Table of contents

1: C programming
   - Declarations
   - Macros
   - Operators
   - Types
   - Pointers
   - Arrays & Strings
   - Complex data structures
   - Dynamic allocation
   - Memory allocators
   - Garbage collecting
   - Advanced control flow

2: Assembly language and compiled code
   - Tools & Basics
   - Addressing modes
   - Moving data
   - Address computation
   - Arithmetic operations
   - Jumps
   - Virtual memory layout
   - Procedure calls and return
   - Stack overflow
   - Compile control flow
   - Data representation
   - Linking
   - Libraries

3: Performance optimization
   - gcc flags
   - Compiler optimizations
   - Optimization blockers
   - Blocking and unrolling
   - Instruction-level parallelism
   - Reassociation
   - Performance
   - SSE3
   - AVX2

4: Computer architecture
   - Caches
   - Exceptions
   - Virtual memory
   - Multiprocessing
   - Devices
1: C programming

Declarations

- inside a block: scope is that block
  - `static` -> value persists between calls
- outside a block: scope is entire program
  - `static` -> scope limited to file
- `x = foo()` is an expression
  - it evaluates to the assigned value

**Note:** Dynamically allocated values are on stack and `static` & global values get allocated when program is loaded.

Declaration vs. definition

- definition is the actual implementation
- declaration says somewhere it exists (prototype -> often headerfile)
  - `extern` -> definition is somewhere else (even other compilation unit)
  - `static` -> definition (also `static`) is in this compilation unit & can't be seen outside of it

**Example**

```c
// file_a.c:
extern const char *motd; // defined in another_file.c
static int private_count; // defined in this file

// file_b.c:
static int private_count = 0; // only in scope of this unit

// another_file.c:
const char *motd = "Smoking is one of the leading causes of statistics.\n\t	-- Fletcher Knebel";
```

Modules

A module is a self-contained piece of a larger program, it consists of:

- externally visible (interface):
  - "public" functions
  - typedefs & global variables
  - `cpp` macros
- not visible:
  - internal functions
  - internal types
  - global variables not visible to client

C headers

- specify the interface, only external declarations
- module `foo` has the interface `foo.h`
- clients #include "foo.h"
- implementation/definition usually in `foo.c`
  - it includes `foo.h`
  - contains only definitions and internal declarations (no external declarations)

C namespaces

1. **Label names**
   - used for `goto`
2. Tags
   - one namespace for all struct s, union s & enum s

3. Member names
   - one namespace for each struct, union & enum

4. Everything else
   - types including typedef etc.

---

Macros

```cpp
#define abc defg
```

With this keyword one can define a cpp directive to replace abc with defg.

Examples

```cpp
// brackets ensure precedence:
#define min(X,Y) ((X) < (Y) ? (X) : (Y))

// multiline possible with escaping
#define DEBUG_BREAK(a)\
  if ((a)) \
    __asm int 3;
```

```cpp
if and ifdef

#define SOME_MACRO
some_code_or_macro();
#endif
```

With this construct it's possible to define macros or insert code depending on whether SOME_MACRO is defined. There's also #ifndef to see if something hasn't been defined and #else to react with an alternative.

---

Operators

List of operators by decreasing precedence (associativity is left-to-right):

- `()` `[]` `.`
- `| ~ ++ -- *` (type) `sizeof` (Right-to-left)
- `/ %` `+ -`
- `<< >>` `<= >=` `== !=` `&` `^` `|`
- `&& ||` `?:` (Right-to-left)
- `= += -= *= /= %= &= ^= |= <<= >>=` (Right-to-left)

Note:

- `i++` has the current value of `i` and `++i` the new one
• \(x \gg y\) shifts right by \(x\)
  ▶ for unsigned: logical right shift
  ▶ for signed: arithmetic right shift
• shifting by negative amount: undefined behaviour
• shifting by amount >= word size: undefined behaviour

Types

Type sizes

The size depends on the architecture, use `sizeof(type)` to find out.

<table>
<thead>
<tr>
<th>type</th>
<th>size (x86_64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>10/16</td>
</tr>
</tbody>
</table>

• int s are signed by default
  ▶ use unsigned to clarify
• **Booleans**: zero -> False & non-zero -> True
  ▶ \(!\) is negation operator (\(!0\) is not clear)
  ▶ void -> False
• casting (usually) doesn't change bit-representation
  ▶ usage: signed int i = (signed int) foo();

Integers

Signed vs. unsigned

The following table assumes \(w=16\):

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>32768</td>
<td>80 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
</tr>
</tbody>
</table>

• \(|\text{TMin}| = \text{TMax} + 1\)
• \(\text{UMax} = 2^w - \text{TMax} + 1\)
• signed integers are represented as two's complement
• \(-x + 1 == -x\)
• constants are by default signed
  ▶ else you'd write \(42u\)
• mixed signed & unsigned values: signed values get casted to unsigned
• assigning a signed value to unsigned type: value is implicitly casted
• C automatically sign-extends when casting signed values \((\text{new}_w > w)\)
• C truncates the bit-pattern when casting (new _w < w)
  ◦ unsigned values: modulus operation
  ◦ signed values: not really expected behaviour
• casting signed <-> unsigned: sometimes 2^w will get added/subtracted!

Example

```c
int max = INT_MAX;  // 2147483647
short u = max;      // -1
unsigned short s = max; // 65535
```

Addition

unsigned :

• we just ignore (w+1)-th bit: addition is modular (mod 2^w)
• addition forms an abelian group
  ◦ closed, commutative, associative
  ◦ 0 is the identity
  ◦ inv(u) = 2^w - u

signed :

• the bit-level operation is equivalent to unsigned addition
  ◦ we just drop the (w+1)-th bit
• it also forms a group (isomorphism), but:
  ◦ inv(u) = -u if u != TMin and TMin if u == TMin

Note: integer addition is associative

• whereas float addition is not associative

Multiplication

unsigned :

• we would require 2w bits for the result
• it's again modular (mod 2^w)
• together with addition it forms an abelian ring:
  ◦ multiplication is associative & commutative
  ◦ 1 is the multiplicative identity
  ◦ multiplication distributes over addition

signed :

• isomorphic algebra to unsigned ring
• doesn't obey ordering properties of integers
  ◦ TMax + 1 == TMin
  ◦ 15213*38426 == -10638 for w=16 (short)

Pointers

• pointer is address in virtual memory
  ◦ 32-bit -> 2^32 bytes (thus 32-bit address)
  ◦ 64-bit -> 2^64 bytes (thus 64-bit address)
• &x returns the address of where x is stored
  ◦ to declare: int *ptr
  ◦ to assign: ptr = &x
• dereference operator * :
  ◦ v = *ptr (v has value of x)
  ◦ *ptr = 42 (x is now changed)
• **NULL** is invalid (segfault), its type is **void** *
• by adding an int n the pointer points n **objects** further
  • new value of p depends on the type of “p

**Pass-by-value**

C passes function parameters by value, callee gets a **copy** of it, use pointers to modify variable in scope of caller:

```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
// ...
swap(&x, &y);
// ...
```

**Function pointers**

```c
int (*func)(int *, char);
```

• func is a pointer to a function returning an int and taking an int * and a char as argument
• function pointers can (as always) also be typedef’d

**Example**

```c
#include <stdlib.h>
// void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void*, const void*))
// base: pointer to first element
// nitems: number of elements in array
// size: byte count of one element
// compar: function pointer to function that compares two elements

int cmp(const void *a, const void *b) {
    int x = *((const int *) a);
    int y = *((const int *) b);
    if (x < y) return -1;
    else if (x > y) return 1;
    return 0;
}
// ...

int arr[] = {49, -38, 20, 19, 101};
qsort(arr, 5, sizeof(int), &cmp);
```

