# Lecture 10
# Structured Data

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2022

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

# Administrivia

- Remember: drop deadline is **Friday**
  - Please come by office hours if you’re concerned and want to talk
  - Or email me and I can schedule a meeting whenever

  - If I’m worried at all, I reached out to you
    - So if you didn’t get an email, you’re doing fine

# Administrivia

- Bomb Lab due on Thursday (2/10)
  - Keep up the hard work on it!

  - Remember, it's tricky, but not trying to trick you
    - You can trust function names to roughly do what they say

- New assignments will be released on Thursday/Friday

# Today's Goals

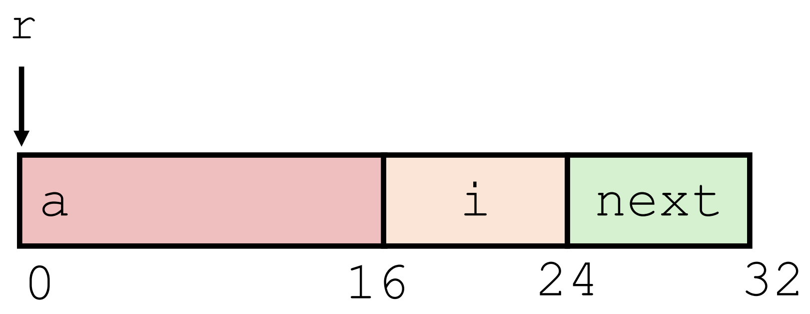
- Wrap up x86-64 assembly!
  - Discuss how structures are accessed
- Memory layout details
  - Explore details about how structure memory is aligned
  - Introduce unions in C
- Bonus: deep-dive into how processors work

# Outline

- **Structure Layout**
- Struct Padding and Alignment
- Unions

# Structure representation in C

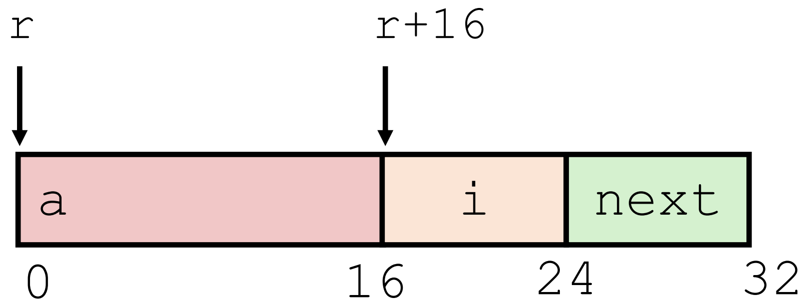
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```




- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering could yield a more compact representation
  - (We'll see how that could happen in a bit)
- Compiler determines overall size + positions of fields
  - Looking at memory, no way to tell it's a struct (like arrays); just bytes
  - It's all in how the code treats that region of memory!

# Structure access

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




- Accessing Structure Member
  - Pointer **r** indicates first byte of structure
  - Access member with offsets
  - Offset of each structure member determined at compile time
    - Another use for Displacement in memory addressing!

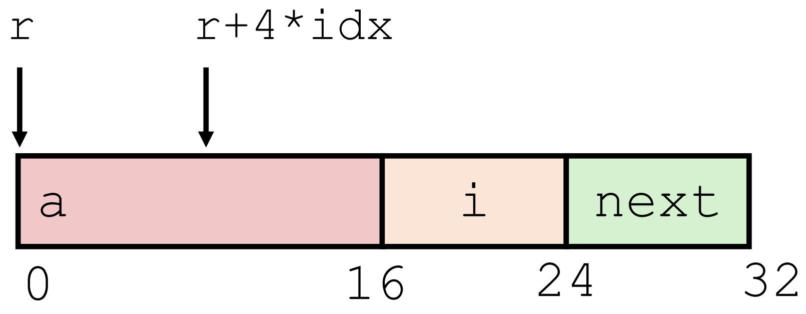
```
size_t get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi
  movq 16(%rdi), %rax
  ret
```

`r` is a pointer to a struct.
Dereference the ponter, then get the `i` field of the struct.

# Array within a struct

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




- Same as before; just need to also index in the array
  - Pointer **r** indicates first byte of structure
    - Offset of each structure member determined at compile time
    - Offset into array determined based on index and type
  - Compute as ***(structAddr + offset + K*index);**
    - Uses full addressing mode!

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

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

# Structure Access Practice 1

Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

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

```
movl %esi , 4(%rdi)
ret
```

# Structure Access Practice 2

Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val)
{
  r->a[1] = val;
}
```

```
movl  %esi , 12(%rdi)
ret
```

# Structure Access Practice 3

Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val,
      int index)
{
  r->a[index] = val;
}
```

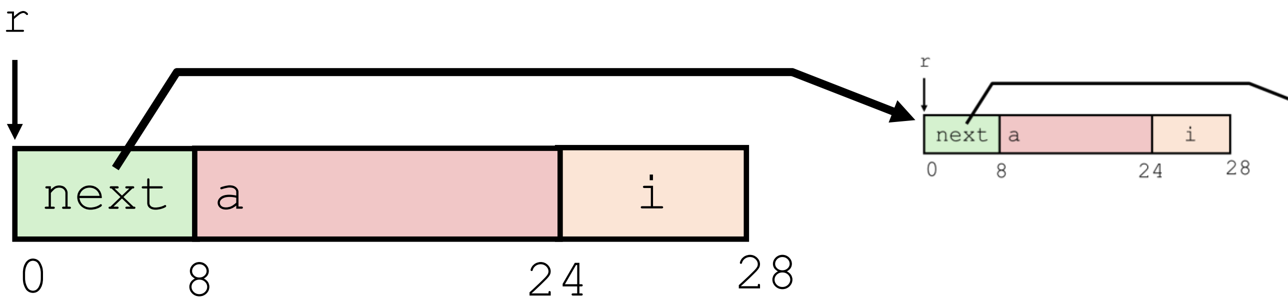
```
movl  %esi , 8(%rdi, %rdx, 4)
ret
```

# Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| **%rdi** | **r** |
| **%rsi** | **val** |

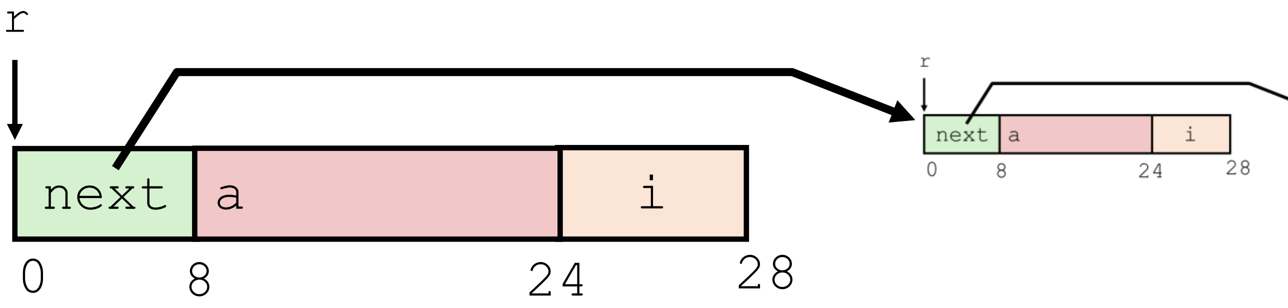
```
.L11:                               # loop:
  movslq  24(%rdi), %rax            #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)      #   M[r+8+4*i] = val
  movq    (%rdi), %rdi              #   r = M[r]
  testq   %rdi, %rdi                #   Test r
  jne     .L11                      #   if !=0 goto loop
```

# Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| **%rdi** | **r** |
| **%rsi** | **val** |

Load `i` →

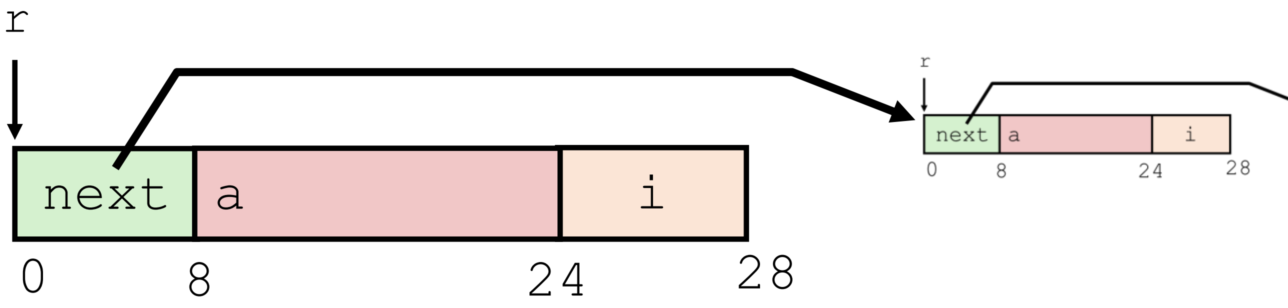
```
.L11:                          # loop:
  movslq  24(%rdi), %rax       #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4) #   M[r+8+4*i] = val
  movq    (%rdi), %rdi         #   r = M[r]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

# Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| **%rdi** | **r** |
| **%rsi** | **val** |

Write `val` into `r->a[i]` →

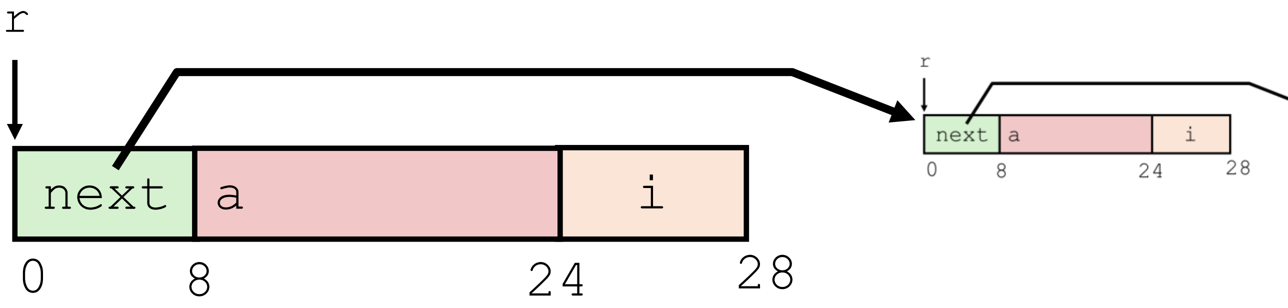
```
.L11:                            # loop:
  movslq  24(%rdi), %rax         #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)   #   M[r+8+4*i] = val
  movq    (%rdi), %rdi           #   r = M[r]
  testq   %rdi, %rdi             #   Test r
  jne     .L11                   #   if !=0 goto loop
```

# Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| **%rdi** | **r** |
| **%rsi** | **val** |

Move to next node in list →

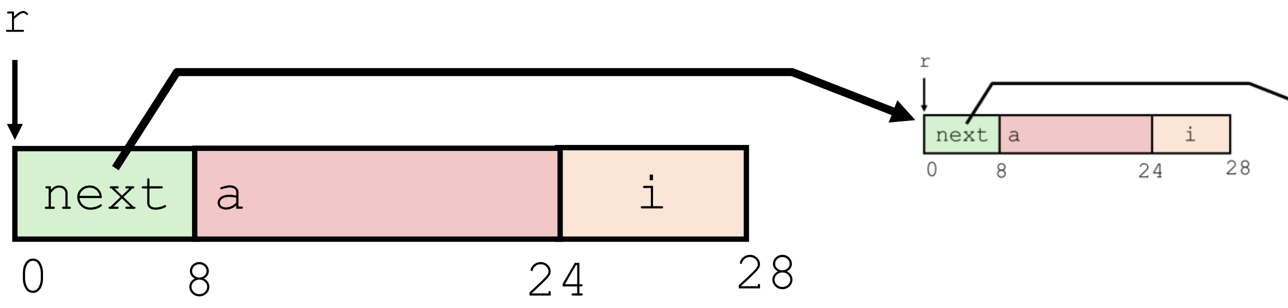
```
.L11:                          # loop:
  movslq  24(%rdi), %rax       #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4) #   M[r+8+4*i] = val
  movq    (%rdi), %rdi         #   r = M[r]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

# Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| %rdi | r |
| %rsi | val |

```
.L11:                           # loop:
  movslq  24(%rdi), %rax        #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)  #   M[r+8+4*i] = val
  movq    (%rdi), %rdi          #   r = M[r]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

NULL check → testq / jne

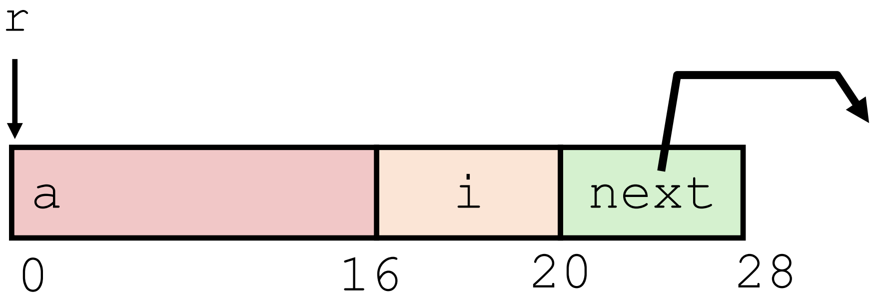
# Outline

- Structure Layout
- **Struct Padding and Alignment**
- Unions

# Problem: reordering can lead to different layouts

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

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| **%rdi** | **r** |
| **%rsi** | **val** |

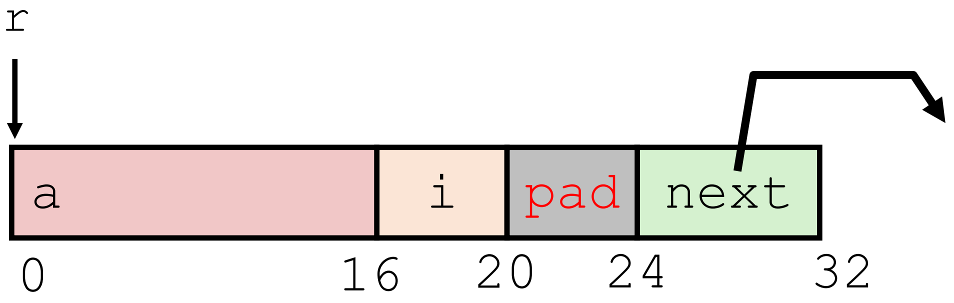
SOMETHING'S WRONG….

```
.L11:                              # loop:
  movslq  16(%rdi), %rax           #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)      #   M[r+4*i] = val
  movq    ??(%rdi), %rdi           #   r = M[r+??]
  testq   %rdi, %rdi               #   Test r
  jne     .L11                     #   if !=0 goto loop
```

# Padding is added to struct to preserve *alignment*

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

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




| Register | Value |
|---|---|
| %rdi | r |
| %rsi | val |

```
.L11:                            # loop:
  movslq  16(%rdi), %rax         #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)    #   M[r+4*i] = val
  movq    24(%rdi), %rdi         #   r = M[r+24]
  testq   %rdi, %rdi             #   Test r
  jne     .L11                   #   if !=0 goto loop
```

# Alignment

- Aligned data
  - Primitive data type requires K bytes
  - Address must typically be a multiple of K (e.g., 1,2,4 or 8)
    - an address that is a multiple of K is called “K-byte aligned”

- Required on some machines; recommended on x86-64
  - But not doing it will really slow down your program

- In our example, pointer needed 8-byte alignment
  - offset 24 ok, 20 was not

# The why and how of alignment

- Motivation for aligning data
  - Inefficient to load or store datum that spans quad word boundaries

  - Hardware is really good at loading, e.g., 8 bytes at address 16, or 24, or 32
    - If you want 8 bytes at address 12, may need two memory reads. Oops…

- Secondary motivations
  - Having one datum spanning 2 cache lines = two cache accesses per access
    - See upcoming lecture on caching
  - Virtual memory very tricky when a datum spans 2 pages
    - See upcoming lecture on virtual memory

- The compiler manages alignment
  - Inserts gaps in structure to ensure correct alignment of fields
  - Also occurs on the stack!

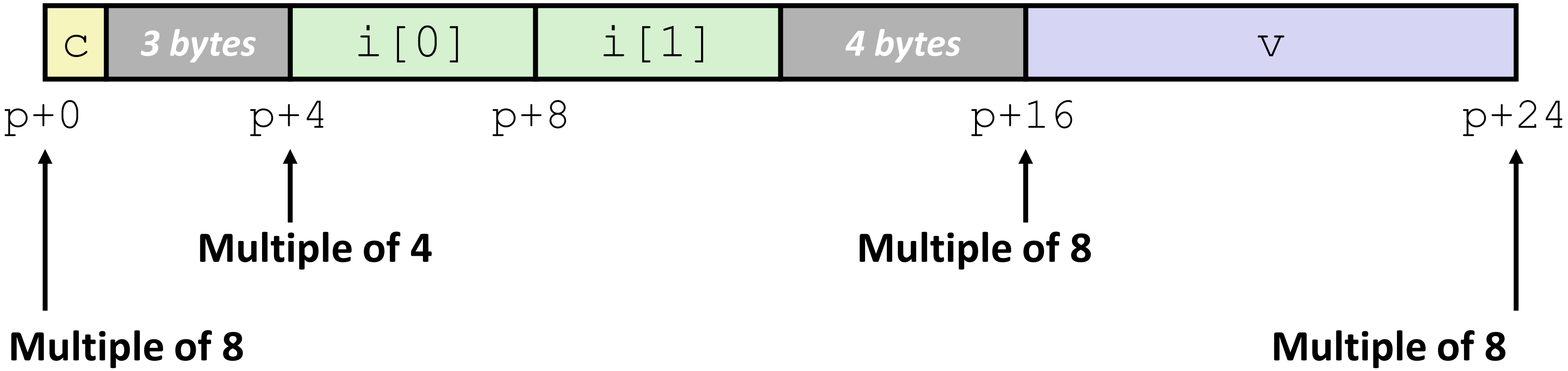
# Specific Cases of Alignment (x86-64, Linux)

- 1 byte: **`char`**
  - 1-byte aligned (no restrictions on address)
- 2 bytes: **`short`**
  - 2-byte aligned (lowest 1 bit of address must be `0`)
- 4 bytes: **`int`**, **`float`**
  - 4-byte aligned (lowest 2 bits of address must be `00`)
- 8 bytes: **`long`**, **`long long`**, **`double`**, **`char*`** (any pointer)
  - 8-byte aligned (lowest 3 bits of address must be `000`)
- 16 bytes: **`long double`**
  - 16-byte aligned (lowest 3 bits of address must be `0000`)
  - Max possible alignment requirement on x86-64

# Satisfying Alignment within Structures

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

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

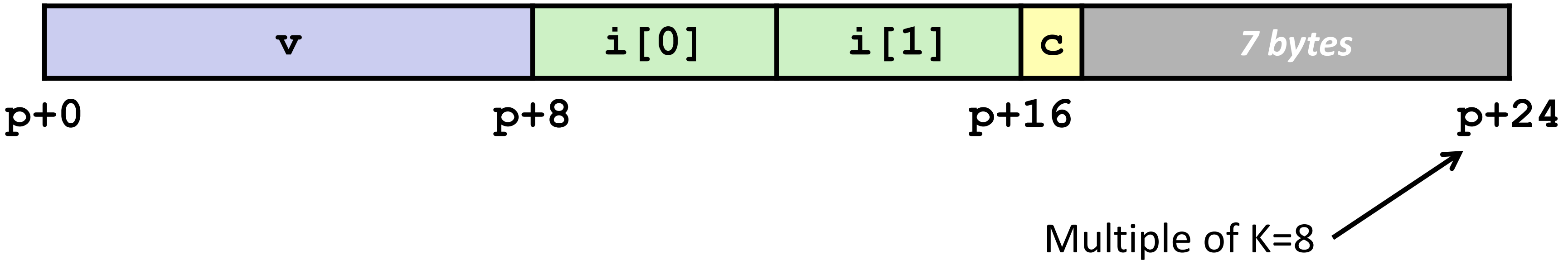


# Meeting Overall Alignment Requirement

- Entire struct must be a multiple of its largest element

- For largest alignment requirement K
- Overall structure must be multiple of K
  - Trailing padding

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

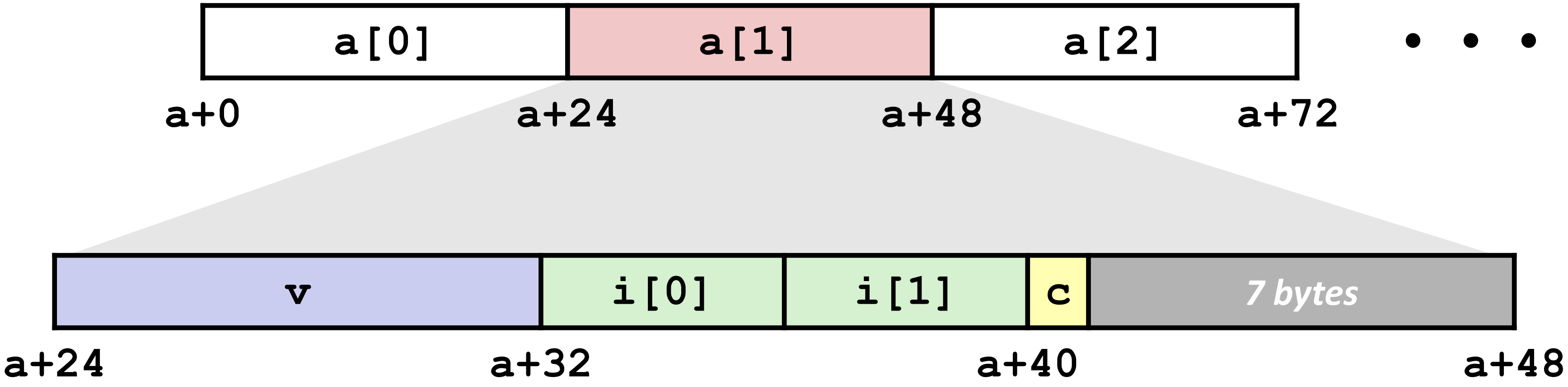


# Arrays of Structures

- Arrays are the reason for the overall length requirement
  - Each struct must start at a multiple of its largest member. This means the member is aligned

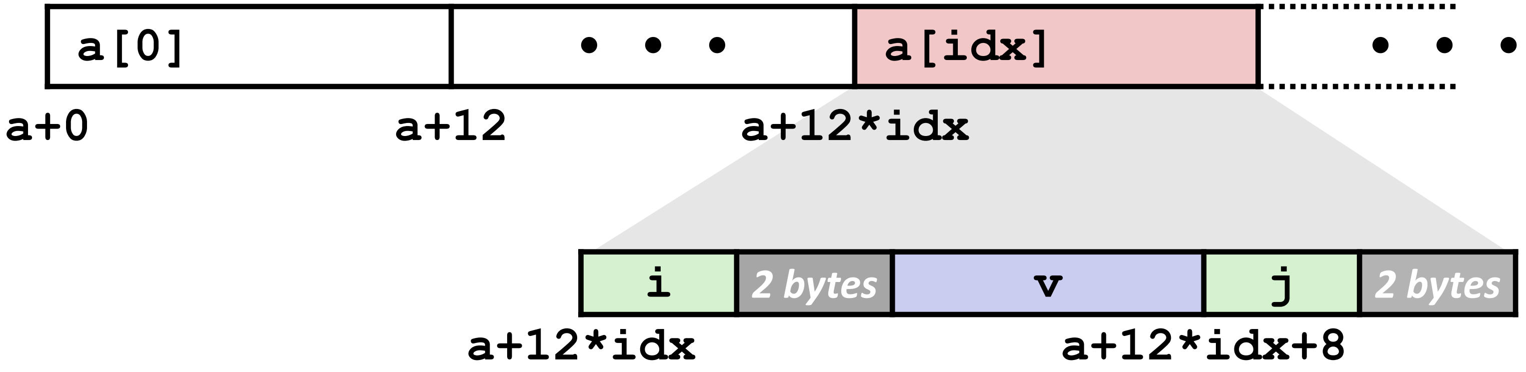
- The compiler adds trailing padding even without array declaration

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



# Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




- `sizeof(S3)=12`, including padding

- Compute array offset `12*idx`
- Element `j` is at offset 8 within structure

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

- Assembly contains displacement `a+8`
  - Compile-time constant resolved during linking

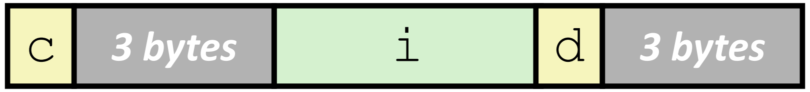
```
 # %rdi = idx
 leaq (%rdi,%rdi,2),%rax # 3*idx
 movzwl a+8(,%rax,4),%eax
```

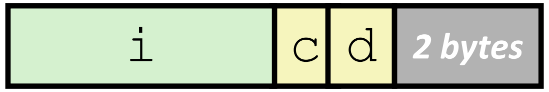
# Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```




- Effect: saved 4 bytes

- C compilers cannot do this automatically!
  - They have to preserve field ordering
  - Programmers must do it manually
  - Other languages aren’t bound to preserve ordering. Rust may reorder for you

# Break + Quiz

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

# Break + Quiz

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

2 bytes for **`a`**

(2 bytes for padding)

4 bytes for **`b`**

(no padding needed, 8-aligned)

24 bytes for **`c`**

(no padding needed, 1-aligned)

1 byte for **`d`**

(7 bytes padding after struct)

= 40 bytes total

Could have been 32 bytes if reordered

# Outline

- Structure Layout
- Struct Padding and Alignment
- **Unions**

# Unions

- Structs = combine multiple pieces of data into one
  - Think: "all of the above"
- Unions = choose between multiple different kinds of data
  - Think: "any of the above"
- Typically used in conjunction with a struct: *variants*
  - That tells us which branch of the union is used
  - E.g., **`which_kind`** of 0 to mean sandwich meal, 1 for pizza, etc.

