# 1 Introduction

## 1.1 *Source* program to executable object

C statements are translated to low-level *machine-language* instructions. These instructions are then packaged in a form called *executable object program* and stored as a binary disk file. The translation is performed in a four phases shown in Fig. 1.

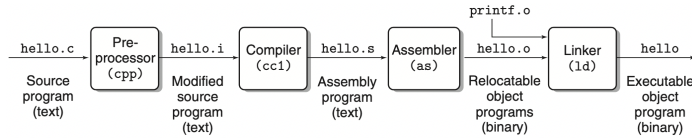



**Figure 1. The compiler system.**

1. **Preprocessing.** The preprocessor (cpp) modifies the C program hello.c according to directives that begin with the # character, resulting C program hello.i. For example, #include <stdio.h> includes the header file directly into the program.
2. **Compilation.** The compiler (cc1) translates hello.i to the *assembly-language program* hello.s.
3. **Assembly.** The assembler (as) translates hello.s into machine-language instructions, packages them in a *reloactable object program*, resulting in the binary object file hello.o.
4. **Linking.** The linker (ld) links the hello.o with the functions that are the part of the *standard C library* such as printf, resulting a executable object file hello.

## 1.2 Hardware organization

- **Main memory.** Main memory is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program.
- **Processor.** The *central processing unit* (CPU), or simply *processor*, is the engine that interprets instructions stored in main memory. At its core is a word-size storage device (or *register*) called the *program counter* (PC). PC points the address of some machine-language instruction in main memory. Processor repeatedly executes the instruction pointed at by the PC and updates the PC to point to the next instruction.

### 1.2.1 Virtual memory

Each process has the same uniform view of memory: *virtual address space*. It consists of a number of well-defined areas with purpose, as shown in Fig. 2.

- *Program code and data* initialized from the executable.
- *Heap* expands and contracts dynamically at run time as a result of malloc and free.
- *Shared libraries* such as C standard library.
- *Stack* which compiler uses to implement function calls.
- *Kernel virtual memory* where application programs are not allowed to read or write.

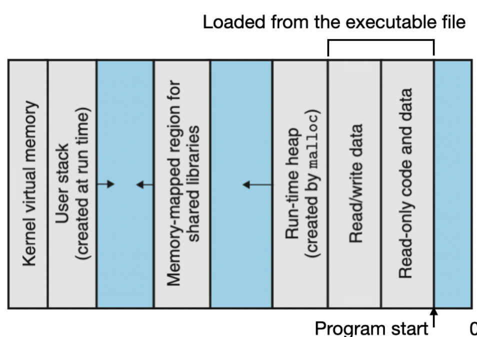



**Figure 2. Process virtual address space.**

# 2 Represent & Manipulate Information

Computers store and process information represented as two-valued signals: *bits*. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable.

## 2.1 Bit operation and logic operation

- **Bit operators** & (and), | (or), ~ (not), ∧ (Exclusive-Or/XOR)
- **Logic operators** &&, ||, ! view 0 as false, any nonzero as true
- **Shift operator**: « (left shift), » (right shift)
  - *Logical shift*: when right shifting unsigned, fill 0's on left
  - *Arithmetic shift*: when right shifting signed, fill most significant bits on left
  - Undefined Behavior when shift amount < 0 or ≥ size

## 2.2 Information storage

### 2.2.1 Hexadecimal notation

Computers use blocks of 8 bits, or *bytes*, as the smallest addressable unit of memory. The values of byte is $00000000_2$ to $11111111_2$ in binary, $0_{10}$ to $255_{10}$ in decimal, $00_{16}$ to $FF_{16}$ in hexadecimal.

Since binary notation is too verbose, we write bit patterns as base-16, or *hexadecimal* numbers (hex). In C, numeric constants starting with 0x are interpreted as hex. When converting a binary number which is not a multiple of 4 to hex, make the *leftmost* group be fewer than 4 bits, and pad with leading zeros.

### 2.2.2 Data sizes

| C data type | 32-bit | 64-bit | x86-64 |
|---|---|---|---|
| char | 1 | 1 | 1 |
| short | 2 | 2 | 2 |
| int | 4 | 4 | 4 |
| long | 4 | 8 | 8 |
| float | 4 | 4 | 4 |
| double | 8 | 8 | 8 |
| long double | - | - | 10/16 |
| pointer | 4 | 8 | 8 |

**Table 1. Size of basic C data types**

Most machines support two different floating-point formats: single precision (float), and double precision (double).

### 2.2.3 Addressing and byte ordering

Multi-byte object is stored sequentially, with the address of object at the smallest address. In *little endian*, least significant byte comes first, and in *big endian*, most significant byte comes first.

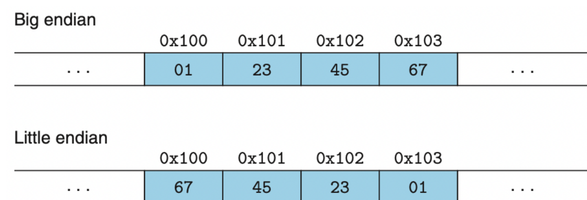



**Figure 3. Little and big endian.**

## 2.3 Integer representation

### 2.3.1 Unsigned encodings

Most significant bit is included as part of the numeric value. C interprets integers if have U suffix, *e.g.*, 422U.

**Definition 1** (Unsigned encoding)**.** For $x = [x_{w-1}, \cdots, x_0]$,

$$\mathrm{B2U}_w(x) = \sum_{i=0}^{w-1} x_i 2^i \tag{1}$$

Unsigned values range from UMin= 0 to UMax= $2^w - 1$.