---

**Arrays & Strings**

**Arrays**

• compiler does **not** check bounds!
• initialization:
  • int vals[24];
  • char matrix[3][4];
  • int a[] = {1, 2, -12, 8};
  • float b[12] = {0}; (zero initialized)
• the first element has lowest address!
• the array name is a pointer to first element, except when:
  • you use & to get address
  • array is string initializer
  • array is operand of sizeof
• A[i] is always rewritten to *(A+i) by compiler
array (also function) name as function parameter is pointer, these are equivalent:

- int arr_fct( int *arr ) { ... }
- int arr_fct( int arr[] ) { ... }
- int arr_fct( int arr[12] ) { ... }

Example (none of them will fail):

```c
assert( &vals == vals );
assert( vals == &vals[0] );
assert( sizeof(vals) == 24*sizeof(int) );
assert( sizeof(&vals[0]) == sizeof(int *) );
assert( vals[3] == 3[vals] );
```

Strings

- use a char array terminated with \0:
  - char str[4] = {'a', 'b', 'c', '\0'};
  - is equivalent to char str[4] = "abc";

Example

```c
printf("Hello\n"); // output: ello (pointer arithmetic)
```

Complex data structures

Enums

TODO

Structures

```c
// Point is a (struct point_st)
// Point is a (struct point_st *)

typedef struct point_st {
    type value;
    // ...
} Point, *PointPtr
```

- instances of a struct can be allocated on the heap or stack
- point_st is called tag
- use . to refer to a field in of a struct
- use -> to refer to a field of a pointer to a struct
- can assign value of struct to another of the same type (copies it)
- can use a struct as argument of function (passed by value as always)
- can use it as a return type as well

Unions

```c
typedef union val_u {
    int ival;
    float fval;
} CVal;
```

- only holds one value of a set of alternatives
- accessed like a struct
- used to access bit-pattern of other values

Generic datastructures

- to implement a generic datastructure use void * as element
client needs to cast the types to (void *) before inserting etc.
client needs to cast the element back to actual type from void *

Example

typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

// Usage:
Node *push(Node *head, void *element) {
  Node *n = malloc( sizeof(Node) );
  if (!n) { exit(-1); }
  n->element = element;
  n->next = head;
  return n;
}

// ...
int *el_1 = malloc( sizeof(int) );
int *el_2 = malloc( sizeof(int) );

Node *list = NULL;
*el_1 = 19;
*el_2 = -84;
list = push(list, (void *) el_2);
list = push(list, (void *) el_1);

printf("First element is: \x0A", *{(int *) list->element} );
printf("Second element is: \x0A", *{(int *) list->next->element} );

Dynamic allocation

With xalloc data gets put on the heap and persists until we explicitly free the memory. xalloc returns a pointer to the first byte of memory and NULL if memory couldn't be allocated.

Allocation

void *malloc(size_t sz)

- allocates sz bytes
- example: float *arr = (float *) malloc(10*sizeof(float));

void *calloc(size_t nm, size_t sz)

- allocates nm*sz bytes
- the memory gets zeroed
- example: float *arr = (float *) calloc(10, sizeof(float));

Deallocation

void free(void * ptr)

- ptr must point to first byte of previously allocated memory
- always call free when done -> memory leaks lead to virtual memory thrashing

Reallocation

void *realloc(void *ptr, size_t sz)

- changes size of memory to sz
- ptr must point to first byte like with free
Memory allocators

The `brk` pointer points to the top of the heap (see virtual memory layout), if an allocator (e.g. `malloc` from `stdlib.h`) needs more space on the heap, it'll use `sbrk(amount)` to request more from the OS.

Informal specification

Applications

- can issue arbitrary sequence of `malloc()`, `realloc()` and `free()` requests
- must use `malloc()`'d block to call `free()` on

Allocator

- must respond immediately to requests (can't buffer them)
- must allocate free memory according to alignment rules
- isn't allowed to move allocated memory

Goals

Some definitions:

- Aggregate payload $P_k$: sum of all allocated memory after $k$ requests
- Current heap size $H_k$: number of bytes on heap (monotonically non-decreasing)
- Peak memory utilization $U_k$: is defined as $(\max_{i<k} P_i) / H_k$
- Internal fragmentation: bytes not used in an allocated block (payload < block size)
  - caused by:
    - overhead for maintaining heap data structure
    - policy decision (e.g. return big block, no matter what)
    - padding for alignment
  - external fragmentation: aggregate heap memory that can't be used because consecutive blocks are too small
  - caused by:
    - unfortunate request patterns

The goal is: Given some sequence $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$ of requests, the allocator should maximize throughput (requests processed per time) and peak memory utilization.

Possible implementations

Implicit list

Introduce a header and boundry tag, that is a word at the beginning and end respectively that marks the size of it. To keep track of whether the block is allocated or not, we can use the last bit of those words. This way we have an implicit list that links all blocks (allocated ones as well).

- when searching for a free block there are different policies:
  - first fit: choose the first block that is large enough
  - next fit: like first fit but continue searching where we stopped last time
  - best fit: search the whole list and choose the one with least internal fragmentation
- to reduce internal fragmentation one might want to split the block
- coalescing is the process of merging neighbouring free blocks on a `free()` request
  - coalesce the block with the neighbouring ones is done by adding block sizes (header/boundry)

Costs:

- `free()` & coalesce in $O(1)$
- `malloc()` in $O(n)$ where $n = H_k$

Explicit list

Like in the implicit list we use a header and boundry but additionally use pointers (next and previous) in free blocks to
connect them, this way we have only free blocks in our list.

When inserting a block in list on `free()` request there are different policies:

- **LIFO:** insert the block at the beginning
  - allows $O(1)$ insertion
  - can cause more fragmentation
- **address-ordered:** insert block such that the addresses are ordered
  - supposed to have lower fragmentation
  - requires searching

**Costs:**

- `free()` & coalesce in $O(1)$
- `malloc()` in $O(f)$ where $f$ is the number of free blocks

**Seglist**

This is an implementation with a segregated list, meaning that we have different lists for free blocks of different size classes.

- often use separate classes for small sizes and big classes for each two-power size
- to allocate a block search the appropriate class, if empty search in the next.
  - if block is found: split it and put the remaining part into its class
  - if no block is found: call `sbrk(n)` and place it in the largest class

**Costs:**

- `free()` & (optional) coalesce in $O(1)$
- `malloc()` in $O(\log(n))$ -> higher throughput

**Blocks sorted by size**

Implement a balanced search tree with block sizes as keys to track free ones, this has the same advantage like the seglist implementation but is a lot more work to do.

**Garbage collecting**

A garbage collector keeps a view of the memory as a directed graph in terms of pointers and blocks, special nodes - called roots - are blocks that aren't in the heap. If there is a component in the graph that is not reachable by a root, the GC can free this memory.

**Mark and sweep collecting**

This implementation builds on top of `malloc/free`; we allocate memory using `malloc()` until we run out of space. When this happens we use an extra `mark bit` in the head of each block, then we:

- **mark:**
  - start at roots and set mark bit on each block that is reachable
  - scan all blocks and `free()` if it's not reachable

For this to work we need (for example) a balanced search tree of allocated blocks to determine if a pointer points to an allocated block. This can be realized by storing two additional words per block - `left` and `right` child in the tree.

**Advanced control flow**

Besides loops (`while (expr) { }`, `do {} while (expr);` or for (`init;test;inc`) `{ }`) and conditions (`if`..`else` if..`else or case`) there are other types of control flow in C.

