- Uniform-size instructions
  - Less decoding logic
  - 16-bit in Thumb mode to increase code density

ARM architecture

ALU features

- Conditional execution built into many instructions
  - Less branches
  - Less power lost to stalled pipelines
  - No need for branch prediction logic
- Operand bit-shifts supported in certain instructions
  - Built-in barrel shifter in ALU
  - Bit shifting plus ALU operation in one
- Support for 3 operand instructions
  - <R> = <Op1> OP <Op2>

ARM architecture

Control state features

- Shadow registers (pre v7)
  - Allows efficient interrupt processing (no need to save registers onto stack)
  - Akin to Intel hyperthreading
- Link register
  - Stores return address for leaf functions (no stack operation needed)

ARM architecture

Advanced features

- SIMD (NEON) to compete with x86 at high end
  - mp3, AES, SHA support
- Hardware virtualization
  - Hypervisor mode
- Jazelle DBX (Direct Bytecode eXecution)
  - Native execution of Java
- Security
  - No-execute page protection
    - Return2libc attacks still possible
  - TrustZone
    - Support for trusted execution via hardware-based access control and context management
    - e.g. isolate DRM processing

x86 vs ARM

Key architectural differences

- **CISC vs. RISC**
  - **Legacy instructions impact per-watt performance**
  - **Atom (stripped-down 80386 core)**
    - **Once a candidate for the iPad until Apple VP threatened to quit over the choice**
- **State pushed onto stack vs. swapped from shadow registers**
- **Bit shifting separate, explicit instructions vs. built-in shifts**
- **Memory locations usable as ALU operands vs. load/store only**
- **Mostly 2 operand instructions (<D> = <D> OP <S>) vs. 3-operand**

ARM vs. x86

Other differences

- Intel is the only producer of x86 chips and designs
  - No SoC customization (everyone gets same hardware)
  - Must wait for Intel to give you what you want
  - ARM allows Apple to differentiate itself
- Intel and ARM
  - XScale: Intel's version of ARM sold to Marvell in 2006
  - Speculation
    - Leakage current will eventually dominate power consumption (versus switching current)
    - Intel advantage on process to make RISC/CISC moot
    - Make process advantage bigger than custom design + RISC advantage (avoid wasting money on license)

Extra slides

Self-referential structures

Declared via typedef structs and pointers

- **What does this code do?**

```
typedef struct tnode *nptr;

typedef struct tnode {
    char *word;
    int count;
    nptr next;
} Node;

static nptr Head = NULL;  // The head of a list
…
nptr np;              // temporary variable

while (… something …){
    // Allocate a new node
    np = (nptr) malloc(sizeof(Node));

    // Do some kind of processing
    np->word = … something …;

    np->next  = Head;
    Head = np;
}
```

Arrays of structures

Pointers/arrays for structures just like other data types

- Can use Rarray[idx] interchangeably with *(Rarray+idx)
- Are arrays of structures passed by value or reference?

```
struct rectangle * ptinrect(struct point p, struct rectangle *r, int n) {
int i;
for(i = 0; i < n; i++) {
              if(p.x >= r->pt1.x && p.x < r->pt2.x
               && p.y >= r->pt1.y && p.y < r->pt2.y)
                    return r;
              r++;
          }
          return ((struct rectangle *)0);
}
struct rectangle * ptinrect(struct point p, struct rectangle *r, int n) {
        int i;
        for (i = 0; i < n; i++) {
                if (p.x >= r[i].pt1.x && p.x < r[i].pt2.x
                    && p.y >= r[i].pt1.y && p.y < r[i].pt2.y)
                    return(&r[i]);
`    }
        return((struct rectangle *) 0);
}

struct rectangle Rarray[N];
ptinrect(p,Rarray,N);
```

Exercise

Given these variables:

```
struct {
   unsigned int is_keyword : 1;
   unsigned int is_extern : 1 ;
   unsigned int is_static : 1;
}flags1;
unsigned int flags2;
```

Write an expression that is true if the field `is_static` is set, using the bit field notation on flags1, and also using bitwise operators on flags2.

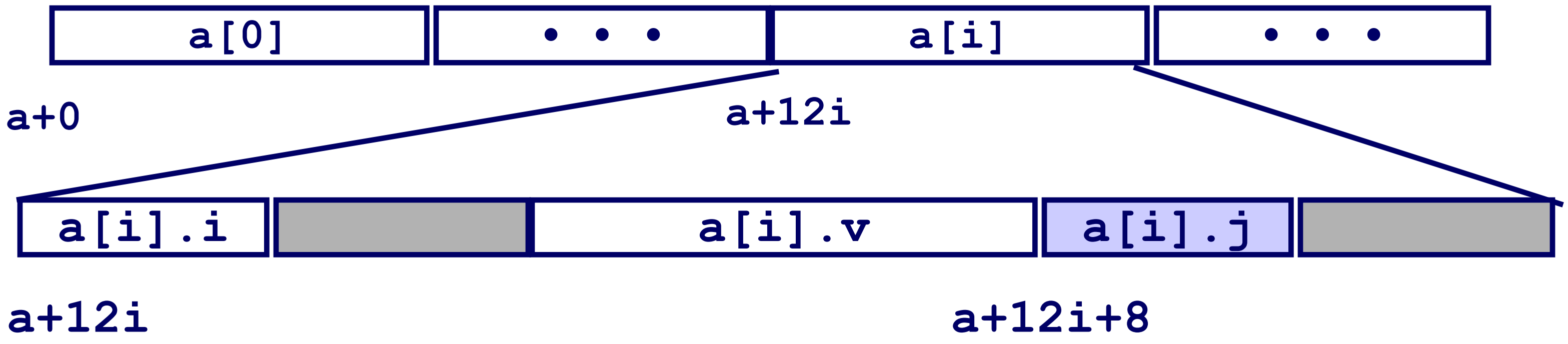
Accessing Elements within Array

- Compute offset from start of array
  - Compute 12**i* as 4*(*i*+2*i*)
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as a + 8

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

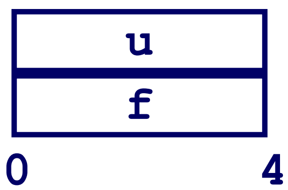
```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```



- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```