#### 2.3.2 Two's complement encoding (Signed encodings)

Most significant bit indicates sign.

**Definition 2** (Two's complement encoding)**.** For $x = [x_{w-1}, \cdots, x_0]$,

$$\mathrm{B2T}_w(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2}$$

Two's complement values range from TMin= $-2^{w-1}$ ($10\cdots0_2$) to TMax= $2^{w-1}-1$ ($01\cdots1_2$). Has asymmetric range.

#### 2.3.3 Conversion and casting

Characteristics of signed and unsigned encodings are:

- **Equivalence**: Same encoding for nonnegative values.
- **Uniqueness**: Encoding is bijection, thus can invert mapping.

Conversion between unsigned and two's complement numbers: **Keep bit representations** and reinterpret. Both explicit and implicit casting behaves same. If there is a mix of unsigned and signed integers in single expression, **signed values implicitly cast to unsigned**. This includes comparison operations.

#### 2.3.4 Expanding and truncating

Expanding (*e.g.*, short to int)

- Unsigned: add zeros
- Signed: extend sign (most significant bit)

Truncating (*e.g.*, int to short)

- Unsigned:$x \bmod 2^k$
- Signed: $\mathrm{U2T}_k(x \bmod 2^k)$

### 2.4 Integer arithmetic

mod $2^w$ the result. Two's complement operation result should be interpreted as two's complement number. Two's complement operation can result in *negative* and *positive overflow*.

#### 2.4.1 Multiply with shift

Since most machines shift and add faster than multiply, compiler automatically converts multiplication to shift and addition. For example, u * 24 to (u « 5) - (u « 3).

#### 2.4.2 Power-of-2 divide with shift

Use logical shift when dividing unsigned, arithmetic shift when dividing signed. x » k when dividing nonnegative number, (x+(1«k)-1) » k when dividing negative number to round toward 0.

#### 2.4.3 Counting down with unsigned

unsigned 0 minus 1 goes to INT MAX.

```
1 size_t i; /* defines unsigned with length = word size */
2 for (i = cnt-2; i < cnt; i--)
3 /* i >= 0 won't work */
4     a[i] += a[i+1];
```

### 2.5 Floating point representation

#### 2.5.1 Fractional binary numbers

Use notation $1.0 - \epsilon$ for $0.11\cdots1_2$. Limitations are:

- Can only exactly represent numbers of the form $x/2^k$
- Just one setting of binary point → limited range

#### 2.5.2 IEEE floating point

$$(-1)^s M 2^E \tag{3}$$

- MSB encodes the sign bit $s$

| Precision | Exp | Frac | Bias |
|---|---|---|---|
| Single | 8 | 23 | 127 |
| Double | 11 | 52 | 1023 |
| Extended | 15 | 63 or 64 | 16383 |

**Table 2. Size of fields by precision**

- Exp encodes the exponent $E$; $E$=Exp-Bias
- Frac encodes the significand $M \in [1, 2)$ with implied leading 1

Below lists special values:

- Exp=0...0 (denormalized numbers): Exponent value $E$=1-Bias. Significand coned with implied leading 0.
- Exp=1...1, Frac=0...0: Infinity.
- Exp=1...1, Frac≠ 0...0: Not-a-number (NaN)

#### 2.5.3 Floating point in C

- double/float to int: truncate Frac and set to TMin when NaN
- int to double: exact conversion as long as int has ≤ 53 bit
- int to float: round (See Sec. 2.6.3)

### 2.6 Floating point arithmetic

First compute exact result and fix into desired precision: overflow if exponent is too large, round to fit Frac. Exact multiplication by:

$$(-1)^{s_1 \wedge s_2}(M_1 \times M_2)2^{E_1+E_2} \tag{4}$$

Exact addition by aligning binary points.

#### 2.6.1 Fixing

- If $M \geq 2$, shift $M$ right, increment $E$
- If $M < 1$, shift $M$ left, decrement $E$
- If $E$ out of range, overflow
- Round $M$ to fit Frac

#### 2.6.2 Round-to-even

When bits to right of rounding position is $10\cdots0_2$, round to LSB=0.

#### 2.6.3 Creating floating point number

1. **Normalize.** Obtain fraction with implied leading 1.
2. **Round-to-even.** Let $M = 1.BBGRX\cdots X$. If guard bit $G = 1$, round bit $R = 1$, sticky bit, *i.e.*, or of $X\cdots X$ is 0, increment.
3. **Postnormalize**. Handle overflow

#### 2.6.4 Mathematical properties

- Closed, commutative, monotonicity (almost)
- Not associative

### 2.7 Other representations

- String: array of characters terminated by the null (having value 0) character
- Code: encoded instructions which are different by machine types

# 3 Machine-level representation

Based on x86-64, the most popular machine language.

## 3.1 Machine code

- **Instruction set architecture** (ISA, or architecture): The parts of a processor design that one needs to understand or write assembly/machine code.
- **Microarchitecture**: Implementation of the architecture.
- **Machine code**: The byte-level program that processor runs
- **Assembly code**: A text representation of machine code

### 3.1.1 Unix commands

Command gcc -Og p1.c p2.c -o p instructs:

- gcc -Og: compile use basic optimizations
- -o p: put resulting binary in file p

Command gcc -Og -S p1.c instructs GCC to run the compiler, generating an assembly file p1.s, and go no further.