**goto**

- don't use it, unless you have to!
- it's fast, but switches are nearly as fast
useful for:
- early termination of nested loops without lots of flag checking
- nested clean up code
- low-level code (eg. libraries or kernel code)

Example (nested clean up code)

```c
try(1); if (failed) goto undo_1;
try(2); if (failed) goto undo_2;
// ...
try(N); if (failed) goto undo_N;
return 0;

undo_N: undo(N); err = N;
// ...
undo_2: undo(2); err = 2;
undo_1: undo(1); err = 1;
return err;
```

**setjmp() & longjmp()**

```c
#include <setjmp.h>
// int setjmp(jmp_buf env);

- saves the current stack state (stack frame) in env buffer
- returns 0 on success

// void longjmp(jmp_buf env, int val);

- jumps back to the point saved by env once again
- this time it will return val (or 1 if val is equal to 0 )
- this can only be done if the code containing setjmp didn't return (think about stack)

**Note:** To get defined behaviour, the ISO C99 standard limits the usage of setjmp to:
- as a condition in an if - or switch -statement
- as the above by using ! or an integer comparison
- as single statement (not using its return value)

Example

```c
// This program will output:
// second
// main

static jmp_buf buf;       // use a global buffer to allow "non-local jump"

void second(void) {
    printf("second\n");
    longjmp(buf,1);       // jumps back to main, but this time it returns 1
}

void first(void) {
    second();
    printf("first\n");
}

int main() {
    if (!setjmp(buf)) {   // save current context to env
        first();
    } else {
        printf("main\n");
    }
```
Coroutines

A coroutine is like a thread that is not concurrent, meaning it has its own execution environment and can be scheduled to be executed later (context switch). It can suspend its execution and give control to another coroutine which in turn can resume that coroutine.

Example usage: P/C-queue with a single thread (eg. Lexer/Parser) or implementing a generator.

```c
typedef struct coro_st {
    void *stack;
    jmp_buf env;
} Coro;

// constructor, destructor, etc.
void Coro_switchTo(Coro *self, Coro *next) {
    if ( !setjmp(self->env) ) {
        longjmp(next->env, 1)
    }
}

* when Coro_switchTo gets called:
  * first return from setjmp will longjmp to the next coroutine
  * the second time we continue where we called Coro_switchTo
```
2: Assembly language and compiled code

Tools & Basics

GNU toolchain

• to get assembly code from a C program use: gcc -O -S code.c
• to disassemble an object file or executable: objdump -d code.o

radare2 - Reversing Framework

Basic commands

• to start issue: r2 a.out
• aaa will analyze the whole binary
• to view disassembled code issue v (switch views with space)
• s sets the cursor, example s main
• afl lists all functions
• pdf prints disassembled code  
  ¨ pdf@label prints the code after label.example pdf@main
• to open visual mode issue v
  ¨ p : next screen
  ¨ q : quits
  ¨ . : goto %rip
• px/4x@%rip will hexdump the memory at %rip

Basic debugging with r2:

• to start issue: r2 -d a.out -AAAA ( -AAAA will do the analysis beforehand)
  ¨ to pass arguments: r2 -d darun2 program.a.out arg1=$(python exploit.py)
  ¨ to pass stdin: r2 -d darun2 program.a.out stdin=$(python exploit.py)
• db main sets breakpoint at main
• db* > file saves all breakpoints to file
  ¨ to load it: . file
• dc continues/stars execution
• s single step and S step-over
• drt all dumps all registers (single register: dr?rax
• dr eax = 0x1 sets register value
• s [address] sets the instruction pointer to [address]
• wa [instruction] rewrites the code at instruction pointer

Note: For more information see here.

>> cat ~/.radare2rc
$decompile=$pipe node /PATH/TO/r2-scripts/decompiler/decompile.js
e asm.cmtright = true
e cmd.stack = true
e asm.describe = true
e asm.syntax = att
e scr.utf8 = true
eco smyck
e scr.pipcolor = true
e scr.pager = less -R

x86_64 architecture

• each instruction is either 1,2 or 3 bytes in size.
• uses 64-bit (8 byte) words
• supports 48-bit addresses (space: 256TB)
• addresses specify byte locations (next word’s addr. is +8)
• uses little endian byte ordering (LSbyte has lowest address):
  ◦ add $0x12ab,$ebx translates to instruction code 81 c3 ab 12 00 00
• integer data consists of 1,2,4 or 8 bytes
• floating point data consists of 4,8 or 10 bytes
• most instructions form a family of four, for example movX where X equals
  ◦ b -> 1 byte
  ◦ w -> word = 2 bytes
  ◦ l -> long word = 4 bytes
  ◦ q -> quad word = 8 bytes

Thus we can access the registers in chunks:

<table>
<thead>
<tr>
<th>64-bit register</th>
<th>Lower 32 bits</th>
<th>Lower 16 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>al</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bl</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>cl</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>si</td>
<td>dl</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>di</td>
<td>dl</td>
</tr>
<tr>
<td>rbp</td>
<td>ehp</td>
<td>bp</td>
<td>bpl</td>
</tr>
<tr>
<td>esp</td>
<td>esp</td>
<td>sp</td>
<td>fp</td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td>r8w</td>
<td>r8b</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td>r9w</td>
<td>r9b</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
<td>r10w</td>
<td>r10b</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
<td>r11w</td>
<td>r11b</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
<td>r12w</td>
<td>r12b</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
<td>r13w</td>
<td>r13b</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
<td>r14w</td>
<td>r14b</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td>r15w</td>
<td>r15b</td>
</tr>
</tbody>
</table>

Limitations:
• D constant displacement value (1,2 or 4 bytes)
• k can only be 1, 2, 4 or 8
• Ra any 64, 32, 16 or 8-bit register
- *Rb* any 64-bit register
- *Ri* any 64-bit register except `%rsp`

**Moving data**

`mov X Src, Dest`

Move bytes from *Src* to *Dest* according to *X*.

**Operand types:**

- **Immediate:** constant integer data
  - like C constant but prefixed with `$`
  - encoded with 1, 2, 4 or 8 bytes
  - example: `$0x400`, `$-533`
- **Register:** one of 16 integer registers
  - example: `%ax`, `%rbx`
- **Memory:** 1, 2, 4 or 8 consecutive bytes in memory
  - see different address modes
  - example: `(Rax)`

The second operand can be any of *Register* or *Memory*. Unless the first operand is *Memory*, then you can only choose *Register*. *Mem -> Mem transfer not possible!*

`movz XY Src, Dest`

Zero-extends *Src* and moves it to *Dest*; the *Dest* can only be a register and *Src* can be either a memory location or a register.

- *X* and *Y* denote one of the word sizes but *X* needs to be smaller than *Y*.
  - `movzlq` isn't valid since `movl` will zero-extend if second operand is a register
  - eg. `movzbq %al, %rbx`
- `cltq` is a shorter encoding for `movslq %eax, %rax`

`movs XY Src, Dest`

Sign-extends *Src* and moves it to *Dest*. Usage is the same as with `movz XY` family (except `movslq` is valid).

**Address computation**

`lea Src, Dest`

Set *Dest* to address mode expression *Src*. (It doesn't fetch the memory content, only computes the location!)

**Arithmetic operations**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add X Src, Dest</code></td>
<td><em>Dest</em> &lt;- <em>Dest</em> + <em>Src</em></td>
</tr>
<tr>
<td><code>sub X Src, Dest</code></td>
<td><em>Dest</em> &lt;- <em>Dest</em> - <em>Src</em></td>
</tr>
<tr>
<td><code>imul X Src, Dest</code></td>
<td><em>Dest</em> &lt;- <em>Dest</em> * <em>Src</em></td>
</tr>
<tr>
<td><code>sal X Src, Dest</code></td>
<td><em>Dest</em> &lt;- <em>Dest</em> &lt;&lt; <em>Src</em></td>
</tr>
<tr>
<td><code>sar X Src, Dest</code></td>
<td><em>Dest</em> &lt;- <em>Dest</em> &gt;&gt; <em>Src</em> (arithmetic)</td>
</tr>
</tbody>
</table>
**Format**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shrX Src, Dest</td>
<td>Dest &lt;- Dest &gt;&gt; Src (logical)</td>
</tr>
<tr>
<td>xorX Src, Dest</td>
<td>Dest &lt;- Dest ^ Src</td>
</tr>
<tr>
<td>andX Src, Dest</td>
<td>Dest &lt;- Dest &amp; Src</td>
</tr>
<tr>
<td>orX Src, Dest</td>
<td>Dest &lt;- Dest</td>
</tr>
<tr>
<td>incX Dest</td>
<td>Dest &lt;- Dest + 1</td>
</tr>
<tr>
<td>decX Dest</td>
<td>Dest &lt;- Dest - 1</td>
</tr>
<tr>
<td>negX Dest</td>
<td>Dest &lt;- -Dest</td>
</tr>
<tr>
<td>notX Dest</td>
<td>Dest &lt;- ~Dest</td>
</tr>
</tbody>
</table>

**Condition codes**

There are 4 single bit registers that get set as a side-effect when performing an arithmetic instruction:

- **CF**: carry flag (-> unsigned overflow)
- **ZF**: zero flag (-> result is zero)
- **SF**: sign flag (-> result < 0 for signed)
- **OF**: overflow flag (-> signed overflow)

**Note**: Since lea isn't an arithmetic instruction, they won't get set!

You can explicitly set the flags by using:

- **cmpX b,a**: as if a-b would get issued but doesn’t set a.
  - all flags get set
- **testX b,a**: as if a&b would get issued but doesn’t set a.
  - OF and CF will get set zero (not interesting anyway...)

To read those flags you can use one of the set?? instructions, they all have a single operand that needs to be an 8-bit register (eg. %al). Don’t forget to clear the top bits of that register! Because set?? will only set the lowest bit - the remaining 7 bits won’t get changed!

**Some instructions:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>setl</td>
<td>setnge</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>sete</td>
<td>setz</td>
<td>equal/zero</td>
</tr>
<tr>
<td>setne</td>
<td>setnz</td>
<td>not equal/not zero</td>
</tr>
<tr>
<td>setg</td>
<td>setnle</td>
<td>greater (signed &gt;)</td>
</tr>
</tbody>
</table>

**Example**