```
typedef struct {
  char which_kind;
  char n_sides;
  char cost;
  MealKind_t mk;
} Meal_t;
```

```
typedef union {
  Sandwich_t s;
  Pizza_t p;
  Burrito_t b;
} MealKind_t;
```

```
typedef struct {
  int n_pieces_bread;
  char *toppings[2];
  float mayo_ounces;
} Sandwich_t;
```

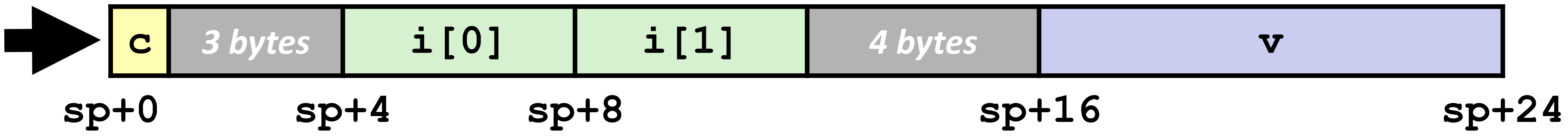
# Union allocation

- Principles
  - Overlay union elements
  - Allocate according to largest element (strictest)
  - Can only use one field at a time

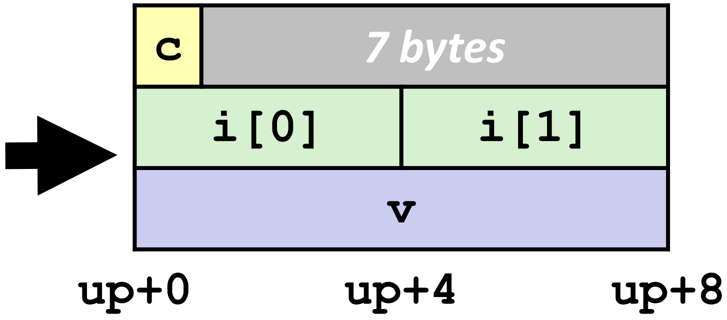
**Structs**: *All* of the above, together, one after the other.

**Unions**: *One* of the above, you pick the one you want.

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




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




- Union: same bits, different contexts
  - 8 bytes are allocated for the union
  - Can be interpreted as any member
  - Changing one member will change some bits of the others

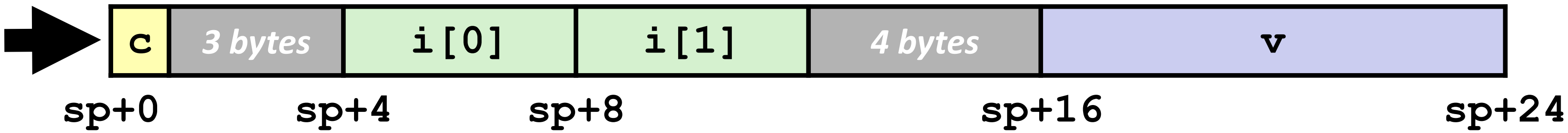
# Union allocation

- Principles
  - Overlay union elements
  - Allocate according to largest element (strictest)
  - Can only use one field at a time

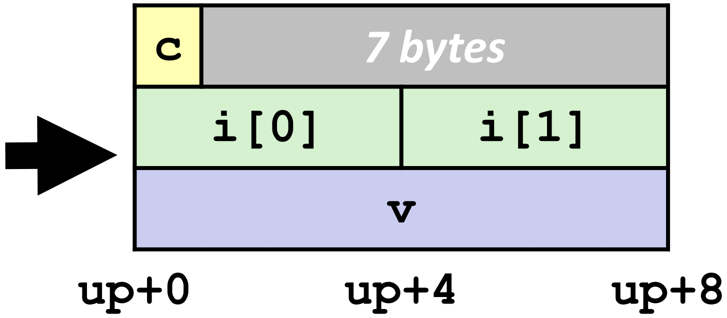
**Structs**: *All* of the above, together, one after the other.

**Unions**: *One* of the above, you pick the one you want.

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




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

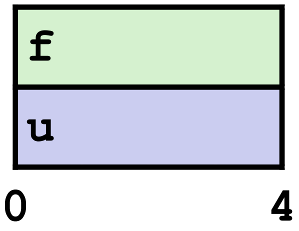



Quiz: If we had 3 `ints` in that array, how much space would the union take?

Answer: 16 bytes (8-byte aligned)

# Using union to access bit patterns

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




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

```
  # procedure with float arg
  # arg1 passed in %xmm0
  # movss = move single-precision
  movss %xmm0, -4(%rsp)
  movl -4(%rsp), %eax
  ret
```

- Store union using one type & access it with another one
- Get direct access to bit representation of float
- float2bit generates bit pattern from float
  - NOT the same as **`(unsigned) f`** !
  - Doesn't convert value to unsigned
  - Keeps the same bits but interprets them differently
- Assembly doesn't have type info
  - Just moves the bytes

# Access to Bit Pattern Non-Solution

```
unsigned float2bit(float f)
{
  unsigned *p;
  p = (unsigned *) &f;
  return *p;
}
```

Undefined behavior in C.

Don't do that.

# Byte ordering revisited

- Idea
  - Words/long words/quad words stored in memory as 2/4/8 consecutive bytes
  - At which byte address in memory is the most (least) significant byte stored?
  - Can cause problems when exchanging binary data between machines

- Little Endian
  - Least significant byte has lowest address
  - Intel x86(-64), ARM Android and IOS

- Big Endian
  - Most significant byte has lowest address
  - Sun/Sparc, Networks

- Have to worry about it when working with unions!

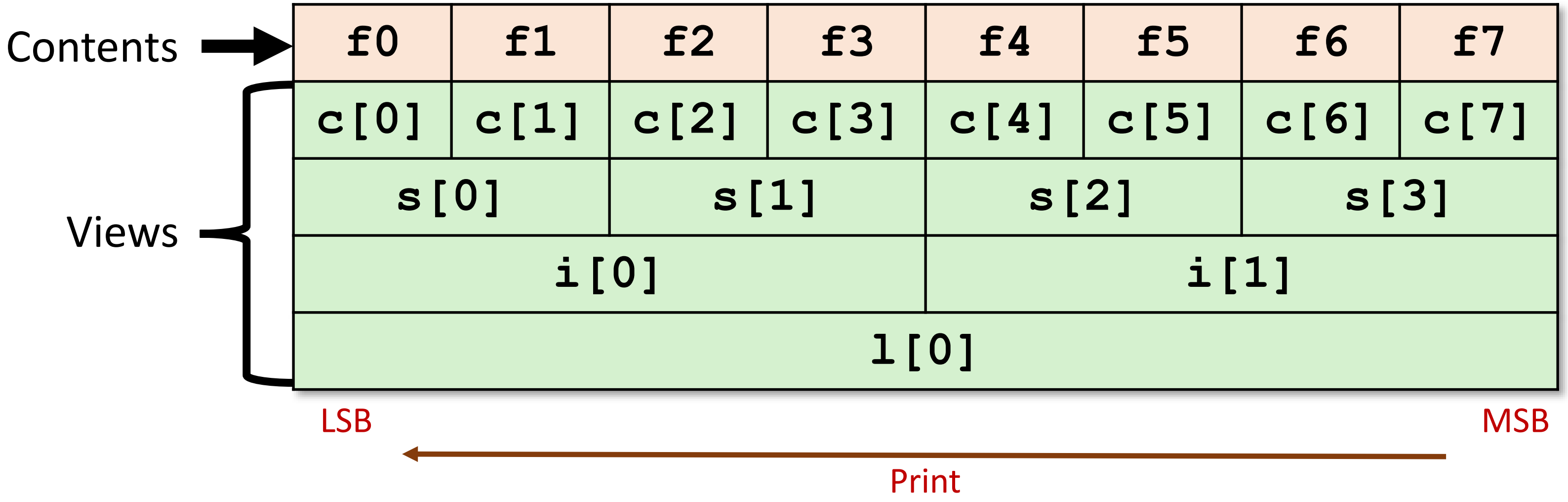
# Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

```
for (int j = 0; j < 8; j++) {
    dw.c[j] = 0xf0 + j;
}

printf("Chars 0-7 ==  [0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x, 0x%x, 0x%x, 0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x, 0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