### 3.1.2 Machine instruction

| C Code | *d = t; | Store t where designated by d |
|---|---|---|
| Assembly | movq %rax, (%rbx) | Move 8-byte value to memory<br>t: Register %rax<br>d: Register %rbx<br>*d: Memory M[%rbx] |
| Object code | 0x40059e: 48 89 03 | 3-byte instruction at 0x40059e |

**Table 3. Machine instruction example**

The *disassemblers* generate a format similar to assembly code from the machine code. x86-64 instructions can range in length from 1 to 15 bytes. From given starting position, there is a unique decoding of the bytes into machine instructions.

**Command.** objdump -d a or gdb a and disassemble func

## 3.2 Registers

Compiler can use registers as temporary storage. Integer registers (8-byte) are total 16 and as follows:

| Register | Lower 4 bytes | LSB |
|---|---|---|
| %rax ··· %rdx | %eax ··· %edx | %al ··· %dl |
| %r8 ··· %r15 | %r8d ··· %r15d | %r8b ··· %r15b |
| %rsi,%rdi | %esi,%edi | %sil,%dil |
| %rsp,%rbp | %esp,%ebp | %spl,%bpl |

**Table 4. Integer registers**

## 3.3 Assembly operations: Basic

### 3.3.1 Operand formats

- Immediate: constant integer data prefixed with $, *e.g.*, $0x400
- Register: one of the integer registers, *e.g.*, %rax
- Memory: 8 bytes memory at given address, *e.g.*, (%rax)

**Memory computation.** D(Rb,Ri,S) denotes Mem[Rb+S*Ri+D]

- D: offset (displacement) of 1, 2, or 4 bytes
- Rb: base register, any of integer registers
- Ri: index register, any except for %rsp

| Form | Address | Example |
|---|---|---|
| D(Rb) | Mem[Rb+D] | **0x8(%rdx)**: 0xf000+0x8=0xf008 |
| (Rb,Ri) | Mem[Rb+Ri] | **(%rdx,%rcx)**: 0xf000+0x100=0xf100 |
| (Rb,Ri,S) | Mem[Rb+S*Ri] | **(%rdx,%rcx,4)**: 0xf000+4*0x100=0xf400 |

**Table 5. Examples. %rdx points 0xf000 and %rcx points 0x0100.**

### 3.3.2 Data types

Suffix (listed in Table 6) denoting data type are added to Assembly operation, *e.g.*, movq, movb, and movl.

| C type | Suffix | Size |
|---|---|---|
| char | b | 1 |
| short | w | 2 |
| int | l | 4 |
| long/char* | q | 8 |
| float | s | 4 |
| double | l | 8 |

**Table 6. C data types and suffices.**

## 3.4 Assembly operations: Control

### 3.4.1 Processor state

- Temporary data: %rax, ···
- Location of runtime stack: %rsp
- Location of current code control point (PC): %rip
- Status of recent tests (condition code): CF, ZF, SF, OF

### 3.4.2 Control codes

Control codes are implicitly (automatically) set as a side effect of arithmetic operations (not lea).

- CF (carry flag) is set when unsigned overflow
- ZF (zero flag) is set when result is zero
- SF (signed flag) is set when result is negative
- OF (overflow flag) is set when two's complement overflow

Compare and test instruction can explicitly set control codes.

### 3.4.3 Condition

goto creates separate code regions for conditional expressions.

**Conditional moves.** a ? b : c. Since both values get computed, may have undesirable effects.

**Bad cases for conditional move**

```
1 val = Test(x) ? Hard1(x) : Hard2(x); // Expensive.
2 val = p ? *p : 0; // Risky.
3 val = x > 0 ? x*=7 : x+=3; // Side effect.
```

### 3.4.4 Loops

**General loop translations**

```
1 goto test; // w/o this line if Do-While
2 loop:
3     // Body
4 test:
5     if (Test) goto loop;
```

### 3.4.5 Switch

Compiler generates *jump table*, an array of addresses.

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

Indirect jump

**Figure 4. Switch example**

## 3.5 Assembly operations: Procedure

- **Passing control**. Set the PC to the beginning of the procedure code and get back to return point.
- **Passing data**. Arguments and return value.
- **Managing memory**. Allocate local variables and deallocate.

### 3.5.1 Runtime stack

x86-64 stack grows toward lower addresses and register %rsp contains the lowest (top) stack address. Data can be stored and retrieved using push, pop instructions. Space for data can be allocated on the stack by simply decrementing the stack pointer and deallocated by incrementing the stack pointer. Decrementing the stack pointer is called *set-up*, *e.g.*, subq $16, %rsp, and incrementing is called *clean-up*, *e.g.*, addq $16, %rsp.

### 3.5.2 Passing control

The *return address* is the address of the next instruction right after the call. Procedure call push the return address on stack and procedure return pop the address.

### 3.5.3 Passing data

First 6 arguments are saved to %rdi, %rsi, %rdx, %rcx, %r8, %r9, and return value is saved to %rax. More arguments are saved to the stack.

### 3.5.4 Stack frame

Stack is allocated in frame, a state for instantiation of a single procedure. *Stack frame* contains the return address, local storage, and temporary space. %rbp points the start of the frame. Argument build is parameters for function about to call, thus optional.

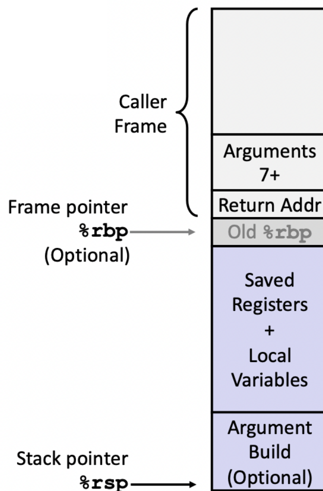



**Figure 5. Stack frame**

### 3.5.5 Register saving

Callee controls (increments/decrements) %rsp.