```c
; int gt (long x, long y) { return x > y }
; %rdi = x
; %rsi = y
; return value in %rax

cmpq %rsi, %rdi ; not the order!
setg %al ; set LSByte of %rax to [x > y]
movzbq %al, %rax ; zero the rest of %rax
```

**Jumps**

Conditional jumps depend on the condition codes, some examples listed below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Condition</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>jmp Label</td>
<td>Direct jump</td>
<td></td>
</tr>
<tr>
<td>jmp *Operand</td>
<td>Indirect jump</td>
<td></td>
</tr>
<tr>
<td>je Label</td>
<td>Equal</td>
<td>ZF</td>
</tr>
<tr>
<td>jne Label</td>
<td>Not equal</td>
<td>~ZF</td>
</tr>
<tr>
<td>jg Label</td>
<td>Greater (signed)</td>
<td>~(SF^OF) &amp; ~ZF</td>
</tr>
</tbody>
</table>

- `Label` is either an absolute or PC-relative address (must fit in 4 bytes)
- `Operand` can be any form of addressing mode

### Virtual memory layout

The virtual memory layout looks like this (note: this is for a 32-bit system!):

- **Kernel space**: User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault
- **Stack (grows down)**
- **Memory Mapping Segment**: File mappings (including dynamic libraries) and anonymous mappings. Example: `/lib/libc.so`
- **Stack growth mechanism**: Random stack offset
- **Program segment**: Random mmap offset
- **Heap**
- **Text segment (ELF)**: Stores the binary image of the process (e.g., `/bin/gerbino`)

The stack "grows" from higher address space to lower one. There are two important registers:

- `%rsp`: stack pointer holds address of the "top" of the stack
- `%rbp`: stack base pointer holds address of the current stack frame

### Procedure calls and return

- `push? Src`
  Fetches the operand at `Src`, decrements `%rsp` by 4 and writes the operand at address `%rsp`.

- `pop? Dest`
  Reads operand at address `%rsp`, increments `%rsp` by 4 and writes the operand to `Dest`.

- `call Label`
Pushes return address (the address of instruction after call) to stack and jumps to Label.

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>08048553</td>
<td>49 89 d5</td>
<td>mov %rdx, %r13</td>
</tr>
</tbody>
</table>

After executing the call instruction, the stack will look like this:

- 0x10c: ...
- 0x108: [previous stuff]
- 0x100: 0x8048554

The stack pointer (%rsp) is 0x100 and %rip = 0x8048b90.

ret

Pop address from stack and jump to that address.

Calling conventions

- caller saved: %r10 and %r11
- callee saved: %rbx, %rbp and %r12 - %r15

Arguments & return values

The x86_64 convention allows up to six registers to transfer arguments, these are:

- %rdi
- %rsi
- %rdx
- %rcx
- %r8
- %r9

If a procedure takes more than six arguments the remaining values are pushed onto the stack (in the opposite order!) before calling the procedure. When pushing arguments onto the stack the data sizes are rounded up to 8 bytes.

The stack might look like this (note: it includes local variables etc.):

```plaintext
Example

; int funct(int a1, int a2, char b1, char b2, char b3, char b4, // passed in registers
;           short *d1, short d2);                          // passed via stack
;```
### Base pointer

The `%rbp` is used when one needs to setup a stack frame for the current context. The reason is because the `%rsp` changes implicitly, this way one can get arguments, saved registers or local variables without thinking about recent pushes and pops.

To setup a stackframe:

- save old `%rbp` by pushing it on the stack
- create a new `%rbp` by setting it to `%rsp`
- do work and clean up stack
- pop the old `%rbp` from stack to restore previous stack frame

### Tail call optimization

When a procedure (non-leaf) doesn't need to keep track of registers, one can modify it to not setup a stack frame, this can save a lot of pushs/pops. This kind of optimization is applicable whenever the code uses `return some_function()` or can be rewritten to that form.

### Example

```assembly
; void swap_neighbours(long arr[], int i); // swaps a[i] and a[i+1]
; <swap> exchanges the elements at (%rdi) and (%rsi)

movslq %esi, %rsi          ; sign-extend long word (int) to pointer
leaq (%rdi,%rsi,8), %rdi   ; %rdi = &a[i]    (long is 8 bytes)
leaq 8(%rdi), %rsi         ; %rsi = &a[i+1]
jmp swap                   ; instead of calling swap and then return
```

### Stackoverflow

If a programmer isn't careful, they might forget to check bounds in a function and introduce a vulnerability - a buffer-/stackoverflow. This happens when they write to a buffer (on the stack) without checking the bounds. An attacker can write a payload to the buffer and overwrite the return address to that of the payload:

#### Possible protections

A possible workaround is the technique of ROP. The bottomline is to code defensively (eg. use `strncpy` instead of `strcpy`).
etc.) because one can get around those protections:

- randomize the stack layout (default on Linux)
- NX flag
- insert stack canaries

To analyze an executable for these kind of things, you can use `checksec.sh`.

---

**Compile control flow**

**Conditional expression**

First translate the code to a `goto`-version:

```c
// evaluate the test
nt = !test_exp;
if (nt) goto Else;
// do the if-part
goto Done;
Else:
// do the else-part
Done:
// ...
```

**Conditional move**

```c
cmovC Src, Dest
```

Moves `Src` to `Dest` if condition holds. This is more efficient than conditional branching but keep in mind that both parts get evaluated.