# Byte ordering on Little Endian

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

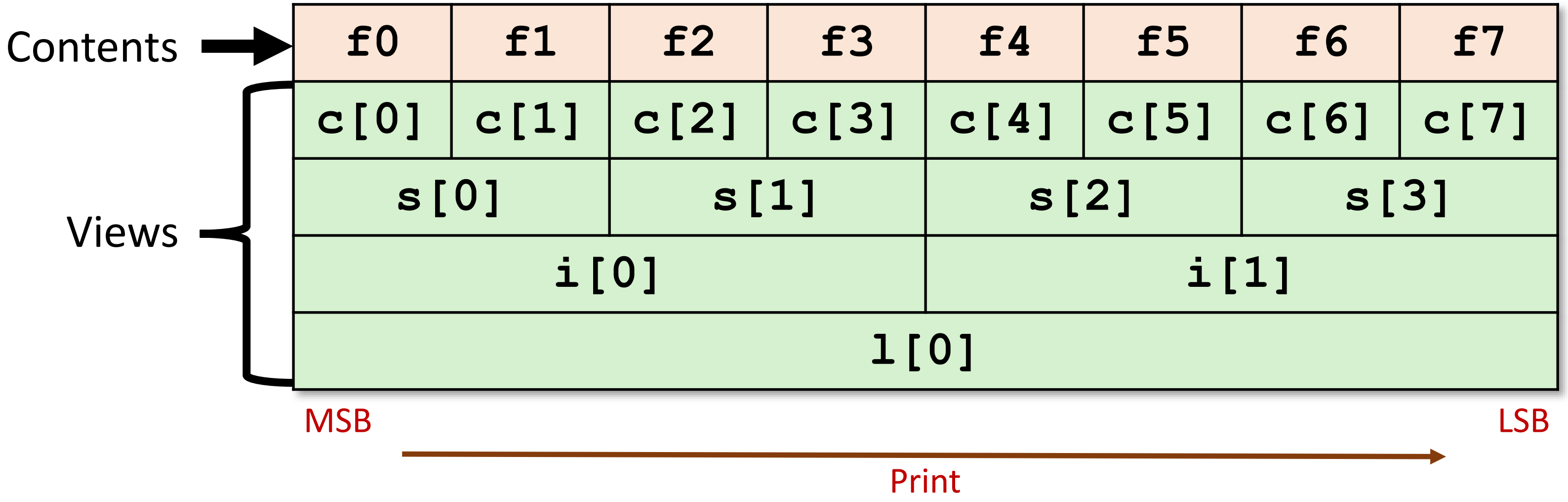



## Output:

```
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts     0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints       0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

# Byte ordering on Big Endian

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```




## Output:

```
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts     0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints       0-1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long       0   == [0xf0f1f2f3f4f5f6f7]
```

# Break + Thinking

- We’ve covered everything we need to from assembly

- Do we know enough to “compile” C++ in x86-64?
  - Yes!

  - Classes are structs
    - Likely with extra members to keep track of things
    - And function pointers as members

  - References are just pointers that the compiler handles for you

# Lecture BONUS Assembly to Transistors

CS213 – Intro to Computer Systems

Branden Ghena

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

# Assembly into machine code