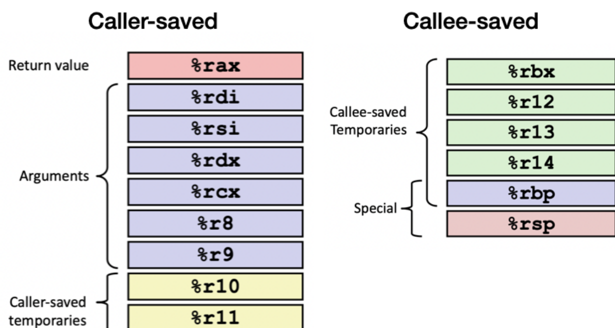



**Figure 6. Saved registers**

## 3.6 Assembly operations: Data

### 3.6.1 Array

Array of data type T and length L is contigously allocated region of L× sizeof(T) bytes in memory.

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

**Figure 7. Accessing an item of an array**

### 3.6.2 Multidimensional array

Array of T A[R][C] is contigously allocated in row-major ordering in R×C×sizeof(T) bytes in memory. We can access A[i][j] via

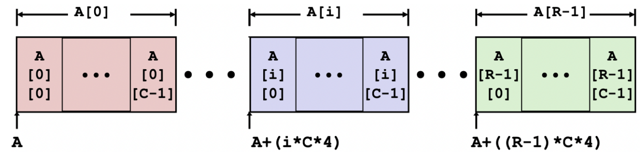



**Figure 8. Allocation and access of 2-dimensional array**

Mem[Mem[A]+sizeof(T)*(C*i+j)].

### 3.6.3 Multi-level array

Multi-level array T *A[R] stores pointer which points to array of T. Element access is different from multidimensional array; We can access A[i][j] via Mem[Mem[A+8*i]+sizeof(T)*j].

### 3.6.4 Structures

Represented as a block of memory where fields ordered according to the declaration.

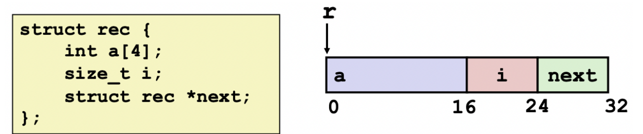



**Figure 9. Allocation of structure**

### 3.6.5 Structure alignment

Each structure field has size $k$. Initial address and structure length should be multiples of max $k$. Gaps are inserted to ensure alignment. To save space, put large data types first.

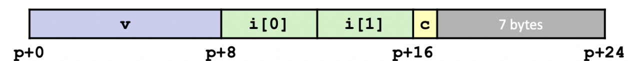



**Figure 10. Example of alignment.** max $k$ = 8**.**

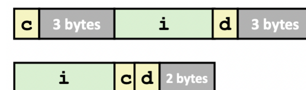



**Figure 11. Order of the fields matters.**

### 3.6.6 Unions

Union allocates according to the largest element, thus can only use one field at a time.

### 3.6.7 Floating point

Floating point registers (XMM registers) are total 16, each 16 bytes. We can have multiple integers or floats within the register for parallel computing.

## 3.7 Memory allocation

malloc *dynamically* allocates the memory of given size and returns the starting address of the allocated memory. *Dynamic* means that the address is set in the runtime.

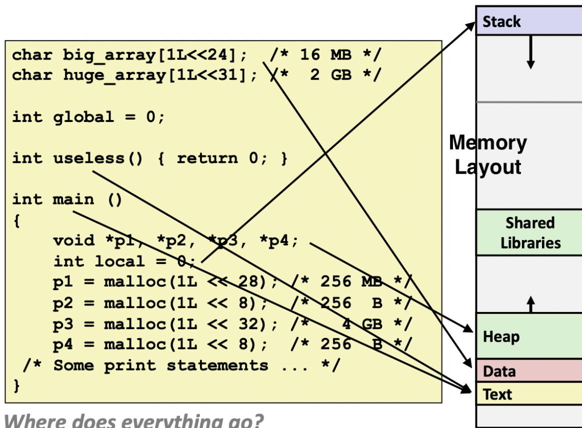



**Figure 12. Memory layout and malloc.**

## 3.8 Buffer overflow

When exceeding the memory size allocated for an array. It can cause security vulnerabilities.

**Vulnerable buffer example**

```
void echo(){
    char buf[4]; // Way too small!
    gets(buf); // Runs until EOF so no way to specify
     limit on number of characters
    puts(buf); }
 void call_echo() {
    echo(); }
```

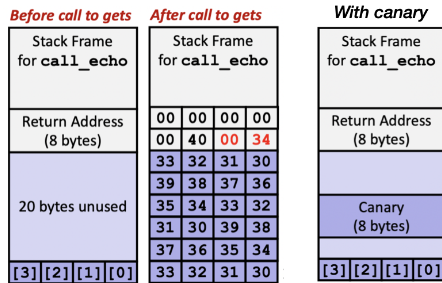



**Figure 13. Buffer overflowed and corrupted the return pointer.**

Buffer overflow can corrupt the return pointer and may corrupt state (cause segmentation fault) or cause undesired action.