**Example (absolute difference)**

```c
; %rdi = x
; %rsi = y
; return value in %eax
movq %rdi, %rax
subq %rsi, %rax ; %rax = x-y
movq %rsi, %rbx
subq %rdi, %rbx ; %rbx = y-x
cmpq %rsi, %rdi ; compare x and y
cmovle %rbx, %rax ; move (y-x) to %rax if x <= y
```

**Do-while loop**

Again, first transform the loop to a `goto`-version and then to assembly:

```c
int factorial(int x) {
    int res = 1;
    do {
        res *= x;
        x--;
    } while (x>1);
    return res;
}
```

Translated to assembly:

```c
; %edi = x
; return value in %eax
```
movl $1, %eax  ; res  = 1
Loop:
imull %edi, %eax  ; res *= x
decl %edi        ; x   -= 1
cmpl $1, %edi  ; jump if (x>1)
jg    Loop        ; jump if (x>1)
rep ret           ; return

While-loop
This construct is easiest translated by first translating it to a do-while -loop and then proceed as usual.

while (test_expr) {  
    // loop body
    }

Intermediate goto -version:

    if (!test_expr) goto Done;
Loop:  
    // loop body
    if (test_expr) goto Loop;
Done:

Another kind of transformation is to jump over the first body evaluation (-O2 flag with recent gcc), this is done because unconditional jumps are pretty cheap compared to unconditional ones.

int factorial_goto_middle(int x) {
    int res = 1;
goto Middle;
Loop:
    res *= x;
x--; 
Middle:
    if (x>1) goto Loop;
return res;
}

For-loop
A for loop is easiest translated by transforming it to a while -loop and then proceeding as usual:

    Init;
for (Init; Test; Update) {  
    // for body
    }  
    Update;

The (unoptimized) goto -version would look like this:

    Init;
    if (!Test) goto Done;
Loop:  
    // for body
    Update;
    if (Test) goto Loop;
Done:

Switch statements
Consider the following code (note 4 is missing and no break at 2):
switch (x) {
    case 1: do_1; break;
    case 2: do_2;
    case 3: do_3; break;
    case 5:
        case 6: do_56; break;
    default: do_default;
}

return res;

The compiler will generate a jump-table that looks something like this:

```
.section .rodata
.align 8
.align 4
.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
```

- **Direct jump**: jmp .L8 -> unconditional jump to .L8
- **Indirect jump**: jmp *.L4(,%rdi,8) -> jumps to label where x = %rdi
  - scaling by 8 is needed because the labels are 8 bytes

The assembly code would look something like this:

; %rdi = x
Switch:
  cmpq $6, %rdi
  ja   .L8            ; jump (above?) to default
  jmp  *.L4(,%rdi,8)  ; jump to table[x]
; ....
.L5:
    ; x = 2
    ; do_2
    jmp .L6             ; fall through (to x = 3)
; ....
.L7:
    ; x = 5 or x = 6
    ; do_56
    ret                 ; break -> return (because after switch there's a return!)
; ....

This could look like this when disassembled:

```
>>> Assuming .L8 becomes 0x0000000000400521
>>> and .L4 becomes 0x00000000004005b8
>>> note: 0x4ed+0x34 = 0x521
>>>
>>> objdump -d p switch.out
... 00000000004004ed <Switch>:
... 00000000004004ed: 77 2b                   ja    400521 <Switch+0x34>
00000000004004ed: ff 24 fd b8 05 40 00    jmpq  *0x4005b8(,%rdi,8)
```

**Note**: If the switch statement doesn't consist of nearby values, the jump table could become huge. An obvious solution is to use if-statements but this would require O(n) unconditional jumps. To get around this the compiler builds a decision tree
and cuts this cost down to $O(\log(n))$.

## Data representation

- Integer data is stored and operated on in (integer) registers
- Floating point data is stored and operated on in floating point registers

## Arrays

- Array \((T \ A[L])\) is a contiguously allocated region of \(L \times \text{sizeof}(T)\) bytes
  - \(A\) can be used as pointer to the first element
  - \(\text{movx}\) instruction provides a nice way for access;
    - suppose \(%rdi = A\) and \(%rsi = i\) then \(%eax = A[i]\) would be:
      \[
      \text{movl} (%rdi, %rsi, 4), %eax
      \]
    \(\text{in this case sizeof}(T) == 4\)

### Example (array access):

```asm
; expects: int A[5] (only digits)
; returns: int(''.join(map(str,A)))

zip_ints:
    movl $0, %edx          ; i = 0 (zeroes-out top 4 bytes by default)
    movl $0, %eax          ; res = 0
.L1:
    leal (%rax,%rax,2), %ecx  ; %ecx = 5 * %rax
    movl (%rdi,%rdx,4), %eax  ; %eax = A[i]
    leal (%rax,%rcx,2), %eax  ; %eax = %eax + %rcx * 2
    addq $4, %rdx           ; i++
    cmpl $20, %rdx
    jne  .L1                ; if (i != 20) goto .L1
ret
```

- Ragged arrays != array... they are just arrays of pointers.
  - With int A[[]] = { {1,9}, {48, 101,-12} } the value A[0][3] is undefined
    because \(A\) is an int * array (that itself is contiguous in memory)

## Structures

- Structures get contiguously allocated in memory
- Compiler inserts gaps to ensure these alignments:
  - char: 1 byte
  - short, etc.: 2 bytes
  - And so on: the types must be aligned corresponding to their size
- The overall size of the \texttt{struct} must be a multiple of \(k\)
  - Where \(k\) is largest type available on system (Linux x64: \texttt{sizeof(double)} == 8)
- This way all the elements in arrays of \texttt{struct}s are correctly aligned

## Unions

- Allocate a block of memory according to largest type in it

## Floating point

- Encoding: \([ s | \text{exp} | \text{frac} \] )
- Single precision: \([1, 0.23]\) with BIAS := 127
- Double precision: \([1,1.52]\) with BIAS := 1023
  - S: sign bit
  - M: mantissa
E: exponent

\[ f = (-1)^s \times M \times 2^E \]

The C standard guarantees
- `float` to be single precision
- `double` to be double precision
- `long double` can be double, extended or even quadruple precision
- casting `float` to `double` -> int
  - truncates the fractional part
  - is like rounding towards zero
- casting `int` to `float`
  - rounds (according to rounding mode)
- casting `int` to `double`
  - exact conversion

Normalized values
Condition is \( \exp \neq 00...0 \) and \( \exp \neq 11...1 \); if that's the case:
- exponent is biased: \( E = \exp - \text{BIAS} \)
- leading 1 implied: for \( M = 1.xx..x \rightarrow \text{frac} = xx...x \)
  - minimal \( (M = 1.0) \) for \( 00...0 \)
  - maximal \( (M = 2.0-\text{EPS}) \) for \( 11...1 \)

Denormalized values
Condition is \( \exp = 00...0 \); if that's the case the exponent is different:
- \( E = 1 - \text{BIAS} \)
- cases for denormalized values
  - \( \text{frac} = 00..0 \): value represents signed 0
  - else: equispaced numbers close in \([-0.5, 0.5]\]

Special values
Condition is \( \exp = 11...1 \); if that's the case the following are defined:
- \( \text{frac} = 00...0 \): Inf (+/-) | \( \text{frac} \) not zero: NaN (sqrt(-1), Inf-Inf etc.)

Special properties of encoding
- floating point 0: all bits zero (sign still matters)
- comparing works (nearly) the same as with unsigned int S
  - NaNs are tricky
  - must consider the sign

Rounding
The idea of floating point arithmetic is to compute the exact result and then round, such that it fits into frac. There are different rounding modes:
- towards zero
- round down (towards -Inf)
- round up (towards Inf)
- nearest even (default)

Mathematical properties
Addition:
- closedness: but it might generate NaN
- commutative
- not associative
- 0 is additive identity
- additive inverses exist (except for $\text{Inf} & \text{NaN}$)
- monotonicity ($a>b \rightarrow a+c>b+c$) (except for $\text{Inf} & \text{NaN}$)

Multiplication:
- closedness: but it might generate $\text{NaN}$
- commutative
- not associative
- 1 is multiplicative identity
- monotonicity (except for $\text{Inf} & \text{NaN}$)

Multiplication does not distribute over addition!

**Linking**

When issuing `gcc -o output main.c foo.c` the two source files get compiled separately to relocatable object files (.o) with `cpp` (GNU C pre-processor), `cc1` (GNU C compiler), finally `as` (GNU assembler) and then they will get linked with `ld` (GNU linker).