```
        test:
            48 8d 04 7e
4011da      lea     (%rsi,%rdi,2),%rax
            48 8d 04 10
4011de      lea     (%rax,%rdx,1),%rax
            48 29 f7
4011e2      sub     %rsi,%rdi
            48 01 f8
4011e5      add     %rdi,%rax
            48 8d 84 08 13 02 00 00
4011e8      lea     0x213(%rax,%rcx,1),%rax
            c3
4011f0      ret
```

- Machine code are the numerical versions of each instruction
- Number breaks down into parts
  - Operation
  - Source
  - Destination
- Immediates are stored in the instruction encoding

# Machine code ideas

- Example:
  - ADD $0x4351FF23, %rax

  - ADD with destination %rax translates into 0x05
  - Immediate is appended on to that

  - Machine code: 0x0523FF5143

- Number of bytes for each instruction is variable
  - 1-15 bytes depending on instruction and operands

- Translation in complicated
  - We’re not going to do it by hand, although Attack Lab will touch it a bit

# Representing instructions as numbers

- Why represent instructions as numbers?

1. Everything in memory is “just a number”
   - And instructions go in memory
2. Hardware can “decode” number to figure out what to do
   - Break number apart into bits (just like floating point)
   - Some bits pick operation
   - Some bits pick register or specify immediate

# Computer Processor (in five easy steps)

1. Reads instruction from memory

2. Decodes it into an Operation plus Configurations
   - Immediates, Registers, Memory, etc.

3. Reads from source (based on configuration)

4. Executes that operation

5. Writes to destination (based on configuration)

# These steps are relatively easy (we'll skip them)

1. Reads instruction from memory

3. Reads from source (based on configuration)

5. Writes to destination (based on configuration)

# This is extremely complicated for x86-64 (skip it too)

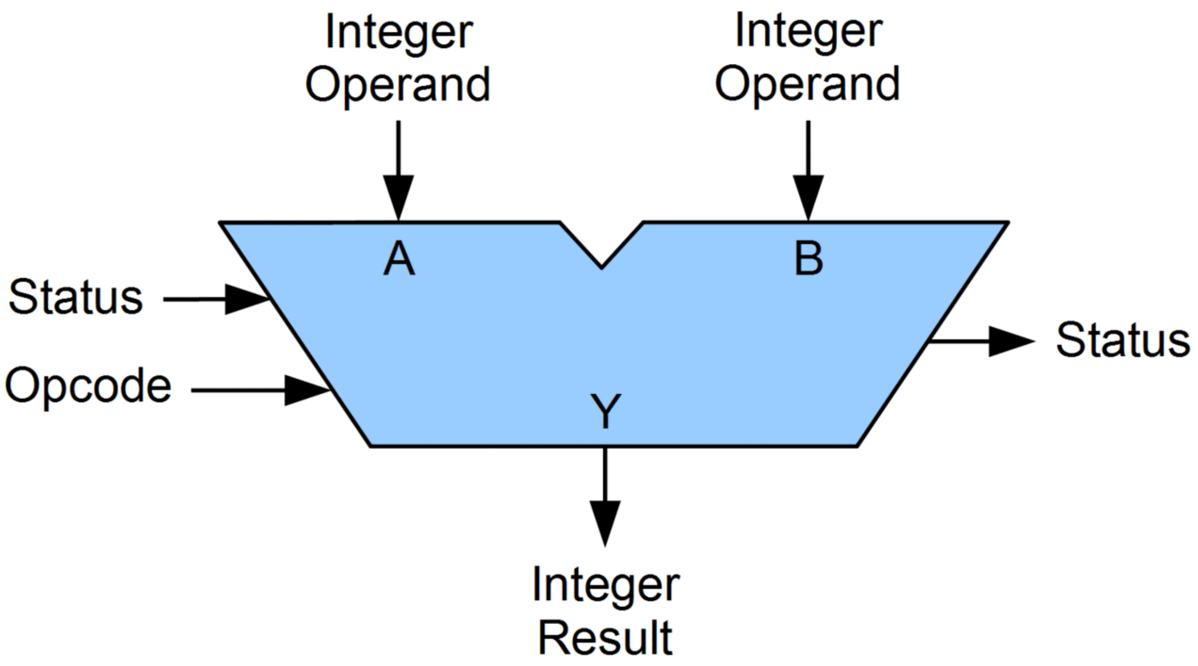
2. Decodes it into an Operation plus Configurations