### 3.8.1 Code injection attack

Return address can be corrupted to point exploit code. To prevent,

- **Avoid vulnerabilities in code.** Use methods that limits string lengths, *e.g.*, fgets, strncpy instead of gets, strcpy.
- **System-level protections.** At the start of the program, allocate random amount of space on stack, making difficult for hacker to predict the beginning of the inserted code.
- **Non-executable code segments.** Mark stack as non-executable.
- **Stack canaries.** Place special value beyond buffer and check for corruption before exiting function. Add -fstack-protector.

```
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
```

**Figure 14. Disassembly of canary.**

### 3.8.2 Return-oriented programming attacks

Use the existing code *gadgets* to construct the program. Hackers also can repurpose byte codes to encode different instruction.

| Operation | Details |
|---|---|
| mov src, dest | Example: mov $0x4, %rax — C analog: temp = 0x4;<br>Example: mov %rax, (%rdx) — C analog: *p = temp;<br>Cannot do memory-memory transfer |
| lea src, dest | Assign src as dest<br>• Computing addresses without a memory reference<br>• Arithmetic expressions of the form x+k*y |
| add src, dest | dest = dest + src |
| sub src, dest | dest = dest - src |
| imul src, dest | dest = dest * src |
| sal src, dest | dest = dest « src |
| sar src, dest | dest = dest » src (arithmetic shift) |
| shr src, dest | dest = dest » src (logical shift) |
| xor src, dest | dest = dest ∧ src |
| and src, dest | dest = dest & src |
| or src, dest | dest = dest | src |
| inc dest | dest = dest + 1 |
| dec dest | dest = dest - 1 |
| neg dest | dest = -dest |
| not dest | dest = ~dest |
| comp src2, src1 | Set flags based on src1-src2<br>• CF=1 if unsigned overflow<br>• ZF=1 if src1==src2<br>• SF=1 if (src1-src2)<0<br>• OF=1 if two's complement overflow |
| test src2, src1 | Set flags based on src1&src2<br>Cannot set CF or OF<br>• ZF=1 if src1&src2==0<br>• SF=1 if src1&src2<0 |
| **setX dest** | Set LSB of dest based on codes |
| **jX dest** | Jump to dest based on codes |
| e | ZF(equal/zero) |
| ne | ~ZF (not equal/nonzero) |
| s | SF (negative) |
| ns | ~SF (nonnegative) |
| g | ~(SF∧OF)& ~ ZF (signed greater) |
| ge | ~(SF∧OF) (signed greater or equal) |
| l | (SF∧OF) (signed less) |
| le | (SF∧OF)|ZF (signed less or equal) |
| a | ~CF&~ZF (unsigned above) |
| b | CF (unsigned less) |
| jmp | Unconditional |
| push src | Decrement %rsp & write operand at address given by %rsp |
| pop dest | Increment %rsp & store value at dest (must be register) |
| call label | Push return address on stack & jump to label |
| ret | Pop address from stack & jump to the address |

**Table 7. Assembly operations.**

# 4 Code

## 4.1 Chpater 1

### Memory reference bug example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; // Possibly out of bounds
  return s.d;
}
```

**Figure 15. Memory reference bug explanation.**

### Memory system performance example

```
void copy(int src[2048][2048], int dst[2048][2048]){
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
    /* Takes 4.3ms */
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
    /* Takes 81.8ms */
}
```

## 4.2 Chapter 2

### Print byte representation of data

```
typedef unsigned char *pointer;
void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
    /* %p print pointer, %x print hexadecimal */
}
```

### Integer puzzles

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;

x < 0 then ((x*2) < 0) // False: underflow
ux >= 0 // True
x & 7 == 7 then (x<<30) < 0 // True
ux > -1 // False
x > y then x < -y // False: -1 promoted to unsigned so
    INT_MAX
x * x >= 0 // False: overflow
x > 0 && y > 0 then x + y > 0 // False: overflow
x >= 0 then x <= 0 // True
x <= 0 then -x >= 0 // False: -TMin == TMin
(x|-x)>>31 == -1 // False: True except for x = 0
ux >> 3 == ux/8 // True
x >> 3 == x/8 // False: when negative x
x & (x-1) != 0 // False: x=0 or 1
```

### Floating point puzzles

```
int x = ...;
float f = ...;
double d = ...;
/* Assume neither d nor f is NaN */

x == (int)(float) x // False: float conversion can lose
precision
x == (int)(double) x // True
f == (float)(double) f // True
d == (double)(float) d // False
f == -(-f); // True
2/3 == 2/3.0 // False
d<0.0 then ((d*2)<0.0) // True
d>f then -f>-d // True
d*d>=0.0 // True
(d+f)-d == f // False: d + f could be infinite
```