**Executable and linkable format (ELF)**

An ELF file consists of different sections, a lot of them are not needed. The most important ones are:

- **ELF header**: defines word size, byte ordering, file type, architecture etc.
- **Program header table**: describes the different segments (executables require this section)
- **.text section**: contains the code
- **.rodata section**: read-only data (e.g., jump table or constants)
- **.data section**: initialized global variables
- **.bss section**: contains no data (most of the time just the size) just "space" for global variables
- **.symtab section**: symbol table, procedure and static variable names, section names & locations
- `.rel.text` section: relocation info for `.text` section, addresses of instructions to modify
- `section header table`: offsets and sizes for each section

There are four kinds of ELF files:

**Relocatable object file (`.o`)**
- Each `.o` file is compiled by exactly one source file
- Contains compiled code and data that can be combined (linked) with other relocatable object files to an executable object file

**Executable object file**
- Contains code and data that is ready to be executed

**Shared object file (`.so`)**
- Special relocatable object file that gets linked dynamically at load or even run-time
- Typically these files get used by multiple programs (e.g., a library)

**Core dump**
- Contains a recorded state of the memory of a computer program
- Often used to keep track of crashes

**Static linking**

The linking follows these two steps:

1. Symbol resolution:
   - A program defines and references symbols
   - These are stored in a symbol table
   - The linker associates each symbol with exactly one definition

2. Relocation:
   - The linker merges the separate code and data sections into a single one
   - It then relocates the symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable

**Linker symbols**

**Global symbols:**
- Symbols defined by some module that can be referenced by other modules
- E.g., non-static global variables or non-static functions

**External symbols:**
-Globals referenced by a module but defined in another one

**Local symbols:**
- Symbols defined and referenced only by one module
- Not local program variables! (the linker doesn’t care about them...)
- E.g., static functions

Those program symbols can be either strong (initialized globals or procedures) or weak (uninitialized globals). With these definitions the linker follows these rules:

- Multiple strong symbols aren’t allowed (linker error!)
- Always choose the strong symbol if there is one, otherwise pick any

In general it’s a good idea to use `extern` and `static` as much as possible to reduce any ambiguity in programs.

**Libraries**
Static libraries

The `ar` tool allows to archive multiple relocatable object files into a single file, this is useful because it allows to update the archive and you don't need to link huge object files.

- to create an archive: `ar rs libfoo.a foo.o bar.o baz.o ...`
- to list object files in archive: `ar -t libfoo.a`

Dynamic libraries

Because every static library includes at least `libc` changes in the system might affect them. The solution is to use dynamic libraries that get loaded during runtime (`dlopen`). Another advantage is that those libraries can be loaded once into memory and be used by multiple programs.

Example (no error handling)

```c
#include <dlfcn.h>

// ...
void *lhandle;
void (*addvec)(int *, int *, int *, int);

lhandle = dlopen("./libvector.so", RTLD_LAZY); // lazily load library
addvec = dlsym(handle, "addvec"); // get function pointer to addvec
addvec(x, y, z, 2); // z = x + y
```
3: Performance optimization

Optimizing code

gcc flags

- the `-Ox` flag tells it to use level `x` optimizations (good values 2 or 3)
- the `-march=xxx` flag tells it more about the architecture (eg. `x86-64`)
  - on Linux use `uname -m` to find out
- the `-mXY` flag tells it enable `XY` memory extension (eg. `-m64`)
- the `-ftree-vectorize` flag tells it to try and vectorize operations

See `man gcc` for a whole lot of more flags.

Compiler optimizations

Code motion

Is when code is rearranged, for example to reduce frequency with which computation is performed (precomputation).

Example: Unnecessary repetitions of computations.

Strength reduction

Reduces a costly operation with a simpler/cheaper one.

Examples: Shifts instead of multiplications or replacing sequence of product with additions.

Common subexpression elimination

Reuse computations that are already performed, this can be difficult for the compiler to detect.

Example: Having `(i-1)*n`, `i*n` and `(i+1)*n`, it's cheaper to only calculate `i*n` and do two adds.

Optimization blockers

Procedure calls

```c
for (int i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
```

The above code has (unexpected) complexity of $O(n^2)$ because `strlen` gets called in each iteration. A solution would be to move this call out of the loop body (code motion/precomputation).

The compiler isn't able to do that by itself because the procedure might have side effects that it isn't able to predict. The compiler will treat procedure calls as a blackbox that it won't analyze.

Solutions:

- do code motion yourself
- inline function calls whenever possible

Memory aliasing

In C often a memory location can be referenced by different variables, because the compiler makes conservative assumptions it might not be able to predict constant values or do similar optimizations.

Solutions:

- make use of local variables to accumulate values

Blocking and unrolling
**Unrolling**

Unrolling is an optimization performed on loops that will turn a loop into a sequence of operations. Full loop unrolling can only be done for constant iteration sizes and comes at the cost of a larger executable.

**Blocking**

Blocking - also called tiling - is when you exchange loops and partially unroll a loop. It helps with locality (see cache) and also allows the CPU to execute multiple instructions in the pipeline (ILP).
A good example is a matrix multiplication where you traverse the matrix in blocks instead of lines, now there will arise common subexpressions that you can replace (see Strassen algorithm).

**Instruction-level parallelism**

Modern hardware can and will perform multiple instructions in parallel. There are some limitations that might arise:

- data dependencies
- compiler not able to detect reordering potential
- lack of associativity & distributivity in floating point arithmetic
- branch misprediction

A superscalar processor can issue and execute multiple instructions in one cycle. These are mostly dynamically scheduled by analyzing the sequential instruction stream (this is done in hardware with an instruction cache).

**Reassociation**

Turning \( x = (x \, OP \, d[i]) \, OP \, x[i+1] \) in a loop into \( x = x \, OP \, (d[i] \, OP \, x[i+1]) \) resolves a data dependency (for non-associative \( OP \) s or if there are more than one unit for the \( OP \) ) and the CPU can already start with an \( OP \) when it's still computing the last one.

Using 2 accumulator variables one can also turn this statement into:

```c
x0 = x0 \, OP \, d[i];
x1 = x1 \, OP \, d[i+1];
}
*res = x0 \, OP \, x1;
```

**Example (unrolling and reassociation):**

The following procedure might be worth it for large iteration sizes:

- unroll to any degree \( L = K \times r \) (for some \( r \))
- can accumulate \( K \) results "in parallel"

This way performance is only limited by the throughput of available functional units for \( OP \).

**Performance**

\[
\text{Execution time} = \text{CPE} \times n + \text{Overhead} \quad \text{where} \quad n \text{ is the number of elements and CPE is cycles per element.}
\]

**Vector instructions**

**Streaming SIMD extension 3: SSE3**

The SSE3 instruction family makes use of 16 registers (\( %\text{xmmN} \, 0-15 \)) each 128 bits wide. This allows packing multiple data into one register (eg. 4-way 4bytes) and operate on these with a single instruction.

**Arithmetic instructions**

\( \text{addXY \ Src, Dest} \)

Where \( x \) is either

- \( p \) : packed (vector instruction) or
- \( s \) : single slot (scalar)
and $Y$ is either
- $s$: single precision
- $d$: double precision.

The $Src$ and $Dest$ can be (only $xmm$ register)
- register, register
- memory, register

Beside $add$ there are other instructions, such as:

- $sub$, $mul$, $div$, $max$, $min$ or $sqrt$

Other instructions
- to set a register 0.0, use: $xorps \%xmmN, \%xmmN$
- $movss (\%rax), \%xmmN$ ($\%rax$ is (float *))
  move scalar single (moves 32 bits)
- $movaps \%xmmK, \%xmmN$ (could also move 128 bits from memory)
  move aligned packed-single (moves 128 bits)
- $cvtXX2YY$ converts $X$ to $Y$, where $XX,YY$ can be:
  - $ss, sd$: single -> double (and vice versa)
  - $si, sd$: int -> double (also for float)
  - $si, ssq$: long long -> float (also for double)
  - $ts, si$: float -> int by truncating (also for double)
  - $tsd, siq$: double -> long long (also for single)

Advanced vector instructions 2: AVX2

The AVX2 extension uses specialized registers as well; there are 16 registers ($\%ymmN$ 0-15) which are 256 bits wide and allow SIMD operations.