    - Immediates, Registers, Memory, etc.

# We can talk about what execution means though!

4. Executes that operation

# Arithmetic Logic Unit (ALU)

- Piece of hardware
- Takes in two operands
  - Source and Destination *values*
- Takes in an Opcode
  - Which operation to run
- Performs operation and outputs result



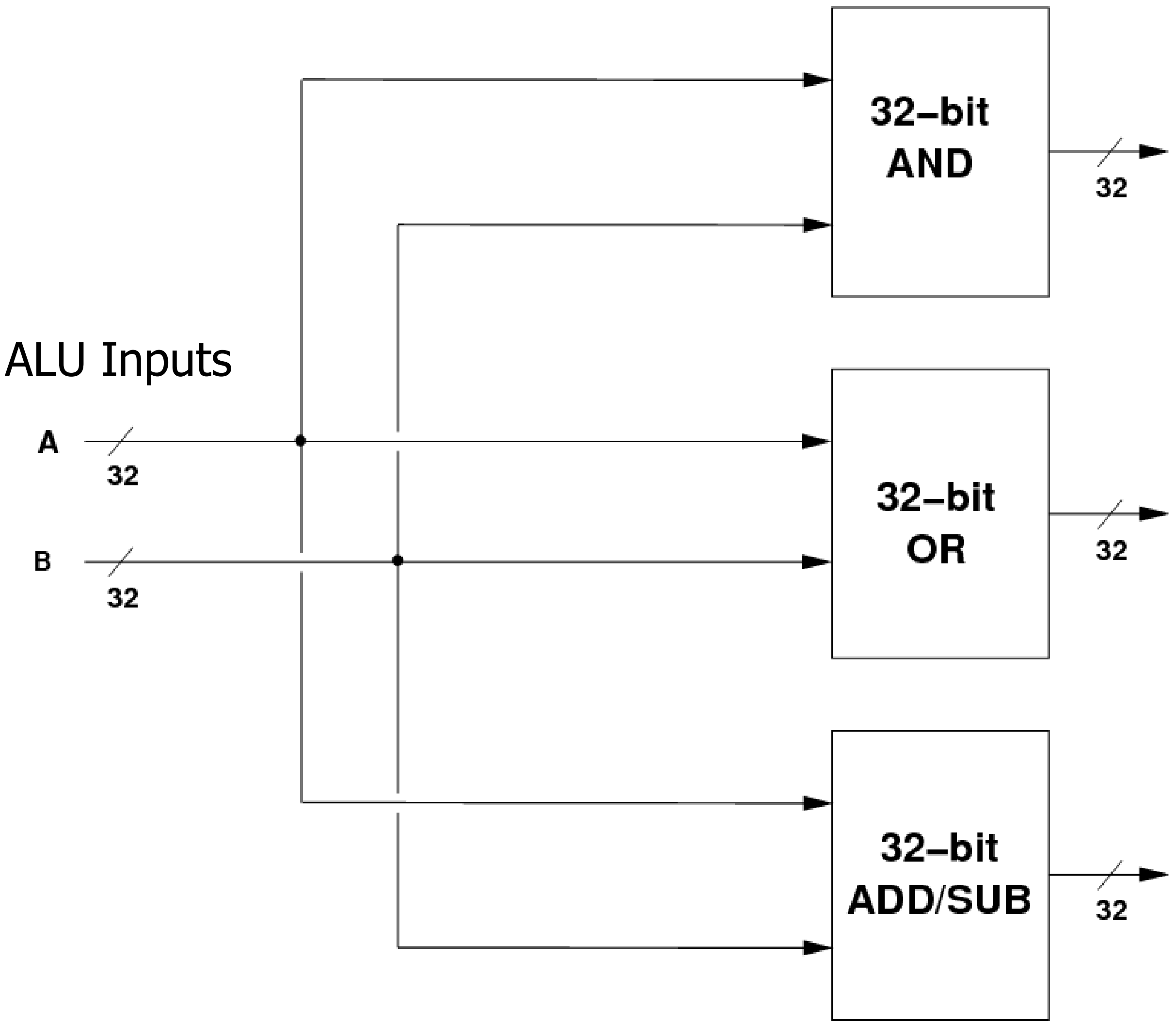
# What can an ALU do?

- All the basic arithmetic operations
  - Add
  - Subtract
  - Bitwise And
  - Bitwise Or
  - Bitwise Xor
  - Arithmetic Shift Right
  - Logical Shift Right
  - Logical Shift Left

- Complex operations are separate hardware
  - Multiply, Divide, Anything floating point

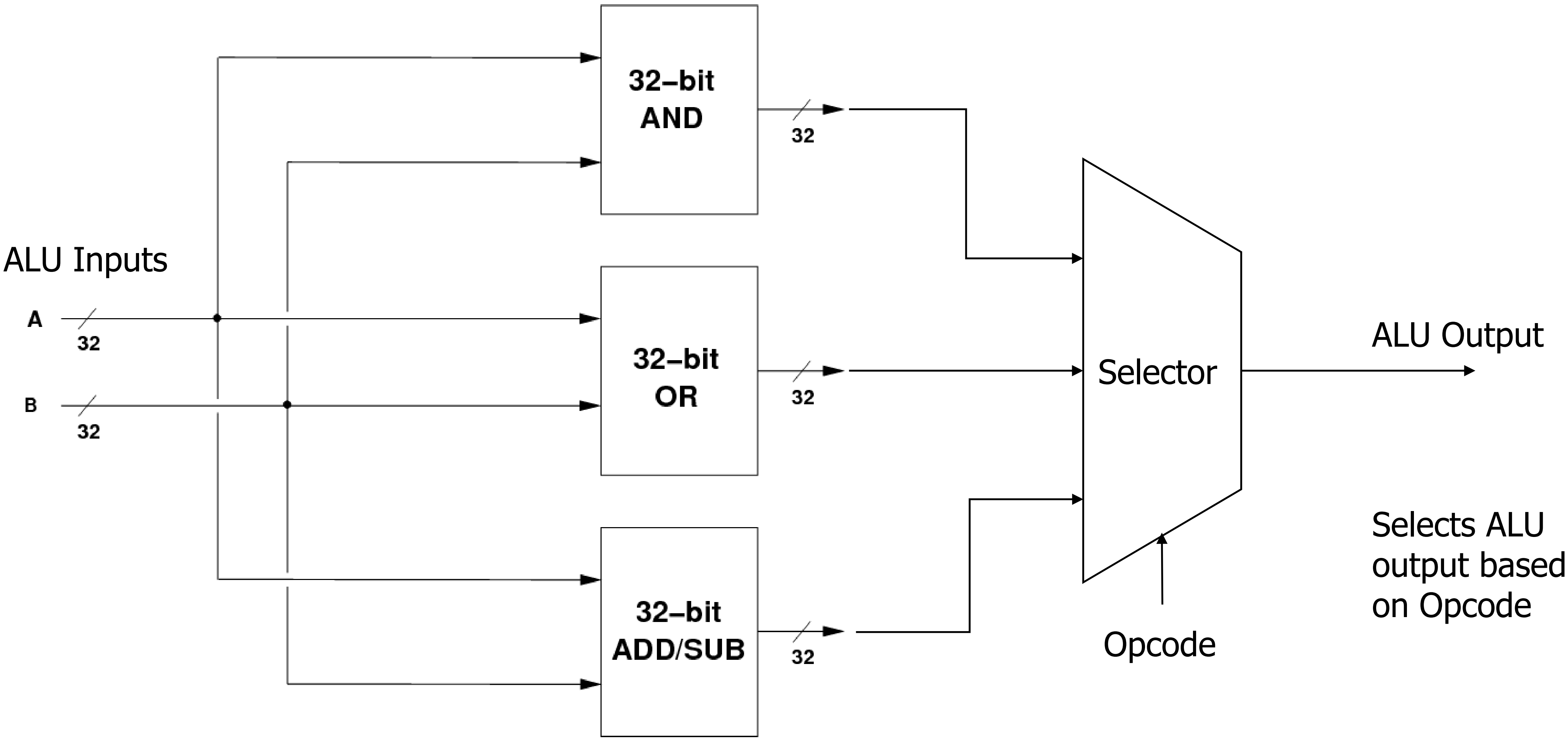
Let’s zoom in

# Inside an ALU




- Input values go into separate hardware blocks for each operation
- Every operation occurs in parallel, simultaneously
  - We are in hardware so this is essentially free

# Inside an ALU – selecting the correct output

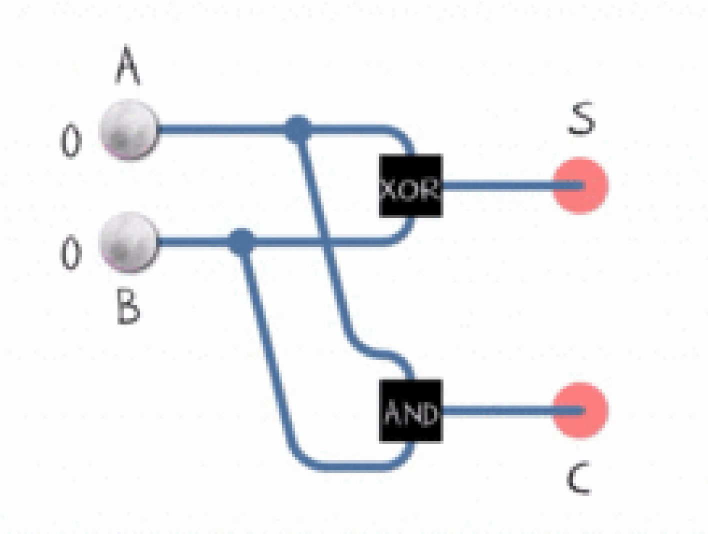


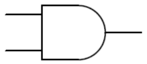

Selects ALU output based on Opcode

Let’s zoom in

# How is an ALU made?

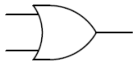
- All of those arithmetic operations can be broken down into a series of 1-bit Boolean operations
  - Add is XOR for result + AND for carry
  - Subtract is Flip bits (NOT), Add one (XOR + AND), then Add (XOR + AND)



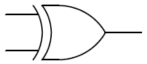
AND

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



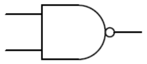
OR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



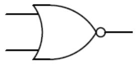
XOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



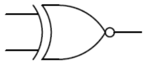
NAND

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



NOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |



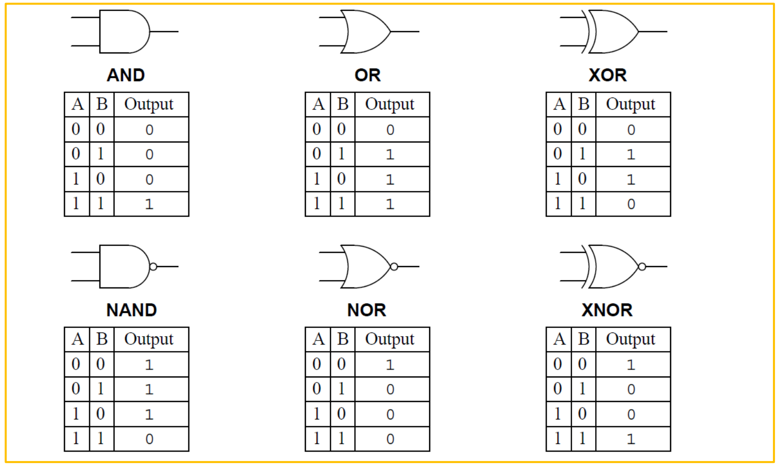
XNOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