## 4.3 Chapter 3

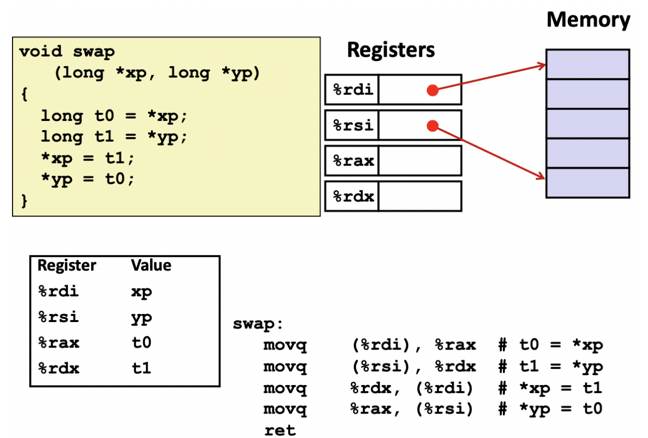



**Figure 16. Example: Swap**

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

**Figure 17. Condition with branch (left) and goto (right)**

```
absdiff:
   cmpq    %rsi, %rdi  # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

**Figure 18. Conditional branch (old style)**

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

Figure 19. Conditional move

## Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Figure 20. Recursive function example

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Figure 21. Array loop example

| Declaration | An Cmp | An Bad | An Size | *An Cmp | *An Bad | *An Size | **An Cmp | **An Bad | **An Size |
|---|---|---|---|---|---|---|---|---|---|
| int A1[3] | Y | N | 12 | Y | N | 4 | N | - | - |
| | Array of int | | | A1[0] | | | - | | |
| int *A2[3] int (*A2[3]) | Y | N | 24 | Y | Y | 8 | Y | N | 4 |
| | Array of int* | | | A2[0] | | | *A2[0] | | |
| int (*A3)[3] | Y | N | 8 | Y | Y | 12 | Y | Y | 4 |
| | Pointer of array | | | *A3 | | | (*A3)[0] | | |

Table 8. Cmp means compilation and Bad means bad reference.

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi           # n*i
  leaq    (%rsi,%rdi,4), %rax  # a + 4*n*i
  movl    (%rax,%rcx,4), %eax  # a + 4*n*i + 4*j
  ret
```

Figure 22. Access of NxN matrix

| Declaration | An Cmp | An Bad | An Size | *An Cmp | *An Bad | *An Size | **An Cmp | **An Bad | **An Size | ***An Cmp | ***An Bad | ***An Size |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| int A1[3][5] | Y | N | 60 | Y | N | 20 | Y | N | 4 | N | - | - |
| | 2d int | | | A1[0] | | | A1[0][0] | | | - | | |
| int *A2[3][5] int *(A2[3][5]) | Y | N | 120 | Y | N | 40 | Y | N | 4 | Y | Y | 4 |
| | 2d int* | | | A2[0] | | | A2[0][0] | | | *(A2[0][0]) | | |
| int (*A3)[3][5] | Y | N | 8 | Y | Y | 60 | Y | Y | 20 | Y | Y | 4 |
| | Point 2d int | | | *A3 | | | (*A3)[0] | | | (*A3)[0][0] | | |
| int (*A4[3])[5] | Y | N | 24 | Y | N | 8 | Y | Y | 20 | Y | Y | 4 |
| | Point 1d int | | | A4[0] | | | *(A4[0]) | | | *(A4[0])[0] | | |

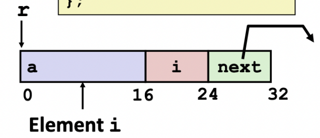
Table 9. Cmp means compilation and Bad means bad reference.

## Following Linked List

- C Code

```
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;
  }
}
```

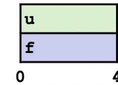



| Register | Value |
|---|---|
| %rdi | r |
| %rsi | val |

```
.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
```

Figure 23. Linked list example

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




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

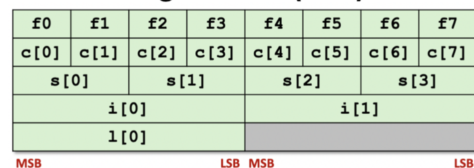
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as (float) u ? No    Same as (unsigned) f ?No

Figure 24. Union allocation example

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

## Big endian (Sun)




## Little endian (x86-64)

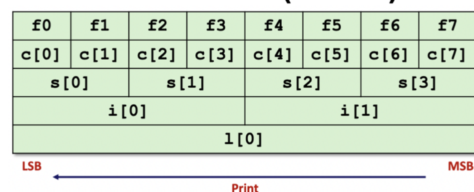



Figure 25. Union allocation example

### Assembly practice 1

```
1  Dump of assembler code for function main:
2  0x00646 <+0>:  sub $0x8,%rsp
3  0x0064a <+4>:  mov $0x0,%eax
4  0x0064f <+9>:  jmp 0x665 <main+31>
5  0x00651 <+11>: movslq %eax,%rcx
6  0x00654 <+14>: lea 0x2009e5(%rip),%rdx # 0x201040 <memo>
7  0x0065b <+21>: movl   $0xffffffff,(%rdx,%rcx,4)
8  0x00662 <+28>: add    $0x1,%eax
9  0x00665 <+31>: cmp    $0x13,%eax
10 0x00668 <+34>: jle    0x651 <main+11>
11 0x0066a <+36>: mov    $0xa,%edi
12 0x0066f <+41>: callq  0x5fa <fib>
13 0x00674 <+46>: mov    $0x0,%eax
14 0x00679 <+51>: add    $0x8,%rsp
15 0x0067d <+55>: retq
16
17 Dump of assembler code for function fib:
18 0x005fa <+0>:  cmp    $0x1,%edi
19 0x005fd <+3>:  je     0x643 <fib+73>
20 0x005ff <+5>:  movslq %edi,%rax
21 0x00602 <+8>:  lea    0x200a37(%rip),%rdx # 0x201040 <
       memo>
22 0x00609 <+15>: mov    (%rdx,%rax,4),%eax
23 0x0060c <+18>: cmp    $0xffffffff,%eax
24 0x0060f <+21>: je     0x613 <fib+25>
25 0x00611 <+23>: repz retq
26 0x00613 <+25>: push   %rbp
27 0x00614 <+26>: push   %rbx
28 0x00615 <+27>: sub    $0x8,%rsp
29 0x00619 <+31>: mov    %edi,%ebx
30 0x0061b <+33>: lea    -0x1(%rdi),%edi
31 0x0061e <+36>: callq  0x5fa <fib>
32 0x00623 <+41>: mov    %eax,%ebp
33 0x00625 <+43>: lea    -0x2(%rbx),%edi
34 0x00628 <+46>: callq  0x5fa <fib>
35 0x0062d <+51>: add    %ebp,%eax
36 0x0062f <+53>: movslq %ebx,%rdi
37 0x00632 <+56>: lea    0x200a07(%rip),%rdx # 0x201040 <
       memo>
38 0x00639 <+63>: mov    %eax,(%rdx,%rdi,4)
39 0x0063c <+66>: add    $0x8,%rsp
40 0x00640 <+70>: pop    %rbx
41 0x00641 <+71>: pop    %rbp
42 0x00642 <+72>: retq
43 0x00643 <+73>: mov    %edi,%eax
44 0x00645 <+75>: retq
```