**AVX2 SIMD**

Also a vectorization extension for the x86_64 architecture but operands are twice as wide (256 bits):

- $vaddsd \%ymm0, \%ymm1, \%ymm1$ (8-way single precision vectorized add)
- $vaddpd \%ymm0, \%ymm1, \%ymm1$ (4-way double precision vectorized add)

**Note**: $mov$ and $cvt$ works with register, memory as well!
4: Computer architecture

Caches

Terminology

- placement policy determines where a block will be stored
- replacement policy determines which block gets replaced when a new block is cached
- miss rate: misses/accesses = 1 - hitrate
- hit time is the time to deliver a block from cache to the processor
- miss penalty is the additional time required because of a miss
- cold/compulsory miss occurs on the first access to a block
- conflict miss occurs when cache is large enough but blocks map to same location
- capacity miss occurs because there is not enough memory
- temporal locality means that recently referenced values are likely to be referenced again
- spatial locality means that values from nearby addresses are likely to be referenced as well
- a cache gets characterized by a triple \( (S,E,B) \); it has \( S \cdot E \cdot B \) bytes of memory
  - \( S = 2^s \) sets
  - \( E = 2^e \) blocks per set
  - \( B = 2^b \) bytes per block

Cache types

Direct mapped cache ( \( E=1 \) )

Old block gets replaced if there is a cache miss.

N-way set-associative cache

On a miss some block gets replaced, possibilities:

- random block
- least recently used (LRU)

Example of a 2-way associative cache:
Writing to cache

There are two different policies on a cache hit when writing data:

Write-through

Write directly to memory on a hit.
- this way memory is always consistent with the cache
- huge disadvantage: it’s slow

Write-back

Defer memory update until block is replaced.
- need a dirty bit in the cache (to remember a potential write)
- faster with the cost of greater complexity

There are also two different policies for a write-hit:

Write-allocate

Load block into cache and update it in cache, commonly used with write-back caches.
- performance gain if more writes follow
- might replace another block

No-write-allocate

Write directly to memory, commonly used with write-through caches.
- slower than write-allocate policy
- simpler to implement

Exceptions

An exception is a transfer of control to the OS in response to some event, when an exception occurs there is typically an exception handler that takes care of it.
Each of these events has a unique exception number (k = index of exception table) - this is called an interrupt vector. For the exception vector k the handler k is responsible of handling it, in the exception table there is a pointer to that handler.

An exception causes a switch to kernel mode (aka ring 0 or privileged mode). The kernel is:
- a set of trap handling functions (always)
- code to create the illusion of user-space processes (always)
- a set of threads in special address space (sometimes)

Synchronous exceptions

This type of exception occurs in response to an instruction, there are
- **traps**: intentional set up handlers that return control to the next instruction
  - eg. system calls or breakpoints
- **faults**: unintentional failures that might be resolvable; either re-executes current instruction or aborts
  - eg. floating point exceptions, page faults or invalid memory reference (SIGSEV)
- **aborts**: unintentional and unrecoverable failures; the program gets aborted
  - eg. parity error or machine check

**Asynchronous exceptions (interrupts)**

This type of exception is caused by events external to the processor, some examples:

- **I/O interrupts**: pressing `C` on keyboard, receive data from network or hard disk
- **hard reset interrupt**: pressing reset button
- **soft reset interrupt**: pressing Ctrl-Alt-Delete on windows

In the event of an interrupt:

- **interrupt-request line (IRQ)** is triggered by I/O device
- **interrupt handler** receives it
- **nothing may happen** (it was masked to ignore or delay)

The interrupt mechanism (interrupt vector -> handler) is also used for exceptions.

**x86 interrupts**

A CPU of this type has different interrupt pins:

**Interrupt requests**

An IRQ is issued via the `INTR` pin, they can be disabled by setting status flags (CLI/STI). If they aren’t disabled, the CPU completes the current instruction and acknowledges the IRQ via the `INTA` pin, then the interrupt vector is supplied on the data bus and the CPU is able to issue the exception with the correct number.

A programmable interrupt controller (PIC) acts as a multiplexer for the `INTR` and `INTA` pins such that each device gets its own pin and the CPU can tell where an exception comes from. It also buffers and prioritizes simultaneous IRQs and is able to selectively mask out some interrupts.

**Non-maskable interrupts**

These interrupts are issued via the `NMI` pin which can’t be disabled by the processor. Once `NMI` is asserted, the CPU completes the current instruction and issues exception number 2. Because there are multiple possible sources of issuing such an exception, the CPU must poll these sources to find out where it came from.

Examples:

- hardware faults (eg. parity errors)
- watchdog timer (responsible for resetting CPU in case of a time-out).

### Virtual memory

**Different address spaces:**

- **virtual address space**: set of \( N=2^n \) virtual addresses: \{0,1,...,N-1\}
- **physical address space**: set of \( M=2^m \) physical addresses: \{0,1,...,M-1\}

Every byte in main memory has one physical address and one (or more) virtual addresses. The **memory mapping unit** (MMU) is responsible of mapping a virtual address to its corresponding physical one.

Often a page table entry contains permission bits (typically \( u/s \) (user/supervisor mode), \( \text{READ} \) and \( \text{WRITE} \)) to ensure memory protection.

![Virtual Address Diagram](image)

The main aspects of using virtual memory are:

- efficient use of RAM (keep only active areas in RAM, memory reuse etc.)
- huge simplification for programmer and compiler/linker
- address space isolation

### Caching virtual memory

In the context of virtual memory one can think of RAM as a caching mechanism for physical memory from disk. A program accesses a set of virtual pages - this is called its working set, it performs well if the working set size is smaller than the main memory size. If this is not the case the OS needs to constantly replace virtual pages in RAM and access the hard disk and the performance will drastically decrease - this is called thrashing.

### Address translation

In the case of a page hit a memory request follows these steps:

1. CPU sends virtual address to MMU
2. MMU fetches the page table entry
3. MMU sends physical address to cache/memory
4. cache/memory sends back the data

In the case of a page fault the following will happen: 1. CPU sends virtual address to MMU 2. MMU fetches the page table entry 3. valid bit is zero -> page fault gets triggered 4. handler identifies victim (if dirty -> write-back) 5. handler gets page from disk and updates PTE 6. handler returns to process, retrying the instruction
Translation lookaside buffer (TLB)

A TLB is a small hardware cache in the MMU specifically for address translation - it stores some PTEs. This eliminates one memory lookup (step 2 above) if the PTE is in this cache. Without this caching mechanism each memory lookup invokes two memory accesses which is expensive.

Multi-level page table

Example for linear PT size:

- 48-bit addresses with 16-bit offset: $2^{32}$ entries
- PTE is 8 bytes in size
- this requires a total of $2^{35}$ bytes or about 34 GB

Linear page tables are not practical, the solution is to use a tree-structured page table. In a $k$-level page table, we only need to keep at least $k$ smaller page tables in memory for a memory lookup.

To lookup a page the virtual address (without offset) gets split into $k$ parts, each of these parts tells us where to look for the next level page table. Finally the PPN is stored in the $k$th page table.

Multiprocessing

Terminology

- symmetric multiprocessing (SMP): all cores will have their own cache but access the same physical memory
communication between processors via shared variables
- typically hardware guarantees cache coherency
- memory becomes a bottleneck
- **power wall**: as frequency gets increased more power gets dissipated
  - the limit of cooling a CPU is reached -> frequency won't increase in near future
- **ILP wall**: serial acceleration using ILP is bounded

These three conditions are the reason modern CPUs have a different approach to using Moore's Law as an advantage.

**Coherency**

Means that values in caches match each other.

- advantage: shared-memory model works -> easy to program
- disadvantages: complex to build and memory is slower

**Consistency**

Means that the order of execution (reading/writing to memory) is consistent within a certain model.