  - And/Or/Xor are just their respective operations
  - Shifts are just move the bits around (simple in hardware, just move wires)

# 32-bit OR operation

- Perform OR operation on each individual bit
  - Pictured is a series of 1-bit OR gates




AND

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NAND

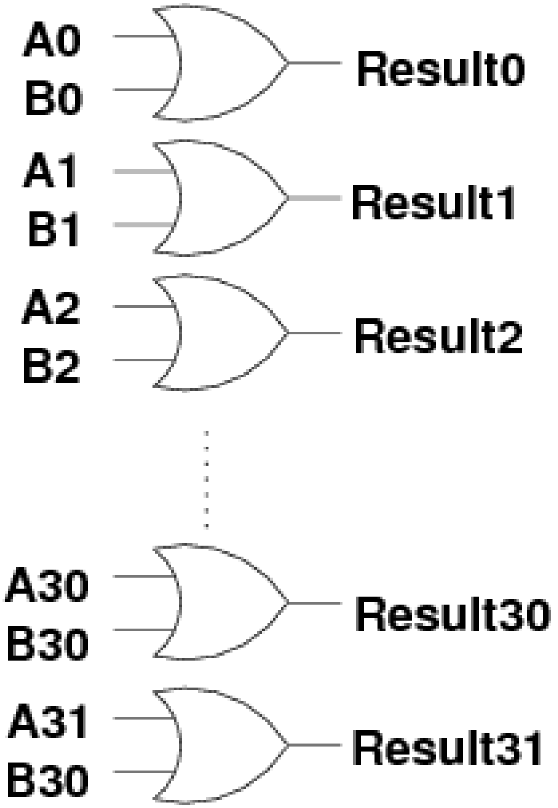
| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

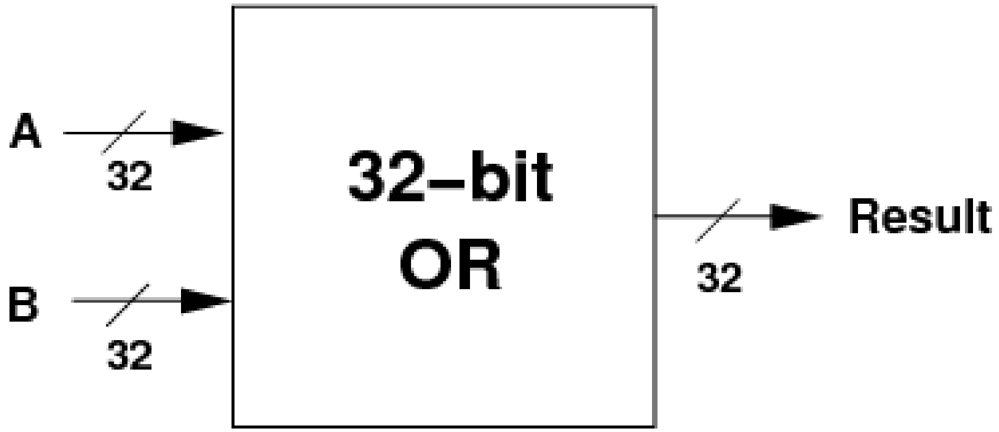
NOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

XNOR

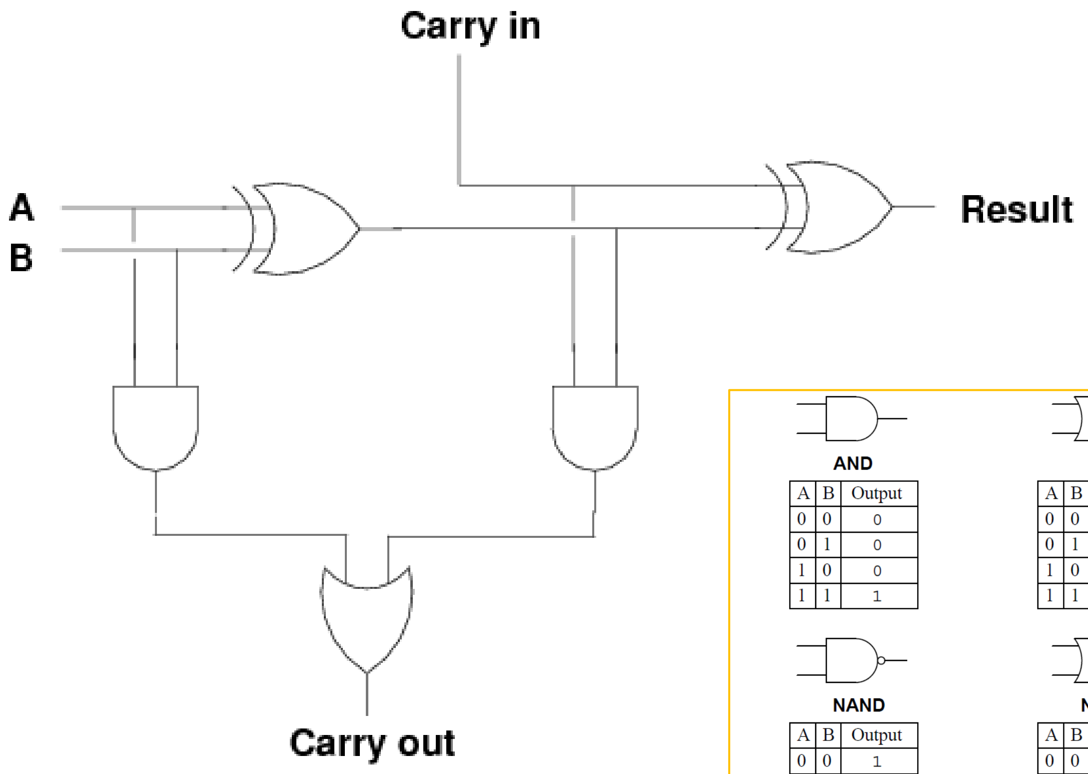
| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

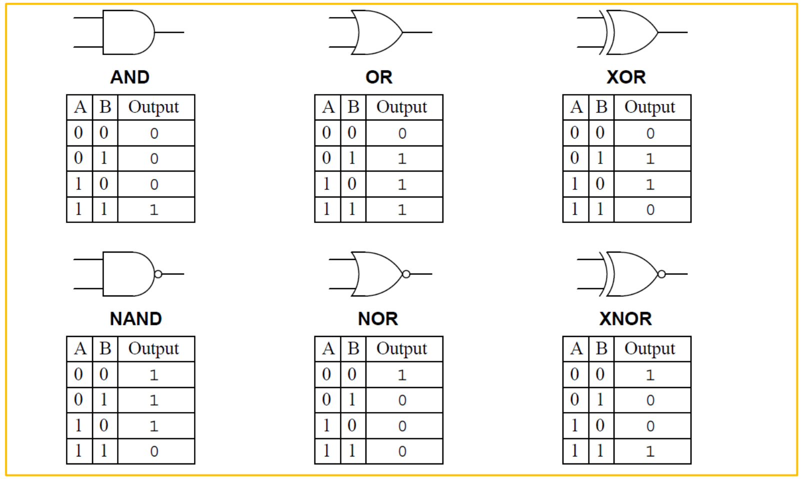


# 32-bit ADD operation

- Below is the 1-bit version with carry-in/out
  - Two 1-bit AND, two 1-bit XOR, one 1-bit OR
  - Repeat 32 times, connecting carries together




AND

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NAND

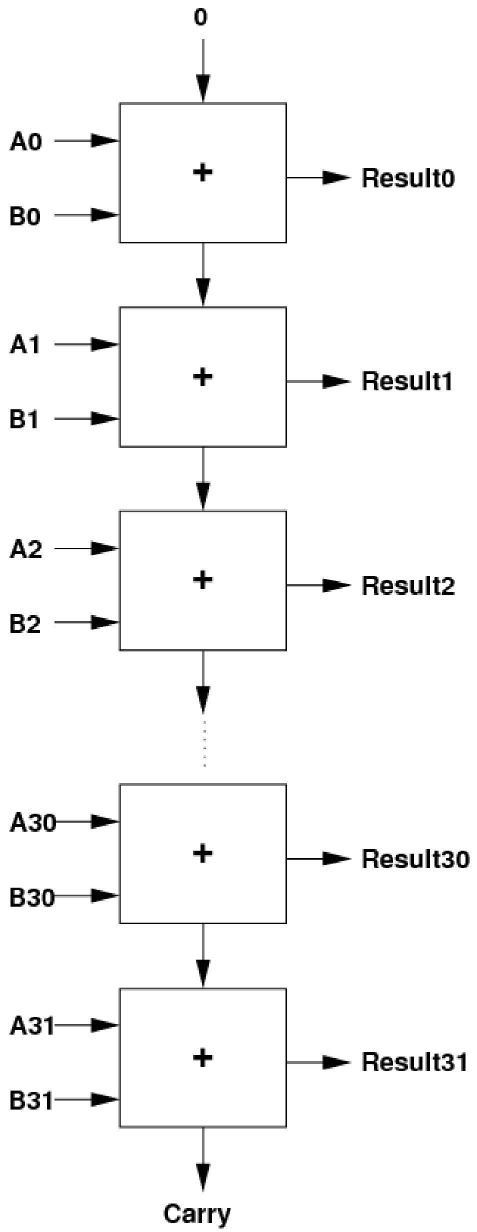
| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NOR

| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

XNOR

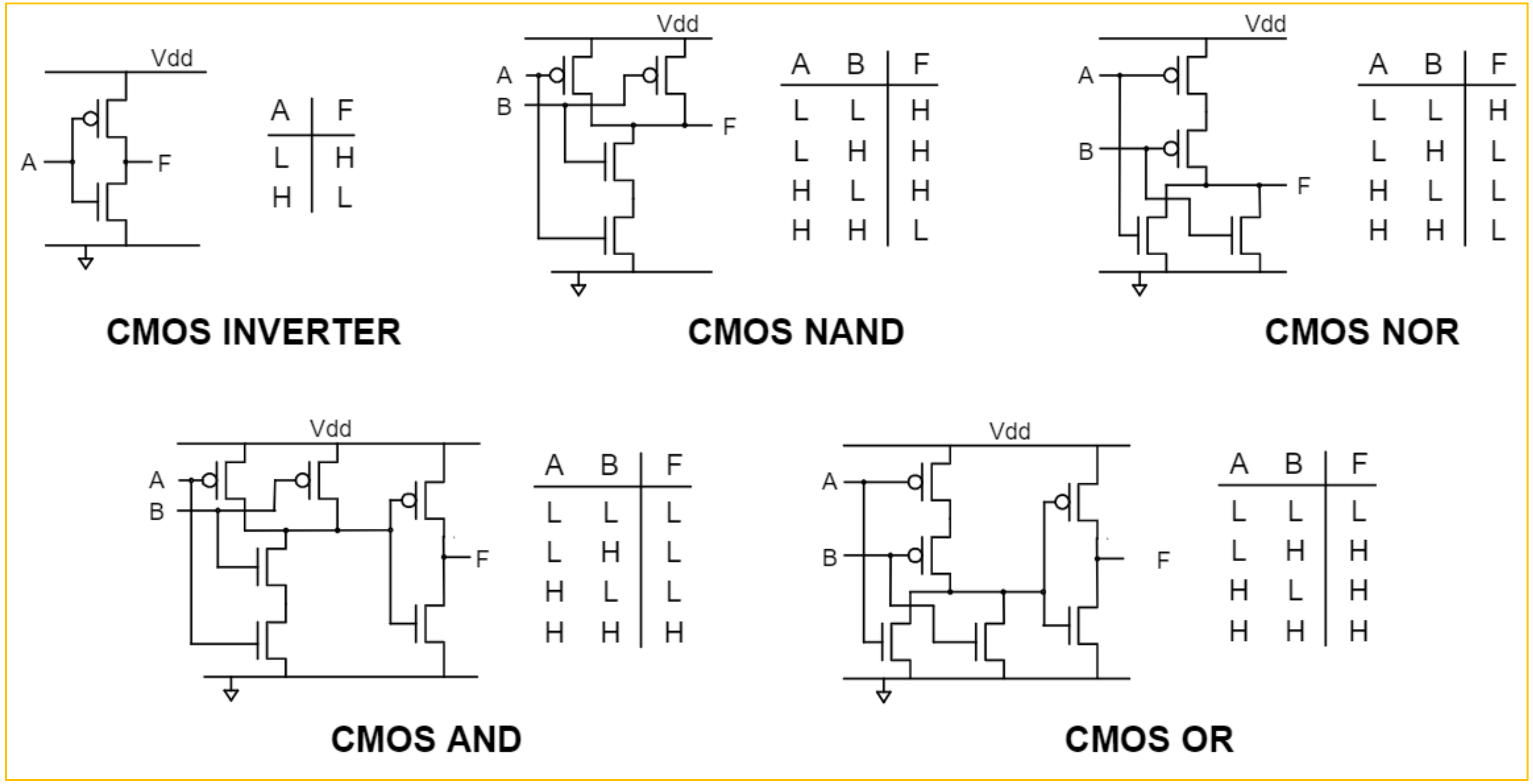
| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



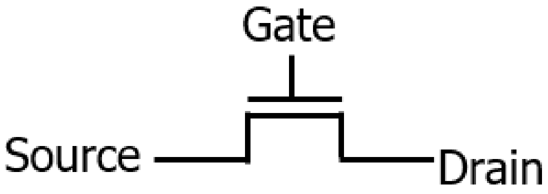
Let’s zoom in

# Logic gates can be created with transistors

- CMOS implementation of logic gates
  - Complementary Metal-Oxide Semiconductor

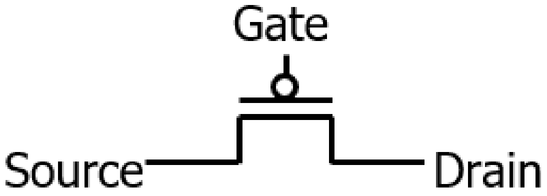



Transistors are just on/off switches




$V_{GS} > V_T$: Switch CLOSED

$V_{GS} < V_T$: Switch OPEN



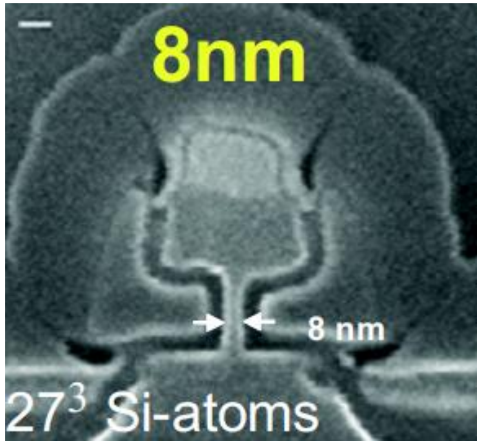

$V_{SG} > |V_T|$: Switch CLOSED

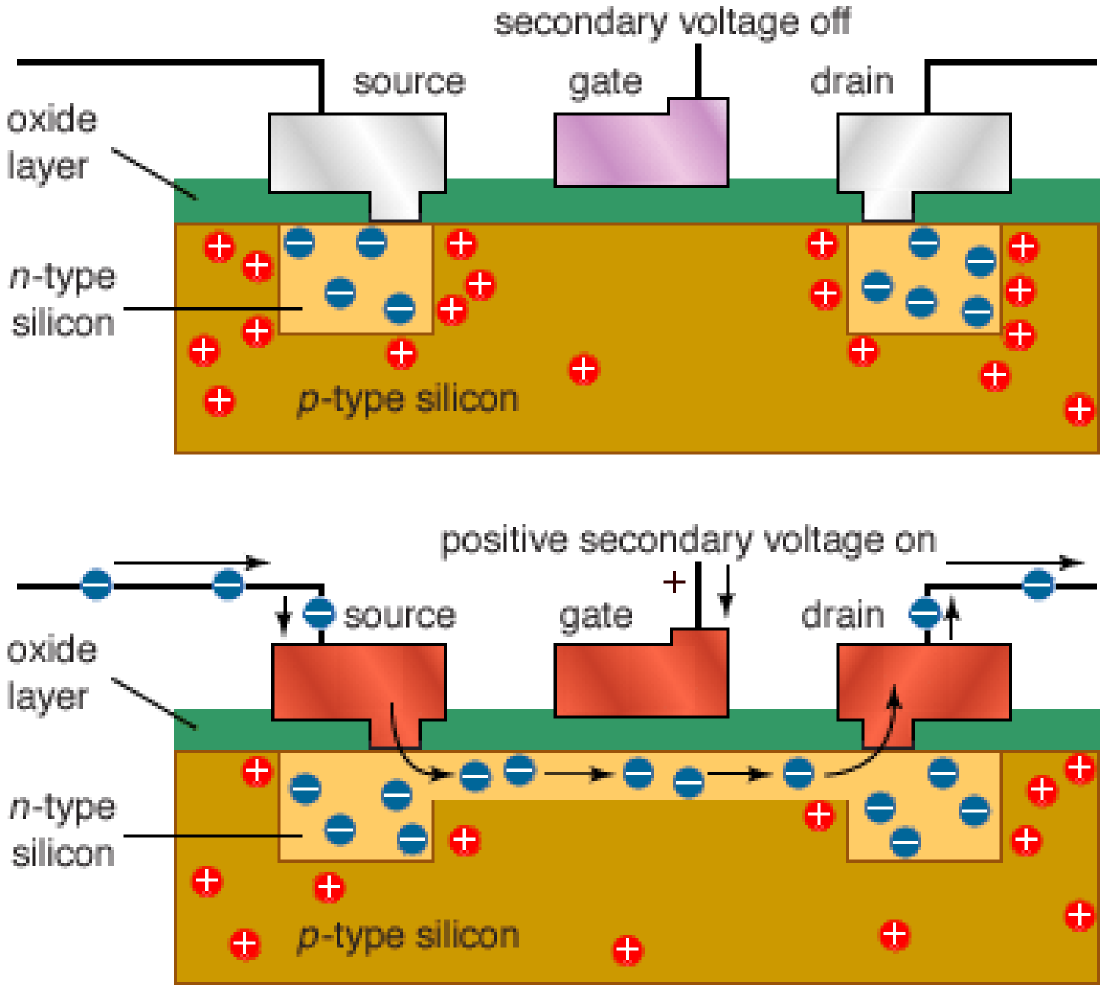
$V_{SG} < |V_T|$: Switch OPEN

Let’s zoom in

# Transistors are made out of silicon and other materials

- Turning gate on/off causes source and drain to connect or disconnect
  - Acts as a switch
- We can make very small transistors

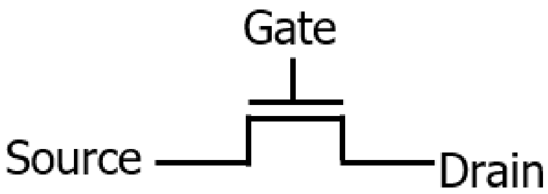


That’s the bottom