### Assembly practice 2

```
1  0x0074d <+0>:  sub    $0x28,%rsp
2  0x00751 <+4>:  mov    %fs:0x28,%rax
3  0x0075a <+13>: mov    %rax,0x18(%rsp)
4  0x0075f <+18>: xor    %eax,%eax
5  0x00761 <+20>: test   %edi,%edi
6  0x00763 <+22>: je     0x76f <add_node+34>
7  0x00765 <+24>: mov    %rsi,%rcx
8  0x00768 <+27>: mov    $0x0,%edx
9  0x0076d <+32>: jmp    0x784 <add_node+55>
10 0x0076f <+34>: mov    %rsp,%rax
11 0x00772 <+37>: mov    %rax,0x8(%rsi)
12 0x00776 <+41>: mov    $0x0,%eax
13 0x0077b <+46>: jmp    0x7a8 <add_node+91>
14 0x0077d <+48>: mov    0x10(%rcx),%rcx
15 0x00781 <+52>: add    $0x1,%edx
16 0x00784 <+55>: cmp    %edi,%edx
17 0x00786 <+57>: jl     0x77d <add_node+48>
18 0x00788 <+59>: mov    0x8(%rcx),%rax
19 0x0078c <+63>: mov    %rax,0x8(%rsp)
20 0x00791 <+68>: mov    %rcx,0x10(%rsp)
21 0x00796 <+73>: mov    0x8(%rcx),%rdx
22 0x0079a <+77>: mov    %rsp,%rax
23 0x0079d <+80>: mov    %rax,0x10(%rdx)
24 0x007a1 <+84>: mov    %rax,0x8(%rcx)
25 0x007a5 <+88>: mov    %rsi,%rax
26 0x007a8 <+91>: mov    0x18(%rsp),%rsi
27 0x007ad <+96>: xor    %fs:0x28,%rsi
28 0x007b6 <+105>:jne    0x7bd <add_node+112>
29 0x007b8 <+107>:add    $0x28,%rsp
30 0x007bc <+111>retq
```

### Hint

```
1 typedef struct node {
2     int value;
3     struct node * prev;
4     struct node * next;
5 } node;
```

### Assembly practice 3

```
1  0x0078d <+0>:   sub    $0x58,%rsp
2  0x00791 <+4>:   mov    %fs:0x28,%rax
3  0x0079a <+13>:  mov    %rax,0x48(%rsp)
4  0x0079f <+18>:  xor    %eax,%eax
5  0x007a1 <+20>:  movslq %esi,%rcx
6  0x007a4 <+23>:  movslq %edi,%rdx
7  0x007a7 <+26>:  lea    0x0(,%rdx,8),%rax
8  0x007af <+34>:  sub    %rdx,%rax
9  0x007b2 <+37>:  add    %rcx,%rax
10 0x007b5 <+40>:  lea    0x200884(%rip),%rdx # 0x201040 <
       arr>
11 0x007bc <+47>:  movl   $0x2,(%rdx,%rax,4)
12 0x007c3 <+54>:  movl   $0xfffffffe,(%rsp)
13 0x007ca <+61>:  movl   $0xffffffff,0x4(%rsp)
14 0x007d2 <+69>:  movl   $0x1,0x8(%rsp)
15 0x007da <+77>:  movl   $0x2,0xc(%rsp)
16 0x007e2 <+85>:  movl   $0x2,0x10(%rsp)
17 0x007ea <+93>:  movl   $0x1,0x14(%rsp)
18 0x007f2 <+101>: movl   $0xffffffff,0x18(%rsp)
19 0x007fa <+109>: movl   $0xfffffffe,0x1c(%rsp)
20 0x00802 <+117>: movl   $0x1,0x20(%rsp)
21 0x0080a <+125>: movl   $0x2,0x24(%rsp)
22 0x00812 <+133>: movl   $0x2,0x28(%rsp)
23 0x0081a <+141>: movl   $0x1,0x2c(%rsp)
24 0x00822 <+149>: movl   $0xffffffff,0x30(%rsp)
25 0x0082a <+157>: movl   $0xfffffffe,0x34(%rsp)
26 0x00832 <+165>: movl   $0xfffffffe,0x38(%rsp)
27 0x0083a <+173>: movl   $0xffffffff,0x3c(%rsp)
28 0x00842 <+181>: mov    $0x0,%eax
29 0x00847 <+186>: jmp    0x84c <vis_knight+191>
30 0x00849 <+188>: add    $0x1,%eax
31 0x0084c <+191>: cmp    $0x7,%eax
32 0x0084f <+194>: jg     0x89e <vis_knight+273>
33 0x00851 <+196>: movslq %eax,%rdx
34 0x00854 <+199>: mov    %edi,%r8d
35 0x00857 <+202>: add    (%rsp,%rdx,4),%r8d
36 0x0085b <+206>: mov    %esi,%ecx
37 0x0085d <+208>: add    0x20(%rsp,%rdx,4),%ecx
38 0x00861 <+212>: cmp    $0x6,%r8d
39 0x00865 <+216>: setbe  %r9b
40 0x00869 <+220>: mov    %ecx,%edx
41 0x0086b <+222>: not    %edx
42 0x0086d <+224>: shr    $0x1f,%edx
43 0x00870 <+227>: test   %dl,%r9b
44 0x00873 <+230>: je     0x849 <vis_knight+188>
45 0x00875 <+232>: cmp    $0x6,%ecx
46 0x00878 <+235>: jg     0x849 <vis_knight+188>
47 0x0087a <+237>: movslq %ecx,%rcx
48 0x0087d <+240>: movslq %r8d,%r8
49 0x00880 <+243>: lea    0x0(,%r8,8),%rdx
50 0x00888 <+251>: sub    %r8,%rdx
51 0x0088b <+254>: add    %rdx,%rcx
52 0x0088e <+257>: lea    0x2007ab(%rip),%rdx # 0x201040 <
       arr>
53 0x00895 <+264>: movl   $0x1,(%rdx,%rcx,4)
54 0x0089c <+271>: jmp    0x849 <vis_knight+188>
55 0x0089e <+273>: mov    0x48(%rsp),%rax
56 0x008a3 <+278>: xor    %fs:0x28,%rax
57 0x008ac <+287>: jne    0x8b3 <vis_knight+294>
58 0x008ae <+289>: add    $0x58,%rsp
59 0x008b2 <+293>: retq
```