- *program order*: is the order in which the CPU appears to issue instructions (not the real order)
- *visibility order*: is the order which all reads/writes are seen by the CPUs (CPUs read the last value in visibility order)

**Sequential consistency**

Operations from a CPU appear in program order for all CPUs and every CPUs visibility order is the same interleaving of all program orders.

This requires:

- atomic memory operations
- each program issues memory operations in program order
- RAM totally orders operations

**Advantages:**

- easy to understand and write code in this model
- easy to analyze automatically

**Disadvantages:**

- hard to build a fast implementation
- can't reorder reads/writes
- can't combine writes to same cache block (buffering)

Relaxing sequential consistency:

- write-to-read: later reads can bypass earlier writes
- write-to-write: later writes can bypass earlier writes
- break write atomicity
- weak ordering: no implicit ordering guaranteed

**Processor consistency**

This is the standard for x86_64 CPUs, it's a write-to-read relaxation - meaning:

- all processors see writes from another processor in the issued order
- processors can see different interleavings of writes

**Snoopy caches**

This is a technique of providing cache coherency by snooping on reads/writes from other processors. If block is valid in local cache and another CPU writes to it, the local block gets invalidated. This method requires a write-through cache, such that the cache can snoop on the bus.

**Write-back coherency protocols**

**MSI**

Each block has one of 3 states (modified, shared, invalid) and can only be dirty in one cache. The cache logic must respond
to reads/writes with a state change and possibly flush the data.

**MESI**

As with MSI there are three states but it introduces another one: exclusive. Blocks that are in this state are clean and in no other cache, this reduces main memory accesses. It adds new bus signals that are snooped by other caches to invalidate other cache blocks.

**MOESI**

Another protocol that adds an additional state: owner. Blocks with this flag own it - meaning that they are the only one allowed to modify it.

**MESIF**

Yet another protocol that adds another state: forward. Blocks with this flag requested this block the most recent and are designated responder - they will respond to a request. Of all the blocks in the caches always one has this state, this allows cache-to-cache speeds without the need of a broadcast to all other caches.

**Overview**

- **Modified**: no other copies, it's dirty
- **Owner**: multiple consistent copies, only one allowed to modify (gets broadcasted)
- **Exclusive**: no other copies, it's clean
- **Shared**: multiple consistent copies, not owned
- **Invalid**: not valid, must be fetched from memory
- **Forward**: like shared, but is designated responder

The above protocols preserve the following invariants:

<table>
<thead>
<tr>
<th>M</th>
<th>O</th>
<th>E</th>
<th>S</th>
<th>I</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>n/a</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Barriers/fences**

The weak consistency models could break functioning algorithms, to get around this problem one can use barriers:

- **compiler barriers**: prevent the compiler from reordering statements
- **memory barriers**: prevent the CPU from reordering statements

These are typically compiler intrinsics, for example with gcc: __asm__ or __volatile__

**Barriers on x86_64**

The x86_64 architecture provides these three barriers:

- **LFENCE**: load fence
- **MFENCE**: memory fence (load & store)
- **SFENCE**: store fence

They guarantee that all the following instructions see the changes to memory that preceed this instruction.

**Multicore synchronization**

There are two ways of synchronizing:

- **Atomic operations**:
also have ordering constraints
- eg. test-and-set or compare-and-swap Interprocessor interrupts (IPIs):
- invoke interrupt handler on other CPU (very slow)
- eg. OS specific tasks or kernel debugging

Test-and-set

```c
bool TAS(bool *lock);
```

- sets *lock = 1 and returns it old value
- the processor won't reorder instructions past a TAS
- during a TAS operation the memory bus must be locked (read-modify-write cycle)

# Example (spinlock):

```c
void acquire(int *lock) {
    do {
        while (*lock == 1);  // most of the time no TAS -> cheaper than RMW
    } while (TAS(lock) == 1);
}

void release(int *lock) { *lock = 0; }
```

Compare-and-swap

```c
void CAS(T *location, T old, T new);
```

- sets *location to new if old == *location
- can implement nearly all wait-free data structures
- requires a RMW cycle (as with TAS)
- doesn't report if value changed to the same value (see ABA problem)
  - solution: ensure that value "always" changes (eg. timestamping)

# General pattern: Wait-free data structure
- readers all read the same data
- writers copy the data structure, modify it and write it back
- old version is only deleted when all readers are finished

x86_64 atomic operations

- XCHG : atomic exchange values (register-register or memory-register)
- LOCK XYZ : locks the bus during XYZ instruction
  - eg. LOCK ADD
- CMPXCHG Src Dest : compare-and-swap
  - compares Dest with %EAX and sets Dest = Src if they're equal

Simultaneous multi-threading

While the CPU is waiting for memory most functional units are idle, during this time it can process other date - this is called SMT (or hyperthreading). To distinguish different instruction streams, in hardware the instructions are labeled with different thread IDs. From OS perspective a multi-threaded CPU look like multiple CPUs.
The advantage is that this is a cheap thing to do (in transistors) but the performance gain is limited to 10-20% and it depends on the workload (sometimes even slower than a single thread).

Non-uniform memory access (NUMA)

This is a type of SMP architecture where the CPUs are not equi-distant from the shared memory. This kind of architecture is useful because it removes the memory bottleneck (interconnect is now a network) and scales really well.
Because there's no bus anymore bus-snooping is not an option anymore, other approaches to provide cache coherency are:

Bus emulation
Similar to bus-snooping but implemented with message-passing.

Cache directory
Each node’s local memory is provided with a cache directory which contains information about blocks such as its owner and on which other nodes the block is present. This is particularly useful when blocks are not widely shared.

False sharing
This is an effect that happens when different data is in a single cached block: On a single CPU this performs well for reasons of locality but on two different CPUs the block ping-pongs between the caches on every write. A possible solution would be to pad the data.

MCS locks
The Mellor-Crummey and Scott lock is a type of lock which only spins on local data and upon a `release()` only one CPU gets signalled.
Problem: Cached block containing a lock is a hot-spot, it gets continuously invalidated.
Solution: Processors add themselves to a queue of waiting cores and spin on this entry.

Devices

Device registers
- device registers are memory mapped
  - access using `movX`
- I/O instructions use 16-bit address space to support older devices
  - special instructions: `inb`, `outb`
- communication via indirection (saves address space)
  - first write an offset to `index register` and then data to `data register`
- contents change without writing to it (incoming data etc.)
- writes will trigger actions

Caching could cause a lot of hassle with device drivers: You can set a "no cache" flag in the PTE and I/O address space is not cached by default.

Direct memory access (DMA)
To avoid using the CPU for copying large amount of device data there is a DMA controller that allows data being transferred directly between devices and memory. With this the CPU only receives one interrupt per transfer and doesn’t need to acknowledge each word.
This requires the DMA buffers to be non-cacheable or OS support for flushing portions of the cache because otherwise data in cache might become inconsistent.

Devices use DMA both for data and control information transfer.

Descriptor rings
Most modern devices use descriptors that carry meta-data, these descriptors get buffered in a circular P/C-queues where one party is the OS and the other one the device itself.

Device over- and underruns
- if the device has no space in buffer -> discards data
- if device has no data to send -> poll memory for next descriptor or sleep

CPU over- and underruns
- if CPU read all received data -> signal device to send an interrupt when new data was sent
- if CPU has no space to in buffer -> signal device to send interrupt when new space in buffer
Discoverable buses: PCI

The Peripheral Component Interconnect is a software-visible interface to I/O hardware which is organized in a tree-like structure: Each device asks for a set of address ranges and bridges on the interconnect will remap addresses to the correct device. This gives an OS the illusion that there is a contiguous address space. To gather all connected devices the OS sends a request to the PCI root complex(es), gets data back containing all detected devices, if there's another bridge it'll recurse. This results in a "depth first then breadth" search of the tree.

Bus mastering

This is DMA over PCI - meaning that PCI devices can read/write data to anywhere in memory (even other PCI ) devices.