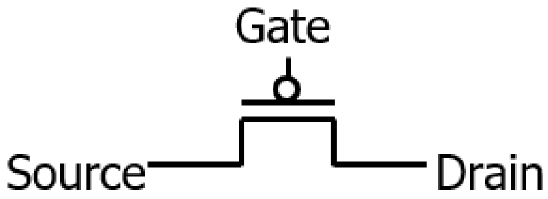
# Zooming out again

- Transistors make logic gates



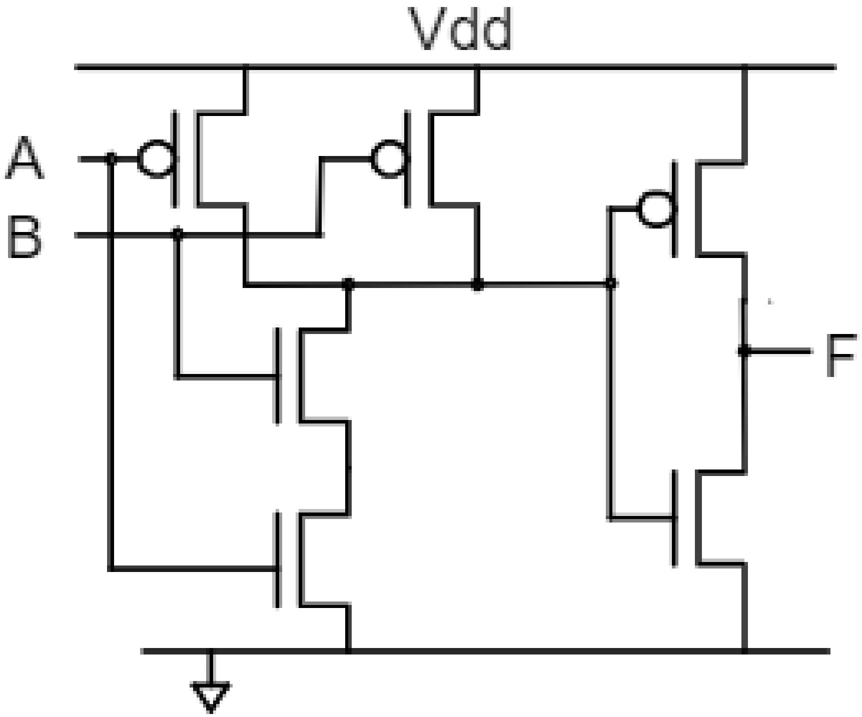

$V_{GS} > V_T$: Switch CLOSED

$V_{GS} < V_T$: Switch OPEN




$V_{SG} > |V_T|$: Switch CLOSED

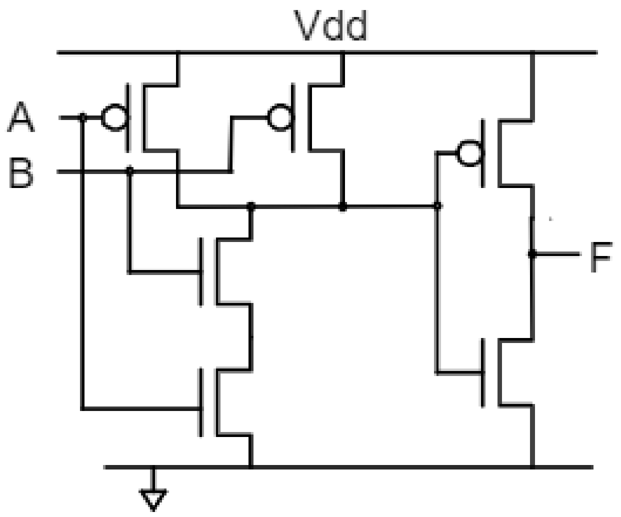
$V_{SG} < |V_T|$: Switch OPEN



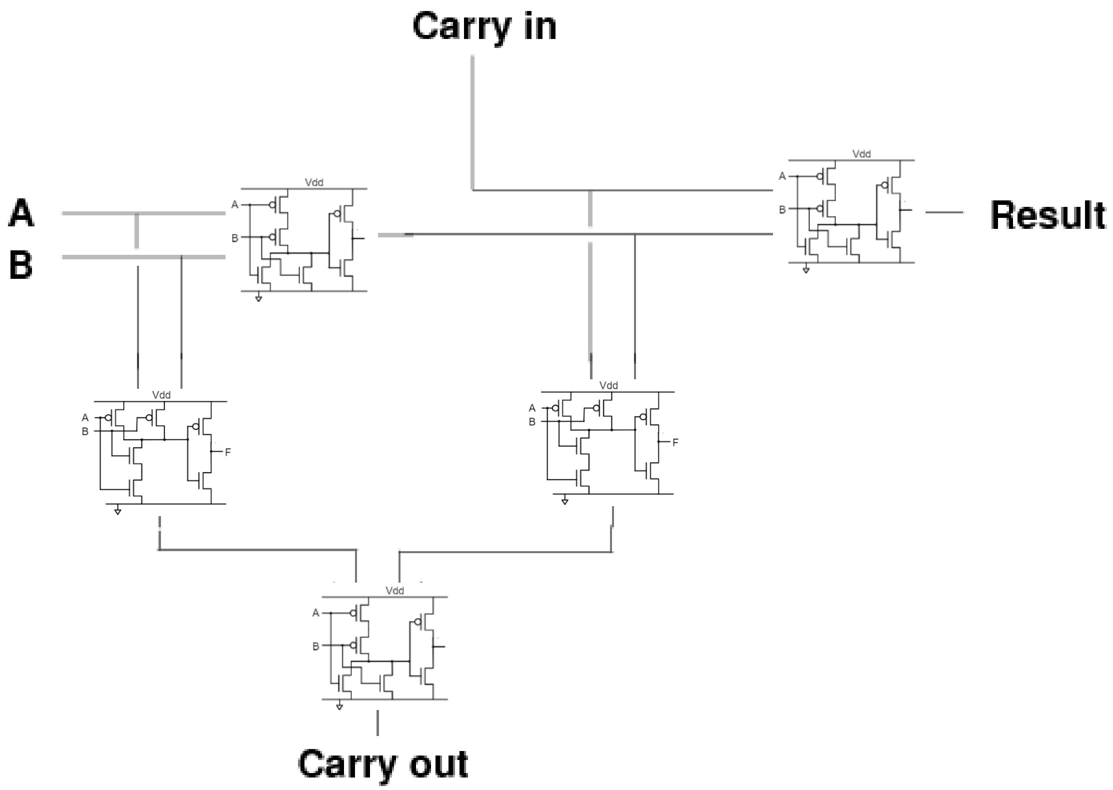

1-bit AND gate

# Zooming out again

- Logic gates make operations



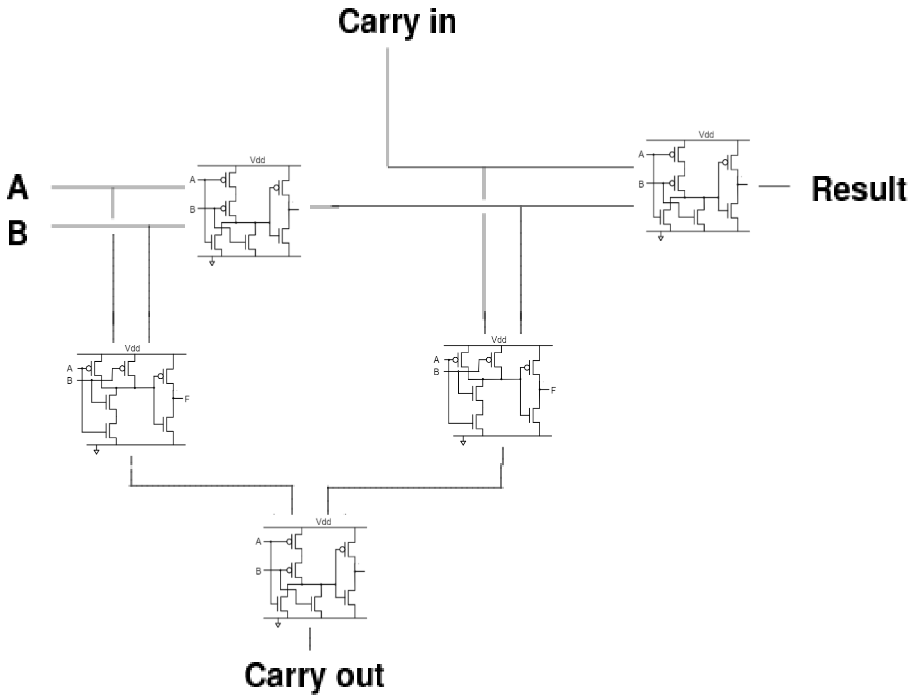

1-bit AND gate



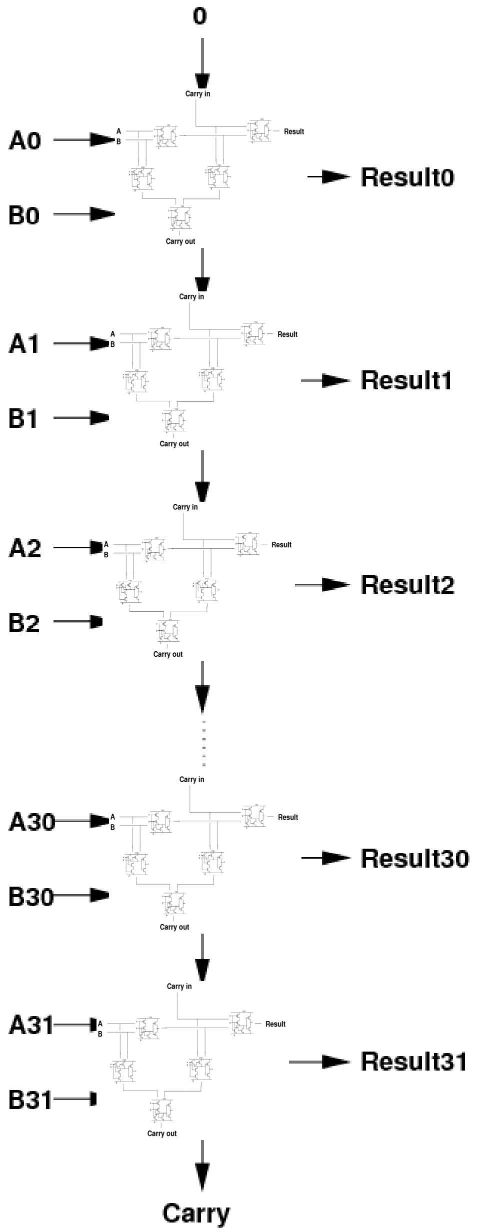

1-bit ADD operation

# Zooming out again

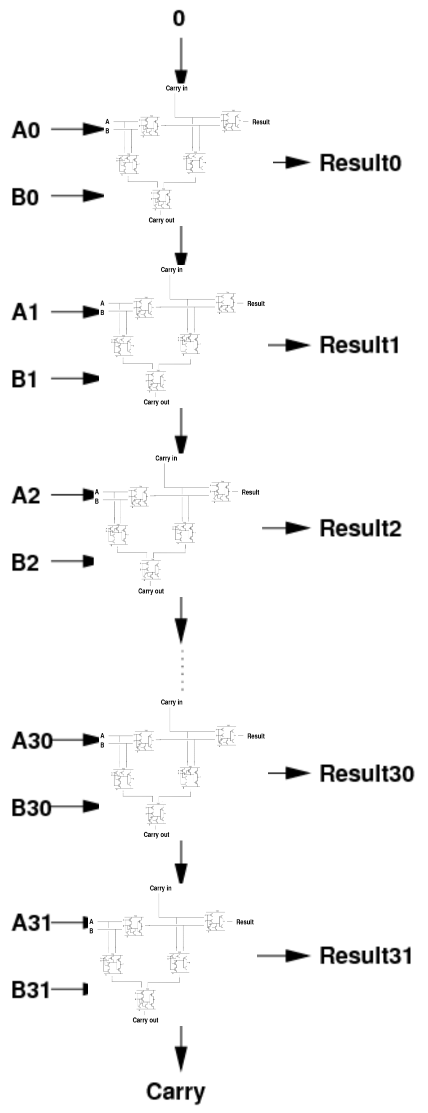
- 1-bit operations make 32-bit operations



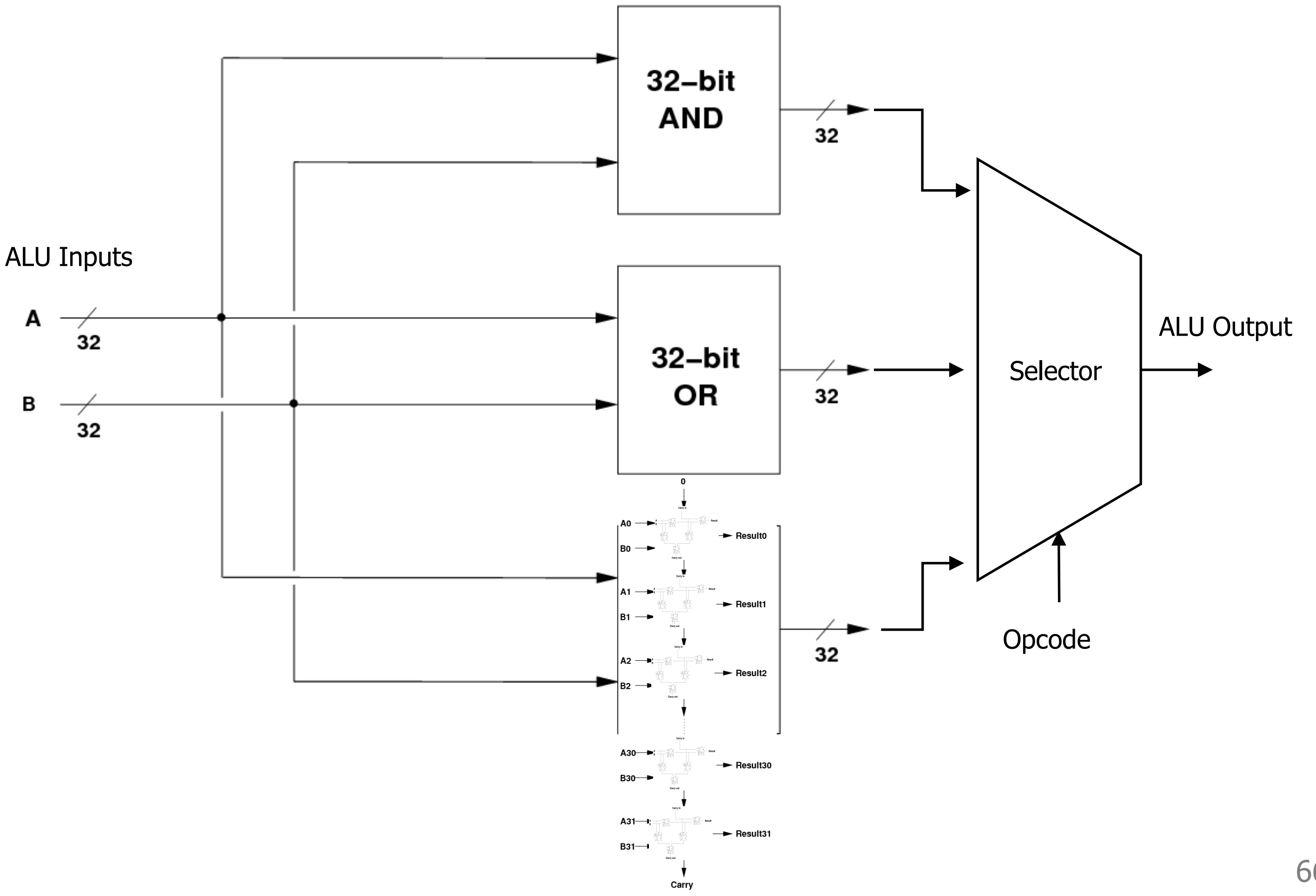

1-bit ADD operation



# Zooming out again




- Operations make an ALU



# ALU allows us to execute operations

1. Reads instruction from memory
2. Decodes it into an Operation plus Configurations
   - Immediates, Registers, Memory, etc.
3. Reads from source (based on configuration)
4. **Executes that operation**
5. Writes to destination (based on configuration)

# All the way back to software

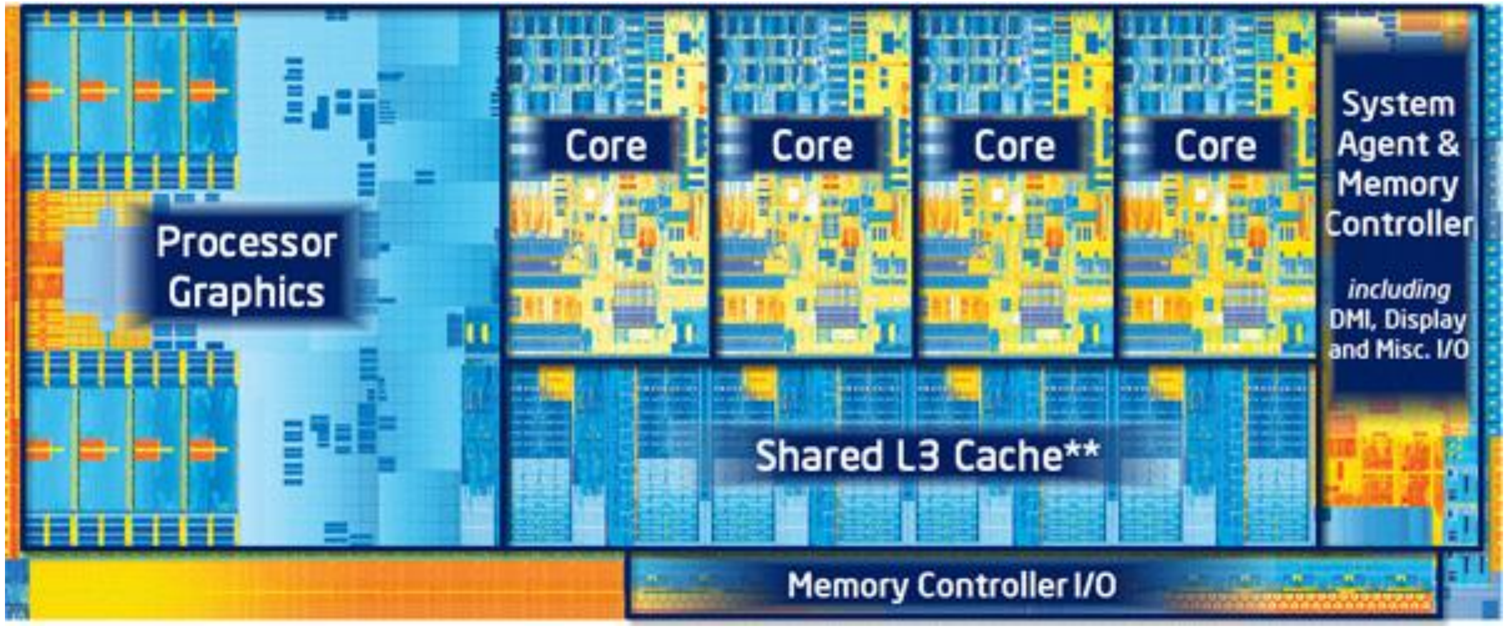
```
        test:
         48 8d 04 7e
4011da   lea    (%rsi,%rdi,2),%rax
         48 8d 04 10
4011de   lea    (%rax,%rdx,1),%rax
         48 29 f7
4011e2   sub    %rsi,%rdi
         48 01 f8
4011e5   add    %rdi,%rax
         48 8d 84 08 13 02 00 00
4011e8   lea    0x213(%rax,%rcx,1),%rax
         c3
4011f0   ret
```

- C compiles into assembly
- Assembly translates into machine code
- Machine code specifies what should be executed

# A processor is just a lot of transistors connected very carefully

- ALU plus other operations make up a Core
  - And decode logic

- Multiple cores, plus registers, plus caches make up a Processor
  - And other stuff these days like graphics



# Outline

- Structure Layout
- Struct Padding and Alignment
- Unions
- Assembly to Transistors (and back)