### Assembly practice 4

```
1  Dump of assembler code for function even:
2  0x0066a <+0>:  test   %edi,%edi
3  0x0066c <+2>:  jne    0x674 <even+10>
4  0x0066e <+4>:  mov    $0x1,%eax
5  0x00673 <+9>:  retq
6  0x00674 <+10>: push   %rbp
7  0x00675 <+11>: push   %rbx
8  0x00676 <+12>: sub    $0x8,%rsp
9  0x0067a <+16>: mov    %edi,%ebx
10 0x0067c <+18>: lea    -0x2(%rdi),%edi
11 0x0067f <+21>: callq  0x66a <even>
12 0x00684 <+26>: lea    (%rax,%rax,1),%ebp
13 0x00687 <+29>: lea    -0x1(%rbx),%edi
14 0x0068a <+32>: callq  0x698 <odd>
15 0x0068f <+37>: add    %ebp,%eax
16 0x00691 <+39>: add    $0x8,%rsp
17 0x00695 <+43>: pop    %rbx
18 0x00696 <+44>: pop    %rbp
19 0x00697 <+45>: retq
20
21 Dump of assembler code for function odd:
22 0x00698 <+0>  push   %rbp
23 0x00699 <+1>  push   %rbx
24 0x0069a <+2>  sub    $0x8,%rsp
25 0x0069e <+6>  mov    %edi,%ebx
26 0x006a0 <+8>  cmp    $0x1,%edi
27 0x006a3 <+11> jne    0x6ae <odd+22>
28 0x006a5 <+13> mov    %ebx,%eax
29 0x006a7 <+15> add    $0x8,%rsp
30 0x006ab <+19> pop    %rbx
31 0x006ac <+20> pop    %rbp
32 0x006ad <+21> retq
33 0x006ae <+22> lea    -0x2(%rdi),%edi
34 0x006b1 <+25> callq  0x698 <odd>
35 0x006b6 <+30> lea    (%rax,%rax,1),%ebp
36 0x006b9 <+33> lea    -0x1(%rbx),%edi
37 0x006bc <+36> mov    $0x0,%eax
38 0x006c1 <+41> callq  0x66a <even>
39 0x006c6 <+46> lea    0x0(%rbp,%rax,1),%ebx
40 0x006ca <+50> jmp    0x6a5 <odd+13>
41
42 Dump of assembler code for function main:
43 0x0065c <+0>:  sub    $0x8,%rsp
44 0x00660 <+4>:  mov    $0x9,%edi
45 0x00665 <+9>:  callq  0x628 <odd>
46 0x0066a <+14>: mov    $0x0,%eax
47 0x0066f <+19>: add    $0x8,%rsp
48 0x00673 <+23>: retq
```

### Assembly practice 1 answer

```c
#include<stdio.h>
int memo[20];
int fib(int a){
    if(a==1) return 1;
    if(memo[a]!=-1) return memo[a];
    else{
        memo[a] = fib(a-1) + fib(a-2);
        return memo[a];
    }
}
int main(){
    for(int i=0;i<20;i++){
        memo[i]=-1;
    }
    fib(10);
}
```

### Assembly practice 2 answer

```c
node* add_node(int i, node * head, int value){
    node * cur = head;
    if(i==0){
        node new_node = {value, NULL, head};
        head->prev = &new_node;
        return &new_node;
    }
    int index = 0;
    while(index < i){
        cur = cur->next;
        index++;
    }
    node new_node = {value, cur->prev, cur};
    cur->prev->next = &new_node;
    cur->prev = &new_node;
    return head;
}
```

### Assembly practice 3 answer

```c
void vis_knight(int i, int j){
    arr[i][j] = 2;
    int xs[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
    int ys[8] = {1, 2, 2, 1, -1, -2, -2, -1};
    for(int k=0;k<8;k++){
        int newx = i + xs[k];
        int newy = j + ys[k];
        if(newx >=0 && newx < 7 && newy >=0 && newy < 7){
            arr[newx][newy] = 1;
        }
    }
}
```

### Assembly practice 4 answer

```c
#include<stdio.h>

int odd(int n){
    if(n==1) return 1;
    return 2 * odd(n-2) + even(n-1);
}

int even(int n){
    if(n==0) return 1;
    return 2 * even(n-2) + odd(n-1);
}

int main(){
    odd(9);
}
```